Embedded Software Engineering 1

HS 2024 – Prof. Reto Bonderer

Autoren: Laurin Heitzer, Simone Stitz

https://github.com/P4ntomime/EmbSW1

V1.0.20240924

Inhaltsverzeichnis

1 Embedded Systems – Allgemein

1.1 Definition

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

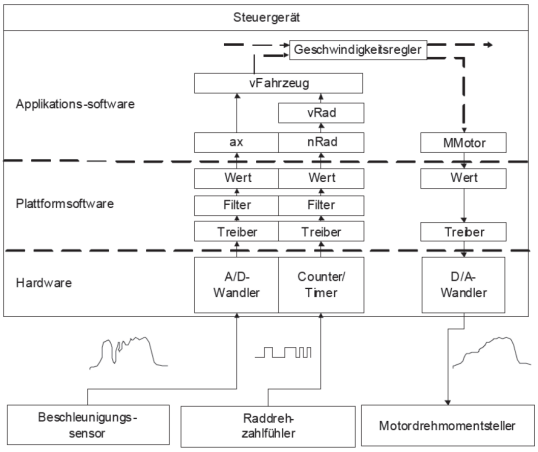
Ein Embedded System beinhaltet typischerweise folgende Komponenten:
- Sensoren
- Aktoren
- Mikrocomputer
- Software (Firmware)
- Hardware (Mechanik, Elektronik)

1.1.1 Charakterisierung von Embedded Systems

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

1.1.2 Typischer Aufbau

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



1.2 Beispiele

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

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

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

➔ Auto ist ein riesiges Embedded System

1.3 Deeply Embedded System

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

1.3.1 Beispiele – Deeply Embedded System

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

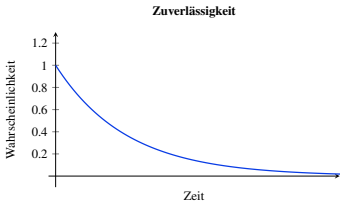
1.4 Betriebssysteme bei Embedded Systems

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

1.5 Bare Metal Embedded System

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

1.6 Zuverlässigkeit



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

Achtung: Hier ist nur die Alterung der Hardware berücksichtigt

1.7 Verfügbarkeit

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

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

1.8 Abstraktionsschichten

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

1.8.1 Hardware-abstraction-layer (HAL)

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

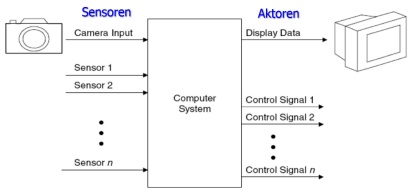
2 Real-Time System (Echtzeitsystem)

2.1 Definitionen

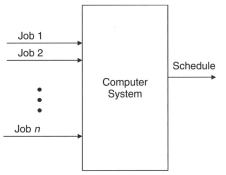
2.1.1 Real-Time System (Echtzeitsystem)

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

Typisches Echtzeitsystem

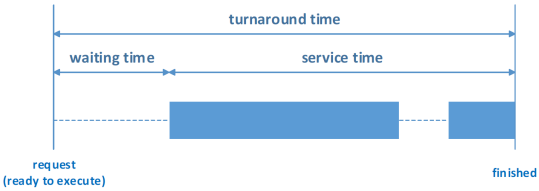


Repräsentation RT-System



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

2.1.2 Zeitdefinitionen (Task)



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

2.2 Fehlverhalten eines Systems (failed system)

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

2.3 Echtzeitdefinition – Verschiedene Echtzeitsysteme

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

2.3.1 Beispiele verschiedener Echtzeitsysteme

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

2.4 Determinsismus (determinacy)

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

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

2.5 Auslastung (utilization)

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

2.5.1 Berechungen zur Auslastung (utilzation)

Annahmen:
- System mit $n \geq 1$ periodischen Tasks T_i und Periode p_1
- Jeder Task T_i het bekannte / geschätzte maximale (worst case) execution time e_i

Auslastungsfaktor eines Tasks

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

➔ utilization factor

Gesamtauslastung des Systems

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

➔ utilization

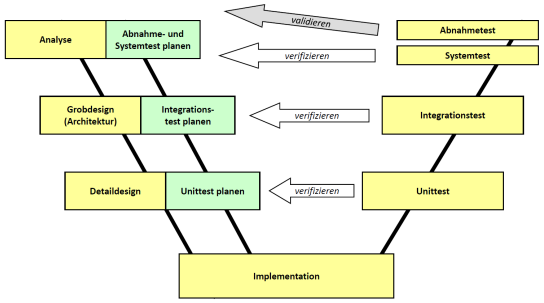
➔ Bei 69 % Auslastung ist das **'theoretical limit'**

2.6 Real-time Scheduling

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

3 Modellierung eines Embedded Systems

3.1 V-Modell für Software-Entwicklungszyklus



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

3.2 Model Driven Development (MDD)

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

3.3 Vorgehen bei der Modellierung

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

Stukturmodellierung (Statische Aspekte)

Modellierung der dynamischen Aspekte

3.4 Systemgrenze definieren & Systemprozesse finden

3.4.1 Systemgrenze definieren

Die Festlegung der Systemgrenze ist das Wichtigste und Allererste bei sämtlichen Systemen!
Man sollte sich die folgenden Fragen stellen und diese beantworten:

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

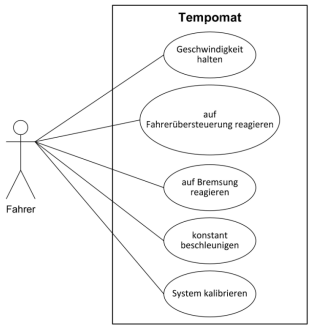
3.4.2 Systemprozesse finden (use-cases)

Da man sich noch immer in der **Analyse** befindet, sollen nur die **Anforderungen** definiert werden. Die Umsetzung ist Teil des Designs!
Um die Use-Cases zu identifizieren, sollte folgendes beachtet werden:

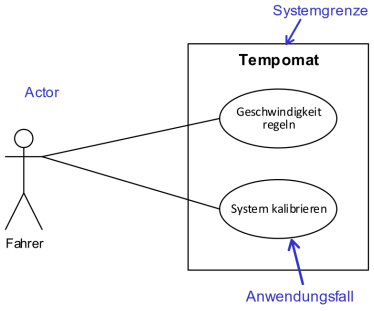
- Aussenbetrachtung des Systems (**oberflächlich!**)
  - Nicht komplizierter als nötig
- System als Blackbox betrachten
  - **Was** soll System können; (nicht: wie soll das System etwas machen)
- RTE-Systeme bestehen häufig aus nur einem einzigen Systemprozess
  - speziell wenn System 'nur' ein Regler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

3.4.3 Kontextdiagramm: Use-Case Diagramm

Tempomat: zu detailliert



Tempomat: verbesserte Version



3.4.4 Kontextdiagramm: Sequenzdiagramm

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

3.5 Verteilungen festlegen

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

3.5.1 Verteilungsdiagramm

Knoten: Darstellung der örtlichen Verteilung der Systeme
Knoten können auch hierarchisch aufgebaut sein
Linien: Physikalische Verbindungen der Knoten (Netzwerke, Kabel, Wireless, etc.)

Beispiel: Tempomat