

# Echtzeitsysteme

## Rangfolge

### Peter Ulbrich

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

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

14. Dezember 2015



## Gliederung

- 1 Rangfolge und gerichtete Abhängigkeiten
  - Datenabhängigkeiten
  - Nebenläufigkeit
  - Abhängigkeits- und Aufgabengraphen
  - Koordinierung
- 2 Umsetzung
  - Naive Implementierung
  - Physikalisch und logische Ereignisse
  - Implementierungsvarianten gerichteter Abhängigkeiten
- 3 Ablaufplanung
- 4 Zusammenfassung



## Fragestellungen

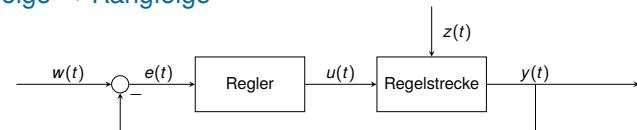
- Was bedeutet **Rangfolge**?
  - Was ist die Ursache von Rangfolge?
  - Wie beschreibt man Rangfolge?
- Wie kann **Rangfolge implementieren** werden?
  - Welche Implementierungsvarianten gibt es?
  - Welche Implikationen haben sie?
- Was bedeuten Rangfolgebeziehungen für die **Ablaufplanung**?



## Rangfolge (engl. precedence)

Abhängigkeit von Kontrollflüssen

- ☞ Ausführung von Arbeitsaufträgen unterliegt häufig einer **bestimmten Reihenfolge**  $\mapsto$  **Rangfolge**



- Beispiel: **Regelungsanwendung**
    - Signalverarbeitungsauftrag muss vor der Regelung gelaufen sein
  - Beispiel: **Kommunikationssystem**
    - Sendeauftrag muss vor Empfangsauftrag gelaufen sein
    - Empfangsauftrag muss vor Bestätigungsaufrag gelaufen sein
  - Beispiel: **Anfragesystem**
    - Eingabeauftrag muss vor Suchauftrag gelaufen sein
    - Suchauftrag muss vor Ausgabeauftrag gelaufen sein
- ⚠ Rangfolge ist oft in **Datenabhängigkeiten** begründet



# Datenabhängigkeit (engl. *data dependency*)

Abhängigkeit von konsumierbaren Betriebsmitteln

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

– Konsumierbares Betriebsmittel muss vor Inanspruchnahme zunächst bereitgestellt werden

- Abhängigkeit: Produzent → Konsument (seltener)

– Konsumierbare Betriebsmittel wird auf endlich viele **wiederverwendbare Betriebsmittel** abgebildet

- Produzent fordert ein wiederverwendbares Betriebsmittel an, welches vom Konsumenten später wieder freizugeben ist
- Beispiel: **begrenzter Puffer** (engl. *bounded buffer*)

# 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.

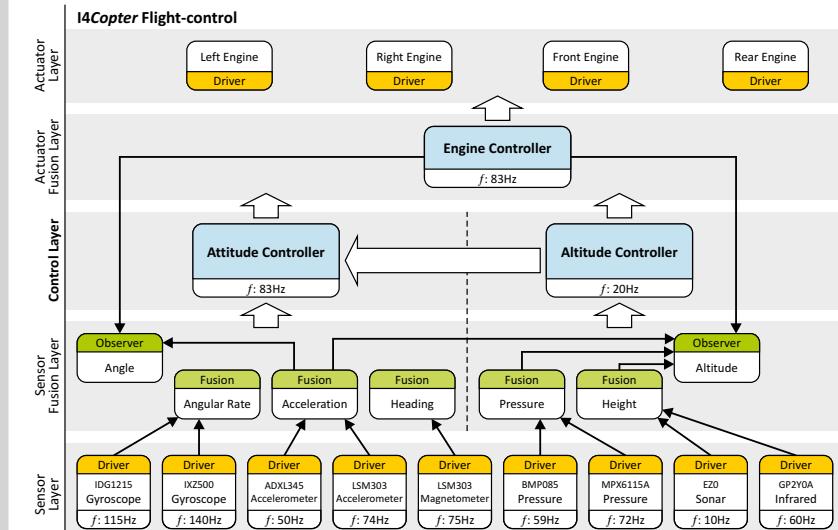
- Ereignisse sind **nebenläufig**, wenn keines Ursache des anderen ist
- Aktionen können nebenläufig ausgeführt werden, wenn keine das Resultat des anderen benötigt

- Beispiel eines nichtsequentiellen Programms:

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

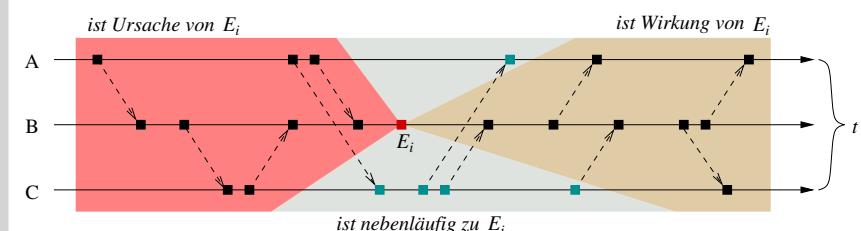
- Zeile 1 kann nebenläufig zu Zeile 2 ausgeführt werden
- Zeile 3 kann nebenläufig zu Zeile 4 ausgeführt werden

# Datenabhängigkeiten im I4Copter



# Kausalordnung

Nebenläufigkeit als relativistischer Begriff von Gleichzeitigkeit



- Relationen:** Ursache ↔ Wirkung ↔ Nebenläufigkeit

- Kausalkette von Ereignissen in Bezug zu einem Ereignis  $E_i$
- Bezogen auf Raum<sup>1</sup> und Zeit

Ein Ereignis  $E_i$  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>1</sup>A, B und C bezeichnen Ausführungsstränge auf einem Rechensystem.

## Kausalordnung (Forts.)

Rangfolge aus Gründen von Daten- und Zeitabhängigkeit

- Ein Arbeitsauftrag kann **nebenläufig** bearbeitet werden, wenn:
  - Im Allgemeinen ■ Er benötigt kein Ergebnis eines Anderen (vgl. VI/7)
    - 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” }  
“ist Wirkung von” } → **sequentiell** (verwirklicht vor/zur Laufzeit)  
“ist nebenläufig zu” → **parallel** (logisch/tatsächlich)



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



pu Echtzeitsysteme (WS 15/16) – Kapitel VI Rangfolge  
2 Rangfolge und gerichtete Abhängigkeiten – 2.2 Nebenläufigkeit

9/34

## Kausalordnung [6, S. 43]

Abhängigkeiten zwischen verschiedenen Arbeitsaufträgen

- Die **Kausalordnung** wird durch eine **Vorgängerrelation** (engl. *precedence relation*) beschrieben:

- $J_i \rightarrow J_k$ : Job  $J_i$  ist **Vorgänger** (engl. *predecessor*) von  $J_k$
- Ausführung des **Nachfolgers** (engl. *successor*)  $J_k$  erfordert die Fertigstellung des Vorgängers  $J_i$

- Beispiel auf Folie 10:

- **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



pu Echtzeitsysteme (WS 15/16) – Kapitel VI Rangfolge  
2 Rangfolge und gerichtete Abhängigkeiten – 2.3 Abhängigkeits- und Aufgabengraphen

11/34

## 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** → gemeinsamer Puffer **msgBuffer**

**Rangfolge** → Wann kann die Nachricht verarbeitet werden?

???

→ Wann wird **TASK(MsgHandler)** aktiv?

pu Echtzeitsysteme (WS 15/16) – Kapitel VI Rangfolge  
2 Rangfolge und gerichtete Abhängigkeiten – 2.3 Abhängigkeits- und Aufgabengraphen

10/34

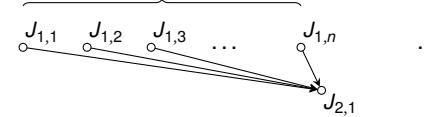
## Beispiel: Serieller Empfang von Nachrichten (Forts.)

Abhängigkeitsbeziehungen der einzelnen Arbeitsaufträge

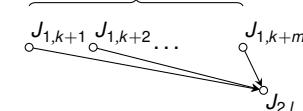
**Aufgabe  $T_1$**  Empfang einzelner Bytes → Jobs  $J_{1,1}, J_{1,2}, \dots$

**Aufgabe  $T_2$**  Bearbeitung der Nachrichten → Jobs  $J_{2,1}, J_{2,2}, \dots$

1. Nachricht = n Bytes



I. Nachricht = m Bytes



- **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  $J_{2,l}$

pu Echtzeitsysteme (WS 15/16) – Kapitel VI Rangfolge  
2 Rangfolge und gerichtete Abhängigkeiten – 2.3 Abhängigkeits- und Aufgabengraphen

12/34

# Koordinierung (engl. coordination)

Gerichtete Abhängigkeiten analytisch/konstruktiv behandeln

- **Statisch durch Einplanung**  $\leadsto$  analytische Verfahren
  - Ablaufpläne berücksichtigen Rangfolgen und Datenabhängigkeiten
    - *à priori* Wissen  $\rightarrow$  periodische Aufgaben
  - Arbeitsaufträge laufen komplett durch (engl. *run to completion*)
    - Warten weder ex- noch implizit, dürfen jedoch verdrängt werden
  - $\rightarrow$  Ergebnis ist ein System von ausschließlich **einfachen Aufgaben**
- **Dynamisch durch Kooperation**  $\leadsto$  konstruktive Verfahren
  - Synchronisationspunkte in den Programmen explizit machen
    - d.h., **Zeitsignale austauschen**  $\rightarrow$  Semaphor
  - Arbeitsaufträge sind Produzenten/Konsumenten von Ereignissen
    - physische Ereignisse** von den kontrollierten Objekten
    - logische Ereignisse** von anderen Arbeitsaufträgen
  - $\rightarrow$  Ergebnis ist ein System von (ggf. vielen) komplexen Aufgaben



## Naive Implementierung

Statische Anordnung im Quelltext: Rangfolge  $\leftrightarrow$  Programmsequenz

- ☞ **Statische Codierung** gerichteter Abhängigkeiten im Quelltext
  - Vorgänger und Nachfolger sind **unveränderlich** und *à priori* bekannt
  - $\rightarrow$  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!



## Gliederung

- 1 Rangfolge und gerichtete Abhängigkeiten
  - Datenabhängigkeiten
  - Nebenläufigkeit
  - Abhängigkeits- und Aufgabengraphen
  - Koordinierung
- 2 Umsetzung
  - Naive Implementierung
  - Physisch und logische Ereignisse
  - Implementierungsvarianten gerichteter Abhängigkeiten
- 3 Ablaufplanung
- 4 Zusammenfassung



## 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 Nachrichtenempfangs:

- Empfang  $\leadsto$  Nicht-periodische Aufgabe  $T_1 = (i_1, e_1)$
- Verarbeitung  $\leadsto$  Nicht-periodische Aufgabe  $T_2 = (i_2, e_2)$
- Empfang mehrere Bytes pro Nachricht  $\leadsto i_1 \ll i_2$
- Verarbeitung ist komplexer als deren Empfang  $\leadsto e_2 \gg e_1$

### Naive Implementierung verletzt zeitlichen Domänen

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

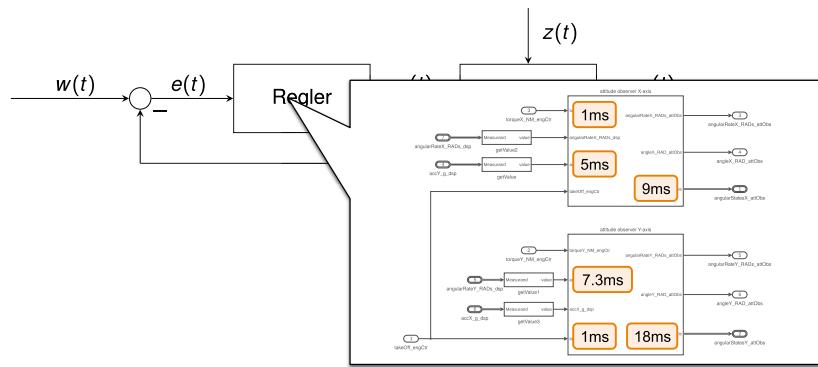
### Gerichtete Abhängigkeiten $\rightarrow$ Hinweis auf **versch.** zeitliche Domänen

- $\rightarrow$  Aufgaben mit dedizierten Auslöseereignissen und zeitlichen Parametern



## Zeitliche Domänen

Am Beispiel des i4Copters



- Signaldatenverarbeitung im Fokus
  - Scheinbar einfache Funktion → 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. VI/6)
  - 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>2</sup>
    - Generierung von Zwischenwerten kompensiert langsam produzierende Konsumenten
  - **Letzter Wert genügt** (engl. *last is best*) → beidseitig
    - Verzicht auf explizite Abstimmung (**simpel**)
    - **Alter unterliegt gewissen Schwankungen**



## Übergang zwischen zeitlichen Domänen (Forts.)

Produzenten und Konsumenten werden mit unterschiedlichen Raten aktiviert

- 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

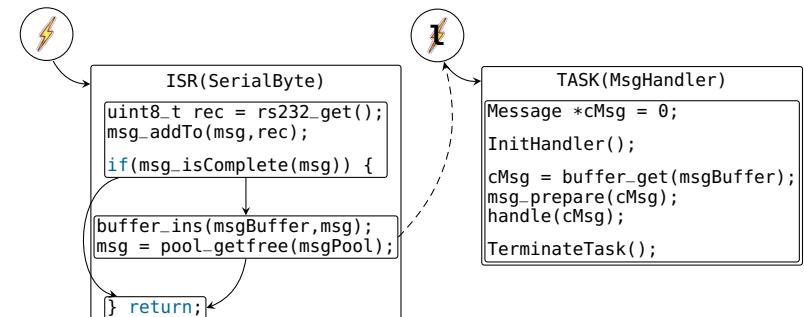
■ Entkopplung zeitlicher Domänen durch **logische Ereignisse**



## 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



## Implementierungsvarianten gerichteter Abhängigkeiten

Rangfolge sicherstellen, ohne eine zeitliche Kopplung vorwegzunehmen

- 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 (vgl. VI/22)  
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 (vgl. VI/23 ff)

## 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) { /* ... */ }
```

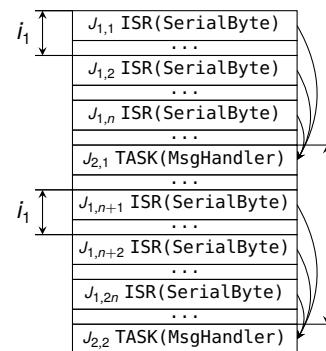
### POSIX [5]

```
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
  - **Ergebnisorientiertes Ausführungsmodell** (*run-to-completion*)

## Rangordnung mittels statischer Ablaufplanung

- Eingabe für die **statische Ablaufplanung** (s. Folie IV-3/18 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  $\phi_m: \phi_m = \max_{J_{i,j} \rightarrow J_{m,n}} r_{i,j} + \omega_{i,j}$

⚠ Einhaltung der Phase wird zur Laufzeit **nicht überwacht**  
→ Laufzeitüberschreitungen  $\leadsto$  ggf. Verletzungen der Rangfolge

## Rangfolge durch den Austausch von Zeitsignalen

Der Konsument wartet explizit auf das Eintreten der Abhängigkeit

### 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
- **Prozessororientiertes Ausführungsmodell**
  - Typ. in Verbindung mit sog. **Do-While-Prozessen**
  - **Do**  $\leadsto$  `InitHandler()`
  - **While**  $\leadsto$  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, rec);  
  
    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  $\leadsto$  SendMessage()  
Nachfolger  $\leadsto$  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*)  $\leadsto$  Wird mit Nachrichtenversand gesetzt

# 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>3</sup>

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

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

<sup>3</sup>D.h. die einzige gemeinsame Ressource ist die CPU und es existieren keine Einschränkungen hinsichtlich der Auslösezeiten der Arbeitsaufträge voneinander.

# Gliederung

## 1 Rangfolge und gerichtete Abhängigkeiten

- Datenabhängigkeiten
- Nebenläufigkeit
- Abhängigkeits- und Aufgabengraphen
- Koordinierung

## 2 Umsetzung

- Naive Implementierung
- Physikalisch und logische Ereignisse
- Implementierungsvarianten gerichteter Abhängigkeiten

## 3 Ablaufplanung

## 4 Zusammenfassung



# Abhängigkeiten $\leadsto$ phasenverschobene Ausführung

Gerichtete Abhängigkeiten in das Planungsproblem aufnehmen

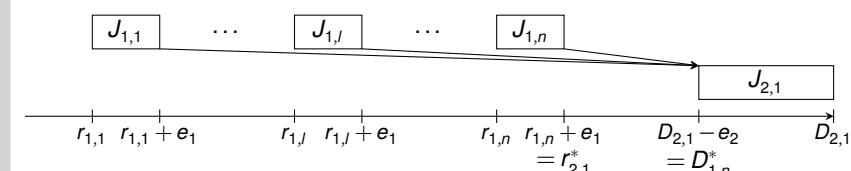


Vorgehen analog zur Berechnung statischer Ablaufpläne

- Abhängigkeiten schränken den zeitlichen Ablauf ein (vgl. VI/22)  
 $\rightarrow$  Umformulierung von **Auslösezeiten** und **Termine** so dass diese mit den Abhängigkeiten übereinstimmen [1]



■ Beispiel: `ISR(SerialByte)` und `TASK(MsgHandler)` (vgl. VI/10)



■ J<sub>2,1</sub> kann frühestens nach J<sub>1,n</sub> starten

$\leadsto$  angepasste Auslösezeit des Nachfolgers  $r_{2,1}^* = \max_{1 \leq j \leq n} r_{1,j} + e_1$

■ J<sub>2,1</sub> benötigt noch genügend Ausführungszeit

$\leadsto$  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

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

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

→ Modifizierung der Termine der Vorgänger

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

☞ Anschließend erfolgt die Ablaufplanung mittels EDF

- EDF ist auch für derartige Systeme optimal (vgl. IV-2/19)
- Für Systeme mit statischen Prioritäten ungeeignet

⚠️ Vorgehen nur für einfache Abhängigkeiten geeignet

- Muster wie 2 von 3 Vorgängern erfordern angepasste Abbildungen



## Resümee

**Rangfolge**  $\rightsquigarrow$  gerichtete Abhängigkeiten

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

**Umsetzung gerichteter Abhängigkeiten**  $\rightsquigarrow$  Koordinierung

- wohlgeordneter Ablauf von Produzent und Konsument
- Übergang zwischen zeitlichen Domänen
- Implementierung gerichteter Abhängigkeiten
  - implizit  $\rightsquigarrow$  statische Ablauftabellen, Phasenverschiebung
  - explizit  $\rightsquigarrow$  Aktivierung, Zeitsignale, Nachrichten

**Ablaufplanung** nutzt die Einschränkung des Ablaufverhaltens

- Nachfolger  $\rightsquigarrow$  modifizierte Auslösezeiten
- Vorgänger  $\rightsquigarrow$  modifizierte Termine



- 1 Rangfolge und gerichtete Abhängigkeiten

- Datenabhängigkeiten
- Nebenläufigkeit
- Abhängigkeits- und Aufgabengraphen
- Koordinierung

- 2 Umsetzung

- Naive Implementierung
- Physikalisch und logische Ereignisse
- Implementierungsvarianten gerichteter Abhängigkeiten

- 3 Ablaufplanung

- 4 Zusammenfassung



## Literaturverzeichnis

- [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. –  
ISSN 1045-9219

- [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 11th International Conference on Software Engineering (ICSE '89)*.  
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

