Echtzeitsysteme

4. Synchronisation und Kommunikation: Nachrichten (*Messages*)

Prof. Dr. Roland Dietrich

Nachrichtenaustausch

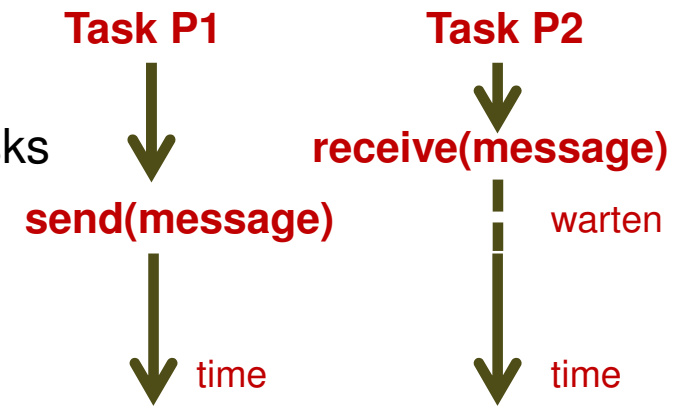
- Synchronisation und Kommunikation über Nachrichten:
  - Eine Sende-Task (***sender***) kann eine Nachricht (***message***) an eine Empfänger-Task (***receiver***) **senden**:

    ```
    send(message)
    ```

  - Eine Empfänger-Task kann eine Nachrichten **empfangen**; falls die Nachricht noch nicht zur Verfügung steht, muss der Empfänger warten, bis die Nachricht eintrifft

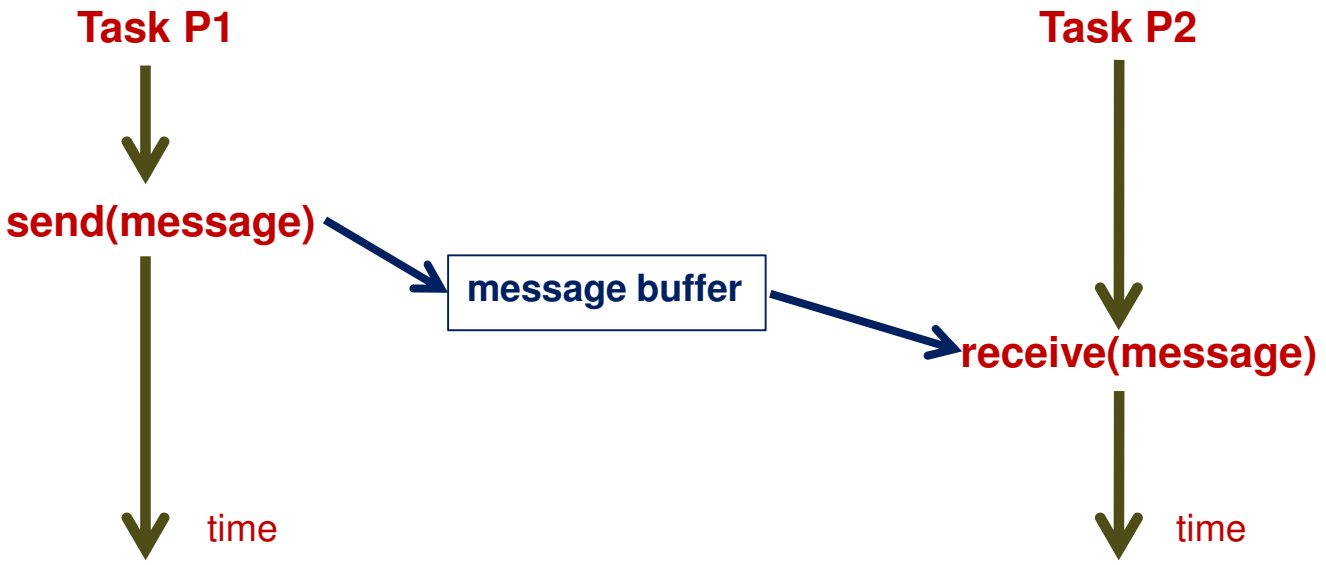
    ```
    receive(message)
    ```

  - Nachrichtenaustausch beinhaltet sowohl Synchronisation als auch Kommunikation (vgl. 3-2)
  - Merkmale von Verfahren zum Nachrichtenaustausch
    - Art und Weise der Synchronisation (**Synchronisationsmodell**)
    - Art und Weise, Sende- und Empfangs-Tasks zu benennen (**Task-Identifikation**)
    - **Struktur** der Nachrichten



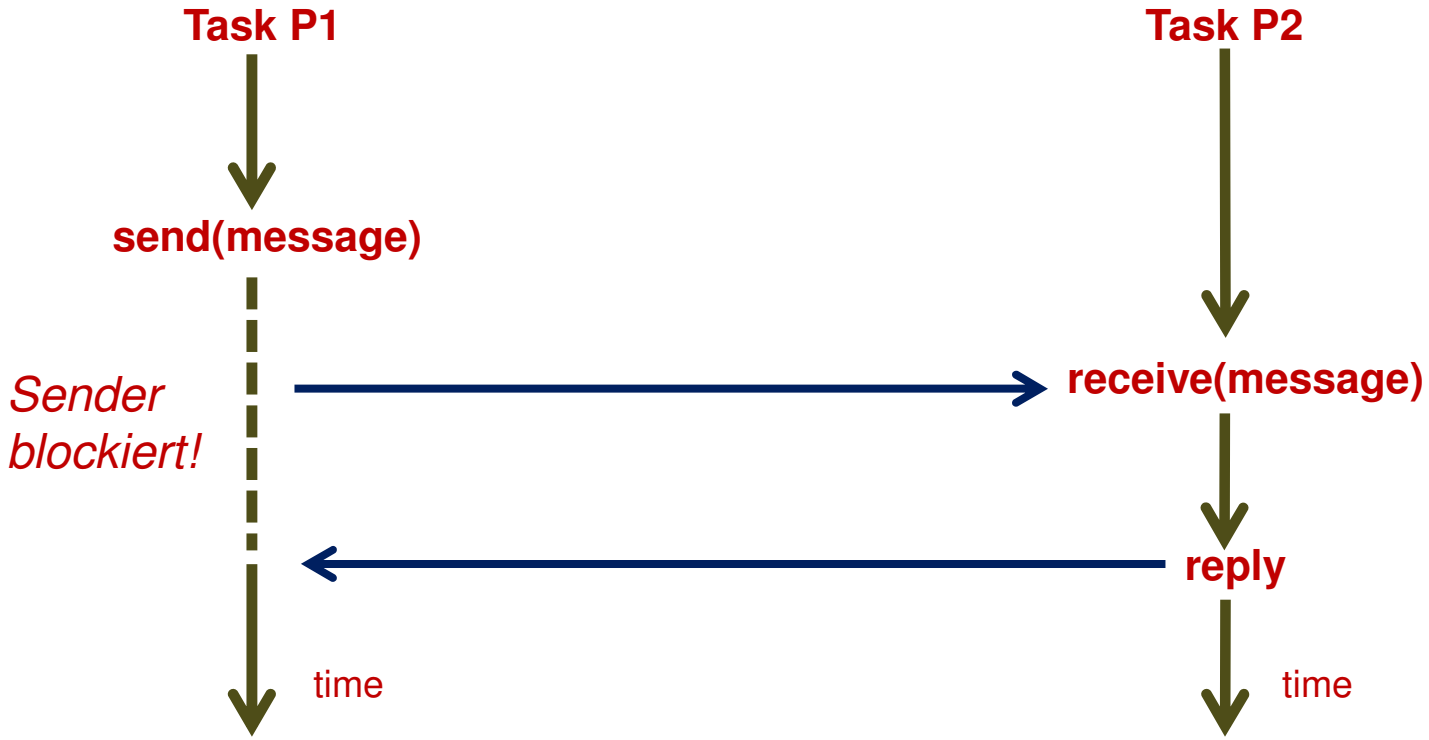
Synchronisationsmodelle

- Unterschiedliche Synchronisationsmodelle ergeben sich aus der Semantik der Send-Operation
- **Asynchrones Senden** (*no-wait*)
  - Nach einer Sende-Operation arbeitet der Sender weiter, unabhängig davon, ob die Nachricht empfangen wurde
  - Notwendig: Puffer-Infrastruktur zur Aufnahme von noch nicht empfangenen Nachrichten (→ was passiert, wenn der Puffer voll ist?)



- **Synchrones Senden**
  - Nach einer Sende-Operation arbeitet der Sender erst weiter, wenn die Nachricht vom Empfänger empfangen wurde.
    - Kein Nachrichten-Puffer erforderlich
    - Bezeichnung: **Rendezvous**

- **Remote-Aufruf** (*Remote invocation*)
  - Nach einer Sende-Operation arbeitet der Sender erst weiter, wenn er eine Antwort vom Empfänger erhalten hat.
    - Der Empfänger erbringt für den Sender einen **Dienst**
    - Bezeichnung: **Erweitertes Rendezvous**



Synchronisationsmodelle

- Realisierung synchroner Kommunikation durch asynchrone *send-/receive-Operationen*:

P1

```
send(message);
receive(acknowledgement);
```

P2

```
receive(message);
send(acknowledgement);
```

- Realisierung von Remote-Aufrufen durch asynchrone *send-/receive-Operationen*

P1

```
send(message);
receive(reply);
```

P2

```
receive(message);
...
compute reply;
...
send(acknowledgement);
```

Task-Identifikation

- Klassifikationskriterien
  - Direkte vs. indirekte Task-Identifikation
  - Symmetrische vs. Asymmetrische Task-Identifikation
- **Direkte** Task-Identifikation
  - Der Sender benennt die Empfänger-Task explizit

```
send <message> to <task-name>;
```

- **Indirekte** Task-Identifikation
  - Der Sender schickt seine Nachrichten über ein Übertragungsmedium (Mailbox, Kanal, Link,…)

```
send <message> to <mailbox>;
```

  - Unterscheide: Wie viele Sender können in eine Mailbox schreiben, wie viele Empfänger können aus einer Mailbox lesen
    - *many-to-one*: Viele Sender, ein Empfänger
    - *many-to-many*: Viele Sender, viele Empfänger
    - *one-to-one*: Ein Sender, ein Empfänger
    - *one-to-many*: Ein Sender, viele Empfänger

Task-Identifikation

- **Symmetrische** Task-Identifikation (direkt oder indirekt)
  - Der Sender identifiziert beim Senden die Empfangs-Task bzw. die Mailbox

```
send <message> to <task-name>;
send <message> to <mailbox>;
```

  - Der Empfänger identifiziert beim Empfangen die Sende-Task bzw. die Mailbox

```
receive <message> from <task-name>;
receive <message> from <mailbox>;
```

- **Asymmetrische** Task-Identifikation
  - Der Empfänger identifiziert beim Empfang einer Nachricht keinen Sender (weder Sende-Task noch Mailbox)

```
receive <message>
```

  - Asymmetrische Task-Identifikation entspricht dem Client-/Server-Kommunikationsmodell:
    - Der Server erbringt einen Dienst, viele Clients können ihn nutzen

Nachrichtenstruktur

- Ideal
  - Beliebige Datenobjekte, die die Programmiersprache ermöglicht sind als Nachrichten erlaubt
    - Standard-Datentypen
    - Programmierer-definierte Datantypen
- Probleme
  - Sender und Empfänger können unterschiedliche Typsysteme haben
  - Sender und Empfänger können unterschiedliche Daten-Repräsentationen haben
  - Besonders problematisch: wenn Datenobjekte Adressen (Pointer) enthalten
    - Diese müssen beim Sender nicht auf dieselben Daten verweisen

Das Ada Rendezvous-Konzept

- Eintrittspunkte in Tasks (***entries***)
  - Damit eine Task eine Nachricht empfangen kann, muss er einen **Eintrittspunkt** (***entry***) definieren
    - Parameter wie Funktionen/Prozeduren
  - *Beispiele:* [Burns & Wellings 2009, Kap. 6.3]

```
task type Screen_Output(Id: Screen_Id) is
  entry Call(Value: Character;
             X_Coordinate, Y_Coordinate: Integer);
end Screen_Output;

Display: Screen_Output(Tty1); -- TTy1 of type Screen_Id

task Time_Server is
  entry Read_Time(Now: out Time);
  entry Set_Time(New_Time: Time);
end Time_Server;
```

Das Ada Rendezvous-Konzept

- Private Eintrittspunkte (***private entries***)
  - können nur aufgerufen werden von Tasks, die lokal sind zum Task-Rumpf, der den privaten Eintrittspunkt definiert
  - *Beispiel:* [Burns & Wellings 2009, Kap. 6.3]

```
task type Telephone_Operator is
   entry Directory_Enquiry(Person: in Name;
                           Addr: in Address;
                           Num: out Number);
   entry Directory_Enquiry(Person: in Name;
                           Zip: in Postal_Code;
                           Num: out Number);
   entry Report_Fault(Num: Number);
private
   entry Allocate_Repair_Worker(Num: out Number);
end Telephone_Operator;
```

Das Ada Rendezvous-Konzept

- Eintrittspunkt-Familien (***entry families***)
  - Felder (Arrays) von Eintrittspunkten
  - *Beispiel:* Ein Multiplexer mit 7 Eingangs-Kanälen [Burns & Wellings 2009, Kap. 6.3]

```
type Channel_Number is new Integer range 1 .. 7;

task Multiplexor is
   entry Channels(Channel_Number)(Data: Input_Data);
end Multiplexor;
```

  - *Anmerkung:* In Ada sind auch Familien von Eintrittspunkten für geschützte Objekte möglich (vgl. Kap. 3)

Das Ada Rendezvous-Konzept

- Nachrichten senden: Aufruf eines Eintrittspunkts
  - Die Empfänger-Task wird direkt benannt
    - Die Task-Identifikation ist direkt
  - Parameter können übergeben werden
  - Resultate können empfangen werden
    - Das Synchronisationsmodell ist Remote-Aufruf (**erweitertes Rendezvous**)
  - *Beispiele:* [Burns & Wellings 2009, Kap. 6.3]

```
Display.Call ('X',10,20);  -- vgl. S. 4-10

Multiplexor.Channels(3)(D); -- vgl. S. 4-12
-- 3 ist der Index der Eintrittspunkt-Familie
-- D ist vom Typ Input_Data

Time_Server.Read_Time(T); -- vgl. S. 4-10
-- T ist vom Typ Time
```

Das Ada Rendezvous-Konzept

- Ausnahme (***Exception***) **`Tasking_Error`**
  - Ada beinhaltet auch Ausnahme-Behandlung (*Exception-Handling*)
  - Falls ein Eintrittspunkt aufgerufen wird, und die gerufene Task nicht mehr aktiv ist, wird die Ausnahme `Tasking_Error` ausgelöst.
  - *Beispiel:*

```
begin
   Display.Call(C,I,J);
exception
   when Tasking_Error ==> Fehlerbehandlung
end;
```

Was ist der Unterschied zu folgendem Programm-Fragment?

```
if Display'Terminated then Fehlerbehandlung
else Display.Call(C,I,J);
```

Das Ada Rendezvous-Konzept

- Nachrichten Empfangen: **`accept`-Anweisung**
  - Der Rumpf (body) einer Task sollte für jeden Eintrittspunkt eine accept-Anweisung enthalten:

```
accept Eintrittspunkt(Parameter) do
   Anweisungen
end Eintrittspunkt;
```

  → *Wenn der Eintrittspunkt von einer anderen Task (Sender) aufgerufen wird **und** der Empfänger die **`accept`**-Anweisung erreicht , werden die Anweisungen ausgeführt.*

  → *Sender und Empfänger haben ein "**Rendezvous**"*

  → *Wer zuerst kommt, muss warten!*

  - Der Empfänger benennt den Sender nicht, der Sender benennt den Empfänger
    → Asymmetrische Task-Identifikation
  - Mehrere Tasks können den selben Eintrittspunkt einer Task aufrufen
    → Die Aufrufe werden in einer Warteschlange verwaltet und nach dem FIFO Prinzip abgearbeitet
    → Der ***Ada Real-Time-Annex*** ermöglicht prioritätsgesteuerte Abarbeitung

Das Ada Rendezvous-Konzept

- Nachrichten Empfangen: **`accept`-Anweisung**
  - Beispiel: (vgl. S. 4-10)

```
task body Screen_Output is
  type XCoord is range 0..799;
  type YCoord is range 0..599;
  type Screen is array(XCoord,YCoord) of Character;
  S: Screen; -- Die Matrix S repräsentiert den Bildschirm
begin
  loop
    accept Call(C: Character; I,J: Integer) do
      S(I,J) := C;
    end Call;
  end loop;
end Screen_Output;
```

Das Ada Rendezvous-Konzept

- **Beispiel:** Zwei Tasks, die 1000 mal Daten tauschen

```
procedure Test is
  Number_Of_Exchanges : constant Integer := 1000;

  task T1 is
    entry Exchange (I : Integer; J : out Integer);
  end T1;

  task T2;

  task body T1 is  -- T1 produziert A, schickt es zu T2
    A,B : Integer; -- bekommt dafür von T1 B und konsumiert es
  begin
    for K in 1 .. Number_Of_Exchanges loop
      -- produce A
      accept Exchange (I : Integer; J : out Integer) do
        J := A;
        B := I;
      end Exchange;
      -- consume B
    end loop;
  end T1;
```

Das Ada Rendezvous-Konzept

- **Beispiel:** Zwei Tasks, die 1000 mal Daten tauschen (Forts.)

```
task body T2 is      -- T2 produziert C, schickt es zu T1
   C,D : Integer;    -- bekommt dafür von T1 D und konsumiert es
 begin
   for K in 1 .. Number_Of_Exchanges loop
     -- produce C
     T1.Exchange(C,D);
     -- consume D
   end loop;
 end T2;

begin
  null; -- Hier werden T1 und T2 gestartet
end Test;
```

  - *Anmerkungen:*
    - T1 und T2 treffen sich ("haben eine Rendezvous"), um Daten auszutauschen
    - Obwohl das Verhalten der beiden Tasks sehr symmetrisch ist, haben sie wegen der asynchronen Task-Identifikation in Ada unterschiedliche Struktur

Das Ada Rendezvous-Konzept

- **Problem**
  - Wenn eine Task mehrere Eintrittspunkte hat, muss jede `accept`-Anweisung warten, bis genau dieser Eintrittspunkt gerufen wird.
  - Beispiel: (Vgl. S. 4-11)

```
task body Telephone_Operator is
...
begin
 loop
  accept Directory_Enquiry(Person : in  Name;
     Addr : in Address; Num : out Number) do
     -- look up telephone number and assign the value to Num
  end Directory_Enquiry;
  accept Directory_Enquiry(Person : in  Name;
     Zip : in Postal_Code; Num : out Number) do
     -- look up telephone number and assign the value to Num
  end Directory_Enquiry;
 ...
 end loop
end Telephone_Operator;
```

Das Ada Rendezvous-Konzept

- Selektives Warten (***selective waiting***): `select`-Anweisung
  - Allgemeine Struktur:

```
select
   alternative1
or
   alternative2
or
...
or
   alternative n
end select;
```

  - Möglichkeiten für die Alternativen:
    - `accept`-Anweisung (*selctive accept*)
    - Bedingte `accept`-Anweisung (*guarded selective accept*)
    - `else`-Alternative
    - Terminierung (`terminate`)
    - Verzögerung (`delay`-Anweisung, siehe später!)

Das Ada Rendezvous-Konzept

- Alternativen in der `Select`-Anweisung
  - selektives `accept`

```
task Server is
  entry S1(...);
  entry S2(...);
end Server;
```

Bei jedem Schleifendurchlauf wir eine der `accept`-Anweisungen ausgeführt, falls eine Task den zugehörigen Eintrittspunkt gerufen hat (d.h. "zum Rendevous bereit" ist)

Danach werden, falls vorhanden, die Anweisungen nach der accept-Anweisung ausgeführt (bis zur nächsten Alternative)

Wenn keine Task zum Rendezvous bereit ist, wartet die Server-Task, bis eine bereit ist.

```
task body Server is
  ...
begin
  loop
    select
      accept S1(...) do
        -- code for this service
      end S1;
      -- eventually more statements
    or
      accept S2(...) do
        -- code for this service
      end S2;
      -- eventually more statements
    end select;
  end loop;
end Server;
```

Das Ada Rendezvous-Konzept

- Alternativen in der `Select`-Anweisung:
  - bedingtes selektives `accept`
    - jede `accept`-Alternative kann mit einem Boole'schen Ausdruck verknüpft sein: einem ***Guard***.
    - Wenn die Select-Anweisung ausgeführt wird, werden zuerst alle Guards ausgewertet (bevor auf Rendezvous-bereite Tasks geprüft wird!)
    - Nur die Alternativen, deren Guard zu `true` evaluiert wird, sind als Alternative auswählbar


```
select
   when Boolean_Expression =>
      accept S1(...) do
         -- code for service
      end S1;
      -- sequence of statements
or
   ...
end select;
```

Das Ada Rendezvous-Konzept

- Alternativen in der `Select`-Anweisung:
  - `terminate`-Anweisung
    - Eine Server-Task muss nur existieren (ausführbereit sein), wenn es Clients gibt, die Dienste nutzen könnten
    - Der Server kennt diese Clients in der Regel nicht
    - Eine `terminate`-Alternative in der `select`-Anweisung erlaubt es einem Server anzuzeigen, dass er bereit ist terminiert zu werden, wenn es keine Clients mehr gibt, die ihn nutzen könnten

```
select
  accept S1(...) do ... end S1;
or
  accept S2(...) do ... end S2;
or
  ...
or
  terminate;
end select;
```

Der Server terminiert, wenn der **Master** des Servers **und** alle seine **abhängigen** Tasks entweder

- terminiert haben oder
- blockiert sind in einer `select`-Anweisung mit `terminate`-Alternative.

Das Ada Rendezvous-Konzept

- Bedingter Aufruf von Eintrittspunkten
  - Ein Sender wird blockiert, wenn der Empfänger nicht Rendezvous-bereit ist
  - Dies kann durch ein "bedingtes" senden verhindert werden
    - "Wenn Empfänger bereit, sende Botschaft, ansonsten fahre fort…"
  - Form: Bedingter Aufruf von Eintrittspunkten (***conditional entry call***)
- Verzögerter Aufruf von Eintrittspunkten
  - Statt einer else-Alternative kann eine maximale Wartezeit angegeben werden (***delay***-Anweisung)
- *Beispiele:* (Vgl. S. 4-21)

```
select
   Server.S1(...);
else
   null;
end select;
```

```
select
   Server.S2(...);
or
   delay 10.0;
   -- maximal 10 Sek. warten
end select;
```

Das Ada Rendezvous-Konzept

- **Beispiel:** (Vgl. S. 4-11)

```
task body Telephone_Operator is
  Workers : constant Integer := 10;
  Failed : Number;
  task type Repair_Worker;
  Work_Force : array (1 .. Workers) of Repair_Worker;
  task body Repair_Worker is ...;
   ...
begin
  loop
    -- prepare to accept next request
    select
      accept Directory_Enquiry(Person : in  Name;
             Addr : in Address; Num : out Number) do
        -- look up number based in adress and assign to Num
      end Directory_Enquiry;
    or
      accept Directory_Enquiry(Person : in  Name;
             Zip : in Postal_Code; Num : out Number) do
        -- look up number based on ZIP and assign to Num
      end Directory_Enquiry;
```

10 lokale Tasks. – Diese dürfen den privaten Eintrittspunkt `Allocate_Repair_Worker` aufrufen.

Das Ada Rendezvous-Konzept

- **Beispiel:** (Forts.)

```
      or
        accept Report_Fault(Num : Number) do
          Failed := Num;
        end Report_Fault;
        -- store failed number in a record of unallocated faults
      or
        when Unallocated_Faults =>
          accept Allocate_Repair_Worker(Num : out Number) do
            Num := ...; -- get next failed number
          end Allocate_Repair_Worker;
          -- update record of failed unallocated numbers
      or
        terminate;
      end select;
      ...
    end loop;
  end Telephone_Operator;
```

Möglichst viele Aktivitäten sollten außerhalb des Rendezvous stattfinden, damit die Client-Tasks möglichst schnell weiter arbeiten können.

Die 10 lokalen `Repair_Worker`-Tasks (Array `Work_Force`) können mit der `Telephone_Operator`-Task über den privaten Eintrittspunkt `Allocate_Repair_Worker` kommunizieren.

Sie erhalten dadurch z.B. eine Fehlerhafte Nummer;

Durch den Guard können Sie das Rendezvous nur beanspruchen, wenn es auch "etwas zu tun gibt".

Nachrichten-Warteschlagen in C/Real-Time POSIX

- Prinzipien
  - Asynchroner Nachrichtenaustausch
  - Indirekte Nachrichtenübertragung über eine Warteschlage
  - Eine Warteschlage kann von vielen Sendern und vielen Empfängern geschrieben/gelesen werden
  - Nachrichten können mit Prioritäten versehen werden
- Nachrichten-Warteschlagen (vgl. [Beispiele zu Kapitel 4])
  - werden durch einen Namen (Zeichenkette) identifiziert
  - besitzen Attribute: maximale Anzahl der Nachrichten, maximale Größe der Nachrichten, aktuelle Anzahl der Nachrichten
  - müssen geöffnet und/oder erzeugt werden: `mq_open()`
  - Können geschlossen bzw. gelöscht werden: `mq_close()` bzw. `mq_unlink()`
  - Beim Öffnen kann man über Parameter Attribute beeinflussen

Nachrichten-Warteschlagen in C/Real-Time POSIX

- Nachrichten Senden: `mq_send()`
  - Nachrichten werden aus einem `char`-Puffer (d.h. als Zeichenkette) mit einer Priorität in die Warteschlage übertragen
  - Wenn die Warteschlange voll ist, wird der aufrufende Prozess blockiert, bis wieder Platz verfügbar ist
    - Ausnahme: die Warteschlagen hat das Flag `O_NONBLOCK` gesetzt; in diesem Fall endet `mq_send` mit Fehler
- Nachrichten Empfangen: `mq_receive()`
  - Die älteste Nachricht mit der höchsten Priorität wird aus der Warteschlage entfernt und in einen `char`-Puffer geschrieben.
  - Wenn die Warteschlagen leer ist, wird der aufrufende Prozess blockiert, bis eine Nachricht in der Schlange eintrifft
    - Ausnahme: analog Nachrichten Senden
- **Beispiel:**
  - Roboterarm-Steuerung, siehe [Beispiele zu Kapitel 4]

Literatur

[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.

[Brinch-Hansen 1973] Per Brinch-Hansen: *Operating Systems Principles.* Englewood Cliffs: Prentice Hall, 1973.

[Hoare 1974] Tony Hoare: *Monitors: An Operating Systems Structuring Concept*. Communications of the ACM, 1978.

[Wörn & Brinkschulte 2005] Heinz Wörn, Uwe Brinkschulte: *Echtzeitsysteme*. Springer, 2005.

[Zöbel 2008] Dieter Zöbel: *Echtzeitsysteme. Grundlagen der Planung*. Springer, 2008.