

# Echtzeitsysteme

## Rangfolge

Lehrstuhl Informatik 4

15. Dezember 2011

# Gliederung

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

# Fragestellungen

- Was bedeutet **Rangfolge**?
  - Was ist die Ursache von Rangfolge?
  - Wie beschreibt man Rangfolge?
  
- Wie kann man **Rangfolge implementieren**?
  - Welche Implementierungsvarianten gibt es?
  - Welche Implikationen haben sie?
  
- Wie geht man in der **Ablaufplanung** mit Rangfolgebeziehungen um?

# Gliederung

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

# Rangfolge (engl. *precedence*)

## Abhängigkeit von Kontrollflüssen

Arbeitsaufträge können gezwungen sein, in einer ganz bestimmten Reihenfolge ausgeführt werden zu müssen

- Beispiel Radarüberwachungsanlage ...
  - Signalaufbereitungsauftrag muss vor Nachführauftrag gelaufen sein
- Beispiel Kommunikationssystem ...
  - Sendeauftrag muss vor Empfangsauftrag gelaufen sein
  - Empfangsauftrag muss vor Bestätigungsauftrag gelaufen sein
- Beispiel Anfragesystem ...
  - Eingabeauftrag muss vor Authentifizierungsauftrag gelaufen sein
  - Authentifizierungsauftrag muss vor Suchauftrag gelaufen sein
  - Suchauftrag muss vor Ausgabeauftrag gelaufen sein

 die Rangfolge ist oft in Datenabhängigkeiten begründet

# Datenabhängigkeit (engl. *data dependency*)

Abhängigkeit von konsumierbaren Betriebsmitteln

Arbeitsaufträge brauchen zum Ablauf ggf. **konsumierbare Betriebsmittel**

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

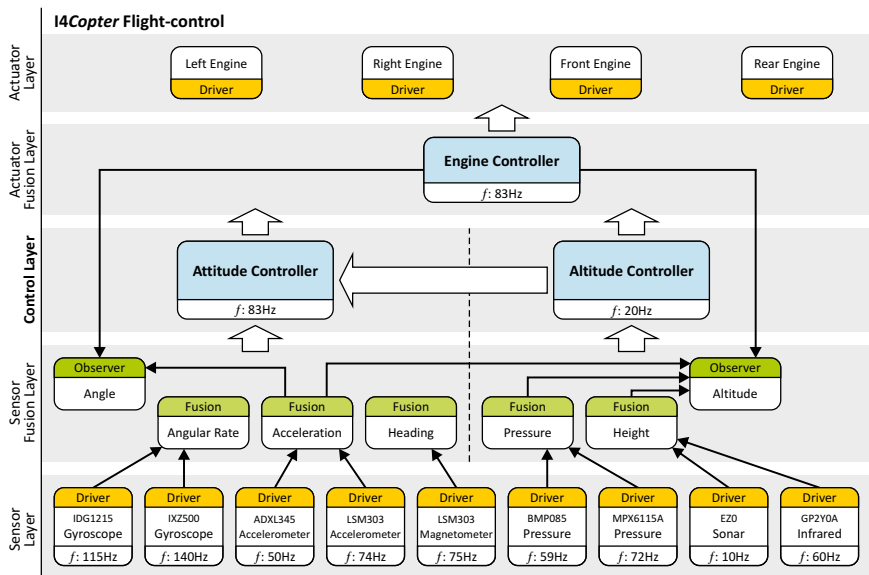
**Produzent** kann beliebig viele davon erzeugen

**Konsument** zerstört sie wieder bei Inanspruchnahme

Produzent und Konsument sind voneinander abhängige Entitäten

- zwischen ihnen besteht eine **gerichtete Abhängigkeit**
- der Konsument vom Produzenten ...
  - weil ein konsumierbares Betriebsmittel erst bereitgestellt werden muss, um es in Anspruch nehmen zu können
- der Produzent vom Konsumenten ...
  - weil konsumierbare Betriebsmittel auf endlich viele wiederverwendbare Betriebsmittel abgebildet werden
  - weil der Produzent dazu erst ein wiederverwendbares Betriebsmittel anfordern muss, das vom Konsumenten später wieder freizugeben ist
  - Beispiel: **begrenzter Puffer** (engl. *bounded buffer*)

# Datenabhängigkeiten im I4Copter



# Nebenläufige Aktivitäten

## Nichtsequentielles Programm

**Nebenläufigkeit** (engl. *concurrency*) bezeichnet das Verhältnis von nicht kausal abhängigen Ereignissen, die sich also nicht beeinflussen

- Aktionen können nebenläufig ausgeführt werden, wenn keine das Resultat des anderen benötigt

```
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

**Kausalität** (lat. *causa*: Ursache) ist die Beziehung zwischen **Ursache** und **Wirkung**, d.h., die ursächliche Verbindung zweier Ereignisse

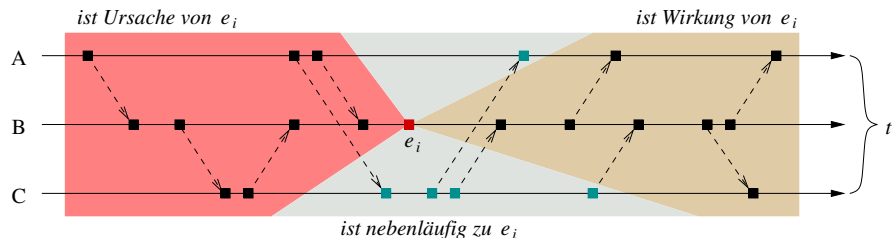
- Ereignisse sind nebenläufig, wenn keines Ursache des anderen ist



# Kausalordnung

Nebenläufigkeit als relativistischer Begriff von Gleichzeitigkeit

Relationen „ist Ursache von“, „ist Wirkung von“, „ist nebenläufig zu“:



- ein Ereignis **ist nebenläufig zu** einem anderen, wenn es im **Anderswo** des anderen Ereignisses liegt
  - d.h., weder in der Zukunft noch in der Vergangenheit des anderen
- das Ereignis ist weder Ursache oder Wirkung des anderen Ereignisses

# Kausalordnung (Forts.)

Rangfolge aus Gründen von Daten- und Zeitabhängigkeit

„ist Ursache von“  
„ist Wirkung von“  
„ist nebenläufig zu“

}  $\leadsto$  **Sequentialisierung** von Ereignissen/Aktionen

Aktionen können im **Echtzeitbetrieb** nebenläufig stattfinden, wenn ...

- keine das Resultat der anderen benötigt (s. Folie VI/8) ✓
- keine die (strikten) Zeitbedingungen der anderen verletzt
  - Zeitpunkte dürfen nicht bzw. nur selten verpasst werden
  - Zeitintervalle dürfen nicht bzw. nur begrenzt zeitlich gedehnt werden
    - Abstand zwischen Ursache (Ereigniszeitpunkt) und Wirkung (Termin)

... Abhängigkeiten hingegen erfordern das **Herstellen von Gleichzeitigkeit**

- z.B. durch den Austausch von Zeitsignalen (s. Folie VI/12)
  - implizit** im Falle analytischer Koordinierung
  - explizit** im Falle konstruktiver Koordinierung

# Abhängigkeits- und Aufgabengraphen [4, S. 43]

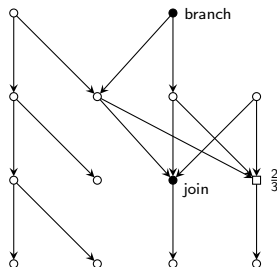
Notationen für Abhängigkeiten zwischen verschiedenen Arbeitsaufträgen

Die Kausalordnung ist eine Halbordnung und wird durch eine **Vorgängerrelation** (engl. *precedence relation*)  $\rightarrow$  beschrieben:

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

Graph  $\mathcal{G} = (\mathcal{J}, \rightarrow)$  dient als Beschreibung der Vorgängerrelation:

- Knoten sind Arbeitsaufträge, Pfeile sind Abhängigkeiten



- **Abhängigkeitsgraph** (engl. *precedence graph*)
  - beschreibt nur die Vorgängerrelation
- **Aufgabengraph** (engl. *task graph*)
  - beschreibt weiter Abhängigkeitstypen
    - zeitliche Abhängigkeiten
    - UND/ODER-Vorgängerrelationen
    - bedingte Verzweigungen
    - ...

# Koordinierung (engl. *coordination*)

Gerichtete Abhängigkeiten analytisch/konstruktiv behandeln

durch **Einplanung**  $\leadsto$  analytische Verfahren

- Ablaufpläne berücksichtigen Rangfolgen und Datenabhängigkeiten
  - **à priori Wissen**  $\mapsto$  periodische Aufgaben
- Arbeitsaufträge laufen komplett durch (engl. *run to completion*)
  - sie warten weder ex- noch implizit, dürfen jedoch verdrängt werden
- Ergebnis ist ein System von ausschließlich einfachen Aufgaben

durch **Kooperation**  $\leadsto$  konstruktive Verfahren

- Synchronisationspunkte in den Programmen explizit machen
  - d.h., **Zeitsignale austauschen**  $\mapsto$  Semaphore
- 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

# Gliederung

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

# Beispiel: Serieller Empfang von Nachrichten

Implementierung orientiert sich an OSEK OS [5] bzw. AUTOSAR OS [2]

Die Nachrichtenverarbeitung besteht aus zwei getrennten Aufgaben:

**Empfang** Abholen einzelner Bytes und Zusammensetzen von Nachrichten

**Verarbeitung** Nachricht vorbereiten und Behandlung aktivieren

## Empfang

```
Pool *msgPool; Buffer *msgBuffer; Message *msg;

ISR(SerialByte) {
    unsigned char 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?

???

- Wann wird TASK(MsgHandler) aktiv?

# Beispiel: Serieller Empfang von Nachrichten (Forts.)

Abhängigkeitsbeziehungen der einzelnen Arbeitsaufträge

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

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



- **keine Abhängigkeiten** zwischen den einzelnen Jobs von  $T_1$  und  $T_2$ 
  - auch wenn der Termin  $D_{1,1}$  die Fertigstellung von  $J_{1,1}$  vor dem Beginn von Job  $J_{1,2}$  erzwingt:  $D_{1,1} \leq r_{1,2}$
- die Jobs  $J_{1,1}, \dots, J_{1,n}$  ermöglichen aber die Ausführung von  $J_{2,1}$ 
  - erst wenn die Nachricht komplett ist, kann sie verarbeitet werden  
 $\rightsquigarrow$  die Jobs  $J_{1,1}, \dots, J_{1,n}$  sind Vorgänger von  $J_{2,1}$
- endgültige Abhängigkeitsbeziehungen erst zur Laufzeit bekannt
  - Nachrichten können unterschiedlich viele Bytes umfassen  
 $\rightsquigarrow$  unterschiedlich viele Vorgänger von  $J_{2,1}$  und  $J_{2,l}$

# Naive Implementierung

Statische Anordnung im Quelltext: Rangfolge  $\mapsto$  Reihenfolge von Unterprogrammen

Gerichtet Abhängigkeiten können statisch im Quelltext kodiert werden:

- falls Vorgänger und Nachfolger à priori bekannt und fix sind
- ↪ Behandlung wird nur aufgerufen, falls die Nachricht vollständig ist

```
Message *msg;

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


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

        msg_prepare(currentMsg);
        handle(currentMsg);

        msg_clear(msg);
    }
}
```

Die Implementierung wird so sichtbar vereinfacht:

- nur ein Aktivitätsträger
- Rangfolge ist unmittelbar ablesbar und muss nicht explizit geregelt werden
- keine Pufferung notwendig

 Allerdings hat diese Variante auch gravierende Nachteile!



# Nachteile implizit kodierter Abhängigkeiten

- die statische Sequentialisierung **verletzt zeitliche Domänen**
  - innerhalb einer zeitlichen Domäne ist das zeitliche Verhalten bekannt
    - unterschiedliche zeitliche Domänen besitzen oft auch verschiedene auslösende Ereignisse mit unterschiedlichen zeitlichen Eigenschaften  
~> sie sind daher auch Kandidaten für verschiedene Aufgaben
  - im betrachteten Beispiel existieren folgende zeitliche Domänen:
    - Empfang** ~> z.B. nicht-periodische Aufgabe  $T_1 = (i_1, e_1)$
    - Verarbeitung** ~> z.B. nicht-periodische Aufgabe  $T_2 = (i_2, e_2)$
- Beziehung zwischen diesen zeitlichen Domänen:
  - Empfang mehrere Bytes pro Nachricht  $\mapsto i_1 < i_2$
  - Verarbeitung ist komplexer als deren Empfang  $\mapsto e_2 > e_1$
- die naive Implementierung **verschmilzt zeitlichen Domänen**
  - Ergebnis ist eine Aufgabe  $T'_1 = (p_1, e_1 + e_2)$
  - das ist **unrealistisch**, schließlich wird  $T_2$  weniger häufig aktiviert



gerichtete Abhängigkeiten deuten auf verschiedene zeitliche Domänen

# Übergang zwischen zeitlichen Domänen

Produzenten und Konsumenten werden mit unterschiedlichen Raten aktiviert

Innerhalb eines Echtzeitsystems können verschiedene zeitliche Domänen existieren (s. Folie VI/7, Beispiel I4Copter).

☞ gerichtete Abhängigkeiten erfordern ihre **Angleichung**

- die produzierten Daten müssen ...
  - in einem gemeinsamen Puffer zwischengespeichert werden
  - für die weitere Verarbeitung fusioniert und gefiltert werden
- abhängig von den zeitlichen Eigenschaften dieser Domänen
  - **Puffergröße** hängt von der Rate vom Produzent/Konsument ab
  - der Fusions- bzw. Filteralgorithmus nutzt eine **Vorausschau** (engl. *lookahead*) des Produzenten im Vergleich zum Konsumenten

☞ eine Verschmelzung zeitlich identischer Domänen ist möglich

- stellt aber immer noch eine Optimierung dar
- ↪ die naive Implementierung nimmt diese Optimierung vorweg
- auch wenn die zeitlichen Domänen verschieden sind

☞ nutze **logische Ereignisse**, um zeitliche Domänen zu entkoppeln

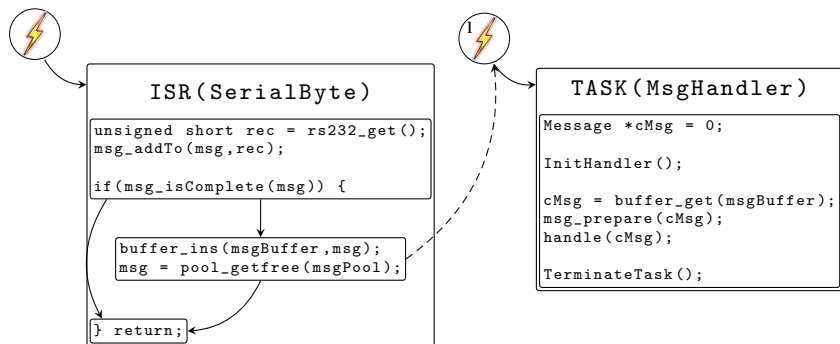
# Physikalische und logische Ereignisse

**physikalische Ereignisse** resultieren aus Zustandsänderungen der Umwelt

- wenn die serielle Schnittstelle den Empfang eines Byte anzeigt
- ↳ infolgedessen wird eine Unterbrechung auslöst

**logische Ereignisse** ruft die Echtzeitanwendung selbst hervor

- wenn eine Nachricht vollständig empfangen wurde
- ↳ das logische Ereignis entkoppelt Empfang und Verarbeitung zeitlich



# Implementierungsvarianten gerichteter Abhängigkeiten

Rangfolge sicherstellen, ohne eine zeitliche Kopplung vorwegzunehmen

Ziel ist die Herstellung der Rangfolge, ohne die zeitliche Nähe durch eine entsprechende Anordnung im Quelltext zu erzwingen:

**ohne Koordinierung**  $\leadsto$  Rangfolge bewusst vernachlässigen

- oft reicht es aus, dass Daten einfach aktuell sind

**analytische Koordinierung**  $\leadsto$  mit Hilfe der Ablaufplanung

- nur für Abhängigkeiten zwischen **periodische Aufgaben** anwendbar

**Taktsteuerung** geeignete Anordnung der Jobs in der Ablaufabelle

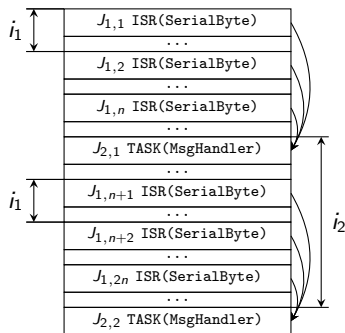
**Vorrangsteuerung** erreichte Anordnung durch Phasenversatz

**konstruktive Koordinierung**  $\leadsto$  mit Hilfe einseitiger Synchronisation

- für **nicht-periodischen Aufgaben**  $\leadsto$  unumgänglich
- in **zeitgesteuerten Systemen**  $\leadsto$  unmöglich
- für Synchronisation existieren eine Vielzahl von Möglichkeiten
  - z.B. die direkte Aktivierung von `TASK(MsgHandler)`, wenn eine Nachricht vollständig empfangen wurde
  - oder den expliziten Austausch von Signalen, um den Empfang der Nachricht anzuzeigen, u.U. verbunden mit der Übertragung von

# Analytische Umsetzung der Rangordnung

Eingabe für die statische Ablaufplanung (s. Folie IV-3/18 ff.) ist ein Abhängigkeits- oder Aufgabengraph (s. Folie VI/11). Die erzeugte Ablauftabelle muss die entsprechenden Randbedingungen einhalten.



- überführe nicht-periodische Aufgaben  $T_1$  und  $T_2$  (s. Folie VI/17) in entsprechende periodische Aufgaben
  - Periode  $p_n =$  Zwischenankunftszeit  $i_n$
- ordne Jobs nach den Abhängigkeiten an
  - $r_{i,j} + e_i \leq r_{n,m} \Leftrightarrow J_{i,j} \mapsto J_{n,m}$
- phasenverschobene Ausführung von  $J_{m,n}$  in vorranggesteuerten System ist analog
  - Rangfolge impliziert passende Phase  $\phi_m$ :  

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

☞ Einhaltung dieser Phase wird zur Laufzeit nicht überwacht!

- Laufzeitüberschreitungen führen u.U. auch zu Verletzungen der Rangfolge!

# Rangfolge durch Bereitstellung des Nachfolgers

## AUTOSAR OS [2]

```
ISR(SerialByte) {
    unsigned char 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 [3]

```
void i_serialbyte(void) {
    unsigned char 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) { /* ... */ }
```

- die Einplanung/Einlastung des Nachfolgers durch dessen Aktivierung in einem der Vorgänger ermöglichen
  - obige Beispiele: Systemaufrufe `ActivateTask` bzw. `pthread_create`
  - der Planer stellt automatisch die richtige Reihenfolge sicher
- **Nachteil:** komplette Sequentialisierung von Vorgänger u. Nachfolger
  - auch wenn dies nicht unbedingt erforderlich wäre
  - erschwert die Umsetzung komplexer Abhängigkeitsszenarien
    - $J_{1,1} \mapsto J_{2,1} \mapsto J_{1,1}$  wäre beispielsweise nicht implementierbar

# Rangfolge durch den Austausch von Zeitsignalen

Der Konsument wartet explizit auf das Eintreten der Abhängigkeit

## POSIX

```
void i_serialbyte(void) {
    unsigned char 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();

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

    pthread_exit(NULL);
}
```

- Betriebssystemabstraktion: der **Semaphor** (engl. *semaphore*)
  - **sem\_wait()** wartet **blockierend** auf das Eintreten einer Abhängigkeit
  - **sem\_post()** zeigt das Eintreten der Abhängigkeit an
- häufig in Verbindung mit sog. **Do-While-Prozessen**
  - siehe `t_msghandler()`
    - Do**  $\rightsquigarrow$  `InitHandler()`
    - While**  $\rightsquigarrow$  Nachrichten verarbeiten
- ermöglicht eine **teilweise nebenläufige Abarbeitung** der beteiligten Jobs
  - `InitHandler()` kann ausgeführt werden, bevor eine Nachricht zur Verarbeitung ansteht

# Nachrichtenversand (engl. *message passing*)

Kombination aus Rangfolge und Datenaustausch

## AUTOSAR OS

```

Message msg,rcvMsg;

ISR(SerialByte) {
    unsigned char rcv = rs232_get();
    msg_addTo(&msg,rcv);

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

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

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

    TerminateTask();
}

```

- Übermittlung des Zwischenergebnisse durch den Versand einer Nachricht
  - Vorgänger  $\rightsquigarrow$  `SendMessage()`
  - Nachfolger  $\rightsquigarrow$  `ReceiveMessage()`
- eigenhändige Verwaltung/Pufferung der Daten entfällt unter Umständen
  - $\rightsquigarrow$  oft Aufgabe des **Kommunikationssystems**
- Besonderheit in AUTOSAR OS: keine Rangfolge durch Nachrichtenversand
  - $\rightsquigarrow$  `ReceiveMessage()` blockiert nicht
    - erfordert Kombination mit Signalen
    - **Ereignisse** (engl. *events*) in AUTOSAR
    - ein zur Nachricht gehörendes Ereignis, wird bei ihrem Versand gesetzt



# Gliederung

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

# Weitere Lockerung der Restriktionen

Aufhebung der Einschränkungen A2 und A5, A4 bleibt weiter bestehen

Mathematische Ansätze zur Analyse periodischer Echtzeitsysteme schränken solche Systeme häufig stark ein:

~~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 voneinander, d.h. die einzige gemeinsame Ressource ist die CPU und es existieren keine Einschränkungen hinsichtlich der Auslösezeiten der Arbeitsaufträge.~~

A6 Der Overhead durch Unterbrechungen, Ablaufplanung oder Verdrängung ist vernachlässigbar.

A7 Alle Aufgaben verhalten sich voll-präemptiv.

# Abhängigkeiten $\rightsquigarrow$ phasenverschobene Ausführung

Gerichtete Abhängigkeiten durch eine Modifikation des Planungsproblems auflösen

Verfahre analog zur Berechnung statischer Ablaufpläne (s. Folie VI/21):

- Abhängigkeiten schränken den zeitlichen Ablauf ein
- $\rightsquigarrow$  formuliere zeitliche Kenngrößen so, dass sie mit der Abhängigkeiten der Halbordnung übereinstimmen [1]
- ① der Nachfolger  $J_i$  kann seine Ausführung erst dann beginnen, wenn seine Vorgänger fertiggestellt wurden

$\rightsquigarrow$  modifiziere die Auslösezeit des Nachfolgers

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

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

$\rightsquigarrow$  modifiziere die Termine der Vorgänger

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

☞ anschließend erfolgt die Ablaufplanung mit EDF

- EDF ist auch für derartige Systeme optimal (s. Folie IV-2/24)
- für Systeme mit statischen Prioritäten ist die Sache kniffliger ...

# Gliederung

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

# Resümee

**Rangfolge**  $\leadsto$  gerichtete Abhängigkeiten

- resultieren oft aus Datenabhängigkeiten
- Abhängigkeitsgraphen und Aufgabengraphen
- 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

# 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] 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

# Literaturverzeichnis (Forts.)

- [4] LIU, J. W. S.:  
*Real-Time Systems*.  
Prentice-Hall, Inc., 2000. –  
ISBN 0-13-099651-3
  
- [5] 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