

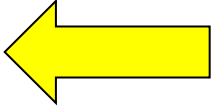
**Folien zur Vorlesung**  
**System – und Echtzeitprogrammierung (BTI4-SY)**  
**Sommersemester 2010**  
**(Teil 1)**

**Prof. Dr. Franz Korf**

[korf@informatik.haw-hamburg.de](mailto:korf@informatik.haw-hamburg.de)

# Kapitel 1 : Einführung

## Gliederung

- Steckbriefe 
- Motivation
- Formalien und Kommentare
- Inhalt der Vorlesung

## „Steckbrief“ von Franz Korf

### Beruflicher Werdegang

Informatik Studium (RWTH Aachen)

Compilerbau, Programmiersprachen, parallele Systeme, SW Entwicklung

Promotion (Universität Oldenburg)

Hardwarebeschreibungssprachen (VHDL, Verilog), Simulationswerkzeuge, formale Verifikation, Controller Synthese

Fujitsu Siemens Computers (Paderborn)

- ASIC Design Prozess, System und RTL Simulation (VHDL, Verilog, C), Synthese, Simulationsumgebungen
- BIOS Entwicklung, Server Management Firmware Entwicklung, eingebettete Systeme, ChipSet- und Rechnerarchitekturen
- OEM / ODM Entwicklung

Hochschule für Angewandte Wissenschaften in Hamburg

- Lehre im Studiengang AI und TI
- Embedded Systems, RTOS, R-ETH, Time-Triggered Systems, FAUST, Server Management
- Informatik in der Schule

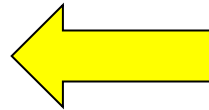
## „Steckbrief“ von Ihnen

- Haben Sie schon eine Vorlesung zu (Real Time) Betriebssystemen gehört?
- Haben Sie Erfahrungen mit parallelen Systemen?
- Haben Sie Erfahrungen mit Kommunikation zwischen Threads, Tasks oder Prozessen?
- Haben Sie Erfahrungen mit Interrupt Handling?
- Was hat Ihnen bisher die meisten Probleme bereitet und was haben Sie daraus gelernt?
- Wer arbeitet neben dem Studium mehr als 10 Stunden pro Woche?
- Wer hat einen Job im Bereich der Informatik?
- Wer denkt über ein Praktikum in der Industrie nach?
- Wer hat eine Aufwandsabschätzung für dieses Semester gemacht?

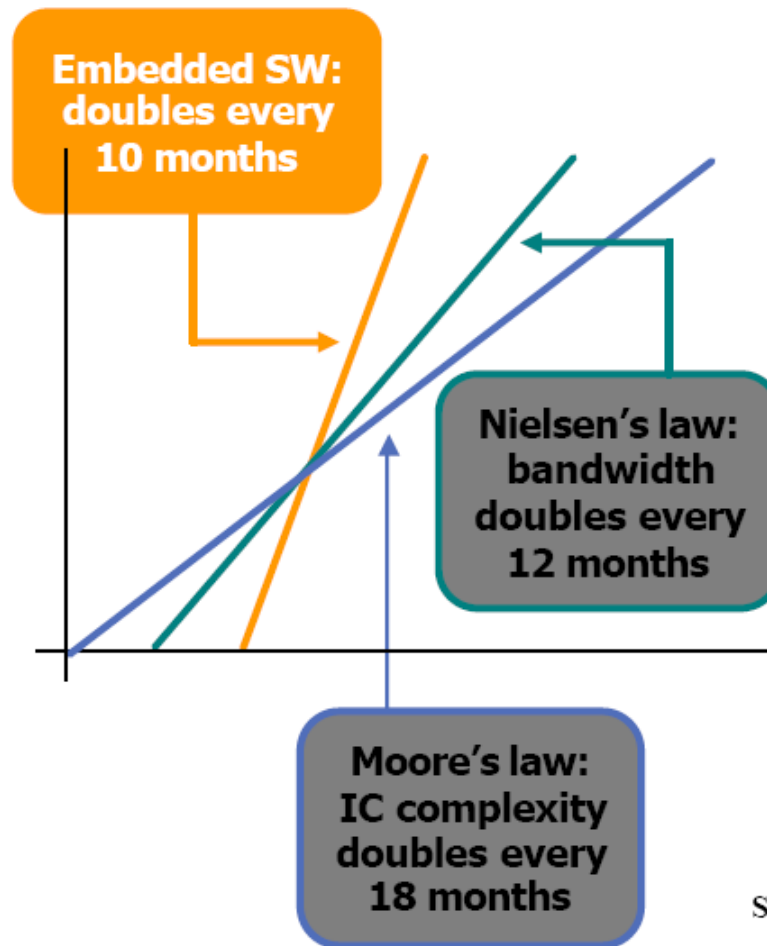
# Kapitel 1 : Einführung

## Gliederung

- Steckbriefe
- Motivation
- Formalien und Kommentare
- Inhalt der Vorlesung



## Motivation



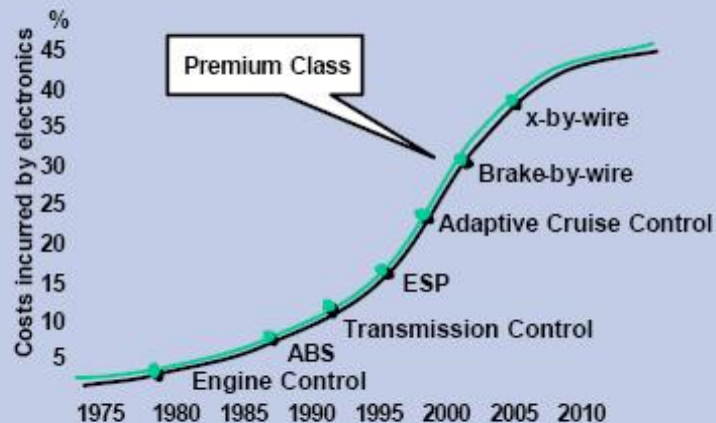
Source: ST Microelectronics



## Embedded SW wächst

### Electronic Content

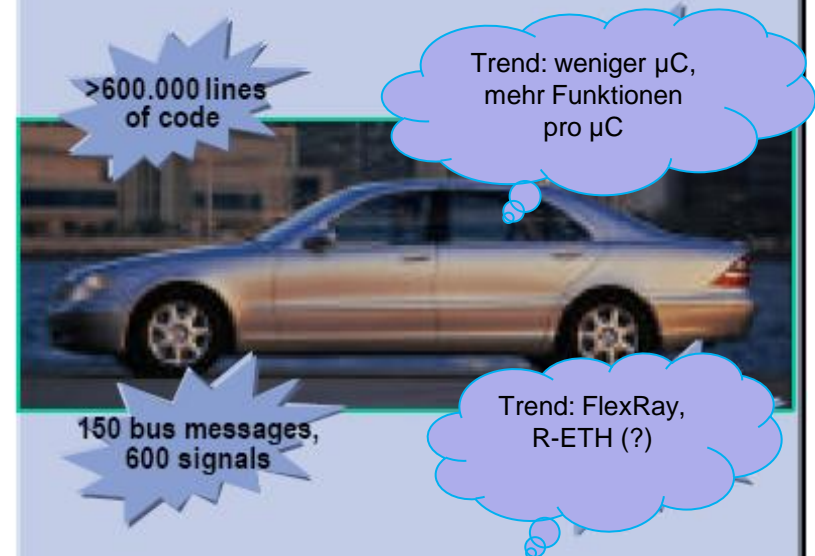
*The continuous growth of vehicle electronics leads to a significant increase in software complexity*



- more than 80% of functions driven by software
- continuously increasing

*Premium Class, 2000*

### State of the Art



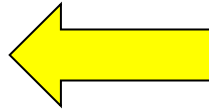
... not even accounting for telematics

*Premium Class, 2000*

# Kapitel 1 : Einführung

## Gliederung

- Steckbriefe
- Motivation
- Formalien und Kommentare
- Inhalt der Vorlesung





## Formalien und Kommentare

### Freie Übungszeiten im TI Labor

- 06.04.2010 bis 09.07.2010  
Mo bis Do 16:00 bis 20:00,  
Fr. 14:00 bis 20:00 (bei planmäßige Veranstaltungen ab 16:00)

### Termine

- Ein Reservetermin

### Sprechstunde

- nach Vereinbarung
- Sprechen Sie mich direkt an (nach der Vorlesung, im Praktikum, ...)

Bitte nutzen Sie die **Mailing Liste** zu dieser Veranstaltung

### Feedback:

- Neue Vorlesung → Feedback ist für mich entscheidend
- Kommentare, Kritik, Verbesserungsvorschläge jeglicher Art sind stets willkommen. Schicken Sie mir eine E-Mail, sprechen Sie mich direkt an, ...
- Zur Halbzeit gibt es eine Feedback Runde in der Vorlesung.

# Struktur des Moduls der Veranstaltung

## Ein Modul

- Software Engineering 2 und Anwendungen

## Vier Veranstaltungen

- Vorlesung: System- & Echtzeitprogrammierung (2 SWS)
- Vorlesung: Software Engineering 2 (2 SWS)
- Vorlesung: Prozesslenkung (2 SWS)
- Praktikum: BTI4-SEP2 (2 SWS)

## Eine mündliche Prüfung

- über alle drei Vorlesungen und das Praktikum

## Kommentar

- Für das Praktikum sind alle drei Vorlesungen entscheidend

## Formalien und Kommentare

### Praktikum

- Anwesenheitspflicht
- Eine große Aufgabe: Steuerung von zwei Transferstrecken zur Sortierung von Bauteilen
- Die Aufgabe enthält Teilaufgaben aus SY, PL und SE
- Es gibt drei Gruppen, jede Gruppe wird von einem anderen Prof. (PRG, FHL, KRF) betreut: Weitere Betreuung: Herrn Lohmann und ein Student
- Prof. hat zwei Funktionen: (a) Auftraggeber und (b) Berater
- Maximal vier Personen arbeiten in einer Gruppe zusammen
- Halten Sie Termine etc. ein.

## Formalien und Kommentare

### Das Praktikum ist bestanden, wenn

- die Anwesenheitspflicht erfüllt wurde &
- geforderte Teilaufgaben pünktlich erfüllt wurden &
- die Gesamtaufgabe erfolgreich bearbeitet und abgenommen wurde

## Formalien und Kommentare

### Unterlagen zur Vorlesung & zum Praktikum

- stehen in Netz bereit  
<http://www.informatik.haw-hamburg.de/korf.html>
- Übungen und Ergänzungen an der Tafel

## Ein Zitat

**Goethe:** „*Denn wir behalten von unseren Studien am Ende nur das, was wir praktisch anwenden.*“

Daraus ergibt sich

- Nehmen Sie an der Vorlesung und am Praktikum aktiv teil.
- Arbeiten Sie die Vorlesung sofort nach.
- Bereiten Sie sich intensiv auf das Praktikum vor.
- Erstellen Sie Ihre persönliche Mitschrift.
- Rechnen / programmieren Sie Beispiele durch.

**Tipp:** Praktikum, Klausur und Vorlesung sind eng miteinander verbunden.

## Literatur & Software

### Literatur:

- C++ Buch Ihrer Wahl, z.B.: **Paul J. Deitel**: C++ How to Program
- **Rob Krten**: Getting Started with QNX Neutrino 2 - A Guide for Realtime Programmers, (2. Auflage), PARSE Software Devices, 2001
- **Andrew S. Tanenbaum**: Modern Operating Systems – 2nd Edition, Prentice Hall 2001, ISBN 0-13-031358-0
- **William Stallings**: Operating Systems – 4th Edition, Prentice Halls
- David R. Butenhof: Programming with POSIX Threads, Addison-Wesley, 1997
- **Giorgio C. Buttazzo**: Hard Real-Time Computing Systems - Predictable Scheduling Algorithms and Applications, Kluwer Academic Publishers
- **Hermann Kopetz**: Real-Time Systems: Design Principles for Distributed Applications, Kluwer Academic Publishers
- **Alan Burns, Andy Wellings**: Real-Time Systems and Programming Processes Languages, Addison-Wesley

## Das Laborsystem



**Entwicklungssystem**

*Ethernet*



*Analog & Digital IO*



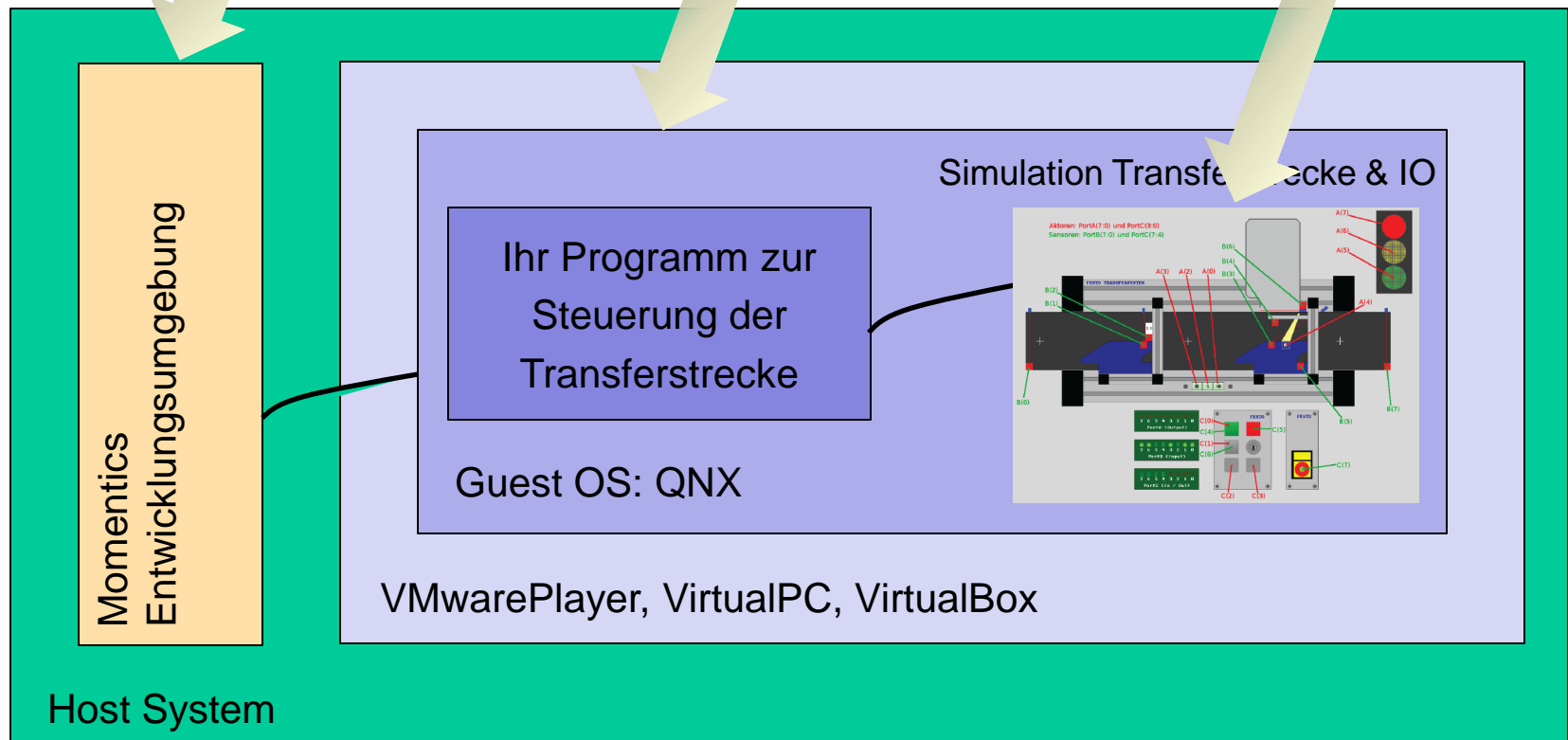
**Zielsystem**

- Entwicklungsumgebung:  
QNX Momentics Eclipse

- x86 Architektur
- OS: QNX Neutrino 2 (kurz QNX)
- MicroKernel Realtime UNIX OS.
- Synchrones Message passing ist die primäre Kommunikationstechnik von QNX.



# Softwareentwicklung@home



## Literatur & Software

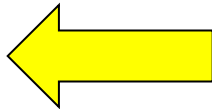
### QNX Software:

- QNX bietet eine kostenfreie Testversion für 1 Jahr (<http://get.qnx.com> oder Herrn Lohmann ansprechen)
- oder die VM
- Simulationsumgebung
- Momentics Entwicklungsumgebung Version

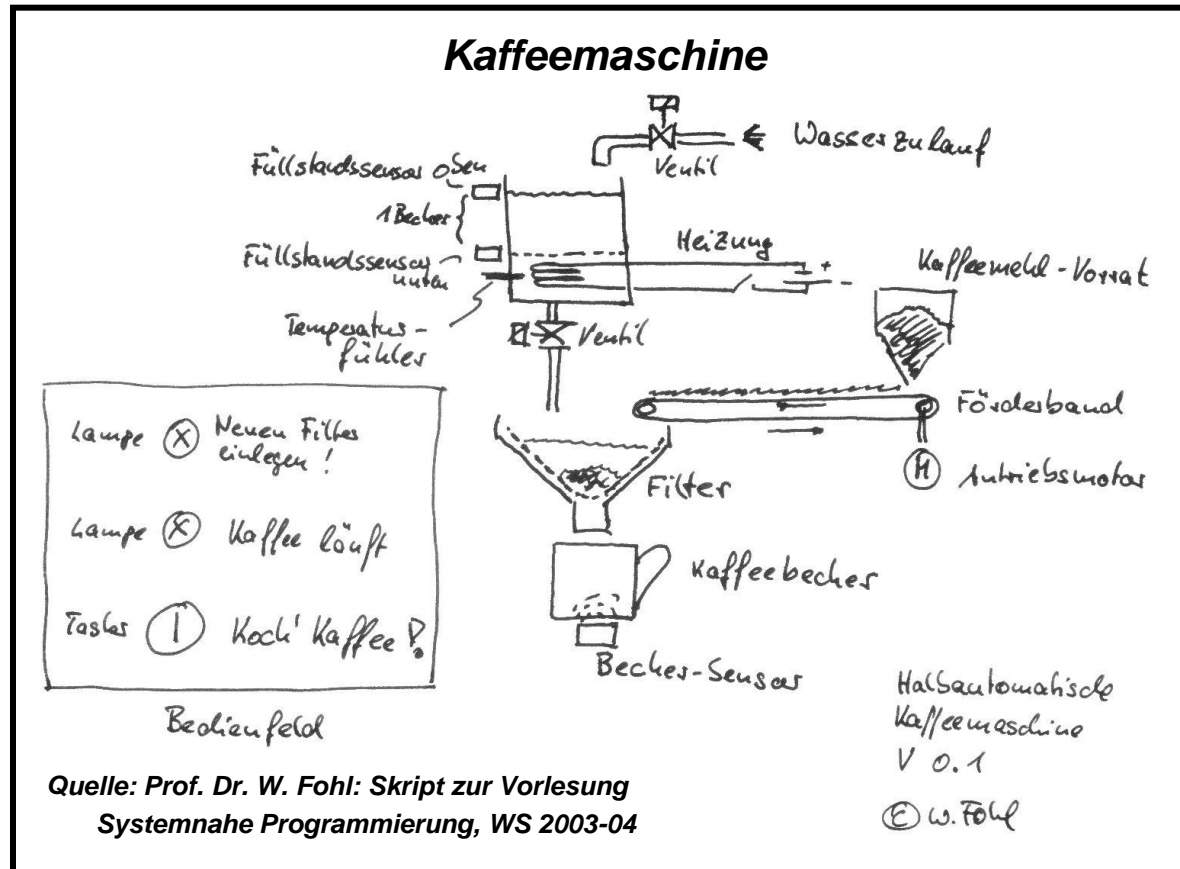
# Kapitel 1 : Einführung

## Gliederung

- Steckbriefe
- Motivation
- Formalien und Kommentare
- Inhalt der Vorlesung



## Inhalt der Vorlesung: Ein Beispiel



## Inhalt der Vorlesung: Ein Beispiel (Fortsetzung)

### Typische Vorgänge, die in der Kaffeemaschine stattfinden:

- Temperaturregelung
- Steuerung des Wasserzuflusses
- Dosierung des Kaffeemehls (Förderband, Kaffeemehlsilo)
- Zustandsanzeige (Steuerung der Lampen)
- Steuerung des Bedienfeldes (Taster „Koch´Kaffee“)
- Überwachung (Becher vorhanden, Überhitzung Heizung, ...)
- ...

Diese Vorgänge  
laufen  
**gleichzeitig**  
**(nebenläufig)** ab.

## Inhalt der Vorlesung: Ein Beispiel (Fortsetzung)

### Anforderungen an ein Computersystem zur Steuerung der Kaffeemaschine:

- Schnell genug auf externe Ereignisse reagieren  
-> Realtime Anforderungen / Echtzeit
- Mehrere Aufgaben nebenläufig (parallel, verzahnt, ...) bearbeiten  
-> Multitasking, Scheduling
- Kommunikation zwischen parallel laufenden Steuerprogrammen  
-> Synchrone & Asynchrone Kommunikation
- Reaktion auf externe Binärsignale reagieren  
-> z.B. Interruptverarbeitung
- Eingangs- und Ausgangssignale verarbeiten  
-> I/O Programmierung
- ...

**In dieser Vorlesung:** Ausgewählte Kapitel aus diesen Gebieten

# Kapitel 1: Zusammenfassung



**Folien zur Vorlesung**  
**System – und Echtzeitprogrammierung (BTI4-SY)**  
**Sommersemester 2010**  
**(Teil 2)**

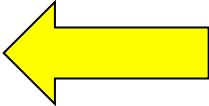
**Prof. Dr. Franz Korf**

[korf@informatik.haw-hamburg.de](mailto:korf@informatik.haw-hamburg.de)



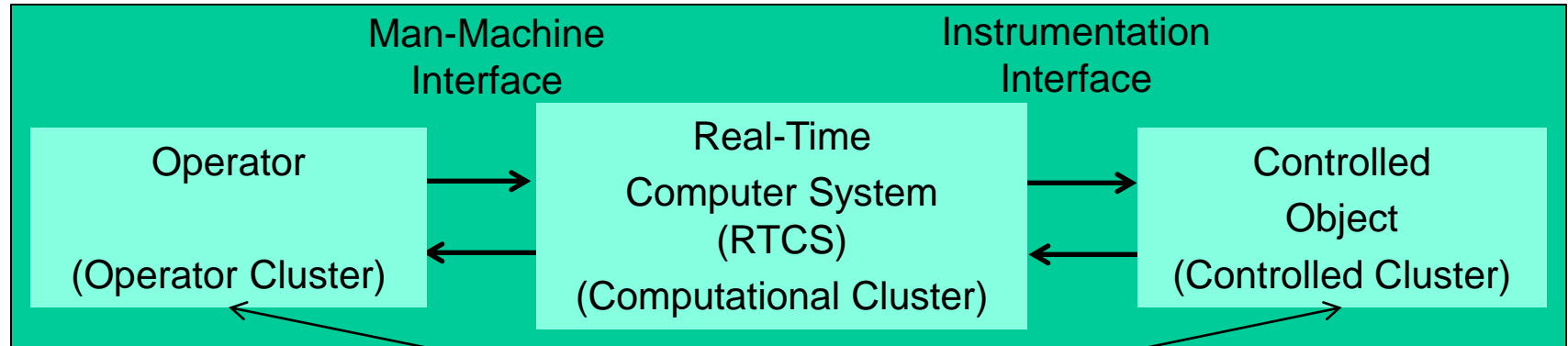
## Kapitel 2 : Definitionen und Begriffsbildung

### Gliederung

- RTS & RTCS 
- Funktionale Anforderungen
- Zeitliche Anforderungen
- Zuverlässigkeits- und Stabilitätsanforderungen
- Klassifizierung
- RTS Markt

Quelle: Kopetz: Real-Time Systems, Design principles for distributed embedded applications, Kluwer Academic Publishers

## Definition: Real-Time System (RTS)



Environment of Real-Time Computer System

### Kommentare

- Ein **real-time system (RTS)** ändert seinen Zustand als Funktion über die Zeit.
- Es ändert seinen Zustand auch, wenn das zugehörige RTCS nicht arbeitet (z.B.: Chemische Anlage, fahrendes Auto & ABS)
- Das RTCS kann auch ein verteiltes System sein
- **MMI**: input- & output devices, nicht nur Bildschirm, Tastatur und Maus
- **Instrumentation Interface**: Beobachtung des controlled object über Sensoren, Steuerung des controlled object über Aktoren

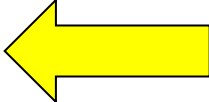
## Definition: Real-Time System (Fortsetzung)

### Kommentare

- Das RTCS muss auf Stimuli (Eingaben) der Umgebung (environment) in definierten Zeitintervallen reagieren.  
**Deadline:** Zeitspanne, bis wann das Ergebnis vorliegen muss.
- **Soft deadline:** Das Ergebnis ist brauchbar, auch wenn es erst nach Überschreitung der Deadline geliefert wird.
- **Firm deadline:** Das Ergebnis ist unbrauchbar, wenn es nach Überschreitung der Deadline geliefert wird.
- **Hard deadline:** Wird das Ergebnis nicht innerhalb der Deadline geliefert, kann es zu einer Katastrophe oder Beschädigung des Systems kommen.
- Ein **hard RTCS** (oder **safety-critical RTCS**) muss mindestens eine harte Deadline erfüllen (entsprechend hard RTS bzw. safety-critical RTS)
- Ein **soft RTCS** muss keine harte Deadline erfüllen (entsprechend soft RTS).
- Unterschiedliche Designanforderungen für hard und soft RTCSs  
Hard RTCS: Es muss stets die zeitlichen Anforderungen einhalten – auch in allen Last-, Fehler- und sonstigen Situationen  
Soft RTCS: Gelegentlich darf eine zeitliche Anforderung nicht erfüllt werden (Auswirkungen auf den kommerziellen Erfolg des Systems etc. werden hier nicht beachtet.)

## Kapitel 2 : Definitionen und Begriffsbildung

### Gliederung

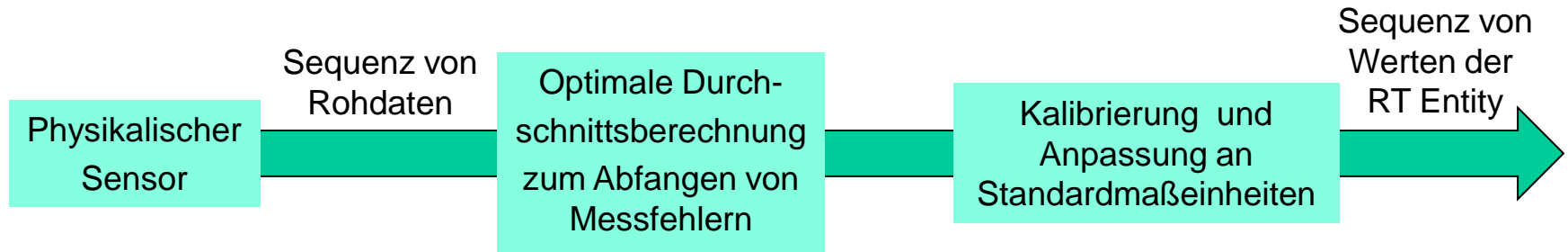
- RTS & RTCS
- Funktionale Anforderungen 
- Zeitliche Anforderungen
- Zuverlässigkeits- und Stabilitätsanforderungen
- Klassifizierung
- RTS Markt

Quelle: Kopetz: Real-Time Systems, Design principles for distributed embedded applications, Kluwer Academic Publishers

## Funktionale Anforderung an das RTCS: Sammlung von Daten

- Eine **Menge von Zustandsvariablen** beschreibt den Zustand eines controlled object , der sich über die Zeit ändert.
- Eine Zustandsvariable ist für die Aufgabe der RTCS relevant oder irrelevant.
- **Real-time entity (RT Entity)**: relevante Zustandsvariable
- **Sphere of control (SOC, Kontrollbereich) einer RT Entity ist das Teilsystem**, das die RT Entity ändern kann.  
Außerhalb des SOC kann die RT Entity nur beobachtet, aber nicht geändert werden.
- Ein RTCS muss die RT Entities beobachten und in einem **RT-Image** abspeichern. Ein RT-Image speichert die Werte der RT Entity zu einem Zeitpunkt.
- Während des **Gültigkeitszeitintervalls** des RT-Images haben die RT Entities die im RT Image gespeicherten Werte.
- Ändert eine RT Entity ihren Wert, muss ein neues RT Image erstellt werden – das Gültigkeitszeitintervall des alten RT-Image ist abgelaufen.
- **Time Triggered (TT) observation**: Das RT Image wird periodisch aktualisiert.
- **Event Triggered (ET) observation**: Das RT Image wird aktualisiert, sobald sich der Wert einer RT-Entity ändert.

## Signalaufbereitung (Signal Conditioning)



- Nach der Signalaufbereitung wird eine Plausibilitätsprüfung der Daten durchgeführt  
→ Erkennung von fehlerhaften Sensoren
- **Agreed data element:** Der Wert einer RT Entity, der die Plausibilitätsprüfung erfolgreich bestanden hat.

## Alarmüberwachung (Alarm Monitoring)

- Das RTCS muss auf Basis der RT Entities das Controlled Object kontinuierlich auf ein anormales Verhalten überprüfen, diese melden bzw. darauf reagieren
- Oftmals tritt in diesem Fall eine Vielzahl von Meldungen auf (**alarm shower, Alarmflut**).
- Das RTCS muss die Analyse der Ursache unterstützen.
  - Log Datei mit exaktem Zeitstempel
  - Wissensbasis
- **Herausforderungen:**
  - Garantie eines vorhersehbaren Verhaltens der RTCS während einer Alarmflut
  - Eine ernste oder gefährliche Situation des RTS, die sehr selten auftritt, heißt **rare-event**. Das richtige Fehlerhalten des RTCS während eines rare-event ist eine wesentliche und oftmals schwierige Anforderung

## Alarmüberwachung (Alarm Monitoring)

- Das RTCS muss auf Basis der RT Entities das Controlled Object kontinuierlich auf ein anormales Verhalten überprüfen, diese melden bzw. darauf reagieren
- Oftmals tritt in diesem Fall eine Vielzahl von Meldungen auf (**alarm shower, Alarmflut**).
- Das RTCS muss die Analyse der Ursache unterstützen.
  - Log Datei mit exaktem Zeitstempel
  - Wissensbasis
- **Herausforderungen:**
  - Garantie eines vorhersehbaren Verhaltens der RTCS während einer Alarmflut
  - Eine ernste oder gefährliche Situation des RTS, die sehr selten auftritt, heißt **rare-event**. Das richtige Fehlerhalten des RTCS während eines rare-event ist eine wesentliche und oftmals schwierige Anforderung



## Funktionale Anforderung an das RTCS: Direct Digital control

- Viele RTCS müssen die Aktoren zur Steuerung des Controlled Object direkt ansprechen (**direct digital control – DDC**).
- Typischer Ablauf der RTCS für DDC:

```
while (1) {  
    // Sampling of RT entities  
    ...  
    // execution of control algorithm  
    ...  
    // calculate new set points for actors  
    ...  
    // output set points to actors  
    ...  
}
```

## Funktionale Anforderung an das RTCS: MMI

### ➤ Aufgaben

- Information des Operators über den aktuellen Zustand des RTS
- Unterstützung des Operators bei der Steuerung des RTS

### ➤ Usability

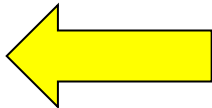
- Zentraler Aspekt
- Studien belegen: Viele katastrophale Fehler in computergestützten safety-critical RTS können auf Fehler im MMI zurückgeführt werden.

### ➤ Qualitätskontrolle

- Kontinuierliche Aufzeichnung der Prozessdaten

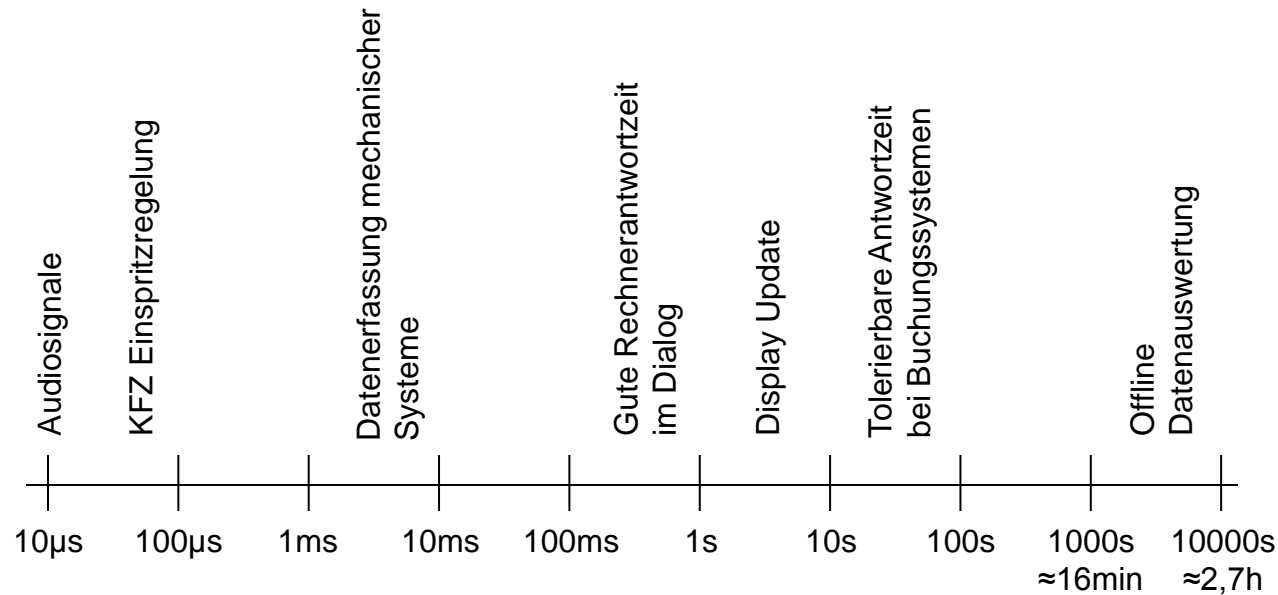
## Kapitel 2 : Definitionen und Begriffsbildung

### Gliederung

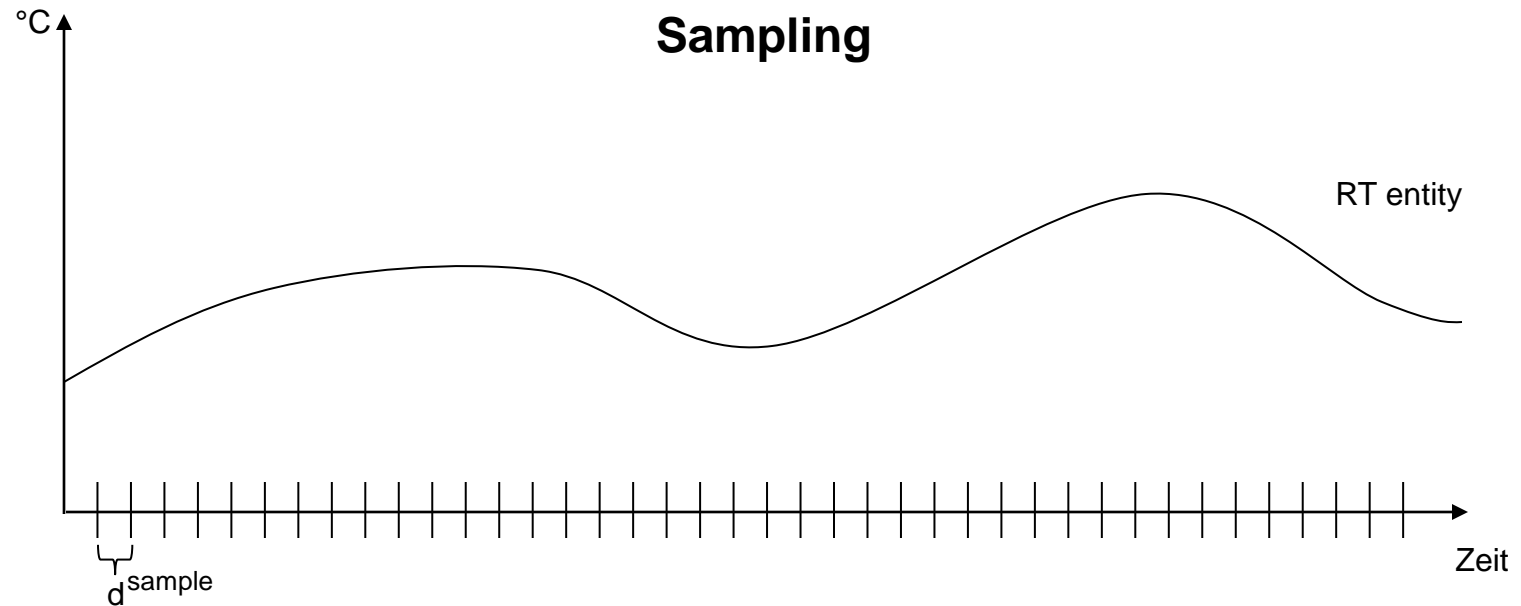
- RTS & RTCS
- Funktionale Anforderungen
- Zeitliche Anforderungen 
- Zuverlässigkeits- und Stabilitätsanforderungen
- Klassifizierung
- RTS Markt

Quelle: Kopetz: Real-Time Systems, Design principles for distributed embedded applications, Kluwer Academic Publishers

## Typische Zeiten



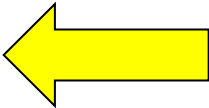
- RTCS: Die Zeitanforderungen oftmals von der Anwendung und nicht vom MMI
- Das Controlled Object gibt die Zeitanforderungen an das RTCS vor



- Eine RT entity wird mit der Sampling Periode  $d^{\text{sample}}$  abgetastet.
- Sampling Frequenz  $f^{\text{sample}} = (d^{\text{sample}})^{-1}$
- Die Sampling Frequenz hängt von der Anwendung ab

## Kapitel 2 : Definitionen und Begriffsbildung

### Gliederung

- RTS & RTCS
- Funktionale Anforderungen
- Zeitliche Anforderungen
- Zuverlässigkeits- und Stabilitätsanforderungen 
- Klassifizierung
- RTS Markt

Quelle: Kopetz: Real-Time Systems, Design principles for distributed embedded applications, Kluwer Academic Publishers

## Zuverlässigkeit (Reliability)

- Sei  $\lambda$  die konstante **Fehlerrate** eines RTS (Anzahl Fehler/Stunde,  $\lambda \geq 0$ )
- Die **Zuverlässigkeit**  $R(t)$  des RTS ist die Wahrscheinlichkeit, dass das System bis zum Zeitpunkt  $t$  fehlerfrei arbeitet.

$$R(t) = e^{(-\lambda(t-t_0))}$$

wobei das RTS zum Zeitpunkt  $t_0$  betriebsbereit war und die Zeiten in Stunden angegeben werden.

- Falls  $\lambda \leq 10^{-9}$  *failures / h* ist eine **ultrahigh reliability** gegeben.
- Die **MTTF (Mean-Time-To-Failure)** in Stunden ergibt als

$$MTTF = \frac{1}{\lambda}$$

## Safety

- **Safety** bezeichnet die Zuverlässigkeit bezüglich kritischer Fehlerzustände.
- In einem kritischen Fehlerzustand (malign error mode) können die Kosten um Größenordnungen höher sein als der Nutzen im Normalzustand.
- Im Bezug auf Safety muss die Fehlerrate ( $\lambda$ ) eines safety-critical RTS die ultrahigh reliability Anforderung erfüllen.
- **Beispiele**
  - Die Fehlerrate eines Bremssystems mit Computern muss kleiner sein als die Fehlerrate einer konventionellen Bremsanlage.
  - Beispielrechnung:
    - Ein Auto fährt im Durchschnitt eine Stunde pro Tag
    - Ein kritischer Fehler pro Jahr bei einer Million Autos führt zu einer Fehlerrate von  $10^{-9} \text{ failures} / h$



## Wartbarkeit (Maintainability)

- Die **Maintainability** ist ein Maß für die Reparaturzeit eines RTS, nachdem ein nicht kritischer Fehler (benign error) aufgetreten ist.

Analog zur Zuverlässigkeit werden Wahrscheinlichkeiten und eine repair rate  $\mu$  verwendet.

- Die **MTTR (Mean-Time-To-Repair)** in Stunden ergibt als  $MTTR = \frac{1}{\mu}$

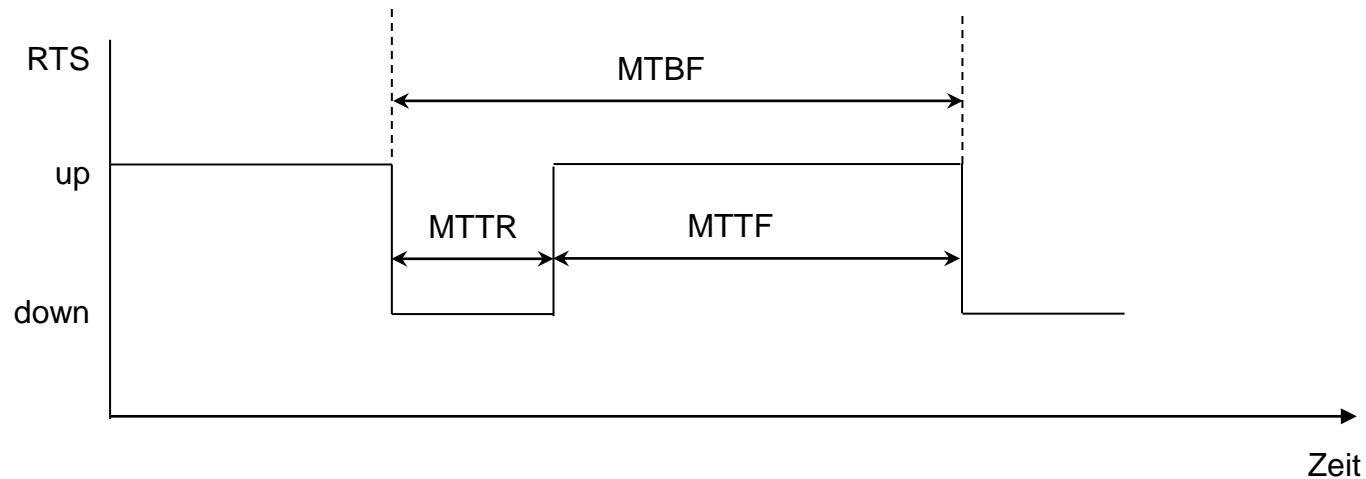
- **Grundlegenden Konflikt** zwischen Wartbarkeit (Maintainability) und Zuverlässigkeit (Reliability)

Ein RTS mit vielen kleinen, leicht austauschbaren Komponenten führt zu kurzen Reparaturzeiten. Aufgrund der vielen elektrischen und mechanischen Schnittstellen steigt die MTTF, was die Zuverlässigkeit reduziert.

## Verfügbarkeit (Availability)

- Die **Verfügbarkeit (Availability) A** eines RTS ist die Wahrscheinlichkeit, dass das RTS zu einem gegebenen Zeitpunkt funktioniert.

$$A = \frac{MTTF}{MTTF + MTTR} = \frac{\mu}{\lambda + \mu}$$




- **MTBF: Mean Time Between Failures**

## Sicherheit (Security)

- Bei Real-Time Systemen steigt die Bedeutung von Security Aspekten an – verhindern des nicht autorisierten Zugriffs auf das System oder Teile des Systems.
- **Beispiele**
  - Angriffe auf sicherheitsrelevante Anlagen
  - Entertainment im Automobil

## Kapitel 2 : Definitionen und Begriffsbildung

### Gliederung

- RTS & RTCS
- Funktionale Anforderungen
- Zeitliche Anforderungen
- Zuverlässigkeits- und Stabilitätsanforderungen
- Klassifizierung 
- RTS Markt

Quelle: Kopetz: Real-Time Systems, Design principles for distributed embedded applications, Kluwer Academic Publishers

## Hard RTS versus soft RTS

Eigenschaft	Hard RTS	Soft RTS
Deadlines	müssen eingehalten halten	sollten meistens eingehalten werden
Performance bei hoher Last (peak-load)	exakt vorhersagbar	darf nachlassen
Safety	gefordert	nicht gefordert
Datengröße	klein - mittel	oftmals groß
Datenintegrität	kurzfristig	langfristig
Fehlererkennung	automatisch	Unterstützung des Anwenders

## Kommentare

- **Response Time:** Hard RTS → Response Time oftmals im Bereich weniger Millisekunden und weniger → automatische Steuerung ohne Interaktion des Operators notwendig
- **Peak-load Performance:** Hard RTS → Deadlines müssen stets eingehalten werden, auch bei hoher Systemlast  
Hohe Systemlast und rare-events treten oftmals zeitlich auf.  
Soft RTS → die durchschnittliche Performance ist entscheidend.
- **Synchronismus RTCS & Umgebung (environment):** Ein hard RTCS muss stets zum Zustand der Environment synchron sein → es muss alle Zustandsänderungen der Umgebung erfassen und pünktlich verarbeiten.  
Bei einem Soft RTCS ist dies nicht notwendig → kann die Umgebung z.B. zeitweise „ausbremsen“

## Kommentare

- **Datengröße:** Die Datengröße wird hauptsächlich durch das aktuelle Image der RT Entities bestimmt, das in kurzen Abständen aktualisiert wird.
- **Datenintegrität:** Dateninkonsistenz in einem Buchungssystem (soft RTS) → Recovery durch Roll-back zum letzten Checkpoint.

Hard RTS: Roll-back oftmals nicht möglich, weil:

- Einhaltung von Deadlines ist gefährdet. Tritt auch beim Reboot des RTCS auf
- Die seit dem Checkpoint durchgeführten Ansteuerungen der Aktoren und somit die Steuerung der Controlled Objects ist oftmals irreversibel
- Gültigkeitszeitintervall des RT-Images des Checkpoints ist oftmals abgelaufen.

## Fail-Save versus Fail-Operational

- **Save State:** Ein Zustand des RTS, der im Fehlerfall erreicht werden kann, in dem kein Schaden auftreten kann (kein Save State, ein Save State, mehrere Save States).
- **Fail-Save RTS:** Ein RTS heißt Fail-Save, wenn in jedem Fehlerfall ein Save State erkannt und erreicht werden kann. Dies muss im vorgegebenen Zeitrahmen stattfinden.

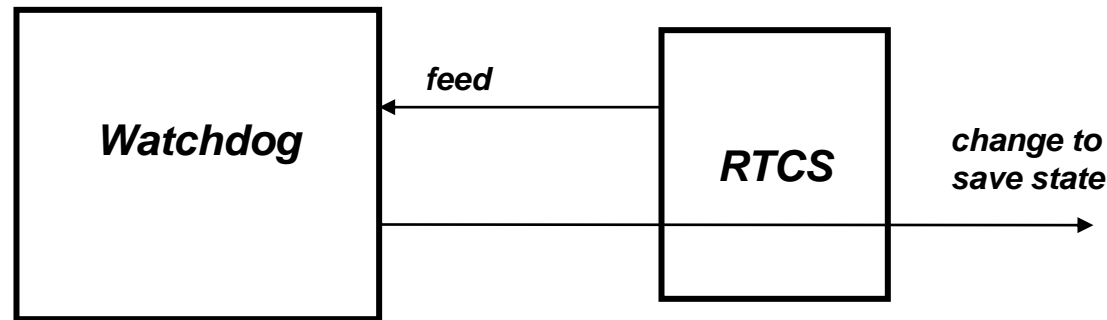
Fail-Save ist eine Eigenschaft des Controlled Objects, nicht des RTCS. Aber: Die Wahrscheinlichkeit, dass das RTCS einen auftretenden Fehler erkennt, muss gegen 1 streben.

- **Fail-Operational:** Existiert kein Save State für das Controlled Object, muss das RTCS einen minimalen Funktionsumfang bereitstellen, so dass keine Katastrophe auftritt.



## Fail-Save versus Fail-Operational

### Watchdog



- Der Watchdog überwacht das RTCS
- Das RTCS muss sich periodisch beim Watchdog melden (feed)
- Bleibt dies aus, fährt der Watchdog das RTS in einen passenden Save State
- Externer vs. interner Watchdog, Realisation des Watchdogs durch einen Timer
- Ein Watchdog, der einen Reboot des RTCS durchführt, ist nicht immer zielführend (z.B.: verpasste Deadlines, Reboot löst das Problem nicht)
- Watchdogs decken nur Fehler bezüglich der Verfügbarkeit (availability) ab

## Guaranteed Response versus Best Effort

- **Guaranteed Response:** Detaillierte Analyse der Fehler und Lastsituation (ohne Wahrscheinlichkeitsbetrachtungen) und deren konsequente Umsetzung vom Systemdesign, über die Wahl der Komponenten bis zum Test führen zu einem RTCS mit garantierten Antwortzeiten.
- **Best Effort:** Sind diese Anforderungen bezüglich Design und Umsetzung nicht realisierbar, bezeichnet man das RTCS als Best Effort RTCS.
- **Rare events:** Das zeitlich und funktional korrekte Verhalten eines Best Effort RTCSs kann in der Regel nicht garantiert werden.

## Resource- Adequate versus Resource-Inadequate

- **Principle of Resource-Adequate:** Das RTCS ist mit genügend Ressourcen ausgestattet, so dass es auch in hohen Lastsituationen und im Fehlerfall die gestellten Anforderungen (insbesondere Deadlines) erfüllen kann.
- **Principle of Resource-Inadequate:** Aus u.a. ökonomischen Gründen ist das RTCS mit ausreichend Ressourcen für den „normalen Betrieb“ ausgestattet. In Lastsituationen und im Fehlerfall wird Resource Sharing zugelassen, auch wenn dann die Anforderungen an das RTCS nicht mehr garantiert werden können.
- **Hard RTCS:** Principle of Resource-Adequate unabdingbar
- **Soft RTCS:** In Zukunft wird wahrscheinlich auch bei Soft RTCS ein Wechsel zum Principle of Resource-Adequate schrittweise stattfinden.

## Event-Triggered versus Time-Triggered

- Ein event-triggered RTS reagiert „sofort“ auf externe Events (Änderung des Werts einer RT Entity), die i.allg. durch einen Interrupt dem System mitgeteilt werden.

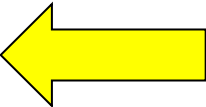
Problem: Häufung von Interrupts

- In äquidistanten Zeitabständen, die gemäß dem RTS festgelegt werden, reagiert ein time-triggered System auf externe Ereignisse bzw. fragt, ob ein externes Ereignis aufgetreten ist. Wird oftmals durch einen HW Timer angestoßen. Die Zeitabstände werden in Abhängigkeit vom Controlled Object gewählt.

Vorteil: keine Häufung von Interrupts

## Kapitel 2 : Definitionen und Begriffsbildung

### Gliederung

- RTS & RTCS
- Funktionale Anforderungen
- Zeitliche Anforderungen
- Zuverlässigkeits- und Stabilitätsanforderungen
- Klassifizierung
- RTS Markt 

Quelle: Kopetz: Real-Time Systems, Design principles for distributed embedded applications, Kluwer Academic Publishers

## RTS Markt

- Bei den meisten RTSs ist das Kosten-Leistung-Verhältnis ein entscheidender Parameter für den Markterfolg. Weitere wichtige Parameter, wie z.B. Zielgruppe, TTM, Lifestyle, Werbung etc. werden hier nicht betrachtet.
- Gesamtkosten eines Produkts: Entwicklungskosten, Produktionskosten, Wartungskosten
- **Eingebettete Echtzeitsysteme:** Aufgrund des fallenden Preis/Leistungsverhältnisses von  $\mu C$  lösen RTCSs immer mehr mechanische und elektrische Kontrollsysteme ab. Das MMI bleibt oftmals gleich, so dass der Wechsel zum RTCS von außen nicht erkennbar ist.

Ein **eingebettetes System** enthält mindestens einen programmierbaren Computer (Micro Controller, DSP etc.). Es ist oftmals nicht erkennbar, dass es sich um ein System mit einem Computer handelt.

## RTS Markt

- Bei eingebetteten Systemen für den **Massenmarkt** müssen die Produktionskosten so gering wie möglich sein → optimale Speicher und CPU Auslastung
- Eingebettete Systeme haben eine **statische Struktur**, die zur Entwicklungszeit bekannt ist. Dies wird in der Entwicklung zur Reduktion der Komplexität genutzt.
- MMI: Muss dem Einsatzgebiet des eingebetteten Systems entsprechen, am besten selbsterklärend.
- Reduktion der Produktionskosten durch eine Reduktion der mechanischen Komponenten aufgrund des RTCS.
- Falls das Programm des RTCS im ROM steht: kein Update vor der Auslieferung oder im Feld möglich → hohe Qualitätsanforderungen an die SW.
- An den Gesamtkosten eines eingebetteten Systems für den Massenmarkt haben die Entwicklungskosten oftmals einen geringen Anteil (< 5%). Entscheidend sind Produktions- und Wartungskosten (z.B.: Rückrufaktionen).



## 4 Phasen von Computersystemen in Produkten

- **Phase 1:** Eine ad hoc Stand-alone Implementierung auf einem  $\mu\text{C}$  löst das konventionelle Kontrollsystem ab. Die SW wird von Ingenieuren mit viel Systemwissen und wenig Informatikwissen entwickelt.
- **Phase 2:** Die Funktionalität des Produkts wird durch weitere SW Funktionen gesteigert.
- **Phase 3:** Die steigende Komplexität der Software durch weitere SW Funktionen führt zu Stabilitätsproblemen. Ein Redesign der SW wird durchgeführt → die Entwicklungskosten steigen signifikant an, ohne dass eine **sichtbare** Aufwertung des Produkts stattfindet. Kritische Phase
- **Phase 4:** Das Produkt wird Teil einer größeren Umgebung und benötigt Kommunikationsschnittstellen zur Umgebung



## Kapitel 2 : Definitionen und Begriffsbildung

### Zusammenfassung



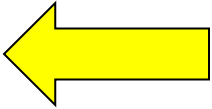
**Folien zur Vorlesung**  
**System – und Echtzeitprogrammierung (BTI4-SY)**  
**Sommersemester 2010**  
**(Teil 3)**

**Prof. Dr. Franz Korf**

[korf@informatik.haw-hamburg.de](mailto:korf@informatik.haw-hamburg.de)

## Kapitel 3 : Inter Process Communication (IPC)

### Gliederung

- Grundbegriffe der Inter Process Communication (IPC) 
- Synchrones Message Passing: die Grundlage des QNX Realtime OS
- Client / Server Konzepte
- Asynchrone Kommunikation: QNX Pulses
- Zusammenfassung

## IPC: Inter Process Communication

- Das Thema IPC wird in der Vorlesung Betriebssysteme behandelt.
- Diese Einleitung wiederholt grundlegende IPC Themen.
- Gemäß der aktuellen Begriffsbildung findet die Kommunikation zwischen Threads (und nicht zwischen Prozessen) statt. Die Threads können in unterschiedlichen Prozessen ablaufen.
- Es gibt zwei prinzipiell unterschiedliche Ansätze:
  - IPC via shared memory
  - IPC via message passing

## IPC via shared memory

- Die Threads operieren auf einem gemeinsamen Speicherbereich  
→ Synchronisationsmechanismen zwischen den Threads notwendig
- Synchronisationsmechanismen
  - **Mutex** (binärer Semaphore): „atomare“ Operationen: lock & unlock, zu einem Zeitpunkt kann nur ein Thread den Mutex locken, Technik zum Schutz von kritischen Bereichen
  - **Semaphor**: Ein Zähler mit „atomaren“ Zugriffsoperationen (wait und post), „wait“ auf Semaphor mit Wert 0 → Synchronisationspunkt, Semaphor als Füllstandanzeiger
  - **Barrier**: N Threads warten aufeinander und passieren die Barriere gleichzeitig
  - **Conditional Variable**: Synchronisation bezüglich Events
  - ...

## IPC via shared memory: Beispiel Mutex

```
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int count = 0; // gemeinsamer Speicher

void* thread1(void* notused) {
    int tmp = 0;
    while(1) {
        /* Begin critical section */
        pthread_mutex_lock(&mutex);
        tmp = count++;
        pthread_mutex_unlock(&mutex);
        /* End critical section */
        printf("Count is %d\n", tmp);
        sleep(1);
    }
    return 0;
}
```

## IPC via shared memory: Beispiel Mutex (Fortsetzung)

```
void* thread2(void* notused)
{
    int tmp = 0;

    while(1) {
        /* Begin critical section */
        pthread_mutex_lock(&mutex);
        tmp = count--;
        pthread_mutex_unlock(&mutex);
        /* End critical section */
        printf("** Count is %d\n", tmp);
        sleep(2);
    }
    return 0;
}

int main( void )
{
    pthread_create(NULL, NULL, thread1, NULL);
    pthread_create(NULL, NULL, thread2, NULL);

    sleep(60);
}
```

Ein Mutex stellt nicht sicher, dass nur der Thread auf die kritische Ressource zugreift, der den Mutex „besitzt“ – auf locked gesetzt hat. **Das muss die Anwendung garantieren.**

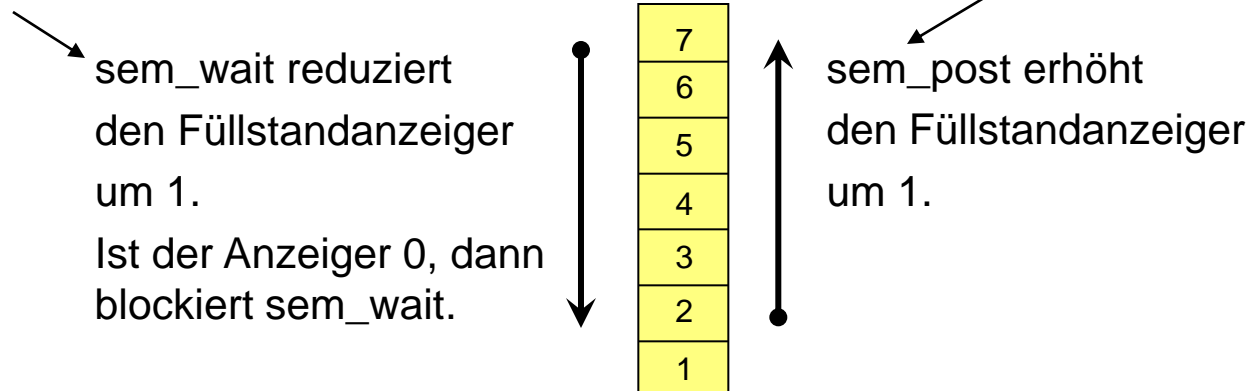
Warten mehrere Threads auf einen Mutex, weist der Kernel den Mutex dem Thread mit der höchsten Priorität zu - sobald der Mutex freigegeben ist.

## IPC via shared memory: Semaphor als Füllstandanzeiger

### Eine Ressource wird belegt:

`sem_wait` reduziert den Füllstandanzeiger.

Alle Ressourcen sind belegt: `sem_wait` blockiert, bis eine Ressource verfügbar ist.



Füllstandanzeiger: Semaphor mit maximalem Füllstand initialisiert.

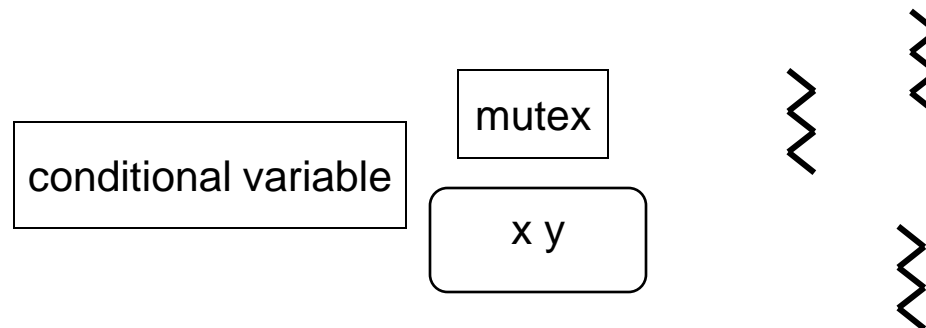
**Beispiel:** 7 Instanzen einer Ressource stehen zur Verfügung.  
Ein Füllstandanzeiger (Semaphor mit 7 initialisiert) verwaltet die Anzahl der noch verfügbaren Ressourcen.



# IPC via shared memory: Conditional Variables

## Situation

- Innerhalb ihres Ablaufes warten die Threads darauf, dass die Variablen x und y eine Bedingung erfüllen. Die Bedingungen können variieren.
- Die Threads verändern x und y.
- x und y sind über einen Mutex geschützt.
- Über eine conditional variable wird ein Thread, der auf das Eintreten einer Bedingung wartet, informiert, dass er die Bedingung erneut überprüfen soll.



## IPC via shared memory: Conditional Variables

```
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int x = 0; int y = 0;

int thread( void ) {
    while( 1 ) {
        ...
        // pattern: Überprüfe Bedingung, modifiziere Variablen und wecke
        // einen Thread
        pthread_mutex_lock( &mutex );
        while(! ( x == 0) && (y > 12)) )
            pthread_cond_wait( &cond, &mutex );
        ... // modifiziere x und y
        pthread_cond_signal( &cond ); // benachrichtige einen Thread,
                                         // der auf die Cond. Var. wartet
        pthread_mutex_unlock( &mutex );
        ...
        // pattern: Modifizieren und alle Threads wecken
        pthread_mutex_lock( &mutex );
        while(! ( x == 0) && (y > 12)) )
            pthread_cond_wait( &cond, &mutex );
        ... // modifiziere x und / oder y
        pthread_cond_broadcast ( &cond );
        pthread_mutex_unlock( &mutex );
        ...
    }
    return( 0 );
}
```

## IPC via message passing

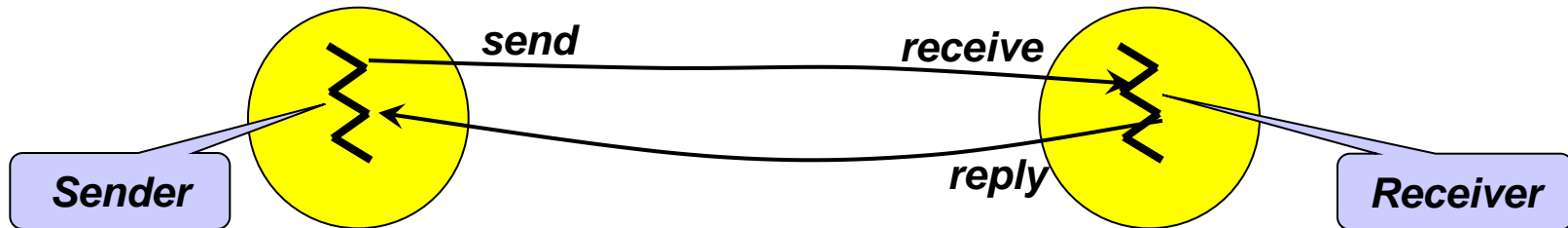
- Threads kommunizieren über den Austausch von Nachrichten
- Vorteil: Implementierung in verteilten - und SMP Systemen möglich
- Grundlegende Operationen:
  - send(Ziel, Nachricht)
  - receive(Quelle, Nachricht)
- Relevante Faktoren von Message Passing Systemen:
  - Synchronisation
  - Adressierung
  - Format
  - Warteschlangenverfahren

## IPC via message passing: Synchronisation

- Fundamentale Bedingung: Der Empfänger kann eine Nachricht erst empfangen, nachdem der Sender sie abgeschickt hat
- **Blockierendes send:** Die send Operation blockiert, bis der Empfang der Nachricht bestätigt wird.
- **Nicht blockierendes send:** Die send Operation ist beendet, sobald die Nachricht abgeschickt wurde.
- **Blockierendes receive:** Wenn zuvor eine Nachricht gesendet wurde, wird diese empfangen und die Ausführung fortgesetzt. Ansonsten blockiert die receive Operation, bis eine Nachricht eingetroffen ist.
- **Nicht blockierendes receive:** Wenn zuvor eine Nachricht gesendet wurde, wird diese empfangen und die Ausführung fortgesetzt. Ansonsten wird die Ausführung mit einer entsprechenden Fehlermeldung fortgesetzt.

## IPC via message passing: Synchronisation

**Blockierendes send & blockierendes receive = synchrone Kommunikation (in QNX)**



- Oftmals werden das Warten auf eine Nachricht (receive) und die Antwort an den Sender (reply) in zwei Operationen aufgeteilt.
- Synchronisation des Senders: send blockiert die Ausführung des Senders, bis die Antwort/Bestätigung des Receivers eingetroffen ist.
- Synchronisation des Receivers: receive blockiert die Ausführung des Receivers, bis eine Nachricht eingetroffen ist.
- Oftmals schickt reply nicht nur eine Empfangsbestätigung sondern auch eine Antwort an den Sender.

## IPC via message passing: Synchronisation

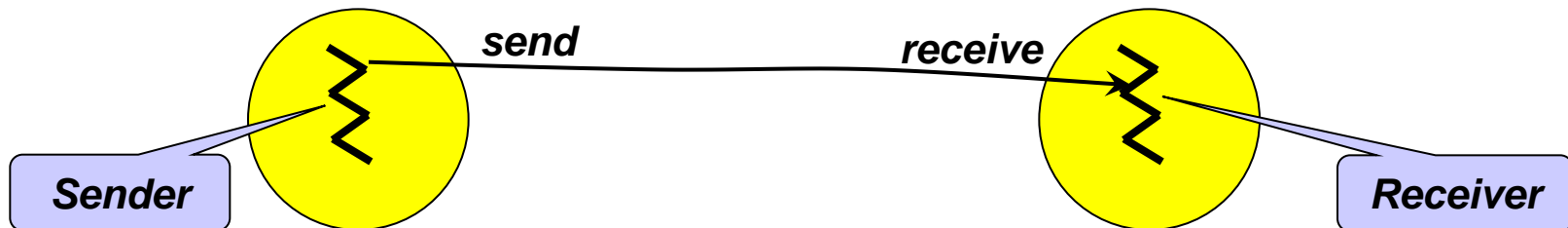
### Nicht blockierendes send & blockierendes receive



- Ggf. Pufferung von nicht abgeholten Nachrichten.
- Dieser Puffer sollte so groß sein, dass er niemals voll ist (kleine Nachrichten sind sehr hilfreich). Ist der Puffer trotzdem voll, kann send auf drei Arten reagieren:
  - send liefert eine Fehlermeldung
  - send blockiert, bis die Nachricht in den Puffer gelegt werden kann
  - send ignoriert die Situation
- Diese Kommunikation wird in QNX als asynchrone Kommunikation bezeichnet.

## IPC via message passing: Synchronisation

### Nicht blockierendes send & nicht blockierendes receive



- Send Operation und Pufferung wie im vorherigen Fall
- **Cyclical Asynchronous Buffers (CAB):** Es wird immer nur eine Nachricht gepuffert und beim Senden einer neuen Nachricht überschrieben. Receive liest stets eine Nachricht aus. Wurde zwischen zwei receives kein send ausgeführt, liest receive die alte Nachricht nochmals.

**Vorteil:** Sowohl send als auch receive blockieren nie.

Wird auch für one-to-many Kommunikation eingesetzt

**Einsatzgebiet:** Zyklische Aufgaben wie Kontrollanwendungen und Sensordatenerfassung.

## IPC via message passing: Adressierung

### ➤ Direkte Adressierung

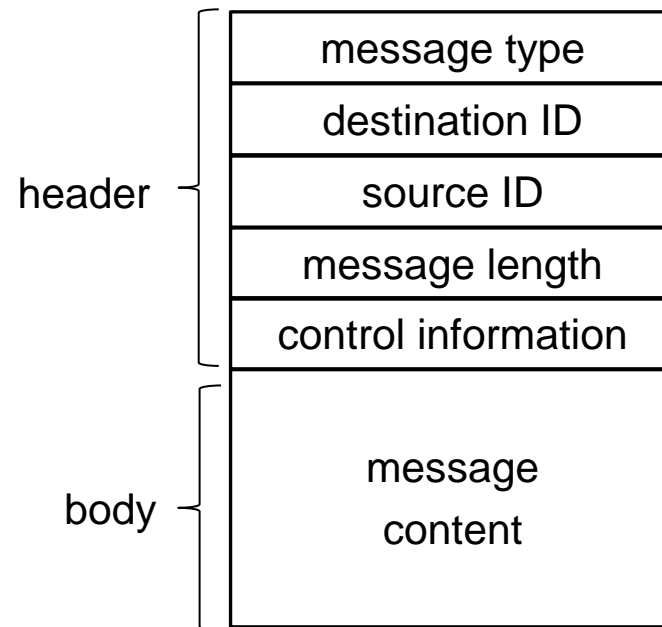
- send Operation: als Ziel wird eine spezifische Kennung des Zielprozesses verwendet
- receive Operation: (a) als Quelle wird der sendende Prozess benannt.  
Problem: Sender ist nicht immer bekannt (z.B.: Druckerspooler)  
(b) Implizite Adressierung – die Adresse muss nicht angegeben werden, da sie implizit durch die receive Operation vorgegeben ist.

### ➤ Indirekte Adressierung

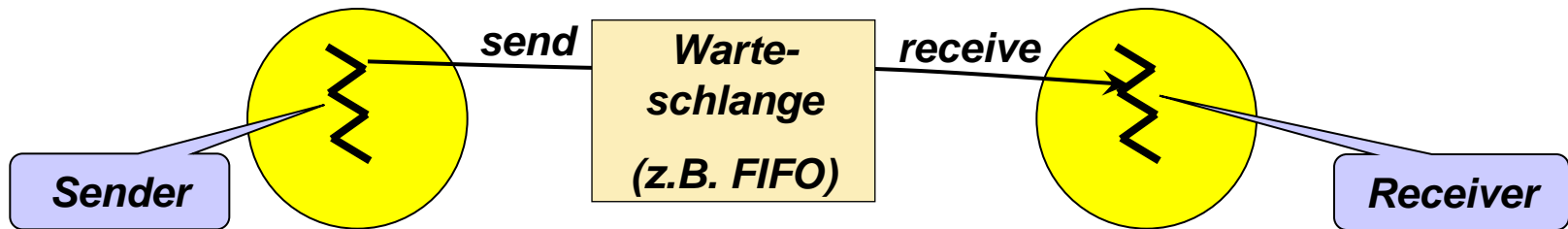
- Die Nachrichten werden nicht direkt vom Sender an den Empfänger geschickt, sondern an eine gemeinsame Datenstruktur, die auch Daten puffern kann.
- 1-zu-1-Kommunikation, n-zu-1-Kommunikation (Client-Server, Port), 1-zu-n-Kommunikation, n-zu-n-Kommunikation möglich



## IPC via message passing: Nachrichtenformat

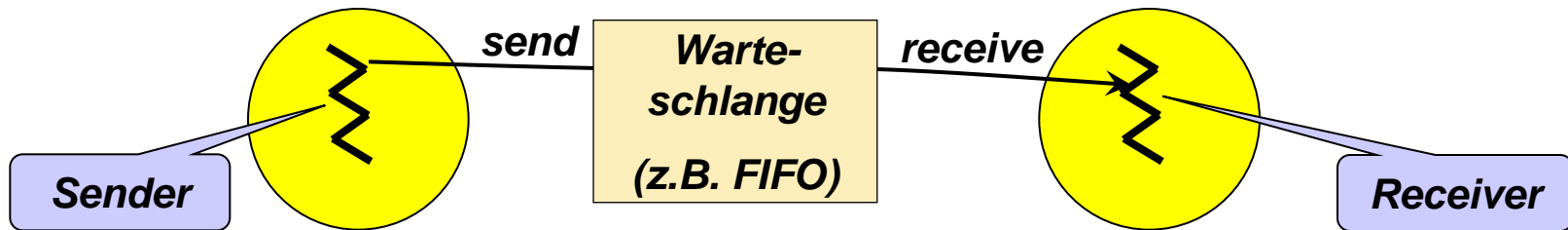


## IPC via message passing: Warteschlangenverfahren



- Der Sender schreibt in einen Puffer, der Receiver liest aus diesem Puffer.
- Aufgrund des Puffers werden Sender und Receiver nicht synchronisiert.
- Die Kommunikation zwischen Warteschlange und Sender bzw. zwischen Warteschlange und Receiver kann auch synchron sein – trotzdem ist die Kommunikation zwischen Sender und Receiver aufgrund der Warteschlange asynchron.
- Als Warteschlange kann auch eine FIFO mit Prioritäten verwendet werden.
- In vielen OS durch die `message_queue` Funktionalität realisiert.

## Warteschlangenverfahren (Fortsetzung)



Send wird gestartet und die Warteschlange ist voll:

- **blockierendes send:** send blockiert den Sender, bis die Warteschlange die Nachricht aufnimmt.
- **nicht blockierendes send:** send blockiert den Sender nicht (und liefert eine Fehlermeldung).

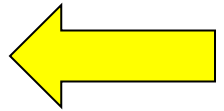
Receive wird gestartet und die Warteschlange ist leer:

- **blockierendes receive:** receive blockiert den Receiver, bis eine Nachricht aus der Warteschlange gelesen werden kann.
- **nicht blockierendes receive:** receive blockiert den Receiver nicht (und liefert eine Fehlermeldung).

## Kapitel 3 : Inter Process Communication (IPC)

### Gliederung

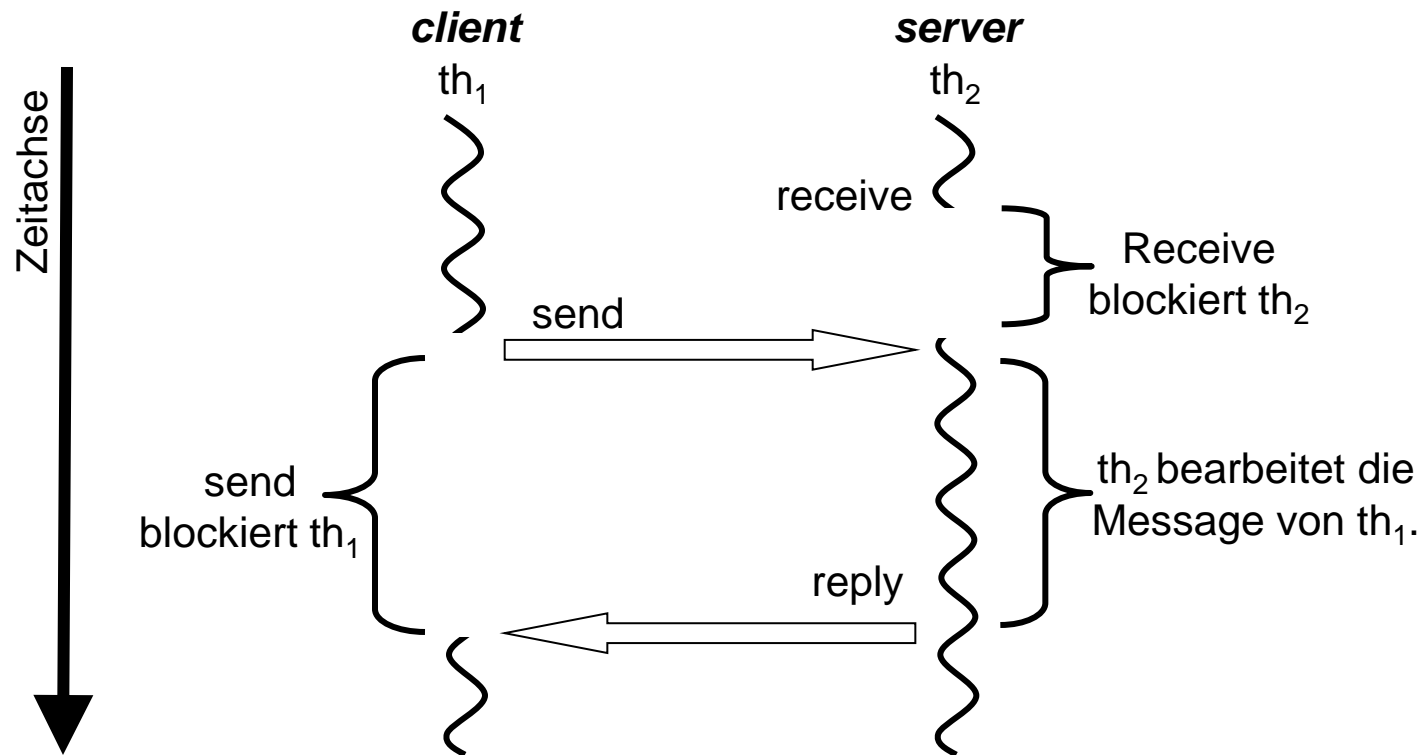
- Grundbegriffe der Inter Process Communication (IPC)
- Synchrones Message Passing: die Grundlage des QNX Realtime OS
- Client / Server Konzepte
- Asynchrone Kommunikation: QNX Pulses
- Zusammenfassung



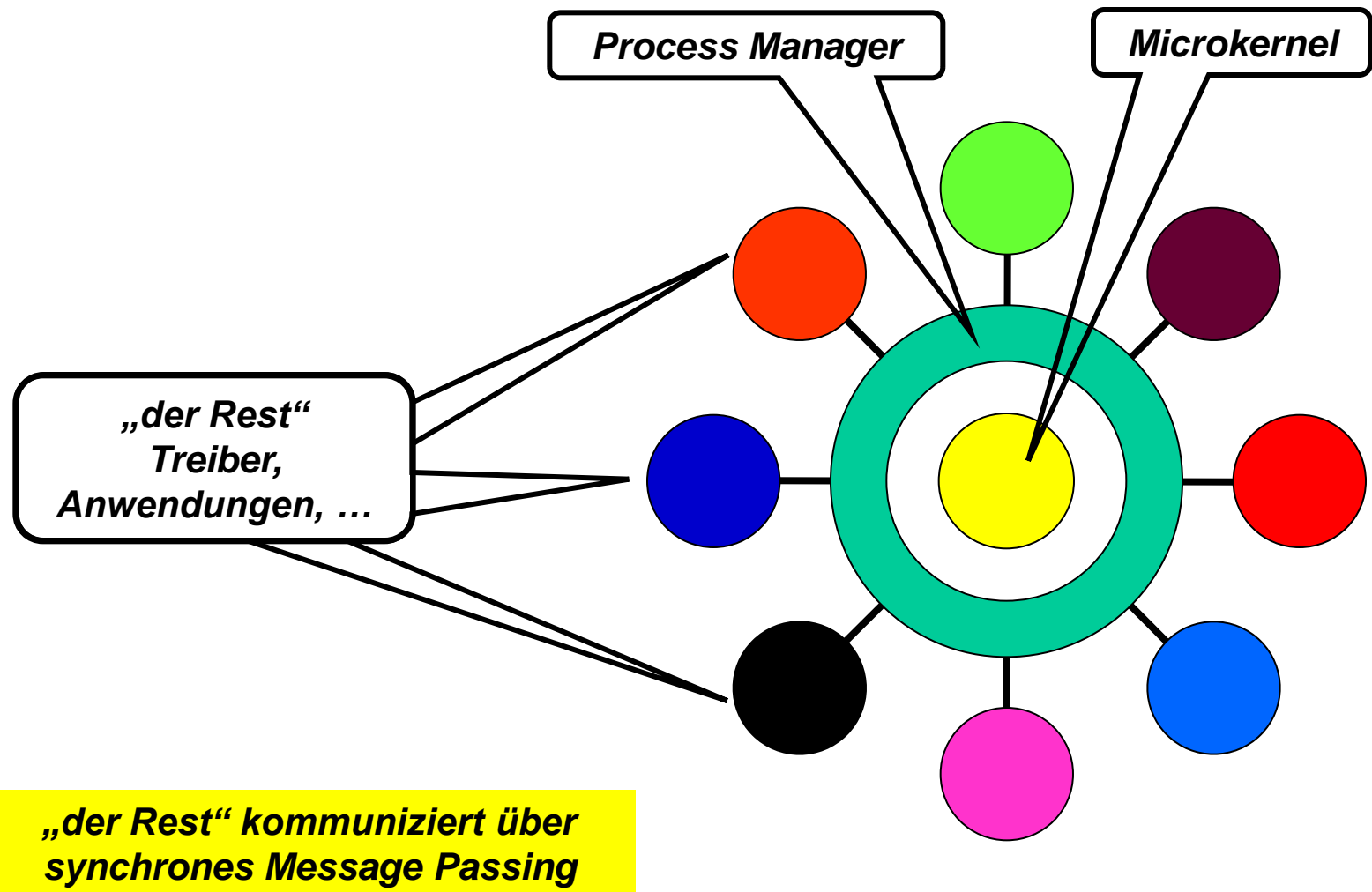
## QNX: Synchrone Kommunikation via Message Passing

### Eigenschaften:

- synchrone Kommunikation → send und receive sind blockierend
- Client/Server Kommunikationsmodell



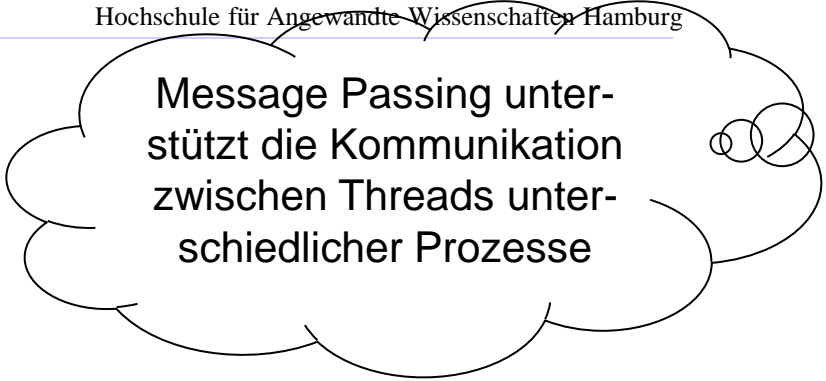
## Synchrones Message Passing: das Kernstück der Kommunikation in QNX



## Diskussion dieses Ansatzes

- Optimale Kapselung der einzelnen Komponenten.
  - Komponenten können leicht ausgetauscht werden (zur Laufzeit).
  - Skalierbarkeit des OS: Komponenten können problemlos aus dem OS entfernt werden (Größe des OS kann auf die Umgebung angepasst werden).
- Hohe Stabilität basierend auf dem synchronen und modularen Ansatz.
- Hohe Wiederverwendbarkeit
  - z.B. Es ist egal, auf welchem Rechner im Netz ein Filesystem liegt - es muss nur eine andere Verbindung geöffnet werden. Der Rest geht stets über Message Passing. Somit kann der Treiber wieder verwendet werden.
- Möglicher Nachteil: Viele Kopiervorgänge. Diese hat QNX sehr effizient realisiert.

## Beispiel



Message Passing unterstützt die Kommunikation zwischen Threads unterschiedlicher Prozesse

```
#include <cpt_terminal.h>
#include <pthread.h>
#include <errno.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/neutrino.h>
```

```
int    chid = -1; // channel id, globale Variable, Synchronisation nötig

void *client (void * not_used)
{
    int    node_id = 0;    // 0 : my node
    pid_t  pid = 0;        // 0 : my process
    int    coid;           // coid = ConnectionId
    char *s_msg = "My first message";
    char  r_msg[512];

    do { // connect to channel
        coid = ConnectAttach (node_id, pid, chid, 0, 0);
        if (coid == -1)
            fprintf (stderr, "Client: ConnectAttach failed\n");
    } while (coid == -1); // infinite retry only for this example
    // send message
    if (-1 == MsgSend(coid, s_msg, strlen(s_msg)+1, r_msg, sizeof(r_msg))) {
        fprintf (stderr, "Error during MsgSend\n");
        exit (EXIT_FAILURE);
    }
    printf("Client: Reply from Server: %s \n", r_msg);
    return NULL;
}
```



## Beispiel (Fortsetzung)

```
void * server (void * not_used) // Server
{
    int  rcvid;                // receive id
    char msg[256];             // message buffer

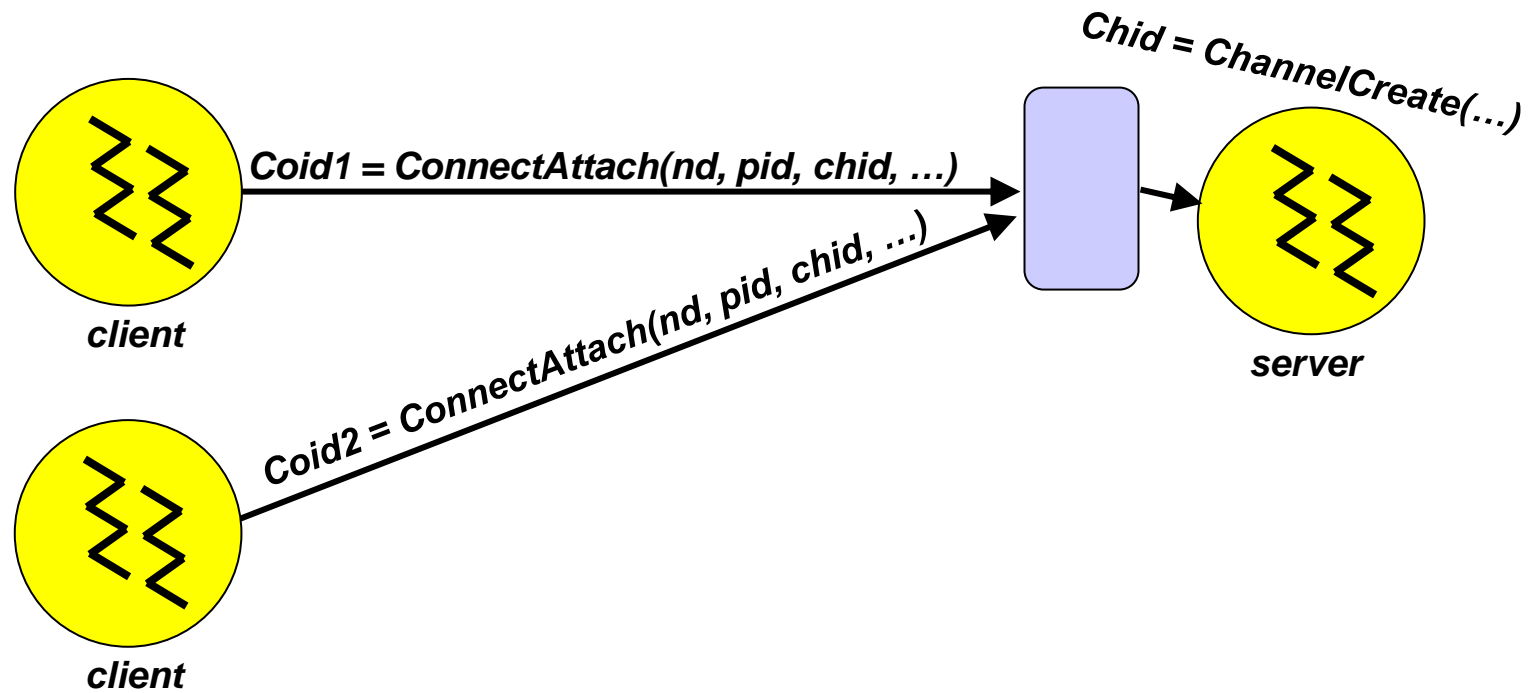
    chid = ChannelCreate(0); // chid = ChannelId

    while (1) { // Server usually works in infinite loop
        // Waiting for a message
        rcvid = MsgReceive (chid, msg, sizeof (msg), NULL);
        // handle the message
        printf("Server: got the message: %s \n", msg);
        strcpy(msg, "Got the message");
        // send reply
        MsgReply(rcvid, EOK, msg, sizeof(msg) );
    }
    // you'll never get here
    return NULL;
}

int main(int argc, char *argv[])
{
    terminal_open(0);
    printf("Create server and client thread\n");
    pthread_create(NULL, NULL, server, NULL);
    pthread_create(NULL, NULL, client, NULL);

    sleep (200);
    terminal_close();
    return (EXIT_SUCCESS);
}
```

## Auf der Kommunikationsstruktur zwischen Client & Server



## Auswahl von QNX Message Passing Routinen

```
#include <sys/neutrino.h>
```

```
int ChannelCreate( unsigned flags );
```

### Funktion:

- Diese Funktion erzeugt einen Channel (Kommunikationskanal). Der Channel gehört dem Prozess, dessen Thread ChannelCreate aufgerufen hat.  
Nur die Threads dieses Prozesses können mit MsgReceive auf diesen Channel (via chid) zugreifen.  
Beliebige Threads (Threads des Prozesses, Threads anderer Prozesse, Threads von Prozessen, die auf anderen Nodes/Rechnern ablaufen) können sich über ConnectAttach an den Channel anschließen.

### Parameter:

- flags: Flags, die Eigenschaften des Channels definieren.  
flags = 0 : default Werte (mehr Details : QNX Dokumentation)
- return value : -1 : failure (Details via errno)  
**ChannelId** : Eindeutige Bezeichnung des Channels innerhalb des Prozesses (benötigt MsgReceive)

## Auswahl von QNX Message Passing Routinen

```
#include <sys/neutrino.h>
```

```
int ConnectAttach( uint32_t nd, pid_t pid,  
                   int chid, unsigned index, int flags );
```

### Funktion:

- Diese Funktion stellt die Verbindung zu dem Channel chid des Prozesses pid, der auf dem Node /Rechner nd läuft, her.

### Parameter:

- nd: Beschreibung des Nodes / Rechners  
no = 0 : Local Node
- pid: Beschreibung des Prozesses, dem der Channel gehört.  
pid = 0 : Der Channel gehört dem selben Prozess, dessen Thread die Funktion ConnectAttach aufgerufen hat.

## Auswahl von QNX Message Passing Routinen

Fortsetzung der Beschreibung von ConnetAttach

### Parameter:

- `chid`: Bezeichnung des Channels.
- `index`: QNX behandelt ConnectionIds und File Descriptors gleich, es sind Objekte des selben Typs. Die von ConnectAttach erzeugte ConnectionId ist größer als `index`. Über diese Weg werden File Descriptors und ConnectionIds eines Prozesses in zwei Gruppen getrennt. Der aktuelle Parameter von `index` sollte `_NTO_SIDE_CHANNEL` sein, da kein File Descriptor  $\geq$  `_NTO_SIDE_CHANNEL` sein kann.  
In Randfällen ist es entscheidend, dass `coid`  $\geq$  `_NTO_SIDE_CHANNEL` ist: Terminiert ein Prozess ohne vorheriges ConnectDetach → OS schickt synchrone Message, die dem close auf einem File Descriptor entspricht
- `flags`: Flags, die Eigenschaften der Verbindung definieren.  
`flags = 0` : default Werte (mehr Details : QNX Dokumentation)
- return value : -1 : failure (Details via errno)  
**ConnectionId** : Eindeutige Bezeichnung der Verbindung innerhalb des Prozesses. MsgSend benötigt die ConnectionId.

## Auswahl von QNX Message Passing Routinen

```
#include <sys/neutrino.h>
```

```
int ChannelDestroy (int chid);
```

### Funktion:

- Diese Funktion löscht einen Channel.  
Der Kernel setzt alle Threads, die aufgrund eines MsgSend an diesen Channel blockiert sind, in den Zustand ready.

### Parameter:

- `chid`: Der Channel, der gelöscht werden soll.
- return value : -1 : failure (Details via errno)  
sonst : Funktion wurde fehlerfrei ausgeführt.

## Auswahl von QNX Message Passing Routinen

```
#include <sys/neutrino.h>
```

```
int ConnectDetach ( int coid);
```

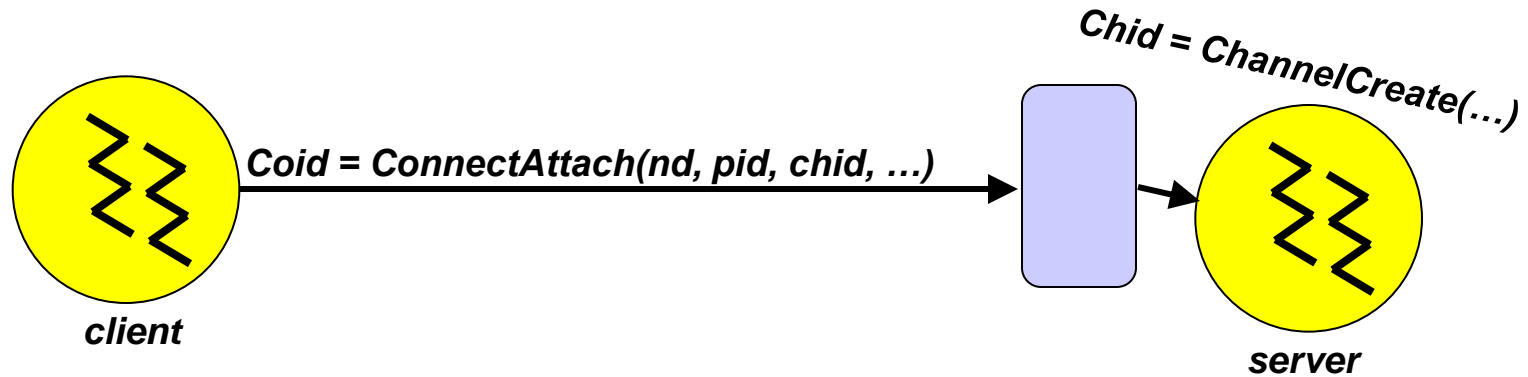
### Funktion:

- Diese Funktion löscht eine Verbindung zu einem Channel. Der Kernel setzt alle Threads des Clients, die aufgrund eines MsgSend bezüglich dieser Verbindung blockiert sind, in den Zustand ready.

### Parameter:

- `coid`: Der ConnectionId der Verbindung, die gelöscht werden soll.
- return value : -1 : failure (Details via errno)  
sonst : Funktion wurde fehlerfrei ausgeführt.

## Wie bekommt ein Client die Daten des Channels eines Servers?



**Situation:** Der Client braucht den Node (nd), die ProcessId (pid) des Servers und die ChannelId des Channels, zu dem die Verbindung aufgebaut werden soll.

**Fall 1:** Client und Server sind Threads des selben Prozesses

- Lösung: nd = 0, pid = 0, chid : globale Variable (s. Beispiel)

**Fall 2:** Der Client ist ein Sohn Prozess des Servers

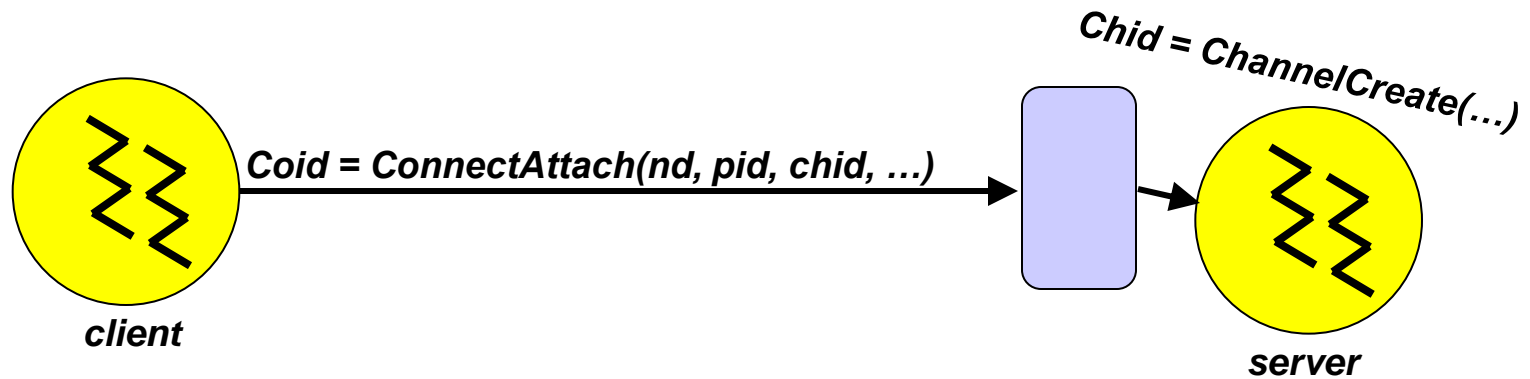
- Lösung: nd = 0, pid und chid werden als Parameter übergeben

**Fall 3:** Client und Server haben keine Relation („perfect strangers“)

- s. nächste Folie



## Wie bekommt ein Client die Daten des Channels eines Servers?



### Fall 3: Client und Server haben keine Relation („perfect strangers“) (Fortsetzung)

- Ein File speichert nd, pid und chid. Der Client kennt den Namen des Files.
  - UNIX Stil
  - Problem: Der Server ist terminiert und das File existiert noch.
- Nd, pid und chid werden im Shared Memory abgelegt.
  - Aufgrund des Shared Memory nur für wenige Situationen geeignet.
- QNX: Der Server wird Resource Manager für einen Teil des File Systems.

## Auswahl von QNX Message Passing Routinen

```
#include <sys/neutrino.h>
```

```
int MsgSend ( int coid, const void* smsg, int sbytes,  
              void* rmsg, int rbytes );
```

### Funktion:

- Diese Funktion schickt eine Message an den Channel, der über die **ConnectionId** **coid** angebunden ist. **MsgSend** blockiert die Ausführung, bis die Antwort auf die Message eingetroffen ist.

SEND-blocked: Die Message ist noch nicht in Bearbeitung. Entweder hat der Server **MsgReceive** noch nicht aufgerufen oder die Message wartet noch in der Schlange.

REPLY-blocked: Der Server hat die Message mit **MsgReceive** eingelesen, aber noch nicht beantwortet.

Die Anzahl der Bytes, die übertragen werden, ist das Minimum aus **sbytes** und der Größe des Puffers, den **MsgReceive** für die eingehende Message bereit hält.

Analog wird die Länge der Antwort bestimmt (**rmsg**).

## Auswahl von QNX Message Passing Routinen

Fortsetzung der Beschreibung von `MsgSend`

### Parameter:

- `coid` : ConnectionId der Verbindung zum Channel.
- `smsg` : Die Message, die übermittelt werden soll.
- `sbytes` : Die Länge von `smsg`.
- `rmsg` : Puffer Speicher für die Antwort des Servers
- `rbytes` : Größe des Puffers `rmsg`.
- `return value` : -1 : failure (Details via `errno`)  
sonst : Der in `MsgReply` übergebene Rückgabewert

## Auswahl von QNX Message Passing Routinen

```
#include <sys/neutrino.h>
```

```
int MsgReceive( int chid, void * msg, int bytes,  
                struct _msg_info * info );
```

### Funktion:

- Diese Funktion liest eine Message aus dem Channel mit der **ChannelID** *chid*. Warten mehrere Messages in dem Channel, so liest **MsgReceive** die Message mit der höchsten Priorität ein. Ansonsten wird die Ausführung blockiert, bis eine Message eintrifft (Blocking state: `STATE_RECEIVE`).

Die Anzahl der Bytes, die eingelesen werden, ist das Minimum aus der Länge der Message und der Größe des Puffers (*bytes*), den **MsgReceive** bereit hält.

### Parameter:

- *Chid*: ChannelID des Channels, aus dem die Message gelesen wird.

## Auswahl von QNX Message Passing Routinen

Fortsetzung der Beschreibung von MsgReceive

### Parameter:

- `msg`: Puffer Speicher für die Message, die gelesen wird.
- `bytes`: Die Länge von `msg`.
- `info`: Informationen über die Message  
`info = NULL`: Informationen werden nicht benötigt
- `return value`:
  - 1 : failure (Details via `errno`)
  - > 0 : Eine Message wurde eingelesen. Der return value ist die ReceiveId (`rcvid`). Über die `rcvid` ordnet `MsgReply` die Antwort der entsprechenden Message zu.
  - 0 : Eine Pulse Message wurde eingelesen.

## Auswahl von QNX Message Passing Routinen

```
#include <sys/neutrino.h>
```

```
int MsgReply(int rcvid, int status, const void* msg, int size)
```

### Funktion:

- Diese Funktion schickt die Antwort auf die Message mit der Receiveld `rcvid`. Die Antwort wird sofort übertragen und der Thread, der die Message geschickt hat, geht in den Zustand READY über.
- Die Anzahl der Bytes, die geschickt werden, ist das Minimum aus `size` und der Größe des Puffers, den der entsprechende `MsgSend` Aufruf für die Antwort bereit hält.

### Parameter:

- `rcvid`: Die Receiveld zur der Message, die beantwortet wird.
- `status`: Der `MsgSend` Aufruf, dessen Message beantwortet wird, gibt `status` als return value zurück.

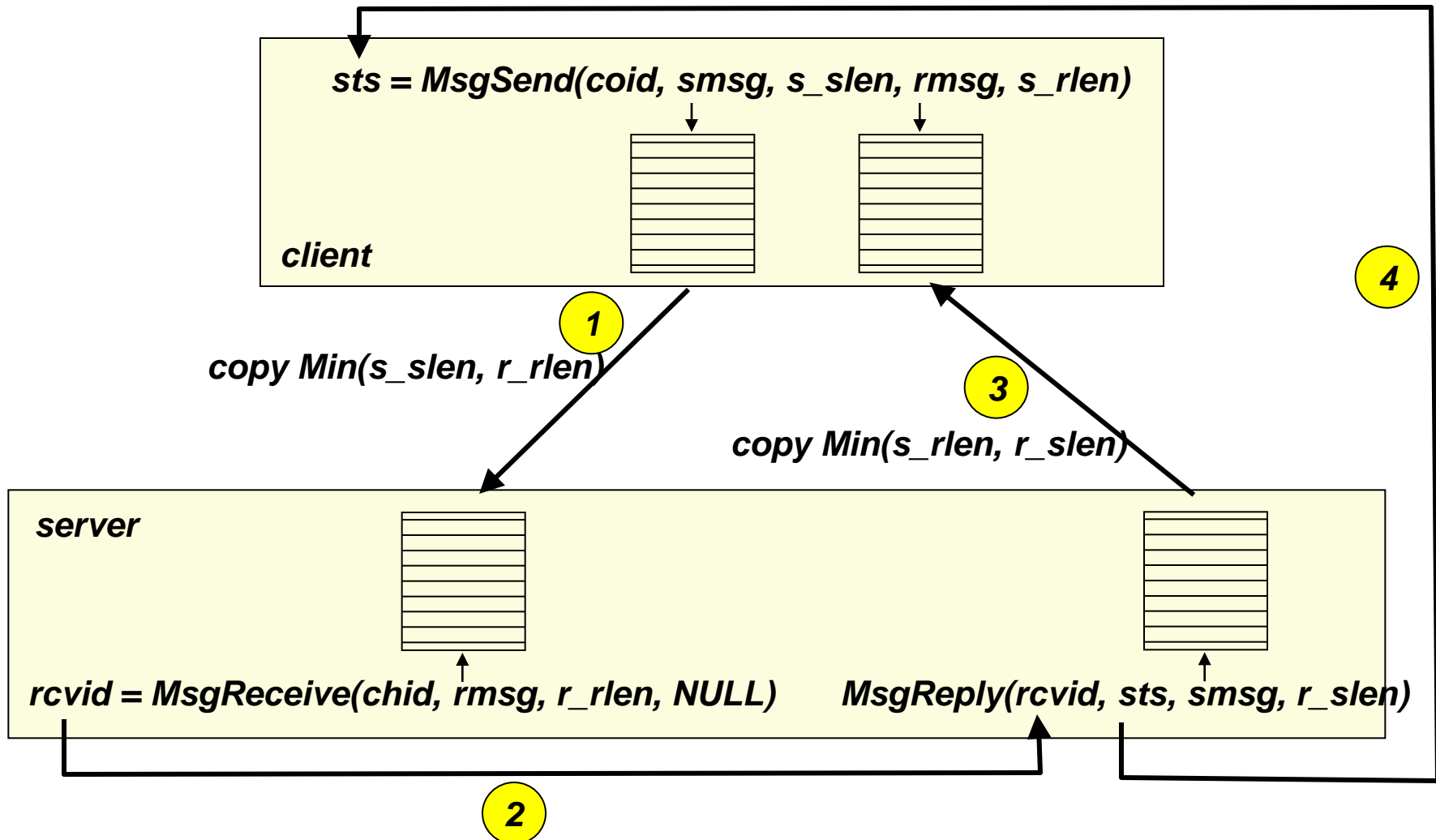
## Auswahl von QNX Message Passing Routinen

Fortsetzung der Beschreibung von MsgReply

### Parameter:

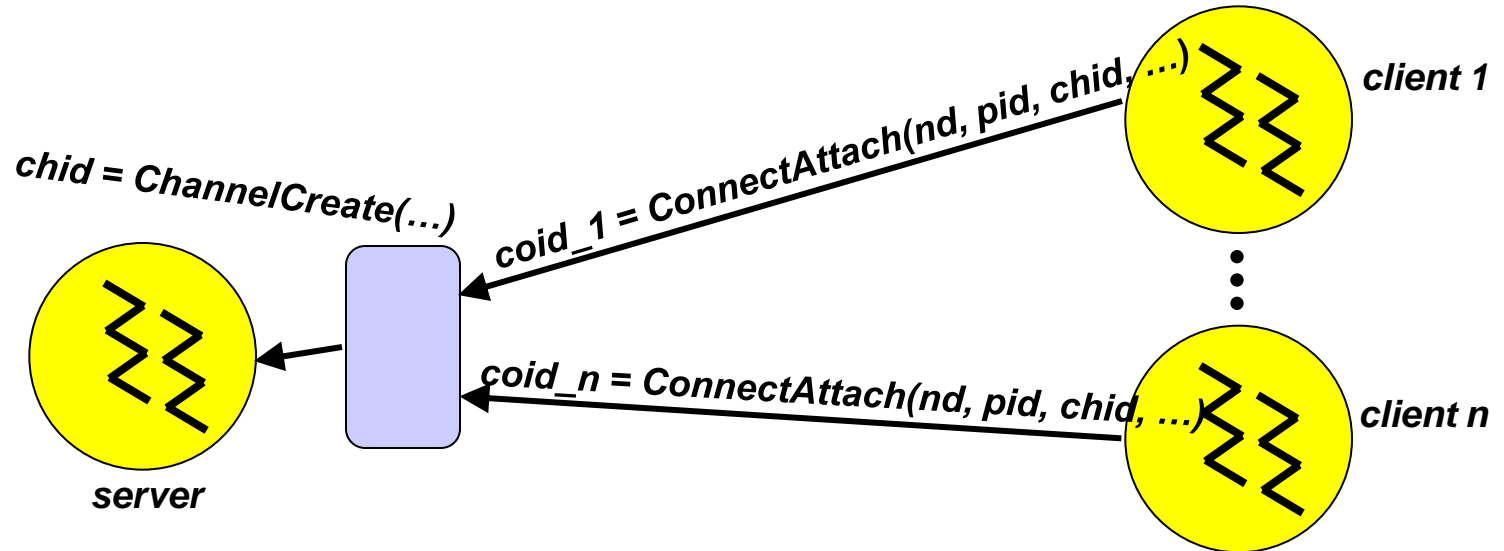
- `msg`: Die Antwort die geschickt wird.
- `size`: Die Länge von `msg`.
- `return value` : -1 : failure (Details via `errno`)  
sonst : Funktion wurde fehlerfrei ausgeführt.

## Zusammenspiel von MsgSend, MsgReceive und MsgReply





## ChannelCreate & MsgReceive ↔ ConnectAttach & MsgSend



- Ein Thread des Servers erzeugt einen Channel. Alle Threads des Servers können mit MsgReceive auf den Channel zugreifen (MsgReceive verwendet die ChannelId - chid).
- Ein Thread eines Clients stellt über ConnectAttach eine Verbindung zu einem Channel her. Alle Threads des Clients können über MsgSend eine Message über den Channel schicken (MsgSend verwendet eine ConnectionId – coid). MsgReceive ist nicht möglich.
- Client-Server Modell: Mehrere Clients können eine Verbindung zum einen Channel zeitgleich unterhalten.
- Anmerkung: Ein Thread des Servers kann über ConnectAttach auch eine Verbindung zu einem Channel dieses Servers aufbauen.

## Anmerkungen

- **Ein Channel ist kein Puffer** (bezüglich der synchronen Kommunikation). Die Daten werden direkt aus dem Speicher des Client in den Speicher des Server kopiert (und umgekehrt).
- **Ein Channel ist einem Prozess zugeordnet.** Nur die Threads dieses Prozesses können Messages, die über diesen Channel eintreffen, annehmen (MsgReceive). Mehrere Threads des Prozesses können den Channel bedienen.
- **Messages, die über einen Kanal geschickt werden, dürfen unterschiedliche Längen haben.** Dies gibt ebenso für die Antworten auf Messages.
- **Beliebige Threads beliebiger Prozesses können Messages an einen Channel schicken** (MsgSend), nachdem über ConnectAttach die Verbindung hergestellt wurde.  
Hat ein Thread die Verbindung zu einem Channel hergestellt, können alle Threads seines Prozesses die Verbindung nutzen.

## Anmerkungen (Fortsetzung)

- **MsgReceive** nimmt eine Message aus einem Channel entgegen. **Es können beliebig viele Messages mit MsgReceive angenommen werden, bevor die erste Message mit MsgReply beantwortet wird.** Die Antworten können „out of order“ geschickt werden.
- **Message Passing zwischen Prozessen, die auf unterschiedlichen Rechnern (Nodes) laufen, wird mit den selben Funktionen realisiert.** Ein Resource Manager stellt die Verbindung über das Netzwerk her.  
Da „Message Passing zwischen mehreren Nodes“ nicht über einen einzigen Kernel abgewickelt wird, arbeiten „Message passing auf einem Node“ und „Message Passing zwischen mehreren Nodes“ an einigen Stellen unterschiedlich (s. Dokumentation).
- **Mit der Funktion `MsgError(rcvid, error)` liefert ein Server eine Fehlernummer an einen Client zurück.** `MsgError` setzt `errno` und `MsgSend` liefert den return value -1.  
Der Aufruf `MsgReply(rcvid, -1, NULL, 0)` kann nicht verwendet werden, da `errno` nicht gesetzt wird.

## Anmerkungen (Fortsetzung)

- Mehrere Threads können eine Message an den selben Kanal schicken.  
**Die Message des Threads mit der höchsten Priorität wird zuerst an den Server weitergeleitet** – wird von `MsgReceive` angenommen.  
Haben zwei Threads die selbe Priorität, wird der Thread mit der längsten Wartezeit zuerst bedient.
- **Priority inversion:** Ein hoch priorisierter Thread wird durch einen niedrig priorisierten Thread indirekt blockiert (s. Kapitel 2)  
Lösung **Priority inheritance:** Während ein Server die Message eines Client bearbeitet, erhält er die Priorität des Client, dessen Message er bearbeitet, bzw. die höchste Priorität aller blockiert Client Threads des Servers.

Dies wird über Flags des Channels ein/ausschalten.

Achtung: Damit werden nicht alle Scheduling – Prioritäten Probleme gelöst, es ist nur ein Werkzeug, das der Entwickler der Anwendung „richtig“ einsetzen muss.

Priority inheritance ist an den Channel gebunden und standardmäßig aktiv.  
Wird bei `ChannelCreate` das Flag `_NTO_CHF_FIXED_PRIORITY` übergeben, dann wird priority inheritance ausgeschaltet.

## MsgRead & MsgWrite

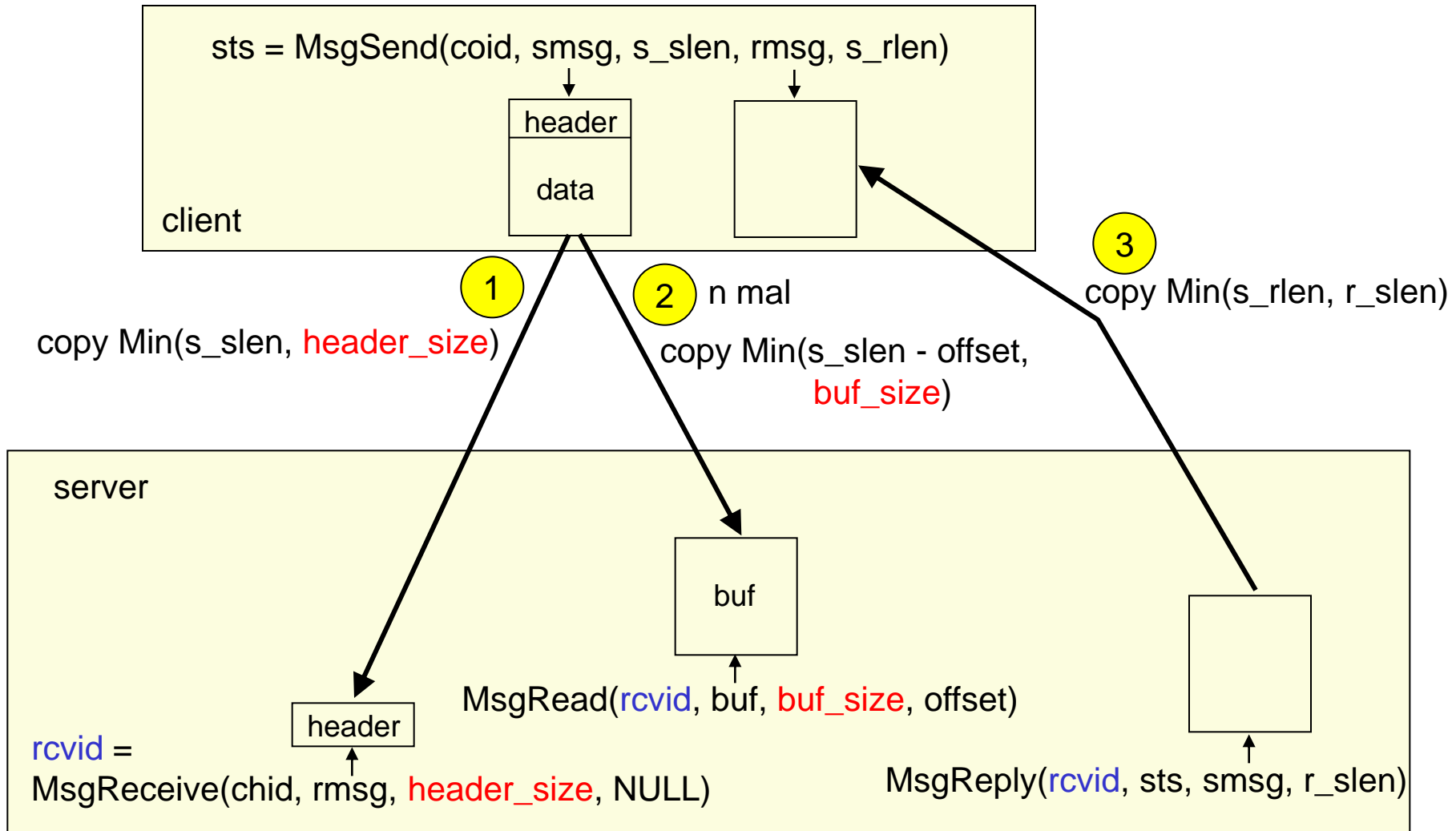
### Motivation:

- QNX: Nahezu die gesamte Kommunikation wird über (synchrone) Messages abgewickelt.
- Insbesondere wird der Zugriff auf das Filesystem über synchrone Messages abgewickelt.  
Das Filesystem ist ein Server, der seine Aufgaben über Messages empfängt.
- Der Funktionsaufruf `write ( file_des, buf, buf_size)` wird auf einen `MsgSend` Aufruf abgebildet.  
Die Puffer Größe ist nicht eingeschränkt ( 1 byte, 1 Mbyte, ...).
- **Problem:** Die Puffergröße der `MsgSend`, `MsgReceive` und `MsgReply` Aufrufe muss variabel sein.  
Im obigen Beispiel kennt der `MsgReceive` Aufruf des Servers Filesystem die notwendige Puffergröße erst, nachdem die Daten des `MsgSend` Befehl angekommen sind.
- **Lösung:** `MsgRead` & `MsgWrite`: Ein Puffer wird in mehreren Schritten ausgelesen.

## Strukturierte Messages

- Eine Message wird in zwei Teile zerlegt:
  1. Header: Der Header enthält Informationen über die Message (z.B. die Länge)
  2. Buffer: Der Buffer enthält die „Nutzdaten“
- Über die Funktion `MsgRead` kann der Server auf Teile des Message Puffer, der im `MsgSend` Befehl übergeben wird, zugreifen.  
Da der Client Thread bis zum `MsgReply` blockiert ist, ist diese Vorgehensweise zulässig (solange kein anderer Thread die Daten im Puffer modifiziert).

## Zusammenspiel



## Auswahl von QNX Message Passing Routinen

```
#include <sys/neutrino.h>
```

```
int MsgRead( int rcvid, void* msg, int bytes, int offset );
```

### Funktion:

- Diese Funktion liest Daten aus dem zur `rcvid` gehörigen Message Puffer. Der entsprechenden Client Thread muss im Zustand `REPLY blocked` sein, d.h. der `MsgReply` Aufruf für `rcvid` hat noch nicht stattgefunden.

### Parameter:

- `rcvid` : Receiveld der Message, auf die zugegriffen werden soll.
- `msg` : Puffer für die Daten, die gelesen werden sollen
- `bytes` : Die Länge von `msg`.
- `offset` : Ab `offset` werden die Daten aus der Message gelesen.
- `return value` : -1 : failure (Details via `errno`)  
sonst : Anzahl der gelesenen Bytes



## Beispiel

```
#include <cpt_terminal.h>           // BEISPIEL OHNE FEHLERBEHANDLUNG
#include <pthread.h>
#include <errno.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/neutrino.h>
#include <malloc.h>

// Msg Typen
#define STR_MSG 0
#define DATA_MSG 1

typedef struct {
    int type;
    int size; // size of data block
} header_t;

int chid = -1; // channel id, globale Variable, Synchronisation notw.
```

## Beispiel (Fortsetzung)

```
void *client (void * unused)
{
    int    node_id = 0 ;    // 0 : my node
    pid_t  pid= 0 ;        // 0 : my process
    int    coid;           // connection id
    int    status;

    char   *msg = "Dies ist ein sehr grosser Datenbereich ... ..";
    int    msg_size = strlen(msg) + 1;

    char   *new_buf; // Ptr. To buffer for header and data of msg
    header_t *header_ptr;

    char   r_msg[512];

    // connect to channel
    do {
        coid = ConnectAttach (node_id, pid, chid, 0, 0);
    } while (coid == -1); // infinite retry only for this example
```

## Beispiel (Fortsetzung)

```
// continue client code

// build message
new_buf = malloc(sizeof(header_t) + msg_size);
header_ptr = (header_t *) new_buf;
header_ptr->type = STR_MSG;
header_ptr->size = msg_size;
memcpy(new_buf + sizeof(header_t), msg, msg_size);

status = MsgSend(coid, new_buf, sizeof(header_t) + msg_size,
                 r_msg, sizeof(r_msg) );

free(new_buf);
return NULL;
}
```

Bei großen Datenmengen  
muss man diesen Kopier-  
befehl vermeiden (s.u.)

## Beispiel (Fortsetzung)

```
void * server (void * not_used) // Server
{
    int  rcvid;// receive id
    char msg[256];// message buffer


    header_t header;

    chid = ChannelCreate (0);
    while (1) {
        // Waiting for a message
        rcvid = MsgReceive (chid, &header, sizeof (header_t), NULL);

        // handle the message
        switch (header.type) {
            case DATA_MSG:
                fprintf(stderr, "DATA_MSG not supported\n");
                exit(EXIT_FAILURE);
```

## Beispiel (Fortsetzung)

Alternative: Kleine Puffer, die mit mehreren MsgRead Befehlen gefüllt werden.

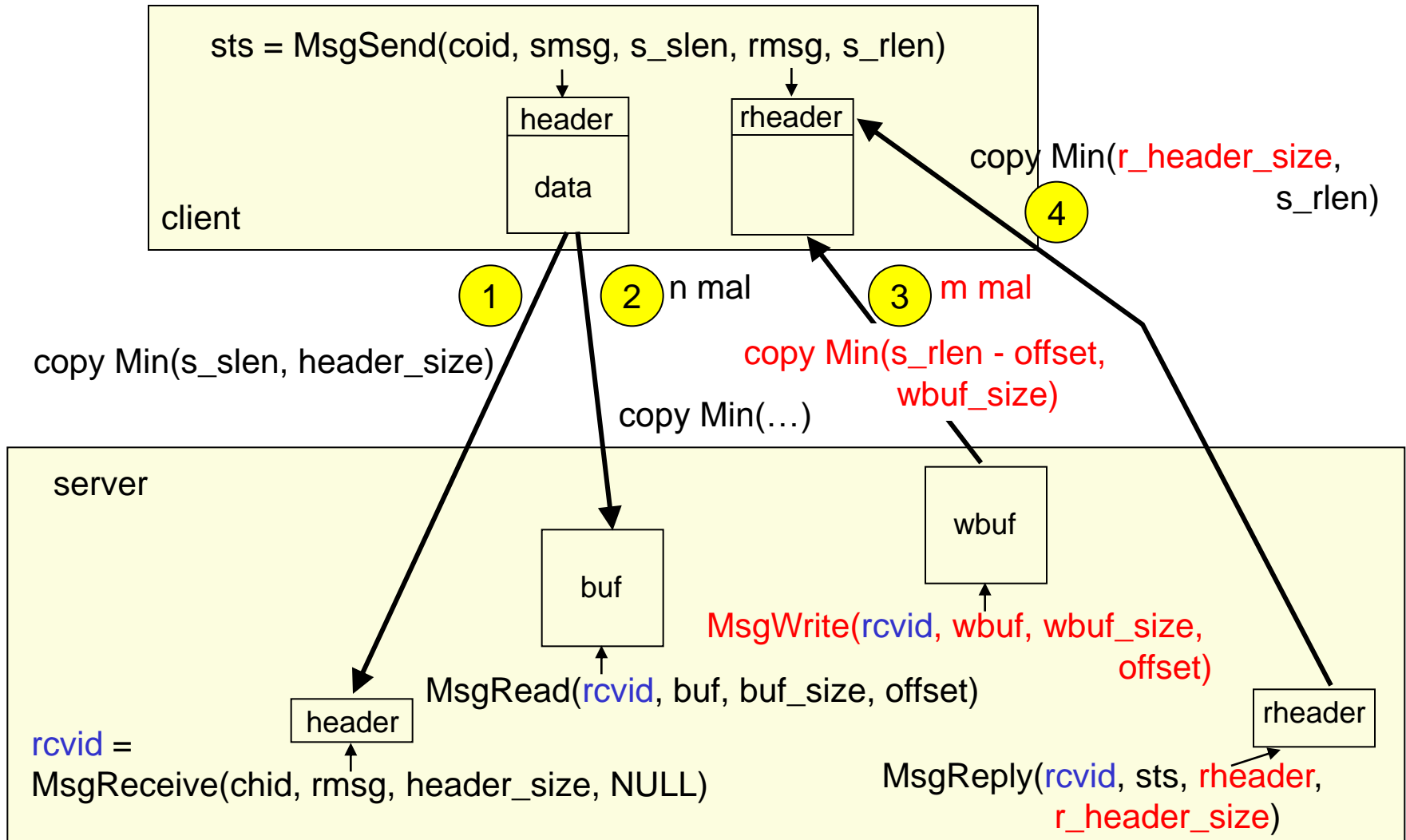


```
case STR_MSG: {
    char * buf;
    buf = malloc(header.size);
    MsgRead(rcvid, buf, header.size, sizeof(header_t));
    printf("Server: got the STR_MSG: %s \n", buf);
    strcpy(msg, "Got the message");
    // send reply
    MsgReply(rcvid, EOK, msg, sizeof(msg) );
    free(buf);
}
break;
default:
    fprintf(stderr, "Wrong message type.\n");
    exit (EXIT_FAILURE);
}
} // infinite while loop
return NULL;
}
```

## Beispiel (Fortsetzung)

```
int main(int argc, char *argv[]) {  
  
    terminal_open(0);  
  
    // start two threads within the same process  
    // message passage works between threads of different processes, too  
  
    printf("create server and client thread\n");  
  
    pthread_create(NULL, NULL, server, NULL);  
    pthread_create(NULL, NULL, client, NULL);  
  
    sleep (30);  
    terminal_close();  
    return (EXIT_SUCCESS);  
}
```

## Zusammenspiel



## Auswahl von QNX Message Passing Routinen

```
#include <sys/neutrino.h>
```

```
int MsgWrite( int rcvid, void* msg, int size, int offset );
```

### Funktion:

- Diese Funktion **schreibt** Daten in den zu `rcvid` gehörigen Message Reply Puffer. Dieser Puffer wurde im `MsgSend` Aufruf übergeben. Der entsprechende Client Thread muss im Zustand `REPLY blocked` sein, d.h. der `MsgReply` Aufruf für `rcvid` hat noch nicht stattgefunden.

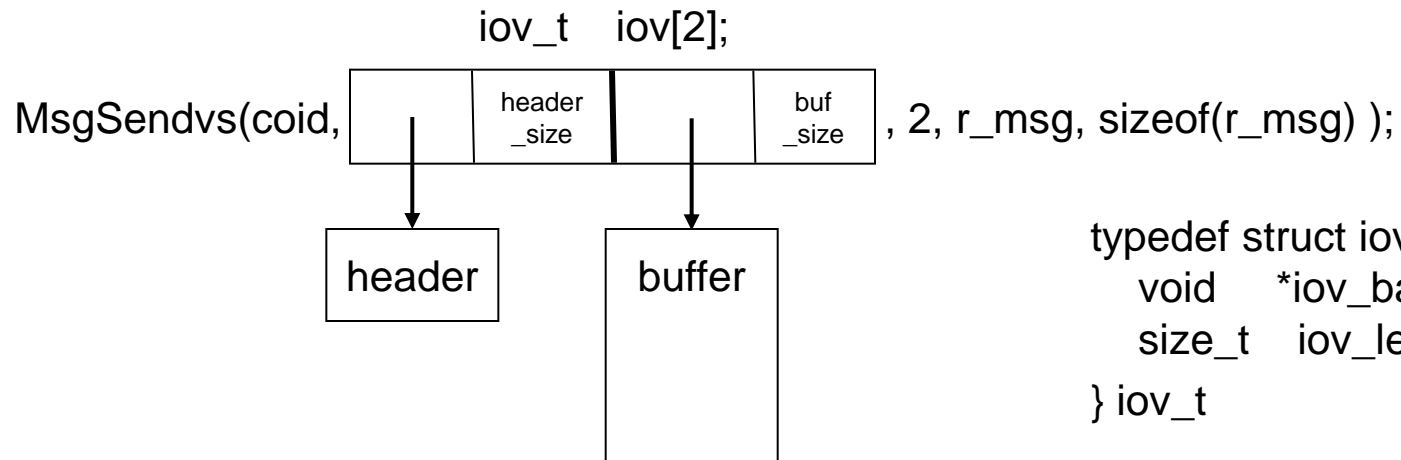
### Parameter:

- `rcvid` : Receiveld der Message.
- `msg` : Puffer für die Daten, die geschrieben werden sollen.
- `size` : Anzahl der Bytes, die geschrieben werden sollen.
- `offset` : Ab `offset` werden die Daten in den Reply Puffer geschrieben.
- return value : -1 : failure (Details via `errno`)  
sonst : Anzahl der geschriebenen Bytes



## Multipart Messages

- Bemerkung Beispiel Folie 172: Der Client kopiert Daten und MsgHeader in einem Puffer zusammen.
- Könnte MsgSend eine Liste von Puffern verarbeiten, entfällt dieser Kopiervorgang.
- Das **Multipart Message** Konzept realisiert diese Anforderung über IOV Input / Output Vectors.
- Dies gilt analog für den Speicherbereich, den MsgSend für die Antwort bereitstellt.



## Beispiel (mit IOV)

```
void *client (void * unused)
{
    // node_id declaration etc
    ...
    char  *msg = "Dies ist ein sehr grosser Datenbereich ... ..";
    int    msg_size = strlen(msg) + 1;

    header_t header;
    iov_t    iov[2]; // vector for multi part message
    ...

    // build message
    // entfällt new_buf = malloc(sizeof(header_t) + msg_size);
    header.type = STR_MSG; header.size = msg_size;
    // entfällt memcpy(new_buf + sizeof(header_t), msg, msg_size);
    SETIOV(iov+0, &header, sizeof(header));
    SETIOV(iov+1, msg, msg_size);

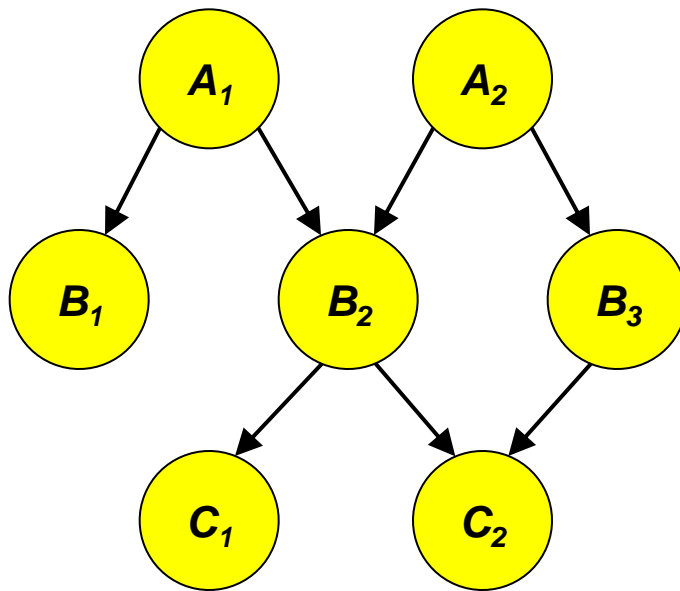
    status = MsgSendvs(coid, iov, 2, r_msg, sizeof(r_msg) );
    return NULL;
}
```

## Multipart Messages

### Anmerkungen:

- Über entsprechende Funktionsaufrufe unterstützt QNX IOVs für Send, Receive, Write, Read und Reply → QNX Dokumentation
- Der Kernel bildet IOVs und lineare Puffer aufeinander ab. Somit kann zum Beispiel ein Send mit IOVs arbeiten und die zugehörigen Receive, Write, Read und Reply Aufrufe des Kernel lineare Puffer verwenden.

## Send Hierarchie

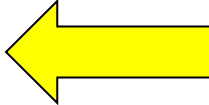
**SENDS****REPLYs**

**Send Hierarchie: Wird nicht durch das OS vorgeschrieben, aber ist die Grundlagen für ein „stabiles“ Design.**

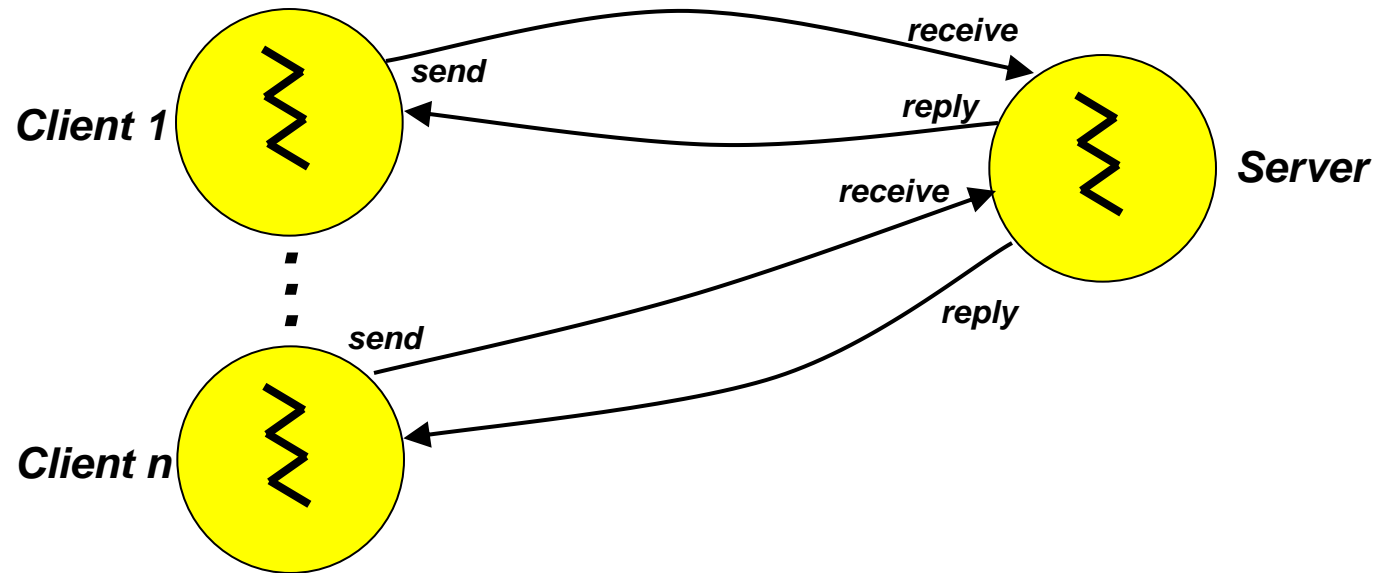
- Da Send und Reply blockierend sind, besteht eine Deadlock Gefahr, wenn Prozess A als Server für Prozess B dient und umgekehrt.
- Die Send Hierarchie ist wie folgt „definiert“: Schickt ein Thread von Prozess A eine Message an einen Thread von Prozess B, dann darf eine Thread von Prozess B niemals eine Botschaft an einen Thread von Prozess A schicken. Dies setzt sich rekursiv über die Threads der Prozesse fort, an die B eine Message schickt.
- Pulses (non blocking Messages) : Kommunikation „in Gegenrichtung“ zum Graph.

## Kapitel 3 : Inter Process Communication (IPC)

### Gliederung

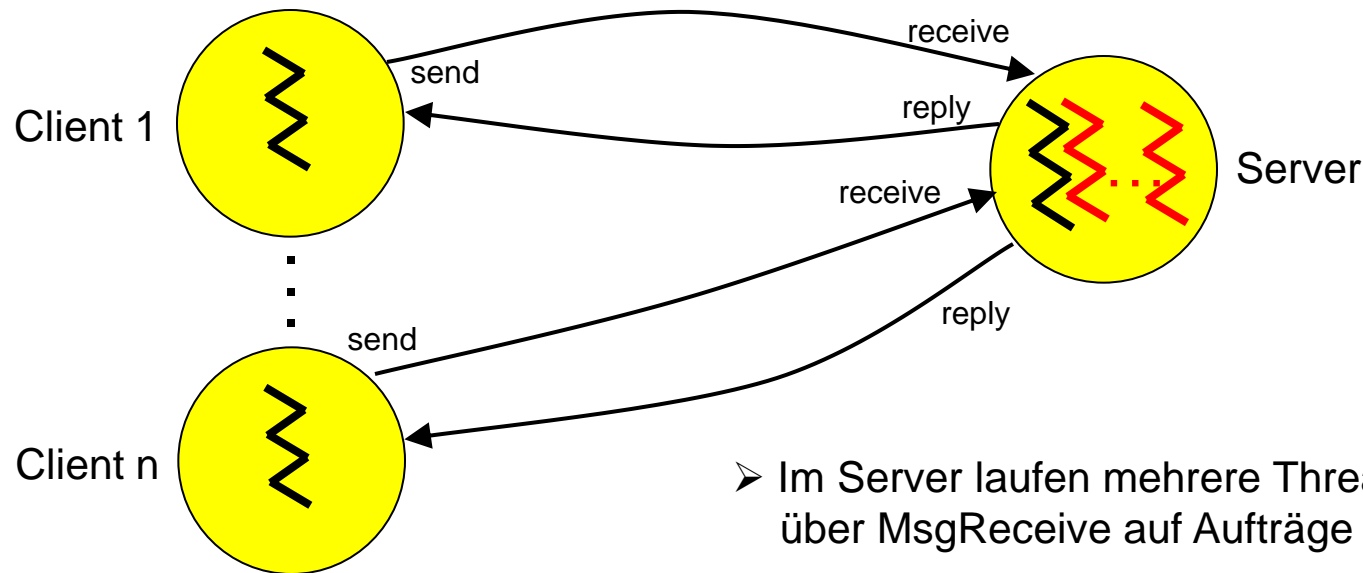
- Grundbegriffe der Inter Process Communication (IPC)
- Synchrones Message Passing: die Grundlage des QNX Realtime OS
- Client / Server Konzepte 
- Asynchrone Kommunikation: QNX Pulses
- Zusammenfassung

## Client / Server Konzept



- Ein Client ist ein Auftraggeber.
- Der Server ist ein Auftragnehmer, der Aufträge mehrerer Clients bearbeiten kann.

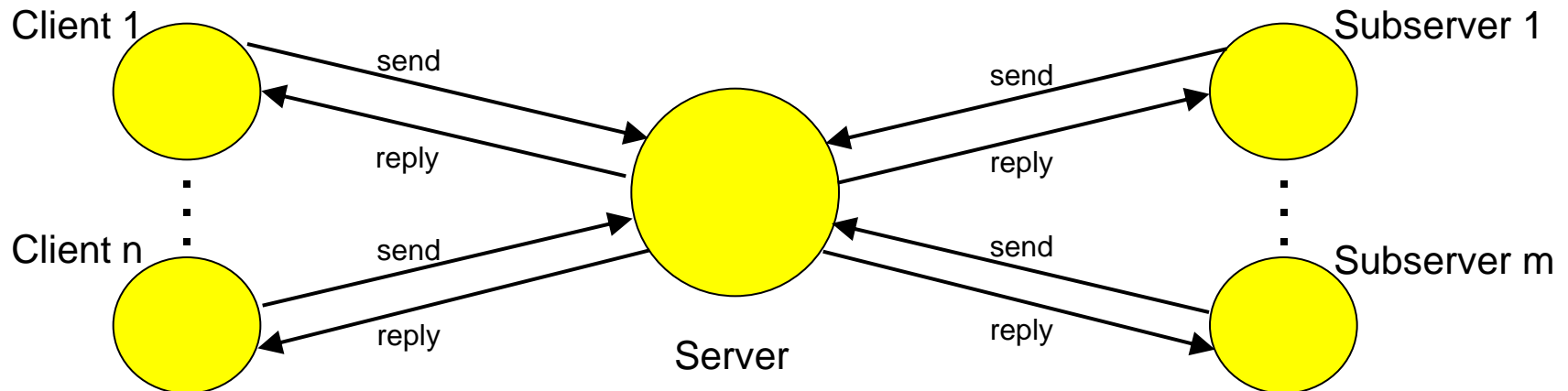
## Server with Multiple Threads



➤ Im Server laufen mehrere Threads, die über `MsgReceive` auf Aufträge warten.

- Mehrere Aufträge werden parallel bearbeitet, wobei das OS und nicht der Server das Scheduling durchführt.
- SMP System: Parallele Ausführung der Threads, da sie auf unterschiedlichen CPUen laufen (aber auf dem selben Rechner, gleiche Sicht auf den Speicher).
- Thread Pools ermöglichen, dass die Anzahl der Threads im Server bedarfsorientiert gesteuert wird.

## Server / Subserver Model



- Die Clients schicken Aufträge an einen Server.
- Der Server leitet die Aufträge an Subserver weiter.
- Der Kommunikation zwischen Server und Subserver ist **reply driven**, d.h. ein Subserver meldet sich beim Server über MsgSend an und wartet auf einen Auftrag, den der Server mit MsgReply an den Subserver weiterleitet.
- Da jeder Subserver ein eigenständiger Prozess ist, können Server und Subserver auf unterschiedlichen Rechnern eines Netzwerkes laufen.



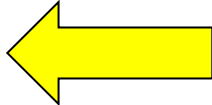
## Client / Server Konzepte

### Anmerkungen:

- Die Konzepte „Server with Multiple Threads“ und „Server / Subserver“ können natürlich zusammen angewendet werden. Dabei wird wie folgt verfahren:
  - Aufgaben, die gut über das Netzwerk verteilbar sind, werden gemäß dem „Server / Subserver“ Ansatz verteilt.
  - Aufgaben, die besser auf einem SMP Knoten gemeinsam ablaufen, werden über den Ansatz „Server with Multiple Threads“ verteilt.
- Der Subserver darf den Server nicht blockieren – ansonsten kann der Server keine weiteren Aufträge bearbeiten. Das garantiert die reply driven Kommunikation zwischen Server und Subservern.  
Würde der Server eine Message an einen Subserver schicken, ist er blockiert, bis diese beantwortet ist.
- Ein Client kann nicht unterscheiden, ob ein Standard Server, ein Server mit mehreren Threads oder ein Server / Subserver Modell implementiert wurde. Die Modelle „Server mit mehreren Threads“ und „Server / Subserver“ liefern eine bessere Performance und Auslastung auf einem SMP System bzw. einem Computer Netzwerk.

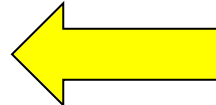
## Kapitel 3 : Inter Process Communication (IPC)

### Gliederung

- Grundbegriffe der Inter Process Communication (IPC)
- Synchrones Message Passing: die Grundlage des QNX Realtime OS
- Client / Server Konzepte
- Asynchrone Kommunikation: QNX Pulses 
- Zusammenfassung

## Kapitel 3 : Inter Process Communication (IPC)

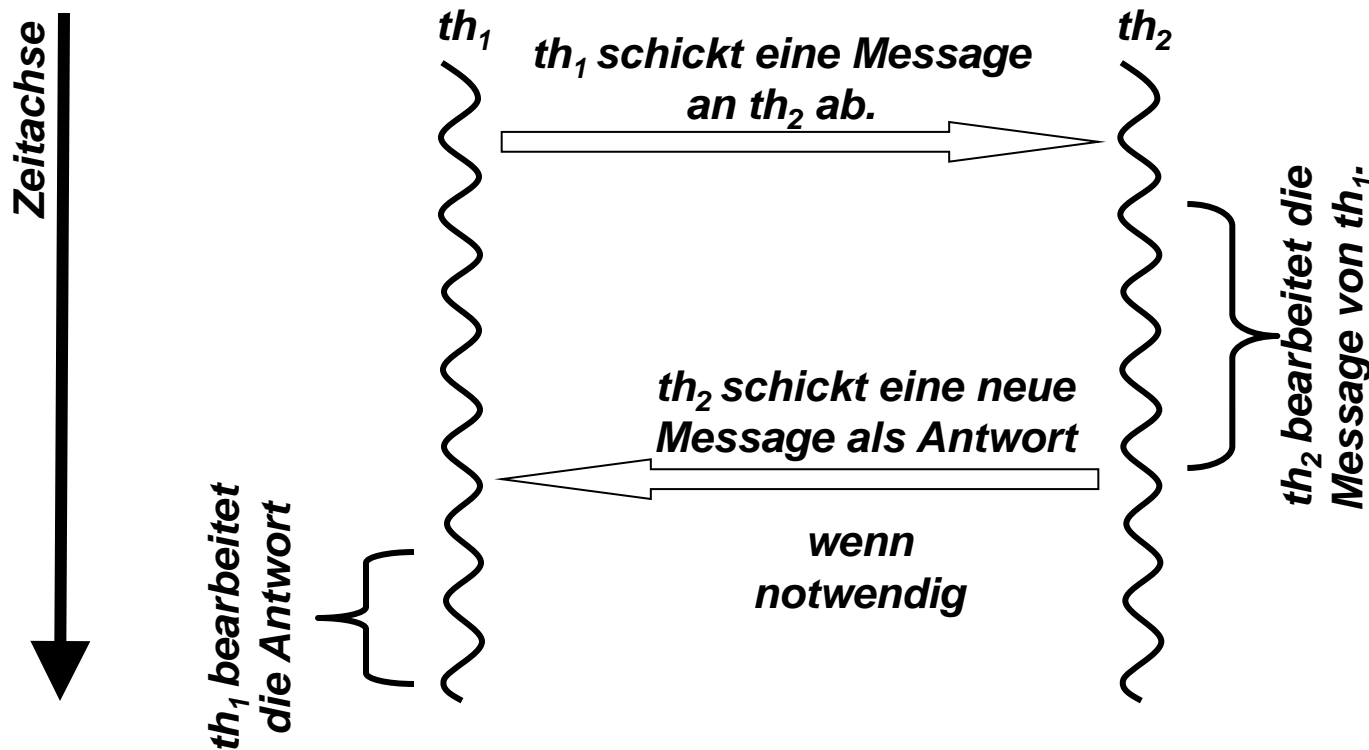
### Gliederung

- Grundbegriffe der Inter Process Communication (IPC)
- Synchrones Message Passing: die Grundlage des QNX Realtime OS
- Client / Server Konzepte
- Asynchrone Kommunikation: QNX Pulses 
- Zusammenfassung

## Asynchrone Kommunikation via Message Passing (Pulses)

Thread  $th_1$  schickt eine Message an Thread  $th_2$ . Während  $th_2$  die Message bearbeitet, rechnet  $th_1$  weiter –  $th_1$  wird nicht blockiert.

Bei Bedarf schickt  $th_2$  die Antwort in einer neuen Message (i.a. asynchron). Eine asynchrone Kommunikation fordert keine Antwort auf eine Message.



## Asynchrone Kommunikation in QNX: Pulses

- Ein Pulse Message ist eine asynchrone Message.
- Ein Pulse hat ein festes Datenpaket von 40 bit, die sich wie folgt aufteilen:
  - 8 bit Code - Ein „Beschreibung“ der Pulse Message
  - 32 bit Daten
- Eine Pulse Message wird wie jede andere Message empfangen (MsgReceive, MsgReceivePulse)
- Falls kein MsgReceive oder MsgReceivePulse auf eine Message wartet, wird die Pulse Message in der entsprechenden Queue gespeichert.
- Eine Pulse Message wird über spezielle Funktionen geschickt (MsgSendPulse und MsgDeliverEvent)

## Struktur eine Pulse Message

```
struct _pulse {  
    _uint16          type;           // 0 für pulses  
    _uint16          subtype;       // 0 für pulses  
    _uint8           code;          // code des Pulse  
    _uint8           zero[3];       // Platzhalter  
    union sigval     value;         // Daten zum code  
    _uint32          scoid;         // Kernel: Server Con Id  
  
}
```

```
union sigval {  
    int              sival_int;  
    void             sival_ptr;  
}
```

- Code 128 bis 255 (Negative Codes) sind für den Kernel reserviert.
- Freie Pulse Codes liegen im Bereich  
    \_PULSE\_CODE\_MINAVAIL bis \_PULSE\_CODE\_MAXAVAIL.

## Empfangen von Pulse Messages via MsgReceive

- Eine Pulse Message ist asynchron. Sie wird nicht mit MsgReply beantwortet.
- Die ReceiveId, die MsgReceive zu einer Pulse Message erzeugt, ist 0.
- Die ReceiveId 0 ist eindeutig für eine Pulse Message.

## Beispiel

```
rcvid = MsgReceive ( chid, msg, ...);  
if (rcvid == 0) { // it's a pulse  
    switch (msg.pulse.code) {  
        case MY_PULSE_TIME : // Ein Timer ist abgelaufen  
            ...  
            break;  
        case MY_PULSE_HWINT : // HW interrupt service routine hat  
                               // einen Pulse geschickt.  
            ...  
            break;  
        case _PULSE_CODE_UNBLOCK : // Kernel hat den Pulse  
                                    // geschickt. Client unblocked  
            ...  
            break;  
        ...  
    }  
} else if (rcvid < 0) { ... // handle error  
} else { // it's a regular sync. message  
    // determine the type of message  
    ...  
    // handle it  
    ...  
}
```



## Empfangen von Pulse Messages via MsgReceivePulse

- Pulse Messages treffen über einen Channel ein – wie jede Message.
- Im Channel können Pulse Messages und synchrone Messages liegen.
- Der Kernel Aufruf MsgReceivePulse liest nur Pulse Messages aus einem Channel aus.  
Liegt keine Pulse Message im Channel, blockiert der Aufruf bis eine Pulse Message eintrifft.
- **Achtung:** Eine Pulse Message kann von einem MsgReceive Aufruf angenommen werden, obwohl ein MsgReceivePulse Call blockiert ist.

## Auswahl von QNX Message Passing Routinen

```
#include <sys/neutrino.h>
```

```
int MsgReceivePulse( int chid, void * pulse, int bytes,  
                    struct msg_info * info );
```

### Funktion:

- Diese Funktion empfängt nur Pulse Messages aus dem Channel Chid.

### Parameter:

- *Chid* : ChannelId des Channels, aus dem die Pulse Message gelesen werden soll.
- *pulse* : Puffer für die Pulse Message
- *bytes* : Die Länge von *pulse*.
- *info*: Nicht benutzt, stets NULL.
- return value : -1 : failure (Details via errno)  
sonst : 0

## Auswahl von QNX Message Passing Routinen

```
#include <sys/neutrino.h>
```

```
int MsgSendPulse(int coid, int priority, int code, int value);
```

### Funktion:

- Diese Funktion schickt eine Pulse Message an einen Channel. Da Pulse Messages asynchron sind, ist dieser Aufruf nicht blockierend. MsgSendPulse über das Netz wird nicht unterstützt.

### Parameter:

- *coid* : ConnectionId des Channels, an den die Pulse Message geschickt werden soll.
- *priority*: Priority der Pulse Message
- *code*: Code der Pulse Message (8 bit)
- *value*: ggf. Daten, die *code* ergänzen
- return value : -1 : failure (Details via errno)

## Asynchrone IPC via Pulse Messages

### Situation:

- Ein Client schickt einen Auftrag an einen Server. Da die Bearbeitung lange dauern wird, möchte der Client nicht blockiert werden.

### Implementationsansatz:

- Der Client schickt eine synchrone Message an den Server.
- Der Server bestätigt mit `MsgReply` nur, dass er die Botschaft erhalten halt.
- Client und Server arbeiten parallel weiter.
- Der Server schickt eine Pulse Message, nachdem der Auftrag erledigt ist.
- Der Client schickt eine synchrone Message an der Server und erhält das Ergebnis in der Antwort auf die Message (wenn notwendig).

Dieser Ansatz verletzt die Send Hierarchie nicht, da Pulse Messages asynchron sind.

## Asynchrone IPC via Pulse Messages (Fortsetzung)

### Implementationsansatz mit **MsgDeliverEvent**:

- Der Client schickt eine synchrone Message an den Server. Die Message enthält u.a. eine **sigevent structure**, die eine Pulse Message beschreibt. Diese Pulse Message soll der Server schicken, wenn der Auftrag fertig bearbeitet ist.
- Der Server quittiert mit **MsgReply** den Eingang des Auftrags. Der Server speichert die **sigevent structure**.
- Client und Server arbeiten parallel weiter.
- Über die Funktion **MsgDeliverEvent** löst der Server das in der **sigevent structure** beschriebene Event aus (eine Pulse Message wird an den Client geschickt), nachdem der Auftrag erledigt ist.

## Beispiel

Header file used by client.c and server.c:

```
// my_hdr.h

struct my_msg
{
    short type;
    struct sigevent event;
};

#define MY_PULSE_CODE _PULSE_CODE_MINAVAIL
#define MSG_GIVE_PULSE 12
```

## Beispiel (Fortsetzung)

Here's the client side that fills in a **struct sigevent** and then receives a pulse:

```
/* client.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/neutrino.h>
#include <sys/iomsg.h>
#include "my_hdr.h"

int main( int argc, char **argv)
{
    int chid, coid, srv_coid, rcvid;
    struct my_msg msg;
    struct _pulse pulse;

    chid = ChannelCreate( 0 ); // channel to receive pulse notification
    /* and we need a connection to that channel for the pulse to be
       delivered on */
    coid = ConnectAttach( 0, 0, chid, _NTO_SIDE_CHANNEL, 0 );
```

## Beispiel (Fortsetzung)

```
/* fill in the event structure for a pulse */
SIGEV_PULSE_INIT( &msg.event, coid, SIGEV_PULSE_PRIO_INHERIT,
                  MY_PULSE_CODE, 0 );
msg.type = MSG_GIVE_PULSE;

/* find the server */
srv_coid = ...

/* send message to server */
MsgSend( srv_coid, &msg, sizeof(msg), NULL, 0 ); // synchron

/* wait for the pulse from the server */
rcvid = MsgReceivePulse( chid, &pulse, sizeof( pulse ), NULL );
printf("got pulse with code %d, waiting for %d\n",
       pulse.code, MY_PULSE_CODE );
return 0;
}
```



## Beispiel (Fortsetzung)

Here's the server side that delivers the pulse defined by the struct sigevent:

```
/* server.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/neutrino.h>
#include <sys/iomsg.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>
#include "my_hdr.h"

int main( int argc, char **argv)
{
    int rcvid;
    struct my_msg msg;
    int chid;

    // create the channel
    ...
}
```

## Beispiel (Fortsetzung)

```
/* wait for the message from the client */
rcvid = MsgReceive(chid, &msg, sizeof( msg ), NULL );
MsgReply(rcvid, 0, NULL, 0);
if ( msg.type == MSG_GIVE_PULSE ) {
    /* wait until it is time to notify the client */
    sleep(2);
    /* deliver notification to client that client requested */
    MsgDeliverEvent( rcvid, &msg.event );
    printf("server:delivered event\n");
} else {
    printf("server: unexpected message \n");
}
return 0;
}
```

## Anmerkungen

- Der Client definiert das Event, das geschickt werden soll. Der Server behandelt die sigevent Struktur wie eine Black Box, die mit `MsgDeliverEvent` verschickt wird.
- sigevent Structure : siehe Interrupt Kapitel
- Im Aufruf von `MsgDeliverEvent` bindet der Server die sigevent Structure an die `rcvid`.  
Achtung: Nach dem Aufruf von `MsgReply` ist die `rcvid` ungültig. `MsgReply` wurde vor `MsgDeliverEvent` abgeschickt. Damit sollte die `rcvid` ungültig sein, aber bei `MsgDeliverEvent` liegt eine Ausnahme vor.

## Auswahl von QNX Message Passing Routinen

```
#include <sys/neutrino.h>
```

```
int MsgDeliverEvent (int rcvid, const struct sigevent* event);
```

### Funktion:

- Diese Funktion schickt eine Pulse Message an einen Client (wurde in der sigevent Struktur entsprechend festgelegt). Die Message hat der Client vorab selbst zusammengestellt.

### Parameter:

- *rcvid* : siehe vorherige Seite.
- *event* : Die Pulse Message, die geschickt werden soll.
- return value : -1 : failure (Details via errno)

## Nachtrag : Channel Flags

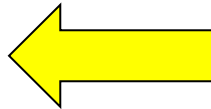
Die Funktion ChannelCreate hat als Eingabe einige Flags, die Eigenschaften des Channels festlegen. Hier eine Auswahl dieser Flags:

- `_NTO_CHF_FIXED_PRIORITY` : Priority Inheritance wird deaktiviert.
- `_NTO_CHF_THREAD_DEATH` : Der Kernel schickt eine Pulse Message an den Server, wenn eine Thread stirbt, der aufgrund des Channels blockiert ist.
- `_NTO_CHF_SENDER_LEN` : Der Kernel speichert die Länge der Message, die der Client geschickt hat, in der `_msg_info` Struktur.
- `_NTO_CHF_REPLY_LEN` : Der Kernel speichert die Länge des Antwort Puffer, den der Client im `MsgSend` bereitgestellt hat, in der `_msg_info` Struktur.
- `_NTO_CHF_UNBLOCK` : Ein Client kann aufgrund eines Signals oder eines Kernel Timeout den Zustand `SEND blocked` bzw. `REPLY blocked` verlassen. Ist dieses Flag gesetzt, wird der Server informiert, dass der Client den Zustand `blocked` verlassen möchte. Der Server befreit dann den Client aus dem `blocked` Zustand.

## Kapitel 3 : Inter Process Communication (IPC)

### Gliederung

- Grundbegriffe der Inter Process Communication (IPC)
- Synchrones Message Passing: die Grundlage des QNX Realtime OS
- Client / Server Konzepte
- Asynchrone Kommunikation: QNX Pulses
- Zusammenfassung



## Zusammenfassung

- Grundlegende Begriffe der IPC
- Synchrones Message Passing: Das QNX Synchronisationskonzept
- MsgRead, MsgWrite und IOV ersparen „unnötiges“ Kopieren von Pufferinhalten.
- Client / Server Modell, Server with multiple threads, Server/Subserver Modell
- Send Hierarchie vermeidet Deadlocks
- Asynchrone Kommunikation via Pulse Message, MsgDeliverEvent
- Channel Flags