

EmbSW1

C. Gwerder, S. Körner, M. Ehrler, L. Leuenberger, C. Ham, L. Däscher

14. Januar 2016

Inhaltsverzeichnis

1	Embedded RT Systems	4
1.1	Charakterisierung Embedded Systems	4
1.2	Definition Real-time Systems	4
1.3	Rate monotonic scheduling (RMS)	5
2	Modellierung	7
2.1	Modellierung eines Embedded Systems	7
2.2	Systemgrenze definieren und Systemprozesse finden	7
2.3	Verteilung festlegen	8
2.4	Systemprozesse Detaillieren	9
3	Hardware/Software Codesign	11
3.1	Ziele	11
3.2	Anforderungen für praktische Anwendung	11
3.3	Spezifikationssprachen	11
3.4	Virtuelle Prototypen	11
3.5	X-in-the-loop	11
4	Finite State Machine (FSM)	12
4.1	Zustandsbasierte Systeme	12
4.2	Statecharts von Harel	14
5	Fine State Machine Realisierung	16
5.1	Realisierung von Flachen FSMs	16
5.2	Realisierung mit Steuerkonstrukt (prozedural in C)	16
5.3	Realisierung mit Steuerkonstrukt (Objektorientiert in C++)	20
5.4	Realisierung mit Tabelle	23
5.5	Realisierung mit State Pattern	28
6	Modularisierung	35
6.1	Motivation	35
6.2	Grundprinzip	35
6.3	Information Hiding	35
6.4	Hauptkriterien der Zerlegung	35
6.5	Definitionen Kopplung	36
6.6	Definitionen Kohäsion	36
6.7	Schlechte vs. gute Modularisierung	37
7	Event Based Systems	38
7.1	Reaktive Systeme	38
7.2	Umsetzung von Ereignissen	38
7.3	Model View Controller (MVC) aka Observer Pattern	38
8	Patterns	40
8.1	Pattern-Philosophie	40
8.2	Einige wichtige Patterns	40
8.3	Creational Patterns	40
8.4	Structural Patterns	41
8.5	Behavioral Patterns	41
8.6	Concurrency Patterns	41

9 Computerarithmetik	42
9.1 Ganzzahlig	42
9.2 Arithmetische Schaltungen	43
9.3 Fixpunkt (Fixed Point)	46
9.4 Gleitkomma (Floating Point)	46
10 Plattformen	49
10.1 Auswahl der Zielpattform	49
10.2 Vergleich der Beispielarchitekturen	49
10.3 Digital Signal Processor (DSP)	49
10.4 ARM-Cortex	51
10.5 MSP432 und Vergleich	51
11 Multicore	52
11.1 Steigern der Geschwindigkeit eines Prozessors	52
11.2 Multicore Prozessor	53
11.3 Caches	54
11.4 Ping Pong Speicher	55
12 Concurrency	56
12.1 Motivation	56
12.2 POSIX Threads Programming	56
12.3 Synchronisation: Zugriff auf gemeinsame Ressourcen	59
12.4 Semaphoren	61
12.5 Thread-Synchronisation in C mit POSIX Threads	61
12.6 Petri-Netze	62
13 Bootstrap	64
13.1 Programmdownload	64
13.2 Startvorgang (Bootting)	64
13.3 Mehrstufige Bootloader (multistage bootloader)	64
13.4 Sicherung des Programmdownloads	64
13.5 Typische Memory Map	65
13.6 Typischer Bootvorgang	65
14 Style guide und Language Linkage	66
14.1 Style guide	66
14.2 Language Linkage	66
15 Eigene Notizen	68

1 Embedded RT Systems

1.1 Charakterisierung Embedded Systems

Ein Embedded System ...

- ... ist ein System, das einen Computer beinhaltet, aber keiner ist
- ... besteht üblicherweise aus HW und SW
- ... ist häufig ein Control System

1.1.1 Charakterisierung von Embedded Systems

- **Reaktive Systeme** : Interagieren mit ihrer Umgebung
- **Echtzeitsysteme/Real-time systems** : Definierbare zeitliche Anforderungen erfüllen
- **Verlässliche Systeme/Dependable systems**: Sehr hohe Zuverlässigkeitsanforderungen erfüllen
- **Weitere Anforderungen/Charakteristiken**:
 - Kleiner Energieverbrauch
 - Kleine physikalische Abmessungen
 - Lärm, Vibration, Feuchtigkeit etc.

1.1.2 Verfügbarkeit

Anteil der Betriebsdauer, in der das System seine Funktion erfüllt:

$$\text{Verfügbarkeit} = \frac{\text{Gesamtzeit} - \text{Ausfallzeit}}{\text{Gesamtzeit}} \quad (1)$$

1.2 Definition Real-time Systems

Definition : System, das Informationen innerhalb einer definierten Zeit (deadline) bearbeiten muss. Es erfüllt explizite Anforderungen an die Antwortzeiten.

Antwortzeit : Zeit vom Stimulus (Vorhandensein Eingangswert) bis zum Erscheinen des Ausgangswert

Fehlerhaftes System: es werden nicht alle formal definierten Systemspezifikationen erfüllt.

Soft real-time system: System wird durch das Verletzen von Antwortzeiten nicht ernsthaft beeinflusst → Komforteinbußen (Bsp.: Geldautomat)

Firm real-time system: Durch Verletzen weniger Antwortzeiten wird das System nicht ernsthaft beeinflusst. Bei vielen Verletzungen jedoch → kompletter Ausfall, katastrophales Fehlverhalten (Bsp.: GPS-gesteuerter Rasenmäher)

Hard real-time system: Durch Verletzen der Antwortzeiten wird das System ernsthaft beeinflusst → kompletter Ausfall, katastrophales Fehlverhalten (Bsp.: Helikoptersteuerung)

Determinismus: Für jeden möglichen Zustand und alle möglichen Eingabewerte ist jederzeit der nächste Zustand und die Ausgabewerte definiert.

Auslastung/Utilization: Auslastung pro Prozess:

Für n periodische Tasks erhalten wir die gesamte Auslastung U:

u_i = Ausführungsfaktor des Task i

$u_i = \frac{e_i}{p_i}$ e_i = Executionzeit des Task i

p_i = Executionperiode des Task i

$$U = \sum_{i=1}^n u_i = \sum_{i=1}^n \frac{e_i}{p_i} \quad (2)$$

1.3 Rate monotonic scheduling (RMS)

1.3.1 Cooperative Scheduling

- Aktiver Task entscheidet selbst, wann er den Prozessor für anderen Task frei gibt.
- Ein unfairer Task blockiert dabei die anderen
- Nächster Task aussuchen via: FCFS, Round Robin, zufällig, Prioritäten, etc.
- Einfach zu implementieren

1.3.2 Preemptive Scheduling

- Task mit höherer Priorität wird ausgeführt
- Task mit niederer Priorität wird verdrängt (preempt)
- Prioritätsverteilungsmöglichkeiten
 - Dynamic-priority algorithm: Prioritäten werden zur Laufzeit aufgrund vorhandener Deadlines angepasst.
 - Static-priority algorithm: Prioritäten werden zur Entwicklungszeit festgelegt und nicht mehr geändert.
→ Einfacher als dynamic-priority
- Viele RTOS verwenden preemptive static-priority Algorithmen

1.3.3 Schedulability

Eine Menge von Tasks ist dann schedulable, wenn alle Tasks zu allen Zeiten ihre Deadlines einhalten können.

1.3.4 Rate-monotonic scheduling

Theorem: Hat man ein Set periodischer Task und preemptive static-priority scheduling, erhält man optimales Scheduling, wenn man die Prioritäten so verteilt, dass die Tasks mit kürzester Periodizität die höhere Priorität erhalten (rate monotonic).

Oder: Ein Set von n periodischen Tasks ist dann rate-monotonic schedulable, wenn die gesamte Systemauslastung U folgenden Wert nicht überschreitet:

$$U \leq n \cdot (2^{\frac{1}{n}} - 1) \quad (3)$$

n	2	3	4	5	10	∞
u [%]	82.4	78.0	75.7	74.4	71.7	69.3

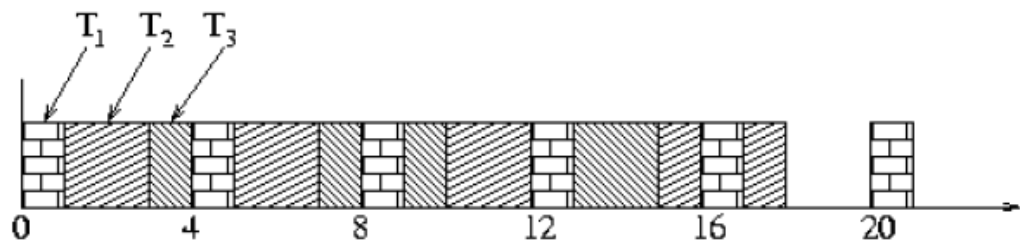
Die maximale Systemauslastung beträgt:

$$\lim_{n \rightarrow \infty} n \cdot (2^{\frac{1}{n}} - 1) = \ln 2 \approx 0.69 \quad (4)$$

Folgerung aus diesem Grenzwert: Das theoretische Limit der Systemauslastung liegt bei 69%.

1.3.5 Beispiel RMS

τ_i	e_i	p_i	$u_i = \frac{e_i}{p_i}$
τ_1	1	4	0.25
τ_2	2	5	0.4
τ_3	5	20	0.25



1.3.6 Interrupts

Da die ISR von der Hardware getrieben wird, ist auch der niederwertigst priorisierte Interrupt immer noch höher priorisiert als der höchst priorisierte Task. Für die RM-Analyse stehen 2 Lösungsvarianten zur Verfügung:

1. Der ISR-Code wird nicht mittels Interrupt ausgelöst, sondern in einem Polling-Loop.
2. Für die RM-Analyse wird mit der Periode der höchsten Priorität gerechnet.

2 Modellierung

2.1 Modellierung eines Embedded Systems

Model Driven Approach (MDA)

- Modelle werden als Werkzeuge der Dokumentation angesehen. Dadurch wird unter Umständen dasselbe zweimal beschrieben (als Code und als Diagramme)
- Ziel ist es aus formalen Modellen lauffähige Software zu erzeugen. Bei State Machine ist dies durch Statechart der Automaten vollständig möglich.

Agile Softwareentwicklung

Agile Softwareentwicklung ist der Oberbegriff für den Einsatz von Agilität (flink, beweglich) in der Softwareentwicklung.

- Eher offen für Änderungen als starres Festhalten an Plänen
- Eher Menschen und Kommunikation als Prozesse und Tools
- Eher „darüber miteinander reden“ als „gegeneinander schreiben“
- Eher Vertrauen als Kontrolle
- Eher „Best Practices“ aus Erfahrung als verordnete Vorgaben
- Eher Angemessenheit als Extremismus
- **Aber keine Anarchie!**

Model Driven Development (MDD)

Bei der modellbasierten Entwicklung kommen in allen Entwicklungsphasen durchgängig Modelle zur Anwendung.

UML:

- Aktivitätsdiagramm
- Sequenzdiagramm
- Zustandsdiagramm (Statecharts)
- Klassendiagramm
- Use Case Diagramm
- Verteilungsdiagramme (Deployment Diagram)

Generelles Vorgehen bei der Modellierung:

1. Systemgrenze definieren (statischer Aspekt)
2. Systemprozesse finden (statischer Aspekt)
3. Verteilung festlegen (statischer Aspekt)
4. Systemprozesse detaillieren (dynamischer Aspekt)

2.2 Systemgrenze definieren und Systemprozesse finden

2.2.1 Systemgrenze definieren

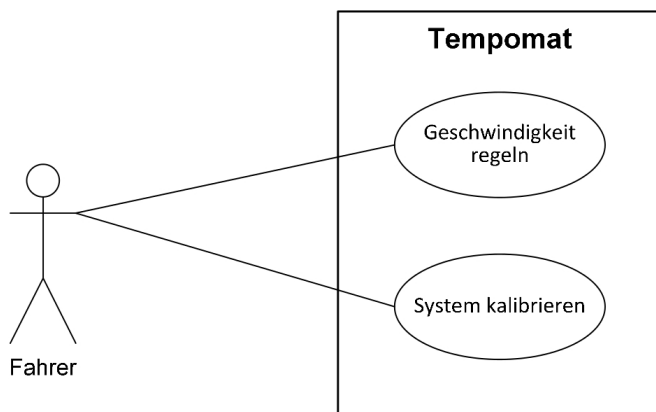
Das wichtigste und allererste bei sämtlichen Systemen ist die Festlegung der Systemgrenze (system boundary) mittels Kontextdiagramm (Use Case oder auch Sequenzdiagramm).

- Was macht das System, d.h. was liegt innerhalb der Systemgrenze
- Mit welchen Teilen ausserhalb des Systems kommuniziert das System
- Welches sind die Schnittstellen zu den Nachbarsystemen

2.2.2 Systemprozesse finden

Die Aufteilung in Hardware und Software sollte erst nach der Analyse der grundlegenden Anforderungen erfolgen. RTE-Systeme bestehen meistens nur aus einem einzigen Systemprozess, speziell wenn das System im Prinzip 'nur' ein Regler ist. Dieser Schritt ist in der Analysephase. Wenn hier von Prozessen die Rede ist, bedeutet das nicht unbedingt ein Betriebssystem Prozess ist. Wichtig ist, dass es nicht zuviele Funktionsblöcke gibt (nicht ins Detail gehen).

Bei Systemen bei denen die Grenze durch Nachrichtenflüsse charakterisiert werden können, eignen sich Sequenzdiagramme für Kontextbeschreibung.

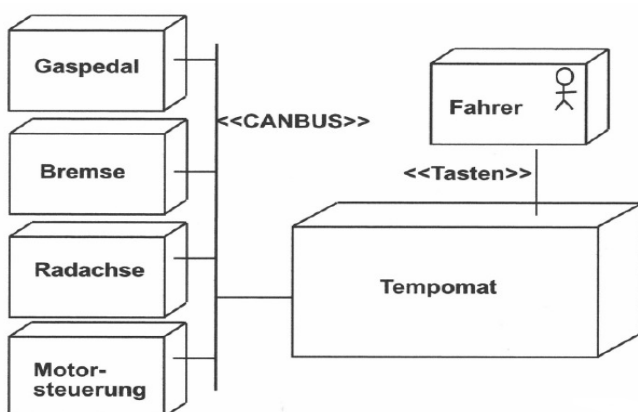


2.3 Verteilung festlegen

Bei Embedded Systems werden häufig mehrere Rechnersysteme verwendet um die Aufgaben zu erledigen. Diese liegen meistens geographisch (cm bis km) verteilt und sind mit einem Kommunikationskanal verbunden. Diese Systeme heissen Verteilte Systeme (Distributed Systems). Zur Darstellung eignet sich in UML das Verteilungsdiagramm (Deployment Diagram).

2.3.1 Verteilungsdiagramm (Deployment Diagram)

- Die Knoten werden zur Darstellung der geographischen Orten oder Subsysteme verwendet
- Die physikalische Verbindung zwischen den Knoten (Netzwerke, Kabel, Wireless, etc.) werden als Linien dargestellt.
- Die Knoten können auch hierarchisch aufgebaut sein.



2.4 Systemprozesse Detaillieren

Die gefundenen Systemprozesse müssen nun genauer spezifiziert werden. Dabei nicht detaillierter spezifizieren als sinnvoll. Jede weitere Detaillierung soll einen „added value“ liefern. Leitidee: Lieber weniger als mehr Details.

2.4.1 Verschiedene Detaillierungsstufen

- **Überblick:** z.B in Form von normalem umgangssprachlichem Text. Diese Form sollte immer erstellt werden.
- **Normale Sicht:** Sie ist für den Systementwickler gedacht und enthält mehr Details.

Für die Detaillierung eignen sich Sequenzdiagramme, Aktivitätsdiagramme, Statecharts und umgangssprachlicher Text.

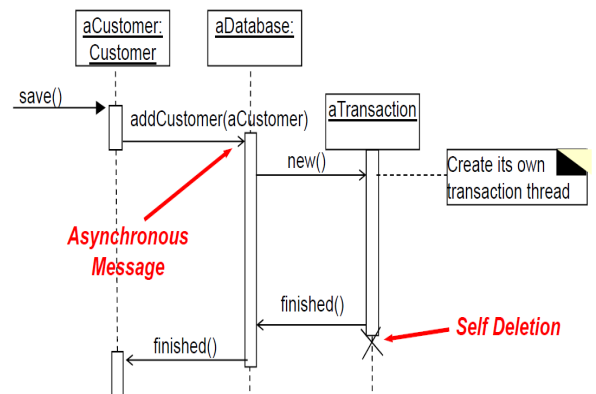
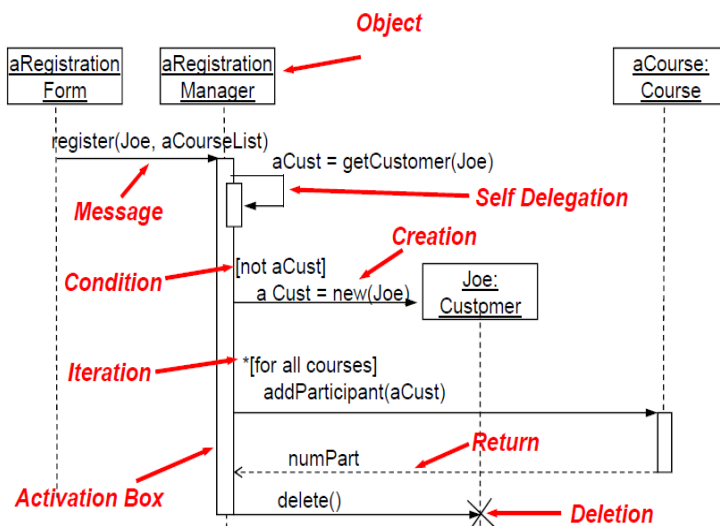
2.4.2 Sequenzdiagramme

Zweck:

- Austausch von Meldungen zwischen Objekten innerhalb einer beschränkten Zeitdauer anzeigen

Ideal:

- Für kurze Zeitdauer mit einigen wenigen Objekten
- Bei wenigen Verschachtlungen und Verzweigungen



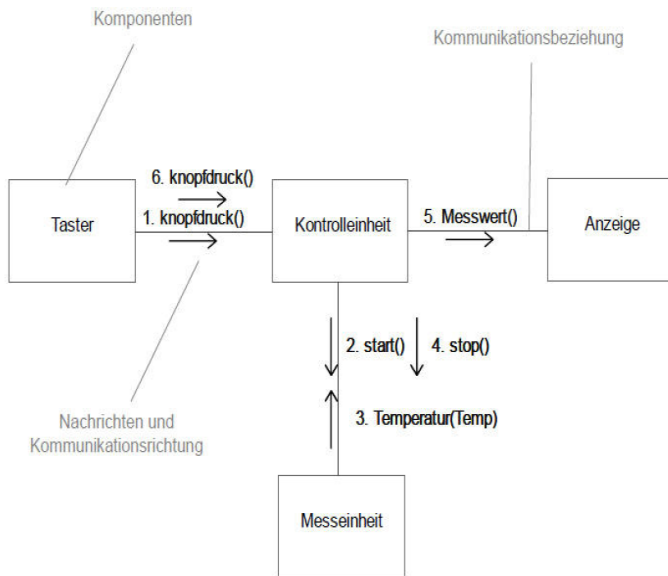
Pfeiltyp	Semantik
	Synchrone Aufrufe
	Asynchrone Nachrichten
	Datenfluss

auch bei return aus Funktionen

Diagram nicht überladen!

2.4.3 Kommunikationsdiagramm

Das Kommunikationsdiagramm zeigt dieselbe Information wie das Sequenzdiagramm. Der Schwerpunkt liegt aber nicht auf dem zeitlichen Ablauf, sondern auf dem Informationsfluss zwischen den Objekten.



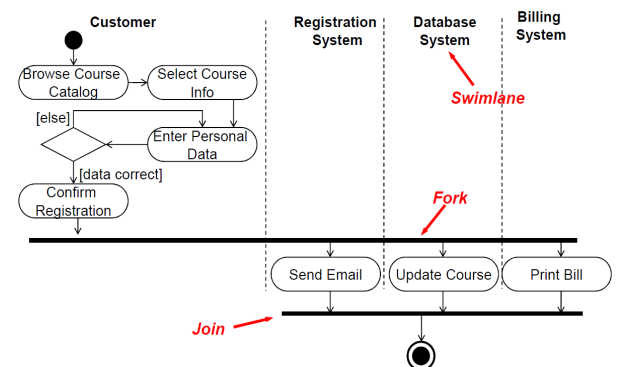
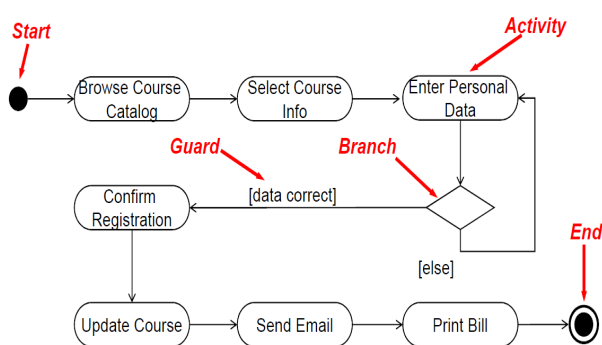
2.4.4 Aktivitätsdiagramme

Wird gebraucht für:

- sequenzielle Abläufe
- Prozess- und Steuerfluss
- Auch für gleichzeitige Prozesse geeignet
- Weniger geeignet für komplexe logische Bedingungen (Wahrheitstabelle)

Ideal:

- Für kurze Zeitdauer mit einigen wenigen Objekten
- Bei wenigen Verschachtlungen und Verzweigungen



3 Hardware/Software Codesign

3.1 Ziele

- Entwurf (Design) soll so lange wie sinnvoll (nicht so lange wie möglich) lösungsneutral erfolgen
- Systemdesign fördern, statt separate Designs für Mechanik, Elektronik, Firmware, Software, etc., die sich unter Umständen auch widersprechen können
- Die Systemspezifikation erfolgt idealerweise mit Hilfe einer eindeutigen Spezifikationssprache, nicht in Prosa
- Die Spezifikation kann simuliert (ausgeführt) werden
- Implementationen können einfach geändert werden, z.B. von HW zu SW oder umgekehrt.

3.2 Anforderungen für praktische Anwendung

- Methoden und Tools sollten beim Systemdesign nicht zu fachlastig sein, d.h. die Methoden sollten für Elektronik-, Firmware- und wenn möglich auch Mechanikentwickler anwendbar sein
- Falls möglich gute Toolunterstützung und automatische Synthese

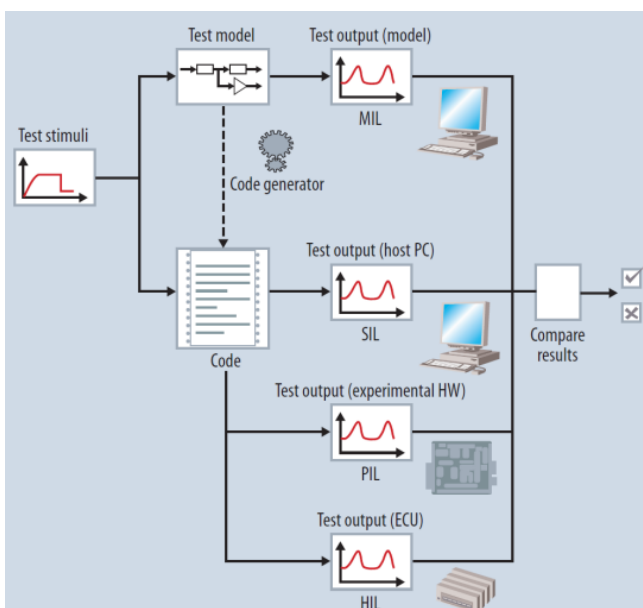
3.3 Spezifikationssprachen

- Formale Sprachen sind eindeutig
- Beispiele für Spezifikationssprachen: SystemC, SysML, SpecC, SystemVerilog, Esterel, Matlab/Simulink, Statecharts

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 Prototypen laufend durch physische Teile ersetzt werden

3.5 X-in-the-loop



- Bei einem vollständig als Modell vorliegenden virtuellen Prototyp spricht man auch von Model-in-the-Loop (MIL)
- Je mehr der Prototyp durch konkretere Implementationen ersetzt wird, spricht man von
 - Software-in-the loop (SIL)
 - Processor-in-the loop (PIL)
 - Hardware-in-the loop (HIL)

4 Finite State Machine (FSM)

4.1 Zustandsbasierte Systeme

4.1.1 Asynchron vs.synchrone FSMs

- Bei (elektronisch implementierten) asynchronen FSMs führen geänderte Inputsignale direkt zu Zustandsänderungen. Sie sind deshalb „schneller“, jedoch äusserst empfindlich auf Glitches und kaum testbar.
- Bei synchronen FSMs werden die Inputsignale nur zu diskreten Zeitpunkten betrachtet, diese Systeme sind getaktet.
- Softwareimplementationen sind eigentlich immer synchron, da die Rechner getaktet sind. Wenn die Inputsignale mittels Interrupts behandelt werden, ist darauf zu achten, dass die Interrupts nicht fälschlicherweise gesetzt werden (ist Aufgabe der Elektronik)
- Rein softwareseitig besteht die Problematik der Asynchronizität nicht

4.1.2 Finite State Machine (FSM)

- 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
- In der Praxis sind zwei FSM-Varianten bekannt:
 - der Mealy-Automat
 - der Moore-Automat

4.1.3 Zustands-Ereignis-Diagramm (State-Event-Diagramm)

- Die Zustände werden mit einem Kreis gekennzeichnet
- Die Ereignisse werden mit Pfeilen zwischen den Zuständen bezeichnet (Transitionen)
- Die Aktionen werden entweder bei den Zuständen geschrieben oder bei den Transitionen (je nach Automatentyp)
- Die Ausführung einer Transition benötigt keine Zeit. Deshalb sind bei der Modellierung oft Zwischenzustände vorzusehen, z.B. „Closing“, „Starting up“, „Booting“, etc.

4.1.4 Mealy-Automat

- Der nächste Zustand Z_{n+1} ist abhängig vom Input X und von Z_n : $Z_{n+1} = f(Z_n, X)$
- Der Output Y ist abhängig vom inneren Zustand Z_n **und vom Input X**: $Y = g(Z_n, X)$
- Die Actions liegen deshalb bei den Transitionen

4.1.5 Moore-Automat

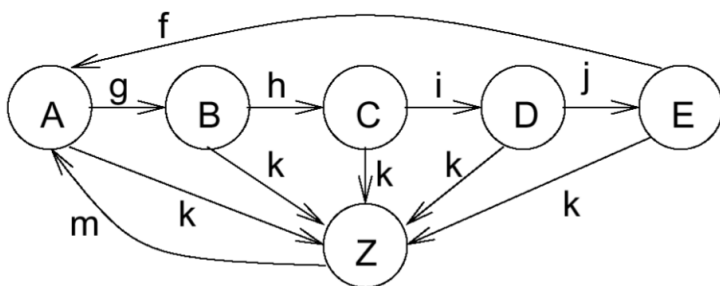
- Der nächste Zustand Z_{n+1} ist abhängig vom Input X und von Z_n : $Z_{n+1} = f(Z_n, X)$
- Der Output Y ist **nur** abhängig vom inneren Zustand Z_n : $Y = g(Z_n)$
- Die Actions liegen deshalb bei den Zuständen

4.1.6 Zustandstabelle

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 anz.
	AUS-Taste	AUS	
Störung	RESET-Taste	AUS	Motor ausschalten Kühlung ausschalten Rote Lampe anz.

4.1.7 Nachteile von Zustandsdiagrammen

- Für einen Resetzustand muss von jedem State zum Resetstate eine Transition gezeichnet werden.
- Da Zustandsdiagramme flach sind (es gibt keine Hierarchie), werden sie bei praktisch relevanten Aufgaben schnell unübersichtlich
- In Zustandsdiagrammen kann keine zeitliche Parallelität modelliert werden



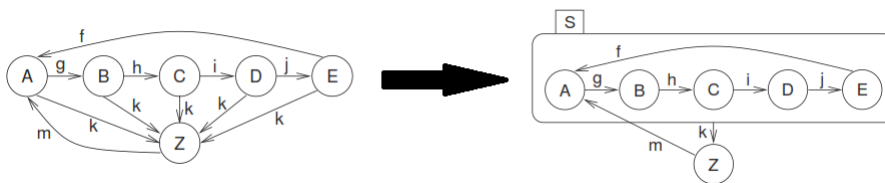
4.2 Statecharts von Harel

4.2.1 Elemente der Statechart

- Anfangszustand (initial pseudo state)
- ◇ Entscheidung (choice)
- Kreuzung (junction)
- × Terminator (terminate pseudo state)
- Ⓗ flache Historie (shallow history): History-Mechanismus merkt sich frühere Zustände
- Ⓗ* tiefe Historie (deep history): Deep History-Mechanismus merkt sich Zustände bis in die unterste Hierarchie
- Eintrittspunkt (entry point)
- ⊗ Austrittspunkt (exit point)

4.2.2 Modellierung von einfacher Hierarchie

- Hierarchie wird mit Hilfe von Superzuständen (superstates) eingeführt.
- Zustände, die andere Zustände enthalten, heissen Superzustände.
- Zustände, die in anderen Zuständen enthalten sind, heissen Unterzustände der Superzustände.

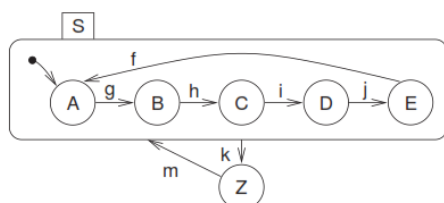


Superzustand S beinhaltet die Zustände A, B, C, D und E. Angenommen, der Automat befindet sich in Zustand Z (wird auch als aktiven Zustand beschrieben). Wenn dann die Eingabe m am Automaten erfolgt, ist A der nächste Zustand. Wenn sich der Automat in Zustand S befindet und es gibt die Eingabe k, so ist Z der nächste Zustand, unabhängig davon, in welchem der Unterzustände A, B, C, D oder E sich der Automat tatsächlich befindet. In diesem Beispiel sind alle in S enthaltenen Zustände nicht-hierarchische Zustände. Im Allgemeinen könnten die Unterzustände von S selbst wieder Superzustände sein, die weitere Unterzustände enthalten.

- Jeder Zustand, der nicht aus anderen Zuständen besteht, heisst Basiszustand.
- Superzustände S heissen ODER-Superzustände, wenn das System, das S enthält, sich zu jedem Zeitpunkt lediglich in einem einzigen Unterzustand von S befinden kann, solange es sich in S befindet.

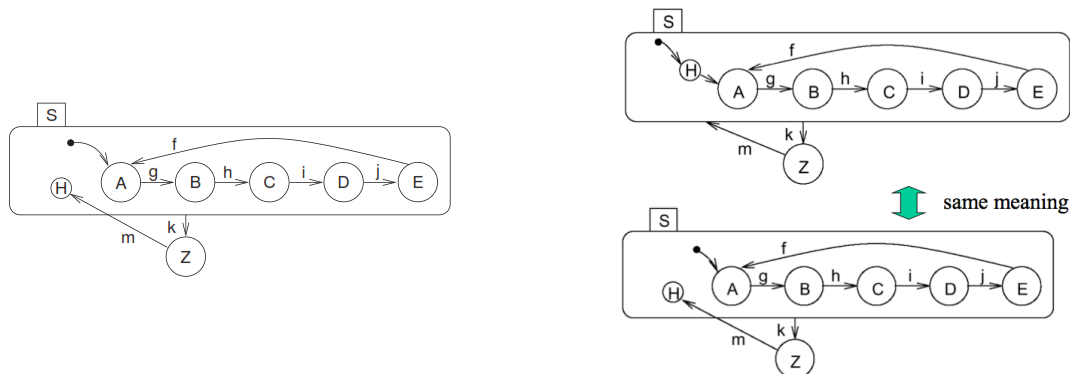
4.2.3 Modellierung von Hierarchie mit Default State (Standardzustand)

Der Standardzustand kann in Superzuständen verwendet werden, um anzuzeigen, welche der Unterzustände betreten werden, wenn der Superzustand betreten wird. Dies wird mit einem kleinen ausgefüllten Kreis dargestellt.



4.2.4 Modellierung von Hierarchie mit History

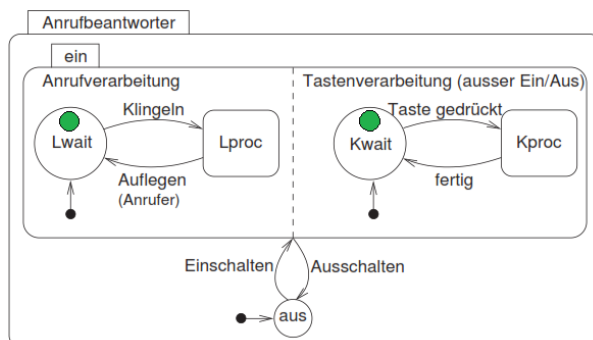
Mit Hilfe des History-Mechanismus ist es möglich, in den letzten Unterzustand zurückzukehren, der aktiv war, bevor der Superzustand verlassen wurde. Der History-Mechanismus wird durch den Buchstaben H in einem Kreis dargestellt.



4.2.5 Parallelität (UND-Superzustände)

Eine Spezifikationstechnik muss auch in der Lage sein, Nebenläufigkeit und Parallelität darzustellen. Zu diesem Zweck gibt es eine zweite Art von Superzuständen, die sogenannten UND-Superzustände.

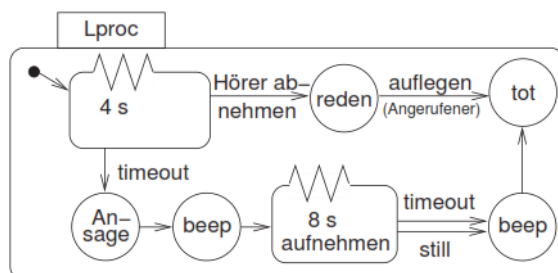
- Superzustände S heissen UND-Superzustände, wenn das System, das S enthält, sich in allen Unterzuständen von S gleichzeitig befindet, solange es sich in S befindet.



4.2.6 Zeitbedingungen

Da es notwendig ist, in eingebetteten Systemen Zeitbedingungen zu modellieren, bietet Statecharts die sogenannten Timer an. Zeitbedingungen werden durch das gezackte Symbol dargestellt.

Wenn das System für die im Timer angegebene Zeitdauer im Timer -Zustand war, wird ein Time-Out ausgelöst und das System verlässt diesen Zustand. Timer können auch hierarchisch verwendet werden.

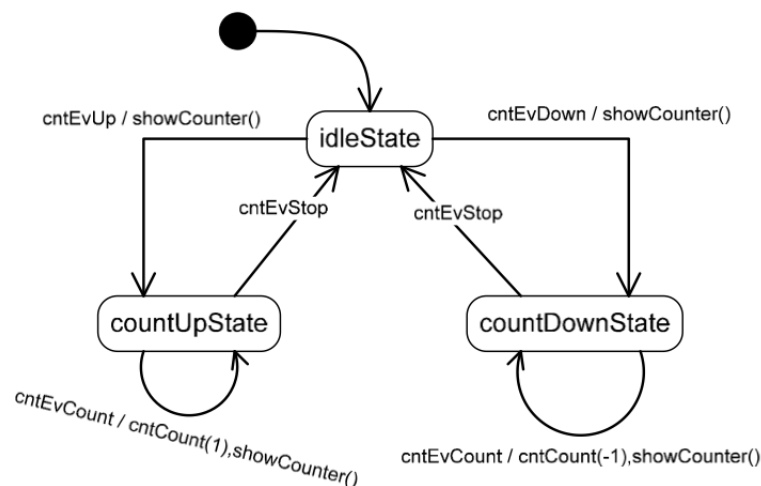


5 Fine State Machine Realisierung

5.1 Realisierung 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

5.2 Realisierung mit Steuerkonstrukt (prozedural in C)



- Zustände (States) werden in einem enum definiert (nicht public!)

```

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

```

- Ereignisse (Events) werden in einem enum definiert (public!)

```

1 | typedef enum {cntEvUp,             // count upwards
2 |               cntEvDown,          // count downwards
3 |               cntEvCount,         // count (up or down)
4 |               cntEvStop}          // stop counting
5 | CntEvent;

```

- Die FSM wird in zwei Funktionen implementiert

```

1 | void cntCtrlInit(int initValue);
2 | // initializes counter FSM
3 |
4 | void cntCtrlProcess(CntEvent e);
5 | // changes the state of the FSM based on the event 'e'
6 | // starts the actions

```

- Der aktuelle Zustand der FSM wird in einer statischen Variablen gehalten

```

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

```

5.2.1 Vollständiger Code für prozedurales Steuerkonstrukt

```

1 | //
2 | // counterCtrl.h
3 | //
4 | // implements the Finite State Machine (FSM) of an up/down-Counter
5 | //
6 | // (C) R. Bonderer, HSR Hochschule Rapperswil, Oct. 2014
7 | //
8 |
9 | #ifndef COUNTERCTRL_H__
10 | #define COUNTERCTRL_H__
11 |
12 | typedef enum {cntEvUp,          // count upwards
13 |              cntEvDown,        // count downwards
14 |              cntEvCount,        // count (up or down)
15 |              cntEvStop}         // stop counting
16 |              CntEvent;
17 |
18 | void cntCtrlInit(int initValue);
19 | // initializes counter FSM
20 |
21 | void cntCtrlProcess(CntEvent e);
22 | // changes the state of the FSM based on the event 'e'
23 | // starts the actions
24 |
25 | #endif

```

```

1 | //
2 | // counterCtrl.c
3 | //
4 | // implements the Finite State Machine (FSM) of an up/down-Counter
5 | //
6 | // (C) R. Bonderer, HSR Hochschule Rapperswil, Okt. 2011
7 | //
8 |
9 | #include <stdio.h>
10 | #include "counterCtrl.h"
11 | #include "counter.h"
12 |
13 | typedef enum {idleState,        // idle state
14 |              countUpState,      // counting up at each count event
15 |              countDownState}    // counting down at each count event
16 |              State;
17 |
18 | static State currentState = idleState; // holds the current state of the FSM
19 |
20 | void cntCtrlInit(int initValue)
21 | {
22 |     currentState = idleState;
23 |     cntInit(initValue);

```

```

24 }
25
26
27 void cntCtrlProcess(CntEvent e)
28 {
29     switch (currentState)
30     {
31         case idleState:
32             printf("State: idleState\n");
33             if (cntEvUp == e)
34             {
35                 // actions
36                 printf("State: idleState, counter = %d\n", cntGetCounter());
37                 // state transition
38                 printf("Changing to State: countUpState\n");
39                 currentState = countUpState;
40             }
41             else if (cntEvDown == e)
42             {
43                 // actions
44                 printf("State: idleState, counter = %d\n", cntGetCounter());
45                 // state transition
46                 printf("Changing to State: countDownState\n");
47                 currentState = countDownState;
48             }
49             break;
50
51         case countUpState:
52             printf("State: countUpState\n");
53             if (cntEvCount == e)
54             {
55                 // actions
56                 cntCount(1);
57                 printf("State: countUpState, counter = %d\n", cntGetCounter());
58                 // state transition
59             }
60             else if (cntEvStop == e)
61             {
62                 // actions
63                 // state transition
64                 printf("Changing to State: idleState\n");
65                 currentState = idleState;
66             }
67             break;
68
69         case countDownState:
70             printf("State: countDownState\n");
71             if (cntEvCount == e)
72             {
73                 // actions
74                 cntCount(-1);
75                 printf("State: countDownState, counter = %d\n", cntGetCounter());
76                 // state transition
77             }
78             else if (cntEvStop == e)
79             {
80                 // actions
81                 // state transition
82                 printf("Changing to State: idleState\n");
83                 currentState = idleState;
84             }
85             break;
86

```

```
87 |         default:
88 |             break;
89 |     }
90 | }

1 | //
2 | // counter.h
3 | //
4 | // implements an up/down-Counter
5 | //
6 | // (C) R. Bonderer, HSR Hochschule Rapperswil, Nov. 2010
7 | //
8 |
9 | #ifndef COUNTER_H__
10 | #define COUNTER_H__
11 |
12 | void cntInit(int val);
13 | // initializes counter to val
14 |
15 | void cntCount(int step);
16 | // counts the counter up (step>0) or down (step<0) by step
17 |
18 | int cntGetCounter();
19 | // returns the counter value
20 |
21 | void cntSetCounter(int val);
22 | // sets the counter to val
23 |
24 | #endif

1 | //
2 | // counter.c
3 | //
4 | // implements an up/down-Counter
5 | //
6 | // (C) R. Bonderer, HSR Hochschule Rapperswil, Nov. 2010
7 | //
8 |
9 | #include "counter.h"
10 |
11 | static int countValue;
12 |
13 | void cntInit(int val)
14 | {
15 |     countValue = val;
16 | }
17 |
18 | void cntCount(int step)
19 | {
20 |     countValue += step;
21 | }
22 |
23 | int cntGetCounter()
24 | {
25 |     return countValue;
26 | }
27 |
28 | void cntSetCounter(int val)
29 | {
30 |     countValue = val;
31 | }

1 | //
2 | // counterTest.c
```

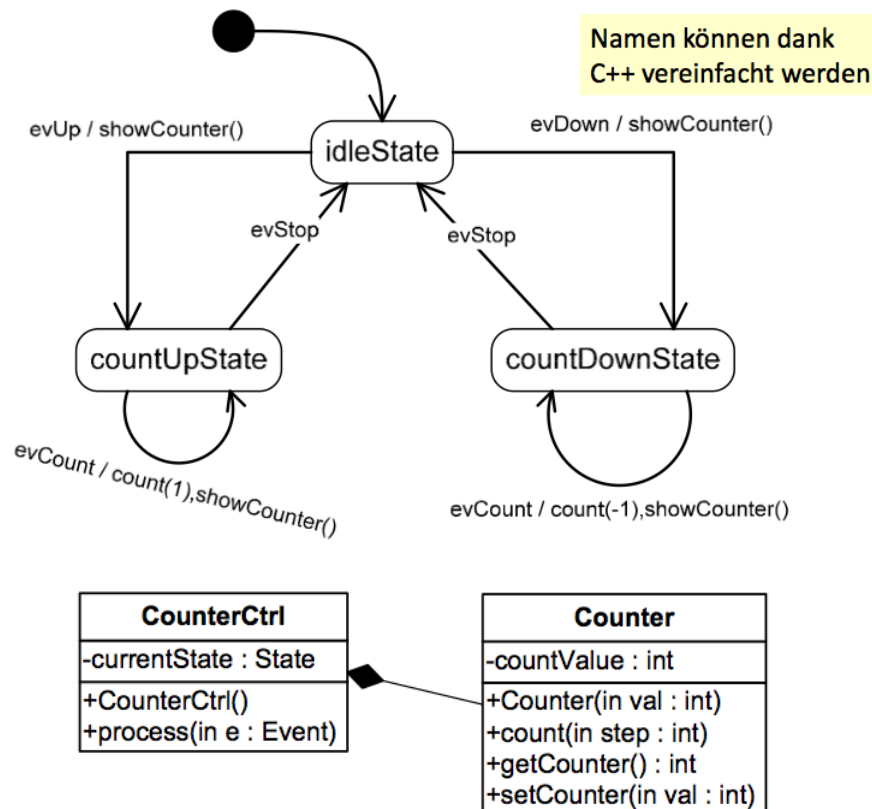
```

3 //
4 // Test program for the Finite State Machine (FSM) of an up/down-Counter
5 //
6 // (C) R. Bonderer, HSR Hochschule Rapperswil, Oct. 2014
7 //
8
9 #include <stdio.h>
10 #include "counterCtrl.h"
11
12 int main(void)
13 {
14     char answer;
15
16     cntCtrlInit(0);
17
18     do
19     {
20         printf("\n-----\n");
21         printf("    u    Count up\n");
22         printf("    d    Count down\n");
23         printf("    c    Count\n");
24         printf("    s    Stop counting\n");
25         printf("    q    Quit\n");
26
27         printf("\nPlease press key: ");
28         scanf("%c", &answer);
29         getchar(); // nach scanf() ist noch ein '\n' im Inputbuffer: auslesen und wegwerfen
30         printf("\n");
31
32         switch (answer)
33         {
34             case 'u':
35                 cntCtrlProcess(cntEvUp);
36                 break;
37             case 'd':
38                 cntCtrlProcess(cntEvDown);
39                 break;
40             case 'c':
41                 cntCtrlProcess(cntEvCount);
42                 break;
43             case 's':
44                 cntCtrlProcess(cntEvStop);
45                 break;
46             default:
47                 break;
48         }
49     } while (answer != 'q');
50
51     return 0;
52 }

```

5.3 Realisierung mit Steuerkonstrukt (Objektorientiert in C++)

- Logisch gesehen funktioniert die objektorientierte Variante völlig identisch wie die prozedurale
- Dank des Klassenkonstrukts kann die FSM sauber gekapselt werden
- Der aktuelle Zustand der FSM (currentState) wird in einem Attribut der Klasse gehalten
- Mehrere Instanzen derselben FSM können einfach erstellt werden
- Ein Modulkürzel ist nicht notwendig, da alle Namen im Kontext von Klassen definiert werden
- Der Code wird eleganter, eine Performanceeinbusse ist nicht vorhanden



- States werden im **privaten-Teil** der Klasse mit einem enum definiert

```

1 | enum State{idleState,           // idle state
2 |           countUpState,        // counting up at each count event
3 |           countDownState};    // counting down at each count event
  
```

- Events werden im **public-Teil** der Klasse mit einem enum definiert (public, weil die Events zur Schnittstelle gehören)

```

1 | enum Event{evUp,               // count upwards
2 |           evDown,             // count downwards
3 |           evCount,            // count (up or down)
4 |           evStop};            // stop counting
  
```

- Die FSM wird in zwei Funktionen implementiert

```

1 | CounterCtrl(int initValue = 0);
2 | void process(Event e);
3 | // changes the state of the FSM based on the event 'e'
4 | // starts the actions
  
```

- **Entry-Actions** müssen überall dort hinzugefügt werden, wo in einen neuen Zustand gewechselt wird. Üblicherweise muss die Entry-Action für einen bestimmten Zustand bei mehreren Transitionen codiert werden.
- **Exit-Actions** müssen überall dort hinzugefügt werden, wo ein Zustand verlassen, d.h. in einen anderen Zustand gewechselt, wird. Üblicherweise muss die Exit-Action für einen bestimmten Zustand bei mehreren Transitionen codiert werden.

5.3.1 Vollständiger Code für objektorientiertes Steuerkonstrukt

```

1 //
2 // counterCtrl.h
3 //
4 // implements the Finite State Machine (FSM) of an up/down-Counter
5 //
6 // (C) R. Bonderer, HSR Hochschule Rapperswil, Okt. 2013
7 //
8
9 #ifndef COUNTERCTRL_H__
10 #define COUNTERCTRL_H__
11 #include "counter.h"
12
13 class CounterCtrl
14 {
15 public:
16     enum Event{evUp,          // count upwards
17                evDown,        // count downwards
18                evCount,        // count (up or down)
19                evStop};        // stop counting
20     CounterCtrl(int initValue = 0);
21     void process(Event e);
22     // changes the state of the FSM based on the event 'e'
23     // starts the actions
24
25 private:
26     enum State{idleState,      // idle state
27                countUpState,    // counting up at each count event
28                countDownState}; // counting down at each count event
29
30     State currentState;        // holds the current state of the FSM
31     Counter myCounter;
32 };
33 #endif

```

```

1 //
2 // counterCtrl.cpp
3 //
4 // implements the Finite State Machine (FSM) of an up/down-Counter
5 //
6 // (C) R. Bonderer, HSR Hochschule Rapperswil, Okt. 2014
7 //
8
9 #include <iostream>
10 #include "counterCtrl.h"
11 #include "counter.h"
12 using namespace std;
13
14 CounterCtrl::CounterCtrl(int initValue) :
15     currentState(idleState),
16     myCounter(initValue)
17 {
18 }
19
20 void CounterCtrl::process(Event e)
21 {
22     switch (currentState)
23     {
24     case idleState:
25         cout << "State: idleState" << endl;
26         if (evUp == e)
27         {
28             // actions
29             cout << "State: idleState, counter = " << myCounter.getCount() << endl;
30             // state transition

```

```

31     cout << "Changing to State: countUpState" << endl;
32     currentState = countUpState;
33 }
34 else if (evDown == e)
35 {
36     // actions
37     cout << "State: idleState, counter = " << myCounter.getCount() << endl;
38     // state transition
39     cout << "Changing to State: countDownState" << endl;
40     currentState = countDownState;
41 }
42 break;
43
44 case countUpState:
45     cout << "State: countUpState" << endl;
46     if (evCount == e)
47     {
48         // actions
49         myCounter.count(1);
50         cout << "State: countUpState, counter = " << myCounter.getCount() << endl;
51         // state transition
52     }
53     else if (evStop == e)
54     {
55         // actions
56         // state transition
57         cout << "Changing to State: idleState" << endl;
58         currentState = idleState;
59     }
60     break;
61
62 case countDownState:
63     cout << "State: countDownState" << endl;
64     if (evCount == e)
65     {
66         // actions
67         myCounter.count(-1);
68         cout << "State: countDownState, counter = " << myCounter.getCount() << endl;
69         // state transition
70     }
71     else if (evStop == e)
72     {
73         // actions
74         // state transition
75         cout << "Changing to State: idleState" << endl;
76         currentState = idleState;
77     }
78     break;
79
80 default:
81     break;
82 }
83 }

```

5.4 Realisierung mit Tabelle

- Die Tabelle kann sowohl prozedural als auch objektorientiert implementiert werden.
- Die objektorientierte Variante verwendet einzig die Datenkapselung, Vererbung und Polymorphismus werden nicht benötigt
- Die objektorientierte Variante kann klarer und schöner strukturiert implementiert werden. Im folgenden wird nur diese Variante gezeigt, die C-Version kann jedoch einfach davon abgeleitet werden

- Die ganze FSM ist in einer Tabelle gespeichert
- Die Aktionen sind als Funktionen implementiert, in der Tabelle steht der entsprechende Funktionspointer
- Die Abarbeitung der FSM erfolgt mit Hilfe einer *Execution Engine*, die in der Tabelle "nachschaut", was zu tun ist
- Die Execution Engine ändert sich nicht, wenn die FSM geändert wird
- Das Testprogramm ist völlig unverändert
- Die Schnittstelle von CounterCtrl (public-Teil) ist ebenfalls identisch
- Die Klasse Counter ändert auch nicht
- Die **einzige Änderung** liegt im privaten Teil der CounterCtrl-Klasse und natürlich in deren Implementation
- Alle (Tansitions-)Aktionen werden als Methoden deklariert, *Action* wird als Funktionspointer definiert. Diese Methoden müssen alle mit dem Funktionspointer übereinstimmen

```

1 | typedef void (CounterCtrl::*Action)(void);    // function ptr for action function
2 |
3 | // action functions
4 | void actionIdleUp(void);
5 | void actionIdleDown(void);
6 | void actionDoNothing(void);
7 | void actionUpUp(void);
8 | void actionDownDown(void);

```

- Die Transition wird als klasseninterne Struktur deklariert. Sie besteht aus
 - Aktueller Zustand
 - Event
 - Funktionspointer auf Aktionsmethode
 - Nächster Zustand

fsm() wird als statischer Array deklariert.

```

1 | struct Transition
2 | {
3 |     State currentState;    // current state
4 |     Event ev;              // event triggering the transition
5 |     Action pAction;        // pointer to action function
6 |     State nextState;       // next state
7 | };

```

- Tabellendefinition in CounterCtrl.cpp

Aktueller Zustand	Event	Aktion	Nächster Zustand
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

```

1 | const CounterCtrl::Transition CounterCtrl::fsm[] = // this table defines the fsm
2 | { //currentState      triggering event  action function      next state
3 |   {idleState,         evUp,            &CounterCtrl::actionIdleUp,   countUpState},
4 |   {idleState,         evDown,          &CounterCtrl::actionIdleDown, countDownState},
5 |   {countUpState,      evCount,         &CounterCtrl::actionUpUp,     countUpState},
6 |   {countUpState,      evStop,          &CounterCtrl::actionDoNothing, idleState},
7 |   {countDownState,    evCount,         &CounterCtrl::actionDownDown, countDownState},
8 |   {countDownState,    evStop,          &CounterCtrl::actionDoNothing, idleState}
9 | };

```

- Aufbau einer Aktionsmethode

```

1 | void CounterCtrl::actionDownDown(void)
2 | {
3 |     myCounter.count(-1);
4 |     cout << "State: countDownState, counter = " << myCounter.getCounter() << endl;
5 | }

```

- Execution Engine

```

1 | void CounterCtrl::process(Event e) // execution engine, this function never changes
2 | {
3 |     for (size_t i = 0; i < sizeof(fsm) / sizeof(Transition); ++i) // determine number of
4 |         transitions automatically
5 |     {
6 |         if (fsm[i].currentState == currentState && fsm[i].ev == e) // is there an entry in
7 |             the table?
8 |         {
9 |             (this->*fsm[i].pAction)();
10 |            currentState = fsm[i].nextState;
11 |            break;
12 |        }
13 |    }
14 | }

```

- Wenn der Zustandsübergang nicht nur durch einen Event, sondern eine komplexe Prüfung (Event und Guard) ausgelöst wird, dann könnte der Eventeintrag in der Tabelle durch einen weiteren Funktionspointer auf eine Checkfunktion ersetzt werden

```

1 | typedef bool (CounterCtrl::*Checker)(Event); // function ptr for checker function
2 |
3 | bool checkIdleUp(Event check);
4 |
5 | struct Transition
6 | {
7 |     State currentState; // current state
8 |     Checker pChecker;    // pointer to checker function
9 |     Action pAction;      // pointer to action function
10 |    State nextState;      // next state
11 | };

```

5.4.1 Vollständiger Code für Tabellen-Realisation

```

1 | //
2 | // counterCtrl.h
3 | //
4 | // implements the Finite State Machine (FSM) of an up/down-Counter as a table
5 | //
6 | // (C) R. Bonderer, HSR Hochschule Rapperswil, Okt. 2013
7 | //
8 |

```

```

9  #ifndef COUNTERCTRL_H__
10 #define COUNTERCTRL_H__
11 #include "counter.h"
12
13 class CounterCtrl
14 {
15     public:
16         enum Event{evUp,          // count upwards
17                     evDown,        // count downwards
18                     evCount,       // count (up or down)
19                     evStop};       // stop counting
20         CounterCtrl(int initValue = 0);
21         void process(Event e);
22         // changes the state of the FSM based on the event 'e'
23         // starts the actions
24
25     private:
26         enum State{idleState,      // idle state
27                     countUpState,   // counting up at each count event
28                     countDownState}; // counting down at each count event
29
30         State currentState;         // holds the current state of the FSM
31         Counter myCounter;
32
33         typedef bool (CounterCtrl::*Checker)(Event); // function ptr for checker function
34         typedef void (CounterCtrl::*Action)(void);   // function ptr for action function
35         // check functions
36         bool checkIdleUp(Event check);
37         bool checkIdleDown(Event check);
38         bool checkUpIdle(Event check);
39         bool checkDownIdle(Event check);
40         bool checkUpUp(Event check);
41         bool checkDownDown(Event check);
42
43         // action functions
44         void actionIdleUp(void);
45         void actionIdleDown(void);
46         void actionUpIdle(void);
47         void actionDownIdle(void);
48         void actionUpUp(void);
49         void actionDownDown(void);
50
51         struct Transition
52         {
53             State currentState; // current state
54             Checker pChecker;    // pointer to checker function
55             Action pAction;      // pointer to action function
56             State nextState;     // next state
57         };
58         static const Transition fsm[];
59 };
60 #endif

```



```

1  //
2  // counterCtrl.cpp
3  //
4  // implements the Finite State Machine (FSM) of an up/down-Counter as a table
5  //
6  // (C) R. Bonderer, HSR Hochschule Rapperswil, Okt. 2013
7  //
8
9  #include <iostream>
10 #include "counterCtrl.h"
11 #include "counter.h"

```

```

12 using namespace std;
13
14 const CounterCtrl::Transition CounterCtrl::fsm[] = // this table defines the fsm
15 {//currentState checker function action function next state
16   {idleState,      &CounterCtrl::checkIdleUp,    &CounterCtrl::actionIdleUp,    countUpState},
17   {idleState,      &CounterCtrl::checkIdleDown,  &CounterCtrl::actionIdleDown, countDownState
18   },
19   {countUpState,   &CounterCtrl::checkUpUp,      &CounterCtrl::actionUpUp,     countUpState},
20   {countUpState,   &CounterCtrl::checkUpIdle,    &CounterCtrl::actionUpIdle,   idleState},
21   {countDownState, &CounterCtrl::checkDownDown,  &CounterCtrl::actionDownDown, countDownState
22   },
23   {countDownState, &CounterCtrl::checkDownIdle, &CounterCtrl::actionDownIdle, idleState}
24 };
25
26 CounterCtrl::CounterCtrl(int initValue) :
27     currentState(idleState),
28     myCounter(initValue)
29 {
30 }
31
32 void CounterCtrl::process(Event e) // this function never changes
33 {
34     for (size_t i = 0; i < sizeof(fsm) / sizeof(Transition); ++i) // determine number of
35         transitions automatically
36     {
37         if (fsm[i].currentState == currentState && // is there an entry in the table?
38             (this->*fsm[i].pChecker)(e))
39         {
40             (this->*fsm[i].pAction)();
41             currentState = fsm[i].nextState;
42             break;
43         }
44     }
45 }
46
47 // check functions
48 bool CounterCtrl::checkIdleUp(Event check)
49 {
50     return evUp == check;
51 }
52
53 bool CounterCtrl::checkIdleDown(Event check)
54 {
55     return evDown == check;
56 }
57
58 bool CounterCtrl::checkUpIdle(Event check)
59 {
60     return evStop == check;
61 }
62
63 bool CounterCtrl::checkDownIdle(Event check)
64 {
65     return evStop == check;
66 }
67
68 bool CounterCtrl::checkUpUp(Event check)
69 {
70     return evCount == check;
71 }
72
73 bool CounterCtrl::checkDownDown(Event check)
74 {

```

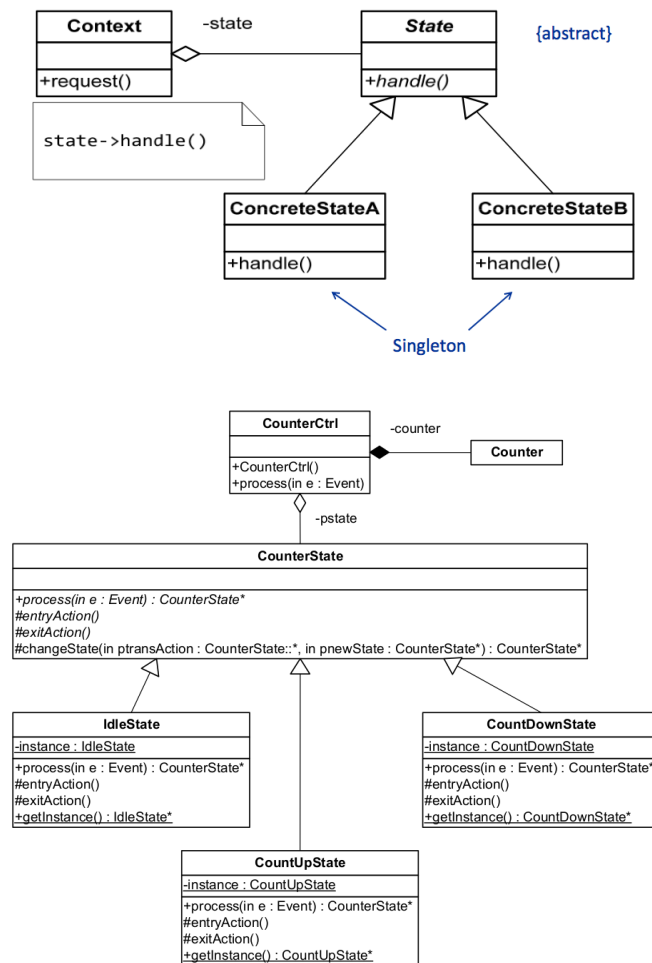
```

72     return evCount == check;
73 }
74
75 // action functions
76 void CounterCtrl::actionIdleUp(void)
77 {
78     cout << "State: idleState, counter = " << myCounter.getCount() << endl;
79 }
80
81 void CounterCtrl::actionIdleDown(void)
82 {
83     cout << "State: idleState, counter = " << myCounter.getCount() << endl;
84 }
85
86 void CounterCtrl::actionUpIdle(void)
87 {
88 }
89
90 void CounterCtrl::actionDownIdle(void)
91 {
92 }
93
94 void CounterCtrl::actionUpUp(void)
95 {
96     myCounter.count(1);
97     cout << "State: countUpState, counter = " << myCounter.getCount() << endl;
98 }
99
100 void CounterCtrl::actionDownDown(void)
101 {
102     myCounter.count(-1);
103     cout << "State: countDownState, counter = " << myCounter.getCount() << endl;
104 }

```

5.5 Realisierung mit State Pattern

- Das Prinzip besteht aus einer abstrakten State-Basisklasse. Pro Zustand muss eine eigene Klasse definiert werden, die von dieser abstrakten Basisklasse erbt.
- Die Realisierung nutzt das Polymorphismus-Konzept, jede konkrete State-Klasse ist ein *Singleton*
- Statt lange switch-case-if-Konstrukte wird beim State Pattern ein bestimmter Zustand vollständig in einer eigenen Klasse realisiert. Die Logik wird somit aufgeteilt.
- **Context** definiert die interessierenden Schnittstellen für Clients, unterhält eine Instanz einer konkreten Unterklasse von State, die den aktuellen Zustand definiert, bzw. repräsentiert
- **State** definiert eine schnittstelle zur FMS in Form einer abstrakten Klasse
- **ConcreteStateX Unterklassen** jede Unterklasse implementiert genau einen Zustand (ist Singleton)
- Das State Pattern definiert die Zuständigkeit der state Transition nicht
- Die Transitionen könnten in der Context-Klasse definiert werden. Der Nachteil dieser Variante ist, dass dort zentral sehr viel Intelligenz vorhanden sein müsste. Da diese Klasse auch den Zugriff zur Aussenwelt darstellt, sollte sie möglichst schlank sein.
- Die bessere Variante ist, wenn die State-Klassen auch gleich ihre Transitionen realisieren. Diese Variante wird oft mittels friend-Deklaration realisiert
- Im folgenden betrachten wir eine saubere Lösung mit friend-Deklaration.



- counterCtrl.h : Schnittstelle zur FSM

```

1 //
2 // counterCtrl.h
3 //
4 // implements the Finite State Machine (FSM) of an up/down-Counter
5 //
6 // (C) R. Bonderer, HSR Hochschule Rapperswil, Oct. 2013
7 //
8
9 #ifndef COUNTERCTRL_H_
10 #define COUNTERCTRL_H_
11 #include "counter.h"
12
13 class CounterState; // forward declaration
14
15 class CounterCtrl
16 // this is the 'Context' class of the State pattern
17 {
18 public:
19     enum Event{evUp, // count upwards
20                evDown, // count downwards
21                evCount, // count (up or down)
22                evStop}; // stop counting
23     CounterCtrl(int initValue = 0);
24     void process(Event e);
25     // changes the state of the FSM based on the event 'e'
26 private:
  
```

```

27     Counter entity;           // entity class
28     CounterState* pState;     // holds the current state
29 };
30 #endif

```

- counterCtrl.cpp

```

1  //
2  // counterCtrl.cpp
3  //
4  // implements the Finite State Machine (FSM) of an up/down-Counter
5  // CounterCtrl is the Context class in the State pattern
6  //
7  // (C) R. Bonderer, HSR Hochschule Rapperswil, Nov. 2014
8  //
9
10 #include "counter.h"
11 #include "counterState.h"
12 #include "counterCtrl.h"
13
14 CounterCtrl::CounterCtrl(int initValue) :
15     entity(initValue),
16     pState(IdleState::getInstance()) // initial state
17 {
18     pState = pState->init(entity);
19 }
20
21 void CounterCtrl::process(Event e)
22 { // delegates all requests to CounterState
23     pState = pState->process(entity, e); // entity: reference to entity class object
24 }

```

- class CounterState: Interface

```

1  //
2  // counterState.h
3  //
4  // implements an up/down-Counter
5  // this file contains all classes of the state machine
6  // it may make sense to have separate files for each state
7  //
8  // (C) R. Bonderer, HSR Hochschule Rapperswil, Nov. 2014
9  //
10
11 #ifndef COUNTERSTATE_H__
12 #define COUNTERSTATE_H__
13 #include "counterCtrl.h"
14
15 class CounterState // abstract base class
16 {
17 public:
18     CounterState* init(Counter& entity);
19     // should be called first, returns new state
20     virtual CounterState* process(Counter& entity, CounterCtrl::Event e) = 0;
21     // returns new state
22 protected:
23     virtual void entryAction(Counter& entity) {};
24     virtual void exitAction(Counter& entity) {};
25     typedef void (CounterState::*Action)(Counter& entity); // ptr to action function
26     CounterState* changeState(Counter& entity, Action ptransAction, CounterState*
27         pnewState);
28
29     // transition actions
30     void emptyAction(Counter& entity) {};

```

```

30     void showCounter(Counter& entity);
31     void countUp(Counter& entity);
32     void countDown(Counter& entity);
33 };
34
35 class IdleState : public CounterState // it's a singleton
36 {
37 public:
38     static IdleState* getInstance();
39     virtual CounterState* process(Counter& entity, CounterCtrl::Event e);
40 protected:
41     virtual void entryAction(Counter& entity);
42     virtual void exitAction(Counter& entity);
43 private:
44     IdleState() {};
45     static IdleState instance;
46 };
47
48 class CountUpState : public CounterState // it's a singleton
49 {
50 public:
51     static CountUpState* getInstance();
52     virtual CounterState* process(Counter& entity, CounterCtrl::Event e);
53 protected:
54     virtual void entryAction(Counter& entity);
55     virtual void exitAction(Counter& entity);
56 private:
57     CountUpState() {};
58     static CountUpState instance;
59 };
60
61 class CountDownState : public CounterState // it's a singleton
62 {
63 public:
64     static CountDownState* getInstance();
65     virtual CounterState* process(Counter& entity, CounterCtrl::Event e);
66 protected:
67     virtual void entryAction(Counter& entity);
68     virtual void exitAction(Counter& entity);
69 private:
70     CountDownState() {};
71     static CountDownState instance;
72 };
73 #endif

```

- class CounterState: Implementation

```

1 //
2 // counterState.cpp
3 //
4 // implements all states of an up/down-Counter
5 // this file contains all classes of the state machine.
6 // it may make sense to have separate files for each state
7 //
8 // (C) R. Bonderer, HSR Hochschule Rapperswil, Nov. 2014
9 //
10
11 #include <iostream>
12 #include "counterState.h"
13 using namespace std;
14
15
16 //class CounterState
17 CounterState* CounterState::init(Counter& entity)

```

```

18 {
19     entryAction(entity);          // executes entry action into init state
20     return IdleState::getInstance();
21 }
22
23 CounterState* CounterState::changeState(Counter& entity,
24                                         Action ptransAction,
25                                         CounterState* pnewState)
26 {
27     exitAction(entity);           // polymorphic call of exit action
28     (this->*ptransAction)(entity); // call of transition action
29     pnewState->entryAction(entity); // polymorphic call of entry action
30     return pnewState;
31 }
32
33 void CounterState::showCounter(Counter& entity)
34 {
35     cout << "counter = " << entity.getCount() << endl;
36 }
37
38 void CounterState::countUp(Counter& entity)
39 {
40     entity.count(1);
41     cout << "counter = " << entity.getCount() << endl;
42 }
43
44 void CounterState::countDown(Counter& entity)
45 {
46     entity.count(-1);
47     cout << "counter = " << entity.getCount() << endl;
48 }
49
50
51 // class IdleState
52 IdleState IdleState::instance;
53 IdleState* IdleState::getInstance()
54 {
55     return &instance;
56 }
57
58 CounterState* IdleState::process(Counter& entity, CounterCtrl::Event e)
59 {
60     cout << "State: idleState" << endl;
61     if (CounterCtrl::evUp == e)
62     {
63         // state transition
64         return changeState(entity, &IdleState::showCounter, CountUpState::getInstance());
65     }
66     else if (CounterCtrl::evDown == e)
67     {
68         // state transition
69         return changeState(entity, &IdleState::showCounter, CountDownState::getInstance());
70     }
71     return this;
72 }
73
74 void IdleState::entryAction(Counter& entity)
75 {
76     cout << "Entering idleState" << endl;
77 }
78
79 void IdleState::exitAction(Counter& entity)
80 {

```

```

81     cout << "Exiting from idleState" << endl;
82 }
83
84
85 // class CountUpState
86 CountUpState CountUpState::instance;
87 CountUpState* CountUpState::getInstance()
88 {
89     return &instance;
90 }
91
92 CounterState* CountUpState::process(Counter& entity, CounterCtrl::Event e)
93 {
94     cout << "State: countUpState" << endl;
95     if (CounterCtrl::evCount == e)
96     {
97         // state transition
98         return changeState(entity, &CountUpState::countUp, CountUpState::getInstance());
99     }
100     else if (CounterCtrl::evStop == e)
101     {
102         // state transition
103         return changeState(entity, &CountUpState::emptyAction, IdleState::getInstance());
104     }
105     return this;
106 }
107
108 void CountUpState::entryAction(Counter& entity)
109 {
110     cout << "Entering countUpState" << endl;
111 }
112
113 void CountUpState::exitAction(Counter& entity)
114 {
115     cout << "Exiting from countUpState" << endl;
116 }
117
118 // class CountDownState
119 CountDownState CountDownState::instance;
120 CountDownState* CountDownState::getInstance()
121 {
122     return &instance;
123 }
124
125 CounterState* CountDownState::process(Counter& entity, CounterCtrl::Event e)
126 {
127     cout << "State: countDownState" << endl;
128     if (CounterCtrl::evCount == e)
129     {
130         // state transition
131         return changeState(entity, &CountDownState::countDown, CountDownState::getInstance());
132     }
133     else if (CounterCtrl::evStop == e)
134     {
135         // state transition
136         return changeState(entity, &CountDownState::emptyAction, IdleState::getInstance());
137     }
138     return this;
139 }
140
141 void CountDownState::entryAction(Counter& entity)
142 {
143     cout << "Entering countDownState" << endl;

```

```
144 | }  
145 |  
146 | void CountdownState::exitAction(Counter& entity)  
147 | {  
148 |     cout << "Exiting from countdownState" << endl;  
149 | }
```

6 Modularisierung

Modularisierung ist die Aufteilung eines Ganzen in Teile, die als Module oder Komponenten bezeichnet werden. Bei geeigneter Form und Funktion können sie zusammengefügt werden oder über entsprechende Schnittstellen interagieren.

6.1 Motivation

Bei Kleinstsoftware braucht es kein systematisches Design, da man diese meist auch nicht ausbauen möchte. Vernünftige Softwaresysteme benötigen einen systematischen Designansatz mit einem strukturierten Aufbau.

Wichtig ist daher:

- Entwicklung im Team vs. One-person-show
- Schnittstellen definieren

6.2 Grundprinzip

- Divide et impera (Teile und herrsche)
 - Problem aufteilen in Unterprobleme, diese einzeln angehen
 - Zuerst Grobentwurf, dann verfeinern (Top-Down-Prinzip)
- **Ziel:** Reduktion der Komplexität

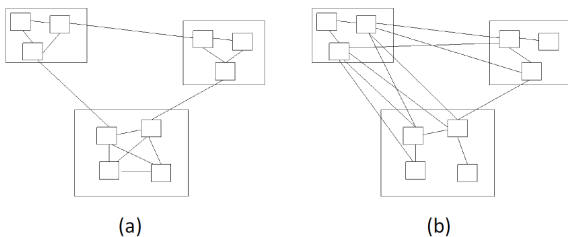
6.3 Information Hiding

- Nutzer des Moduls braucht nur die Schnittstelle zu kennen (*.h files)
- Über inneren Aufbau dürfen keine Annahmen getroffen werden
- Der innere Aufbau (*.cpp files) muss nicht bekannt sein

6.4 Hauptkriterien der Zerlegung

- Kopplung/Coupling
 - Mass für Komplexität der Schnittstelle
- Kohäsion/Cohesion
 - Aussage wie stark eine funktionale Einheit wirklich zusammengehört
 - Mass für die Stärke des inneren Zusammenhangs

Ziel: Schwache Kopplung, starke Kohäsion



- Bild a: Schwache Kopplung, starke Kohäsion (gut)
- Bild b: Starke Kopplung, schwache Kohäsion (schlecht)

6.5 Definitionen Kopplung

- Keine direkte Kopplung (Schwache Kopplung → **GUT**)
- Datenkopplung
 - Kommunikation ausschliesslich über Parameter
- Datenbereichskopplung
 - Modul hat Zugriff auf Datenstruktur eines anderen Moduls
 - Es werden nur einzelne Komponenten benötigt
- Steuerflussskopplung (Control Flow)
 - Modul beeinflusst Steuerfluss eines anderen Moduls
- Globale Kopplung
 - Kommunikation über globale Variablen
 - Jedes Modul hat Zugriff
- Inhaltskopplung (Starke Kopplung → **SCHLECHT**)
 - Aus einem Modul werden lokale Daten eines anderen Moduls modifiziert, obwohl dieses Modul NICHT vom anderen Modul aufgerufen worden ist.
 - **Todsünde**

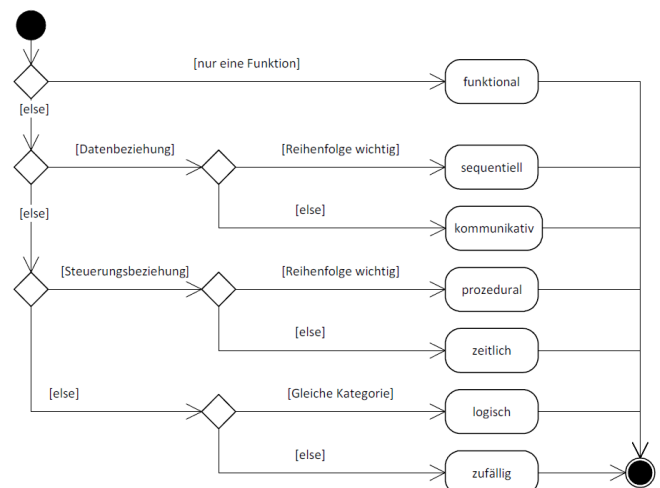
6.6.1 Ziele bezüglich Kohäsion

- Kohäsion maximieren
- Idealerweise führt eine Unit nur eine Aufgabe (Funktion) aus
- Starke Kohäsion steigert
 - Wartbarkeit
 - Änderbarkeit
 - Verständlichkeit
- **Zusammengehörendes zusammen nehmen!**
- **Starke Kohäsion führt zu schwacher Kopplung, aber nicht umgekehrt!!**

6.6 Definitionen Kohäsion

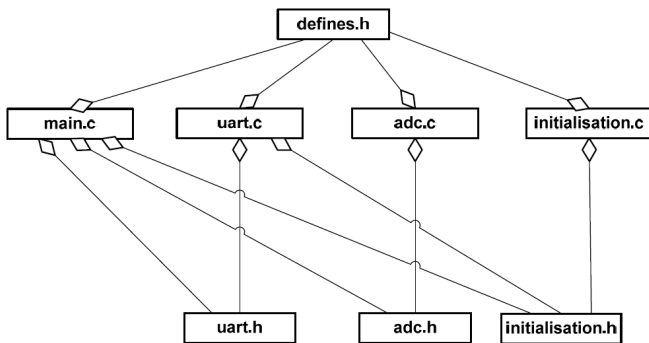
- Funktionale Kohäsion (Starke Kohäsion → **GUT**)
 - Teile einer Einheit bilden zusammen eine Funktion
- Sequentielle Kohäsion
 - Teilfunktionen einer Einheit werden nacheinander ausgeführt
 - Das Ergebnis einer Teilfunktion ist die Eingabe der nächsten
- Kommunikative Kohäsion
 - Teilfunktionen einer Einheit werden auf den gleichen Daten ausgeführt
 - Reihenfolge spielt keine Rolle
- Prozedurale Kohäsion
 - Teilfunktionen werden nacheinander ausgeführt
 - Sind über Steuerfluss verknüpft
- Zeitliche Kohäsion
 - Teile einer Einheit sind alle zu einer bestimmten auszuführen
- Logische Kohäsion
 - Teilfunktionen einer Einheit gehören zu einer Klasse
- Zufällige Kohäsion (Schwache Kohäsion → **SCHLECHT**)
 - Teilfunktionen einer Einheit haben keinen sinnvollen Zusammenhang

6.6.2 Bestimmung der Kohäsion

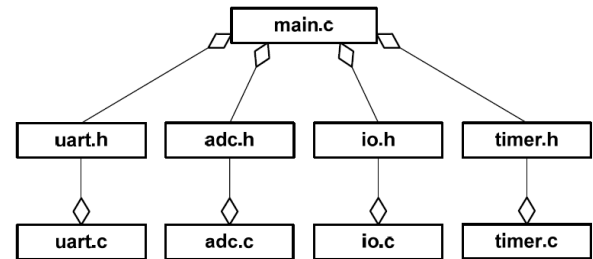


6.7 Schlechte vs. gute Modularisierung

Beispiel schlechter Modularisierung:



Beispiel guter Modularisierung:



7 Event Based Systems

7.1 Reaktive Systeme

Ein Reaktives System befindet sich in ständiger Interaktion mit der Umgebung, wobei die Umgebung dominiert und das System sich dieser unterordnet.

- Viele Embedded Systems sind auch reaktive Systeme
- Reaktive Systeme reagieren auf (oft externe) Ereignisse
 - Digitale Inputs
 - Erreichen von Analogen Schwellwerten
 - Timer
 - Buttonclicks und ähnliches in GUI
 - etc.

7.2 Umsetzung von Ereignissen

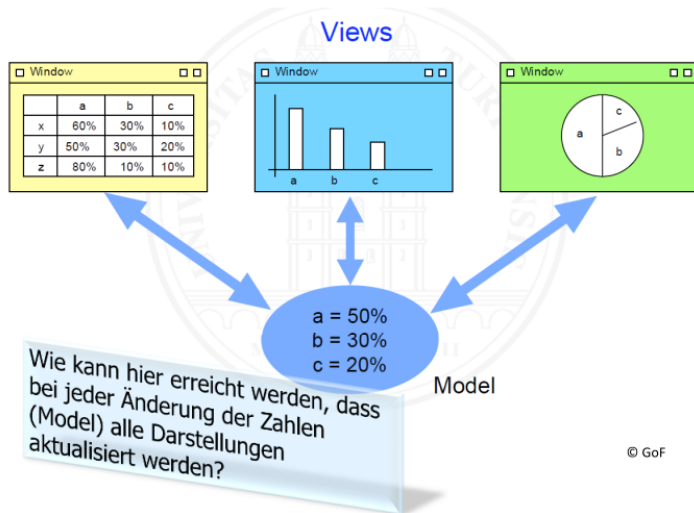
- Ereignisse sind per Definition asynchron
- Ein "normales" Programm ist immer synchron(zuerst das, dann das, dann...)
- Polling
 - + einfach zu implementieren
 - sehr viele Leerabfragen → unnötige Prozessorbelastung
 - Leerabfragen können durch periodisches Abfragen (Kopplung an Timer) reduziert aber nicht vermieden werden.
 - Die maximale Reaktionszeit wird durch die Abfrageperiode und die Anzahl Abfragen definiert
- Interrupt
 - Non-vectored Interrupt(zentral)
 - * Alle Interrupts verzweigen zu einer gemeinsamen Adresse. Dort wird die Ursache bestimmt und zu einer spezifischen Behandlungsroutine verzweigt.
 - Vectored Interrupt(spezifisch)
 - * In einer Tabelle(Interruptvektortabelle) wird gespeichert, wohin bei welchem Interruptvektor verzweigt werden muss.
- 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.

7.3 Model View Controller (MVC) aka Observer Pattern

- Beim MVC wirkt das Model als Server, die Views sind Clients
- Jeder Client meldet beim Server an, welche Ereignisse ihn interessieren
- Diese Anmeldung, bzw. Registrierung erfolgt über eine Funktion, welche der Server anbietet:

```
1 || int Foo_registerCb(Foo_Event e,
2 ||   Foo_CbFunction f);
   || // registers function 'f' on event 'e'
```

- Der Server trägt diesen Funktionspointer in eine Tabelle ein und ruft bei Eintreten des Ereignisses alle registrierten Funktionen auf.



© GoF

7.3.1 Umsetzung der Callback-Funktionen

- Funktionspointer zu Callback-Funktionen definieren Annahme: Callback-Funktionen haben die Schnittstelle `void f(int);`

```
1 || typedef void (*Foo_CbFunction)(int);
```

- Tabelle von Funktionspointern für jeden Event definieren

```
1 || Foo_CbFunction evClient[evNum] = {0};
```

- Aufruf der registrierten Clientfunktionen beim Eintreten des Events

```
1 || for (i = 0; i < evNum; ++i)
2 || {
3 ||     if (evClient[i] != 0) // entry found
4 ||     {
5 ||         evClient[i](arg); // call client through function ptr
6 ||     }
7 || }
```

7.3.2 Vorteile der Callback-Funktionen

- Die Views werden asynchron genau informiert, wenn sich etwas geändert hat im Model
- An und für sich sind alle registrierten Funktionen nichts anderes als Eventhandler eines bestimmten Events
- Dadurch wird die Darstellung (die Definition der registrierten Funktionen) sauber von den Daten (Model) entkoppelt

8 Patterns

8.1 Pattern-Philosophie

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

- Vorteile
 - Funktionierende Lösungen
 - Getestete Lösungen
 - Rad muss nicht immer neu erfunden werden
- Herausforderungen
 - Problemstellungen müssen als solche erkannt werden
 - Wichtige Patterns müssen bekannt sein

8.2 Einige wichtige Patterns

- Implementation von Finite State Machines
 - State Pattern
 - Singleton Pattern
 - Steuerkonstrukt mit switch-case
 - Tabellenvariante
- Mini-Patterns
 - Setzen und Löschen von einzelnen Bits
 - Behandlung von asynchronen Ereignissen
 - * Interrupts
 - * Polling
 - etc.
- Creational Patterns
 - Factory (Dependency injection)
 - RAI
 - Singleton
- Structural Patterns
 - Adapter
 - Facade
 - Proxy
- Behavioral Patterns
 - Adapter
 - Mediator
 - Observer (MVC)
- Concurrency Patterns
 - Active Object
 - Lock
 - Monitor

8.3 Creational Patterns

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

8.3.1 Factory (Dependency injection)

- Definition einer Schnittstelle zur Erzeugung eines Objekts, statt der direkten Erzeugung auf der Client-Seite.

8.3.2 RAI (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.

8.3.3 Singleton

- Ein Singleton stellt sicher, dass eine Klasse nur ein einziges Objekt besitzt.

8.4 Structural Patterns

Structural Patterns vereinfachen Beziehungen zu anderen Teilen.

8.4.1 Adapter (Wrapper, Translator, glue code)

- Wandelt (adaptiert) eine Schnittstelle in eine für einen Client passendere Schnittstelle um.

8.4.2 Facade

- Bietet eine einfache Schnittstelle für die Nutzung einer meist viel grösseren Library.

8.4.3 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.

8.5 Behavioral Patterns

Behavioral Patterns identifizieren gemeinsame Kommunikationspatterns zwischen Objekten und implementieren diese.

8.5.1 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.

8.5.2 Observer (MVC)

- Wird als objektorientierte Variante präsentiert.
- Wie wir gesehen haben, kann das MVC-Prinzip auch prozedural mit Callbackfunktionen implementiert werden.

8.6 Concurrency Patterns

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

8.6.1 Active Object

- Entkoppelt den Methodenaufruf von der Methodenausführung, die Methode soll sich nicht kümmern, in welchem Kontext sie aufgerufen wird.

8.6.2 Lock

- Ist eine Synchronisationsprimitive, welche den unteilbaren Zugriff read-modify-write implementiert.

8.6.3 Monitor

- Der Monitor versteckt Synchronisationsanforderungen vor dem Client.

9 Computerarithmetik

9.1 Ganzzahlig

Darstellung	Sign Magnitude	1er Komplement	2er Komplement
Zahlenbereich	$-(2^{n-1} - 1)$ bis $+(2^{n-1} - 1)$	$-(2^{n-1} - 1)$ bis $+(2^{n-1} - 1)$	-2^{n-1} bis $+(2^{n-1} - 1)$
Vorteil	Vorzeichen kann einfach durch Vorzeichenbit festgestellt werden.	Vorzeichen kann einfach durch MSB festgestellt werden. Addition und Subtraktion einfach realisierbar.	Vorzeichen kann einfach durch MSB festgestellt werden. Addition und Subtraktion einfach realisierbar
Nachteil	Positive und negative Null. Addition/Subtraktion kompliziert.	Positive und negative Null.	Bereich nicht ganz symmetrisch
Beispiel +5	0101	0101	0101
Beispiel -5	1101	1010	1011

9.1.1 Addition/Subtraktion

Wird direkt mit der Arithmetic Logic Unit (ALU) durchgeführt.

9.1.2 Multiplikation

9.1.2.1 Multiplikation mit Zweierpotenzen

Die Multiplikation mit Zweierpotenzen ist ein Spezialfall, der sehr effizient umgesetzt werden kann. Die Multiplikation einer Zahl x mit 2^n entspricht einem Schieben der Zahl x um n Stellen nach links.

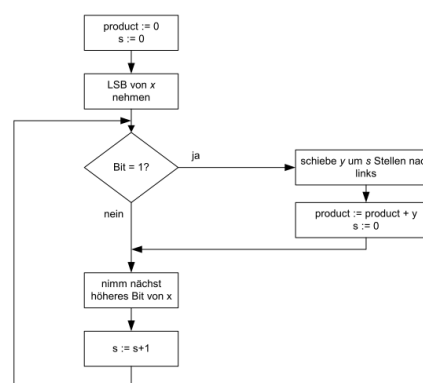
9.1.2.2 Allgemeine Multiplikation

Implementation algorithmisch: Langsam, wenig Aufwand

- Mikroprogramm
- Softwareemulation

Implementation Hardware: Schnell, viel Aufwand

- In vielen uC's drin, vor allem RISC (MSP430)
- Im DSP ist HW-Multiplizierer ein Muss



9.1.3 Division

9.1.3.1 Division mit Zweierpotenzen

Die Division durch Zweierpotenzen ist ein Spezialfall, der sehr effizient umgesetzt werden kann. Die Division einer Zahl x durch 2^n entspricht einem Schieben der Zahl x um n Stellen nach rechts.

9.1.3.2 Allgemeine Division

Implementation algorithmisch: Langsam, wenig Aufwand

- Mikroprogramm
- Softwareemulation

Implementation Hardware: Schnell, viel Aufwand

- In vielen uC's drin, vor allem RISC (MSP430)
- da die Division die seltenste Operation ist, wird speziell bei Microcontrollern nur selten ein Hardware-Dividierer verwendet

9.1.4 Skalierung

- Aus Effizienzgründen wird insbesondere bei Mikrocontrollern versucht, so viel wie möglich mit Integer-Arithmetik zu lösen
- Quantisierung auf 1 Bit kann bei kleinen Werten zu Ungenauigkeiten führen
- Lösung: durch Skalierung soll Dynamikbereich, d.h. der Wertebereich der ganzen Zahl, möglichst ausgenutzt werden
- Skalierungen mit Zweierpotenzen sind viel effizienter als solche mit Dezimalzahlen
- Wenn möglich immer mit Zweierpotenzen skalieren

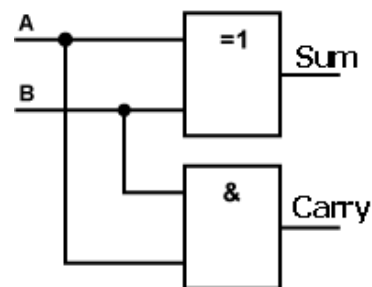
9.2 Arithmetische Schaltungen

9.2.1 Addierer

Wichtigste arithmetische Funktion, n-Addierer werden auf 1-Bit Addierer zurück geführt.

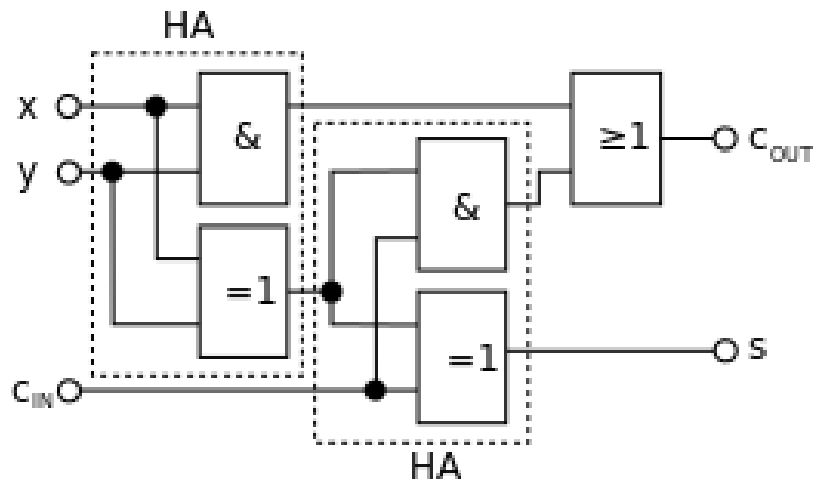
9.2.1.1 1 Bit Halbaddierer:

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

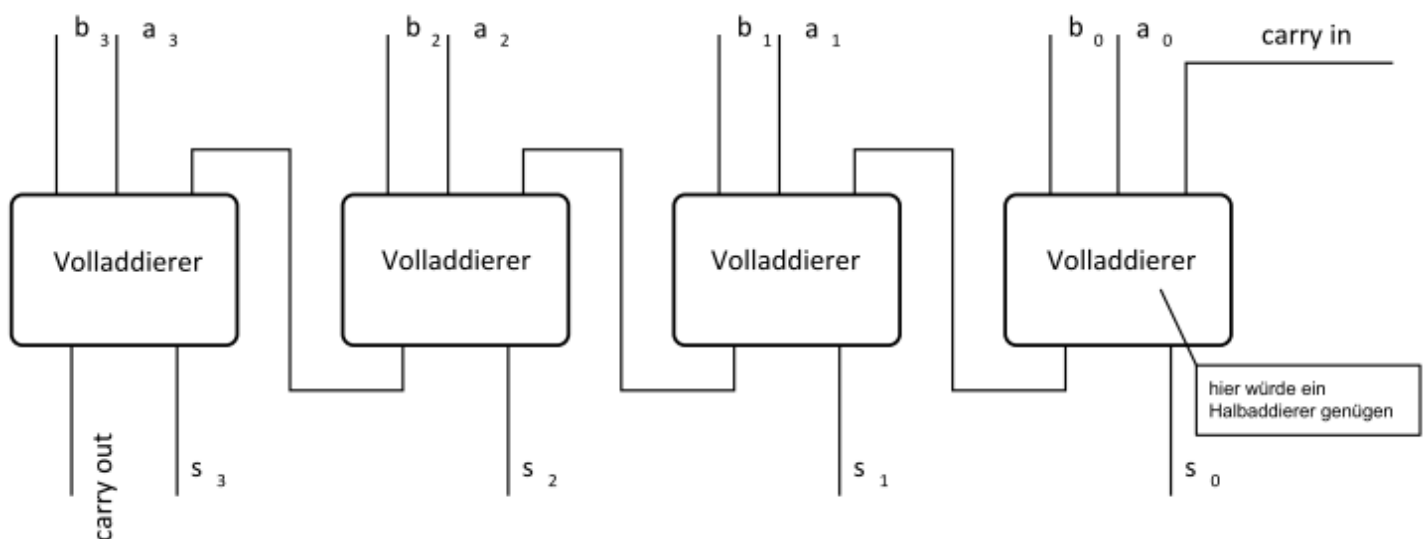


9.2.1.2 1 Bit Volladdierer:

A	B	Carry In	Sum	Carry Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



9.2.1.3 Ripple Carry Adder (4 Bit)



Das niederwertige Carry-Bit wird immer an die nächst höhere Position weitergereicht. Dadurch dauert es eine gewisse Zeit, bis das MSB korrekt ist. Für hohe Wortbreiten zu langsam.

9.2.1.4 Carry Select Adder

- Obere Worthälfte wird zweimal berechnet
 - einmal für Carry-In = 0
 - einmal für Carry-In = 1
- Die untere Worthälfte und die beiden oberen Worthälften werden gleichzeitig berechnet.
- Das richtige Resultat der oberen Worthälfte wird ausgewählt sobald das Carry der unteren Worthälfte feststeht.
- Bei einem n Bit breiten Adder wird idealerweise in $\sqrt{n}$ breite Blöcke aufgeteilt

- Die Ausführungszeit steigt dann nicht linear, sondern nur mit $\sqrt{n}$
- Kosten: Erhöhter Hardwareaufwand

9.2.1.5 Carry Look-Ahead Adder

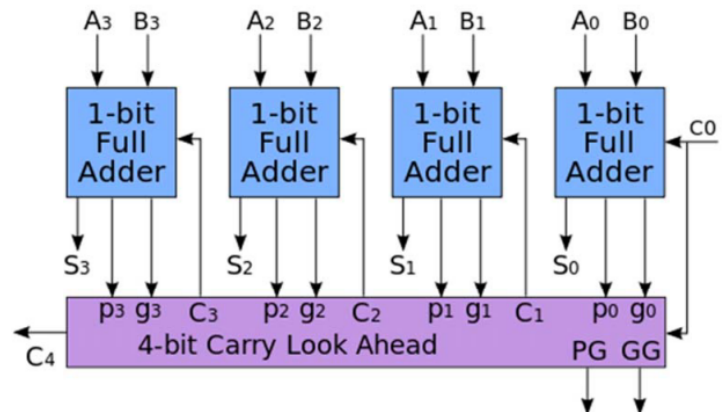
- Carry muss bei jeder Bitposition i entweder fortgepflanzt (propagate, p_i) oder generiert (generate, g_i) werden.
- p_i und g_i sind nur von den Eingängen a_i und b_i abhängig, nicht aber von den Carries der vorherigen Stufen.

$$p_i = a_i \oplus b_i \quad (5)$$

$$g_i = a_i * b_i \quad (6)$$

$$c_{i+1} = g_i + p_i * c_i \quad (7)$$

- Jedes einzelne Carry kann somit als zweistufige Logik von verschiedenen p_i und g_i implementiert werden.



9.2.2 Subtrahierer

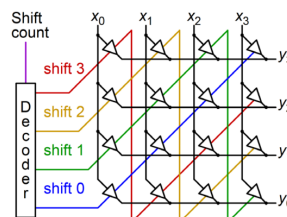
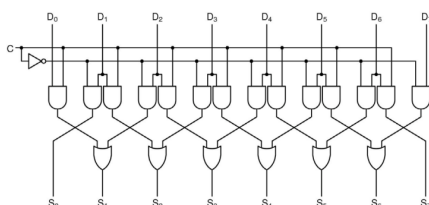
$$A - B = A + (-B) \quad (8)$$

- Subtraktion wird zurückgeführt auf Addition
- $(-B)$ kann gebildet werden, indem alle Bits von B invertiert werden und 1 addiert wird.
- 1 addieren erfolgt durch Anlegen einer '1' bei $Carry_0$
- Damit der Addierer mit minimalem Zusatzaufwand auch als Subtrahierer verwendet werden kann, wird deshalb auch an der Bitposition 0 ein Volladdierer eingesetzt

9.2.3 Schiebereinheit (Shifter)

- Ein Shifter kann mit einem Schieberegister implementiert werden
- Der Nachteil dabei ist, dass pro Clockzyklus nur um ein Bit nach links oder rechts geschoben werden kann. Zudem ist das Schieberegister eine sequentielle Schaltung und benötigt Flip Flops.
- Oft muss um eine beliebige Anzahl Stellen möglichst in einem Schritt geschoben werden
- Schieben kann auch mittels einer kombinatorischen Schaltung erreicht werden
- In der Praxis wird meist ein Barrel Shifter eingesetzt. Dieser kann innerhalb eines Zyklus um eine beliebige Anzahl Stellen schieben. Der Barrel Shifter ist rein kombinatorisch.

9.2.3.1 Kombinatorischer 8 Bit Shifter nach links (C=0) bzw. rechts (C=1)



9.2.4 Sättigungsarithmetik

- Besonders bei DSPs wird die Sättigungsarithmetik als Option eingesetzt
- Wenn der maximal darstellbare Wert inkrementiert werden soll, ergibt das nicht einen Overflow sondern der Wert bleibt auf seinem Maximalwert
- Analog wird der minimale Wert behandelt: bei signedTypen verbleibt der minimale negative Wert, bei unsignedTypen der Wert Null

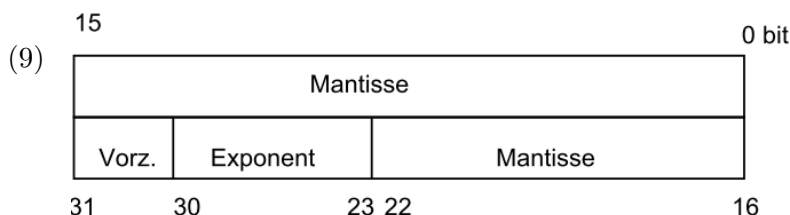
9.3 Fixpunkt (Fixed Point)

- Die Hardware für Fixed Point Arithmetic ist einfacher als die Floating Point Arithmetic, d.h. sie kann einfacher implementiert werden
- Wie der Name sagt, sitzt der Dezimalpunkt an einer fixen Stelle. Die Stellen rechts vom Dezimalpunkt werden als negative Zweierpotenzen interpretiert
- Für Fixed Point-Berechnungen kann die Integereinheit verwendet werden
- Eigentlich erhält das LSB einfach nicht mehr den Wert 1, sondern beispielsweise den Wert 2^{-6} , d.h. $\frac{1}{64} = 0.015625$
- Davon ausgehend kann mit der Integereinheit ganz normal gerechnet werden. Fixed Point Berechnungen sind deshalb sehr effizient

9.4 Gleitkomma (Floating Point)

$$(-1)^S \cdot M \cdot B^E$$

S = Vorzeichen (float: 1 bit double: 1 bit)
M = Mantisse (float: 23 bit double: 52 bit)
B = Basis
E = Exponent (float: 8 bit double: 11 bit)



9.4.1 Floating Point Unit (FPU)

- Eine FPU ist eine Hardwareimplementation für die Berechnung von Floating Point Zahlen
- Eine FPU-Implementation rechnet die FP-Operationen mindestens eine Grössenordnung schneller als Softwareimplementationen
- Nebst den Grundrechenarten rechnet die FPU auch Wurzeln, teilweise auch trigonometrische Funktionen, etc.

9.4.2 Addition

1. Die beiden Exponenten subtrahieren, um festzustellen, welcher der grössere ist. Der grössere Exponent wird als resultierender Exponent verwendet. Die Mantisse der kleineren Zahl muss zudem vor der Addition um diesen Differenzbetrag verschoben werden.
2. Die Mantisse der kleineren Zahl verschieben und zur grösseren addieren.
3. Die Mantisse wieder normalisieren, d.h. für die Mantisse M muss immer gelten: $1 \leq M \leq 2$. Dies bedingt unter Umständen auch eine Anpassung des Exponenten.

Für einen Floating Point Adder werden mindestens folgende Elemente benötigt: Integer-Addierer/Subtrahierer, Komparatoren und Schiebereinheiten (Shifter)

9.4.3 Subtraktion

Der Floating Point Subtrahierer kann analog zum Addierer implementiert werden.

9.4.4 Multiplikation

$$m_1 \cdot B^{e_1} \cdot m_2 \cdot B^{e_2} = m_1 \cdot m_2 \cdot B^{e_1+e_2} \quad (10)$$

1. Die beiden Exponenten müssen addiert werden
2. Die beiden Mantissen müssen multipliziert werden
3. Die Mantisse wieder normalisieren, d.h. für die Mantisse M muss immer gelten: $1 \leq M \leq 2$. Dies bedingt unter Umständen auch eine Anpassung des Exponenten.

Für einen Floating Point Multiplier werden mindestens folgende Elemente benötigt: Integer-Addierer/Subtrahierer, Integer-Multiplizierer und Schiebereinheiten (Shifter)

9.4.5 Division

$$\frac{m_1 \cdot B^{e_1}}{m_2 \cdot B^{e_2}} = \frac{m_1}{m_2} \cdot B^{e_1-e_2} \quad (11)$$

1. Die beiden Exponenten müssen subtrahiert werden
2. Die beiden Mantissen müssen dividiert werden
3. Die Mantisse wieder normalisieren, d.h. für die Mantisse M muss immer gelten: $1 \leq M \leq 2$. Dies bedingt unter Umständen auch eine Anpassung des Exponenten.

Für einen Floating Point Divider werden mindestens folgende Elemente benötigt: Integer-Addierer/Subtrahierer, Integer-Dividierer und Schiebereinheiten (Shifter)

Die Floating Point Division ist relativ selten (ca. 5% der Grundrechenarten im Normalfall). Aus diesem Grund wird oft kein Hardware-Dividierer angeboten. Häufig wird eine Multiplikation mit der Reziproken gerechnet. Die Reziproke kann nur mit Additionen und Multiplikationen gerechnet werden

$$\frac{x}{y} = \frac{1}{y} \cdot x \quad (12)$$

9.4.6 Berechnung von beliebigen Kurvenverläufen

- Der Verlauf von beliebigen Kurven wird häufig in Lookup Tables gespeichert
- Für Werte zwischen zwei Stützwerten wird interpoliert, normalerweise linear
- Die Genauigkeit nimmt offensichtlich zu, je mehr und je genauere Stützwerte gespeichert werden
- Denkbar sind auch Approximationen, wobei nur mit Additionen, Subtraktionen und Multiplikationen gearbeitet werden darf

9.4.7 Newton-Raphson

Um eine unäre Funktion wie $\sin(x)$ zu berechnen, kann man die Taylorreihe verwenden $\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} \dots$. Die Koeffizienten könnten gespeichert werden, jedoch ist die Konvergenz trotz dieser Vereinfachung langsam. Eine Methode, welche schneller konvergiert ist das Newton-Raphson Verfahren:

$$y_{k+1} = y_k - \frac{f(y_k)}{\frac{\partial}{\partial y} f(y_k)} \quad (13)$$

Zur Berechnung einer Reziproken $g(x) = \frac{1}{x} = y$ ergibt sich folgende Formel:

$$y_{k+1} = y_k \cdot (2 - x \cdot y_k) \quad (14)$$

Wobei x die Zahl aus dem Reziprokwert ist und y_k ein vernünftiger Anfangswert ist.

Somit kann auch die Quadratwurzel berechnet werden und zwar via: $\frac{1}{\sqrt{x}} \cdot x = \sqrt{x}$

Wobei der Bruch $\frac{1}{\sqrt{x}}$ wie folgt berechnet werden kann:

$$y_{k+1} = 0.5y_k \cdot (3 - x \cdot y_k^2) \quad (15)$$

Das Newton Verfahren konvergiert quadratisch, d.h mit jedem Durchlauf kann die Genauigkeit verdoppelt werden.

9.4.7.1 Herleitung $\frac{1}{x}$

$$g(x) = \frac{1}{x} \equiv y \rightarrow x = \frac{1}{y} \quad (16)$$

$$f(y) \equiv \frac{1}{y} - x = 0 \quad (17)$$

$$\frac{\partial}{\partial y} f(y) = -y^{-2} = -\frac{1}{y^2} \quad (18)$$

$$y_{k+1} = y_k - \frac{\frac{1}{y_k} - x}{-\frac{1}{y_k^2}} \quad (19)$$

$$y_{k+1} = y_k \cdot (2 - x \cdot y_k) \quad (20)$$

9.4.7.2 Herleitung $\frac{1}{\sqrt{x}}$

$$g(x) = \frac{1}{\sqrt{x}} \equiv y \rightarrow x = \frac{1}{y^2} \quad (21)$$

$$f(y) \equiv \frac{1}{y^2} - x = 0 \quad (22)$$

$$\frac{\partial}{\partial y} f(y) = \frac{-2}{y^3} \quad (23)$$

$$y_{k+1} = y_k - \frac{\frac{1}{y_k^2} - x}{\frac{-2}{y_k^3}} \quad (24)$$

$$y_{k+1} = 0.5 \cdot y_k \cdot (3 - x \cdot y_k^2) \quad (25)$$

10 Plattformen

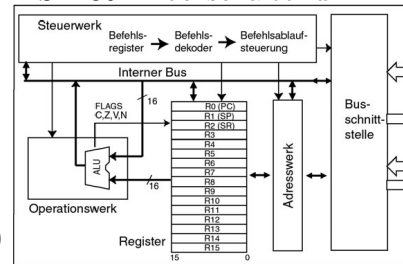
10.1 Auswahl der Zielplattform

- was muss gerechnet werden?
- wie schnell?
- wie viel Adressraum?

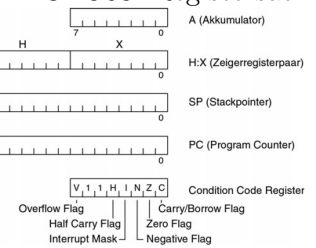
Erst wenn das feststeht → Peripherie miteinbeziehen (Grund: zu jeder CPU gibt es meist unzählige Varianten der Peripherie!)

10.2 Vergleich der Beispielarchitekturen

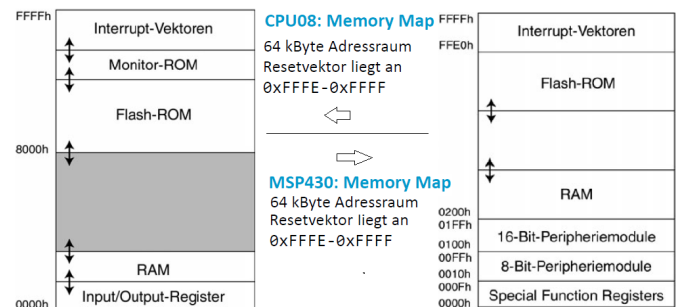
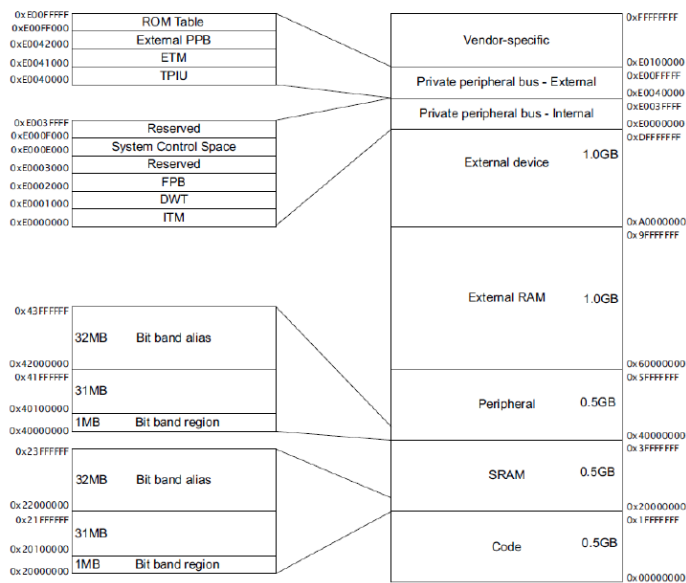
MSP430: Blockschaltbild



CPU08: Registersatz



ARM Cortex-M3 : Memory Map



	CPU08	MSP430	Cortex-M3
Verarbeitungsbreite	8 Bit	16 Bit	32 Bit
Register	5	16	16
Befehle	91	27	> 100
Adressraum	64 kB	64 kB	4 GB
Adressierungsarten	16	7	4+ Varianten
Arbeitstakt	4-8 MHz	8-25 MHz	20 - 190 MHz
Taktzyklen pro Befehl	1-9	1-6	1 - 7
Opcode-Länge	1-2 Byte	2 Byte	2-4 Byte
Multiplikation	MUL-Befehl	Software/Hardware	Hardware
Division	DIV-Befehl	Software-Emulation	Hardware
Gleitkommaverarbeitung	Nein	Nein	Ja
Interrupt-Prioritäten	128 (fest)	16 (fest)	240 (programmierbar)
Schutzmechanismen	-	-	MPU, Privilege Levels

10.3 Digital Signal Processor (DSP)

10.3.1 DSP Eigenschaften

- Mikrocontroller, der speziell für Digital Signal Processing geeignet ist
- Für Faltung oder FFT müssen (Abtast-)Werte mit einem Gewicht multipliziert und zu einem bestehenden Wert addiert werden
- Diese Multiply-Add-Folge wird häufig als Multiply-Accumulate (MAC) bezeichnet
- Viele Daten, immer wieder dieselben Operationen
- meist Harvard: gleichzeitig Daten und Instruktionen aus dem Speicher holen
- Pipelining, DMA und FPU

10.3.2 TI fixed Point DSP (C28x)

- Low-cost 32-bit fixed-point processor
- An 8-phase protected pipeline prevents a write to and a read from the same location from occurring out of order
- 32 bit Arithmetic logic unit (ALU)
- Address register arithmetic unit (**ARAU**) generates data- memory addresses and increments or decrements pointers in parallel with ALU operations
- 32 bit multiplier performs 32 x 32 bit 2s-complement multiplication with a 64 bit result
- 4Gwords (1word = 16bit) data space, 4Mwords program space

10.3.3 C28x Instruction Pipeline

- Each instruction passes through eight independent phases that form an instruction (Instr) pipeline
- At any given time, up to eight instructions may be active, each in a different phase of completion
- pipeline-protection mechanism ensures that reads & writes to the same location happen in the order in which they are programmed
- To maximize pipeline efficiency, an instruction-fetch mechanism attempts to keep the pipeline full:
 - Its role is to fill an instruction-fetch queue
 - Instr-fetch mechanism: 1 32bit or 2 16bit Instr
 - uses 3 program-address counters: program counter (PC), the instruction counter (IC), and the fetch counter (FC)
- Branches are taken in D2: - Content in F1-D1 could then be wrong - This requires a flush of F1-D1 - Pipeline flushes should be avoided → machen das System langsam! (z.B. if-Abfrage in einer for-Schleife)

10.3.6 C28x Pipeline Beispiele

Example: Diagraming Pipeline Activity

Address Opcode Instruction
 00 0040 F345 I1: MOV DP,#VarA ; DP = page that has VarA
 00 0041 F346 I2: MOV AL,@VarA ; Move content of VarA to AL
 00 0042 F347 I3: MOV B,AR0,#VarB ; AR0 points to VarB
 00 0043 F348 I4: ADD AL,*XAR0++ ; Add content of VarB to AL, add 1 to XAR0
 00 0044 F349 I5: MOV @VarC,AL ; Replace content of VarC with cont. of AL
 00 0045 F34A I6: ADD AL,*XAR0++ ; Add cont. of (VarB + 1) to AL, add 1 to XAR0
 00 0046 F34B I7: MOV @VarD,AL ; Replace content of VarD with content of AL
 00 0047 F34C I8: ADD AL,*XAR0 ; Add content of (VarB + 2) to AL

Phase	Activity Shown
F1	Drive address 00 0042 ₁₆ on the program address bus (PAB).
F2	Read the opcodes F347 and F348 from addresses 00 0042 ₁₆ and 00 0043 ₁₆ , respectively.
D1	Isolate F348 in the instruction-fetch queue.
D2	Use XAR0 = 0066 ₁₆ to generate source address 0000 0066 ₁₆ and then increment XAR0 to 0067 ₁₆ .
R1	Drive address 00 0066 ₁₆ on the data-read data bus (DRDB).
R2	Read the data value 1 from address 0000 0066 ₁₆ .
E	Add 1 to content of AL (1230 ₁₆) and store result (1231 ₁₆) to AL.
W	No activity

10.3.4 C28x Instruction Pipeline Phases

- Instruction fetch (F1 and F2) loads instr from program memory to instr-fetch queue
- Instruction decode (D1 and D2) requests an instr from the instr-fetch queue, decodes the instr, loads it into instr register, branches are taken in D2 if required (ARAU)
- Read phase (R1 and R2) reads data from memory if data has to be read
- Execute (E) CPU performs all multiplier, shifter and ALU ops
- Write (W) writes data to memory if required

10.3.5 C28x Instruction Pipeline Protection

- unprotected pipeline und read&write an selbe Stelle → Fehler (falsche Reihenfolge)
- CPU auto adds inactive cycles (NOP) to ensure that reads&writes happen as intended
- Zwei Typen von Fehlern: 1. read&write to same data-space location 2. Register conflicts

Example: Visualizing Pipeline Activity for I4: ADD AL,*XAR0++

F1	F2	D1	Instruction	D2	R1	R2	E	W
00 0040								
		F345						
	F348:F347	F346	I1: MOV DP,#VarA	DP = 8				
00 0044		F347	I2: MOV AL,@VarA	Generate VarA address	-			
	F34A:F349	F348	I3: MOV B,AR0,#VarB	XAR0 = 66	00 0203	-		
00 0046		F349	I4: ADD AL,*XAR0++	XAR0 = 67	-	1230	-	
	F34C:F34B	F34A	I5: MOV @VarC,AL	Generate VarC address	00 0066	-	AL = 1230	-
		F34B	I6: ADD AL,*XAR0++	XAR0 = 68	-	0001	-	-
		F34C	I7: MOV @VarD,AL	Generate VarD address	00 0067	-	AL = 1231	-
			I8: ADD AL,*XAR0	XAR0 = 68	-	0003	-	-
					00 0068	-	AL = 1234	00 0204
						0005	-	1231
							AL = 1239	00 0205
								1234

C-Code / Assembler-Code und Darstellung der Instruction Pipeline

```

; Y = sqrt(X)
; Ye = Estimate(1/sqrt(X));
; Ye = Ye*(1.5 - Ye*Ye*X*0.5)
; Ye = Ye*(1.5 - Ye*Ye*X*0.5)
; Y = X*Ye
_sqrt:
    EISQRTF32    R1H, R0H          ; R0H = X on entry
    MPYF32       R2H, R0H, #0.5    ; R2H = Ye = Estimate(1/sqrt(X))
    MPYF32       R3H, R1H, R1H     ; R3H = Ye*Ye
    NOP
    MPYF32       R3H, R3H, R2H     ; R3H = Ye*Ye*X*0.5
    NOP
    SUBF32       R3H, #1.5, R3H    ; R3H = 1.5 - Ye*Ye*X*0.5
    NOP
    MPYF32       R1H, R1H, R3H     ; R2H = Ye = Ye*(1.5 - Ye*Ye*X*0.5)
    NOP
    MPYF32       R3H, R1H, R2H     ; R3H = Ye*X*0.5
    NOP
    MPYF32       R3H, R1H, R3H     ; R3H = Ye*Ye*X*0.5
    NOP
    SUBF32       R3H, #1.5, R3H    ; R3H = 1.5 - Ye*Ye*X*0.5
    CMPF32       R0H, #0.0        ; Check if X == 0
    MPYF32       R1H, R1H, R3H     ; R2H = Ye = Ye*(1.5 - Ye*Ye*X*0.5)
    NOP
    MOV32        R1H, R0H, EQ      ; If X is zero, change the Ye estimate to 0
    MPYF32       R0H, R0H, R1H    ; R0H = Y = X*Ye = sqrt(X)
    LRETR
  
```

F1	F2	D1	D2	R1/D	R2/R	E/E1	W/E2	Cycle
I1								1
I2	I1							2
I3	I2	I1						3
NOP	I3	I2	I1					4
I4	I3	I2	I1					5
NOP	I4	I3	I2	I1				6
I5	I4	I3	I2	I1				7
NOP	I5	I4	I3	I2	I1			8
I6	I5	I4	I3	I2	I1	I1/I2		9
NOP	I6	I5	I4	I3	I2	I2/I3	I10	10
I7	I6	I5	I4	I3	I2	I3	I11	11
NOP	I7	I6	I5	I4	I3	I4	I12	12
I8	I7	I6	I5	I4	I3	I5	I13	13
NOP	I8	I7	I6	I5	I4	I6	I14	14
I9	I8	I7	I6	I5	I4	I7	I15	15
I10	I9	I8	I7	I6	I5	I8	I16	16
I11	I10	I9	I8	I7	I6	I9	I17	17
NOP	I11	I10	I9	I8	I7	I10	I18	18
I12	I11	I10	I9	I8	I7	I11	I19	19
I13	I12	I11	I10	I9	I8	I12	I20	20
I14	I13	I12	I11	I10	I9	I13	I21	21
	I14	I13	I12	I11	I10	I14	I22	22
		I14	I13	I12	I11	I15	I23	23
			I14	I13	I12	I16	I24	24
				I14	I13	I17	I25	25
					I14	I18	I26	26
						I19	I27	27
							I28	28

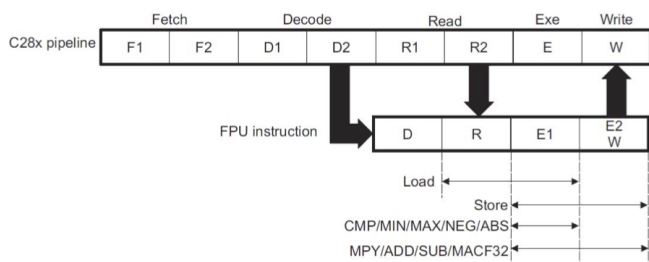
Gründe für Freeze in Pipeline Activity:

Wait states: Lesen von oder Schreiben ins Memory dauert länger als 1 stage

Activity in the pipeline freezes depending on where the wait state occurred (F1, R1, W)

Instr not available: if instr is not available in D2 phase.

10.3.7 C28x FPU Pipeline

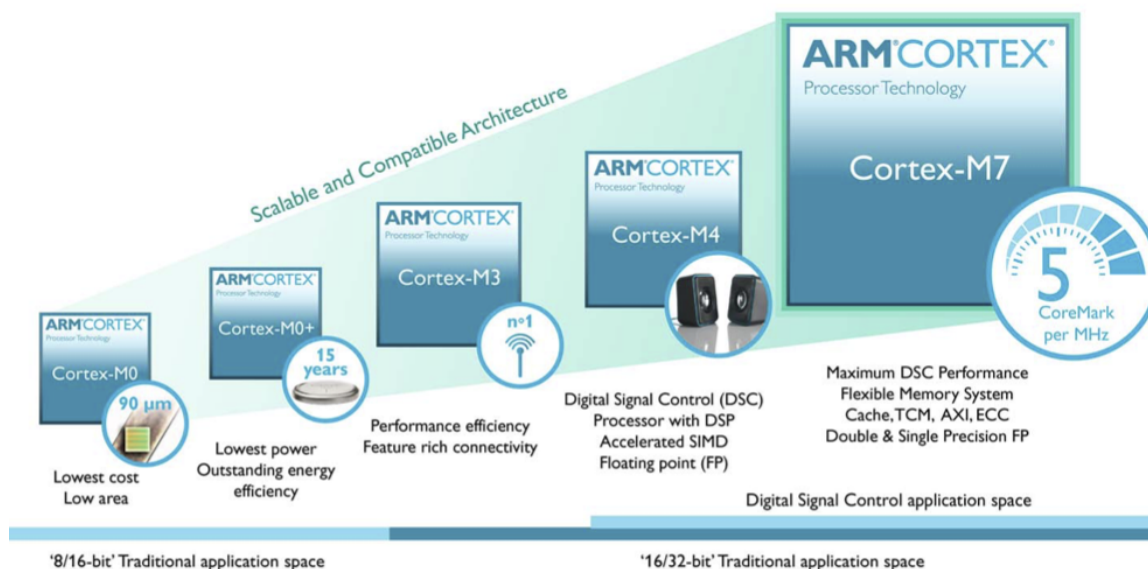


10.4 ARM-Cortex

10.4.1 ARM-Cortex Family

- ARMv8-A: Application profile: High-performance (Smartphones, Tablets) Cortex-A72, -A57, -A53
- ARMv8-R: Real-time profile: Embedded applications in automotive and industrial control: Cortex-R4, -R5, -R7
- ARMv8-M: Microcontroller profile: Embedded applications, optimized for energy efficiency and cost Cortex-M0, -M1, -M3, -M4, -M7

10.4.2 Cortex-M family



ARM Cortex-M instruction sets ^{[6][7]}									
ARM Cortex-M	Thumb	Thumb-2	Hardware multiply	Hardware divide	Saturated math	DSP extensions	Floating-point	ARM architecture	Core architecture
Cortex-M0 ^[1]	Most	Subset	1 or 32 cycle	No	No	No	No	ARMv6-M ^[6]	Von Neumann
Cortex-M0+ ^[2]	Most	Subset	1 or 32 cycle	No	No	No	No	ARMv6-M ^[6]	Von Neumann
Cortex-M1 ^[3]	Most	Subset	3 or 33 cycle	No	No	No	No	ARMv6-M ^[6]	Von Neumann
Cortex-M3 ^[4]	Entire	Entire	1 cycle	Yes	Yes	No	No	ARMv7-M ^[7]	Harvard
Cortex-M4 ^[5]	Entire	Entire	1 cycle	Yes	Yes	Yes	Optional	ARMv7E-M ^[7]	Harvard

Befehle für den Cortex-M4 mit FPU (M4F) beginnen immer mit "V" z.B. VADD, VMUL, VDIV, VSQRT;

10.5 MSP432 und Vergleich

- Successor of MSP430
- No proprietary core anymore but ARM Cortex-M4F
- Ultra-low-power (90 μ A/MHz, static 25 nA)
- Wide configuration of device options is planned

Criterion	MSP432	MSP430	TI M4F
Core	Cortex-M4F	MSP430	Cortex-M4F
Architecture	Harvard	Von Neumann	Harvard
Bits	32	16	32
Power [μ A/MHz]	95	100	370
Low-power mode [nA]	25	20	1600
Max. clock freq. [MHz]	48	16	80
Floating Point Unit	yes	no	yes
Price [CHF/1ku]	3.58	2.40	5.44

11 Multicore

11.1 Steigern der Geschwindigkeit eines Prozessors

Die Geschwindigkeit eines Prozessors kann mit den folgenden Methoden gesteigert werden.

11.1.1 Clockfrequenz erhöhen

- PCB-Design wird sehr anspruchsvoll (Leitungslängen, Reflexionen, etc.)
- Elektrische Verlustleistung steigt linear mit der Clockfrequenz (Hitzeproblematik)

$$P = f_{cl} \cdot C_L \cdot V_{dd}^2 \quad (26)$$

- Lichtgeschwindigkeit ist die Grenze.

11.1.2 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

11.1.3 Thread-level parallelism (TLP)

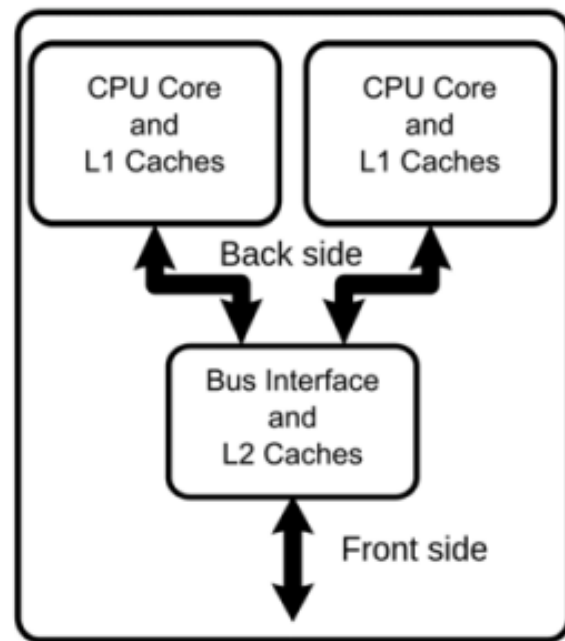
- Parallelisiert wird auf einer umfangreicheren Stufe als nur auf der Instruktion
- Die parallelisierte Einheit ist in der Größenordnung einer Funktion
- Zugriffe auf gemeinsame Ressourcen, z.B. shared-memory, müssen geregelt werden
- Design Issues
 - Nicht einfach beliebig viele Threads spendieren
 - Jeder Thread erzeugt Overhead
 - Die Threads sollten möglichst unabhängig voneinander sein
 - Wenn zu viele Threads definiert werden (wenn Zusammengehörendes auseinandergenommen wird), dann werden Daten unnötigerweise shared

11.1.3.1 Umsetzung

- Uniprocessor(single-core)
 - Die Threads werden time-sliced
 - Pseudo-TLP
 - Context switch notwendig
- Multiprocessor(Computer mit mehreren Prozessoren)
 - Mehrere parallele Prozessoren können je einen Thread bearbeiten
 - Echter TLP
 - Clockfrequenz kann tiefer gehalten werden
 - Einfachere Hardware wird multipliziert
 - Datenaustausch zwischen Prozessoren muss irgendwie geregelt werden, z.B. mit Message Passing oder SharedMemory

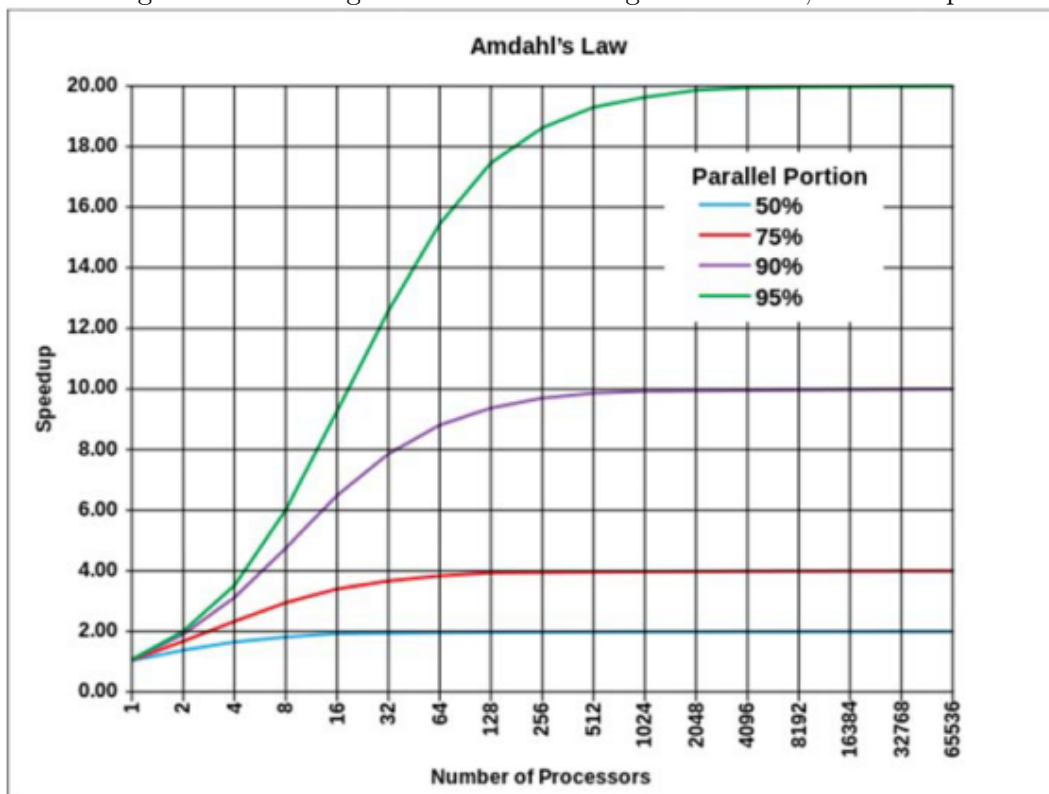
11.2 Multicore Prozessor

- Ein Multicore Prozessor ist ein spezieller Multiprocessor. Alle Prozessoren (Cores) befinden sich auf demselben Chip
- Multicore Prozessoren sind MIMD (Multiple Instructions Multiple Data). Jeder Core führt einen eigenen Code aus und arbeitet auf unterschiedlichen Daten
- Die Speicherorganisation ist oft komplex
- Zwei Typen
 - Homogener Multicore Prozessor (mehrere gleiche Cores)
 - Heterogener Multicore Prozessor (mehrere gleiche/unterschiedliche Cores). Häufig bei Embedded Multicore Prozessoren



11.2.1 Amdahl's Law

Gesamtaufgabe soll auf möglichst viele Cores aufgeteilt werden, um eine optimale Ausführungszeit zu erzielen.



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

$$S \leq \frac{1}{1 - p}$$

S : Speedup
 p : Parallel Portion
 n : Number of Processors

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

$$S \leq \frac{T}{T - t_p} = \frac{T}{t_s}$$

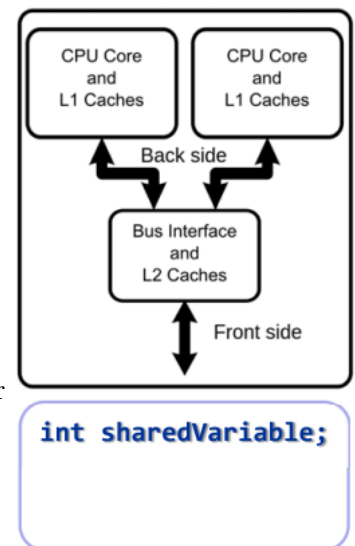
T : Total execution time
 t_p : Parallel execution time
 t_s : Serial execution time

11.2.2 Speicherorganisation

- Shared Memory
 - Alle Prozessoren nutzen einen gemeinsamen Speicherbereich
 - Einfache Implementation
 - Shared Memory kann zum Flaschenhals werden
- Distributed Memory
 - Jeder Prozessor hat seinen eigenen lokalen Speicher
 - Für den Datenaustausch unter den Prozessoren muss wiederum ein Mechanismus zur Verfügung gestellt werden, z.B. Message Passing

11.2.2.1 Programmierung von Shared Memory

- Die beiden Cores haben eigene Adressbereiche
- Die Variable `sharedVariable` liegt auf einer festen Adresse im Shared Memory
- `sharedVariable` wird von beiden Cores genutzt
- Der Zugriff auf `sharedVariable` muss synchronisiert werden
- Zwischen den CPU Cores und `sharedVariable` können mehrere Caches liegen
- Jeder Core muss für sich diese Variable definieren
- Bei `int sharedVariable;` darf der Compiler eine Kopie in einem Arbeitsregister anlegen. Unbedingt verhindern!
- Die Definition muss zwingend mit `volatile` versehen werden: `volatile int sharedVariable;`



11.2.2.2 Kennzeichnung von Variablen mit `volatile`

- Alle Variablen, die ausserhalb des Programmkontextes des Prozessors geändert werden können müssen gekennzeichnet werden.
- Dies sind:
 - Variablen, die speziell bei Embedded Systems ein Hardwareregister darstellen
 - Globale Variablen, auf die in mehreren Tasks (concurrency, gleichzeitige Ausführungseinheiten) zugegriffen wird
 - Globale Variablen, auf die in Interruptserviceroutinen (ISR) zugegriffen wird
- Es muss verhindert werden, dass irgendwann mit einer nicht aktuellen Kopie in einem Arbeitsregister gearbeitet wird

11.3 Caches

- Caches sind (für den Prozessor) versteckte schnelle Speicher, die zwischen dem Prozessor und dem Hauptspeicher liegen
- Caches führen eine Kopie von häufig benötigten Hauptspeicherdaten
- Im besten Fall sind die Hauptspeicherdaten, auf welche der Prozessor zugreifen will, bereits im Cache vorhanden. Dann ist der Speicherzugriff schneller als wenn wirklich auf den Hauptspeicher zugegriffen werden müsste
- Bei Multicore Systemen können mehrere Cores je Caches auf dieselben Daten haben, d.h. es kann mehrere Kopien derselben Daten geben

- Cache Coherency: Wie erfährt ein Cache eines Cores, dass ein anderer Core diese Daten geändert hat? `volatile` löst das Cache Coherency Problem nicht. Damit wird einzig sichergestellt, dass der Core einen Zugriff Richtung Speicher macht und sich nicht eine Kopie in einem Arbeitsregister hält

11.3.1 Speicherzugriff zur Laufzeit

- `volatile` hat Einfluss zur Compile-time und stellt sicher, dass der Compiler keinen Code generiert, bei dem er sich lokale Kopien in einem Register hält. Dank `volatile` besteht der Code nun also aus einem expliziten Speicherzugriff
- Dieser Speicherzugriff sollte so schnell wie möglich sein. Hoffnung: die Daten werden in einem möglichst prozessornahen Cache gefunden
- Beim ersten Zugriff auf diese Daten befinden sie sich logischerweise noch nicht im Cache. Ein Geschwindigkeitsgewinn ergibt sich erst ab dem zweiten Zugriff
- Triviale Folgerung: Daten, auf die nur einmal zugegriffen werden muss, sollen nicht gecached werden

11.3.2 Lokalität

- Zeitliche Lokalität (Temporal locality)
 - Soeben verwendete Daten oder Instruktionen werden mit hoher Wahrscheinlichkeit bald wieder verwendet
 - Caches nutzen die zeitliche Lokalität aus
- Räumliche Lokalität (Spatial locality)
 - Bei soeben verwendeten Daten oder Instruktionen werden mit hoher Wahrscheinlichkeit auch benachbarte Daten oder Instruktionen bald verwendet
 - Die räumliche Lokalität wird in Caches folgendermassen umgesetzt: 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

11.4 Ping Pong Speicher

Szenario: Ein System empfängt Daten und schreibt diese in einen Buffer. Die Daten werden gefiltert und wieder ausgegeben. Wie kann bei Multicore Systemen verhindert werden, dass der Filterschritt mit inkonsistenten Daten arbeitet?

Schlechte Lösung: Buffer wird synchronisiert, oder der Empfangsbuffer wird in einen zweiten Buffer kopiert

Gute Lösung: Ping Pong Buffer

11.4.1 Funktion

- Zwei identische Buffer werden definiert
- Ein Core bestimmt, in welchen Buffer die Daten geschrieben werden
- Nach abgeschlossenem Empfang übergibt der Core dem Filterschritt einen Pointer auf diesen Buffer zur weiteren Bearbeitung
- Dem Empfänger übergibt der Core einen Pointer auf den zweiten Buffer, damit nun in diesen geschrieben wird
- Daten werden nicht kopiert, nur Pointer werden von einem Buffer zum anderen hin-und hergewechselt (Ping Pong)

12 Concurrency

12.1 Motivation

12.1.1 Wieso wird Concurrency verwendet?

- Praktische Programme führen meist mehrere Arbeiten „gleichzeitig“ aus.

12.1.2 Parallelität vs. Concurrent Computing

- Parallel Computing
 - execution of different tasks is really at the same time
 - is not possible on a single-core machine
- Concurrent Computing
 - execution of different tasks only seems to be at the same time
 - different tasks get consecutive time slices
 - there's only one task really running at a particular time slice
 - can be done on a single- or multicore machine

12.1.3 Gründe, Concurrency nicht zu verwenden

- Concurrency (mit Prozessen, Tasks, Threads) kostet immer
 - Stack
 - Context switch (Umschalten vom einen zum anderen)
 - * dauert
 - * Alter Context muss gespeichert, neuer wieder geladen werden
 - Zugriff auf gemeinsame Ressourcen muss synchronisiert werden
 - * kostet
 - * fehleranfällig (wird vergessen oder falsch gemacht)
- Komplexität steigt
 - Sequentielle Programme sind einfacher zu verstehen als parallele
 - Ziel ist immer, ein System möglichst einfach zu halten

Folgerung: Concurrency nur dann einsetzen, wenn wirklich ein Nutzen vorhanden ist!

12.2 POSIX Threads Programming

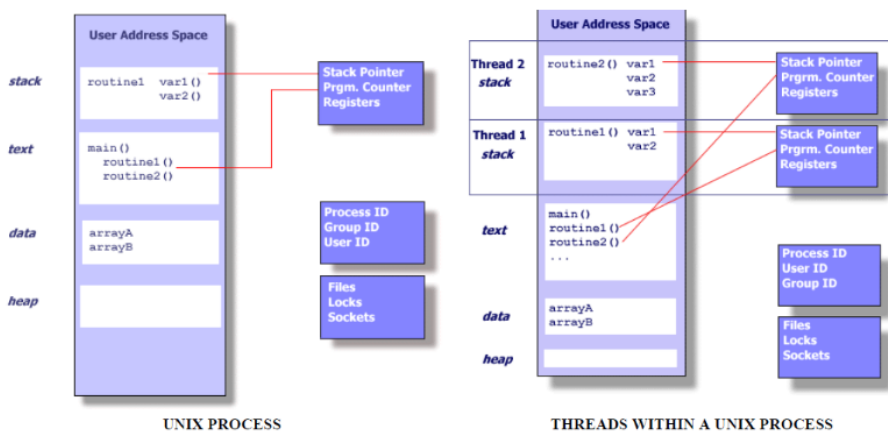
12.2.1 UNIX Process

- Heavyweight process (created by the operating system)
- Processes require a fair amount of overhead; they contain information about program resources and program execution state, including: Process ID, process group ID, user ID, and group ID; Environment; Program instructions; Registers; Stack; Heap; File descriptors; Signal actions; Shared libraries; Inter-process communication tools

12.2.2 UNIX Thread

- Lightweight "process"
- Threads use and exist within the process resources
- A thread uses the same address space as other threads of the same process
- Threads are able to be scheduled by the operating system
- Independent stream of instructions that may run simultaneously to other streams of instructions
- Procedure that runs independently from its main program
- A thread maintains its own: Stack pointer; Registers; Scheduling properties; Set of pending and blocked signals; Thread specific data
- Concurrent programs are usually achieved with threads

12.2.3 Process vs. Thread in UNIX



Man beachte: Alles was ein Thread inne hat, hat ein Prozess auch inne (aber NICHT umgekehrt)!

12.2.4 What are POSIX Threads?

- For UNIX systems, a standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard.
- The short form of POSIX threads is Pthreads or pthreads.

12.2.5 The pthreads API

- Routines of the pthreads API start with `pthread_`.
- The header file `pthread.h` must be included
- The link command must include `-lpthread`

12.2.6 Starting and Terminating a Thread

- Any routine with the following interface may become a thread routine:

```
void* threadRoutine(void* arg);
```

- A thread is started with:

```
int pthread_create(pthread_t* thread, const pthread_attr_t* attr, void* (*startRoutine)(void*), void* arg);
```

- thread: pointer to a pthread_t instance
- attr: pointer to a pthread_attr_t structure, often 0 (default attributes)
- arg: a single argument that may be passed to startRoutine
- returns 0 on success

- A thread terminates in one of the following ways:

- it calls pthread_exit()
- it returns from startRoutine
- it is canceled with pthread_cancel()

Example - Starting a Thread

```

1  #include <pthread.h>
2
3  void* threadFunction(void* arg);
4
5  int main(void)
6  {
7      int ret;
8      pthread_t myT;
9      ret = pthread_create(&myT, 0, threadFunction, 0);
10     if (ret)
11     { // handle error
12         return -1;
13     }
14     while (1) // endless loop
15     {}
16     return 0;
17 }
18
19 void* threadFunction(void* arg)
20 { // implement this
21     return 0;
22 }
```

Example - Waiting on a Thread to Finish

```

1  #include <pthread.h>
2
3  void* threadFunction(void* arg);
4
5  int main(void)
6  {
7      int ret;
8      pthread_t myT;
9      ret = pthread_create(&myT, 0, threadFunction, 0);
10     if (ret)
11     { // handle error
12         return -1;
13     }
14     ret = pthread_join(myT, 0);
15     if (ret)
16     { // handle error
```

```

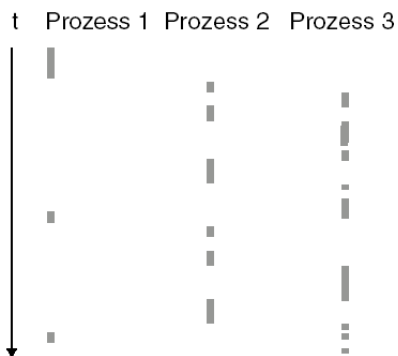
17     return -1;
18 }
19     return 0;
20 }
21
22 void* threadFunction(void* arg)
23 { // implement this
24     return 0;
25 }

```

12.2.7 Thread-Sicherheit (Thread-safeness)

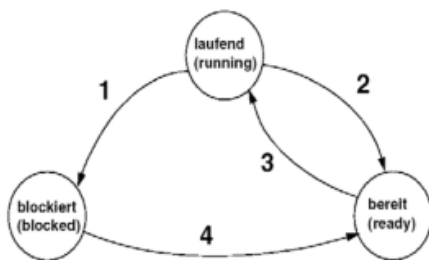
- Thread-Sicherheit besagt, dass eine Komponente gleichzeitig von verschiedenen Programmbereichen mehrfach ausgeführt werden kann, ohne dass diese sich gegenseitig behindern.
- Änderungen der einzelnen Threads müssen koordiniert werden, um einen chaotischen Zustand des Speichers zu verhindern, da das Programm dabei häufig gleichzeitig auf einen gemeinsamen Speicherbereich (Shared Memory) des Computers zugreifen will.
- Vorsicht: `rand()` ist nicht thread-safe (weil bei dieser Fkt. eine interne Variable verwendet wird, welche schon zum Voraus berechnet wurde); `rand_r()` hingegen schon.

12.2.8 Quasiparallelität



- Prozesse/Threads warten die meiste Zeit (blocked).
- Scheduler ordnet CPU denjenigen Prozessen/Threads zu, die im Zustand ready sind und „etwas zu tun haben“

Prozesse/Threads können folgende Zustände und Übergänge erfahren:



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

12.3 Synchronisation: Zugriff auf gemeinsame Ressourcen

12.3.1 Kritischer Abschnitt (Critical Section, CS)

- Codebereich, in dem nebenläufige Prozesse oder parallele auf gemeinsame Ressource zugreifen.
- Zu jeder Zeit darf sich höchstens ein Prozess im kritischen Bereich befinden.
- Der exklusive Zugriff durch höchstens einen Prozess wird mittels gegenseitigem Ausschluss (mutual exclusion, Mutex) sichergestellt.

12.3.2 Lösungsstruktur für gegenseitigen Ausschluss



- Bei der Zugriffsprüfung wird gewartet, bis der Zugang frei wird
- Beim Sperren wird das Signal auf Rot gesetzt (lock a mutex)
- Beim Freigeben wird das rote Signal wieder gelöscht (unlock a mutex)

12.3.3 Forderungen an die Synchronisation (Dijkstra, 1965)

1. Es dürfen sich nicht zwei Prozesse gleichzeitig in ihrem kritischen Abschnitt befinden (mutual exclusion).
2. Über die Abarbeitungsgeschwindigkeit, bzw. die Anzahl Prozesse dürfen keine Annahmen getroffen werden.
3. Kein Prozess darf ausserhalb eines kritischen Abschnitts einen anderen Prozess blockieren.
4. Jeder Prozess, der am Eingang eines kritischen Abschnitts wartet, muss irgendwann den Abschnitt betreten dürfen (fairness condition).

Synchro-Versuch 1

Prozess 1

```

1 while(grant != 1)
2     wait();
3 CS; //Eintritt in Critical Section
4 grant = 2;
  
```

Prozess 2

```

1 while(grant != 2)
2     wait();
3 CS; //Eintritt in Critical Section
4 grant = 1;
  
```

Forderung 2 nicht erfüllt, da sich alle Prozesse selbst daran hindern 2mal nacheinander daran zu kommen.

Synchro-Versuch 2

Prozess 1

```

1 while(in2)
2     wait();
3 in1 = true;
4 CS; //Eintritt in Critical Section
5 in1 = false;
  
```

Prozess 2

```

1 while(in1)
2     wait();
3 in2 = true;
4 CS; //Eintritt in Critical Section
5 in2 = false;
  
```

Initialisierung ist nicht gelöst.

Synchro-Versuch 3 (Algorithmus von Peterson)

Prozess 1

```

1 request1 = true;
2 grant = 1;
3 while(grant==1 && request2)
4     wait();
5 CS; //Eintritt in Critical Section
6 request1 = false;
  
```

Prozess 2

```

1 request2 = true;
2 grant = 2;
3 while(grant==2 && request1)
4     wait();
5 CS; //Eintritt in Critical Section
6 request2 = false;
  
```

Synchronisation mit Signalen

- Jeder Prozess wartet vor Betreten des kritischen Bereichs auf ein gemeinsames Signal.
- Wenn Signal gesetzt → kritischer Bereich frei
- `waitFor(signal)` blockiert aufrufenden Prozess, falls signal nicht gesetzt
- Jeder Prozess, der fertig ist, setzt Signal mit `send(signal)`
- Mehrere Prozesse können gleichzeitig warten

Prozess 1

```

1 ||      waitFor(signal);
2 ||      CS; //Eintritt in Critical Section
3 ||      send(signal);

```

Prozess 2

```

1 ||      waitFor(signal);
2 ||      CS; //Eintritt in Critical Section
3 ||      send(signal);

```

12.4 Semaphoren

- Das Signal für den Zutritt in den kritischen Bereich = Semaphor
- Ein Semaphor `s` hat zwei atomare (= nicht unterbrechbare) Operationen
 - `P(s)` : Passieren → Beim Eintritt in CS (`waitFor`)
 - `V(s)` : Verlassen → Beim Austritt aus CS (`send`)
- Busy waiting
 - Beim Busy Waiting warten die Prozesse aktiv in einer Schleife (spin lock)
 - Wartende Prozesse belasten so unnötigerweise den Prozessor
 - **Lösung:** Wartende Prozesse werden in eine Warteschlange eingetragen (sleep und wakeup)
- Probleme bei Semaphoren
 - Anwendung erfordert Disziplin
 - * Für jedes `P(s)` braucht es auch ein `V(s)`
 - * Probleme treten auf, wenn `V(s)` vergessen geht!
 - In grösseren Programmen können subtile Probleme entstehen, falls z.B. das `V(s)` in einer if-Bedingung gemacht wird.
 - Beim Auftreten einer Exception kann das Freigeben ebenfalls schwierig sein.

12.5 Thread-Synchronisation in C mit POSIX Threads

```

1 | static volatile int val = 0; // shared resource
2 | static pthread_mutex_t valMtx;
3 |
4 | int main(void)
5 | {
6 |     pthread_t t;
7 |     pthread_mutex_init(&valMtx, 0); // init mutex
8 |     pthread_create(&t, 0, threadRoutine, 0);
9 |     pthread_join(t, 0);
10 |    pthread_mutex_destroy(&valMtx); // destroy mutex
11 |    return 0;
12 | }
13 |
14 | void* threadRoutine(void* arg)
15 | {
16 |     while (1)

```

```

17 {
18     // non critical section
19     pthread_mutex_lock(&valMtx); // start of critical section
20     // critical section; access shared resource
21     pthread_mutex_unlock(&valMtx); // end of critical section
22     // non critical section
23 }
24 }

```

12.5.1 Das Monitorprinzip

- Grundprinzip: Abstrakten Datentyp (ADT) definieren, der genau die Funktionen in der Schnittstelle anbietet, die notwendig sind.
- Der Aufrufer ruft diese Funktionen auf, er muss sich aber nicht um die Synchronisation kümmern.
- Die Synchronisation (z.Bsp. mit Semaphoren) ist in der Implementation des Monitors lokal gelöst.
- Problem wird einmal im Monitor gelöst, die Aufrufer müssen sich nicht mehr darum kümmern.

12.6 Petri-Netze

12.6.1 Was sind Petri-Netze?

- Petri-Netze (Petri Nets) sind eine graphische Darstellung von parallelen Prozessen.
- Petri-Netze sind bipartite Graphen (Graphen mit zwei Arten von Knoten)
- Petri-Netze sind mathematische Modelle. Es bestehen Methoden zur Optimierung und formalen Verifikation (Deadlock-Analyse).

12.6.2 Elemente

P	Stelle (Place)	Stellen sind passive Elemente und können als Zustände oder Datenspeicher interpretiert werden, die Marken beinhalten können. Sie werden mit einem kurzen Text angeschrieben. Stellen können mit einer ganzzahligen Anzahl Marken vorbelegt werden.
		
T	Transition (Transition)	Transitionen sind die aktiven Elemente in einem Petri-Netz. Sie werden mit einem kurzen Text angeschrieben. Transitionen können feuern, sobald alle Stellen, die in die Transition hineinführen, mit mindestens einer Marke belegt sind. Transitionen stellen Ereignisse dar.
		
	Marke (Token)	Marken sind die dynamischen Elemente in einem Petri-Netz. Sie können verfügbare Ressourcen symbolisieren. Die Marken können sich nur in Stellen befinden und werden durch das Feuern verschoben. Üblicherweise sind mehrere Stellen gleichzeitig markiert, d.h. mit mindestens einer Marke belegt. Jede markierte Stelle stellt einen aktiven Zustand dar.
		
	Kante (Arc)	Kanten sind gerichtete, unidirektionale Verbindungen, die Stellen mit Transitionen oder Transitionen mit Stellen verbinden. Gleichartige Elemente (Stellen mit Stellen, bzw. Transitionen mit Transitionen) können nicht verbunden werden.

12.6.3 Feuern von Transitionen

- Das Feuern ist die einzige Aktion, die in einem Petri-Netz ausgeführt werden kann.
- Eine Transition kann dann und nur dann feuern, wenn alle Stellen, die in die Transition hineinführen (Eingabestellen), mit mindestens einer Marke belegt sind. Dann wird bei jeder dieser Stellen eine Marke abgezogen.
- Jede Stelle, die aus dieser Transition hinausführt (Ausgabestellen), erhält umgekehrt eine Marke.
- Das Feuern ist somit das Verschieben von Marken unter definierten Bedingungen.

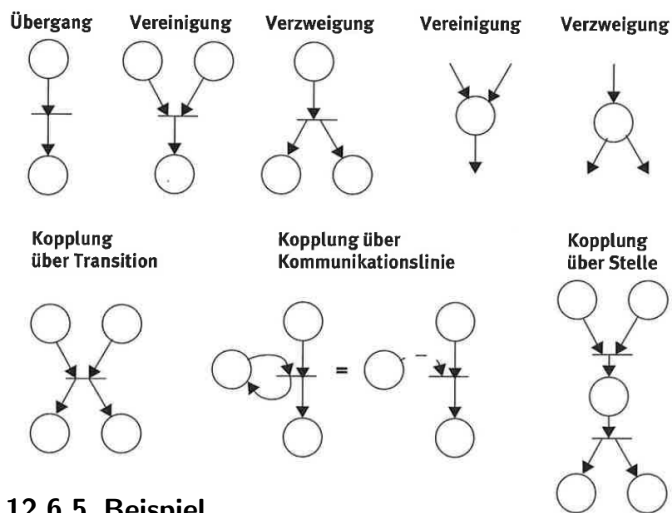


Race Condition Das Ergebnis einer Operation hängt von zeitlichen Verhalten bestimmter Einzeloperationen ab.

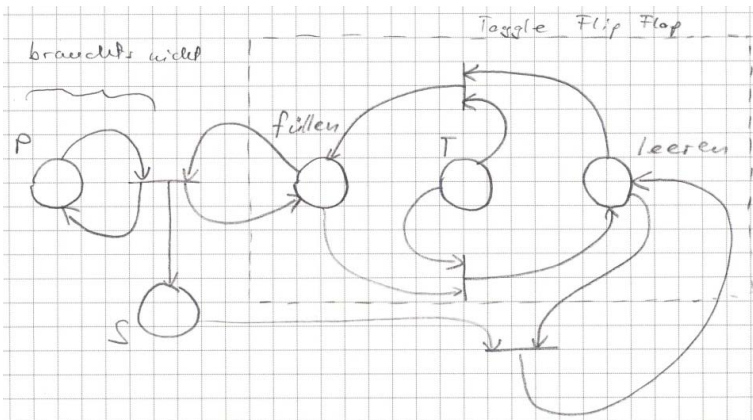
Starvation Ist ein Zustand, bei dem ein Prozess nie dran kommt (er verhungert). Die Fairness condition besagt, dass Starvation verhindert werden muss.

Deadlock Ist eine Situation in denen sich zwei Prozesse gegenseitig blockieren. Ein Deadlock kann vermieden werden, indem alle Prozesse die gemeinsamen Ressourcen immer in derselben Reihenfolge anfordern.

12.6.4 Grundstrukturen



12.6.5 Beispiel



In diesem Beispiel produziert der Produzent P bei jedem Takt ein Marke in S. Sobald eine Marke in T gesetzt wird, wird S geleert. P ist während dieses Vorgangs gesperrt. P produziert wieder, wenn erneut eine Marke in T gesetzt wird. Der Leerprozess wird dann gestoppt.

13 Bootstrap

13.1 Programmdownload

- Während der Entwicklung ist das Embedded System häufig über eine JTAG-Verbindung mit der Entwicklungsumgebung verbunden
- JTAG-Schnittstellen erfordern häufig (teure) Adapter, die für die Entwicklung akzeptabel, im Betrieb aber zu teuer sind
- Für den Programmdownload im Betrieb steht häufig nur eine serielle Schnittstelle zur Verfügung

13.2 Startvorgang (Bootting)

- Das Starten eines Systems wird auch mit Booting bezeichnet
- Das Programm, welches beim Starten als erstes ausgeführt wird, heisst Bootloader oder Bootstrap loader
- Der Bootloader ist für die ersten Initialisierungen verantwortlich und für das Starten des eigentlichen Programms (oft das Betriebssystem)

13.3 Mehrstufige Bootloader (multistage bootloader)

- Der Bootloader muss immer auf einem bootfähigen nicht flüchtigen Medium gespeichert sein
- Häufig wird der Bootloader in mehreren Stufen aufgebaut, z.B.
 - erste Stufe initialisiert die Hardware und lädt ein bestimmtes Filesystem
 - die nächste Stufe ist als File in diesem Filesystem gespeichert und startet beispielsweise das Betriebssystem
- Vorteile:
 - die erste Stufe ist nur hardwarespezifisch, jedoch völlig unabhängig von Filesystem und Betriebssystem
 - dadurch können diese Teile beliebig ausgetauscht und geändert werden
 - wenn die erste Stufe "nie" ändert, kann auch vermieden werden, dass sich das System durch eine fehlerbehaftete neue Version selbst abschießt, d.h. dass dann das System nicht mehr ansprechbar wird

13.4 Sicherung des Programmdownloads

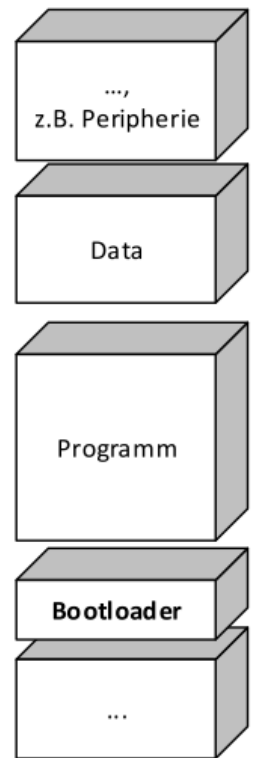
- Um die sichere Übertragung zu gewährleisten, wird diese meist mittels CRC (Cyclic Redundancy Check) gesichert
- Teilweise wird die Übertragung auch verschlüsselt, damit eine bössartige Änderung des Programms verhindert werden kann

13.5 Typische Memory Map

- Der Bootloader liegt im nicht flüchtigen Bereich und wird nie geändert
- Das Programm ist meist im Flash EPROM und kann geändert werden oder das Programm wird von einer SD-Card ins RAM geladen
- Der Datenblock ist immer im RAM
- Weitere Speicherbereiche können für die Peripherie vorgesehen werden (Memory mapped I/O) oder bleiben ungenutzt

13.6 Typischer Bootvorgang

1. Beim Aufstarten wird immer zuerst das Bootloader-Programm gestartet
2. In diesem Programm wird z.B. geprüft, ob über eine Kommunikationsverbindung ein neues Programm in den Programmbereich geschrieben werden soll
3. Wenn ein neues Programm geschrieben werden soll, dann wird dieses mit Hilfe eines definierten Protokolls übertragen und geschrieben
4. Wenn das Programm fertig übertragen wurde oder das bestehende ausgeführt werden soll, wird mit der Ausführung an dieser Stelle fortgefahren



14 Style guide und Language Linkage

14.1 Style guide

- Programme werden für Programmierer geschrieben, nicht für Compiler
- Programmierkonventionen erleichtern einen konsistenten Stil, der auch von anderen verstanden wird
- Unwichtiges nicht überbewerten
- Alles, was der Compiler entdeckt und mit Warnings meldet (z.B. nicht initialisierte Variablen) muss nicht explizit in den Codierrichtlinien erwähnt werden
- Seien Sie konsistent und konsequent!

14.1.1 Namenskonventionen

- Wenige Underscores (.) verwenden
- Keine Namen mit Underscore (.) beginnen
- Zusammengesetzte Namen mit `camelCase` betonen, nicht `camel_case`
- Präfixe bei Variablen sehr zurückhaltend einsetzen oder ganz vermeiden (z.B. `i_myVariable` für Integer)
- Selbstdefinierte Typen und Klassen mit Grossbuchstaben beginnen
- Funktionen und Methoden mit Kleinbuchstaben beginnen
- Variablen und Objekte mit Kleinbuchstaben beginnen
- Makros (`#define`) ausschliesslich mit Grossbuchstaben definieren, Trennung falls notwendig mit Underscores, z.B. `MY_UGLY_MACRO`

14.1.2 Typen mit genauer Breite

- Ab C99 werden im Headerfile `<stdint.h>` verschiedene Typen mit genauer Breite definiert. Wenn solche Typen benötigt werden, sollen genau diese Namen verwendet werden oder sind allenfalls mittels `typedef` zu definieren.
- Die Konventionen lauten wie folgt:
 - `intN_t`, wobei $N = 8, 16, 32$ für `signed int`-Typen
Beispiele: `int8_t`, `int16_t`, `int32_t`
 - `uintN_t`, wobei $N = 8, 16, 32$ für `unsigned int`-Typen
Beispiele: `uint8_t`, `uint16_t`, `uint32_t`

14.2 Language Linkage

14.2.1 Plain Old Data Types (POD Types)

- Datentypen, die bereits in C vorhanden sind, werden als POD Types bezeichnet
- Dazu gehören `char`, `short`, `int`, `long`, jeweils `signed` und `unsigned`, `float` und `double`
- POD types funktionieren in C++ identisch zu C

14.2.2 Name Mangling

Beim Übersetzen in Assembler müssen eindeutige Labels entstehen. In C ist dies kein Problem, da die Funktionsnamen bereits eindeutig sind. Hingegen ist dies in C++ ist nicht der Fall. Deshalb weist der Compiler den Funktionen einen eindeutigen Namen zu (dieser Prozess wird als Mangling bezeichnet).

C	<code>foo()</code>	→	<code>_foo</code>
C++	<code>foo(int)</code>	→	<code>_foo.i</code>
	<code>foo(double, int)</code>	→	<code>_foo.d.i</code>
	<code>MyClass::foo(int)</code>	→	<code>_MyClass.foo.i</code>

14.2.3 Einbinden von C-Bibliotheken unter C++

Wird in C++ eine Funktion `foo(int)` aufgerufen welche in einer compilierten C-Library vorliegt, sucht der Compiler nach einer Funktion `_foo.i`. Diese existiert aber nicht da sie `_foo` heisst. Deshalb muss dem Compiler mitgeteilt werden, dass es sich um eine C-Funktion handelt.

```

1 | extern "C" void foo(int);      \\ use C language linkage
2 | extern void goo(int);         \\ use C++ language linkage
3 | void hoo(int);                \\ use C++ language linkage
4 | extern "C++" void koo(int);   \\ use C++ language linkage
5 |
6 | //optimale Anwendung von extern "C" (koennen in C und C++ Dateien included werden)
7 | #ifdef __cplusplus
8 | extern "C"
9 | {
10 | #endif
11 | // list multiple prototypes with C linkage, or
12 | // #include C header file(s)
13 | #ifdef __cplusplus
14 | }
15 | #endif

```

15 Eigene Notizen

