

# ***Echtzeitsysteme***

## 5. Programmieren mit Zeit

*Prof. Dr. Roland Dietrich*

- Zugriff auf Zeit
  - damit "verstrichene Zeit" gemessen werden kann
  - damit Prozesse bis zu bestimmten (zukünftigen) Zeitpunkten verzögert werden können
  - damit "timeouts" programmiert werden können
    - Reaktion auf innerhalb eines Zeitraums nicht eingetretene Ereignisse
- Spezifikation von Zeitanforderungen
  - Ausführungsraten (z.B. "alle 3 ms")
  - Deadlines (z.B. "spätestens um 12:00 Uhr", "Spätestens nach 5 s")
- Erfüllen von Zeitanforderungen
  - Die Tasks eines Echtzeitsystems so planen, dass alle spezifizierten Zeitanforderungen erfüllt sind
  - Echtzeitplanung, siehe später!

- Was ist Zeit?

*Was also ist "Zeit"?*

*Wenn mich niemand danach fragt, weiß ich es.*

*Will ich es einem Fragenden erklären, weiß ich es nicht.*

*St. Augustinus*

- Antworten von Philosophen

- Kant

- Zeit ist eine Kategorie, die es uns erlaubt, Ereignisse in einer Reihenfolge (vorher-/nachher-Beziehung) anzuordnen

- Platonisten

- Zeit ist eine grundlegende Eigenschaft der Natur
    - Ohne Anfang, kontinuierlich, ohne Ende
    - Unser Zeitbegriff bildet Ereignisse in diesen absoluten Zeitbezug ab

- Reduktionisten

- Die Zeit ergibt sich aus Ereignissen
    - keinen Zeit-Fortschritt ohne Ereignisse
    - Regelmäßige Ereignisse ermöglichen Zeitmessung (Sonne, Atome)

- Antwort der Physik
  - Einstein: spezielle Relativitätstheorie
    - Über große Distanzen betrachtet sind Raum und Zeit nicht unabhängig
      - Zeit ist relativ
    - Der "Beobachter" eines zeitlichen Ereignisses befindet sich in einem Bezugsrahmen (*frame of reference*)
    - Beobachter in unterschiedlichen Bezugsrahmen können Ereignisse unterschiedlich geordnet wahrnehmen
      - Beobachter 1: A kommt vor B
      - Beobachter 2: B kommt vor A
    - Ursächliche Ordnung (*causal ordering*) von Ereignissen
      - A ist ursächlich für B, wenn alle möglichen Beobachter wahrnehmen, dass A vor B kommt
- Gleichzeitigkeit: Zwei Ereignisse finden gleichzeitig statt,...
  - Platonisten: ... wenn sie zur selben Zeit passieren
  - Reduktionisten: ... wenn sie "zusammen" passieren
  - Einstein: ... wenn es keine ursächliche Beziehung zwischen beiden gibt

- Mathematisches Modell der Zeit
  - Zeit entspricht den reellen Zahlen
  - Jeder Zeitpunkt entspricht einem Punkt auf dem reellen Zahlenstrahl
  - Eigenschaften der Zeit:
    - **Transitivität:**  $\forall x, y, z : (x < y \wedge y < z) \Rightarrow x < z$
    - **Linearität:**  $\forall x, y : x < y \vee y < x \vee x = y$
    - **Irreflexivität:**  $\forall x : \text{not}(x < x)$
    - **Dichte:**  $\forall x, y : x < y \Rightarrow \exists z : (x < z < y)$
- **Echtzeit**
  - In Echtzeitsystemen muss die Ausführung koordiniert werden mit der "Zeit" in der Umgebung (= Echtzeit, *real time*)
  - Die philosophische Deutung der Umgebungs-Zeit ist unerheblich
  - Oft reicht für Echtzeitsysteme ein **diskretes** Zeitmodell
    - Die Eigenschaft "Dichte" entfällt

- Zeitmessung
  - in der Regel werden Zeitmaße definiert mit Hilfe regelmäßig wiederkehrender Ereignisse
  - Zeitstandards (Beispiele):

Standard	Beschreibung
Sonnentag	Zeit zwischen zwei aufeinander folgenden Kulminationspunkten (höchster Stand) der Sonne
UT0	<i>Universal Time</i> : Mittlere Sonnenzeit am 0-Meridian (Greenwich)
UT1	Korrektur von UT0 aufgrund der Bewegung der Erdachse
UT2	Korrektur von UT1 aufgrund von Variationen in der Geschwindigkeit der Erdrotation
ITA	<i>International Atomic Time</i> : Basiert auf der Periodendauer des Übergangs zwischen zwei Energieniveaus des Caesium-133-Atoms. - <b>1 Sekunde</b> ist das 9 192 631 779-fache davon.
UTC	<i>Universal Coordinated Time</i> : Synchronisation der ITA-Zeit durch gelegentliche Ergänzung von Schaltsekunden

- Anforderung
  - Programme, die Echtzeitsysteme steuern müssen in irgendeiner Weise auf Zeit zugreifen können:
    - feststellen "wie spät es ist"
    - messen, wie viel Zeit seit einem Ereignis vergangen ist
- Mögliche Arten des Zeit-Zugriffs
  - Direkter Zugriff auf die Umgebungszeit
    - Empfang von Zeitsignalen über Radio-Wellen
      - Genauigkeit: 0.1 -10 ms
    - GPS stellt einen UTC-Dienst zur Verfügung
      - Genauigkeit: 1  $\mu$ s
    - Internetdienst
  - Nutzung einer internen Hardware-Uhr
    - ermöglicht eine approximative Messung der in der Umgebung verstrichenen (Echt-) Zeit

- Probleme mit der internen Hardware-Uhr
  - Generierung:
    - Zählen von Schwingungen in einem Quarz-Kristall und teilen durch eine feste Zahl
    - Speichern des Werts in einem Register (zugänglich für Programme)
  - **Drift:** Unterschied zwischen externer und interner Zeit
    - Grund: Die Schwingungen im Quarz-Kristall sind nicht immer gleich lang (z.B. aufgrund von Temperatureinflüssen)
      - Standard Quarz-Kristall:  $10^{-6}$  sec/ 1 sec  $\rightarrow$  1 sec / 11.6 Tage
      - Präzisions-Uhr:  $10^{-7}$  ...  $10^{-8}$
  - **Versatz** (*skew*): Unterschiede zwischen mehreren internen Uhren
    - Bei verteilten Systemen mit mehreren Prozessoren mit eigenen internen Uhren
    - Synchronisation über einen globalen Zeitdienst erforderlich
  - **Alterung:** Wenn ein aktueller Zeitwert der internen Hardware-Uhr gelesen und z.B. in einer Variablen gespeichert wird, ist er bereits veraltet!



- Hilfsmittel für Programmierer
  - Gerätetreiber
    - zum Lesen des internen Hardware-Uhr
    - zum Empfangen von Radio- oder GPS-Signalen
  - Zeit-Abstraktionen in der Programmiersprache
    - Ada: Bibliotheks-Pakete
      - **Calendar**-Package (obligatorisch)
      - **Real\_Time**-Pacckege (optional)
    - C/Real-Time POSIX: Zeitbezogene API (Datentypen und Funktionen)

- Das **calendar**-Package (vgl. Beispiel 5-1)
  - Elementare Datentypen
    - Zum Zählen von Jahren, Tagen und Monaten (ganzzahlig)
      - `Year_Number`, `Month_Number`, `Day_Number`
    - Zeiträume in Sekunden (`Real`-Festkommazahlen)
      - `Duration`, Wertebereich mindestens -86.400,0 ... 86.400,0
      - speziell: `Day_Duration`, Wertebereich 0 ... 86.400,0
  - Abstrakter Datentyp `Time`
  - Lesen der aktuellen Zeit
    - `function Clock return Time;`
  - Umwandeln von `Time`-Werten in lesbare Form (und umgekehrt)
    - `function Year(Date:Time) return Year_Number;`
    - `function Month(Date:Time) return Month_Nuber;`
    - `function Day(Date:Time) return Day_Nuber;`
    - `function Seconds(Date:Time) return Day_Duration;`
  - Rechnen mit Zeiten: Operatoren `+`, `-`
  - Vergleichen von Zeiten: Operatoren `<`, `<=`, `>`, `>=`

- Der Datentyp **Time**
  - Ein Wert des Typs **Time** ist eine Kombination des Datums und der Uhrzeit eines Tages
  - Die Uhrzeit des Tages ist die Anzahl der Sekunden seit Mitternacht
    - Sekunden werden beschrieben als Wert des Typs **Day\_Duration** (s.o.)
    - `Day_Duration` ist Subtyp von `Duration`
- Der Datentyp **Duration**
  - vordefinierter, elementarer Datentyp
  - Wertebereich implementierungsabhängig (mindestens die Sekunden eines Tages, s.o.)
  - Genauigkeit:
    - Der kleinste Wert (`Duration'Small`) darf nicht größer als 20 ms sein
    - Empfehlung Ada Reference Manual: sollte nicht größer als 100 µs sein

- Rechenzeit messen

**declare**

Old\_Time, New\_Time : Time;

Interval : Duration;

**begin**

Old\_Time := Clock;

*-- other computations*

New\_Time := Clock;

Interval := New\_Time - Old\_Time;

**end;**

- Das **Real\_Time**- Package (vgl. Beispiel 5-2)
  - Ähnlich wie `Calendar`, aber feinere Granularität und monoton
  - **Monotone Zeit**: Zeit wird gezählt in Sekunden, beginnend bei einer von der Sprache nicht festgelegten "**Epoche**" (**epoch**) `E`
    - z.B. Zeitpunkt des Systemstarts
    - z.B. vorgegeben durch einen Zeit-Standard
    - Schaltjahre- oder Sekunden werden nicht berücksichtigt
  - **Time**: Datentyp für Zeiten
    - **Time\_Unit**: `constant Time`
      - Die kleinste durch `Time` darstellbare Zeiteinheit (in Sekunden)
      - Dadurch ist der Zeitstrahl diskretisiert: Jedes `t:Time` entspricht einer Ganzzahl  $I_t$  und repräsentiert das Zeitintervall
$$[ E + I_t * \text{Time\_Unit} , E + I_t * (\text{Time\_Unit} + 1) )$$
  - **Time\_Span**: Datentyp für Zeiträume
    - **Time\_Span\_Unit**: `constant Time_Span`
      - Konstante für den kleinsten durch `Time_Span` darstellbaren Zeitraum (in Sek.)
      - Jedes `d:Time_Span` entspricht einer Ganzzahl  $I_d$  und repräsentiert den Zeitraum  $I_d * \text{Time\_Span\_Unit}$

- Das **Real\_Time**- Package (Forts.)
  - **function Clock return Time;**
    - Liefert die aktuelle Zeit
  - **Tick: constant Time\_Span;**
    - Ein *Clock Tick* ist das Echtzeit-Intervall, in dem ein Aufruf der Funktion `Clock` einen unveränderten Wert ergibt
    - Die Konstante `Tick` ist die Durchschnittliche Länge dieses Intervalls
  - **Seconds\_Count**
    - Datentyp zum zählen von Sekunden
    - Verwendet in Prozedur `Split` und Funktion `Time_of` (siehe Beispiel 5-2)
  - Minimalanforderungen an die Implementierung
    - `Tick`  $\leq$  1 ms
    - Wertebereich von `Time`: Mindestens **E** + 50 Jahre

- ANSI C Schnittstelle für kalendarische Zeit (s. Beispiel 5-3)
  - Datentyp `time_t`: repräsentiert Zeiten als elementarer Wert
    - Aktuelle Zeit = Anzahl Sekunden seit 1.1.1970, 00:00 Uhr GMT
  - `struct tm`: Repräsentiert Zeiten in Sekunden, Minuten,..., Tage
  - Funktionen zum Manipulieren von `time_t`-Werten
    - `time_t time(time_t *timer)`
      - Die aktuelle Zeit wird geliefert und, falls `timer != 0` dort abgelegt
    - `struct tm * local_time(time_t * timer)`
      - Die Zeit `*timer` wird in einer `tm`-Struktur abgelegt und ein Pointer darauf zurückgegeben
    - `time_t mktime(struct tm *timeptr)`
      - Die in der Struktur `*timeptr` repräsentierte Zeit wird als `time_t`-Wert zurückgegeben
    - `char * asctime ( const struct tm * timeptr )`
      - Die Zeit in der Struktur `*timeptr` wird in lesbarer Form als Zeichenkette abgelegt und ein Pointer darauf zurückgegeben.
        - » z.B. Sat May 20 15:21:51 2000

- C/Real-Time POSIX Schnittstelle für Uhren (*Clocks*)  
(s. Beispiel 5-4)
  - Eine Implementierung kann mehrere Uhren unterstützen
  - Jede Uhr hat einen eigenen Identifier des Typs `clockid_t`
  - Mindestens eine Uhr muss unterstützt werden:
    - Identifiziert durch die Konstante `CLOCK_REALTIME`
    - Kalendarische Zeit seit 1.1.1970
  - Minimale Auflösung: nicht mehr als 20 ms
  - `struct time_spec`
    - Die Zeit in Sekunden
    - Die zusätzlichen Nanosekunden
  - Funktionen haben die ID der gewünschten Uhr als Parameter
  - Monotone Zeit kann ebenfalls unterstützt werden
    - z.B. als Uhr mit ID `CLOCK_MONOTONIC`



- **Verzögerung (Delay):**
  - eine gewisse (definierte) Zeit die Ausführung einer Task unterbrechen, möglichst ohne den Prozessor weiter zu blockieren
  - **Relative Verzögerung:** für einen gewissen **Zeitraum**
  - **Absolute Verzögerung:** bis zu einem gewissen **Zeitpunkt**
- Relative Verzögerung mit "Busy waiting" in Ada

```
Start := Clock; -- from calendar
loop
    exit when (Clock - Start) > 10.0
end loop;
```
- Relative Verzögerung ohne Busy-Waitung in Ada: **delay**

```
delay 10.0; -- Task wird 10 Sekunden blockiert
```
- Verzögern in C/Real-Time POSIX
  - `sleep(unsigned seconds);`
  - Mit `CLOCK_REALTIME`: `nanosleep()` (vgl. Beispiel 5-4)

- Absolute Verzögerung

- Durch Berechnen der erforderlichen relativen Verzögerung
- Beispiel (Ada): Eine Zweite Aktion soll genau 10s nach einer ersten Aktion beginnen

```
START := Clock;  
FIRST_ACTION;  
delay 10.0 - (Clock - START);  
SECOND_ACTION;
```

- *Problem*: die Task könnte während und nach der Verzögerungsberechnung verdrängt werden
- Besser:

```
START := Clock;  
FIRST_ACTION;  
delay until START + 10.0;  
SECOND_ACTION;
```

- **Drift**

- Verzögerungen (*delay*, *delay until*) definieren nur eine untere Schranke, wann die Task frühestens wieder ausführbereit ist
- Der Zeitpunkt, wann sie nach einer Verzögerung wieder rechnet, kann später sein
- Die Differenz zwischen gewünschter und tatsächlicher Aktivierungszeit nach einer Verzögerung heißt **lokaler Drift** (*local drift*)
- Der lokale Drift kann nicht vermieden werden
- Es ist möglich, das Kumulieren von lokalen Drifts (*cumulative drift*) zu verhindern
- Beispiel mit kumulierenden Drifts:

Verzögerung mindestens  
7 Sekunden → lokaler und  
kumulierender Drift

```
task T;  
  
task body T is  
begin  
  loop  
    Action;  
    delay 7.0;  
  end loop;  
end T;
```

- Drift
  - Verhindern von kumulierendem Drift

```
task body T is  
    Interval : constant Duration := 7.0;  
    Next_Time : Time;  
begin  
    Next_Time := Clock + Interval;  
    loop  
        Action;  
        delay until Next_Time;  
        Next_Time := Next_Time + Interval;  
    end loop;  
end T;
```

## Anmerkungen

- Die Schleife wiederholt sich durchschnittlich alle 7 Sekunden
- Nur lokaler Drift
- Wenn die Action länger als 7 Sekunden braucht, hat delay keinen Effekt

- Eine Task muss innerhalb einer definierten Zeit erkennen, dass ein erwartetes Ereignis nicht eingetreten ist, und darauf reagieren
- Konkret
  - Timeout als einschränkende Bedingung (constraint) über der Zeit, die eine Task auf eine Kommunikation wartet
    - z.B. bis ein Semaphore freigegeben ist
    - z.B. bis ein Mutex oder eine Bedingungsvariable freigegeben ist
    - z.B. bis ein Eintrittspunkt eines geschützten Objekts freigegeben ist
    - z.B. bis ein gewünschtes Rendezvous stattfindet
  - Timeout als einschränkende Bedingung (constraint) über die Zeit, in der ein Code-Fragment ausgeführt sein muss

- Warten auf Semaphore mit Timeout (C/RealTime POSIX)  
(vgl. Beispiel 3-1)

- `int sem_timedwait(sem_t *sem,  
                                struct timespec *abstime);`

- Wenn der Semaphor `sem` innerhalb `abstime` gesetzt werden kann ist das Resultat `0`
    - Ansonsten ist das Resultat `-1` und die globale Konstante `errno` hat den Wert `ETIMEDOUT`

- Anwendung:


```
if(sem_timedwait(&sem, &timeout) < 0) {  
    if (errno == ETIMEDOUT) {  
        /* Fehlerbehandlung für den Timeout-Fall */  
    }  
    else {  
        /* Fehlerbehandlung für andere Fehler */  
    }  
    else { /* Semphor konnte gesetzt werden */ ... };
```

- Analog für Mutexe und Bedingungsvariablen (vgl. Beispiel 3-3)

- Nachrichtenaustausch ohne Timeout
  - Beispiel (Ada): Eine Controller-Task bekommt regelmäßig Nachrichten mit Temperaturwerten, die verarbeitet werden müssen

```
task Controller is
  entry Call(T : Temperature);
end Controller;

task body Controller is
  -- declarations, including
  New_Temp : Temperature;
begin
  loop
    accept Call(T : Temperature) do
      New_Temp := T;
    end Call;
    -- other actions
  end loop;
end Controller;
```



Hier **muss** ein  
Rendezvous stattfinden!

- Empfangen von Nachrichten mit Timeout (Ada)
  - **delay-Alternative** in der **select**-Anweisung

```
task Controller is
  entry Call(T : Temperature);
end Controller;

task body Controller is
  -- declarations ...
begin
  loop
    select
      accept Call(T : Temperature) do
        New_Temp := T;
      end Call;
    or
      delay 10.0;
      -- action for timeout
    end select;
    -- other actions
  end loop;
end Controller;
```

Wenn nach 10 Sekunden  
kein Rendezvous stattfand,  
geht's weiter.



- Senden von Nachrichten mit Timeout (Ada)
  - **delay-Alternative** in einem "Bedingten Aufruf eines Eintrittspunkts" (*conditional entry call*, vgl. [4-24](#))
  - Beispiel: Struktur einer "Treiber-Task", die die Controller-Task nutzt, ohne Timeout

```
loop
  -- get new temperature T
  Controller.Call(T);
end loop;
```

Wenn Controller zu spät rendezvous-bereit ist, und schon neue Temp-Werte vorliegen, ist es nicht sinnvoll, den alten noch zu senden.

- Beispiel: Treiber-Task mit Timeout beim Senden

```
loop
  -- get new temperature T
  select
    Controller.Call(T);
  or
    delay 0.5;
  null;
end select;
end loop;
```

Der Treiber wartet höchstens  $\frac{1}{2}$  Sekunde auf das Rendezvous, dann wird ein neuer Temperaturwert geschickt.

Die null-Anweisung ist eigentlich überflüssig. – Sie soll hier deutlich machen, dass in der delay-Alternative auch noch andere Anweisungen stehen können

- Aufruf von Eintrittspunkten von geschützten Objekten mit Timeout (Ada, vgl. 3-39ff)
  - Analog zum Senden von Nachrichten (Rendezvous)
  - Beispiele: Sei **E** ein Eintrittspunkt eines geschützten Objekts **P**

- Normaler Aufruf

**P.E** ←

Wenn der Eintrittspunkt "geschlossen" ist, wird die Aufrufende Task blockiert, bis der Eintrittspunkt für sie frei ist.

- Mit else-Alternative

**select**

**P.E** ; -- *E is an entry in a Protected Object P*

**else** ←

-- *Statements*

**end select**;

Wenn der Eintrittspunkt "geschlossen" ist, werden sofort die Anweisungen des `else`-Teils ausgeführt

- Mit Timeout

**select**

**P.E** ; -- *E is an entry in a Protected Object P*

**or** ←

**delay 0.5**;

-- *Statements*

**end select**;

Wenn der Eintrittspunkt "geschlossen" ist, werden nach ½ Sekunde die Anweisungen nach `delay` ausgeführt

- Anweisungen mit Timeout (Ada)
  - Eine Menge von Anweisungen soll innerhalb einer definierten Zeitspanne ausgeführt sein
  - Beispiel:

```
select  
  delay 0.1;  
  -- emergency action  
then abort  
  -- action  
end select;
```

Wenn `action` nicht nach spätestens 100 ms beendet ist, wird `action` abgebrochen und die `emergency action` ausgeführt.

- Anweisungen mit Timeout (Ada)
  - Beispiel: Ein Algorithmus mit "Pflichtteil" und optionalem Teil

**declare**

Precise\_Result : Boolean;

**begin**

Completion\_Time := ...

-- compute Result

Results.Write(...); -- call to procedure in  
-- external protected object

Pflicht

**select**

**delay until** Completion\_Time;

Precise\_Result := False;

**then abort**

**while** Can\_Be\_Improved **loop**

-- improve result

Results.Write(...);

**end loop;**

Precise\_Result := True;

**end select;**

**end;**

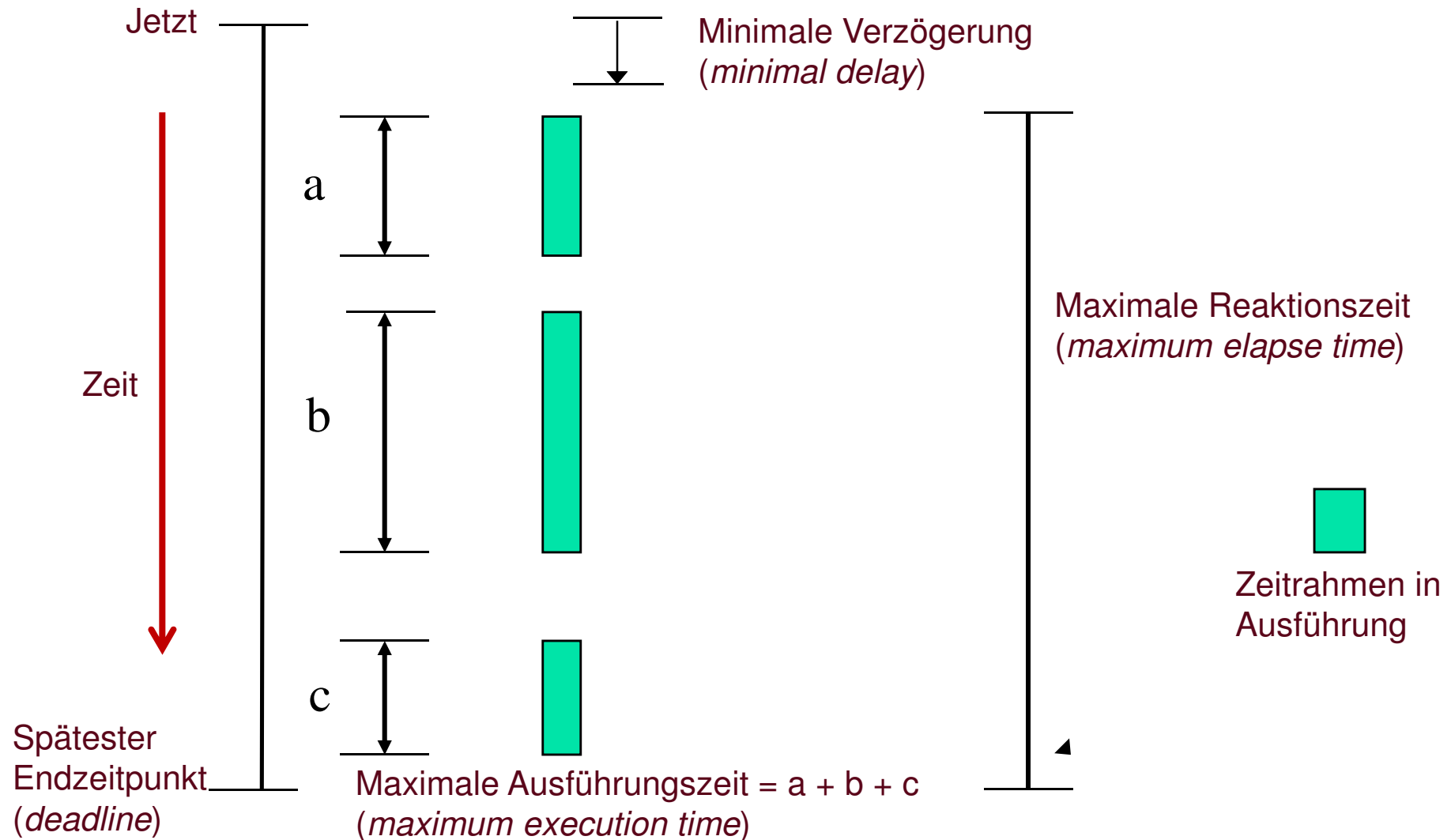
Optional

Es wird solange versucht, das Ergebnis zu verbessern, bis es nichts mehr zu verbessern gibt oder die Completion\_Time erreicht ist.

- Ein **Zeitraahmen** (*temporal scope*) ist eine Menge von Anweisungen, die mit einer oder mehreren Zeitbedingungen verknüpft ist
- Zeitbedingungen
  - **Spätester Endzeitpunkt** (*deadline*)
    - Der Zeitpunkt, an dem die Ausführung des Zeitrahmens spätestens beendet sein muss
  - **Minimale Wartezeit** (*minimum delay*)
    - Der Zeitraum, nach dem die Ausführung des Zeitrahmens frühestens beginnen darf
  - **Maximale Ausführungszeit** (*maximum execution time*)
    - Summe der Zeiträume, die der Zeitrahmen höchstens in Ausführung (auf dem Prozessor) sein darf
  - **Maximale Reaktionszeit** (*maximum elapsed time*)
    - Zeitraum nach Start des Zeitrahmens, in dem die Ausführung des Zeitrahmens spätestens beendet sein sollte (inklusive Verzögerungen)

# Zeitraahmen (Temporal Scopes)

- Zeitbedingungen für einen Zeitrahmen



- Kombination von Zeitbedingungen
  - Ein Zeitraahmen kann mit mehreren Zeitbedingungen verknüpft sein
  - Die Zeitbedingungen von mehreren sequentiellen Zeitraahmen können verknüpft sein
  - **Beispiel:** Drei sequentielle Zeitraahmen für einen einfachen Steuerungsprozess (Zeitpunkte  $d1, d2, d3$  sind relativ zum Start des Zeitrahmens)
    1. Messwert lesen  
Deadline =  **$d1$**
    2. Stellwert berechnen  
Minimale Verzögerung =  $d1$   
Deadline =  $d1 + d2$
    3. Stellwert schreiben  
Minimale Verzögerung =  $d1 + d2$   
Deadline =  $d1 + d2 + \mathbf{d3}$

**Anmerkungen:**

- **$d1$**  begrenzt "Eingabeflattern" (***input jitter***)
- **$d3$**  begrenzt "Ausgabeflattern" (***output jitter***)

- **Periodische Zeitrahmen**

- Der Zeitrahmen werden regelmäßig mit festen Zeitbedingungen wiederholt

```
task periodic_T;  
    ...  
begin  
    loop  
        IDLE  
        start of temporal scope  
        ...  
        end of temporal scope  
    end;  
end;
```

- Zeibedingungen:

- Minimale/Maximale Zeiten für IDLE ("Leerlauf")
- Deadlines für den Zeitrahmen
  - absolute Zeit
  - Maximale Ausführungszeit/Reaktionszeit des Zeitrahmens



- Periodische Task mit mehreren Zeitraahmen

```
loop every 100ms
  start of 1st temporal scope
    ... z.B. Messwerte lesen
  end of 1st temporal scope  -- Deadline: 5ms
  start of 2nd temporal scope
    ... z.B. Stellwerte berechnen
  end of 2nd temporal scope  -- Deadline: 70ms
  IDLE 70 ms -- um Output-Jitter zu verhindern
               -- mit Zeitintervall relativ zum loop-Beginn
  start of 3rd temporal scope
    ... z.B. Stellwerte schreiben
  end of 3rd temporal scope  -- Deadline: 80ms
end;
```

- Problem: Scheduling-Verfahren schränken häufig die Struktur von Tasks ein, insbesondere
  - Maximal eine IDLE-Anweisung zu Beginn der Task
  - Maximale eine Deadline

- Nebenläufige periodische Tasks mit je einem Zeitraahmen
  - Deadlines jeweils relativ zum Start des Zeitraahmens

```
taks periodic_PartA;  
  loop every 100ms  
    start of temporal scope  
    Messwerte lesen;  
    Messwerte puffern;  
    end of temporal scope  
    -- deadline 5ms  
  end loop;  
  
task periodic_PartB;  
  loop every 100ms  
    IDLE 5ms;  
    start of temproal scope  
    Messwertpuffer lesen;  
    Stellwert berechnen;  
    Stellwert puffern;  
    end of temporal scope;  
    -- deadline 65ms
```

```
task periodic_PartC;  
  loop every 100ms  
    IDLE 70ms;  
    start of temporal scope  
    Stellwertpuffer lesen;  
    Stellwerte schreiben;  
    end of temporal scope  
    -- deadline 10ms
```

- **Aperiodische Zeitraahmen**

- Sind durch Ereignisse von außen (*interrupts*) gesteuert

```
task aperiodic_T;  
...  
begin  
  loop  
    wait for interrupt;  
    start of temporal scope  
    ...  
    end of temporal scope  
  end loop;  
end;
```

- **Sporadische Zeitraahmen**

- Aperiodische Task, für die ein minimaler Zeitraum zwischen zwei Ereignissen ist definiert ist (entspricht einer minimalen Wiederholrate der Schleife)

- **Ada**

- Keine Unterscheidung zwischen "Tasks" und "Echtzeittasks"
- Deshalb: Verwendung von "niedrigen" Mechanismen wie Zeitgeber und/oder Verzögerungen
- Beispiel: Struktur einer Periodischen Task in Ada

```
task body Periodic_T is
    Next_Release : Time;
    Release_Interval : constant Time_Span := Milliseconds(...);
begin
    -- read clock and calculate the next
    -- release time (Next_Release)
    loop
        -- sample data (for example) or
        -- calculate and send a control signal
        delay until Next_Release;
        Next_Release := Next_Release + Release_Interval;
    end loop;
end Periodic_T;
```

- **C/Realtime POSIX**

- Keine Unterscheidung zwischen "Tasks" und "Echtzeittasks"
- Beispiel:

```
void periodic_thread() /* destined to be the thread */
{
    struct timespec next_release, remaining_time;
    struct timespec thread_period; /* actual period */

    /* read clock and calculate the next
       release time (next_release) */

    while(1) {
        /* sample data (for example) or
           calculate and send a control signal */

        clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME,
                        &next_release, &remaining_time);
        add_timespec(&next_release, &next_release, &thread_period);
    }
}
```

- **C/Realtime POSIX**

- Beispiel (Forts.):

```
// Start periodic thread
int init(){
    pthread_attr_t attributes;      /* thread attributes */
    pthread_t PT;                  /* thread pointer */

    pthread_attr_init(&attributes); /* default attributes */
    pthread_create(&PT, &attributes,
                  (void *)periodic_thread, (void *)0);
}
```

- **Ada:** Behandlung einer Unterbrechung durch ein Geschütztes Objekt (Monitor) mit zwei Operationen:
  - Gemeinsame Boole'sche Variable `Call_Outstanding`
  - **procedure** `Interrupt`: muss aufgerufen werden, falls eine Unterbrechung ausgelöst wurde
    - setzt `Call_Outstanding` auf `True`
  - **entry** `Wait_For_Next_Interrupt`: Aufrufer wird blockiert, bis `Interrupt` aufgerufen wurde (d.h. solange **not** `Call_Outstanding`)
    - setzt nach "Eintritt" `Call_Outstanding` auf `False`

```
protected Aperiodic_Controller is  
  procedure Interrupt; -- mapped onto interrupt  
  entry Wait_For_Next_Interrupt;  
private  
  Call_Outstanding : Boolean := False;  
end Aperiodic_Controller;
```

- Ada: Behandlung von Unterbrechungen durch ein Geschütztes Objekt (Forts.):

```
protected body Aperiodic_Controller is  
  
  procedure Interrupt is  
  begin  
    Call_Outstanding := True;  
  end Interrupt;  
  
  entry Wait_For_Next_Interrupt when Call_Outstanding is  
  begin  
    Call_Outstanding := False;  
  end Wait_For_Next_Interrupt;  
  
end Aperiodic_Controller;
```



- Ada: Aperiodische Task

```
task body Aperiodic_T is
begin
  loop
    Aperiodic_Controller.Wait_For_Next_Interrupt;
    -- action
  end loop;
end Aperiodic_T;
```

- **C/Real-Time POSIX**

- Realisierung aperiodischer Threads analog zu Ada mit Mutexen und Bedingungsvariablen

- Ereignisgesteuerte Programmierung
  - Das Eintreten eines **Ereignisses** (*event*) löst die Ausführung einer **Ereignisbehandlungsroutine** (*event handler*) aus
    - Unterscheide:
      - Welche Task löst ein Ereignis aus?
      - Welche Task behandelt ein Ereignis?
    - Wenn beide Tasks unterschiedlich sein können, ist dies auch eine Form der Task-Kommunikation
  - Gegebenenfalls werden laufende Prozesse blockiert
  - Spezialfälle
    - SW-/HW-Interrupts (Ada: Interrupt-Handling, C/Real-Time POSIX: Signale)
      - auslösende und verarbeitende Task können unterschiedlich sein
    - Ausnahmebehandlung (Exception Handling)
      - auslösende und verarbeitende Task sind in der Regel die selben
    - Zeitereignisse
      - Zeitpunkt ist erreicht
      - Zeitraum ist abgelaufen

- Zeitereignisse in Ada
  - das Package `Ada.Real_Time.Timing_Events`
    - Datentyp für Zeitereignisse `Timing_Event`
    - Datentyp für Behandlungsroutinen: `Timing_Event_Handler`
      - Zeiger auf eine geschützte Prozedur (d.h. die Behandlungsroutine muss in einem geschützten Objekt realisiert werden)
    - Zeitereignisse definieren: `Set_Handler(E, T, H)`
      - `E: in out Timing_Event` (Das zu definierende Ereignis)
      - `T: Time / T:Time_Span` (Der damit verbundene Zeitpunkt bzw. Zeitraum)
      - `H: Timing_Event_Handler` (Zeiger auf die Behandlungsroutine)
        - » Ein Zeitereignis kann mehrfach neu definiert (überschrieben) werden
    - Zeitereignisse deaktivieren
      - `cancel_Handler(E, canceled)`
      - `set_Handler(E, T, null)`
    - Ablauf
      - Nach Eintreten eines definierten Zeitereignisses wird die Behandlungsroutine "so schnell als möglich" ausgeführt
      - Danach ist das Zeitereignis deaktiviert und kann wieder neu definiert werden

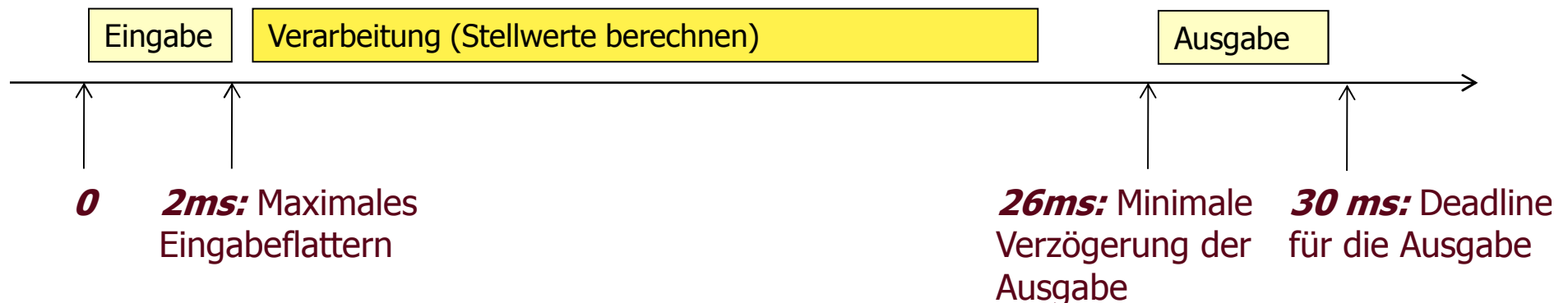
- Das Ada- Package **Ada.Real\_Time.Timing\_Events**

```
package Ada.Real_Time.Timing_Events is
  type Timing_Event is tagged limited private;
  type Timing_Event_Handler
    is access protected procedure (Event : in out
                                   Timing_Event);
  procedure Set_Handler (Event : in out Timing_Event;
                        At_Time : Time; Handler: Timing_Event_Handler);
  procedure Set_Handler (Event : in out Timing_Event;
                        In_Time: Time_Span;
                        Handler: Timing_Event_Handler);
  function Is_Handler_Set (Event : Timing_Event)
    return Boolean;
  function Current_Handler (Event : Timing_Event)
    return Timing_Event_Handler;
  procedure Cancel_Handler (Event : in out Timing_Event;
                           Cancelled : out Boolean);
  function Time_Of_Event (Event : Timing_Event)
    return Time;
private
  ... -- Not specified by the language.
end Ada.Real_Time.Timing_Events;
```

Zeiger auf eine  
Prozedur eines  
geschützten Objekts

Überladene  
Prozedur  
(für Zeitpunkte  
und Zeiträume)

- In **Ada** durch Zeitereignisse
  - **Beispiel:** periodischer Steuerungs-Algorithmus (vgl. S. 31)
    - Sensorwert lesen – Stellwert berechnen – Stellwert schreiben
    - Zeitbedingungen
      - Periode: **40ms**
      - Eingabeflattern: maximal **2 ms**
      - Ausgabe-Deadline: **30ms** nach Periodenbeginn
      - Ausgabeflattern: maximal **4 ms**
      - Ergibt eine minimale Verzögerung der Ausgabe von **26ms**
    - Verarbeitung durch **eine** Task:



- **Ada-Beispiel (Forts.)**
  - Ein geschütztes Objekt für die Eingabe: **Sensor\_Reader**
    - Muss gestartet werden: Prozedur **Start**
      - Ersten Sensorwert lesen, im Objekt Speichern, Zeitereignis definieren: "in 40ms"
    - Behandlungsroutine für das Zeitereignis: Prozedur **Timer**
      - nächsten Sensorwert lesen, im Objekt Speichern
      - neues Zeitereignis (40ms später) definieren
    - Eintrittspunkt **Read**: wird akzeptiert wird, sobald ein neuer Messwert gespeichert ist
  - Ein geschütztes Objekt für die Ausgabe: **Actuator\_Writer**
    - muss 26ms nach der Eingabe gestartet werden (Prozedur **Start**)
    - Prozedur **Write**: Ein Ausgabewert wird im Objekt gespeichert wird
    - Alle 40ms (nach Start) wird der im Objekt gespeicherte Wert ausgegeben
      - Durch ein Zeitereignis mit Behandlungsroutine **Timer**
  - Steuerungstask **Cotrol\_Algorithm**:
    - **SR.Read** – *Stellwertberechnung* – **AW.Write**
      - das "Timing" übernehmen die generierten Zeitereignisse

- **Ada-Beispiel (Forts.)**

```
protected type Sensor_Reader is
  pragma Interrupt_Priority (Interrupt_Priority'Last);
  procedure Start;
  entry Read(Data : out Sensor_Data);
  procedure Timer(Event : in out Timing_Event);
private:
  Reading: Sensor_Data;
  Data_Available: Boolean;
  Next_Time: Time;
end Sensor_Reader;
```

```
Input_Jitter_Control : Timing_Event;
Input_Period : Time_Span := Milliseconds(40);
```

```
protected body Sensor_Reader is
  procedure Start is
  begin
    Reading := Read_Sensor;
    Next_Time := Clock + Input_Period;
    Data_Available := True;
    Set_Handler(Input_Jitter_Control,
                Next_Time, Timer'Access);
  end Start;
```

- **Ada-Beispiel (Forts.)**

```
entry Read(Data : out Sensor_Data) when Data_Available is
  begin
    Data := Reading;
    Data_Available := False;
  end Read;

procedure Timer(Event : in out Timing_Event) is
  begin
    // obtain reading from sensor interface
    Data_Available := True;
    Next_Time := Next_Time + Input_Period;
    Set_Handler(Input_Jitter_Control, Next_Time,
                Timer'Access);
  end Timer;

end Sensor_Reader;
```



- **Ada-Beispiel (Forts.)**

```
protected type Actuator_Writer is
  pragma Interrupt_Priority (Interrupt_Priority'Last);
  procedure Start;
  procedure Write(Data : Actuator_Data);
  procedure Timer(Event : in out Timing_Event);
private
  Next_Time : Time;
  Value : Actuator_Data;
end Actuator_Writer;

protected body Actuator_Writer is ...  -- Übung!

Output_Jitter_Control : Timing_Event;
Output_Period : Time_Span := Milliseconds(40);

SR: SensorReader; AW: Actuator_Writer;

SR.Start;  -- Was ab jetzt mit SR passiert regeln Zeitereignisse!
delay 0.026;  -- 26 ms später ...
AW.Start;  -- Was ab jetzt mit AW passiert regeln Zeitereignisse!
```

- **Ada-Beispiel (Forts.)**

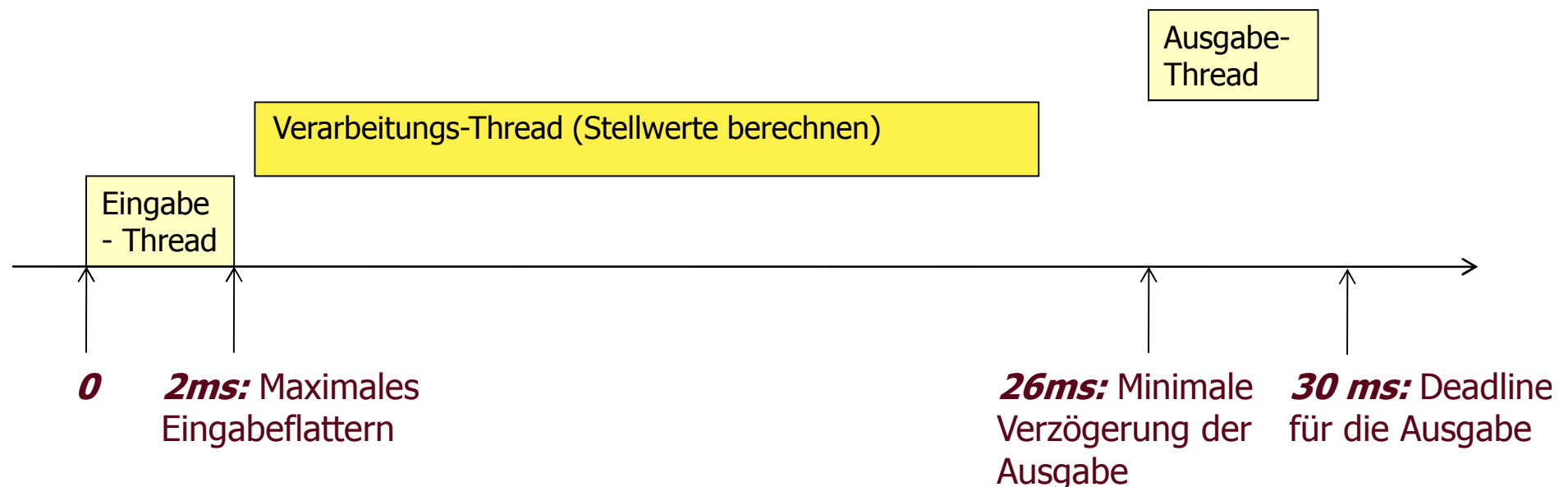
- Steuerungstask Control\_Algorithm
- Muss im selben Block gestartet werden, in dem auch SR und AW deklariert sind
  - Die Task bekommt über Pointer (Access-Parameter) Zugriff auf die geschützten Objekte

```
task type Control_Algorithm
    (Input : access Sensor_Reader;
     Output : access Actuator_Writer);

task body Control_Algorithm is
    Input_Data : Sensor_Data;
    Output_Data : Actuator_Data;
begin
    loop
        Input.Read(Input_Data);
        -- process data;
        Output.Write(Output_Data);
    end loop;
end Control_Algorithm;
```

```
CA: Control_Algorithm(SR'Access, AW'Access);
```

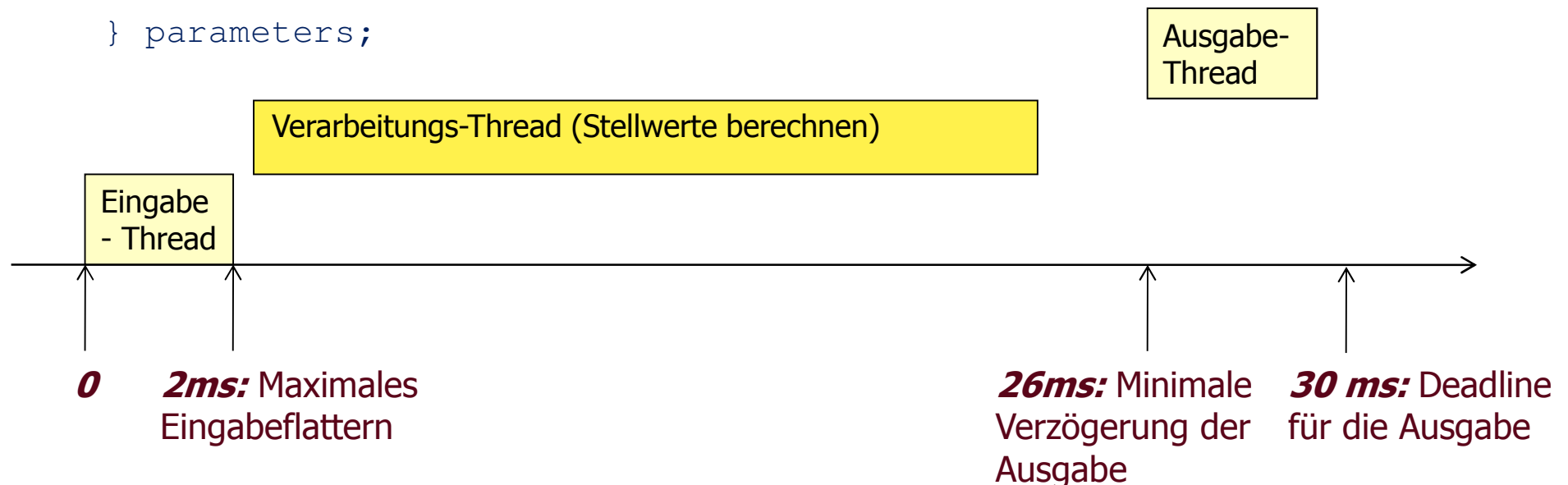
- In **C/Real-Time-POSIX**
  - **Beispiel:** periodischer Steuerungs-Algorithmus (vgl. S. 45ff)
    - Realisiert mit 3 nebenläufigen, periodischen Tasks
    - Durch Zeit-Verzögerungen wird erreicht, dass sie nie gleichzeitig rechnen



- **C/Real-Time-POSIX-Beispiel (Forts.)**

- Struktur für die Zeit-Parameter

```
typedef struct {  
    struct timespec start;           // Startzeit  
    struct timespec period;         // Periode (→ 40ms)  
    struct timespec max_input_jitter; // max. Eingabeflattern (→ 2ms)  
    struct timespec min_latency;    // min. Verzögerung Ausgabe  
                                    // (→ 26ms)  
    struct timespec deadline;       // Ausgabe-Deadline (→ 30ms)  
} parameters;
```



- **C/Real-Time-POSIX-Beispiel (Forts.):** Eingabe-Thread

```
void sensor_thread(parameters *params)
/* destined to be the thread that reads the sensor */
{
    struct timespec next_release, remaining_time;

    /* wait until start time */
    next_release = params->start;
    clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME,
                    &next_release, &remaining_time);

    while(1) {
        /* read sensor data and store in a global data -
           protected by a mutex */
        add_timespec(&next_release, &next_release, &params->period);
        clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME,
                        &next_release, &remaining_time);
    }
}
```

- **C/Real-Time-POSIX-Beispiel (Forts.):** Stellwertberechnung

```
void processing_thread(parameters *params)
/* destined to be the thread that processes the sensor data */
{
    struct timespec next_release, remaining_time;

    /* wait until first release time */
    add_timespec(&next_release, &(params->start),
                &(params->max_input_jitter));
    clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME,
                    &next_release, &remaining_time);

    while(1) {
        /* get data written by input_thread,
           process data and write the value to be written to
           the actuator in global data - protected by a mutex */
        add_timespec(&next_release, &next_release, &(params->period));
        clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME,
                        &next_release, &remaining_time);
    }
}
```

- **C/Real-Time-POSIX-Beispiel (Forts.):** Ausgabe-Thread

```
void actuator_thread(parameters *params)
/* destined to be the thread that write to the actuator */
{
    struct timespec next_release, remaining_time;

    /* wait until first release time */
    add_timespec(&next_release, &params->start,
                &params->min_latency);
    clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME,
                    &next_release, &remaining_time);

    while(1) {
        /* get data written by processing_thread,
           process data and write the value to the actuator */
        add_timespec(&next_release, &next_release, &params->period);
        clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME,
                        &next_release, &remaining_time);
    }
}
```

- **C/Real-Time-POSIX-Beispiel (Forts.)**

- Threads initialisieren und starten

```
void init(){  
  
    struct time_spec start = ... // set start_time  
  
    /* set Timing-Parameters:  
       {start, start+40ms, start+2ms, start+26ms, start+30ms} */  
    parameters P = ... ;  
  
    pthread_attr_t attributes_input;      /* thread attributes */  
    pthread_t PTInput;                    /* thread pointer */  
    pthread_attr_init(&attribute_input); /* default attributes */  
  
    pthread_create(&PTInput, &attribute_input,  
                  (void *) sensor_thread, &P);  
  
    // Similarly for the other threads  
    ...  
}
```



- C/Real-Time-POSIX
  - Außer der Lösung mit den 3 Tasks und geeigneten Delays gibt es auch in C/Real-Time-Posix die Möglichkeit Zeitereignisse zu definieren
    - Erforderliche Schnittstellen:
      - **Signale** (vgl. Beispiel 5-5, 5-6, [Burns & Wellings 2009, Kap. 7.5.1]): zur Ereignisgesteuerten Programmierung
      - **Timer** (vgl. Beispiel 5-7, [Burns & Wellings 2009, Kap. 10.4.2]): zur Definition von Signalen, die durch Zeitereignisse ausgelöst werden
    - Probleme
      - In einem Prozess mit mehreren Threads werden alle Signale an den ganzen Prozess gesendet (also alle Threads)
        - Aktionen der Behandlungsroutine (z.B. "Terminieren") können sich auf alle Threads des Prozesses auswirken
      - Der Zeitpunkt, wann eine Behandlungsroutine ausgeführt wird, ist in C/Real-Time-Posix nicht definiert. (Ada: "so schnell als möglich")
  - Probleme mit der Multitasking-Lösung:
    - Overhead durch Multitasking (→ Prozesswechsel)
    - Die Qualität ist abhängig von der Scheduling Strategie

- **Asynchrone Benachrichtigung**  
(*asynchronous notification*)
  - Motivation: Eine Task soll auf Ereignisse reagieren, die außerhalb des Einflussbereichs der Task liegen
    - z.B. Fehlerzustände in der Hardware
    - z.B. Ausnahmesituationen in einem Überwachten technischen Prozess
  - Erforderlich: Eine Task/Prozess (in dem das Ereignis stattfindet) muss "die Aufmerksamkeit" einer anderen Task/Prozesses erzwingen können (der das Ereignis behandelt)
  - Anwendungen:
    - Fehlerbehandlung
    - Betriebsarten-Wechsel
      - können oft geplant werden
      - werden manchmal durch Ereignisse erzwungen
    - Unterbrechungen durch Benutzer

- **Modelle** für Asynchrone Benachrichtigung
  - **Wiederaufnahme (*resumption*)**
    - Eine Task zeigt an, welche Ereignisse sie behandeln möchte
    - Wenn das Ereignis eintritt, wird die Task unterbrochen und eine Ereignisbehandlungsroutine ausgeführt
    - Anschließend setzt die Task ihre Ausführung fort (→ Wiederaufnahme)
    - Entspricht dem ereignisgesteuerten Programmieren (→ S. 42) bzw. einem "Software-Interrupt"
  - **Terminierung**
    - Eine Task definiert einen Ausführungsbereich (Code-Sequenz), in dem sie bereit ist, eine asynchrone Benachrichtigung zu empfangen
    - Wenn die Benachrichtigung eintrifft, wird dieser Ausführungsbereich beendet (→ Terminierung)
- Asynchrone Benachrichtigung in C/Real-Time-POSIX
  - Mit Hilfe von Signalen
    - nach dem Wiederaufnahme-Modell
    - Sie Beispiel 5-4 und 5-5, [Burns & Wellings 2009, Kap. 7.5.1]

- Asynchrone Benachrichtigung in Ada
  - Asynchrone **Select**-Anweisung

**select**

```
X.entry_call; -- X is a task or protected object
-- optional sequence of statements
-- to be executed after the entry_call has been
-- received (event handler)
```

**then abort**

```
-- abortable sequence of statements
```

**end select;**

- Asynchrone Benachrichtigung nach dem Terminierungsmodell
  - Wenn bei Ausführung der **Select**-Anweisung der Eintrittspunkt nicht gerufen wurde, wird der Teil nach **abort** ausgeführt.
  - Ansonsten wird der Teil nach **select** ausgeführt
  - Falls der Eintrittspunkt während der Ausführung des Teils nach **abort** gerufen wird, wird dieser abgebrochen (beendet) und der optionale Teil ausgeführt
- Spezialfall: Behandlung von Timeout-Fehlern, Siehe [S. 27](#)

- Ablaufbeispiel: Rendezvous sofort

```
task Server is  
    entry ATC_Event;  
end Server;
```

```
task body Server is  
begin  
1  ...  
2  accept ATC_Event do  
5      Seq2;  
    end ATC_Event;  
    ...  
end Server;
```

```
task To_Interrupt;  
task body To_Interrupt is  
begin
```


```
3  ...  
4  select  
    Server.ATC_Event;  
6  Seq3;  
    then abort  
    Seq1;  
    end select  
7  Seq4;  
end To_Interrupt;
```

- Ablaufbeispiel: Kein Rendezvous vor Ende Seq1

```
task Server is  
    entry ATC_Event;  
end Server;  
  
task body Server is  
begin  
    1 ...  
    accept ATC_Event do  
        Seq2;  
    end ATC_Event;  
    ...  
end Server;
```

```
task To_Interrupt;  
task body To_Interrupt is  
begin  
    2 ...  
    3 select  
        Server.ATC_Event;  
        Seq3;  
    then abort  
        4 Seq1;  
    end select  
        6 Seq4;  
    end To_Interrupt;
```

5 Abbruch



- Ablaufbeispiel: Rendezvous fertig vor Ende Seq1


```
task Server is
  entry ATC_Event;
end Server;
```

```
task body Server is
begin
  1 ...
  5 accept ATC_Event do
  6   Seq2;
  end ATC_Event;
  ...
end Server;
```

```
task To_Interrupt;
task body To_Interrupt is
begin
```

```
  2 ...
  3 select
    Server.ATC_Event;
  8 Seq3;
    then abort
  4 Seq1;
    end select
  9 Seq4;
end To_Interrupt;
```

Abbruch 7



- Ablaufbeispiel: Rendezvous fertig nach Ende **Seq1**

```
task Server is
  entry ATC_Event;
end Server;
```

```
task body Server is
begin
  1 ...
  5 accept ATC_Event do
  7   Seq2;
  end ATC_Event;
  ...
end Server;
```

```
task To_Interrupt;
task body To_Interrupt is
begin
```

```
  2 ...
  3 select
    Server.ATC_Event;
  8 Seq3;
    then abort
  4 Seq1;
  6 end select
  9 Seq4;
end To_Interrupt;
```



- [Burns & Wellings 2009] Alan Burns, Andy Wellings: *Real-Time Systems and Programming Languages. Ada, Real-Time Java and C/Real-Time POSIX*. Addison Wesley, 2009.
- [Wörn & Brinkschulte 2005] Heinz Wörn, Uwe Brinkschulte: *Echtzeitsysteme*. Springer, 2005.
- [Zöbel 2008] Dieter Zöbel: *Echtzeitsysteme. Grundlagen der Planung*. Springer, 2008.