

# Echtzeitsysteme

## Rangfolge und gerichtete Abhängigkeiten

**Peter Ulbrich**

Lehrstuhl für Verteilte Systeme und Betriebssysteme  
Friedrich-Alexander-Universität Erlangen-Nürnberg

<https://www4.cs.fau.de>

08. Januar 2018




## Terminvereinbarung zur mündlichen Prüfung

- Erfolgt elektronisch (Poll) → [Windhundverfahren](#)
- Implizite Terminbestätigung → Übertrag in meinCampus

 **Anmeldefrist:** Semesterende (09.02.2018)

## Anmeldevorgang

- Link zur Teilnahme → Mail (08.01.18, ca. 14 Uhr) an alle Angemeldeten<sup>1</sup>

 **Teilnehmerformat:** <Nachname, Vorname> EZS<7,5/5>  
(Beispiel: Müller, Klaus EZS7,5 oder Meier, Emelie EZS5)

## ■ Terminliche Probleme, Änderungen und Abmeldung

- Individuelle Terminvereinbarung in Ausnahmefällen möglich
- Rechtzeitige Abmeldung erlaubt uns Reorganisation

→ Wir beißen nicht!

---

<sup>1</sup>Sonst bitte umgehend Mail an uns!



## Evaluation der Veranstaltung

- Eure Meinung (**Lob/Kritik**) ist uns wichtig!
  - Eure Rückmeldung hat Konsequenzen (z.B. Folien-Redesign)
- Bitte evaluiert **Vorlesung** und **Übungen**



Rückläuferquote im Durchschnitt  $\mapsto$  **2 – 10%**

- Zu wenig für eine sinnvolle Einschätzung
- Aber: Typische Rückläuferquote in EZS  $\mapsto$  **60 – 80%**

## Motivationsanreiz zur Evaluation



- **Traditionell:** Kaffee und Kekse in der letzten Vorlesung
- **Feste Bedingung:**  $\geq 60\%$  der ausgegebenen TANs werden evaluiert!



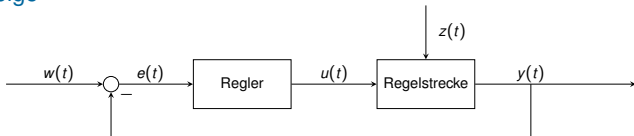
- Was bedeutet **Rangfolge**?
  - Was ist die Ursache von Rangfolge?
  - Wie beschreibt man Rangfolge?
  
- Wie kann **Rangfolge implementieren** werden?
  - Welche Implementierungsvarianten gibt es?
  - Welche Implikationen haben sie?
  
- Was bedeuten Rangfolgebeziehungen für die **Ablaufplanung**?

- 1 Grundlagen
  - Datenabhängigkeiten
  - Nebenläufigkeit
  - Abhängigkeits- und Aufgabengraphen
  - Koordinierung
- 2 Effekte in Echtzeitsystemen
  - Zeitliche Domänen
  - Physikalisch und logische Ereignisse
- 3 Lösungsverfahren
  - Analytische Koordinierung
  - Konstruktive Koordinierung
- 4 Ablaufplanung
- 5 Zusammenfassung





Ausführung von Arbeitsaufträgen unterliegt häufig einer **bestimmten Reihenfolge**  
↳ **Rangfolge**



- Beispiel: **Regelungsanwendung**
  - Signalverarbeitungsauftrag muss vor der Regelung gelaufen sein
- Beispiel: **Kommunikationssystem**
  - Sendeauftrag muss vor Empfangsauftrag gelaufen sein
  - Empfangsauftrag muss vor Bestätigungsauftrag gelaufen sein
- Beispiel: **Anfragesystem**
  - Eingabeauftrag muss vor Suchauftrag gelaufen sein
  - Suchauftrag muss vor Ausgabeauftrag gelaufen sein



Rangfolge ist oft in **Datenabhängigkeiten** begründet





Arbeitsaufträge benötigen ggf. **konsumierbare Betriebsmittel**

- Anzahl ist (log.) unbegrenzt: Nachrichten, Signale, Interrupts

**Produzent** kann beliebig viele davon erzeugen

**Konsument** zerstört sie wieder bei Inanspruchnahme

→ Zwischen ihnen besteht eine **gerichtete Abhängigkeit**



Produzent und Konsument sind voneinander **abhängige Entitäten**

- Abhängigkeit: Konsument → Produzent

- Betriebsmittel muss vor Inanspruchnahme zunächst bereitgestellt werden

- Abhängigkeit: Produzent → Konsument (seltener)

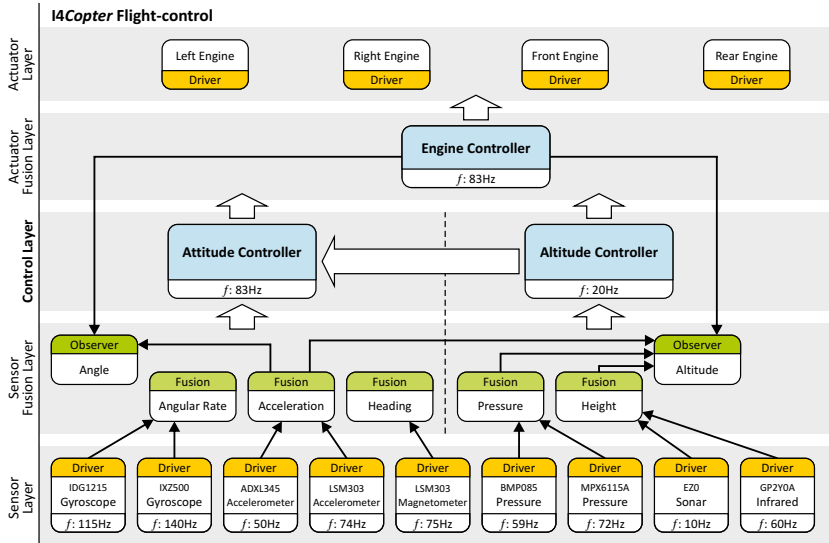
- Abbildung **konsumierbare** ↔ **wiederverwendbare Betriebsmittel**

- Beispiel: **begrenzter Puffer** (engl. *bounded buffer*)

→ Produzent fordert ein wiederverwendbares Betriebsmittel an, welches vom Konsumenten später wieder freizugeben ist



# Datenabhängigkeiten im I4Copter





# Nebenläufige Aktivitäten

## Kausalität (lat. *causa*: Ursache)

Die Beziehung zwischen **Ursache** und **Wirkung**, d.h., die ursächliche Verbindung zweier Ereignisse.

## Nebenläufigkeit (engl. *concurrency*)

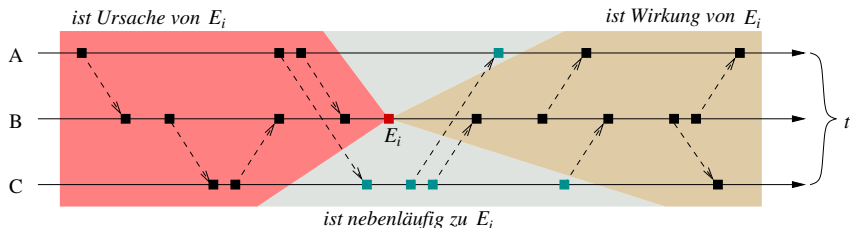
Bezeichnet das Verhältnis von nicht kausal abhängigen, sich entsprechend nicht beeinflussenden, Ereignissen.

- Ereignisse sind **nebenläufig**, wenn keines Ursache des anderen ist
  - Aktionen können nebenläufig ausgeführt werden, wenn keine das Resultat des anderen benötigt
- Beispiel eines nichtsequentiellen Programms:

```
1:   foo = 4711;  
2:   bar = 42;  
3:  foobar = foo + bar;  
4:  barfoo = bar + foo;  
5:   hal = foobar + barfoo;
```

- Zeile 1 kann nebenläufig zu Zeile 2 ausgeführt werden
- Zeile 3 kann nebenläufig zu Zeile 4 ausgeführt werden





- **Relationen:** Ursache  $\leftrightarrow$  Wirkung  $\leftrightarrow$  Nebenläufigkeit
  - Kausalkette von Ereignissen in Bezug zu einem Ereignis  $E_i$
- Bezogen auf Raum<sup>2</sup> und Zeit

☞ Ein Ereignis  $E_j$  ist **nebenläufig** zu einem anderen:

- Es ist weder in der Zukunft noch in der Vergangenheit des Anderen
  - Es ist weder Ursache oder Wirkung des anderen Ereignisses
- Es liegt im **Anderswo** anderen Ereignisses

<sup>2</sup>A, B und C bezeichnen Ausführungsstränge auf einem Rechensystem.

- Ein Arbeitsauftrag kann **nebenläufig** bearbeitet werden, wenn:
  - Im Allgemeinen** ■ Er benötigt kein Ergebnis eines Anderen (vgl. Folie 9)
    - Abwesenheit von **Datenabhängigkeiten**
  - Im Speziellen** ■ Er hängt zeitlich nicht von anderen Aufträgen ab
    - Termintreue (weich/fest bzw. hart) wird beibehalten
    - Periodizität wird beibehalten
    - Abwesenheit von **Zeitabhängigkeiten**
- Zusammenwirken von Ereignissen **beschränkt Nebenläufigkeit**

### Ereigniskorrelation vs. Bearbeitungsmodell

“ist Ursache von” } → **sequentiell** (verwirklicht vor/zur Laufzeit)  
“ist Wirkung von” }  
“ist nebenläufig zu” → **parallel** (logisch/tatsächlich)



Minimierung von **sequentiell** Programmcode ist (auch) in Echtzeitsystemen von Bedeutung



# Beispiel: Serieller Empfang von Nachrichten

Implementierung orientiert sich an OSEK OS [7] bzw. AUTOSAR OS [2]

- Nachrichtenverarbeitung besteht aus **zwei getrennten Aufgaben**
  - Empfang** Abholen einzelner Bytes und Zusammensetzen von Nachrichten
  - Verarbeitung** Nachricht verarbeiten und Behandlung aktivieren

## Empfang

```
Pool *msgPool;  
Buffer *msgBuffer; Message *msg;  
  
ISR(SerialByte) {  
    uint8_t rec = rs232_get();  
    msg_addTo(msg, rec);  
  
    if(msg_isComplete(msg)) {  
        buffer_ins(msgBuffer, msg);  
        msg = pool_getfree(msgPool);  
    }  
    return;  
}
```

## Verarbeitung

```
TASK(MsgHandler) {  
    Message *cMsg = 0;  
  
    InitHandler();  
  
    cMsg = buffer_get(msgBuffer);  
    msg_prepare(cMsg);  
    handle(cMsg);  
  
    TerminateTask();  
}
```



**Datenabhängigkeit**  $\rightsquigarrow$  gemeinsamer Puffer msgBuffer



**Rangfolge**  $\rightsquigarrow$  Wann kann die Nachricht verarbeitet werden?

???



$\rightarrow$  Wann wird TASK(MsgHandler) aktiv?

- Die **Kausalordnung** wird durch eine **Vorgängerrelation** (engl. *precedence relation*) beschrieben:
  - $J_i \rightarrow J_k$ : Arbeitsauftrag  $J_i$  ist **Vorgänger** (engl. *predecessor*) von  $J_k$
  - Ausführung des **Nachfolgers** (engl. *successor*)  $J_k$  erfordert die Fertigstellung des Vorgängers  $J_i$
- Beispiel auf Folie 12:
  - **ISR(SerialByte)** ist der Vorgänger
    - Zuerst muss die Nachricht vollständig empfangen werden, ...
  - **TASK(MsgHandler)** ist der Nachfolger
    - ... anschließend findet die eigentliche Nachrichtenbehandlung statt.



**Koordinierte Ausführung** von ISR(SerialByte) und TASK(MsgHandler) ist für **korrekte Funktion** notwendig



**Aufgabe  $T_1$**  Empfang einzelner Bytes  $\rightsquigarrow$  Aufträge  $J_{1,1}, J_{1,2}, \dots$

**Aufgabe  $T_2$**  Bearbeitung der Nachrichten  $\rightsquigarrow$  Aufträge  $J_{2,1}, J_{2,2}, \dots$



- **Keine Abhängigkeiten** zwischen Aufträgen von  $T_1$  und  $T_2$ 
  - Termin  $D_{1,1}$  erzwingt lediglich Fertigstellung von  $J_{1,1}$  vor  $J_{1,2}$ :  $D_{1,1} \leq r_{1,2}$
- Arbeitsaufträge  $J_{1,1}, \dots, J_{1,n}$  ermöglichen die Ausführung von  $J_{2,1}$ 
  - Verarbeitung der Nachricht nach vollständigem Empfang
  - $J_{1,1}, \dots, J_{1,n}$  sind Vorgänger von  $J_{2,1}$
- ⚠ **Endgültige Abhängigkeitsbeziehungen erst zur Laufzeit bekannt**
  - Nachrichten können unterschiedlich viele Bytes umfassen
  - Unterschiedlich viele Vorgänger von  $J_{2,1}$  und  $J_{2,l}$





- **Statisch durch Einplanung**  $\rightsquigarrow$  **analytische Verfahren**
  - Ablaufpläne berücksichtigen Rangfolgen und Datenabhängigkeiten
    - **à priori Wissen**  $\mapsto$  periodische Aufgaben
  - Arbeitsaufträge laufen komplett durch (engl. *run to completion*)
    - Warten weder ex- noch implizit, dürfen jedoch verdrängt werden

→ Ergebnis ist ein System von ausschließlich **einfachen Aufgaben**
  
- **Dynamisch durch Kooperation**  $\rightsquigarrow$  **konstruktive Verfahren**
  - Synchronisationspunkte in den Programmen explizit machen
    - d.h., **Zeitsignale austauschen**  $\mapsto$  Semaphor
  - Arbeitsaufträge sind Produzenten/Konsumenten von Ereignissen
    - physikalische Ereignisse** von den kontrollierten Objekten
    - logische Ereignisse** von anderen Arbeitsaufträgen

→ Ergebnis ist ein System von (ggf. vielen) **komplexen Aufgaben**



- 1 Grundlagen
  - Datenabhängigkeiten
  - Nebenläufigkeit
  - Abhängigkeits- und Aufgabengraphen
  - Koordinierung
- 2 Effekte in Echtzeitsystemen
  - Zeitliche Domänen
  - Physikalisch und logische Ereignisse
- 3 Lösungsverfahren
  - Analytische Koordinierung
  - Konstruktive Koordinierung
- 4 Ablaufplanung
- 5 Zusammenfassung





## Implizite Codierung gerichteter Abhängigkeiten im Quelltext

- Vorgänger und Nachfolger sind **unveränderlich** und **à priori bekannt**
- Hier: Behandlung nach vollständigem Empfang der Nachricht

```
Message *msg;

ISR(SerialByte) {
    uint8_t received = rs232_getByte();
    msg_addTo(msg, received);

    if(msg_isComplete(msg)) {
        InitHandler();

        msg_prepare(currentMsg);
        handle(currentMsg);

        msg_clear(msg);
    }
}
```

### Einfache Implementierung

- Nur ein Aktivitätsträger
- Rangfolge unmittelbar ablesbar
- Keine Pufferung/Koordinierung notwendig



Entwurfsvariante mit gravierenden Implikationen!





# Nachteile implizit codierter Abhängigkeiten

## Zeitliche Domänen

Innerhalb einer **zeitlichen Domäne** (engl. *temporal domain*) ist das zeitliche Verhalten einheitlich:

- Ereignisse mit gleichen zeitlichen Eigenschaften
  - Typischerweise durch eine Aufgaben behandelbar
- 
- Zeitliche Domänen des Nachrichteneingangs:
    - Empfang  $\rightsquigarrow$  Nicht-periodische Aufgabe  $T_1 = (i_1, e_1)$
    - Verarbeitung  $\rightsquigarrow$  Nicht-periodische Aufgabe  $T_2 = (i_2, e_2)$
    - Empfang mehrere Bytes pro Nachricht  $\rightsquigarrow i_1 \ll i_2$
    - Verarbeitung ist komplexer als deren Empfang  $\rightsquigarrow e_2 \gg e_1$



Naive Implementierung **verletzt zeitlichen Domänen**

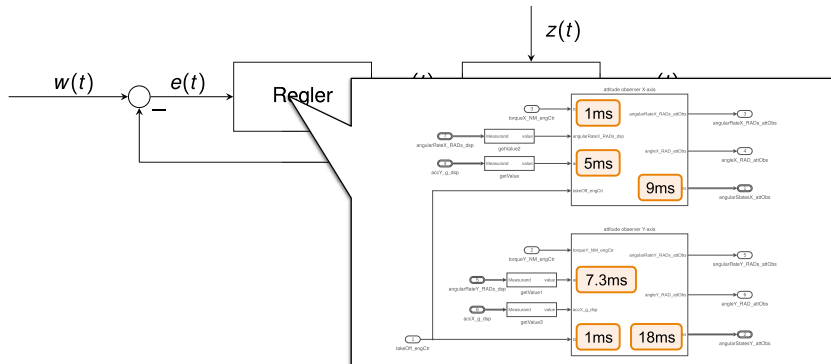
- Ergebnis ist eine Aufgabe  $T'_1 = (\min(i_1, i_2), e_1 + e_2)$
- **Unrealistische** zeitliche Parameter  $\rightsquigarrow$  Überabschätzung des Aufwands



Gerichtete Abhängigkeiten  $\mapsto$  Hinweis auf **versch.** zeitliche Domänen

- Aufgaben mit dedizierten Auslöseereignissen und zeitlichen Parametern





- **Signaldatenverarbeitung im Fokus**
  - Scheinbar einfache Funktion  $\rightarrow$  Potpourri zeitlicher Domänen
  - Jeder Sensor ist einem physikalischen Ereignis zugeordnet
  - Werte werden in Fusionsfiltern zusammengeführt





# Übergang zwischen zeitlichen Domänen

Produzenten und Konsumenten werden mit unterschiedlichen Raten aktiviert



Koordinierung verschiedener zeitlicher Domänen (vgl. Folie 8)

- Unterschiedliche Raten in den Bereichen des Echtzeitsystems
- Gerichtete Abhängigkeiten erfordern **Angleichung**

■ Datenaustausch zwischen Produzent und Konsument

- Erfolgt in Abstimmung → Konsument erwartet Daten
- Aufwand abhängig von der Diskrepanz der Raten



Typisches Vorgehen in Echtzeitanwendungen

- **Gemeinsamer Puffer** als Zwischenspeicher → Produzent schneller
  - Problem: Puffergröße und WCET (Abarbeitung des Rückstands)
- **Prädikation** durch Beobachter → Konsument schneller<sup>3</sup>
  - Generierung von Zwischenwerten kompensiert langsamen Produzenten
- **Letzter Wert genügt** (engl. *last is best*) → beidseitig
  - Verzicht auf explizite Abstimmung (**simpel**)
  - **Alter unterliegt gewissen Schwankungen**

<sup>3</sup>Sonderfall in der digitalen Signalverarbeitung: Zukünftige Messwerte lassen sich mittels Modellen des physikalischen Systems in gewissem Umfang vorhersagen.



- Verschmelzung **zeitlich identischer Domänen** ist möglich
  - Stellt eine **Optimierung der Implementierung** dar



Letzter Schritt des Systementwurfs [3, 4]

**1** Identifikation der zeitlichen Domänen

- Exklusive Abbildung jeder Domäne auf eine Aufgabe

**2** Vereinigung **äquivalenter** zeitlicher Domänen

- Reduktion von Aufgaben mit **gleichartigen Parametern**
- **Zeitliche Kohäsion**: Aufgaben werden immer gleichzeitig aktiviert
- **Sequentialisierung**: (Teil-)Aufgaben laufen immer nacheinander ab



Naive Implementierung nimmt diese Optimierung vorweg

- Auch wenn die zeitlichen Domänen **verschieden** sind



Entkopplung zeitlicher Domänen durch **logische Ereignisse**



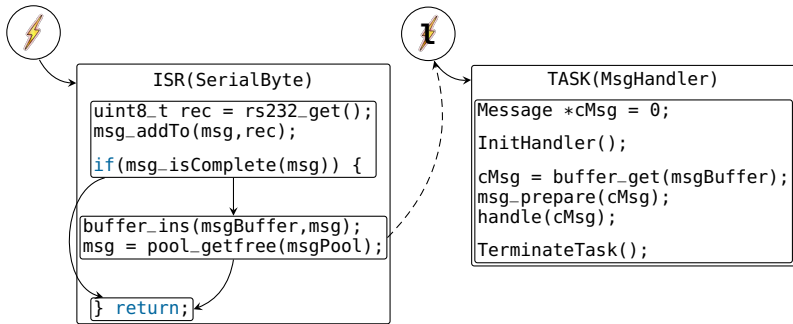


# Physikalische und logische Ereignisse

- **Physikalische Ereignisse** → Zustandsänderungen der Umwelt
  - Empfang eines Byte auf der seriellen Schnittstelle
  - Auslösung einer Unterbrechung
- **Logische Ereignisse** ruft die Echtzeitanwendung selbst hervor
  - Vollständiger Empfang einer Nachricht



Das logische Ereignis entkoppelt Empfang und Verarbeitung zeitlich



- 1 Grundlagen
  - Datenabhängigkeiten
  - Nebenläufigkeit
  - Abhängigkeits- und Aufgabengraphen
  - Koordinierung
- 2 Effekte in Echtzeitsystemen
  - Zeitliche Domänen
  - Physikalisch und logische Ereignisse
- 3 Lösungsverfahren
  - Analytische Koordinierung
  - Konstruktive Koordinierung
- 4 Ablaufplanung
- 5 Zusammenfassung





Herstellung der Rangfolge ohne die zeitliche Nähe durch eine entsprechende Anordnung im Quelltext zu erzwingen

- **Ohne Koordinierung**  $\leadsto$  Rangfolge bewusst vernachlässigen

→ **Last is best**: Schwankungen in der Aktualität sind tolerierbar

- **Analytische Koordinierung**  $\leadsto$  mithilfe der Ablaufplanung

- Nur für Abhängigkeiten zwischen **periodischen Aufgaben** anwendbar

→ **Arbeitsaufträge** werden nicht parallel ausgeführt (Folie 25)

**Taktsteuerung**: Überlappungsfreie Anordnung in der Ablauftabelle

**Vorrangsteuerung**: Analog durch Phasenversatz

- **Konstruktive Koordinierung**  $\leadsto$  mithilfe expliziter Synchronisationsmechanismen des Echtzeitbetriebssystems

- Für **nicht-periodischen Aufgaben** unumgänglich

- In zeitgesteuerten Systemen **unsinnig**

→ Es existiert eine Vielzahl Synchronisationsmechanismen (Folie 26 ff)

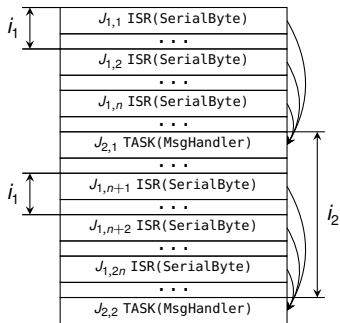






# Rangordnung mittels statischer Ablaufplanung

- Eingabe für die **statische Ablaufplanung** (s. Folie IV-3/19 ff) ist ein Abhängigkeitsgraph. Die erzeugte Ablauftabelle muss die folgenden Randbedingungen einhalten:



- Überführung von  $T_1$  und  $T_2$  in äquivalente **periodische Aufgaben**
  - Periode  $p_n =$  Zwischenankunftszeit  $i_n$
- Anordnung nach Abhängigkeit
  - $r_{i,j} + e_i \leq r_{n,m} \Leftrightarrow J_{i,j} \mapsto J_{n,m}$
- **Phasenverschobene Ausführung**
  - Analoges Vorgehen bei ereignisgesteuerten Systemen
  - Rangfolge impliziert passende Phase  $\phi_m$ :  

$$\phi_m = \max_{J_{i,j} \mapsto J_{m,n}} r_{i,j} + \omega_{i,j}$$



Einhaltung der Phase wird zur Laufzeit nicht überwacht

→ Laufzeitüberschreitungen  $\rightsquigarrow$  ggf. Verletzungen der Rangfolge



# Rangfolge durch Bereitstellung des Nachfolgers

## Konstruktive Umsetzung der Rangordnung

### AUTOSAR OS [2]

```
ISR(SerialByte) {  
    uint8_t rec = rs232_get();  
    msg_addTo(msg, rec);  
  
    if(msg_isComplete(msg)) {  
        buffer_ins(msgBuffer, msg);  
        msg = pool_getfree(msgPool);  
        ActivateTask(MsgHandler);  
    }  
    return;  
}
```

```
TASK(MsgHandler) { /* ... */ }
```

### POSIX [5]

```
void i_serialbyte(void) {  
    uint8_t rec = rs232_get();  
    msg_addTo(msg, rec);  
  
    if(msg_isComplete(msg)) {  
        buffer_ins(msgBuffer, msg);  
        msg = pool_getfree(msgPool);  
        pthread_create(&thread, attr,  
                      t_msghandler, NULL);  
    }  
    return;  
}
```

```
void t_msghandler(void* arg)  
{ /* ... */ }
```

#### ■ Explizite Aktivierung des Nachfolgers durch den Vorgänger

- Systemaufrufe: `ActivateTask` bzw. `pthread_create`

→ Planer stellt die richtige Reihenfolge sicher

#### ⚠ Absolute Sequentialisierung von Vorgänger und Nachfolger

- Erschwert die Umsetzung komplexer Abhängigkeitsszenarien

→ Auftragsorientiertes Ausführungsmodell (run-to-completion)





# Rangfolge durch den Austausch von Zeitsignalen

## POSIX

```
void i_serialbyte(void) {
    uint8_t rec = rs232_get();
    msg_addTo(msg, rec);

    if(msg_isComplete(msg)) {
        buffer_ins(msgBuffer, msg);
        msg = pool_getfree(msgPool);
        sem_post(&msg_sem);
    }
    return;
}

void t_msghandler(void* arg) {
    Message *cMsg = 0;
    InitHandler();

    while(1) {
        sem_wait(&msg_sem);
        cMsg = buffer_get(msgBuffer);
        msg_prepare(cMsg);
        handle(cMsg);
    }

    pthread_exit(NULL);
}
```

- Betriebssystemabstraktion:  
Semaphore (engl. *semaphore*)
  - `sem_wait()` wartet **blockierend** auf das Eintreten einer Abhängigkeit
  - `sem_post()` zeigt das Eintreten der Abhängigkeit an
- Prozessorientiertes Ausführungsmodell
  - Typ. in Verbindung mit sog. Do-While-Prozessen
  - Do  $\rightsquigarrow$  InitHandler()
  - While  $\rightsquigarrow$  Nachrichten verarbeiten
- Ermöglicht teilweise **nebenläufige Abarbeitung**
  - Ausführung von InitHandler(), bevor eine Nachricht ansteht





# Rangfolge durch Nachrichtenversand

Kombination aus Rangfolge und Datenaustausch (engl. *message passing*)

## AUTOSAR OS

```
Message msg,rcvMsg;
```

```
ISR(SerialByte) {  
    uint8_t rcv = rs232_get();  
    msg_addTo(&msg,rcv);
```

```
    if(msg_isComplete(&msg))  
        SendMessage(serialMsg,&msg);  
    return;  
}
```

```
TASK(MsgHandler) {  
    Message *cMsg = 0;  
    InitHandler();
```

```
    while(1) {  
        WaitEvent(msgEvent);  
        ClearEvent(msgEvent);  
        ReceiveMessage(serialMsg,  
                        &rcvMsg);  
        msg_prepare(&rcvMsg);  
        handle(&rcvMsg);  
    }  
    TerminateTask();  
}
```

- Übermittlung der Daten durch den Versand einer Nachricht
  - Vorgänger  $\rightsquigarrow$  `SendMessage()`
  - Nachfolger  $\rightsquigarrow$  `ReceiveMessage()`

- Verwaltung/Pufferung der Daten entfällt typischerweise
  - $\rightarrow$  Aufgabe des **Nachrichtendiensts**



AUTOSAR OS: Keine Rangfolge durch Nachrichtenversand

- `ReceiveMessage()` blockiert nicht
  - $\rightarrow$  Erfordert Kombination mit **Signalen** (engl. *events*)  $\rightsquigarrow$  Wird mit Nachrichtenversand gesetzt



## 1 Grundlagen

- Datenabhängigkeiten
- Nebenläufigkeit
- Abhängigkeits- und Aufgabengraphen
- Koordinierung

## 2 Effekte in Echtzeitsystemen

- Zeitliche Domänen
- Physikalisch und logische Ereignisse

## 3 Lösungsverfahren

- Analytische Koordinierung
- Konstruktive Koordinierung

## 4 Ablaufplanung

## 5 Zusammenfassung



# Restriktionen des periodischen Modells

Weitere Lockerung durch Aufhebung von A2 und A5 (vgl. IV-1/9)



Mathematische Ansätze zur zeitlichen Analyse periodischer Echtzeitsysteme bedingen häufig **starke Einschränkungen**:

~~A1~~ Alle Aufgaben sind periodisch

~~A2~~ **Alle Arbeitsaufträge können an ihren Auslösezeitpunkten eingeplant und ausgeführt werden**

**A3** Termine und Perioden sind identisch

**A4** Kein Arbeitsauftrag gibt die Kontrolle über den Prozessor ab

~~A5~~ **Alle Aufgaben sind unabhängig**<sup>4</sup>

**A6** Die Kosten durch Unterbrechungen, Ablaufplanung und Verdrängung sind vernachlässigbar

**A7** Alle Aufgaben verhalten sich voll-präemptiv

---

<sup>4</sup>D.h. die einzige gemeinsame Ressource ist die CPU und es existieren keine Einschränkungen hinsichtlich der Auslösezeiten der Arbeitsaufträge voneinander.



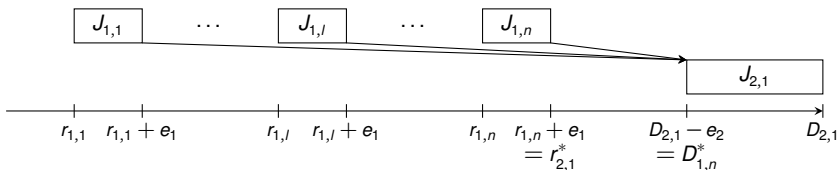
## Abhängigkeiten $\rightsquigarrow$ phasenverschobene Ausführung

Gerichtete Abhängigkeiten in das Planungsproblem aufnehmen

☞ Vorgehen analog zur Berechnung statischer Ablaufpläne

- Abhängigkeiten schränken den zeitlichen Ablauf ein (vgl. Folie 25)
- Umformulierung von **Auslösezeiten** und **Termine** so dass diese mit den Abhängigkeiten übereinstimmen [1]

■ Beispiel: **ISR (SerialByte)** und **TASK (MsgHandler)** (vgl. Folie 12)



- $J_{2,1}$  kann frühestens nach  $J_{1,n}$  starten  
     $\rightsquigarrow$  angepasste Auslösezeit des Nachfolgers  $r_{2,1}^* = \max_{1 \leq j \leq n} r_{1,j} + e_1$
- $J_{2,1}$  benötigt noch genügend Ausführungszeit  
     $\rightsquigarrow$  angepasster Termin des Vorgängers  $D_{1,n}^* = D_{2,1} - e_2$



- 1 Nachfolger  $J_i$  kann Ausführung erst mit Fertigstellung seiner Vorgänger beginnen


→ Modifizierung der Auslösezeit des Nachfolgers

$$r_i^* = \max \left\{ r_i, \left\{ r_j^* + e_j \mid J_j \rightarrow J_i \right\} \right\}$$


- 2 Die Vorgänger  $J_i$  müssen rechtzeitig fertig werden, so dass der Nachfolger seinen Termin einhalten kann

→ Modifizierung der Termine der Vorgänger

$$D_i^* = \min \left\{ D_i, \left\{ D_j^* - e_j \mid J_i \rightarrow J_j \right\} \right\}$$

 Anschließend erfolgt die Ablaufplanung mittels EDF

- EDF ist auch für derartige Systeme optimal (vgl. IV-2/23)
- Für Systeme mit statischen Prioritäten ungeeignet

 Vorgehen nur für einfache Abhängigkeiten geeignet

- Muster wie 2 von 3 Vorgängern erfordern angepasste Abbildungen





## 1 Grundlagen

- Datenabhängigkeiten
- Nebenläufigkeit
- Abhängigkeits- und Aufgabengraphen
- Koordinierung

## 2 Effekte in Echtzeitsystemen

- Zeitliche Domänen
- Physikalisch und logische Ereignisse

## 3 Lösungsverfahren

- Analytische Koordinierung
- Konstruktive Koordinierung

## 4 Ablaufplanung

## 5 Zusammenfassung



**Rangfolge**  $\leadsto$  gerichtete Abhängigkeiten

- resultieren oft aus Datenabhängigkeiten
- gerichtete Abhängigkeiten in nebenläufigen Ausführungsumgebungen erfordern Koordinierung

**Umsetzung gerichteter Abhängigkeiten**  $\leadsto$  Koordinierung

- wohlgeordneter Ablauf von Produzent und Konsument
- Übergang zwischen zeitlichen Domänen
- Implementierung gerichteter Abhängigkeiten

**implizit**  $\leadsto$  statische Ablauftabellen, Phasenverschiebung

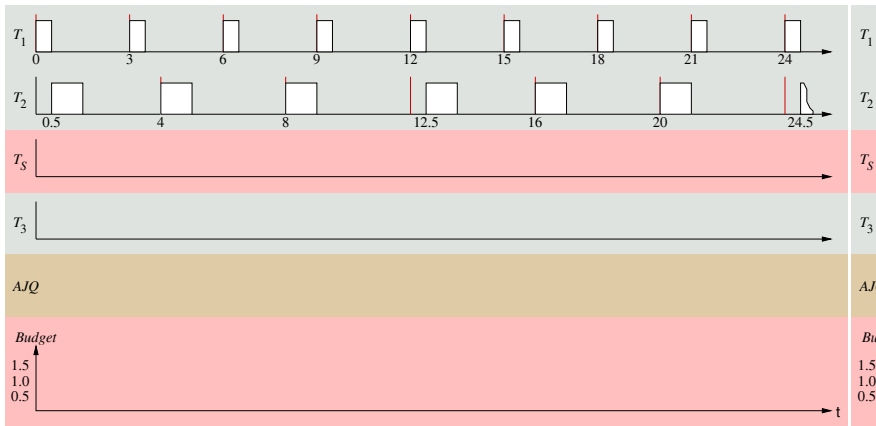
**explizit**  $\leadsto$  Aktivierung, Zeitsignale, Nachrichten

**Ablaufplanung** nutzt die Einschränkung des Ablaufverhaltens

- **Nachfolger**  $\leadsto$  modifizierte Auslösezeiten
- **Vorgänger**  $\leadsto$  modifizierte Termine



# Beispiel: SpSL



Genau genommen hat auch der SpSL-Zusteller seine Probleme!

- Die praktischen Probleme (vgl. PSS V-2/21) überwiegen hier jedoch die konzeptionellen
- <http://www.cs.fsu.edu/~baker/realtime/restricted/notes/servers.html>



- [1] Abdelzaher, T. F. ; Shin, K. G.:  
Combined Task and Message Scheduling in Distributed Real-Time Systems.  
In: *IEEE Transactions on Parallel and Distributed Systems* 10 (1999), Nr. 11, S. 1179–1191.  
<http://dx.doi.org/10.1109/71.809575>. –  
DOI 10.1109/71.809575
- [2] AUTOSAR:  
Specification of Operating System (Version 4.0.0) / Automotive Open System Architecture GbR.  
2009. –  
Forschungsbericht
- [3] Gomaa, H. :  
A software design method for real-time systems.  
In: *Communications of the ACM* 27 (1984), Nr. 9, S. 938–949.  
<http://dx.doi.org/10.1145/358234.358262>. –  
DOI 10.1145/358234.358262. –  
ISSN 0001–0782
- [4] Gomaa, H. :  
Structuring criteria for real time system design.  
In: *Proceedings of the 10th International Conference on Software Engineering (ICSE '88)*.  
New York, NY, USA : ACM Press, 1989. –  
ISBN 0–8186–1941–4, S. 290–301



- [5] IEEE:  
*ISO/IEC IEEE/ANSI Std 1003.1-1996 Information Technology — Portable Operating System Interface (POSIX®) — Part 1: System Application: Program Interface (API) [C Language].*  
IEEE, New York : IEEE, 1996. –  
784 S. –  
ISBN 1–55937–573–6
- [6] Liu, J. W. S.:  
*Real-Time Systems.*  
Englewood Cliffs, NJ, USA : Prentice Hall PTR, 2000. –  
ISBN 0–13–099651–3
- [7] OSEK/VDX Group:  
Operating System Specification 2.2.3 / OSEK/VDX Group.  
2005. –  
Forschungsbericht. –  
<http://portal.osek-vdx.org/files/pdf/specs/os223.pdf>, visited 2009-09-09



# EZS – Cheat Sheet

## Typographische Konvention

Der erste Index gibt die Aufgabe an (z.B.  $D_i$ ), der Zweite (optional) bezieht sich auf den Arbeitsauftrag (z.B.  $d_{i,j}$ ). Exponenten zeigen verschiedene Varianten einer Eigenschaft an (z.B.  $T^{HI}, T^{MED}, T^{LO}$ ). Funktionen beschreiben zeitlich variierende Eigenschaften (z.B.  $P(t)$ ).

## Eigenschaften

$t$  (Real-)Zeit  
 $d$  Zeitverzögerung (engl. delay)

## Strukturelemente

$E_i$  Ereignis (engl. event)  
 $R_i$  Ergebnis (engl. result)  
 $T_i$  Aufgabe (engl. task)  
 $J_{i,j}$  Arbeitsauftrag (engl. job) der Aufgabe  $T_i$

## Temporale Eigenschaften

Allgemein

$r_i$  Auslösezeitpunkt (engl. release time)  
 $e_i$  Maximale Ausführungszeit (WCET)  
 $D_i$  Relativer Termin (engl. deadline)  
 $d_i$  Absoluter Termin  
 $\omega_i$  Antwortzeit (engl. response time)  
 $\sigma_i$  Schlupf (engl. slack)

Periodische Aufgaben

$p_i$  Periode (engl. period)  
 $\phi_i$  Phase (engl. phase)

## Nicht-Periodische Aufgaben

$i_i$  Minimale Zwischenankunftszeit (engl. minimal interarrival-time)

## Aufgaben – Tupel

$T_p = (p, e, D, \phi)$  Periodische Aufgabe ohne Priorität (zeitgesteuert oder dynamische Taskpriorität),  $D = p$  und  $\phi = 0$  sind der Reihe nach optional

$T_i^S = (i_i, e_i, D_i)$  Nicht-periodische Aufgabe (Schreibweise mit  $i_i$ )

$T_i^S = ([r_i^{nach}; r_i^{vor}], e_i, D_i)$  Nicht-periodische Aufgabe (Schreibweise mit Auslöseintervall)

$J_{i,j} = (r_{i,j}, e_{i,j}, d_{i,j})$  Arbeitsauftrag

## Ablaufplanung

$P_i$  Priorität (engl. priority) der Aufgabe  $T_i$

$\Omega_i$  Prioritätsebenen (engl. number of priorities)

$h_{\Delta_i}$  Rechenzeitbedarf (engl. demand)

$u_{\Delta_i}$  CPU-Auslastung (engl. utilisation)

$U$  Absolute CPU-Auslastung

$H$  Hyperperiode (großer Durchlauf, engl. major cycle)

$f$  Rahmenlänge (kleiner Durchlauf, engl. minor cycle)

$I_i$  Intervall (engl. interval)

$\Delta_i$  Dichte (engl. density) von  $I_i$

## Zusteller

$T_{PS}$  Abfragender Zusteller (engl. polling server)

$T_{DS}$  Aufschiebbarer Zusteller (engl. deferable server)

$T_s$  Sporadischer Zusteller (engl. sporadic server)

$T_s$  Sporadischer Zusteller (engl. sporadic server)

$rt_i$  Wiederauffüllzeitpunkt (engl. replenishment time)

