Echtzeit-Betriebsysteme, Realzeit-Programmierung und Eingebettete Systeme
(Realtime Operating Systems, Realtime Programming and Embedded Systems)

Inhaltsverzeichnis

Document History

Version Date	Author(s) email address	Changes and other notes
8.4.2007	ludwig.eckert@fh-sw.de	Übung zu Counting Semaphore und Mutual Exclusion Semaphore eingefügt
25.4.2007	ludwig.eckert@fh-sw.de	Übungen zur Tasksychronisation eingefügt
13.10.2009	ludwig.eckert@fhws.de	Änderungen Kap.3 u. Formatierungen vorgenommen
2.10.2010	ludwig.eckert@fhws.de	Kap. 2: Anwendungsfelder von Embeddded Systemen erweitert
25.10.2011	ludwig.eckert@fhws.de	In Kap. 3.2.3 Bild zu Timer-Strukturen hinzugefügt.
2.6.2012	ludwig.eckert@fhws.de	Kap. 7.2.9. Übung 5 zur Tasksynchronisation eingefügt
14.10.2013	ludwig.eckert@fhws.de	Literaturverzeichnis aktualisiert

1. LITERATUR

[1] VxWorks Documentation
VxWorks Programmer's Guide V 5.5 (Windriver Inc.) (539 Seiten, 3.9 MB)
VxWorks Whitepapers und Infos zu weiteren Entwicklungstools

[2] D. Zöbel, W. Albrecht:
"Echtzeitsysteme: Grundlagen und Techniken",
Thomson, Bonn, 1995

[3] Mark H. Klein, et al.:
"A practitioner's handbook for real-time analysis",
Kluwer, Boston, 1993

[4] André M. von Tilborg, Garx M. Doob
"Foundations of Real-Time Computing Formal Specifications and Methods",
Kluwer Academic Publishers 1991

[5] Erik Jacobson
"Prozessdatenverarbeitung"
Hanser Verlag 1989

Prozessrechentechnik

[6] Färber, Go
Prozessrechentechnik
Prozessautomatisierung, Prozessrechner-Hardware, Echtzeitverhalten
3. Auflage, Springer, 1994

[7] Lauber: Prozessautomatisierung
Technische Prozesse, Prozessrechner
Band I, Springer 1989

[8] Schnieder: Prozessinformtik
Technische Prozesse, Prozessautomatisierung, Echtzeitsysteme
Vieweg 1986

Betriebssysteme, Netzwerktechnik

[9] Tanenbaum A.S.:
Moderne Betriebssysteme
Grundlagen, Betriebssystemkonzepte, Fallbeispiele
2. Auflage, Hanser, 1995

[10] Eonic
Virtuoso user guide 401, book I (verteiltes DSP-Betriebssystem)
2000

Echtzeitanalyse, Design und Programmierung

[11] Halang, Wolfgang A. and Alexander D. Stoyenko
Constructing Predictable Real Time Systems
Norwell, Massachusetts, Kluwer Academic Publishers Group, 1991

[12] Ganssle, Jack G.
The Art of Programming Embedded Systems
San Diego, California, Academic Press, Inc., 1992

[13] Allworth, Steve T.
Introduction To Real-Time Software Design
New York, New York, Springer-Verlag, 1981

[14] Klein, Mark H., Thomas Ralya, Bill Pollak, Ray Harbour Obenza,
and Michael Gonzlez
A Practioner's Hardbook for Real-Time Analysis
Guide to Rate Monotonic Analysis for Real-Time Systems
Norwell, Massachusetts, Kluwer Academic Publishers Group,

[15] Ripps, David L.
An Implementation Guide To Real-Time Programming
Englewood Cliffs, New Jersey, Yourdon Press, 1989

[16] Laplante, Phillip A.
Real-Time Systems Design and Analysis, An Engineer's Handbook
Piscataway, NJ 08855-1331, IEEE Computer Society Press

[17] Lehoczky, John, Lui Sha, and Ye Ding
The Rate Monotonic Scheduling Algorithm: Exact Characterization
and Average Case Behavior.
Proceedings of the IEEE Real-Time Systems Symposium, Los Alamitos, CA,
1989, pp. 166-17, IEEE Computer Society

[18] Wood, Mike and Tom Barrett
A Real-Time Primer
Embedded Systems Programming, February 1990 p20-28

[19] Awad, M.; Kuusela, J.; Ziegler, Jo
Object-Oriented Technology for Real-Time Systems,
Objekt-orientiertes Design für Echtzeitsysteme (Octopus Methode), Fallbeispiele
Prentice-Hall, 1996

[20] Barr, M.
Programming Embedded Systems
Programmierung, Entwicklungswerkzeuge, Speicher und Peripherie
O'Reilly, 1999

[21] Cooling
Software Design for Real-Time Systems
Software-Design Methodik, Echtzeitsprachen, Betriebssysteme, Analysetools, Test
Chapmann and Hall, 1991

[22] Hatley, Pirbhai
Strategien für die Echtzeitprogrammierung
Analyse und Design von Echtzeitsystemen
Hanser, 1993

[23] Zöbel, D.; Albrecht, W.
Echtzeitsysteme - Grundlagen und Techniken
Echtzeitanalyse, Echtzeitprogrammierung
Informatik Lehrbuch Reihe, International Thomson Publishing, 1995

2. ANWENDUNGSFELDER VON EMBEDDED SYSTEMEN

Automatisierung von Maschinen	Automatisierung technischer Anlagen
Werkzeugmaschinen	Kraftwerksanlagen
Industrieroboter	(Dampferzeuger,
Sensorsysteme (Bildverarbeitung)	Turbinen, Generatoren)
Kraftfahrzeuge	Energieversorgungsanlagen
(Motorsteuerung,	(Lastverteiler
Bremssystem,	Netzleitsysteme)
Getriebesteuerung,	Fertigungssysteme
Fahrtplanung,	Stahlwerksanlagen
Abstandswarnsystem, usw.)	Walzwerksanlagen
Prüfmaschinen	Schienenverkehrssysteme
Messmaschinen	(Fernbahnen,
Küchengeräte	Stadtbahnen,
Waschmaschinen	U-Bahnen)
Nähmaschinen	Gasversorgungssysteme
Alarmanlagen	Prüfstände, Prüffelder
Navigationssysteme	Verfahrenstechnische Anlagen
Heizungssysteme	Laborautomatisierung
Spielzeuge	Gebäude- und haustechnische Anlagen,
Filmkameras	Aufzüge
Telefonanrufbeantworter	Intensivstation
	Hochregallager
	Verkehrstechnik
	Papiermaschinen
	Textilmaschinen

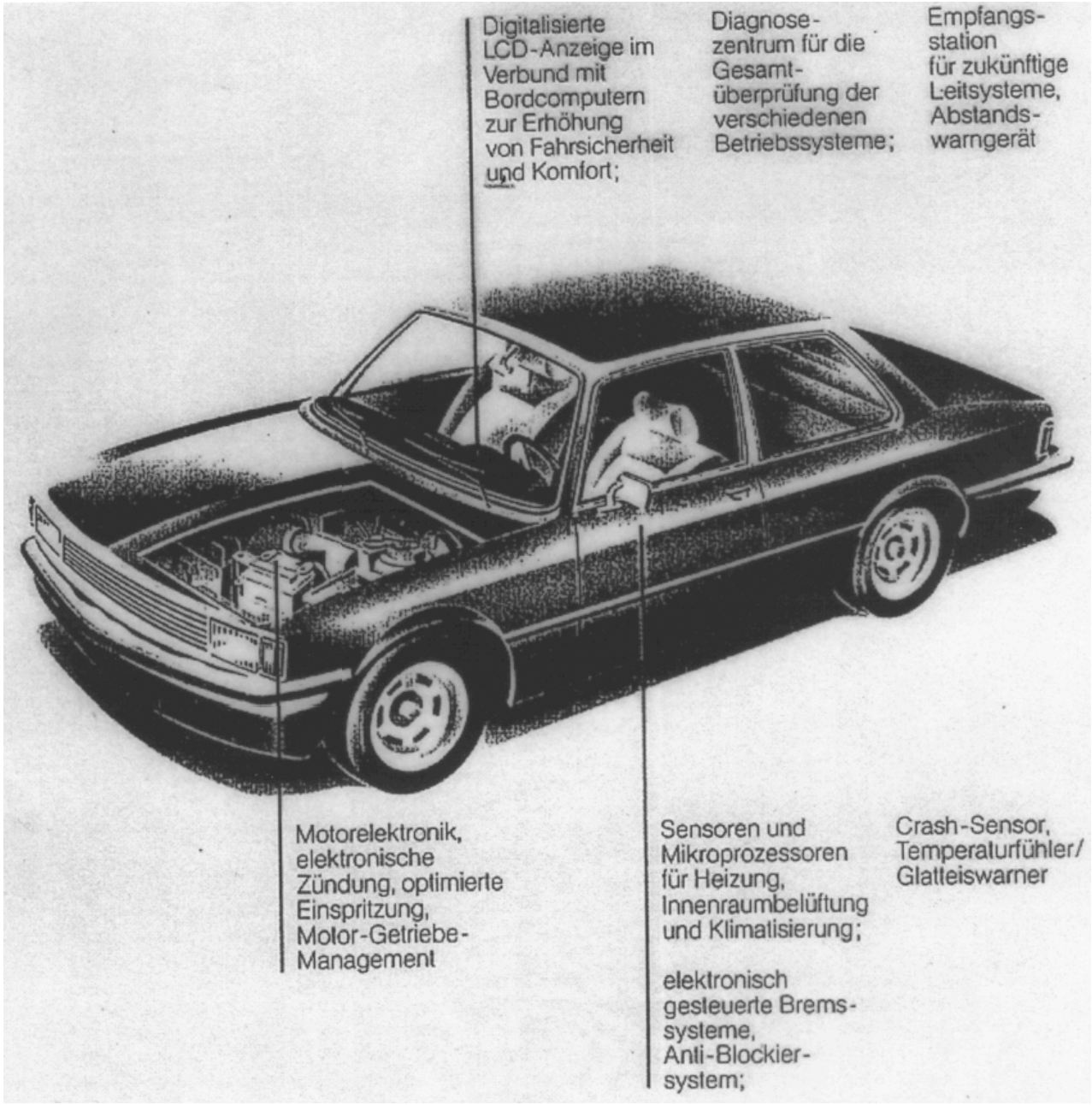


Bild: Beispiele für den Einsatz der Elektronik im Auto - gestern

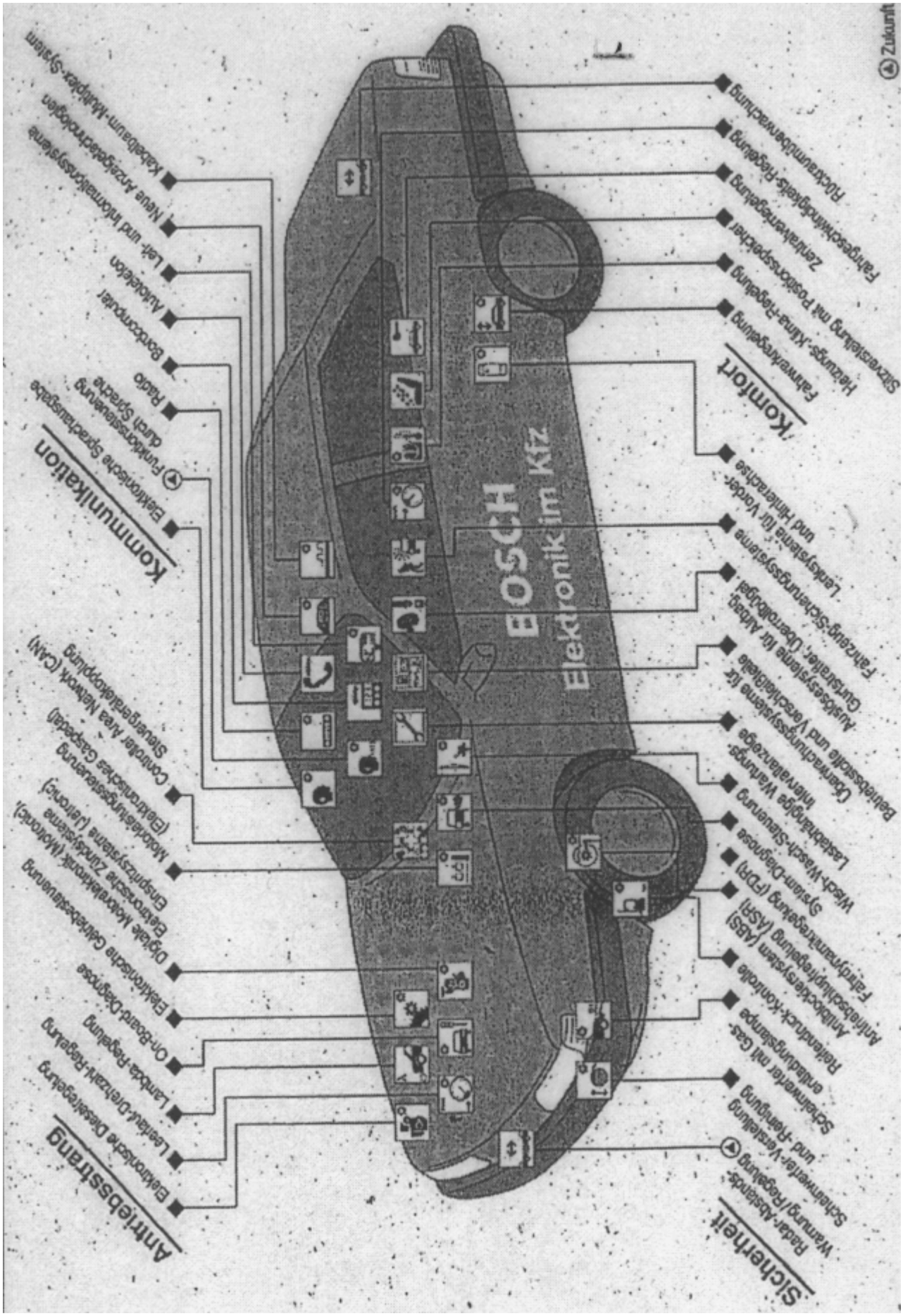
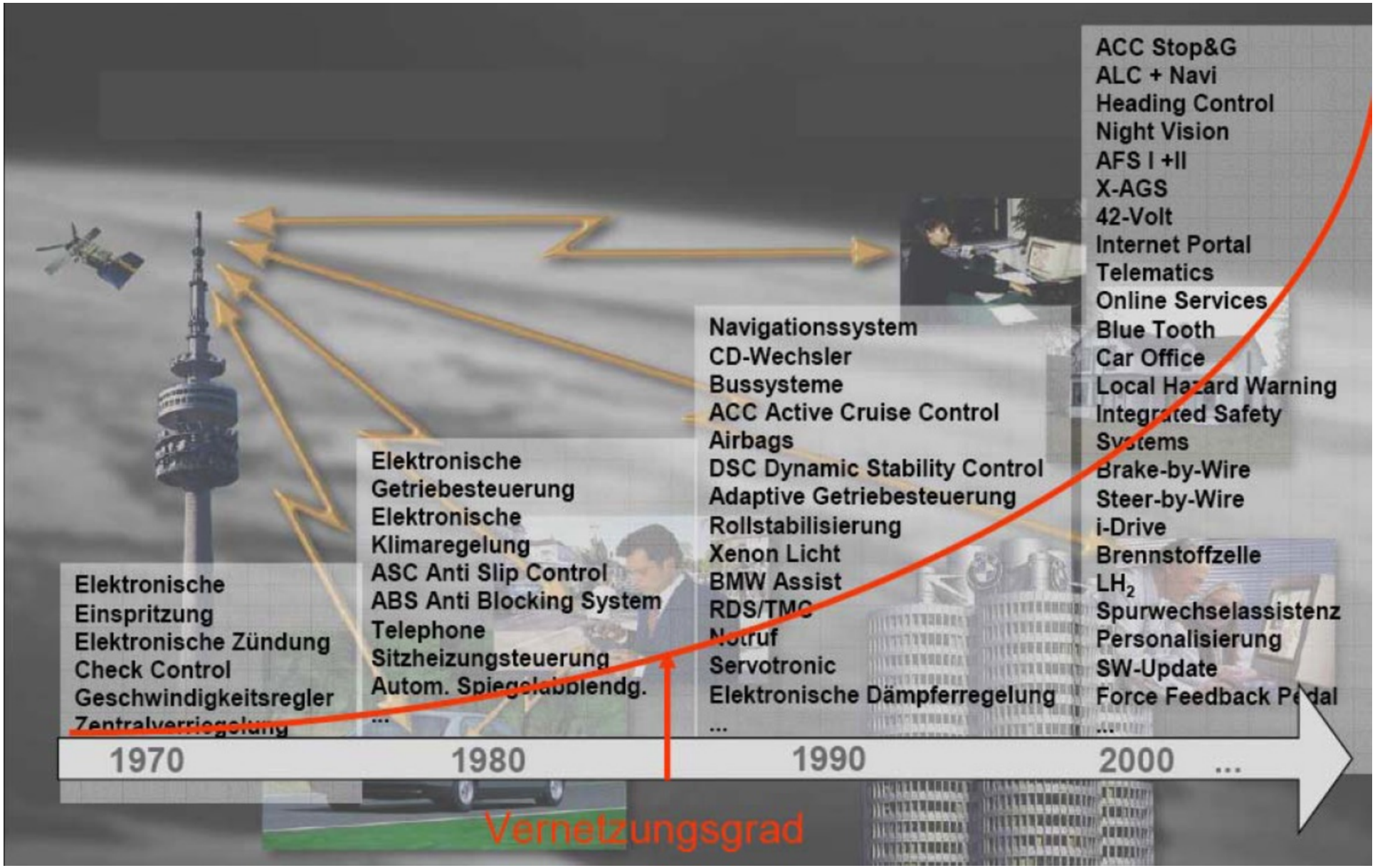


Bild: Beispiele für den Einsatz der Elektronik im Auto - und heute

Elektronik gehört zur Steuerung von Sicherheit (s. a. Fahrerassistenzsysteme) und Komfort im Auto längst zur unverzichtbaren Ausstattung.


ACC Stop&G
ALC + Navi
Heading Control
Night Vision
AFS I +II
X-AGS
42-Volt
Internet Portal
Telematics
Online Services
Blue Tooth
Car Office
Local Hazard Warning
Integrated Safety
Systems
Brake-by-Wire
Steer-by-Wire
i-Drive
Brennstoffzelle
LH_2
Spurwechselassistenz
Personalisierung
SW-Update
Force Feedback Pedal
...
Navigationssystem
CD-Wechsler
Bussysteme
ACC Active Cruise Control
Airbags
DSC Dynamic Stability Control
Adaptive Getriebesteuerung
Rollstabilisierung
Xenon Licht
BMW Assist
RDS/TMC
Notruf
Servotronic
Elektronische Dämpferregelung
...
Elektronische
Getriebesteuerung
Elektronische
Klimaregelung
ASC Anti Slip Control
ABS Anti Blocking System
Telephone
Sitzheizungsteuerung
Autom. Spiegelabblendg.
...
Elektronische
Einspritzung
Elektronische Zündung
Check Control
Geschwindigkeitsregler
Zentralverriegelung
1970
1980
1990
2000
...
Vernetzungsgrad

Beispiele für den Einsatz von Embedded Systemen in der Prozessautomatisierung

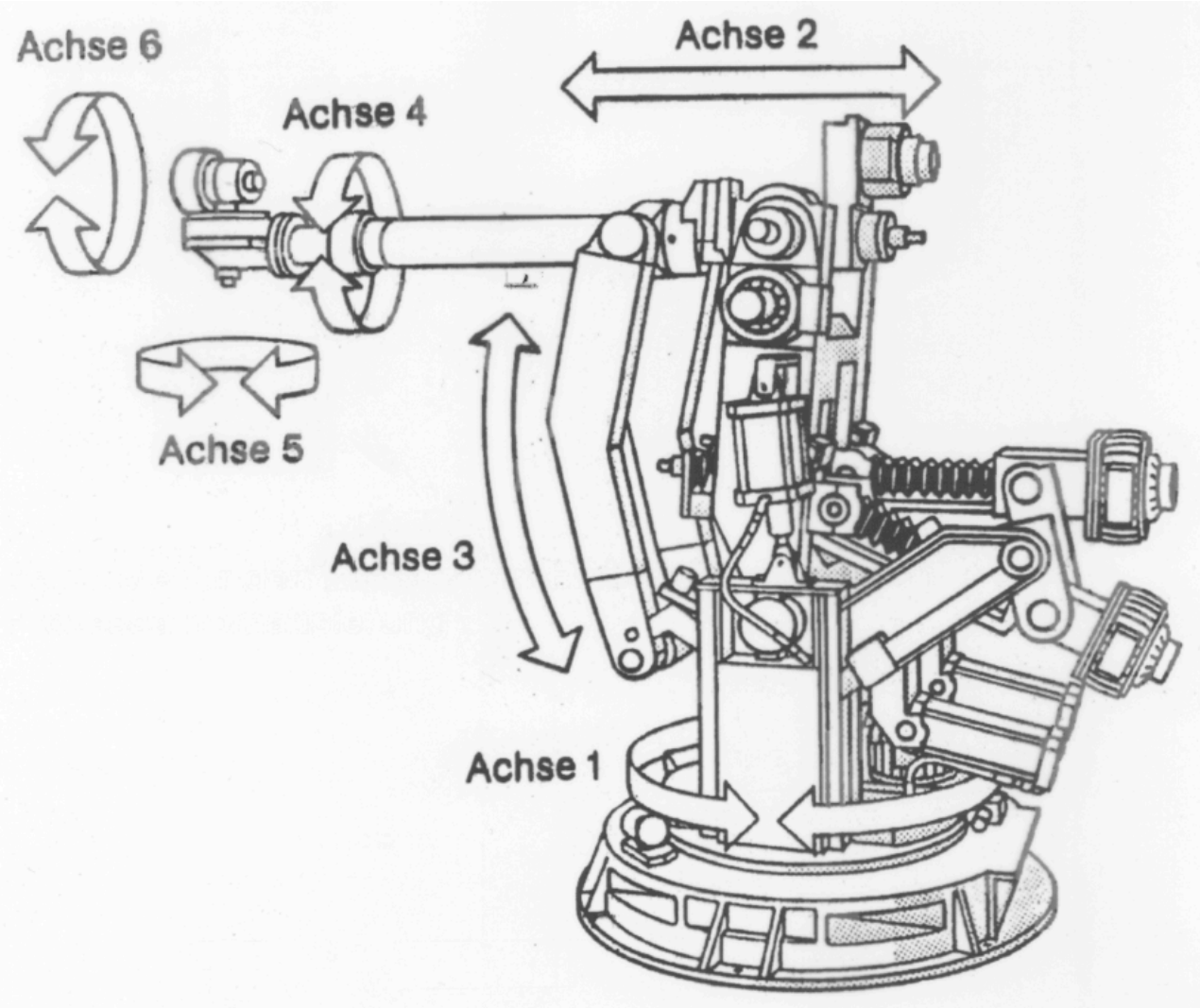


Bild: Industrieroboter mit 6 Achsen

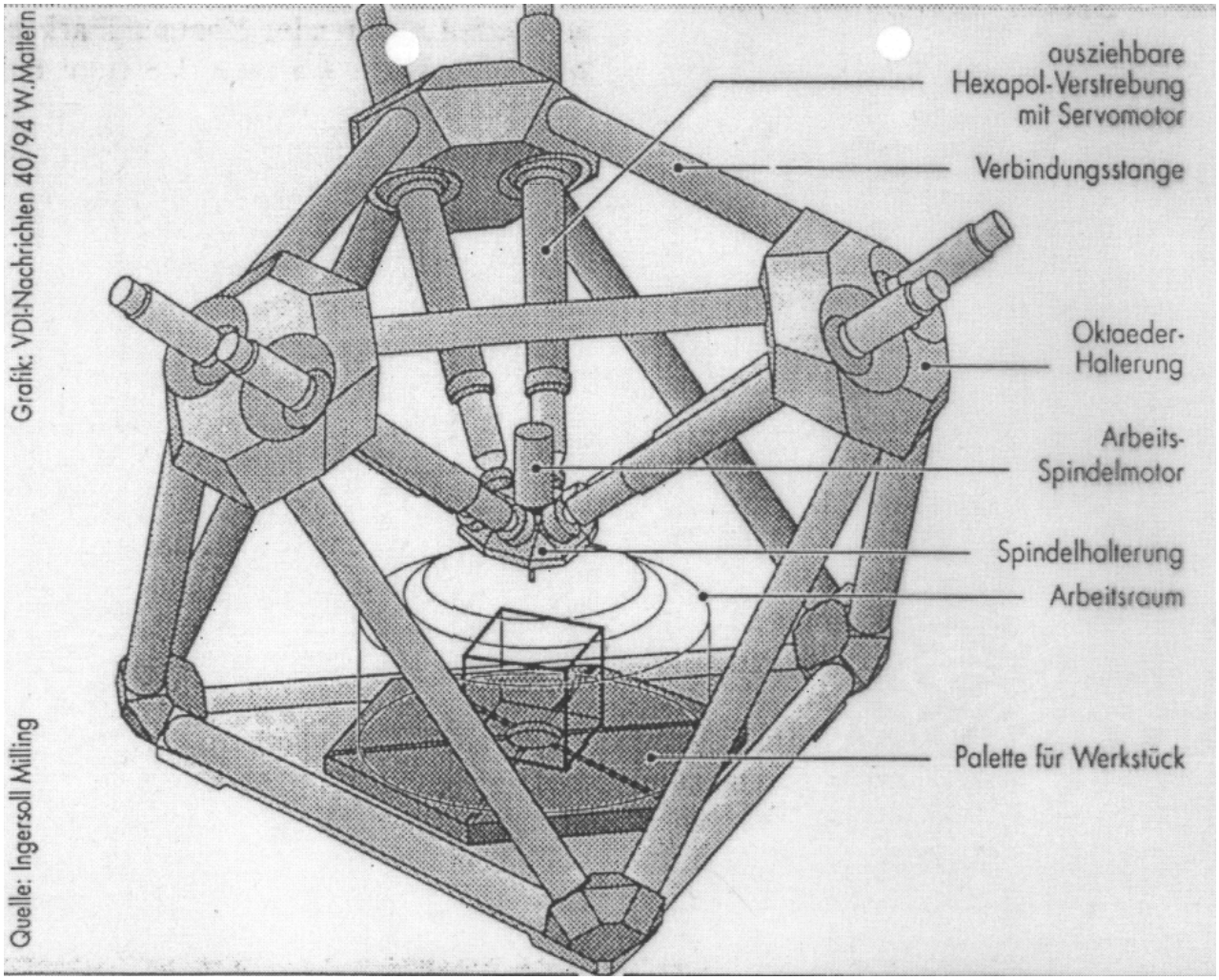
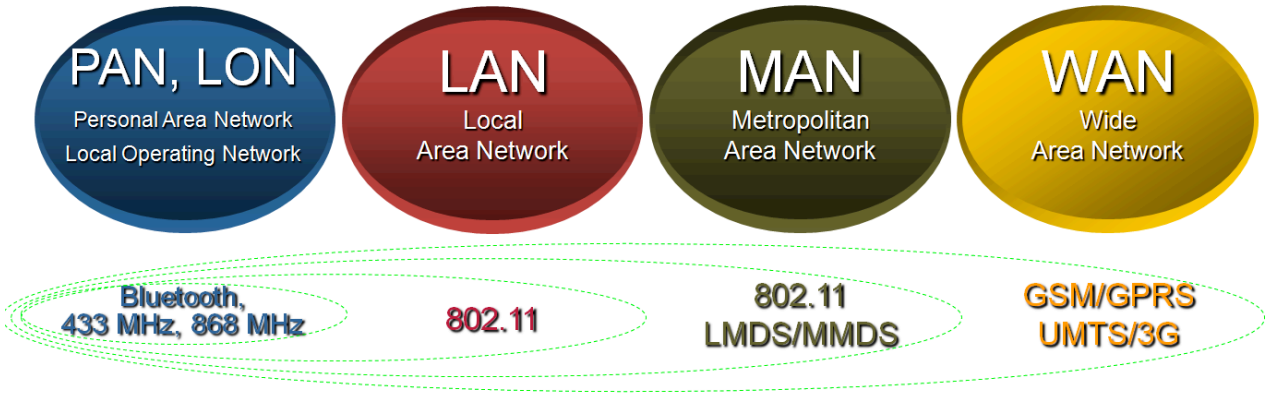


Bild: Moderne Werkstückbearbeitung mit der Hexapod-Maschine

Der oktaederförmige Maschinenkörper basiert auf zwölf Streben, die acht Dreieckselemente bilden. Über der Werkstück-Spannfläche im unteren Teil der 3 m hohen „Octahedral Hexapod"-Maschine befindet sich die Trägerplattform für Bearbeitungseinheiten, die von sechs Teleskopstangen frei im Raum bewegt werden kann. Alle Kräfte werden an den Ecken der acht Flächenelemente eingeleitet, so daß keine Biegemomente auftreten. *Bild: Ingersoll*

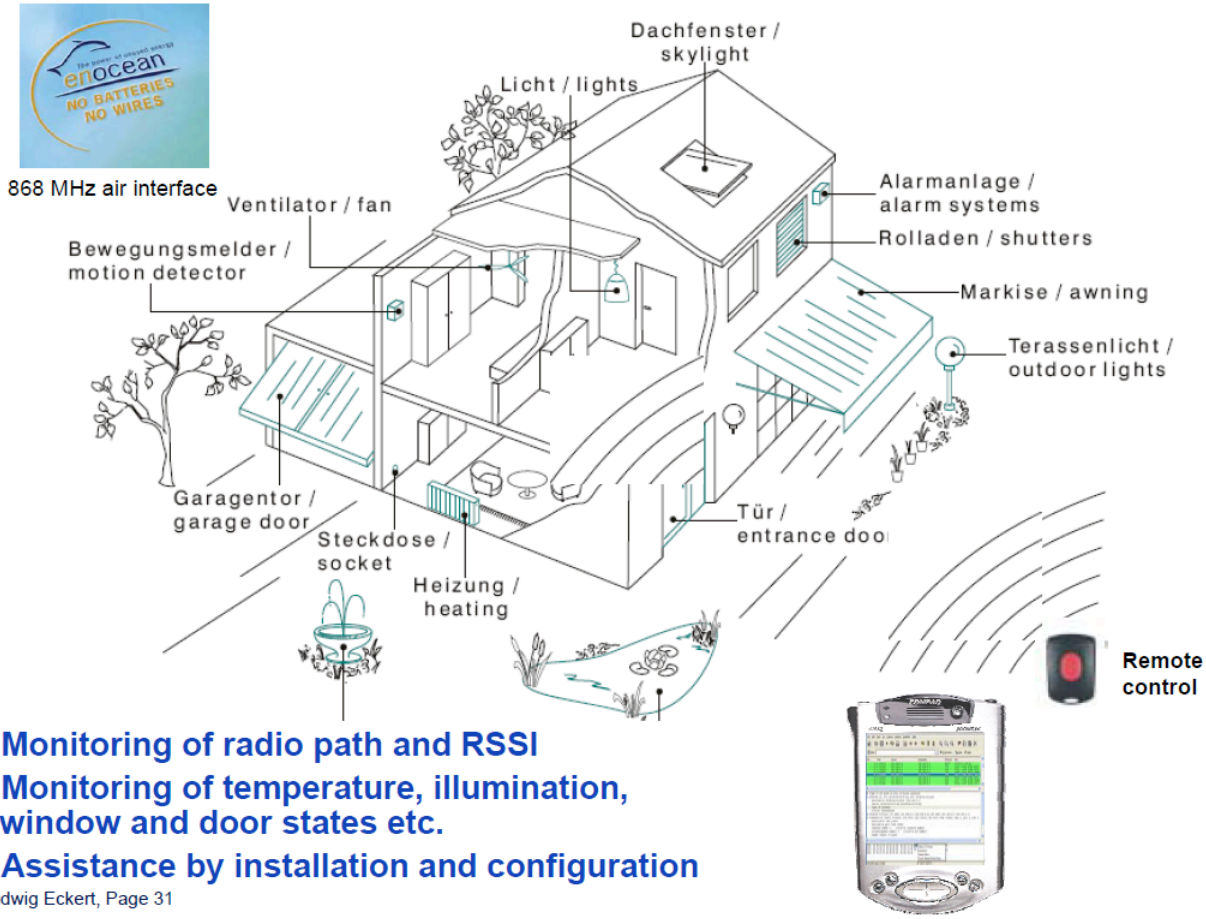
Anwendungsfeld: Medizintechnik

Point of Interest: Wireless Frequency Ranges



Short distances	Medium distances	Med-long distances	Long distances
Low cost	Medium cost	Medium cost	High cost
< 1 Mbps	1 to 54+ Mbps	22+ Mbps	10 Kbps to 2Mbps
Laptop/PC to devices (printer/keybrd/phone sensors, actuators)	Computer-computer and to the Internet	Fixed, last-mile access	PDA devices and handhelds to Internet

Usage: Multilink Wireless Networks in Home Automation Systems



- Monitoring of radio path and RSSI
- Monitoring of temperature, illumination, window and door states etc.
- Assistance by installation and configuration

Ludwig Eckert, Page 31

WSN = Wireless Sensor Network

Drahtlose, meist funkbasierte Vernetzung von intelligenten Sensordatenerfassungssystemen, sog. Motes

Möglichst niedriger Energieverbrauch

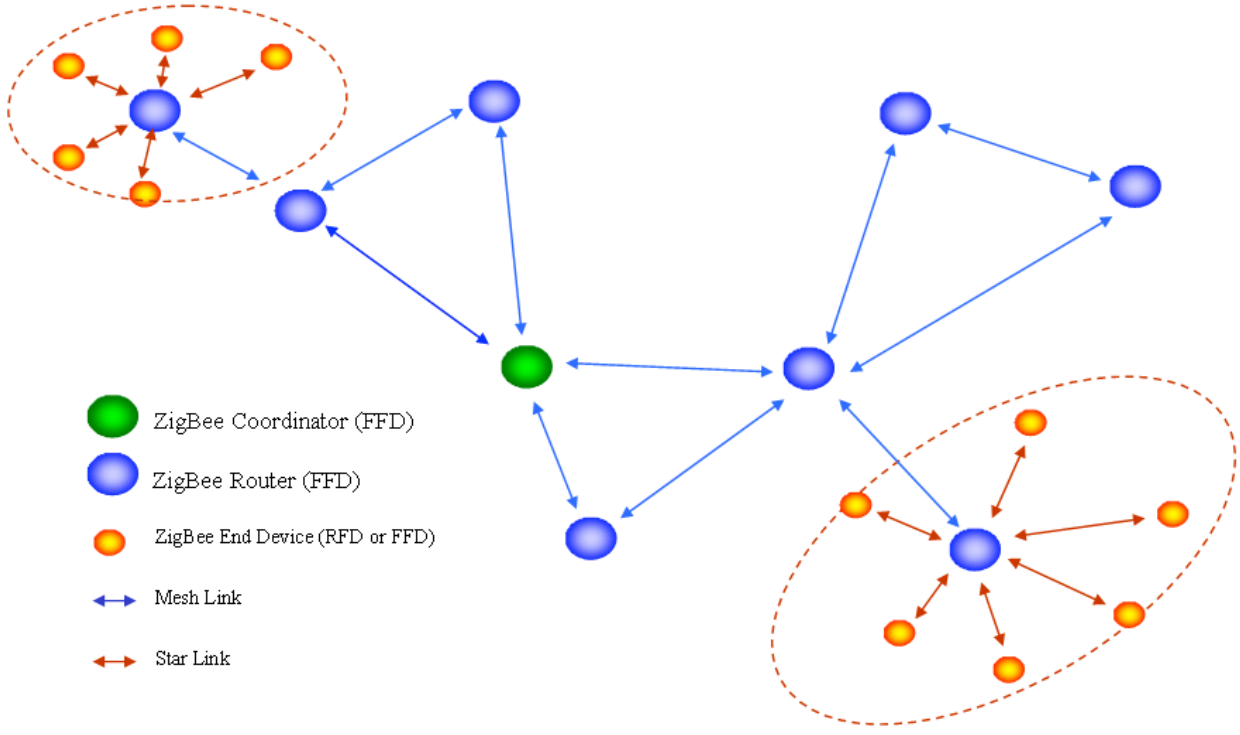
Motes bilden ein Maschennetzwerk, andere Topologien werden auch genutzt

Dynamisches Netz
- -Motes bilden ein Maschennetzwerk oder andere Topologien
- -Multi-Hop oder Single-Hop Routing mit den verschiedensten Algorithmen
- -Kommunikation durch proprietäre Protokolle im ISM-Band oder ZigBee
- -Uni-, Multi- und Broadcast Unterstützung

SoCs geeignet für größere Stückzahlen

Batterien als Stromquelle mit Möglichkeit von energy harvesting

Beispiel: ZigBee Netz



Einsatzzweck:
- Großfläche Datenerfassung (Schneebrettüberwachung, Waldbrandüberwachung etc.)
- Gebietsüberwachung (Gefechtsfeldüberwachung)
- Gebäudesteuerung (HVAC)
- Lichtschalter (EnOcean)
- Rauch-/Feuermelder
- u. a.

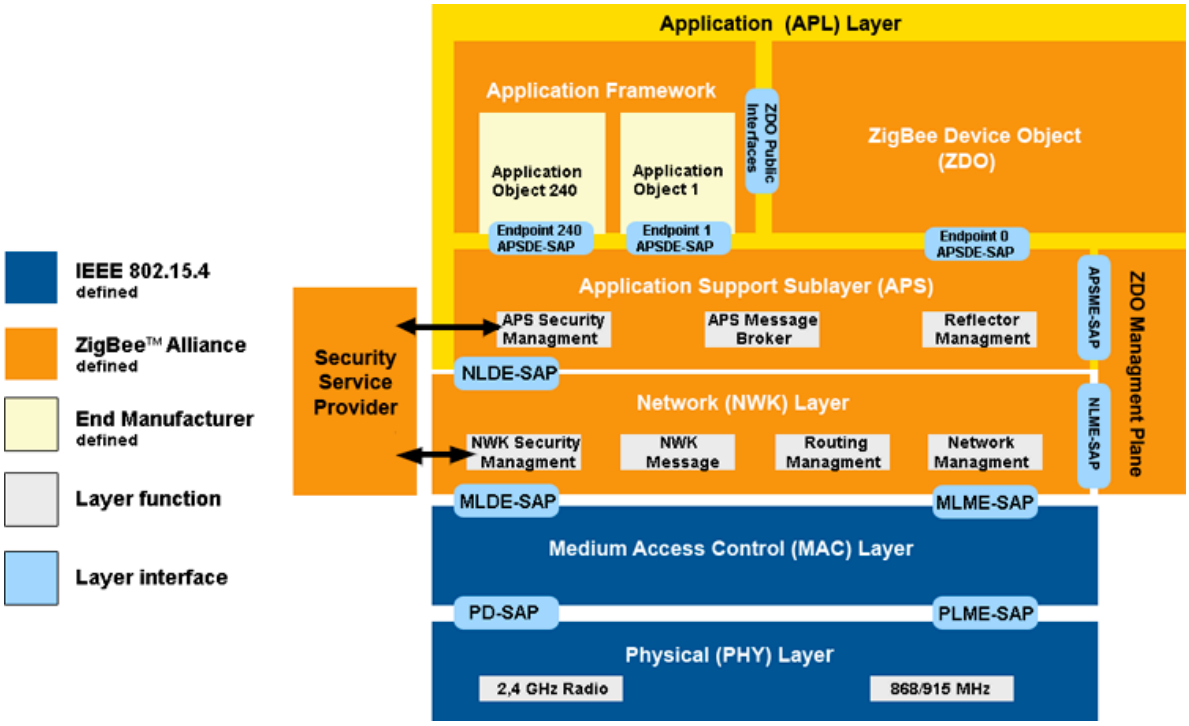
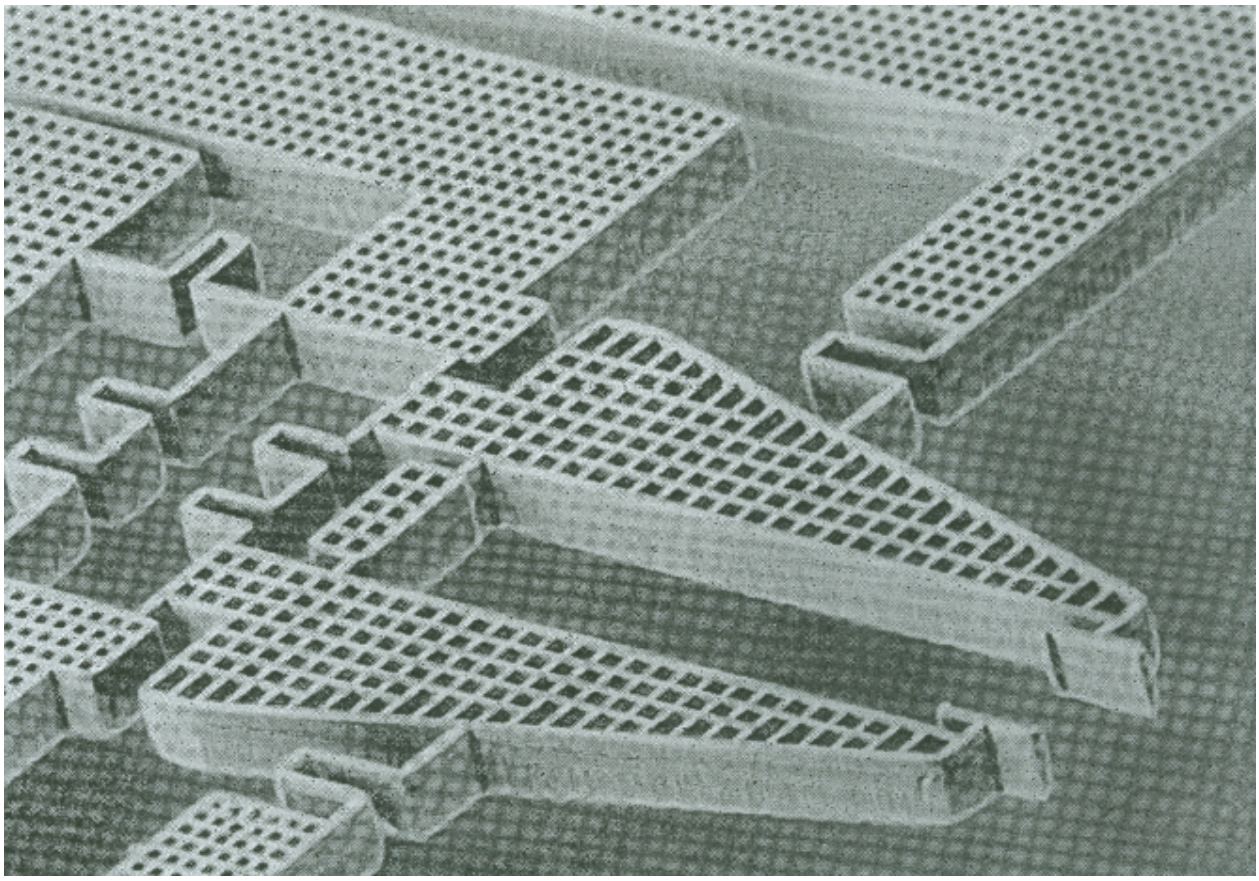


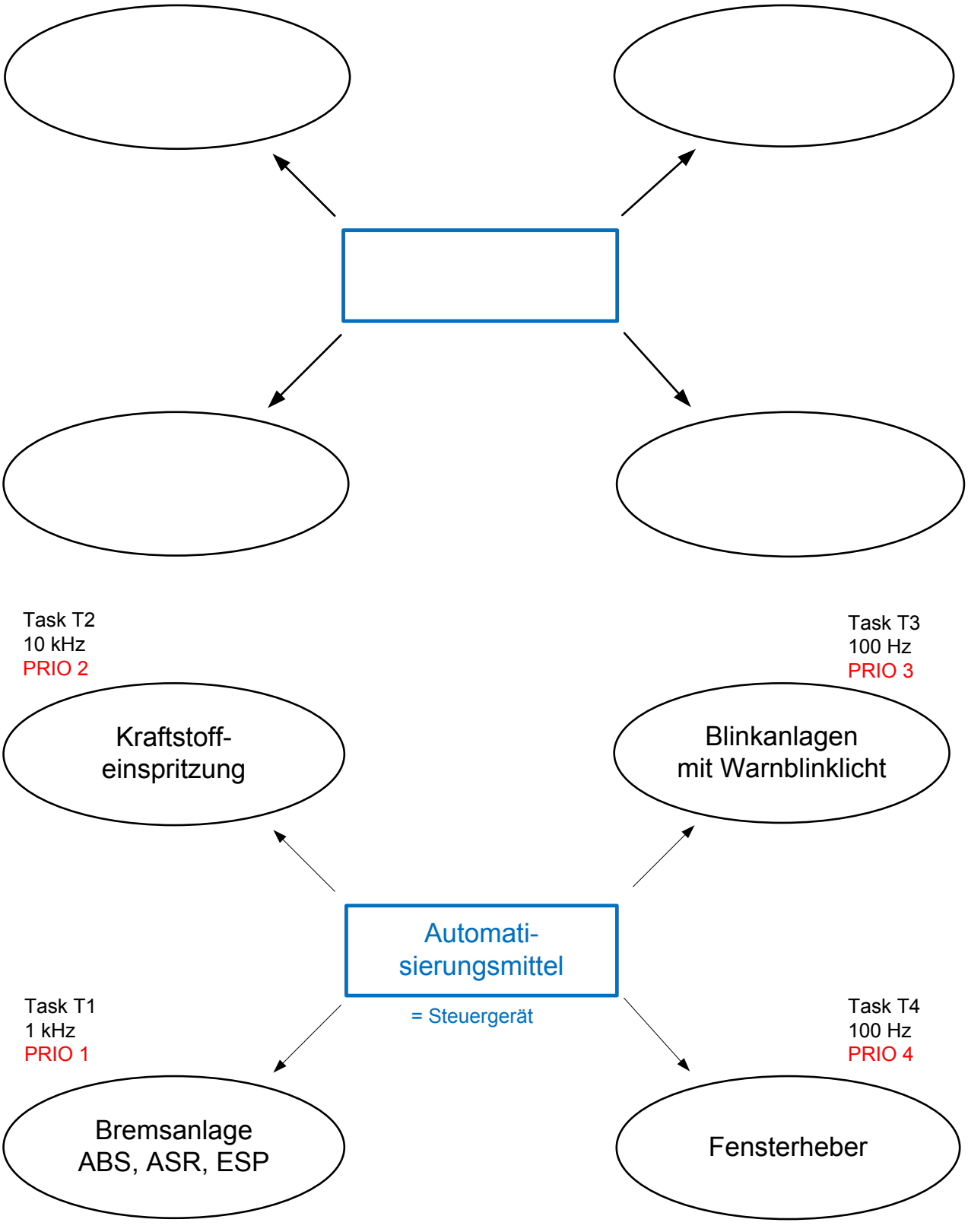
Bild: ZigBee Protokollstack



Einen winzigen Greifer haben kalifornische Wissenschaftler für die Montage von Mikrostrukturen entwickelt. Die Greifzangen auf der Abbildung sind etwa hundert Mikrometer weit geöffnet. Mit dem sogenannten Hexsil-Verfahren können solche Hohlstrukturen aus Polysilizium hergestellt und in einem weiteren Schritt mit dotiertem Polysilizium oder mit Metall aufgefüllt werden. Allerdings besteht nicht die gesamte Konstruktion des Greifers aus gefüllten Hohlstrukturen, sondern nur der thermische Antrieb und die elektrischen Zuleitungen. Für die Finger und alle übrigen Teile ist dieser Aufwand nicht nötig. Das Füllmaterial für den thermischen Antrieb ist dotiertes Polysilizium, das sich im Gegensatz zum undotierten Polysilizium ausdehnt, sobald ein elektrischer Strom zu fließen beginnt. Dieser Effekt läßt sich dazu verwenden, über entsprechende Hebel die Finger des Greifers in Bewegung zu setzen. Die Hohlstrukturen für die elektrischen Zuleitungen des thermischen Antriebs werden dagegen mit Nickel aufgefüllt. Foto Universität von Kalifornien, Berkeley

2.1. Anwendungsszenario: Automatisierung im PKW

Aufgabenstellung:



∑ Automatisierungsmittel = Automatisierungssystem (Steuerungssystem)

Anforderungen an das Steuergerät:

- Gleichzeitigkeit
- Rechtzeitigkeit
- Schnelligkeit
- Vorhersagbarkeit (Determinismus)
- !!!Zuverlässigkeit / Verfügbarkeit / Sicherheit (Safety, Betriebsbewährtheit, Security
- !!!Diagnosefähigkeit

Gegebenheiten

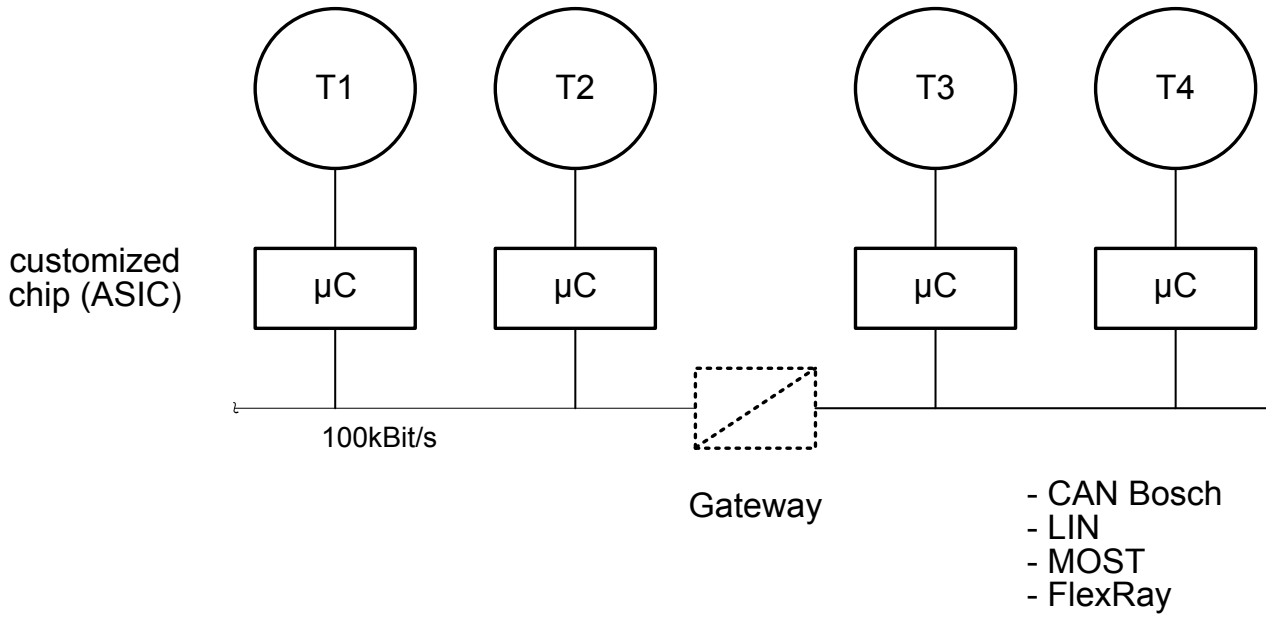
- endliche Rechenzeit
- unterschiedliche Prioritäten der Anwendungen
- unterschiedliche Periodizitäten

2.2. Prinzipielle Lösungsarchitekturen

Nachstehend werden prinzipielle Lösungsarchitekturen zur parallelen und gleichzeitigen Aufgabenbearbeitung vorgestellt.

2.2.1. Lösungsvariante 1: Einsatz eines Mehrprozessorsystems

Prinzip:



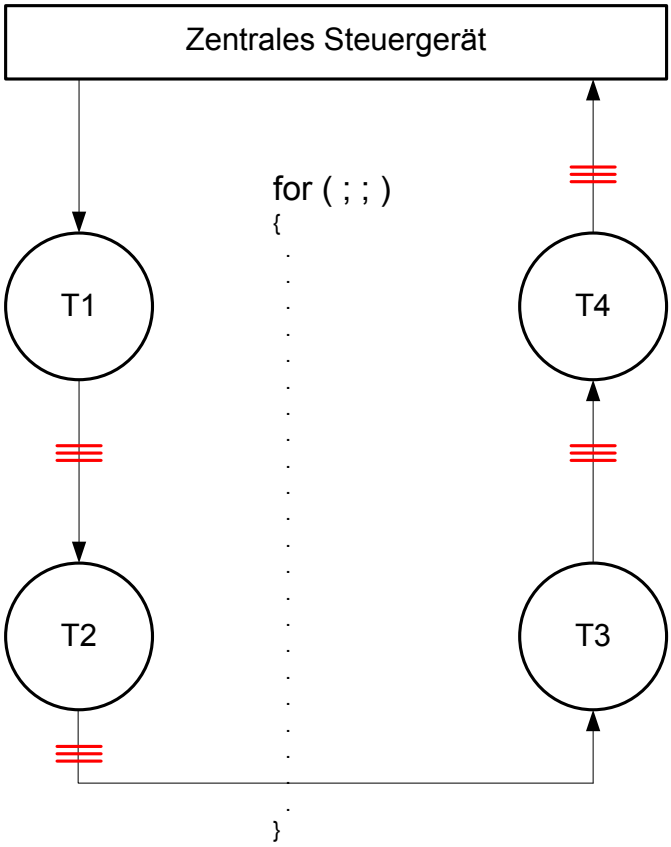
Vorteile

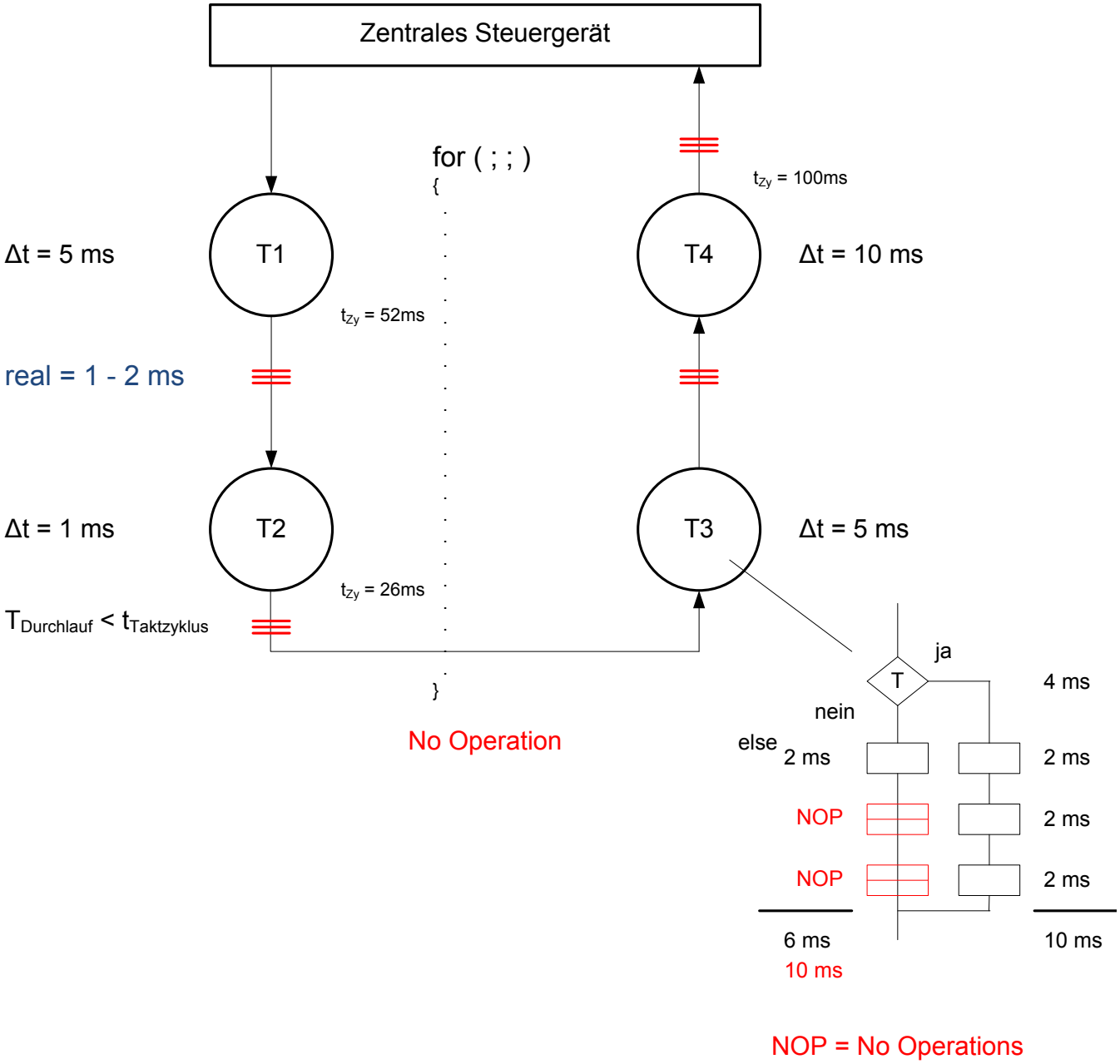
- Transparente Aufbaustruktur
- Einfaches Handling im Reparaturfall
- Parallele Entwicklung der Steuereinheiten möglich, Möglichkeit für Zukaufteile

Nachteile

- Teure Lösung wegen verteilter Multiprozessorarchitektur
- Problem der externen Kommunikation (Interoperabilität, Konformität)

2.2.2. Lösungsvariante 2: Ein-Prozessorlösung (Daisy-Chain)


Zentrales Steuergerät
for (; ;)
{
T1
T2
T3
T4
}



Vorteile

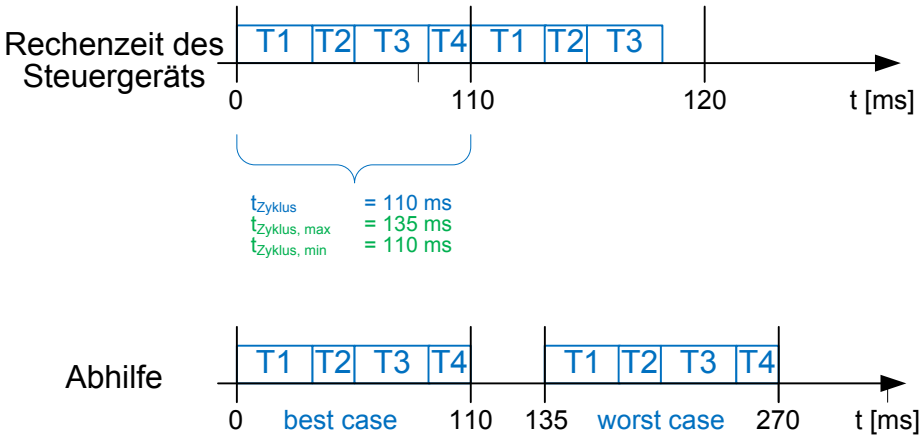
- Preiswerte Ein-Prozessor-Architektur
- Ein-Prozessor-Lösungen sind preiswerter

Nachteile

- Unterschiedliche zeitliche Anforderungen der Tasks (Periodizitäten) der Tasks

und variable Durchlaufzeiten erschweren die Programmierung

- Gefahr der blockierenden Tasks
- Rechtzeitigkeit (Äquidistanz) durch Verzögerungen bei variablem Kontrollfluß schwierig herstellbar -> Bsp. Praktikum




```
Funktion  void main(void)
    char zeichen;                   // Speicher fuer empfangenes Zeichen
    WDTCTL = WDTPW + WDTHOLD;       // Stop watchdog timer
    init_uart();
    for ( ;; )
        while ( (IFG2 & URXIFG1)==0 )
            ;   // Warten auf empfangenes Zeichen Interrupt Flag auslesen!
        zeichen=RXBUF1;             // empfangenes Zeichen im Speicher lesen
        while ( (IFG2 & UTXIFG1)==0 )
            ;   // Ist Transmitter frei fuer neue Daten?
        TXBUF1=zeichen;             // Zeichen zurueck senden
```

Bild: Negativbeispiel für blockierendes Empfangen und Senden von Daten über die V.24 Schnittstelle

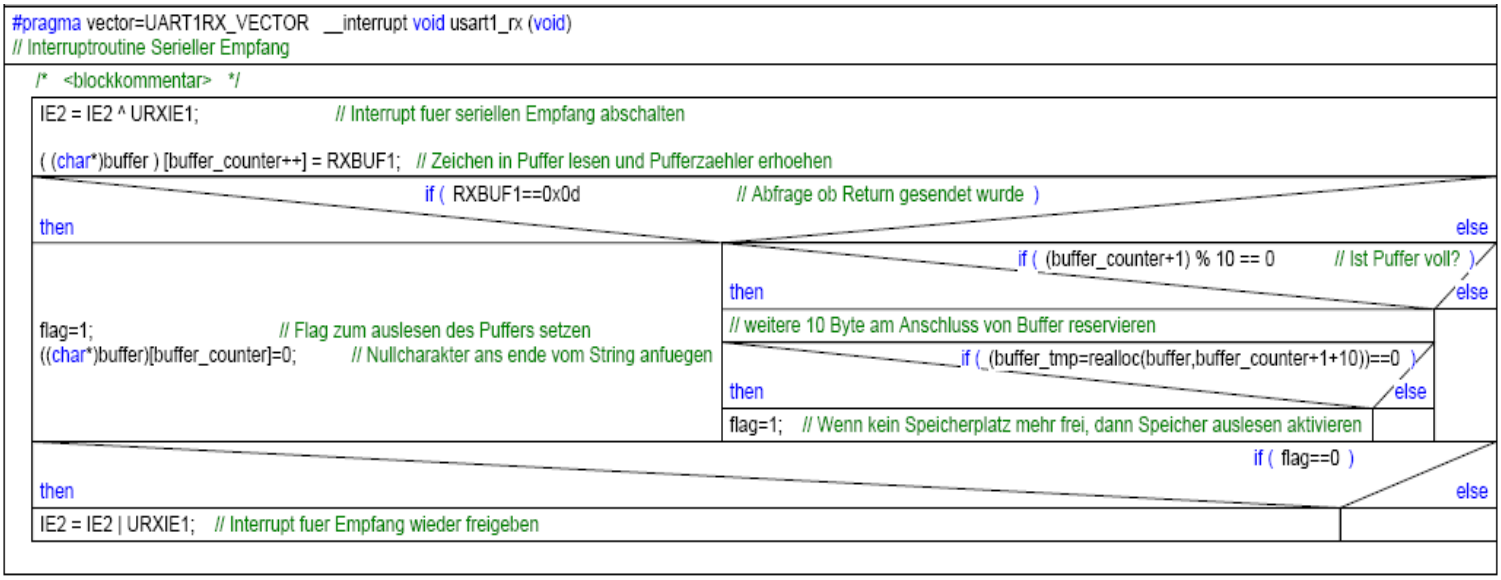
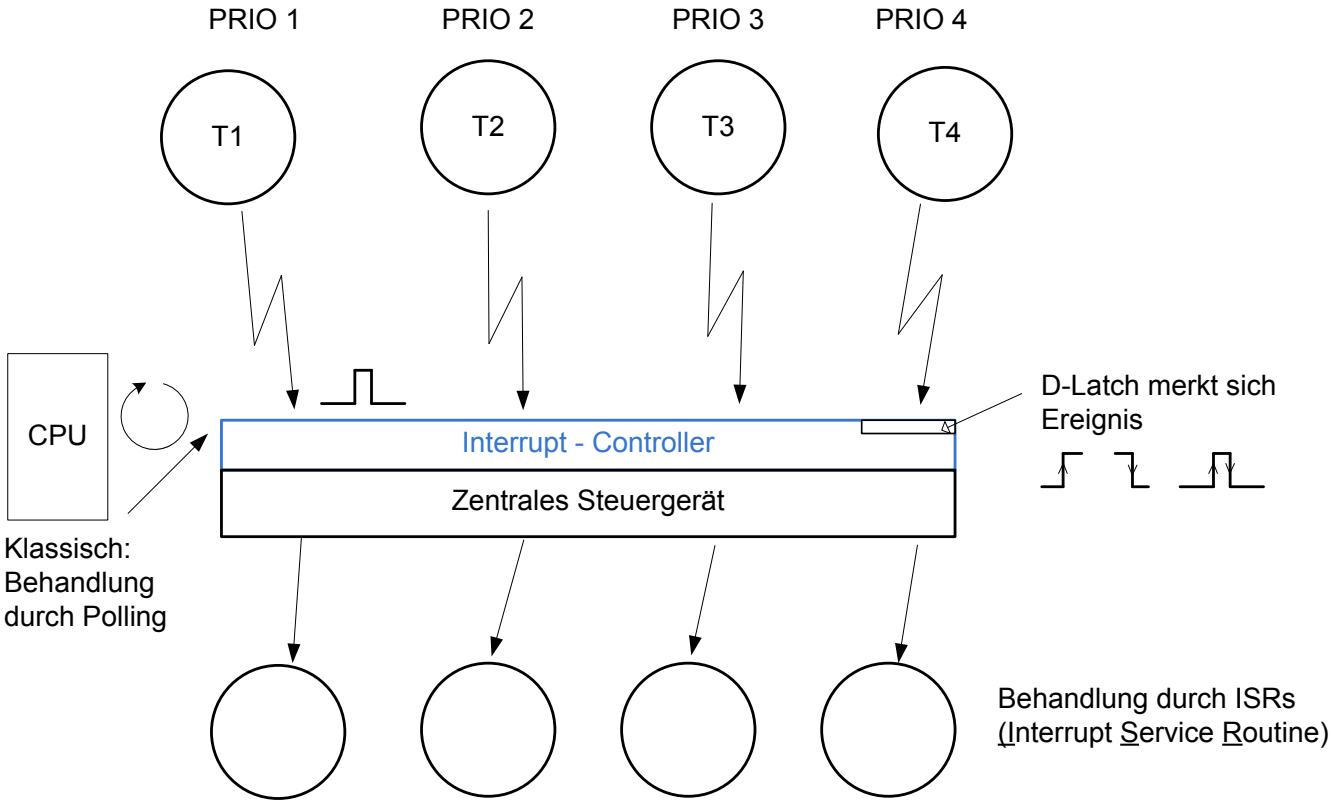


Bild: Alternative Lösung des Datenempfangs mit Hilfe einer ISR

2.2.3. Lösungsvariante 3: Ereignisgesteuerte Aufgabenbearbeitung

Prinzip: Signaländerung am Eingangsport führt zu einer Ressourcenzuweisung (Rechenzeit, Betriebsmittel etc.)



Vorteile

- gute Priorisierung der Tasks möglich
- preiswerte Ein-Prozessor-Lösung

Nachteile

- Ungünstig für passive Sensoren und für periodische Vorgänge, da zentrale Zeitbasis fehlt.
  Abhilfe: Verwendung eines Timers; Timer löst periodisch Interrupts aus, die mit den periodisch auszuführenden Aufgaben verknüpft werden können.
- Nicht alle Peripherieeinheiten sind interruptfähig.

Signaltypen in Embedded Systems

Periodische Vorgänge

- bei digitalen Regelungen
- Abtastung
- Signalerzeugung (digital)
- digitale Filterung

Spontane Ereignisse/Reaktionen

- Alarmmeldungen
- Eingaben
  - Netzwerk-Controller
  - A/D-Wandler, D/A-Wandler
  - Feldbus-Controller
  - Tastatur

Behandlung periodischer Vorgänge

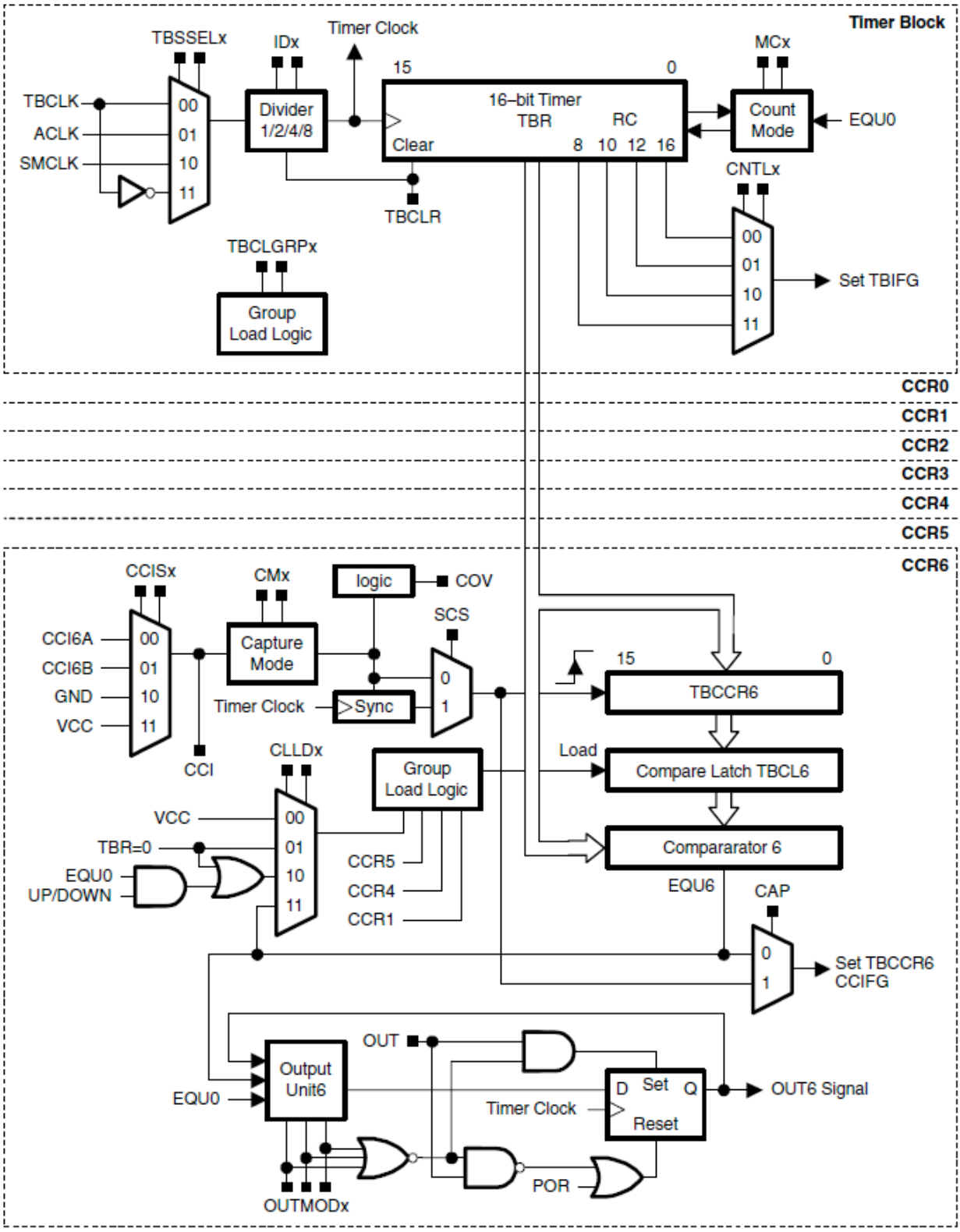
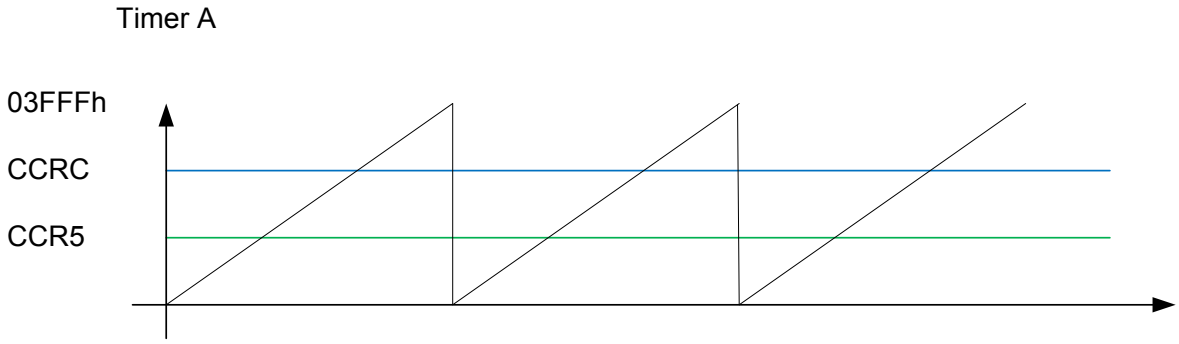


Bild: Timer_B Block Diagram

Timer A und CCR etc. Bild aus User Guide

Signalverlauf vervollständigen

Timer A im Continuous Mode



Abhilfe:

Nutzung des Timer-Überlauf-Interrupts als periodische Zeitbasis (für periodische Aufgaben)

Noch flexibler bei Verwendung der Compare-Unit in Verbindung mit dem

CCRx

CCRO

CCR2

…

Register.

2.2.4. Lösungsvariante 4: Einsatz eines Echtzeit-Betriebssystems

Prinzip:

Delegation der Ressourcenzuteilung (Rechenzeit, Kommunikationsinterfaces, u.a. Peripheriehardware) an eine spezialisierte Instanz -> Real-time Operating System (RTOS-Microkernel, 2 kBytes

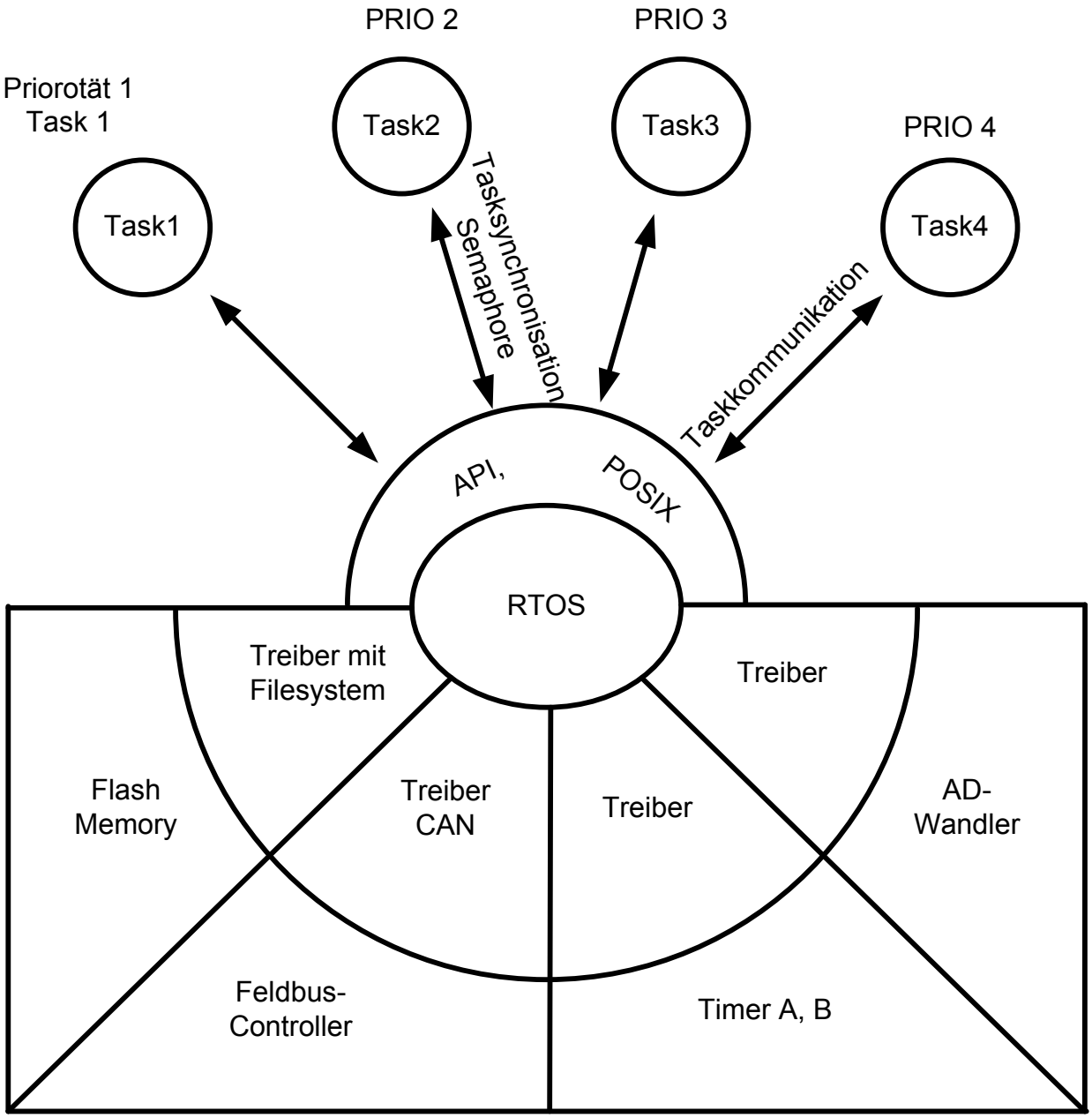


Bild: Lösungsarchitektur Einsatz eines RTOS-Kernels

Legende

HAL	Hardware Abstraction Layer
RTOS-API	Application Programming Interface
MCU	Microcontroller Unit
CPU	Central Processing Unit
Scheduler	wählt den nächsten Task zur Bearbeitung aus
RTOS-API	Stellt Services zur Synchronisation, Kommunikation zur Verfügung

Vorteile

- Periodische und asynchrone (spontane) Rechenzeitanforderungen werden

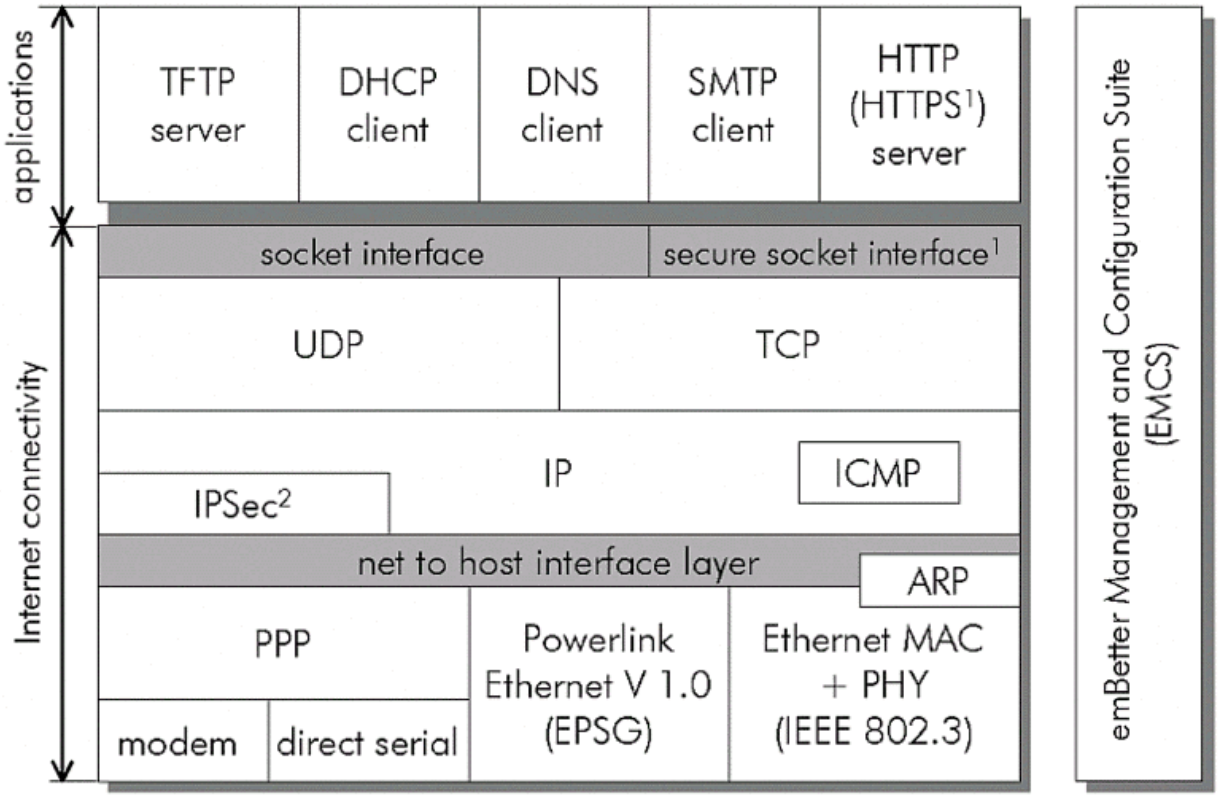
berücksichtigt.

- RTOS berücksichtigt unterschiedliche Prioritäten und Periodizität der jeweiligen Aufgaben
- hohe Transparenz durch ein standardisiertes Interface auf den RT-Kernel (RTOS-API, System Call Interface, User Services etc.)
  z. Bsp. POSIX - Standardisiertes System Call Interface
- kurze Entwicklungszeiten durch Taskorientierung -> time to market
- hohe Portabilität durch POSIX, ANSI C/C++ und Java Micoredition (JME) (Embedded Java, Realtime Java)
- Paralleles Arbeiten an verschiedenen Aufgaben ist durch die Taskorientierung möglich. Kopplung der Tasks erfolgt über ein standardisiertes Interface (= User Services); Auftragsarbeiten sind möglich.
- Synchrone und asynchrone Anforderungen werden ebenso berücksichtigt, wie die unterschiedlichen Periodizität, Prioritäten und Echtzeitanforderungen der jeweiligen Aufgaben.

Nachteile

- Lizenzkosten (pro Stück oder für den kompletten Source Code)

Embedded Mikrocontroller TCP/IP Protokollsuite zur Realisierung der Internet-kommunikation



[1] secure socket layer requires additional modules out of the TCP/IP suite

[2] under development

Bild: Embedded Mikrocontroller TCP/IP Protokollsuite am Beispiel von emBetter

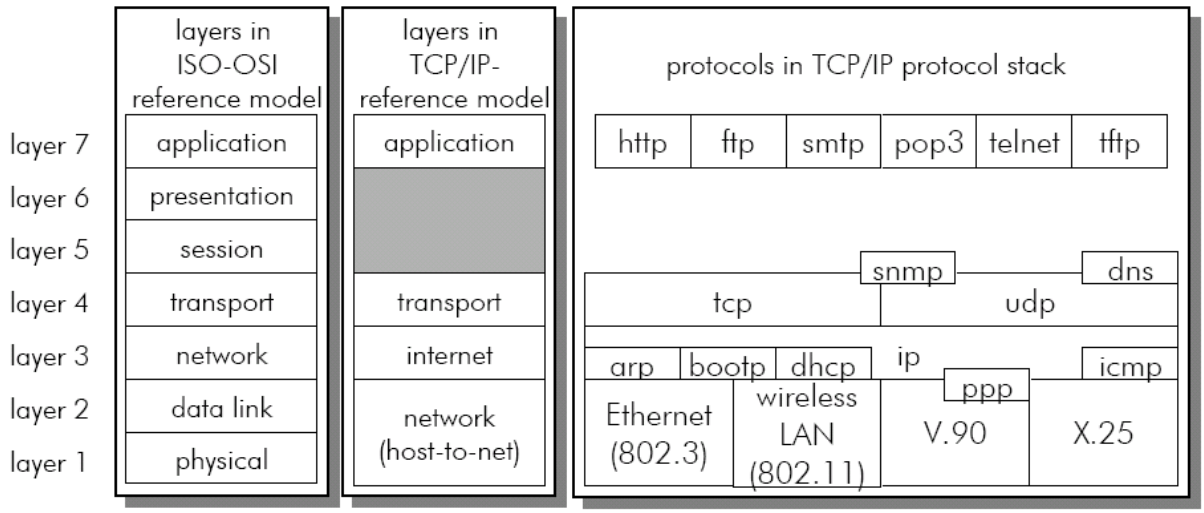


Bild: ISO-OSI reference model and TCP-IP reference model and protocols

HTTP Server		Telnet Server		SMTP Client	
CGI Scripting		TFTP Server		DNS Resolver	
TCP	UDP	ARP	DHCP	PPP	SLIP
Ethernet		Modem UART		Debug UART	

Bild: TCP-IP Stack protocols for Embedded Connectivity

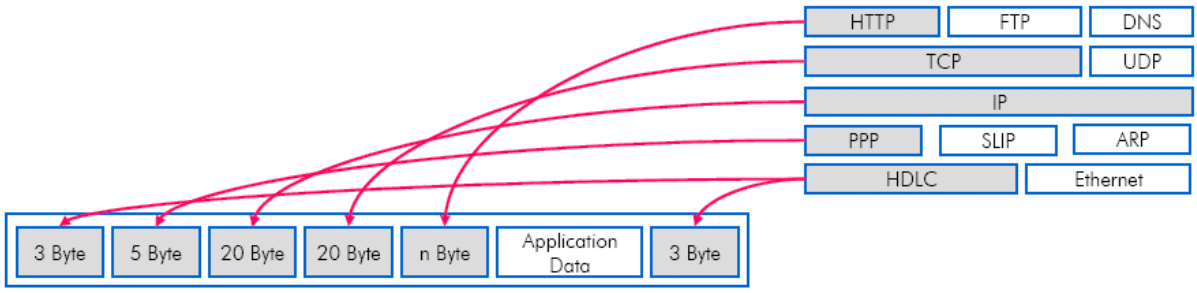


Bild: Encapsulation, i.e. HTTP packets via a serial links takes 51 Bytes

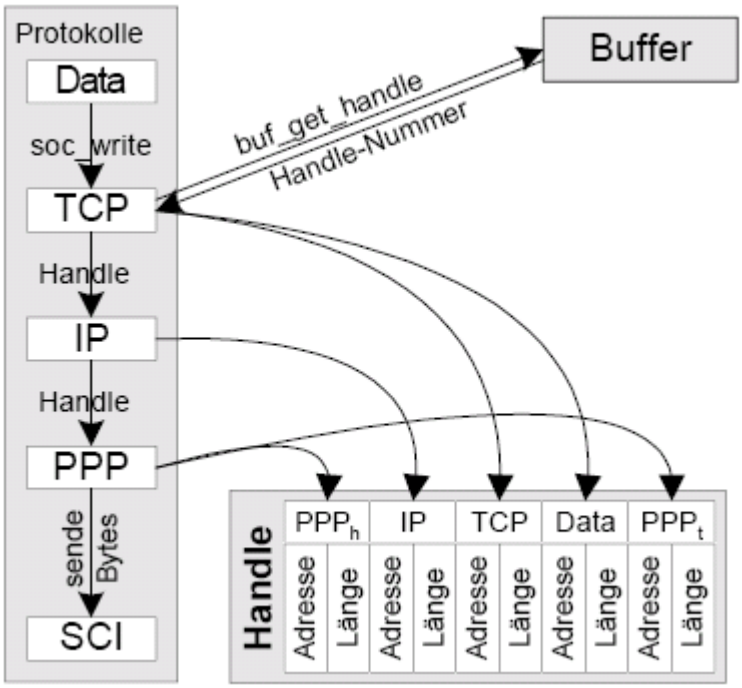
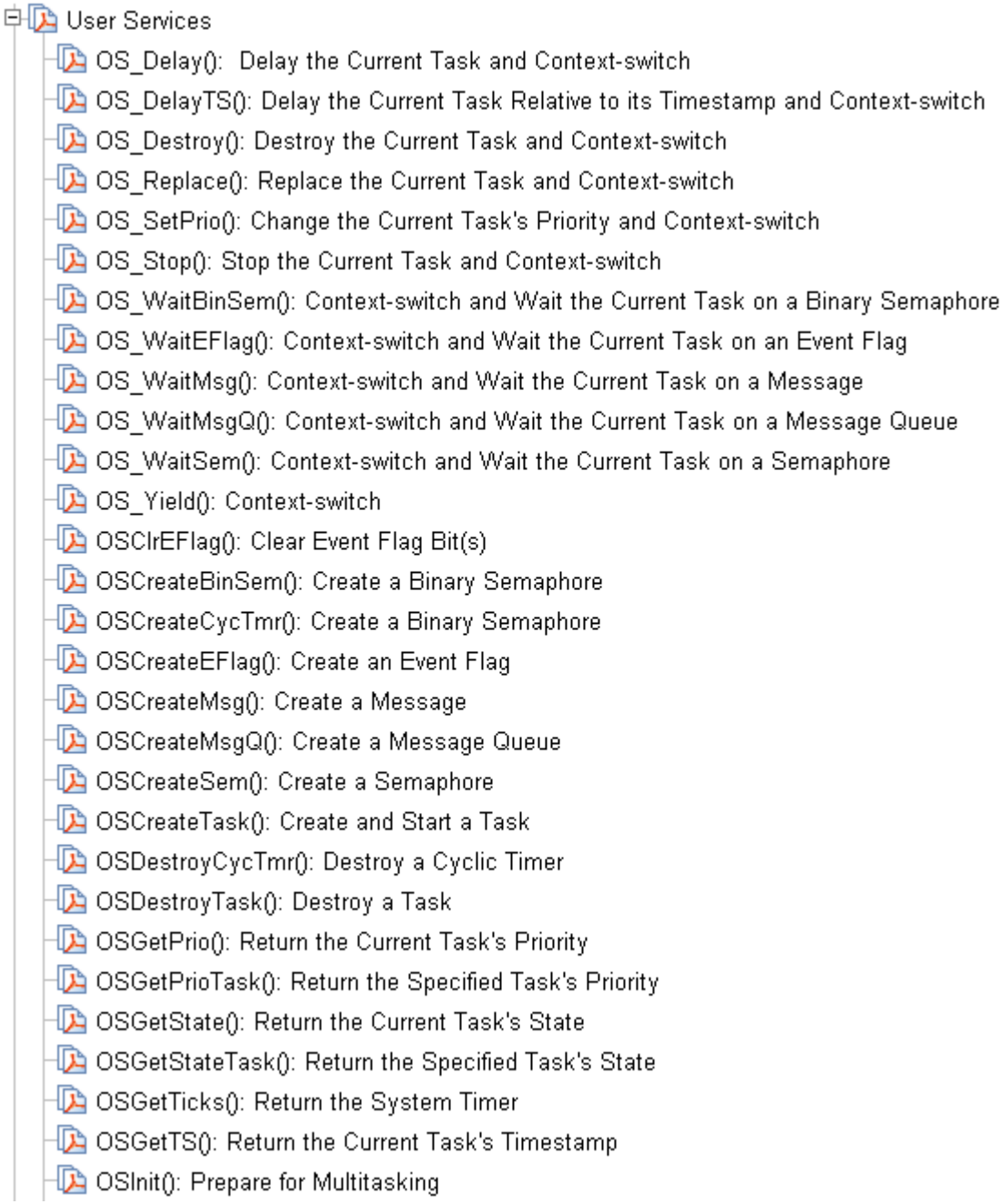


Bild: Management of output buffers

Übersicht der User Services unter Salvo.

This section describes the Salvo user services that you will use to build your multi-tasking application. Each user service description includes information on:


User Services
OS_Delay(): Delay the Current Task and Context-switch
OS_DelayTS(): Delay the Current Task Relative to its Timestamp and Context-switch
OS_Destroy(): Destroy the Current Task and Context-switch
OS_Replace(): Replace the Current Task and Context-switch
OS_SetPrio(): Change the Current Task's Priority and Context-switch
OS_Stop(): Stop the Current Task and Context-switch
OS_WaitBinSem(): Context-switch and Wait the Current Task on a Binary Semaphore
OS_WaitEFlag(): Context-switch and Wait the Current Task on an Event Flag
OS_WaitMsg(): Context-switch and Wait the Current Task on a Message
OS_WaitMsgQ(): Context-switch and Wait the Current Task on a Message Queue
OS_WaitSem(): Context-switch and Wait the Current Task on a Semaphore
OS_Yield(): Context-switch
OSClrEFlag(): Clear Event Flag Bit(s)
OSCreateBinSem(): Create a Binary Semaphore
OSCreateCycTmr(): Create a Binary Semaphore
OSCreateEFlag(): Create an Event Flag
OSCreateMsg(): Create a Message
OSCreateMsgQ(): Create a Message Queue
OSCreateSem(): Create a Semaphore
OSCreateTask(): Create and Start a Task
OSDestroyCycTmr(): Destroy a Cyclic Timer
OSDestroyTask(): Destroy a Task
OSGetPrio(): Return the Current Task's Priority
OSGetPrioTask(): Return the Specified Task's Priority
OSGetState(): Return the Current Task's State
OSGetStateTask(): Return the Specified Task's State
OSGetTicks(): Return the System Timer
OSGetTS(): Return the Current Task's Timestamp
OSInit(): Prepare for Multitasking

- OSMsgQCount(): Return Number of Messages in Message Queue
- OSMsgQEmpty(): Check for Available Space in Message Queue
- OSReadBinSem(): Obtain a Binary Semaphore Unconditionally
- OSReadEFlag(): Obtain an Event Flag Unconditionally
- OSReadMsg():Obtain a Message's Message Pointer Unconditionally
- OSReadMsgQ(): Obtain a Message Queue's Message Pointer Unconditionally
- OSReadSem(): Obtain a Semaphore Unconditionally
- OSResetCycTmr(): Reset a Cyclic Timer
- OSRpt(): Display the Status of all Tasks, Events, Queues and Counters
- OSSched(): Run the Highest-Priority Eligible Task
- OSSetCycTmrPeriod(): Set a Cyclic Timer's Period
- OSSetEFlag(): Set Event Flag Bit(s)
- OSSetPrio(): Change the Current Task's Priority
- OSSetPrioTask(): Change a Task's Priority
- OSSetTicks(): Initialize the System Timer
- OSSetTS(): Initialize the Current Task's Timestamp
- OSSignalBinSem(): Signal a Binary Semaphore
- OSSignalMsg(): Send a Message
- OSSignalMsgQ(): Send a Message via a Message Queue
- OSSignalSem(): Signal a Semaphore
- OSStartCycTmr(): Start a Cyclic Timer
- OSStartTask(): Make a Task Eligible To Run
- OSStopCycTmr(): Stop a Cyclic Timer
- OSStopTask(): Stop a Task
- OSSyncTS(): Synchronize the Current Task's Timestamp
- OSTimer(): Run the Timer
- OSTryBinSem(): Obtain a Binary Semaphore if Available
- OSTryMsg(): Obtain a Message if Available
- OSTryMsgQ(): Obtain a Message from a Message Queue if Available
- OSTrySem(): Obtain a Semaphore if Available

- Additional User Services
  - OSAnyEligibleTasks (): Check for Eligible Tasks
  - OScTcbExt0|1|2|3|4|5, OStcbExt0|1|2|3|4|5(): Return a Tcb Extension
  - OSCycTmrRunning(): Check Cyclic Timer for Running
  - OSDi(), OSEi(): Control Interrupts
  - OSProtect(), OSUnprotect(): Protect Services Against Corruption by ISR
  - OSTimedOut(): Check for Timeout
  - OSVersion(), OSVERSION: Return Version as Integer
- User Macros
  - _OSLabel(): Define Label for Context Switch
  - OSECBP(), OSEFCBP(),OSMQCBP(), OSTCBP(): Return a Control Block Pointer
- User-Defined Services
  - OSDisableIntsHook(), OSEnableIntsHook(): Interrupt-control Hooks
  - OSIdlingHook(): Idle Function Hook
  - OSSchedDispatchHook(), OSSchedEntryHook(), OSSchedReturnHook(): Scheduler Hooks

3. BEGRIFFSDEFINITIONEN UND ANFORDERUNGEN

3.1. Begriffsdefinition Embedded System

Szenario: Regelung eines Düsentriebwerkes

Begriffsdefinition Embedded System

Begriffsdefinition Technisches System

Begriffsdefinition Echtzeit

Begriffsdefinition Echtzeit-System,

Begriffsdefinition Echtzeit-Betriebssystem

3.2. Anforderungen an ein Echtzeit-Betriebssystem

- Rechtzeitigkeit
- Bereitstellung paralleler Prozesse
- Vorhersehbarkeit (Determinismus)

Rechtzeitigkeit: Rechnersystem muss mit den im technischen Prozess ablaufenden Vorgängen Schritt halten, um Zeitbedingungen erfüllen zu können.

Das heißt: auf jedes Ereignis des Prozesses muss innerhalb einer vorgegebenen Zeit spanne reagiert werden. Das Ereignis muss in dieser Zeit erfasst, die Berechnungen ausgeführt und die Reaktion zum Prozess ausgegeben sein.

Reaktionszeiten werden von dem technischen Prozess bestimmt und müssen für je den auftretenden Ereignistyp spezifiziert werden.

Gleichzeitigkeit: Rechnersystem muss die Fähigkeit besitzen, mehrere gleichzeitig ablaufende Vorgänge eines Systems zu bearbeiten.

Forderung kann durch parallel arbeitende Hardware-basierende Einheiten (Prozessoren, E/A-Einheiten, etc.) erfüllt werden.

Forderung kann auch durch die nebenläufige Abarbeitung von Rechenprozessen auf einem Prozessor erfüllt werden. Hierbei müssen die Verarbeitungszeiten im Vergleich zu den Abläufen des technischen Prozesses klein sind (quasi-parallele Verarbeitung).

Vorhersehbarkeit: Das System-Verhalten muss auch bei zufällig anfallenden Ereignissen deterministisch und vorhersagbar sein.

Bei gleichzeitigem Auftreten mehrerer Ereignisse, das zu einer Konkurrenzsituation bzgl. der Verarbeitung führt, muss das Verhalten bekannt und vorhersagbar sein. Die Bearbeitung der einzelnen Ereignisse darf nur in einer vorhersagbaren Form

verzögert werden. Die Reaktionszeiten müssen weiterhin in den Rechtzeitigkeitsgrenzen liegen.

Nicht formal nachweisbar, wenn periodische Anforderungen (einplanbar) und unbekannte spontane Anforderungen (Signale, Alarme) gemischt auftreten. Logische und zeitliche Korrektheit einzelner Routinen reicht nicht aus, da Nebenläufigkeit und zufällige Verteilung von Ereignissen die Verifkationsmöglichkeiten beschränken.

Analyse für den schlechtesten Fall (worst case), d. h. für maximal zu erwartende Signalraten, notwendig. Hierzu müssen die Programmlaufzeiten bekannt sein.

4. AUFBAUSTRUKTUR VON EMBEDDED SYSTEMEN

Hardware-Aufbaustruktur von Embedded Systemboards

On Chip-Peripherie

On Board Peripherie, Folie Komponentenaufbaustruktur auflegen

Softwarekomponenten: EBS-Kernel

4.1.1. Embedded-Systemarchitekturen

- z. B. SoC - System on Chip
- Grundsätzliche Ansatz für das Design einer generischen Aufbaustruktur
- siehe Dissertation generischer Ansatz der Hardware-Aufbaustruktur
- siehe Hardwareaufbaustrukturen im Messebuch Embedded Systems 2003/2004
- Klassifikation von Hardwaresystemen
- Designansätze:
- Rein Hardwarebasierend mittels LCA, XPGAs etc. Mikroblaze etc. (siehe Berufungsvortrag an der FH Augsburg 2004), spezialisierte Hardware (Netzwerkprozessor)
- Hardware und Softwarebasierend (Universal-Embedded Board, wie x-Board, Tiny-Boards)
- Kriterien für die Auswahl eines Embeddedboard (Kosten des Enticklungssystems, laufende Lizenzkosten, Time-to-market Entwicklungszeiten und -unterstützung, Leistungsfähigkeit (Benchmarking) etc.

4.1.2. Hauptfunktionskomponenten eines Embedded-Boards

Der Hardware-Core: Mikrocontroller
Mikrocontroller-OnChip Funktionseinheiten

- Digitale I/O

- Standard I/O
- Capture- and Compare-Einheiten
- PWM (Pulsweitenmodulierte Ausgänge)
- Analoge I/O
- AD/DA-Wandler
- RS232-asynchrone und synchrone Einheiten

Das Interrupt-System (siehe Unterlagen Dr. Jacob)
- Interruptcontroller
- Daisy-Chain
- Mechanismus Interruptleitung, Interruptvektor und Interruptserviceroutine (ISR)

Das Speichersystem
- Speicherarten -> siehe Berufungsvortrag an der FH Augsburg
- Flash-Speicher, z.B. für Images und Disklesssysteme
- EPROM
- DRAMs, SRAMs etc.

Kommunikationsadapter
- Digitale Chip-to-Chip-Bussysteme (I2C, SPI etc.)
- Ethernet-Controller (Dallas D800C40 etc.)
- Feldbus-Controller (Aufbau und Schnittstelle zum Mikrocontroller)
- Ggf. USB, Wireless etc.

Signalkonditionierung
- Bsp. Signal-Conditioning--AD7710.pdf

Bsp. Board
- 68000 Board aus dem Praktikum
- Universal-Boards (Tiny, x-Boards, Board im Automobilbereich für den Embeddedbereich etc.)
- Router-Board
- Settop-Box Board

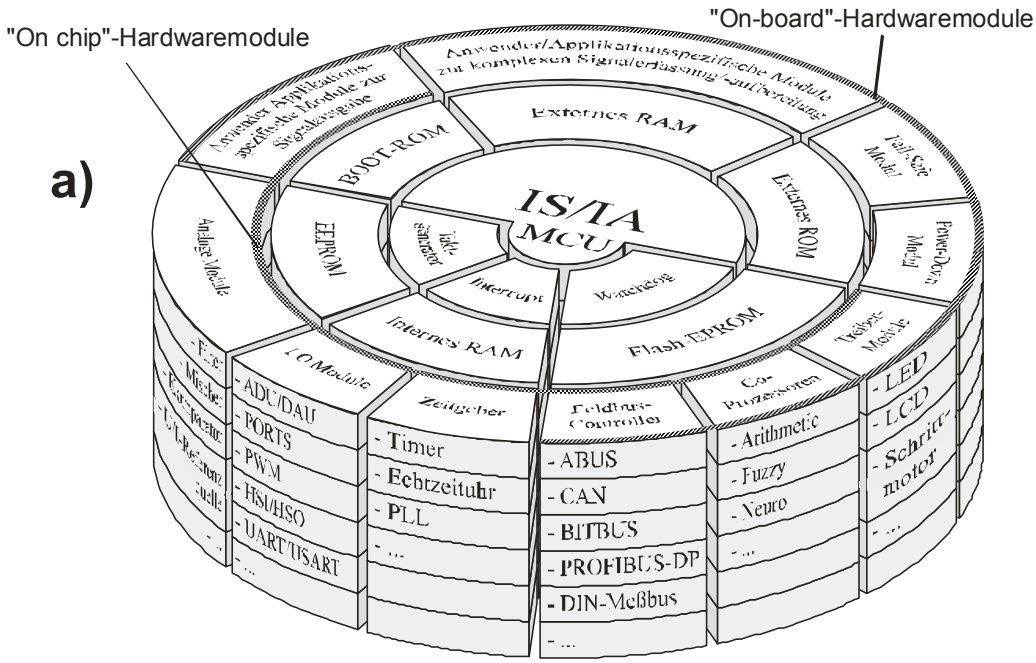


Bild: Komponentenmodell eines mikrocontrollergestützten Embedded Systems

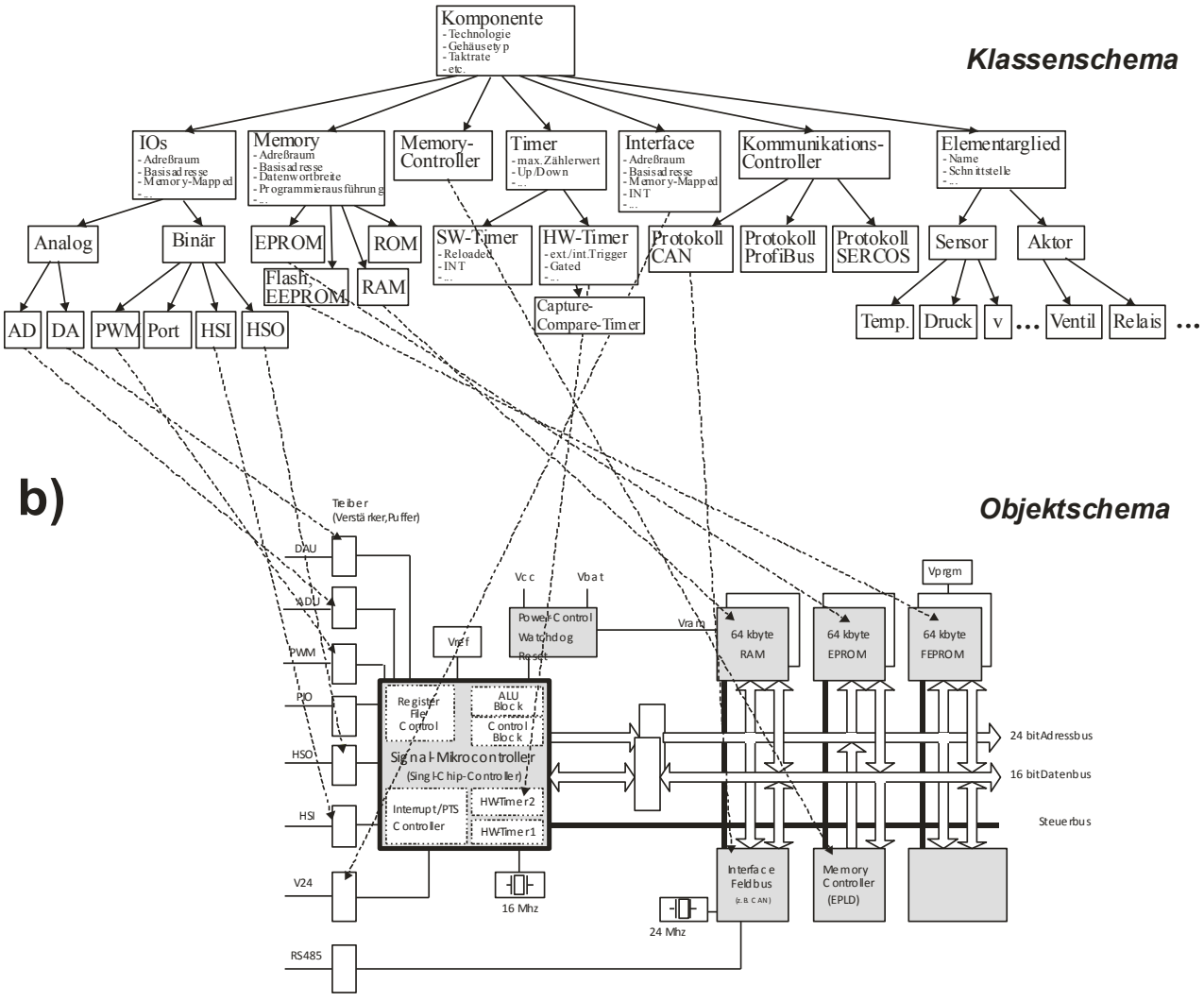
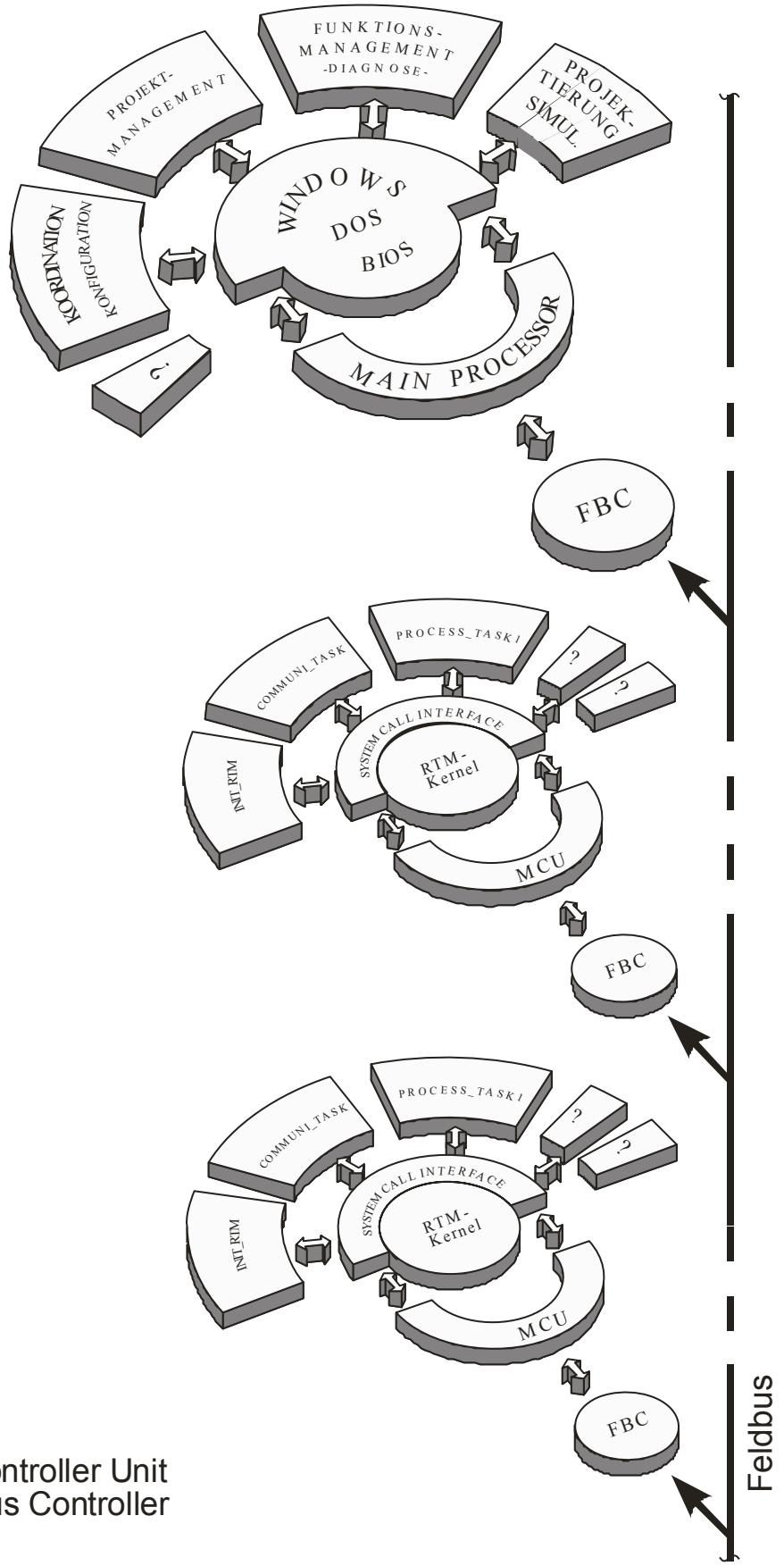


Bild: Klassenschema und Objektscheme zur topologischen Modellierung eines Em-

bedded Systems



Legende:

MCU- Mikrocontroller Unit
FBC - Fieldbus Controller

Bild: Komponenten eines verteilten intelligenten Sensor-/Aktorsystems

5. ARCHITEKTUR VON ECHTZEIT-BETRIEBSSYSTEMEN

Aufbau eines Echtzeit-Betriebssystems für Embedded Systeme

I. Begriffsdefinition

II. Anforderungen

III. Struktur und Komponenten

Anwendungsbeispiel: Düsentriebwerkes

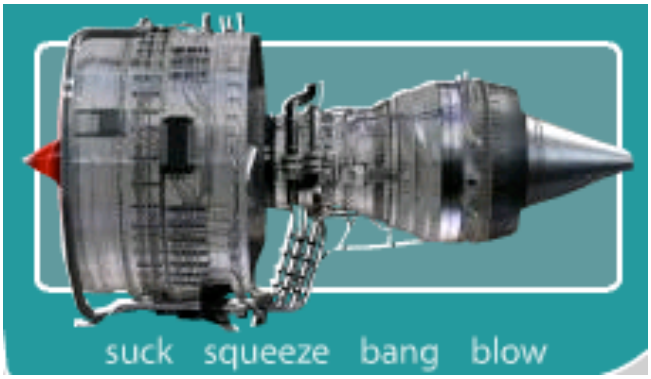
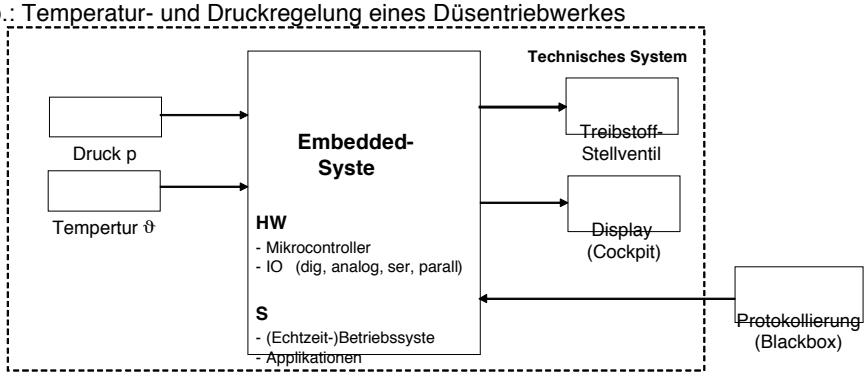


Bild: Aufbau eines Düsenstrahltriebwerkes

Funktionsweise eines Düsentriebwerkes:
siehe www.rolls-royce.com/education/schools/how_things_work/journey02/flash.html

I. Begriffsdefinition: Embedded System

Bsp.: Temperatur- und Druckregelung eines Düsentriebwerkes



Embedded System
- ist ein mit SW ausgestatteter Mikroprozessor/Mikrocontroller,
- ist ein Rechensystem, dass Teil einer größeren technischen Anordnung ist
- ist auf einen bestimmten Zweck zugeschnitten
- meist über die gesamte Lebenszeit unverändert
- Bsp.: ABS, Motorsteuerung

DIN 44300 (Begriffe der Informationsverarbeitung, 1985):

Echtzeitbetrieb ist ein Betrieb eines Rechensystems, bei dem ***Programme zur Verarbeitung anfallender Daten ständig betriebsbereit*** sind, und zwar derart, *dass die* ***Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar*** *sind*. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu vorbestimmten Zeitpunkten anfallen.

John A. Stankovic, K. Ramamritham (University of Virginia):

What is predictability for Real-Time Systems? (1990): **Real-time systems** are those systems in which *the* ***correctness of the system*** *depends not only on the* ***logical results of computations***, but also on the ***time at which the results are produced***.

Bild: Definition Echtzeitbetrieb und Real-Time Systems

=> Ein Echtzeitsystem ist ein Rechensystem, dessen Korrektheit
- nicht nur von der logischen Korrektheit der Ergebnisse sondern auch
- von der Einhaltung vorgegebener Zeitbedingungen abhängt.

Die Nichteinhaltung der vorgegebenen Zeitforderungen ist gleichbedeutend mit dem Versagen des Systems.

Definition Echtzeit (1)

Rechtzeitigkeit
innerhalb garantierter Zeiten auf Signale von außen reagieren

Gleichzeitigkeit
gewährleisten, dass die Rechenprozesse alle (quasi-) gleichzeitig vom Rechner (Prozessor) abgearbeitet werden

Bild: Definition Echtzeit

5.1. Anforderungen an ein Echtzeit-Betriebssystem

5.1.1. Rechtzeitigkeit

Die Forderung nach Rechtzeitigkeit des Rechensystems bedeutet, dass

- die Eingabedaten rechtzeitig abgerufen werden müssen und
- dass die Ausgabedaten rechtzeitig in Bezug auf die Anforderungen des betreffenden technischen Prozesses verfügbar sein müssen.

Man unterscheidet 2 Kategorien von Zeitbedingungen

- Absolutzeit-Bedingungen, z.B. um 13.30 soll ein Signal zur Abfahrt eines Zuges ausgegeben werden
- Relativ-Zeitbedingungen, z.B. Nach dem Auftreten des Streckensignals soll das Stellsignal für die Weichenstellung innerhalb von 10sec. ausgegeben werden

Definition Echtzeit (2) - , harte—versus , weiche—Echtzeit

Harte Echtzeit (hard real time):
- definierte Zeitanforderungen: $P\ (T_A \leq T_{Amax}) = 1$
- Verletzung der Zeitforderungen -> Ergebnis unbrauchbar oder sogar Schaden
- deterministisch

Weiche Echtzeit (soft real time):
- definierte Zeitforderungen, aber $P\ (T_A \leq T_{Amax}) \leq 1$
- Verletzung der Zeitforderungen für einen Teil der Ereignisse tolerierbar, Ergebnis hat nach Zeitüberschreitung noch gewissen Wert

Nicht-Echtzeit:
- kein T_{Amax} vorgegeben, nur TA minimieren
- System trifft keine Vorkehrung zur Einhaltung von Zeitschranken

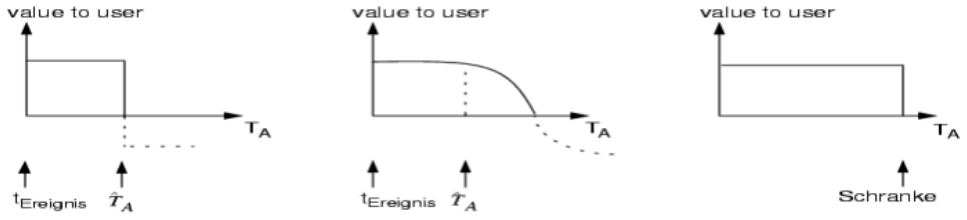


Bild: Definitionen zur Rechtzeitigkeit

4 Fälle

Fall 1: Das erfassen eines Meßwertes oder die Ausgabe eines Stellsignals hat genau zu bestimmten Zeitpunkten zu erfolgen

Fall 2: Eine Funktion hat innerhalb eines Zeitfensters zu erfolgen (t1 + t_{Delta}) (Toleranzbereich)

Fall 3: Eine Funktion hat spätestens zu einem bestimmten Zeitpunkt zu erfolgen

Fall 4: Eine Aktion darf frühestens zu einem bestimmten Zeitpunkt erfolgen

5.1.2. Definition Echtzeitbedingungen (harte u. weiche Echtzeit)

Harte Echtzeit (Hard Real-Time): Nichteinhaltung der Zeitbedingung wird als Fehler gewertet und fUhrt zu einem Versagen des Systems.

Weiche Echtzeit (Soft Real-Time): Ergebnis kann nach Ablauf der Zeitbedingung noch verwendet werden, ist allerdings mit Nachteilen verbunden (z.B. Verspätung, höhere Kosten). Zeitbedingung sollte nur selten überschritten werden.

Definition: Real-Time conditions (newsgroup.comp.realtime)

Hard Real-Time: Provides, without fail, a response to some kind of event within a specified time window (timeliness of result), The response must be predictable and independent of other operating System or application activities.

Soft Real-Time: It has reduces constraints on "Lateness", but still must operate quickly within foirly consistent time constraints. The response to the serviced events should be satisfactory, on average.

Laufzeitverzögerungen (Latenzzeiten) durch Betriebssystemaufrufe müssen von dem Betriebssyste innerste Her genannt werden. Maximale Verzögerungszeiten müssen durch den Betriebssystem-Hersteller garantiert werden und im laufenden Betrieb eingehalten werden.

Harte und weiche Echtzeit-Systeme

Gegenüber weichen Echtzeit-Systemen müssen harte Echtzeit-Systeme eine höhere „Zeit-Qualität“ erfüllen. Sie sind charakterisiert durch:

- Sie besitzen mindestens eine spezifizierte harte Echtzeitbedingung.
- Harte Echtzeibedingungen müssen unter allen Systembelastungen (auch bei Systemfehlern) eingehalten werden, ansonsten liegt ein Systemversagen vor (siehe Def. 1.6).
- Das Einhalten der Echtzeitbedingungen lässt sich trotz deterministischem Verhalten der
- Hardware, des Betriebssystems und der Applikationen nicht *vollständig* testen.
- Harte Echtzeitbedingungen haben keine Korrelation mit schneller Verarbeitung oder hoher Prozessor-Leistung.
- Harte Echtzeitbedingungen sind selten und treten in der Regel verkoppelt mit weichen Echtzeitbedingungen auf.

Bei dem Design und der Bewertung von harten Echtzeitsystemen müssen nichtdeterministische Komponenten berücksichtigt werden:

5.1.3. Gleichzeitigkeit

Die Forderung nach Gleichzeitigkeit ergibt sich aus der Tatsache, dass Echtzeit-Rechensysteme auf Vorgänge in ihrer Umwelt reagieren müssen, die gleichzeitig

ablaufen

z.B.: Schienenverkehrssystem: ein Prozessrechensystem muss auf die gleichzeitige Fahrt mehrerer Züge reagieren und entsprechend die Weichenstellung vornehmen, d..h. gleichzeitig anfallende Meßwerte müssen erfaßt und ausgewertet werden, und ggf. müssen auch gleichzeitig mehrere Stellsignale ausgegeben werden.

Oder ein Beisp. Aus dem Betrieb eines Rechners: Hier möchte man gleichzeitig ein Programm editieren, (d. h. am Bildschirm eingeben, ändern etc.) und ein gleichzeitig ein Peripheriegerät (z.B. Drucker) betätigen. Der Prozessor muss so zwischen dem EditierProzess und dem Druckprozess ständig hin und her schalten.

Wie lässt sich Gleichzeitigkeit herstellen?

am einfachsten durch einen getrennten Rechner (z.B. Hardware-Druckerboxen: Spooler). In diesem Fall arbeiten dann die Programme des so entstehenden Mehrrechner-Systems echt parallel und gleichzeitig. (NT: sehr teuer)

Die Forderung nach Gleichzeitigkeit kann jedoch auch mit einem einzigen Computer näherungsweise erfüllt werden, wenn man voraussetzt, dass die Vorgänge in der Prozess-Umwelt langsam ablaufen gegenüber der Programmabarbeitung im Computer. Durch schnelles und intelligentes Umschalten in kurzen Zeitabständen gelingt es so gleichzeitig ablaufende Vorgänge in der „Umwelt“ gleichzeitig zu bedienenen.

Durch Verwendung eines Schedulers.

II. Anforderungen an ein Echtzeit-Betriebssystem

- **Gleichzeitigkeit: Bereitstellung paralleler Prozesse**
  Alle Rechenprozesse werden (quasi-)gleichzeitig vom Prozessor abgearbeitet.
- **Betriebsmitteleinplanung mit Echtzeitverfahren (Kernproblem: Prozessor,..)**
- **Interaktion der Prozesse (Kommunikation, Synchronisation)**
- **Rechtzeitigkeit: Reaktion innerhalb garantierter Zeiten auf Signale von außen**
  z.B. kurze Reaktionszeiten auf einen Interrupt.
- **Prozesse müssen zugunsten wichtigerer Prozesse unterbrechbar sein**
- **Ein-/Ausgabeabwicklung mit determiniertem Zeitverhalten**
- **Zeitfunktionen, Uhrensynchronisation**

Bild: Anforderungen an ein Echtzeit-Betriebssystem

5.2. Entwurf eines Minimal-Betriebssystems

5.2.1. Vereinfachende Annahmen

Entwicklung eines Minimal-EBS

Vereinfachende Annahmen:

(1) Die Summe aller Abarbeitungszeiten der Rechenprozesse sei kleiner als die Taktzeit T

=> Damit ist gewährleistet, dass beim Eintreffen des nächsten Taktimpulses alle Rechenprozesse beendet sind

(2) Es werden keine Tasks verwaltet, die durch einen Interrupt angestoßen werden

=> d.h. Konzentration zunächst nur auf zyklisch auszuführende Rechenprozesse, die per Timer gestartet werden sollen

(3) Die Ausführungszeiten für die Ein- und Ausgabegeräte sollen zunächst vernachlässigt werden

=> somit ist keine Betriebsmittelverwaltung für die im allgemeinen deutlich langsamere Ein- und Ausgabeperipherie erforderlich.

Bild: Vereinfachende Annahmen für die Entwicklung eines Minimal-EBS

Für die Entwicklung eines Minimal-EBS sollen folgende Vereinfachungen zugrunde gelegt werden:

(1) Die Summe aller Abarbeitungszeiten der Rechenprozesse sei kleiner als die Taktzeit T, d. h. die Rechenzeit einer Task ist klein gegenüber dem zyklischen Zeittakt; d.h. die Summe der Abarbeitungszeiten von z. B. 10 Tasks muss kleiner sein als die Zykluszeit sein.
Damit ist gewährleistet, dass beim Eintreffen des nächsten Taktimpulses alle Rechenprozesse beendet sind

(2) Es werden keine Tasks verwaltet, die durch einen Interrupt angestoßen werden, d. h. wir konzentrieren uns zunächst nur auf zyklisch auszuführende Rechenprozesse, die per Timer gestartet werden sollen.

(3) Die Ausführungszeiten für die Ein- und Ausgabegeräte sollen zunächst vernachlässigt werden
-> somit ist keine Betriebsmittelverwaltung für die im allgemeinen deutlich langsamere Ein- und Ausgabeperipherie erforderlich, und Ein-/Ausgabeoperationen können hinsichtlich der Abarbeitung wie andere Operationen betrachtet werden.

Bei der Entwicklung des erweiterten EBS wírd nun schrittweise auf die vorgenannten Vereinfachungen verzichtet.

5.2.2. Komponenten des Minimal-Echtzeitbetriebssystems

Aufgaben der „Zeitverwaltung“

Aufgaben der „Taskverwaltung“

Aufgaben der „Prozessorverwaltung

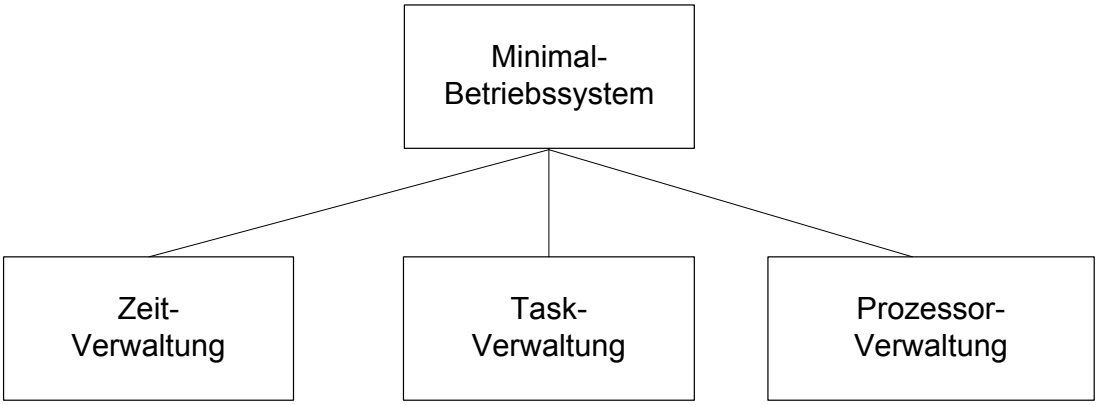


Bild: Komponenten des Minimal-Echtzeitbetriebssystems

Bild: Arbeitsweise eines Echtzeitbetriebssystem; Bsp.: Regelung eines Düsentriebwerks

5.2.3. Aufgaben der Zeitverwaltung

Bildung der unterschiedlichen Zykluszeiten

Zeitverwaltung wird zyklisch von dem Uhr-Impulstakt angestoßen

Zeitverwaltung ermittelt, wann welcher Rechenprozess ablaufen soll (Sollzeitpunkte) und teilt dies der Taskverwaltung mit -> Taskverwaltung::Aktivierung.

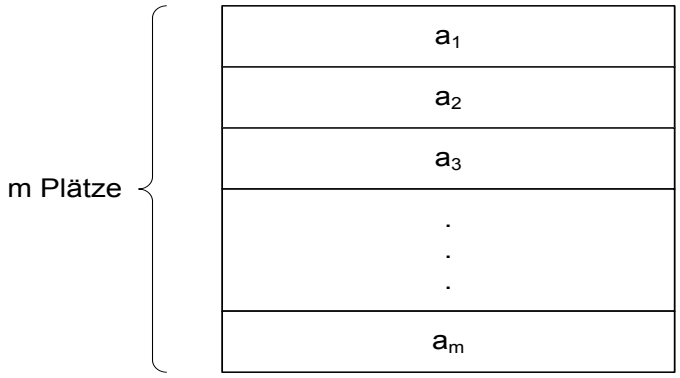


Bild: Bereitstellung der Zykluszeit-Faktoren a_i in einer Liste ZYKLUS

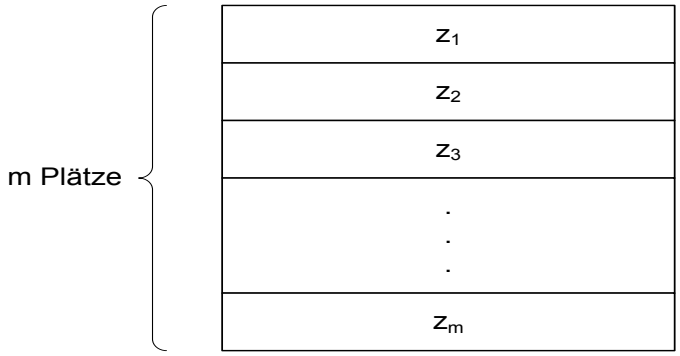


Bild: Ablage der Zeitdauervariablen z_i in einer Liste ZEITZAEHLER

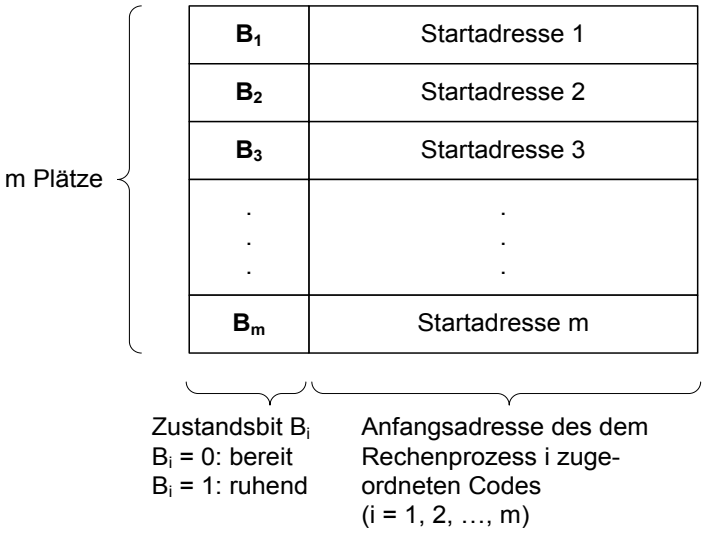


Bild: Listenstruktur der Taskverwaltung

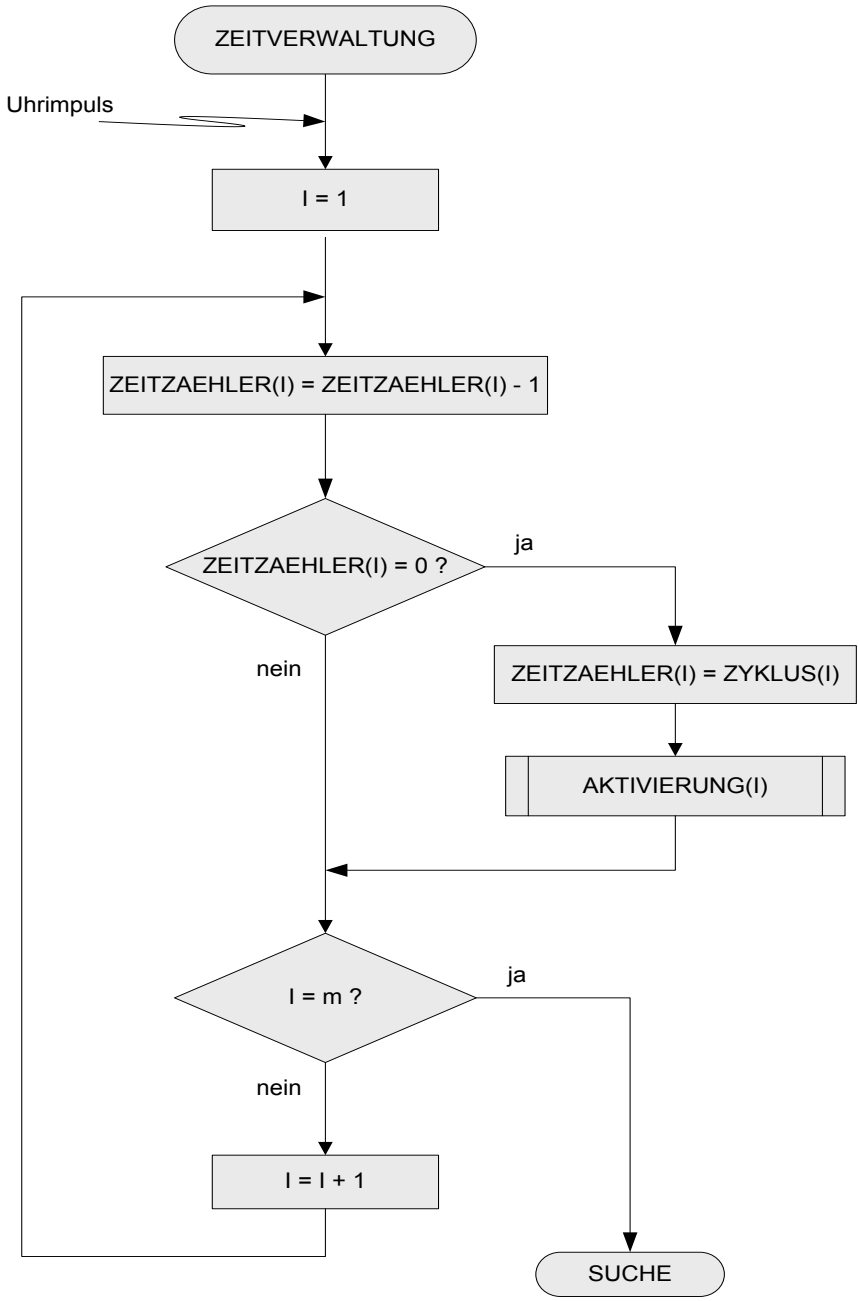


Bild: Ablaufdiagramm der Zeitverwaltung

5.2.4. Aufgaben der Taskverwaltung

Programmodule (Funktionen)

- Aktivierung
- Suche
- Deaktivierung
- Verwaltung der Rechenprozesse

Ablaufbereite Rechenprozesse werden von der Taskverwaltung an die Prozessorverwaltung gemeldet, d. h. Taskverwaltung::Suche -> Prozessorverwaltung::Dispatcher

Nach Beendigung der jeweiligen Tasks setzt die Taskverwaltung::Deaktivierung für

den Task den Zustand "ruhend" (nach Meldung durch die Prozessorverwaltung).

Taskverwaltung::Aktivierung wird von der Zeitverwaltung angestoßen und setzt das Zustandsbit der i-ten Task auf "bereit" = 1.

Nach jedem Uhrzeittakt bzw. bereits früher nach Aufruf der Taskverwaltung::Deaktivierung wird jeweils die Funktion Taskverwaltung::Suche gestartet.
Taskverwaltung::Suche sucht die Verwaltungsliste nach bereiten Tasks (Zustandsbit = 1) ab und meldet diese im Erfolgsfall an die Prozessorverwaltung. (IST-Zeitpunkte)

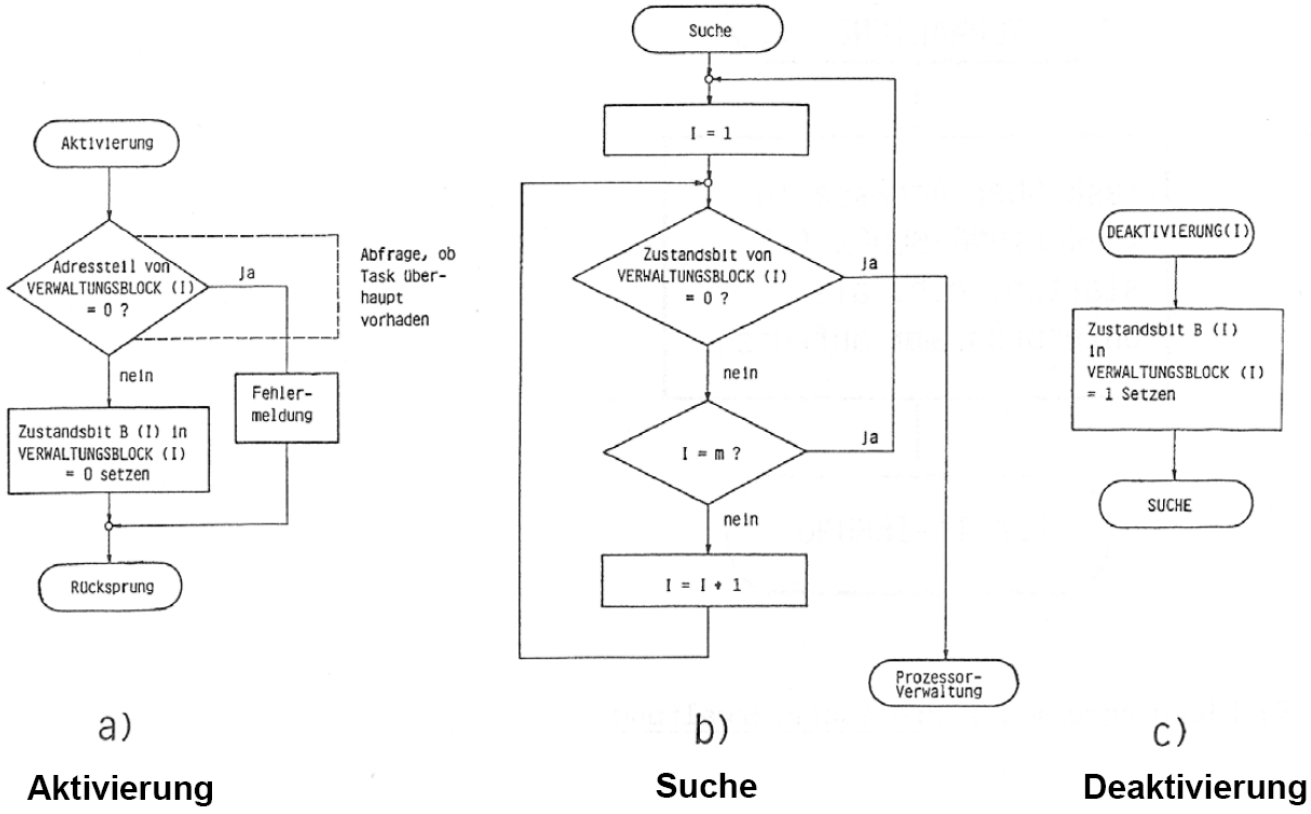


Bild: Ablaufdiagramme der Funktionen der Taskverwaltung: Aktivierung, Suche, Deaktivierung

5.2.5. Strategien zur Prozess-Koordination (Scheduling)

Definition: Scheduling

> Strategien zur Koordination und zur Festlegung des Ablaufes des nächsten Rechenprozesses oder Interruptroutine werden Scheduling genannt.

Ein Scheduling-Algorithmus besteht aus Regeln, die bestimmen, welcher Task zu einer bestimmten Zeit ausgeführt wird. Ein **durchführbares Scheduling** muss eine Menge von Prozessen, $P = P_1, P_2, ..., P_n$, so anordnen, dass alle Prozesse ihre Zeitschranken einhalten: $P' = P_{i1}, P_{i2}, ..., P_{in}$

Ein **stabiler Scheduling-Algorithmus** kann bei einer kurzzeitigen Prozessorüberlast weiterhin eine Teilmenge $P' = P_{i1}, P_{i2}, ..., P_{im}$ $(m < n)$ aller Prozesse *durchführbar*

koordinieren.

Zu der Teilmenge P' sollten die wichtigsten Prozesse gehören. Prozesse, die zu P' gehören, dürfen nicht durch Prozesse blockiert werden, die nicht P' angehören. Ein optimaler Scheduling-Algorithmus kann zu einer Prozessmenge P= P_1, P_2, ..., P_n eine *durchführbare* finden, wenn sie exisitiert.

Scheduling kann auf zwei grundsätzlich verschiedenen Arten vorgenommen werden:

Statisches Scheduling (Zeit-gesteuert), d.h. die Planung des zeitlichen Ablaufes der Tasks vor ihrer Ausführung betreffend.

Vorab Festlegung der Task-Abarbeitung in einem festen Schema, dass als eine optimale Reihenfolge festgelegt werden sollte.

Voraussetzungen, um statisches Scheduling durchführen zu können:

- Alle Tasks müssen bekannt sein, eine dynamische Erzeugung von Tasks während der Laufzeit ist nicht möglich.
- Zeitpunkte (Start, Deadline) der Abläufe der Tasks müssen bekannt sein.
- Sporadisch auftretende Ereignisse (z.B. Fehlermeldungen) ergeben Einplanungsprobleme. Mögliche Lösung: Verwendung von festen eingeplanten Zeitscheiben, die ungenutzt verstreichen, wenn die Ereignisse nicht eintreten.
- Besonders geeignet für harte Echtzeit sowie sicherheitskritische Anwendungen, z.B. Flugzeugsteuerungen, etc.. Formale Verifikation der Zeitbedingungen möglich.

Dynamisches Scheduling (Ereignis-gesteuert), d.h. die Organisation des zeitlichen Ablaufes der Tasks während der Ausführung der Programme betreffend.

Der Scheduler wird zu bestimmten Zeitpunkten aktiviert:

- In festen Zeitintervallen (zyklisch),
- Bei Auftritt von Ereignissen (Interrupt, Messages, etc.),
- Nach einer Tasksynchronisation, -kommunikation, -erzeugung oder -ende, d.h. immer nach der Abarbeitung eines OS-Kommandos.

Algorithmus benötigt Prozessorzeit. Taskwechsel von der Anwendung in das Betriebssystem ist in der Regel notwendig.

Anforderungen an Scheduling-Algorithmen in Echtzeitsystemen:

- Vorhersagbare Antwortzeiten, zumindest für zeitkritische Tasks.
- Alle zeitkritischen Tasks sollten so koordiniert werden, dass sie ihre Zeitschranken einhalten, falls überhaupt möglich.
- Stabiles Verhalten während des normalen Betriebes und bei Überlast (transient overload).

- Falls die Echtzeitbedingungen nicht für alle Prozesse eingehalten werden können, so zumindest für eine Teilmenge der Prozesse.
- Effiziente Ausführung der Task-Koordination. Algorithmus sollte möglichst wenig Rechenzeit benötigen.

Die Auswahl von Scheduling-Algorithmen ist in der Praxis nicht einfach, da In der Regel nicht alle Informationen für die Planung zur Verfügung stehen. Optimale Algorithmen zur Bestimmung des am besten geeigneten nächsten Tasks zu komplex sind und damit zu viel Rechenzeit beanspruchen würden.

Auswahl eines bestimmten Scheduling-Algorithmus hängt von verschiedenen Faktoren ab:

- Taskerzeugung während der Initialisierung oder auch dynamische Taskerzeugung während der Laufzeit.
- Prioritäten und Laufzeitverhalten (bekannt/unbekannt) wichtiger Tasks.
- Überwiegend periodisches oder stochastisches Taskverhalten.
- Harte Zeitschranken periodischer oder stochastischer Tasks.
- Unterbrechbarkeit (preemptiori) der Tasks.
- Logische Abhängigkeiten von Tasks untereinander.
- Scheduling von Ein- oder Mehrprozessorsystemen.

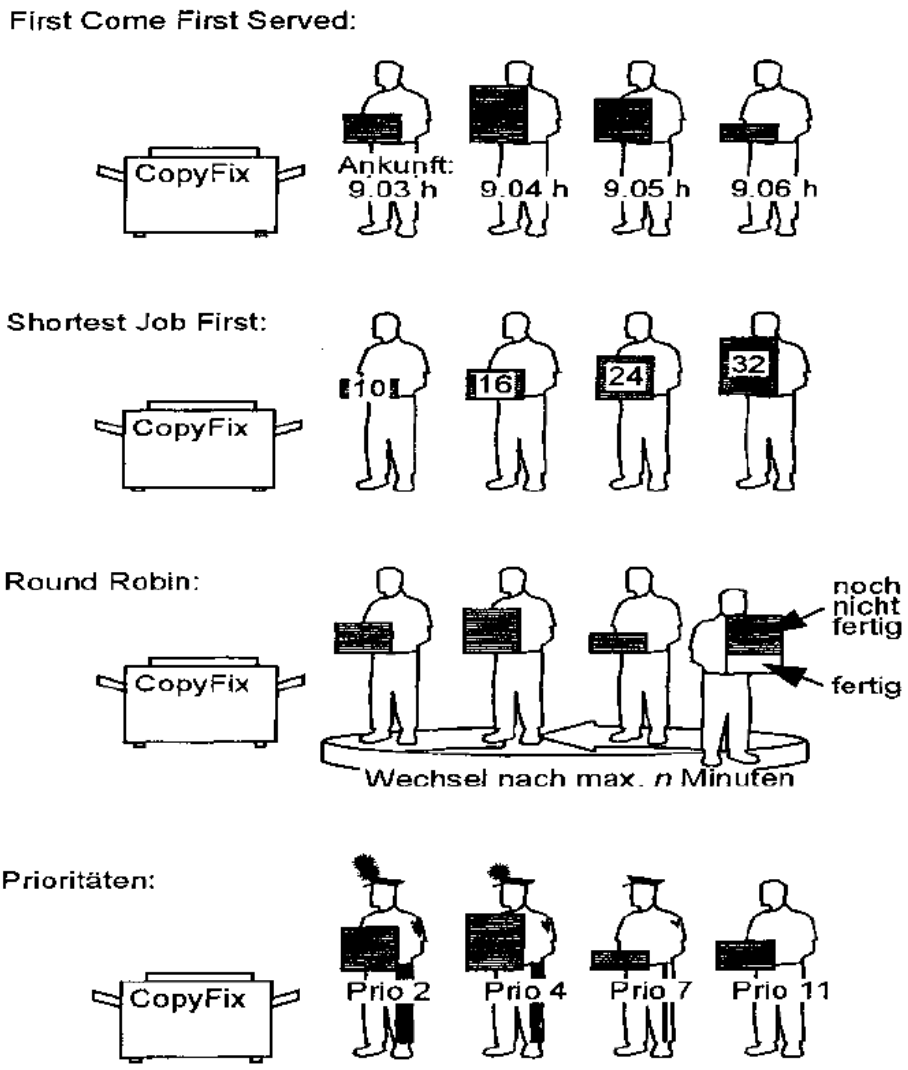


Bild: Schedulingstrategien am Beispiel einer Warteschlange vor einem Kopiergerät; aus „Betriebsysteme“, Spektrum Verlag

5.2.6. Scheduling Algorithm

Zur Realisierung von Echtzeitsystemen existieren verschiedene Möglichkeiten der Taskzuteilung (Scheduling). Man unterscheidet statische (zeitgesteuerte), dynamische (ereignisgesteuerte) und gemischte Verfahren.

Beim statischen Scheduling liegt bereits vor Systemstart ein vollständig berechneter Zeitplan vor. Innerhalb eines festen Betriebssystemzyklus werden den Rechenprozessen genau vordefinierte Zeitscheiben zugeteilt.

Bei der dynamischen Variante wird die zu bearbeitende Task während der Laufzeit bestimmt. Dadurch kann der Prozessor auch auf seltene, asynchrone Ereignisse reagieren.

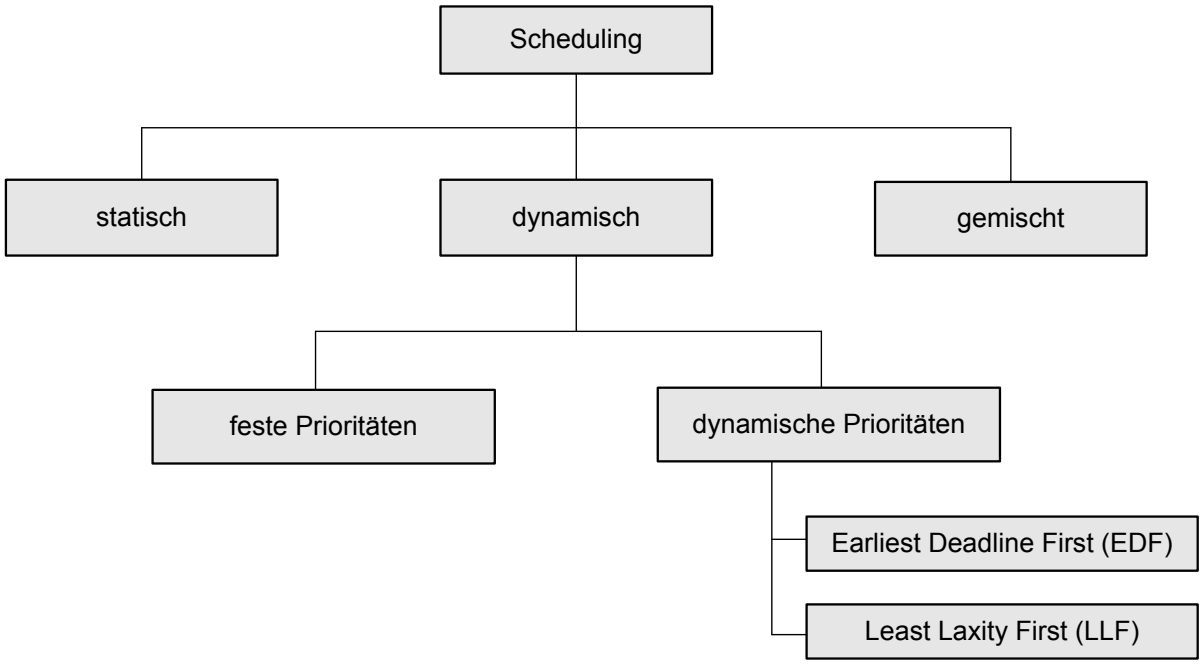


Bild: Klassifizierung der Schedulingverfahren

Dynamische Schedulingmethoden werden wiederum in zwei Varianten unterteilt. Wird einer Task immer die gleiche Priorität zugeordnet, spricht man von Scheduling mit festen Prioritäten, ändert sich die Priorität einer Task von Aufruf zu Aufruf, findet eine Zuordnung mit dynamischen Prioritäten statt. Beispiele für dynamisches Scheduling mit ebenfalls dynamischen Prioritäten sind der Least Laxity First (LLF)- und Earliest Deadline First (EDF) Algorithmus.

Wie der Name bereits andeutet, stellt das gemischte Verfahren eine Kombination aus statischem und dynamischem Scheduling dar.

Wichtig für die Multitaskingumgebung ist die Taskzuteilung. Die beiden häufigsten Scheduler Verfahren sind

- Preemptive priority based Scheduling: In erster Linie erfolgt die Taskzuteilung mit dem Preemptive Priority Verfahren. Dabei wird jeder Task bei ihrer Erzeugung eine Priorität zugeteilt. Sobald eine Task in den Zustand ready kommt, wird geprüft, ob diese Task eine höhere Priorität aufweist als die momentan laufende Task. Falls ja, wird der Kontext der laufenden Task gesichert und die Task mit der höheren Priorität erhält die CPU.

  Anschließend wird die unterbrochene Task an der Unterbrechungsstelle fortgesetzt. Entsprechend muss die Task bei einer niederen Priorität auf die CPU-Zuteilung durch das RTOS warten bis die laufende Task terminiert. In VxWorks existieren bspw. 256 Prioritätsniveaus, 0 hat die höchste und 255 die niedrigste Priorität.

  Die Priorität kann zur Laufzeit mit taskPrioritySet() geändert werden.

  Unterbrechungsanforderungen haben immer die höchste Priorität, so dass prinzipiell jede Task zu jeder Zeit unterbrochen werden kann.

- Round Robin Scheduling: Ergänzt wird das Preemptive Priority Verfahren durch das Round Robin Verfahren. Dieses Verfahren wird für den Fall benutzt, dass mehrere Tasks mit gleicher Priorität gleichzeitig existieren.

  Beim Round Robin Verfahren wird die Rechenzeit unter gleichpriorisierten Tasks gerecht verteilt. Das Round Robin Verfahren funktioniert in der Weise, dass die Zeit in gleichlange Zeitscheiben aufgeteilt wird und jede Task bekommt eine Zeitscheibe zugewiesen (time slicing). Solange keine anderen Tasks mit höheren Prioritäten auftreten, läuft die Task innerhalb dieser Zeitscheibe ab. Nach dieser Zeitscheibe bekommt eine andere gleichberechtigte Task die CPU zugeteilt.

  Time Slicing kann mit der Kernel-API Funktion kernelTimeSlice() aktiviert werden.

- Deadline Scheduling Strategies
  - Earliest Deadline First (EDF) Scheduling

    Der EDF-Algorithmus beruht darauf, die Task, deren Deadline am frühesten abläuft, die höchste Priorität zuzuordnen, somit diese dann als erste abgearbeitet wird. Taskwechsel können nur durch Verdrängung der aktuell zu bearbeitenden Task oder nach Fertigstellung derselben stattfinden.

    Der häufige Einsatz des EDF-Verfahres liegt in der Möglichkeit begründet, sämtliche Deadlines einzuhalten, vorausgesetzt, die Prozessorauslastung ist nicht größerer als 100%. Für Einprozessorsysteme ist EDF der optimale Algorithmus.

  - Least Laxity First (LLF)

    Dieses Verfahren ähnelt dem EDF-Verfahren, allerdings wird für die Auswahl der nächsten auszuführenden Task nicht nur deren Deadline, sondern auch die zugehörige Rechenzeit betrachtet.

    Unter Laxity versteht man die Zeit, die einer Task bis zur nächsten Deadline übrigbleiben würde, wenn sie ab sofort bis zur Beendigung des Aufrufs den gesamten Prozessor zur Verfügung hätte. Die höchste Priorität bekommt die Task mit der geringsten Laxity.

    LLF ist theoretisch besser als EDF, da früher erkannt wird, wenn eine Deadline nicht mehr eingehalten werden kann. Das Verfahren benötigt jedoch mehr Rechenaufwand und die Laufzeiten der Tasks als zusätzliche Eingabewerte.

  - Maximum Urgency First (MUF) Algorithm

    Das LLF-Scheduling eignet sich hervorragend für zeitkritische Tasks, ist aber für weniger zeitempfindliche Aufgaben ein Overkill. Deshalb gibt es eine dritte Variante des Deadline-Scheduling, das so genannte Maximum Urgency First (MUF)-Scheduling, welches das LLF-Scheduling mit dem traditionellen prioritätsbasierten preemptiven Scheduling kombiniert. Beim MUF-Scheduling werden zeitkritische Tasks mit hoher Priorität mit dem LLF-Deadline-Scheduling Verfahren abgearbeitet, während im selben Scheduler die anderen Tasks mit niedrigerer Priorität mit dem prioritätsbasierten preemptiven Ablauf abgearbeitet werden.

- Partition-Scheduler

  Das Deadline-Scheduling Verfahren sind gut geeignet, um die Abläufe von zeitkritischen Tasks zu verbessern, aber sie lösen nicht das Problem, dass Tasks „verhungern“ können. Schutz gegen die Nichtausführung von Tasks bieten Partition-Scheduler.

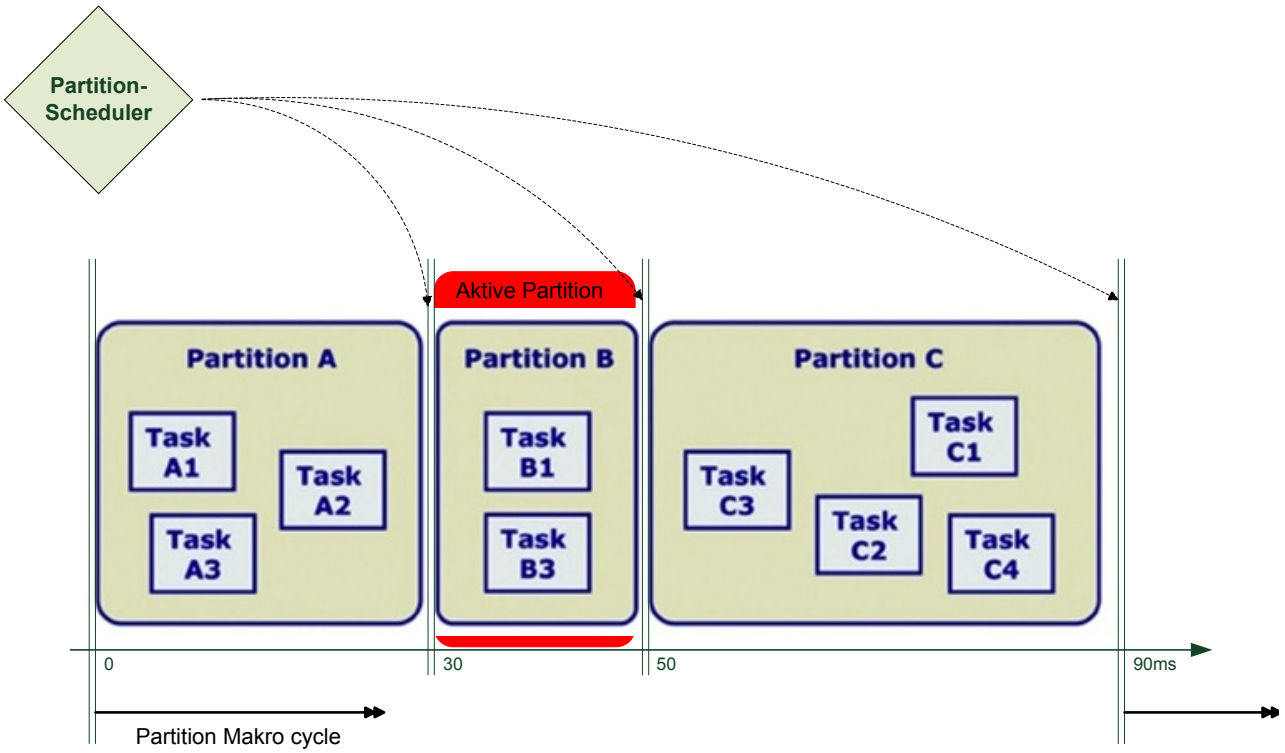


Bild: Partition-Scheduler

Hier werden Tasks in Gruppen zusammengefasst, die als Partitionen bezeichnet werden. Typischerweise besteht eine Partition aus mehreren Tasks, die eng zusammen arbeiten. Bei dem Partition-Scheduler werden jeder Partition ein oder mehrere Zeitfenster für die Ausführung in einer sich wiederholenden Zeitlinie (Makro cycle) zugeordnet. Innerhalb jedes Zeitfensters können nur die Tasks der derzeit aktiven Partition ablaufen. Die Tasks in der aktiven Partition werden auf konventionelle Weise abgearbeitet, d.h. preemptiv prioritätsorniertiert mit Round-Roubin für gleichpriorisierte Tasks.

Ist das Zeitfenster der Partition einer bestimmten Applikation aktiv, hat diese Task-Gruppe garantierten Zugriff auf die CPU des Prozessors. Ein „Verhungern“ beim Zugriff auf den Prozessor wird innerhalb der Task-Gruppen vermieden.

Common scheduling disciplines
http://en.wikipedia.org/wiki/Scheduling_%28computing%29#Common_scheduling_disciplines

The following is a list of common scheduling practices and disciplines:

- Borrowed-Virtual-Time Scheduling (BVT)
- Completely Fair Scheduler (CFS)
- Critical Path Method of Scheduling
- Deadline-monotonic scheduling (DMS)
- Deficit round robin (DRR)
- Dominant Sequence Clustering (DSC)
- Earliest deadline first scheduling (EDF)
- Elastic Round Robin
- Fair-share scheduling
- First In, First Out (FIFO), also known as First Come First Served (FCFS)
- Gang scheduling
- Genetic Anticipatory
- Highest response ratio next (HRRN)
- Interval scheduling
- Last In, First Out (LIFO)
- Job Shop Scheduling (see Job shops)
- Least-connection scheduling
- Least slack time scheduling (LST)
- List scheduling
- Lottery Scheduling
- Multilevel queue
- Multilevel Feedback Queue
- Never queue scheduling
- Proportional Share Scheduling
- Rate-monotonic scheduling (RMS)
- Round-robin scheduling (RR)
- Shortest expected delay scheduling
- Shortest job next (SJN)
- Shortest remaining time (SRT)
- Staircase Deadline scheduler (SD)
- "Take" Scheduling
- Two-level scheduling
- Weighted fair queuing (WFQ)
- Weighted least-connection scheduling
- Weighted round robin (WRR)
- Group Ratio Round-Robin

- Atropos is a real-time scheduling algorithm developed at Cambridge University. It combines the Earliest Deadline First algorithm with a best effort scheduler to make use of slack time, while exercising strict admission control.

Siehe auch http://en.wikipedia.org/wiki/Scheduling_algorithm

5.2.7. Aufgaben der Prozessorverwaltung

Funktionen

Zuteilung des Betriebsmittels Prozessor (hier Ein-Prozessorsystem); für Mehrprozessorsysteme (=Transputersysteme) -> Verwendung spez. Programmiersprache wie z. B. OCCAM

Taskumschaltung (Context-Switch) mit Start des Rechenprozesses

Informiert Taskverwaltung::Deaktivierung, um beendeten Task in den Zustand "ruhend" zu überführen.

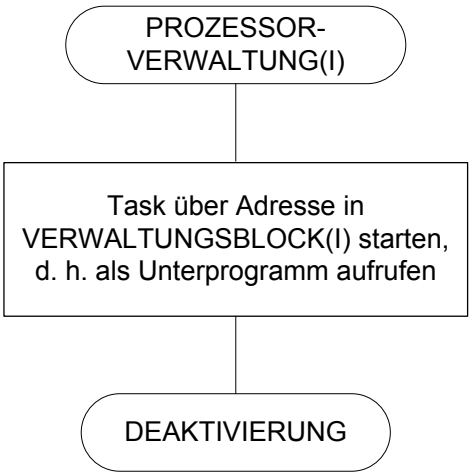


Bild: Ablaufdiagramm der Prozessorverwaltung

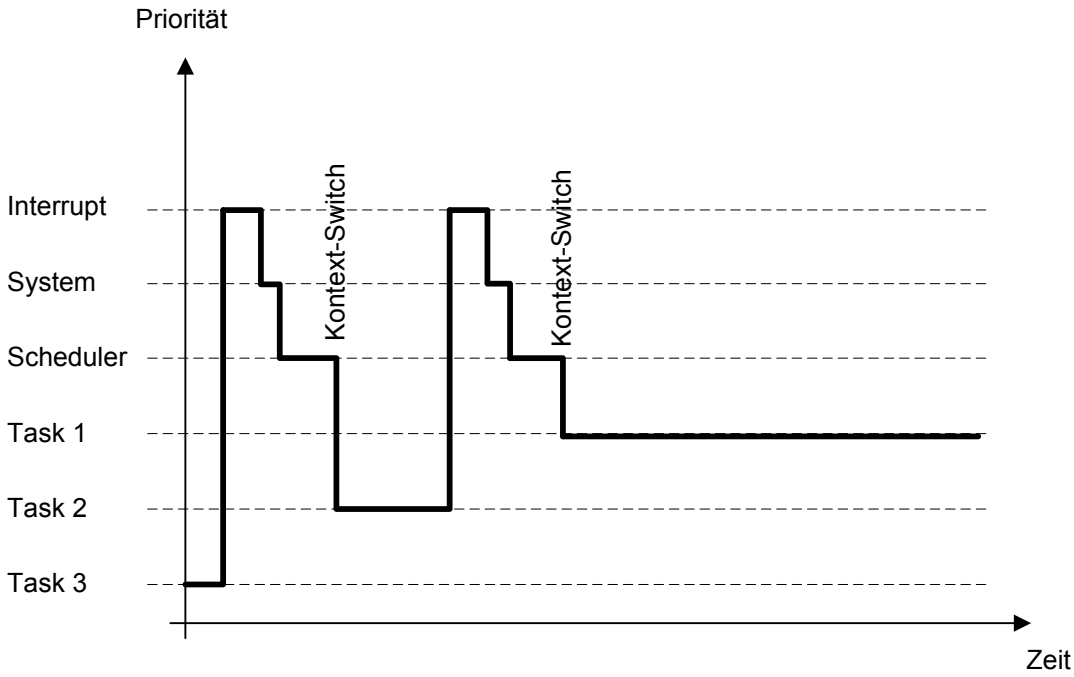


Bild: Taskdiagramm -Tasks hoher Priorität können jederzeit laufende Tasks unterbrechen

5.2.8. Zusammenwirken der einzelnen EBS-Module

Wie arbeiten diese elementaren Programmmodule nun zusammen?

Der von der Echtzeituhr gelieferte Taktimpuls (Uhrimpuls-Takt) stößt zyklisch die Zeitverwaltung an.

Die **Zeitverwaltung** ermittelt, wann welcher der existenten Rechenprozesse ablaufen soll (Soll-Zeitpunkte) und teilt dies der Taskverwaltung mit.

Ablaufbereite Rechenprozesse werden von der Taskverwaltung an die Prozessorverwaltung gemeldet.

Die Prozessorverwaltung veranlaßt dann den Start der Rechenprozesse.

Ist der Rechenprozess beendet, meldet dies die Prozessorverwaltung an die Taskverwaltung zurück. Die Taskverwaltung markiert für die Task den aktuellen Zustand „ruhend“.

Betrachten wir die einzelnen Module noch genauer und evaluieren, welche Daten und Aktionen für das jeweilige Programmmodul erforderlich ist.

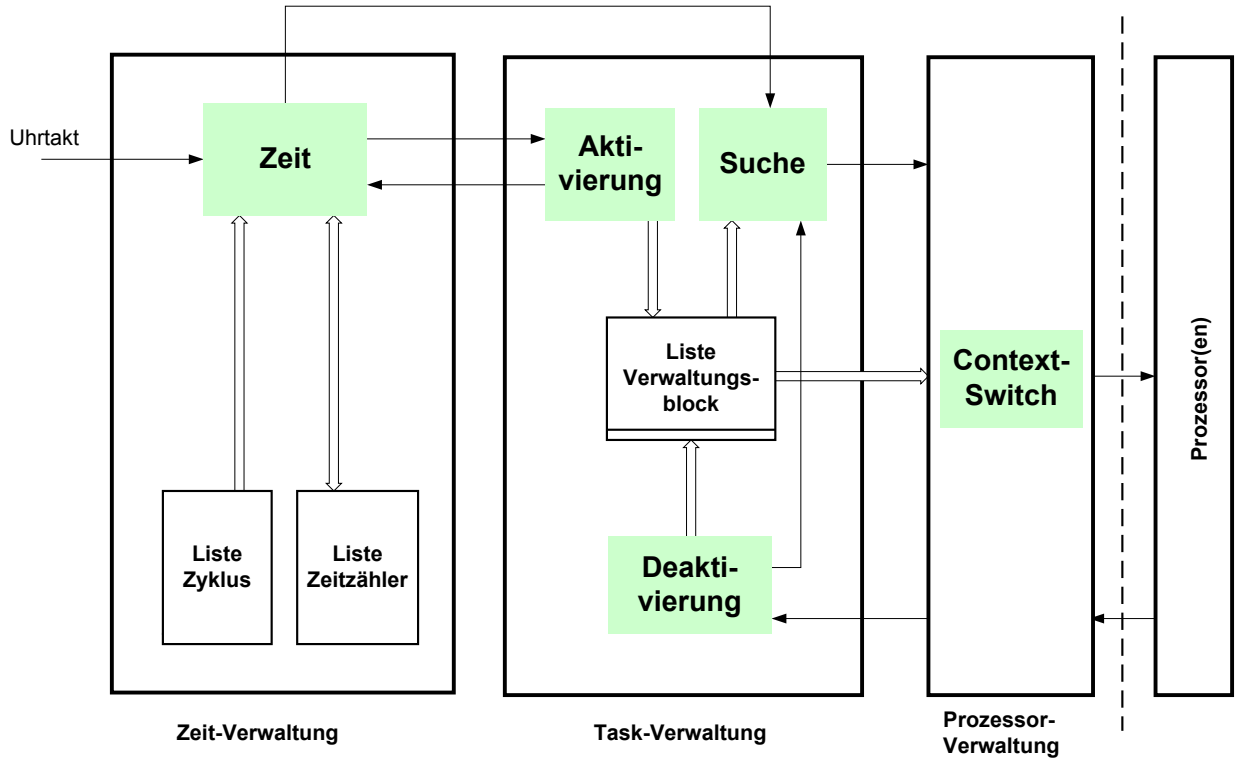


Bild: Aufbaustruktur des Minimal-Echtzeitbetriebssystems

Echtzeitprozesse können spontan durch Interrupt initiiert werden oder aber periodisch auftreten. -> siehe Folie Statisch periodische Echtzeitsysteme (Bsp.: 2 Tasks: 1. Task: Periodendauer 40ms, 2.Task: Periodendauer 60ms)

Für periodische Echtzeitprozesse muss es in dem Modul Zeitverwaltung eine Liste/Tabelle geben, die die unterschiedlichen Zykluszeiten verwaltet (Liste Zykluszeiten).

Die Zeitverwaltung muss nun der Taskverwaltung mitteilen, das Prozess1 nach 40ms und Prozess2 jeweils nach 60ms gestartet werden soll. Wie kann dies realisiert werden? -> Einführung von Listen.

Man verwendet zur Zykluszeit-Bildung Zähler ein, die bei jedem Taktimpuls um eins dekrementiert werden. Bei Null-Durchlauf wird die entsprechende Task-Nr der Taskverwaltung gemeldet. Gleichzeitig setzt die Instanz Zeit nach dem Null-Durchlauf den aktuellen Zeitzähler wieder auf den Anfangszählwert, d. h. der Zykluszeit (der Wecker wird erneut gestellt).

Die Zeitverwaltung benötigt somit zwei Listen, die Liste der Zykluszeiten und die Liste Zeitzähler mit jeweils m Speicherplätzen sowie die Instanz Zeit.

Jeder Taktimpuls der Echtzeituhr (hier alle 1ms) stößt die Aktion Zeit an. Die Aktion Zeit liest darauf den Zeitzähler der Task1, dekrementiert diesen und speichert den Wert wieder ab. Anschließend wird der Zeitzähler der Task2 gelesen, um eins dekrementiert und wieder in der Liste der aktuellen Zeitzähler abgespeichert.

Nach jeder Dekrementierung findet eine Prüfung statt, weist nämlich der akt. Zeitzähler für eine Task eine Null auf, wird jetzt aus der Liste Zykluszeiten der Anfangszählerwert wieder in den entsprechenden Listenplatz von akt. Zeitzähler geschrieben. Anschließend wird das Programmmodul Taskverwaltung angestoßen.

Welche Aufgaben übernimmt nun das **Programmmodul Taskverwaltung**?

Das Programmmodul Taskverwaltung hat die Aufgabe die Rechenprozesse (Tasks) zu verwalten.

Hierzu verwendet das Modul ebenfalls eine Listenstruktur zur Verwaltung der Tasks. (Aufzeichnen des Verwaltungsblockes). In diesem Verwaltungsblock stehen die aktuellen Task-Zustände und die Anfangsadressen der Rechenprozesse.

Dieser Verwaltungsblock besteht ebenfalls aus m Listeneinträgen, von denen das erste Bit den Zustand (0 : ruhend, 1 : bereit), der restliche Teil des Speicherworts die Startadresse der Task enthält.

Die Zeitverwaltung ermittelt, wann welche Rechenprozesse ablaufen sollen (Soll-Zeitpunkte) und teilt dies der Taskverwaltung mit.

Das Modul Taskverwaltung hat die Aufgabe ablaufbereite Rechenprozesse an die Prozessorverwaltung weiterzugeben. (=> Ist-Zeitpunkte)

Wie macht sie das:

Sie durchforstet den Verwaltungsblock und und meldet die entsprechende Task an die Prozessorverwaltung.

Möglichkeiten der Prioritätssteuerung -> siehe Schedulerverfahren.

Das Programmmodul Prozessorverwaltung

Das Programmmodul Prozessorverwaltung wird zur Zuteilung des Betriebsmittels Prozessor (hier Ein-Prozessorsystem) und zum Start der Rechenprozesse benötigt. Die Prozesse (Tasks) gehen in den Zustand „running“ über.

5.3. Erweiterung 1, 2 und 3 des Minimalen Echtzeitbetriebssystems

1. Erweiterung: Zulassen längerer Rechenzeiten- Erweiterte Aufbaustruktur als Folie auflegen und Text auf Folie schreiben

- Gegenüberstellen Listenstruktur bei dem Minimalsystem und bei dem 1. erweitertem System mit den Context-Registern

2. Erweiterung: Vorsehen der Möglichkeit von Alarminterrupts (und später von ISR mit niedrigen Prioritäten)- Erweiterte Aufbaustruktur als Folie auflegen und Text auf Folie schreiben

- Funktionesweise der Unterbrechungsverwaltung erläutern am Bsp. eines Task/EBS-Zeitdiagramms

3. Erweiterung: Unterstützung von Betriebsmitteln mit längeren Ein-/ u. Ausgabezeiten

- Gesamte erweiterte Aufbaustruktur als Folie auflegen und Text auf Folie schreiben

5.3.1. Erweiterung 1: Längere Prozess-Rechenzeiten zulassen

Um das grundsätzliche Vorgehen bei der Lösung der Organisationsaufgaben eines Betriebssystems zu zeigen, wurde in den vorherigen Abschnitten die Aufgabenstellung stark vereinfacht. Es sollen nun schrittweise die Einschränkungen wegfallen, um einem realen Echtzeit-Betriebssystem näher zu kommen.

Bisher wurde davon ausgegangen, dass die Summe der Rechenzeiten aller Rechenprozesse kleiner sei als die Uhrimpuls-Taktzeit T. Entfällt diese Simplifizierung, d. h. werden längere Rechenzeiten der Tasks zugelassen, kann der Fall eintreten, dass beim Eintreffen eines Zeittaktes ein noch laufender Rechenprozess mit längerer Ausführungsdauer und niedriger Priorität unterbrochen werden muss, um einen Rechenprozess höherer Priorität zum Zuge kommen zu lassen.

Zumindest aber muss der Programmteil Zeitverwaltung nach jedem Uhrimpuls immer ablaufen, um die Zeitzähler entsprechend zu aktualisieren. In jedem Fall muss dafür gesorgt werden, dass der unterbrochene Rechenprozess später wieder an der Unterbrechungsstelle mit den zur Zeit der Unterbrechung gültigen Werten fortgesetzt werden kann.

Um diese zusätzlichen Anforderungen erfüllen zu können, wird das Minimal-Echtzeitbetriebssystem wie folgt erweitert:

Es wird ein zusätzliches Verwaltungsprogramm "Unterbrechungsverwaltung" eingeführt. Dieses hat zur Aufgabe, beim Auftreten eines Uhrzeit-Interrupts die, von einem eventuell gerade laufenden Rechenprozess verwendeten Register des Prozessors (Akkumulator, Zustandsregister, Programmzähler, Arbeitsregister etc.) zu retten und dann die Zeitverwaltung anzustoßen. Bild 6.20 zeigt das entsprechend erweiterte Hierarchiediagramm.

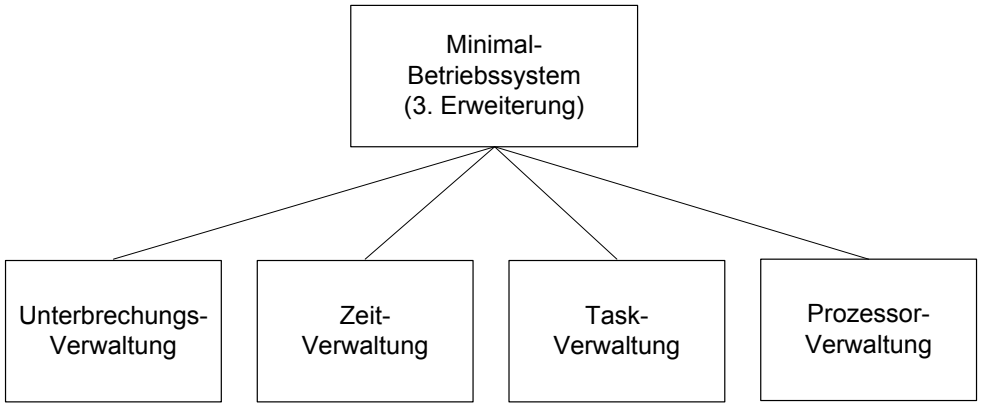


Bild: Erweitertes Hierarchiediagramm nach dem Zulassen längerer Rechenzeiten der Tasks.

Die Liste Verwaltungsblock, die bisher nur die Anfangsadresse der Tasks und ein Zustandsbit enthielt, wird um die Startadresse nach einer Unterbrechung und die notwendige Anzahl von Register-Speicherplätzen erweitert, um in diese Speicherplätze bei der Unterbrechung einer Task die Registerinhalte retten zu können. Das nachstehende Bild zeigt die erweiterte Listenstruktur.

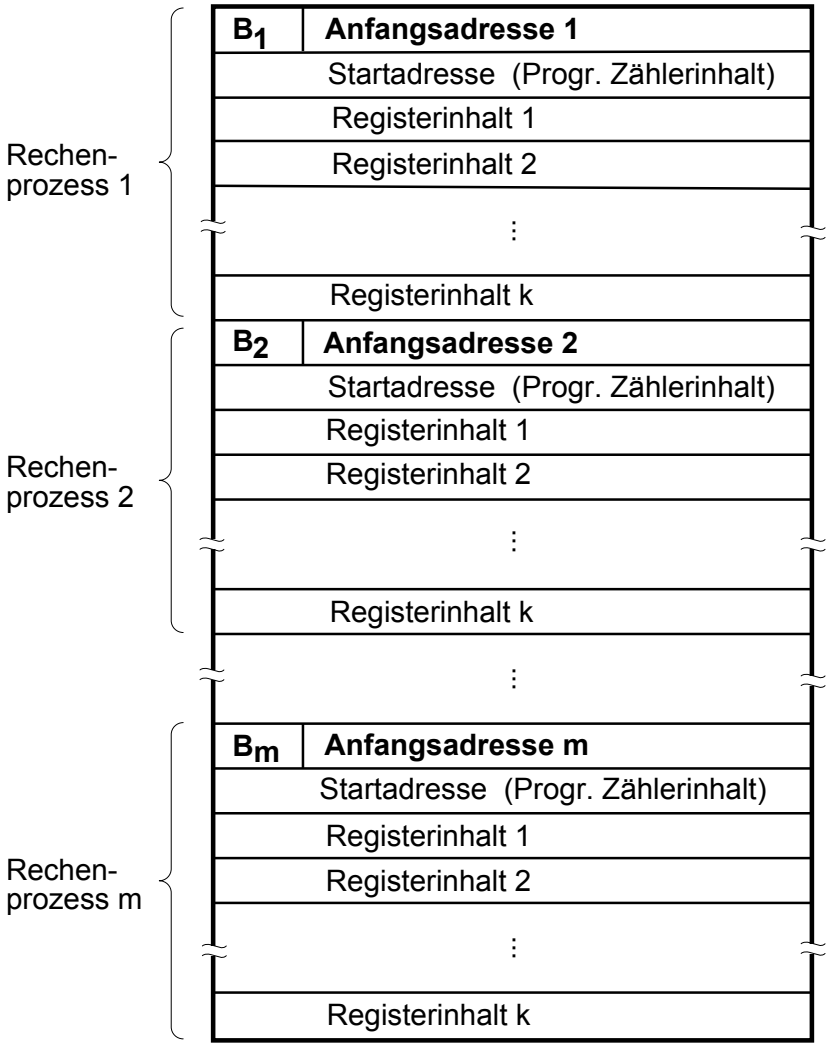


Bild: Erweiterung der Listenstruktur der Taskverwaltung zur Speicherung des Zustandsvektors
(z. B. Inhalte der div. Register = Context-Register) des Prozessors bei Unterbrechungen

Der Programmteil Prozessorverwaltung erhält die zusätzliche Aufgabe, vor dem Start eines als ablaufbereit gemeldeten Rechenprozesses die Register mit den in der Liste Verwaltungsblock stehenden Inhalten zu laden (bei nicht unterbrochenen Rechenprozessen stehen dort die Anfangswerte 0) und dann den Rechenprozess an der in der Zelle "Startadresse" im Verwaltungsblock angegebenen Stelle zu starten.

Der Programmteil Deaktivierung erhält die Zusatzaufgabe, nach Beendigung eines Rechenprozesses die Anfangsadresse der Programmcodes in die Zelle "Startadresse" im Verwaltungsblock zu laden (und eventuell die Registerinhalte im Verwaltungsblock mit

Null zu initialisieren). Damit ist gewährleistet, dass der Task bei der nächsten Beauftragung wieder von vorne abläuft.

Mit den erwähnten Erweiterungen sieht das Blockdiagramm des erweiterten Minimal-Betriebssystems wie folgt aus:

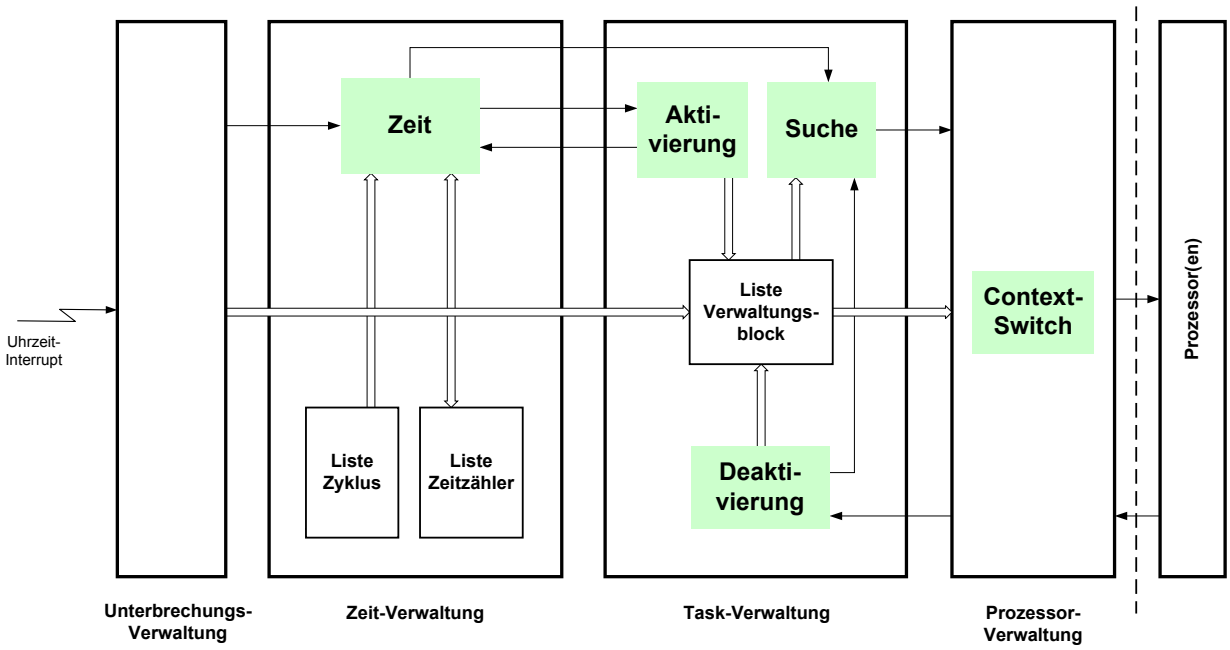


Bild: Blockdiagramm des Echtzeit-Betriebssystems (1. Erweiterung)

5.3.2. Erweiterung 2: Möglichkeit von Alarminterrupts vorsehen

In einem nächsten Schritt soll nun die Vereinfachung fallengelassen werden, dass nur zyklisch aktivierte Tasks vorhanden sind. Vielmehr sollen zusätzlich bis zu k Rechenprozesse eingeplant werden, deren Aktivierung von - zeitlich nicht vorhersehbaren - Alarminterrupts ausgelöst werden soll.

Außer der natürlich aufgrund der größeren Anzahl von Rechenprozessen notwendigen Vergrößerung der Listen ist eine Erweiterung der Unterbrechungsverwaltung erforderlich. Sie muss nun neben der Register-Rettung je nach Art des Interrupts (Uhrzeit- oder Alarminterrupt) entweder die Zeitverwaltung anstoßen oder aber mit Hilfe des Programmteils Aktivierung ein dem Alarminterrupt zugeordnetes Antwortprogramm in den Zustand "bereit" setzen und anschließend das Programmteil Suche starten. Die Adressen der Alarminterrupt-Antwortprogramme werden vor denen der zyklisch aktivierten Tasks in den Verwaltungsblock geschrieben, erhalten somit die gewünschte höhere Priorität.

Das nachstehende Bild zeigt die so erweiterte Version des Mini-Betriebssystems mit den zusätzlichen Alarm-Interrupts und der Möglichkeit, die Aktivierung direkt von der Unterbrechungsverwaltung anzustossen.

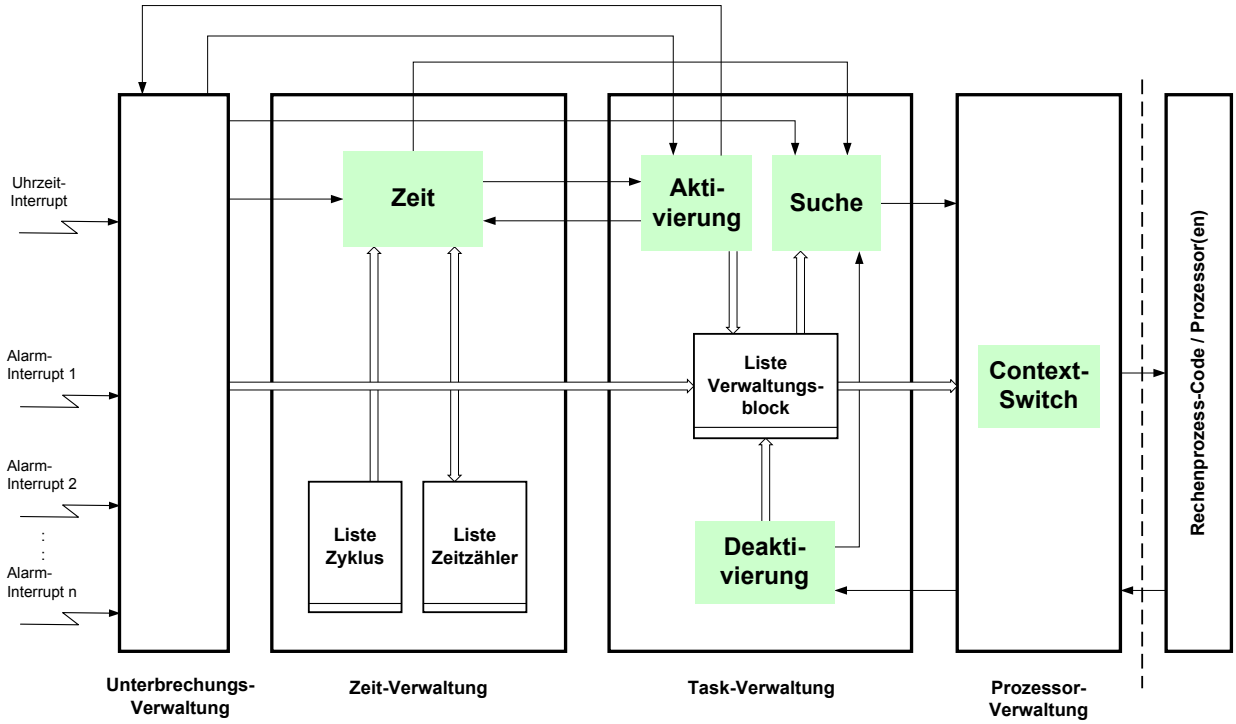


Bild: Aufbaustruktur für das Minimal-Betriebssystem (2. Erweiterung)

5.3.3. Erweiterung 3: Betriebsmittelverwaltung für E/A-Geräte

Bisher war immer noch davon ausgegangen worden, dass Ein-/Ausgabeoperationen wie alle anderen Operationen kleine Ausführungszeiten haben. Das ist allerdings bei langsamer Peripherie - z. B. einem Analog-Digital-Umsetzer mit einer Wandlungszeit von 10 Millisekunden - nicht der Fall.

Soll deshalb die vorherige Vereinfachung „Annahme vernachlässigbarer Ausführungszeiten für E/A-Operationen“ wegfallen, muss ein zusätzliches Verwaltungsprogramm E/A-Verwaltung eingeführt werden, das die Organisation langsamer Ein-/Ausgabeoperationen übernimmt. Das nächste Bild zeigt das neue und erweiterte Minimal-Betriebssystem.

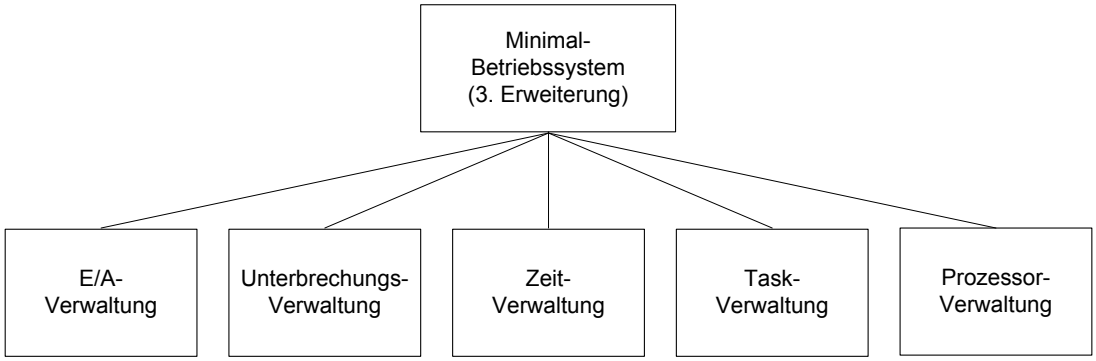


Bild: Blockdiagramm des Minimal-Betriebssystems (3. Erweiterung)

Im Wesentlichen muss die E/A-Verwaltung ermöglichen, dass während der langsamen Ein-/Ausgabe-Operationen, z. B. der Analog/Digital-Wandlung, der Prozessor

andere Rechenprozesse bearbeiten kann und dann durch ein Interruptsignal vom Peripheriegrät, das die Beendigung der E/A-Operation anzeigt, darauf aufmerksam gemacht wird, dass im Rechenprozess der die E/A-Anweisung enthält, fortgefahren werden kann.

5.4. Anforderungen an ein reales produktives Echtzeit-Betriebssystem

"Normales" Betriebssystem	“Echtzeit”-Betriebssystem
hohe Antwortzeiten auf Ereignisse von außen	Latenzzeiten gering (typ. 20 µs) und konstant
hohe Schedulingzeiten (ab 10 ms)	Scheduling im µs Bereich (30 – 200 µs)
lange Interruptsperren	spezielle, schnelle Interruptbehandlung (teilweise mit Speicherung)
nichtunterbrechenbare Kernelaufrufe	teilweise preemptive Kernel
"gerechte" Rechenzeitaufteilung	Prioritätsklassen mit "ungerechten" Schedulern; optimiert auf Einhaltung der zeitlichen Anforderungen der jeweiligen Prozesse
Annahme jedes Prozesses	Lastanalysen möglich

Bild: Vergleich „Normales“ und „Echtzeit“-Betriebssystem

5.4.1. Übergang zum realen Echtzeit-Betriebssystem

Bei der Entwicklung des erweiterten Betriebssystems wurde schrittweise auf anfangs getroffene Vereinfachungen verzichtet. Nach wie vor ist aber der Ausdruck "Minimal"-Betriebssystem gerechtfertigt, da von zahlreichen weiteren vereinfachenden Bedingungen ausgegangen wurde, die bei Echtzeit-Betriebssytemen in der Praxis nicht vorausgesetzt werden dürfen.

Erst ein Verzicht auf diese Annahmen, von denen nachfolgend einige aufgeführt sind, würde aus dem Minimal-Betriebssytem ein in der Realität einsetzbares Echtzeit-Betriebssystem entstehen lassen. Die wichtigsten zusätzlich implizit getroffenen Vereinfachungen waren:

- die Betriebssystemprogramme selbst sind nicht unterbrechbar
- eine Mehrfachbeauftragung eines Rechenprozesses, d. h. eine erneute Beauftragung vor der Beendigung, ist ausgeschlossen
- eine gegenseitige Beauftragung von Tasks untereinander (vgl. die Anweisung ACTIVATE in der Programmiersprache PEARL) ist nicht möglich
- eine Synchronisierung der Rechenprozesse, z. B. mittels Semaphoroperationen, ist nicht möglich

- es gibt keine Datenkommunikation zwischen den Rechenprozessen, d. h. Austausch von Daten, gemeinsame Benutzung von Daten etc.
- eine dynamische Änderung der Prioritäten der Tasks während der Programmdurchführung ist nicht möglich
- die Rechenprozesse befinden sich stets im Arbeitsspeicher, Hintergrundspeicher sind nicht vorhanden.

5.4.2. Embedded-Betriebssysteme in der Praxis - Eine Übersicht

Embedded-Betriebssysteme www.elektronikpraxis.de/DEEP

DEEP
Ihr www.Kontakt zum Anbieter!

Hersteller/Anbieter	Produktname	Betriebssystem – Art: Embedded-OS	Echtzeit-OS	Spezielle Kernel	Unterstützte Prozessoren- und Controller-Familien	Echtzeit-Werkzeuge: Entwicklungssystem	Compiler	Debugger	Bibliotheken	Anwendungen: Internet	Sonstiges	Kennziffer
3Soft	Pro OSEK	●	●	●	HC08, HC12, STAR12, 683xx, PPC 555, C16x, TriCore, ST9, ST10, F2MC-16L(x),TMS470, V850	●			●	●	Schulung, Hardware-Abstraction-Layer, Treiber	401
Aonix	Object Ada/Raven			●	x86, PPC	●	●	●			Schulung	402
Digalog	RBOS, iCON-L	●	●	●	68k, 386, Z80	●	●	●	●	●		403
Embedded Power/ AK Elektronik	RTXC	●	●	●	x86, PPC, 68k, TI, AMD	●	●	●	●	●		404
Engelmann & Schrader	RTOS		●		8051, 166	●	●	●	●	●		405
esd	RTOS-UH	●	●		68k, PPC, 405	●	●	●	●		Schulung, BSP	406
	RTAI-Linux	●	●		PPC, 405	●	●	●	●	●	Schulung, BSP	
ETAS	ERCOS	●	●		MPC5xx, C16x, TMS470, V85x, M68HC12, SH7055F, TriCore, 683x6	●						407
GreenHills	INTEGRITY	●	●		PPC, MIPS, ARM	●	●	●	●	●		408
	ThreadX	●	●		32-Bit-Controller	●	●	●	●	●		
HighTec	PXROS	●	●	●	x86, 68k, PPC/MPC, C16x, TriCore	●	●	●	●	●	Visualisierung, Schulung, Linux-Emulation	409
IEP	RTOS-UH	●	●		MC68xxx, PPC	●	●	●	●	●	Feldbusse	410
Keil	FR51	●	●	●	8051	●	●	●	●			411
	FR166	●	●	●	166/167, ST10	●	●	●	●	●		
Lineo	Embedix	●	●		x86, ARM, SH, MIPS, PPC, Coldfire, Dragonball, StrongARM	●	●	●	●	●	Training	412
	RTXC Quadros		●		ARM, PPC, MCORE, DSP, StarCore, Coldfire	●	●	●	●	●	Training	
Live Devices/AK Elektronik	RTA-Realogy	●		●	PIC18x, HC08, HC12, MPC5xx, CPU32, 7TDM, H2, H8S, Blackfin, C16x, TriCore	●			●	●	Automotive (OSEK), Schulung	413
Lynux Works/SYSGO	Lynux OS	●	●	●	x86, PPC, ARM, MIPS	●	●	●	●		Schulungen	414
MICROSOFT	Windows XP Embedded	●			x86	●	●	●	●	●		415
	Windows CE.NET	●	●		x86, SH3, MIPS, xScale, ARM	●	●	●	●	●		
MontaVista	Montavista Linux	●	●	●	MIPS, NEC, QED, Au1000, Lexra4189, Tx3927	●	●	●	●		Realtime-Kernel	416
On Time Informatik	RTOS-32	●	●			●		●	●	●		417
OSE Systems	OSE	●	●		68k, ARM, StrongARM, MIPS, PPC, TMS320Cx, Tigersharc, ST100, C166, TriCore	●	●	●	●	●	Training, DSP-Technologie	418
PRAHM	RTMOS	●	●	●	x86, 68k, 8051, HC11, ARM, DSP	●	●	●	●		Grafische Programmier-Tools, Software-SPS	419
Precise/HSP	MQX	●	●		68k, PPC, TI-DSP, ARM, ARC, ColdFire, MCore	●		●	●	●		420
QNX	QNX	●	●		x86, PPC, ARM, MIPS, StrongARM, XScale	●	●	●	●	●	Schulung & Training	421
RadiSys Microware	OS-9	●	●		x86, XScale, 68k, ARM, MIPS, StrongARM, SH3/4/5	●	●	●	●			422
SEGGER	embOS	●	●		MIPS7700, M16Cx, M32C, AT90, ARM7/9, F2MC-16LX, FR30, 64180, H8S, SuperH, C166/167, K0/4, V25/850/850E, ST10, Z80/180	●			●	●	TCP/IP-STACK	423
Smart Networks Devices	HyNetOS 2.0	●	●	●	Hyperstone E1	●	●	●	●	●	Java (CLD C) als Option Bluetooth, Schulungen	424
SORCUS Computer	OSX		●		x86	●		●	●		Visualisierung, Schulung	425
SYSGO	ELinOS	●	●		x86, PPC, ARM	●	●	●			Schulung	426
TenAsys/CC&I	INtime		●		x86	●	●	●	●		Schulung	427
Willert Software	New Dimension		●	●	C166, XC166, TriCore, M16C, M32C, ST10	●	●	●	●	●	Start-up-Training, UML-Framework, Grafikbibliotheken	428
WindRiver	VxWorks		●		Motorola, IBM, ARM, Hitachi, Intel, MIPS, SPARC u.a.	●	●	●	●	●	Training	429
	Virtuoso		●		Analog Devices, HammerHead, TI							

Bild: Embedded-Betriebssysteme - Eine Übersicht

5.5. Klassifizierung/Strukturierung kommerz. Echtzeit-Betriebssysteme

- Folie: Gliederung der Softwarekomponenten auf einem Prozessrechner (mit Betriebssystem, Laufzeitsystem und Systemunterstützung)
  Übungsaufgabe
- Folie: Schichtenförmiger Aufbau eines Realzeitsystems darstellen und Nucleus definieren
- Folien: EBS-Komponentenmodell und Klassenschema
- Nucleus definieren und auf Objekttypen eingehen (gemäß Posix), wie Tasks, Mailboxen und Semaphoren etc.
- Aufsatz Embedded Programming

5.5.1. Gliederung der Prozessrechnerprogramme

In den bisher behandelten Abschnitten wurde das Betriebssystem als vom Hersteller mitgeliefertes Programmsystem erläutert, das dem Anwender eine hardwareunabhängige Schnittstelle für seine Programme bietet, die ihm fundamentale Aufgaben wie asynchrone Durchführung von Rechenprozessen, Speicherverwaltung usw. abnimmt.

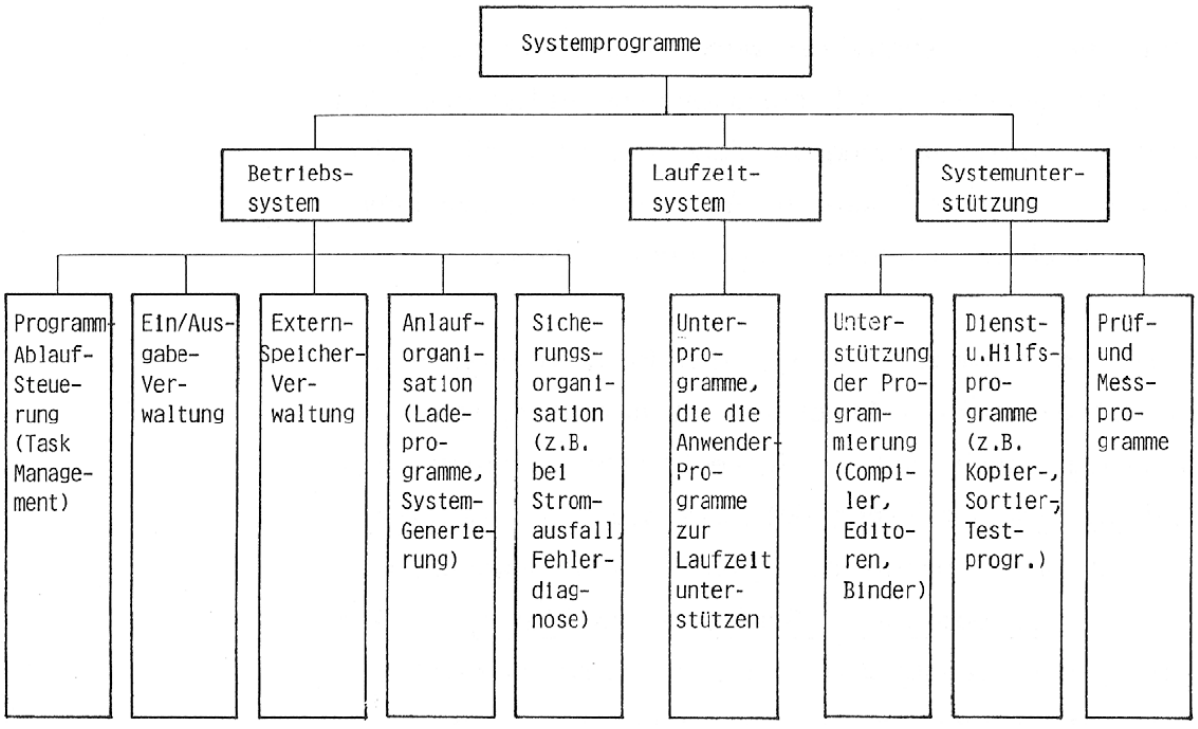


Bild: Gliederung der Softwarekomponenten auf einem Prozessrechner in Betriebssystem, Laufzeitsystem und Systemunterstützung

Für die effektive Programmentwicklung, den Programmablauf und den Programmtest auf einem Prozessrechner ist darüber hinaus noch weitere Unterstützung durch andere Systemprogramme notwendig. Außer dem Betriebssystem unterscheidet man noch das Laufzeitsystem zur Unterstützung während des Ablaufs der Anwen-

derprogramme und die allgemeine Systemunterstützung. Als Laufzeitsystem wird die Gesamtheit der Programme bezeichnet, die zur Laufzeit der Anwenderprogramme benötigt werden, die aber der Anwender nicht jedes mal selbst schreiben muss (z.B. Programme für trigonometrische Funktionen, wie etwa ein Programm das den Sinus berechnet).

Zu den allgemeinen Systemunterstützungsprogrammen gehören vor allem die gesamten Programme, die für die Programmerzeugung notwendig sind, wie ein Editor zur interaktiven Erstellung des Programmquellcodes, ein Compiler bzw. Assembler zur Erzeugung von Maschinencode, sowie ein Linker zum Binden verschiedener zusammengehöriger Programm-Module etc., sowie eine Reihe von Standard-Hilfs-, Dienst- und Prüfprogrammen. (siehe Bild Gliederung der auf einem Prozessrechner verfügbaren Systemprogramme)

Der Anwender kann sich aus diesen Systemprogrammen seine individuelle, den jeweiligen Forderungen angepasste Programmier- und Laufzeitumgebung zusammenstellen.

5.5.2. Komponentenmodelle Embedded Systems und EBS

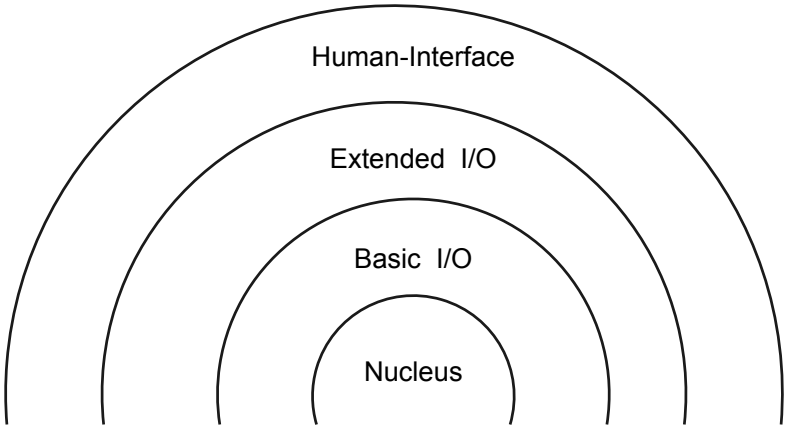


Bild: Schichtenförmiger Aufbau eines Realzeitsystems, z. B. Bei iRMx

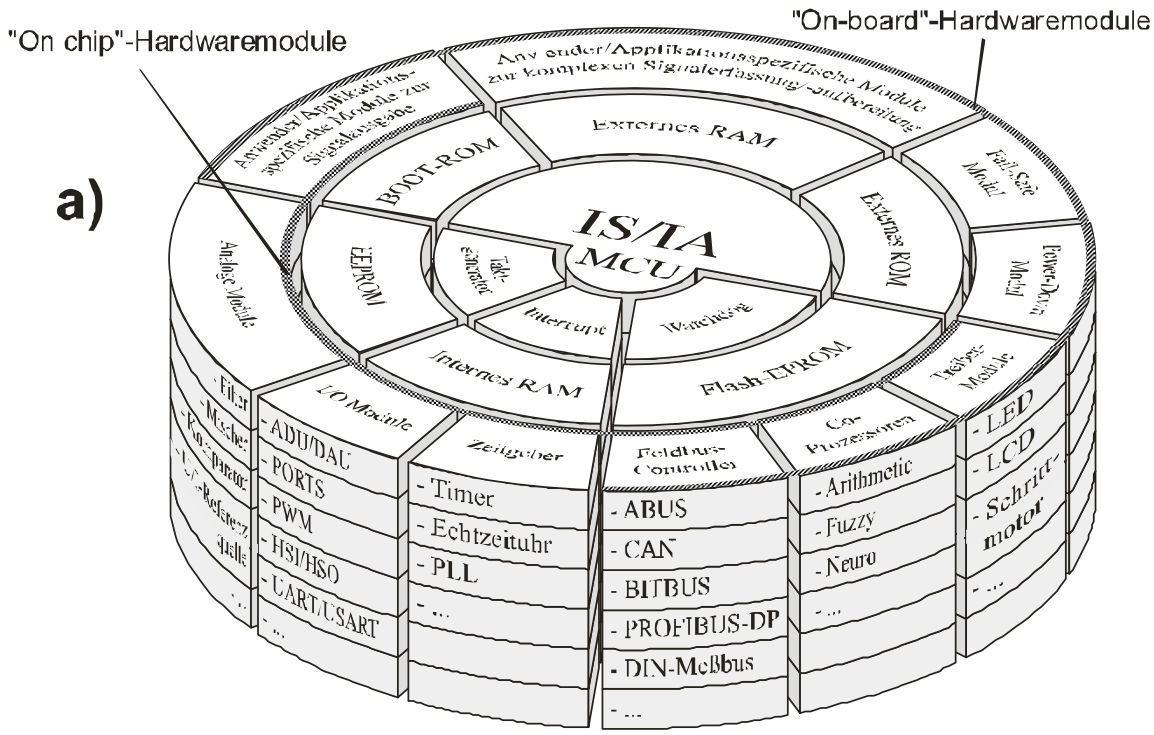


Bild: Komponentenmodell eines Embedded Systems

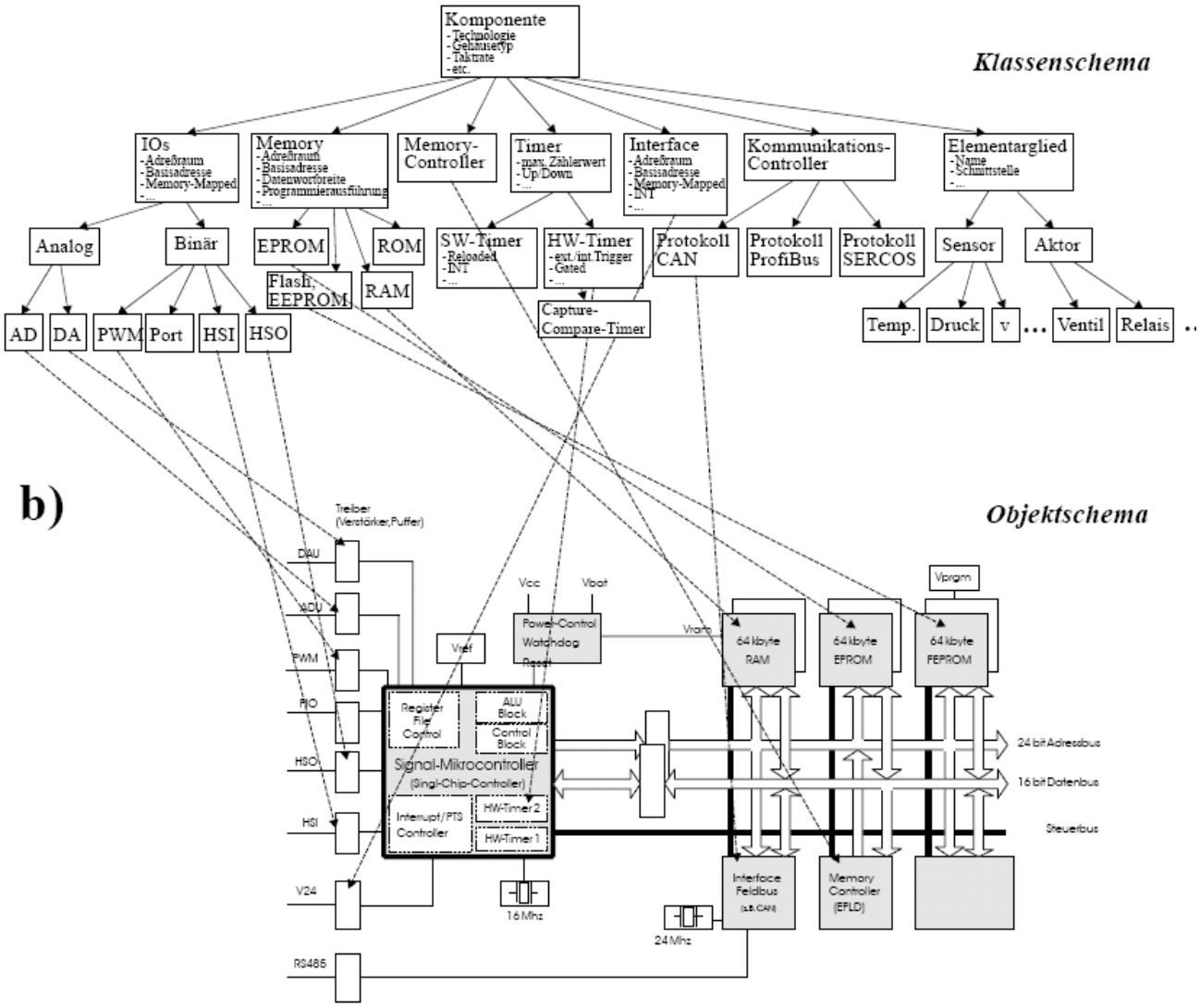


Bild: Komponentenmodell eines Embedded Systems und Klassenschema

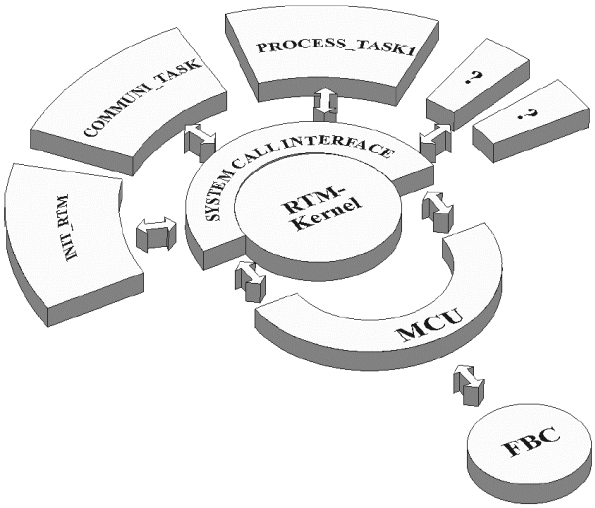


Bild: Distributed Realtime Kernes (DCX) von intel und Taskanordnung

Komponentenmodell eines Embedded Systems und Klassenschema

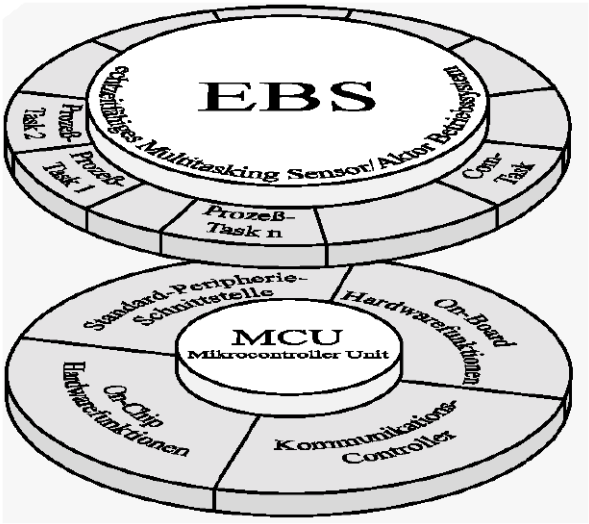


Bild: Integration Hardware und Echtzeit-Betriebssystem; Taskanordnung

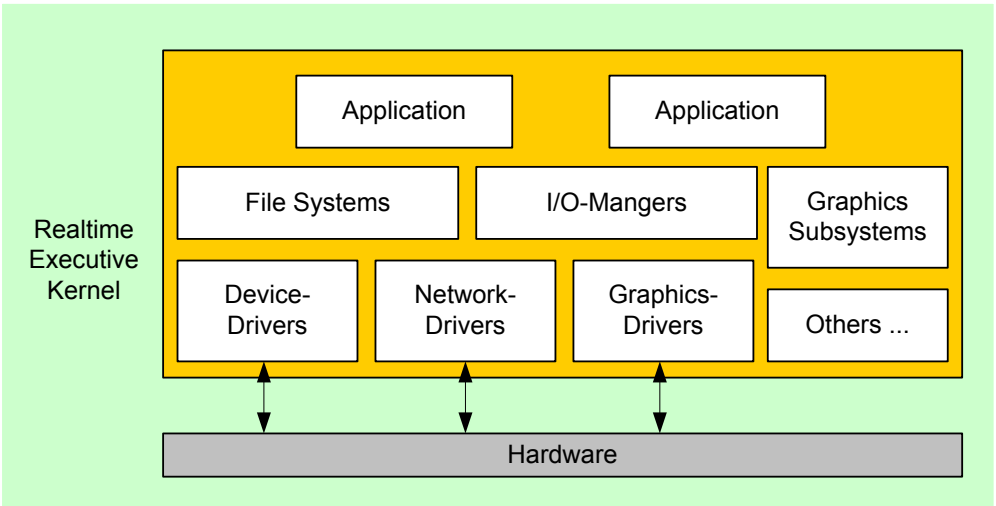


Bild: Monolithische EBS-Architektur

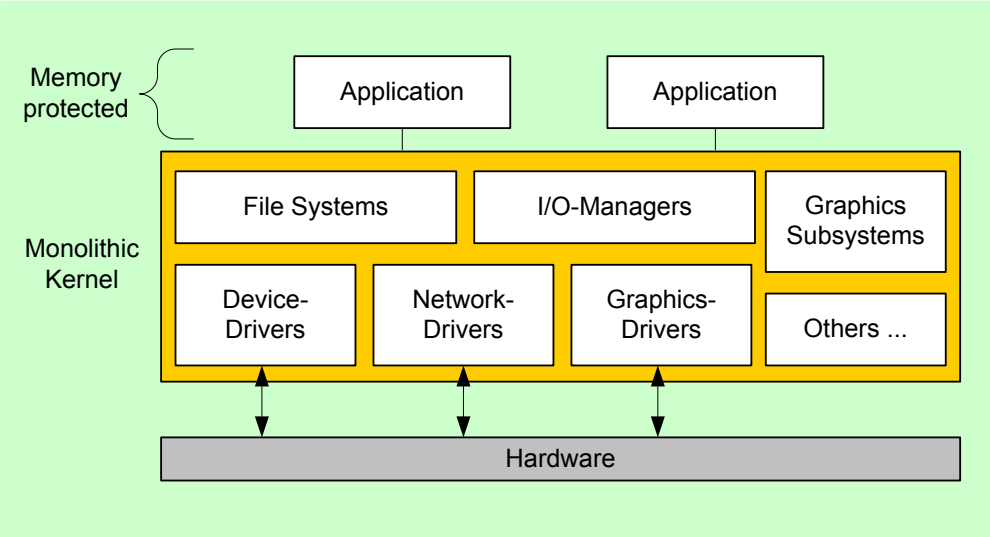


Bild: Monolithische Architektur mit Speicherschutz für Anwendungen

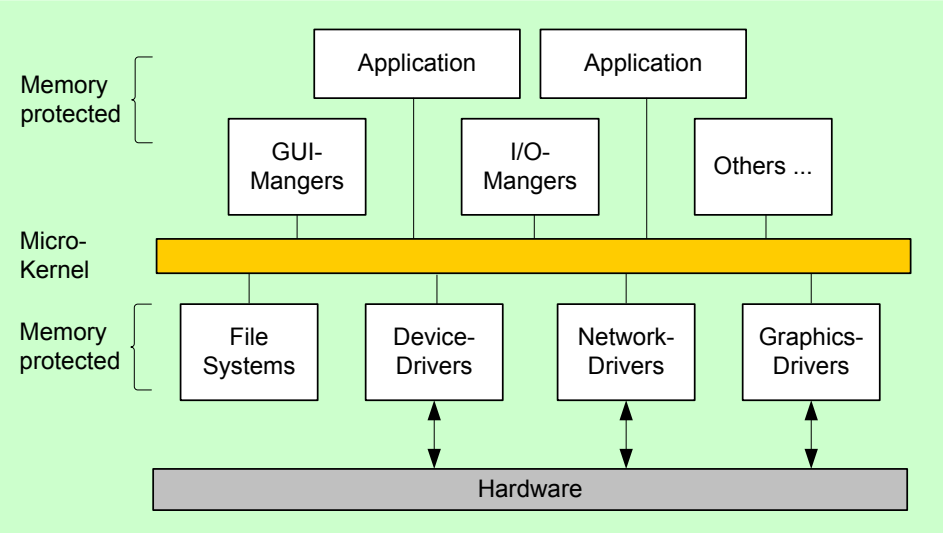


Bild: EBS-Architektur nach dem Universal Prozess Modell (UPM)

EBS-Architektur nach dem Universal Prozess Modell (UPM) mit Speicherschutz für jede Anwendung, jeden Treiber, I/O-Manager, jedes Protokoll, Dateisystem, Grafik-subsystem (laufen in eigenen speichergeschützten Adressräumen)
Microkernel nur mit Scheduler, Interprozesskommunikation und Hardware-Interrupt Handler.

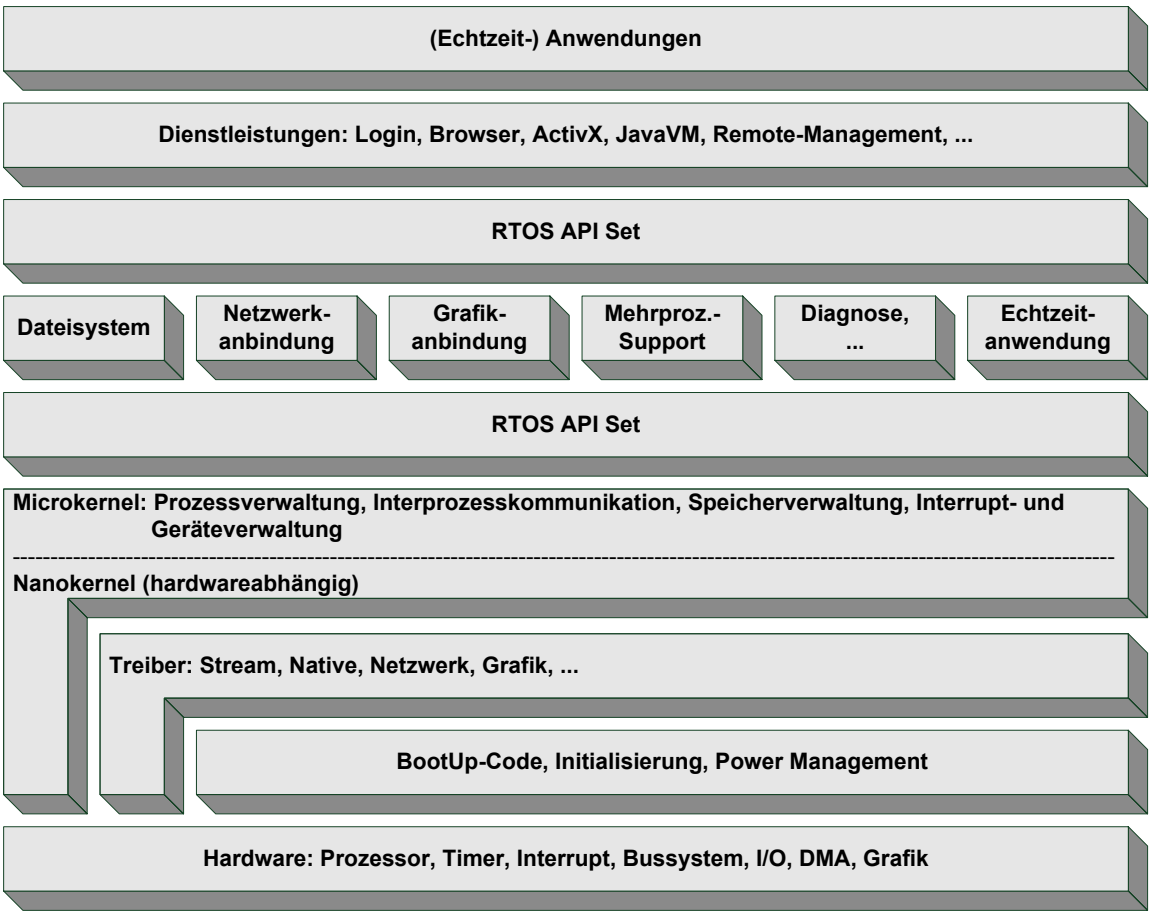


Bild: Strukturierung des Prozessrechnersystems VxWorks (Fa. Windriver)

5.5.3. Verwaltung von Hardware-Ressourcen

Die Verwaltung von Hardware-Ressourcen beinhaltet insbesondere:

Prozessorverwaltung: Sie umfasst das physikalische Zuteilen von Rechenzeit (Prozessorzeit) an die System- und Anwenderprogramme.

Speicherverwaltung: Dient dazu, jedem Programm den nötigen Programm- und Datenspeicher bereitzustellen. Mechanismen zum Speicherschutz können implementiert sein, sowie die Verwaltung von realem und virtuellem Speicher kann unterstützt sein.

Geräteverwaltung: Externe Geräte (z.B. Drucker) müssen Applikationsprogrammen zugeordnet und gesteuert werden (zwei Programme dürfen nicht gleichzeitig auf das selben Blatt drucken). Anwendungen sollen keine Kenntnisse über Gerätedetails besitzen müssen, so dass das Betriebssystem logische Anforderungen an Geräte in spezifische Geräte-Steuerbefehle umsetzt.

Dateiverwaltung: Eine Datenhaltung auf externen Speichern, wie Festplatten, muss organisiert werden. Hierfür wird in der Regel die Erstellung eines Dateisystems mit einer Verzeichnisstruktur benötigt. Unterstützung von Datenablage und Dateizugriff.

5.5.4. Verwaltung von Anwendungsprogrammen

Zur Steuerung und Verwaltung von Anwendungsprogrammen werden eine große Anzahl von Systemmodulen benötigt:

Prozess- und Threadverwaltung: Der Ablauf mehrerer unabhängiger Programme in einem System muss verwaltet werden (Multitaskingsysteme). Prozesse müssen erzeugt, gestartet, gestoppt und beendet werden. Die Bearbeitungsreihenfolge der einzelnen Prozesse muss nach bestimmten Strategien durchgeführt werden. Aktueller Zustand eines Prozesses muss ermittelbar sein.

Asynchrone Kommunikation: Verteilung von kurzen Systemmeldungen und Austausch größerer Informationen (Nachrichten) zwischen Prozessen. Die Informationen müssen sicher transferiert werden sowie ein gegenseitiger Ausschluß beim Lesen und Schreiben der Nachrichten garantiert werden.

Synchrone Kommunikation: Synchronisation von unabhängigen Prozessen. Beispielsweise kann ein Prozess erst mit seiner Aufgabe fortfahren, wenn ein anderer Prozess gewisse Aufgaben erledigt hat. Mechanismen umfassen Methoden zum gegenseitigen Ausschluss wie auch zur Signalisierung.

Interrupt-Behandlung: Äußere Ereignisse werden dem System in der Regel durch Interrupts mitgeteilt. Interrupt-Ereignisse werden zentral über das Betriebssystem bearbeitet, verwaltet und weitergegeben. Interrupts müssen maskiert und priorisiert werden sowie Threads über Tabelleneinträge (Interrupt-Vektortabelle) aktiviert werden.

Zeitdienste: Stellen den System-Tick zur Verfügung, verwalten den "Real-Time Clock" (Kalenderzeit) und unterstützen Timer- und WatchDog-Funktionalität.

Einfache Ein-/Ausgabe-Dienste: Unterstützung von elementaren Ein-/Ausgabe-Diensten, z.B. Ausgabe von Zeichen auf ein Terminal.

Fehlerbehandlung: Erkennung und Behandlung von Hard- und Softwarefehlern im Rechensystem sowie Erkennung von Fehlern durch nicht spezifizierte äußere Ereignisse. Fehlerbehandlung umfasst die Begrenzung der Auswirkungen des Fehlers sowie eine eventuelle Protokollierung.

Benutzerkommunikation: Ist im strengen Sinne nicht Teil des Betriebssystems, sondern gehört zur Menge von Dienstprogrammen. Es umfasst eine Mensch-Maschine-Schnittstelle, wie z.B. einen Kommandointerpreter.

5.6. Entwicklungsumgebung für VXWORKS

VxWorks ist ein kommerzielles Softwareprodukt der Firma WindRiver. Gegenüber anderen Systemen wie RTEMS zeichnet es sich durch Features wie einen vollständigen TCP/IP Stack und vor allem durch eine sehr komfortable Entwicklungsumgebung aus.

Zur Programmentwicklung dient wiederum ein auf dem Entwicklungs-PC laufender Cross Compilers. Der Programmcode wird in Objektfiles übersetzt und zunächst *nicht* gelinkt. Diese Objektfiles können auf das Targetboard übertragen werden, in-

dem für die Kommunikation das TCP/IP Protokoll benutzt wird. Als Übertragungsmedium dient wie bei SPYDER-CORE-P1 z. B. Ethernet, aber auch andere physikalische Medien sind möglich.

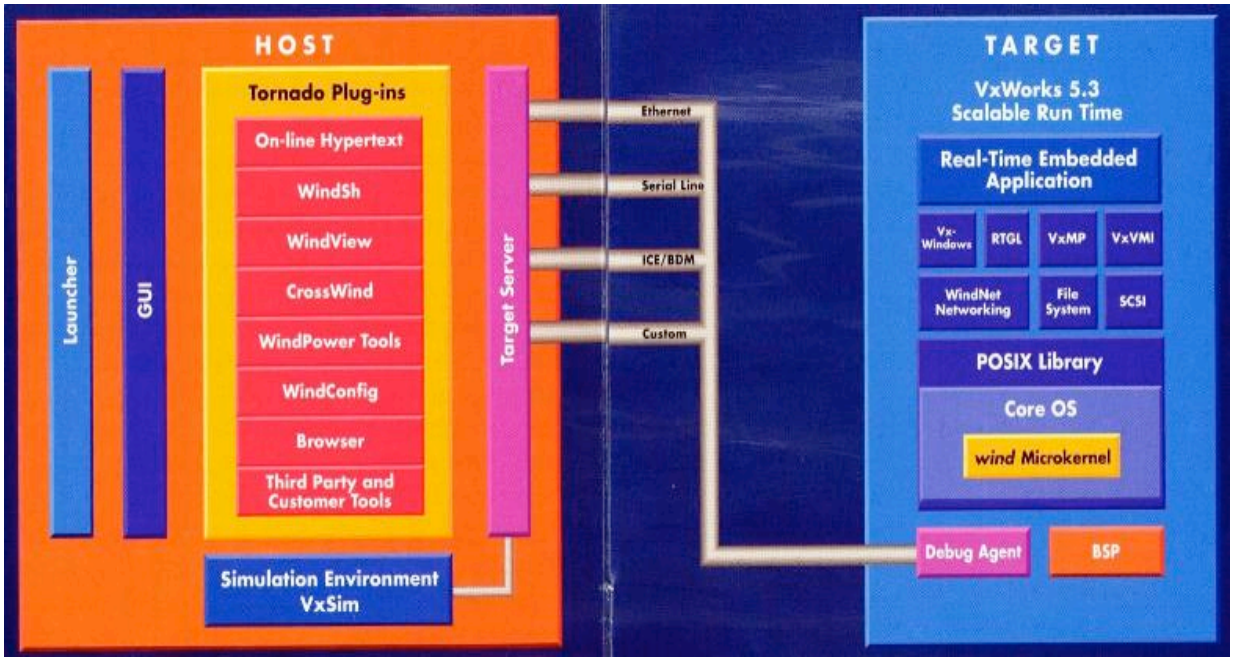


Bild: Entwicklungsumgebung für VxWorks

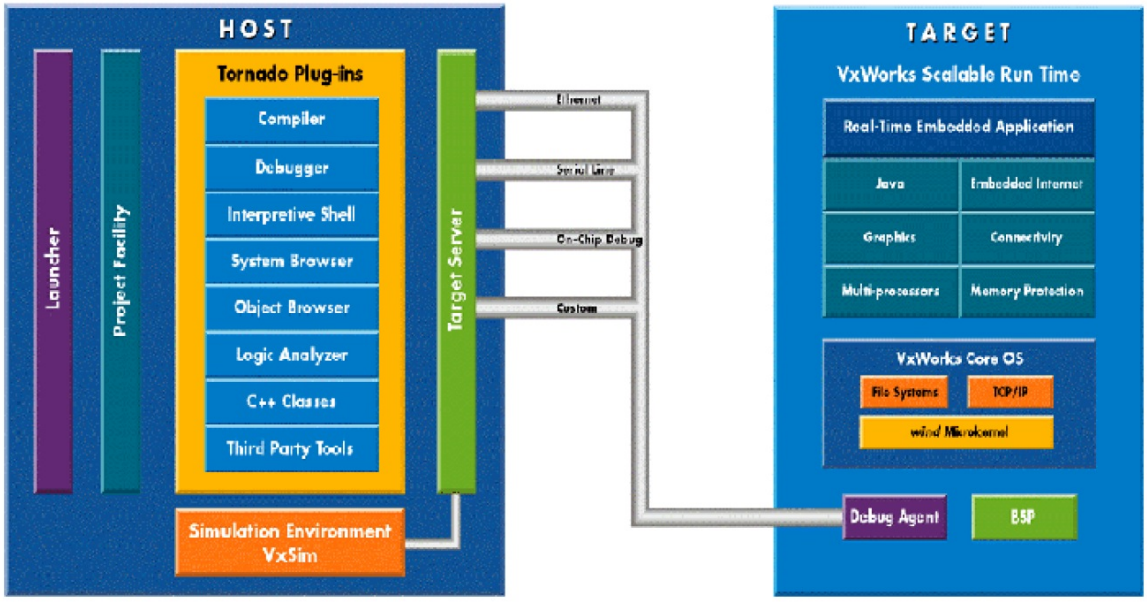


Bild: Hostentwicklung und EBSedd-Targetsysteme

Host (SUN)

- Solaris V 2.7
- Tornado V2
- Visicom “Real Time”

Target (VME-BUS-System)

- VxWorks Kernel
- VxWorks Modules
- Agents

Hosts (PC)

- MS Windows 2000 professional
- StethoScope
- Matlab
- Simulink

Um diese Kommunikation zu realisieren, findet ein sogenannter *Target-Server* Verwendung. Dieser kommuniziert mit einem auf dem Zielsystem laufenden *Target-Agent*. Der Target-Agent läuft als spezielle Task unter VxWorks. In Abbildung 4.4 ist dieses Prinzip schematisch dargestellt.

Damit der Bootloader nach dem Einschalten zur Verfügung steht, muss er sich neben anderen wichtigen Initialisierungsroutinen auf einem nichtflüchtigen Speichermedium befinden, z. B. einem EPROM oder Flash-Memory. Die Spyder-Architektur bietet beide Varianten für den initialen Start des Systems an. Target-Server und Target-Agent erlauben, das eigentliche Betriebsystem über eine Netzwerkschnittstelle zu booten. Des weiteren ist es möglich, die anfangs erwähnten Objektfiles zur Laufzeit in das Betriebssystem einzubinden. Ebenso ist der gezielte Aufruf von implementierten C-Funktionen durch eine Art Interpreter möglich.

Die gesamte Funktionalität wie das Nachladen von Objektfiles, Messung des Ressourcenverbrauchs der laufenden Tasks usw. ist unter einer einheitlichen Bedienoberfläche zusammengefasst.

Während das eigentliche Betriebsystem den Namen VxWorks trägt, wird das gesamte Paket aus Entwicklungsumgebung und RTOS unter der Bezeichnung *Tornado* vertrieben.

5.7. Übungen zur Aufbaustruktur eines Echtzeit-Betriebsystems

a) Definieren Sie bitte nachstehende Begriffe:
   - Technischer Prozess
   - Echtzeit-System
   - Embedded System
   - Echtzeit-Betriebssystem

b) Was versteht man unter der Echtzeitfähigkeit eines Systems?

c) Geben Sie je ein Beispiel für ein „hartes“ und für ein „weiches“ Echtzeitsystem an.

d) Geben Sie die Bestandteile eines Minimal-Echtzeitbetriebsystems an und beschreiben Sie deren Aufgaben.

e) Auf welche Weise können Tasks gestartet werden?

f) Ein Prozessrechner zur Automatisierung eines Hochregallagers hat u. a. die

Aufgabe, die Organisation des Ein- und Auslagerungsablaufes zu steuern. Gehört das Programm, dass diese Organisation durchführt, zu den Anwenderprogrammen, zur Systemunterstützung, zum Laufzeitsystem oder ist dies Teil des Betriebssystems?

g) Stückgutprozess versus Fließgutprozess. Eine Paketverteilanlage soll automatisiert werden. Um welchen Prozesstyp handelt es sich?

h) Komponenten eines Automatisierungssystems. Nennen Sie die Komponenten (sog. Automatisierungsmittel) eines Technsichen Systems und eines Embedded Systems. Grenzen Sie die Begriffe voneinander ab.

i) Verteiltes versus zentrales Prozessrechensystems. Nennen Sie einige Vorteile eines verteilten Prozessrechensystems gegenüber einem zentralen Prozessrechensystem.

j) Auflösung AD-Wandler. Bei einem Digital-Analog-Wandler mit einem Spannungsbereich von 0V bis 10V beträgt der kleinste Ausgabewert 40mV. Wie viele Bits werden zur Ansteuerung des Digital-Analog-Wandlers benötigt?

k) Definieren Sie die Begriffe Rechtzeitigkeit und Gleichzeitigkeit.

l) Was versteht man unter Echtzeit-Programmierung?

m) Aufgabe: Zu welcher Gruppe von Systemprogrammen (Betriebssystem, Laufzeitsystem, Systemunterstützung) würden sie die folgenden Programme/Programmteile rechnen?
   - ein Assembler
   - ein Programm, das die Hintergrund- bzw. die Arbeitsspeicherzuteilung vornimmt und automatisch Daten zwischen diesen Speichern austauscht
   - ein Programm, das den Namen einer Datei auf den Hintergrundspeicher ändert

6. TASKZUSTÄNDE UND TASKMANAGEMENT

- Allgemeine Strukturierung von RTOS Systemen

- Tasks- und MuIti-Tasking
- Definition einer Task
- Was gehört alles zu einer Task, Aufbau des Task Control Blocks (TCB)
- Was heißt eigentlich Reentrant Taks, was sind reentrant EBS?
- Vollständiges Programmbeispiel beispielhaft erläutern

- Taskzustände und Taskwechsel (Scheduler-Algorithmen)
- Zustandsdiagramm erläutern
- Symbole erläutern (Legende)
- Mit Taskzustände
- Und Taskübergänge

- Task-Zustandsdiagramm (siehe Skript Küfner, Roderer, eDonkey Skript, FernUni Hagen etc.)
    - Task nicht existent
    - Task ruhend, stopped, suspended
    - Task bereit, ready, lauffähig, eligible
    - Task wird ausgeführt, laufend, running, executing
    - Task verzögert, delayed
    - Task ist blockiert, pended

- Scheduling Strategien
    - First Come First Serve
    - Shortest Time First
    - Round Robin
    - Priority
    - Deadline

- Beispiel 1: Ablauf der Ausgabe auf einen Drucker
- Beispiel 2: Überprüfung der Echtzeitfähigkeit Interpollationsberechung und Polynomkoeffizientenberechnung
- Beispiel 3: Entwurf eines Task-Zeitdiagramms

- Übung zum Themenbereich Multitasking, Taskzustände und Taskplanung
- Übungen zur Taskplanung
- Aufgabe 1: Übung zum Themenbereich Multitasking, Taskzustände und Taskplanung
- Aufgabe 2: Übung zum Themenbereich Multitasking, Taskzustände und Taskplanung

- Kontext-Switch und Preemption
- Aufsatz: Echtzeit-Betriebssysteme - Mehr Funktionalität
- Multitasking - Taskeigenschaften - Taskzustände -Taskwechsel - Taskplanung
- Von sequentiell bis preemptiv - Einführung in die Welt der Taskscheduler (6 Seiten, 468kB)

- Übungen: Multitasking - Taskzustände - Taskplanung
- Übersicht Realtime Embedded Betriebssysteme

Multitasking - Taskeigenschaften - Taskzustände - TaskWechsel - Taskplanung

6.1. Tasks und Multi-tasking

Multi-Tasking Systeme beinhalten den Begriff Task. Eine Task oder zu deutsch Rechenprozess bildet als organisatorische Einheit den Kern von Echtzeitsystemen. Entsprechend ist das Management der im System vorhandenen Tasks die Hauptaufgabe des Kernels.

Ziel einer effizienten Taskverwaltung ist die rechtzeitige Reaktion auf externe Ereignisse. Das Echtzeitsystem muss deterministisch arbeiten, das bedeutet die Reaktionszeiten oder auch Latenzzeiten müssen reproduzierbar und vorhersagbar sein. Zur Sicherstellung des Determinismus in Echtzeitsystemen ist es notwendig, dass Tasks parallel bearbeitet werden und gleichzeitig aber auch asynchrone Unterbrechungen möglich sind.

Die Zuordnung von Prioritäten zu Tasks ermöglicht die Steuerung des Prozessgeschehens nach den Prozessnotwendigkeiten.

Definition:

Eine Task ist eine organisatorische Einheit, die aus ausführbarem Programmcode, eigenem Speicher und Variablen besteht. Sie ist gekennzeichnet durch eine Priorität und einem Taskzustand.

Begreift man eine Task als lebendige Einheit, so wird klar, dass

eine Task erzeugt werden muss,

eine gewisse Lebensdauer hat und

je nach Notwendigkeit auch wieder aus dem System genommen werden kann.

6.1.1. Der Task Control Block (TCB)

Formal besteht eine Task aus einem sogenannten Task Control Block (TCB), der je nach Echtzeitsystem eine Reihe von Taskvariablen enthält.

Der TCB ist ein Form einer C-Datenstruktur organisiert, die per Task-Identifikationszeiger referenziert werden kann. Hierin sind unter anderem

die Taskoptionen (z. B. Taskpriorität),

Ein- Ausgabezeiger für Standard-I/O,

und die CPU Kontextinformationen wie aktueller Programm- und Stackzeiger, benutzte CPU Register und, falls vorhanden, die Register der Gleitkommaeinheit (FPU) abgespeichert,

Wird eine Task erzeugt, muss der sogenannte Taskstack deklariert werden. Der Taskstack ist ein lokaler Speicherbereich, der ausschließlich von dem Programmcode benutzt wird, der in eben diesem Taskkontext abgearbeitet wird.

In der Regel bleibt die Größe des Stackspeicherbereiches einer Task nach der Deklaration fest. Hierin liegt eine der häufigsten Fehlerquellen. Meist kommt es zu fatalen Systemabstürzen oder auch nicht vorhersagbaren Systemfehlern, wenn eine Task über den ihr zugewiesenen Stack hinaus schreibt und ggf. Daten anderer Tasks somit manipuliert.

Manche Entwicklungssysteme bieten hierzu spezielle Stackmonitore, die online den Stackverbrauch überwachen und auswerten.

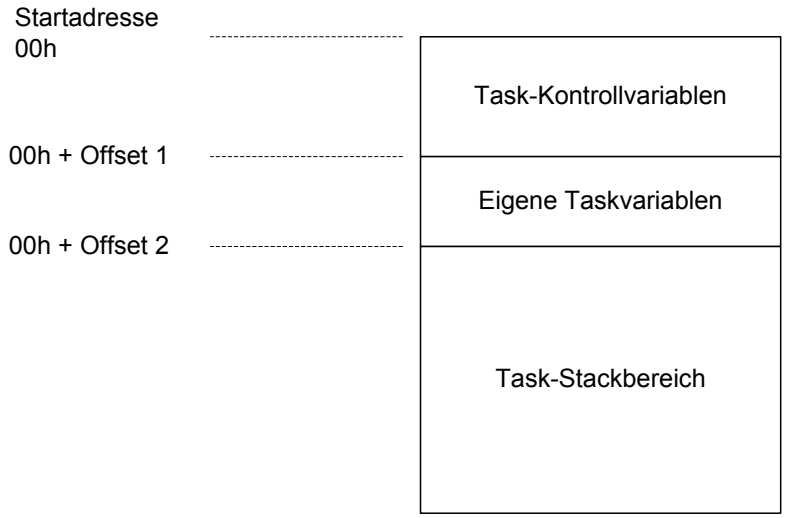


Bild: Speicherorganisation einer Standardtask

In bekannten Betriebsystemen wie z.B. Windows NT oder UNIX besitzt jeder Rechenprozess einen eigenen Speicherbereich, der exklusiv reserviert ist. Dort hält jede Task ihren eigenen Datenbereich (typisch: "data Sektion" für initialisierten Speicher und "bss Sektionen" für uninitialisierten Speicher). Der Programmspeicher (text Sektion) kann von mehreren Prozessen geteilt werden.

In Echtzeitsystemen wie z.B. VxWorks residieren alle Tasks in demselben Speicherbereich. Das bedeutet, alle Sektionen liegen in demselben Speicherbereich.

Damit wird die Taskinterkommunikation sehr stark vereinfacht und der Kontextwechsel verläuft im Prinzip ohne besonderen Adressierungs-Overhead. Der Nachteil liegt im fehlenden Speicherzugriffsschutz.

6.1.2. Reentrant Taks

Eine Besonderheit von Echtzeitsystemen ist, das der Programmcode in einer beliebigen Anzahl von Taskkontexten ablaufen kann. Man bezeichnet diesen Programmmcode als sogenannten "Shared Code", d. h. verschiedene Task benutzen

denselben Code zu unterschiedlichen Zeiten und ggf. auch Aufgaben.

Die gemeinsame Nutzung von einem Programmcodefragment durch mehrere Tasks setzt eine wichtige Eigenschaft voraus. Der gemeinsame Code muss "reentrant" sein, d. h. der gemeinsame Code muss über Mechanismen verfügen, die erlauben, denselben Code gleichzeitig von mehreren Tasks aus zu benutzen.

Hierbei kann es Konflikte geben, wenn der "Shared Code" statische oder globale Variablen verwendet, die von einer Task manipuliert werden. Eine zweite Task kann somit z. B. einen vermeintlich richtigen Wert aus einer Variablen auslesen, die zwischenzeitlich aber bereits geändert wurde.

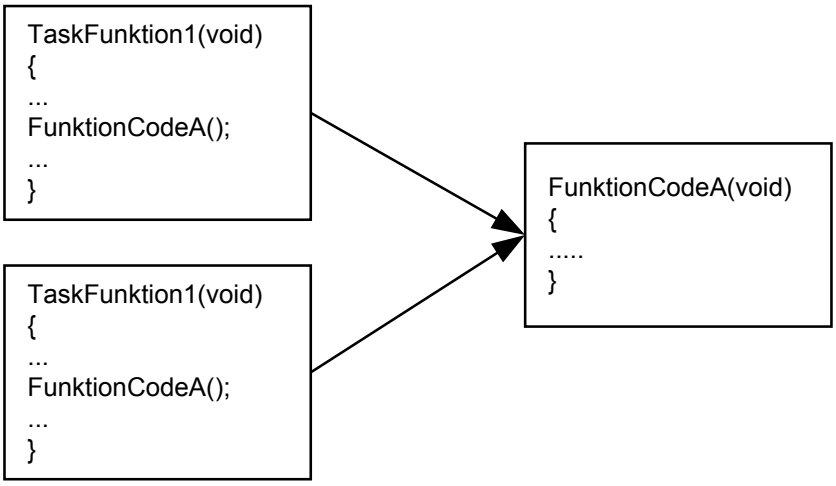


Bild: Gemeinsam benutzter Programmcode

Folgende Vorkehrungen machen einen Programmcode reentrant:

Ausschließliche Verwendung von dynamischen Stackvariablen, d. h. die aufrufende Taskfunktion übergibt die Variable als Zeiger auf den taskeigenen Speicher. Damit ist sichergestellt, dass im gemeinsam benutzten Programmteil keine Variablenkonflikte auftreten.

Schutz der gemeinsamen Programmstücke durch Semaphoren. Semaphoren sind globale Steuervariablen zur Synchronisation von Tasks.

Nutzung von Taskvariablen. Die Taskvariable erlaubt z. B. unter VxWorks die Ergänzung des Taskkontrollblocks um eine 32 Bit Variable, die wie die regulären Taskvariablen bei einem Kontextwechsel automatisch abgespeichert werden und bei Neulauf der Task automatisch restauriert werden. Zur Senkung der Kontextwechselzeiten sollte nur eine Taskvariable benutzt werden. Diese Variable kann dann ein Zeiger sein, der auf eine Struktur zeigen kann.

Reentrante Funktionen werden bei der Programmierung von Funktionen benutzt, die Zugriff auf gemeinsame Ressourcen geben oder z. B. einen gemeinsamen Speicherbereich verwalten. Typischerweise werden Schnittstellenfunktionen als reentrante Funktionen programmiert, indem sie über Semaphoren geschützt werden.

Konkret wird eine Task z.B. in VxWorks folgendermaßen kreiert und aktiviert:

```
Id = taskSpawn (name, priority, options, stacksize, entrypoint, arg1, .. arg10);
```

name	Name der Task, bestehend aus ASCI Zeichen; "tMeine_Task"
priority	in VxWorks wählbar zwischen 0 (höchste) bis 255 (niedrigste) Priorität
options	Optionen z. B. für das "Task-Debugging"
stacksize	Größe des Stackspeichers für eine Task in Bytes
entrypoint	Hier wird der Name der C-Funktion eingetragen, die im Taskkontext ablaufen soll
arguments	Diese Argumente sind die C-Funktionsübergabeparameter der entrypoint-Funktion.

Eine derartig im System deklarierte und aktivierte Task unterliegt der Kernelkontrolle und kann verschiedene Zustände einnehmen.

6.1.3. Eigenschaften einer Task

Eine *Task* besteht aus einer C-Funktion ohne Parameter und einem Stack. Eine Task hat eine Priorität zwischen 0 (höchste) und 255 (niedrigste). Eine hohe Priorität (0) bedeutet hier eine hohe Dringlichkeit. Dabei sind lediglich die relativen Prioritäten innerhalb eines Programms wichtig. Zum Beispiel ist das Verhalten eines Programms mit zwei Tasks identisch, wenn die Prioritäten 1 und 2 sind oder 50 und 60.

Referenziert werden Tasks durch Task-Handles. Diese Task-Handles kann man sich wie File-Handle vorstellen. Beim Erzeugen einer Task übergibt taskspawn(..) an die erzeugende Task (Start- oder Steuertask) ein eindeutiges Handle, durch die die neue Task in Zukunft angesprochen werden kann (z. B. um ihr Daten zu senden).

Auf dem **Stack einer Task** werden alle Variablen abgelegt, die lokal zu der Task-Funktion deklariert sind (nicht jedoch static-Variablen). Das gleiche gilt natürlich auch für die lokalen Variablen aller von dieser Task aufgerufenen Funktionen.

Es können also mehrere Tasks mit der gleichen Task-Funktion gestartet werden; jede erhält dabei aber einen anderen Stack und somit andere lokale Variablen. Verschiedene Tasks können auch ein und dieselbe Funktion aufrufen. Es wird dann zwar derselbe Code ausgeführt, da aber jede Task ihren eigenen Stack hat, kommt es zu keinen Reentrance-Problemen - solange solche Funktionen nur ihre Parameter und lokale Variablen verwenden.

Die üblichen C-Sichtbarkeitsregeln gelten unverändert für Multitasking-Programme. Alle Tasks können auf global deklarierte Daten zugreifen.

Sobald der Echtzeitbetriebssystemkern initialisiert wird, existieren bereits zwei Tasks im Programm: die *Idle Task* und die *Main Task.* Die Idle Task hat die nied-

rigste Priorität (255). Sie läuft immer dann, wenn keine andere Task laufen kann. Ihre Notwendigkeit ergibt sich aus der Arbeitsweise des Schedulers, der immer mindestens eine Task im Zustand Ready benötigt, die er aktivieren kann.

6.1.4. Vollständiges Programmbeispiel

Realisierung eines FIFO (First In First Out)

```
/* fifo.c - demonstrate the system function taskspawn  */
/* one producer task, two consumer tasks */

#include "vxWorks.h"
#include "msgQLib.h"
#include "taskLib.h"
#include "semLib.h"
#include "wvLib.h"
#include "stdio.h"

#define MSGS_MAX (5)            /* queue dimensions */
#define MSG_SIZE_MAX (64)

#define PRI_RECV_LOW (150)     /* task priorities */  /* low priority task */
#define PRI_RECV_MED (125)     /* higher priority task */
#define PRI_SEND     (100)     /* higher priority task */

#define OPTS         (0)
#define STACK_SIZE   (20000)

MSG_Q_ID theQ = NULL;
SEM_ID   syncSem = NULL;

int tSend;                  /* Task IDs */
int tRecvLow;
int tRecvMed;

char message [] = "My bonnie lies over the ocean.";

int countLow;
int countMed;

LOCAL BOOL started = FALSE;

LOCAL void send (void);             /* Task entry point functions */
LOCAL void recvLow (void);
LOCAL void recvMed (void);


STATUS fifoQStart (void)
   {
   if (started)
       {
       printf ("Already started.\n");
       return ERROR;
       }

   countLow = 0;
```

```
    countMed = 0;

    if (theQ == NULL)
        theQ = msgQCreate (MSGS_MAX, MSG_SIZE_MAX, MSG_Q_FIFO);

    if (syncSem == NULL)
        syncSem = semBCreate (SEM_Q_FIFO, SEM_EMPTY);

    if (theQ == NULL || syncSem == NULL)
        {
        printf ("Couldn't start demonstration.\n");
        return ERROR;
        }

    if (   (tRecvLow = taskSpawn ("tRecvLow", PRI_RECV_LOW, OPTS, STACK_SIZE,
                                  (FUNCPTR) recvLow, 0,0,0,0,0,0,0,0,0,0))
           == ERROR
        || (tRecvMed = taskSpawn ("tRecvMed", PRI_RECV_MED, OPTS, STACK_SIZE,
                                  (FUNCPTR) recvMed, 0,0,0,0,0,0,0,0,0,0))
           == ERROR
        || (tSend =    taskSpawn ("tSend", PRI_SEND, OPTS, STACK_SIZE,
                                  (FUNCPTR) send,    0,0,0,0,0,0,0,0,0,0))
           == ERROR)
        {
        printf ("Couldn't spawn tasks.\n");
        return ERROR;
        }

    started = TRUE;
    return OK;
    }


STATUS fifoStop (void)
    {
    char dummy [1];

    if (!started)
        {
        printf ("Not started.\n");
        return ERROR;
        }

    taskDelete (tSend);                 // eliminate the task
    taskDelete (tRecvMed);
    taskDelete (tRecvLow);

    while (semTake (syncSem, NO_WAIT) == OK)
        ;

    while (msgQReceive (theQ, dummy, 1, NO_WAIT) != ERROR)
        ;

    started = FALSE;
    return OK;
    }


LOCAL void send (void)
```

```
    {
    FOREVER
        {
        taskDelay (1);

msgQSend (theQ, message, sizeof (message), MSG_PRI_NORMAL, WAIT_FOREVER);

        semGive (syncSem);
        }
    }


LOCAL void recvLow (void)
    {
    char msg [MSG_SIZE_MAX];

    FOREVER
        {
        msgQReceive (theQ, msg, sizeof (msg), WAIT_FOREVER);

        ++countLow;

        wvEvent (1, (char *) &countLow, sizeof (countLow));

        printf ("Message received: %s\n", msg);
        }
    }


LOCAL void recvMed (void)
    {
    char msg [MSG_SIZE_MAX];

    FOREVER
        {
        semTake (syncSem, WAIT_FOREVER);

        msgQReceive (theQ, msg, sizeof (msg), WAIT_FOREVER);

        ++countMed;

        wvEvent (2, (char *) &countMed, sizeof (countMed));
        }
    }
```

6.2. Taskzustände

Bezüglich des taskeigenen Speicherbereichs haben sich spezifische Begriffe eingebürgert. Ein Rechenprozess im Betriebssystem UNIX besitzt einen festen, eigenen und geschützten Speicherbereich. Hier steht die Prozesssicherheit und weniger die Ausführungsgeschwindigkeit im Vordergrund. Dort ist es möglich, im Kontext einer Task quasi "Untertasks" zu erzeugen, die keinen eigenen Speicher bekommen, sondern im Bereich der übergeordneten Task residieren. Diese Unterprozesse ohne dedizierten Speicher bezeichnet man als "Threads".

Da z. B. in VxWorks alle Tasks in einem gemeinsamen nicht geschützten Speicher residieren, könnte man VxWorks-Tasks generell als Threads bezeichnen. Für den weiteren Sprachgebrauch werden alle Rechenprozesse ungeachtet der Speicherzuteilung und Schutzmechanismen als Tasks bezeichnet.

Eine Eigenheit von Tasks sind ihre spezifischen Zustände. Hierzu existieren verschiedene Modelle. Gemeinhin werden folgende Zustände unterschieden:

Zustand N non existent

Die Einführung einer Task kann als der Übergang des Prozesses aus dem Pseudozustand N in den Zustand ruhend („suspended“) erklärt werden. Man spricht hier von dem Erzeugen einer Task.

Zustand ruhend (suspended, eingeplant, dormant, inaktiv):

Die Task ist dem Echtzeitbetriebssystem bekannt, aber noch nicht bereit (ready). Die Task kann jedoch aus diesem Zustand heraus nicht ausgeführt werden. Dies ist der erste Zustand im Lebenszyklus einer Task.

Eine Task ruht solange, bis sie durch eine Anforderung (z. B. Zeitanforderung) gestartet wird; sie gelangt dann in den Zustand „bereit“.

Zustand bereit (ready, runnable, lauffähig, bereit):

Die Task ist ablaufbereit und wartet auf die Prozessorzuteilung. Diejenige Task, die beim folgenden Scheduleraufruf die höchste Priorität besitzt und gleichzeitig im Zustand „ready“ ist bekommt die CPU zugeteilt.

Alle Tasks im Zustand „bereit“ haben in der Regel die gleichen - oder niedrigere Prioritäten wie die aktive Task.

Zustand laufend (running, current, aktiv):

Die gerade aktive Task ist im Zustand „laufend“ und wird von der CPU abgearbeitet. Bei Echtzeitbetriebssystemen hat immer genau eine Task diesen Zustand (eventuell die Idle Task).

Die Task wird vom Prozessor bearbeitet,

- bis die Programmabarbeitung beendet ist,
- oder das Programm auf eine Wartebedingung stößt,
- oder die Task durch eine andere höherpriore Task (ISR) verdrängt wird.

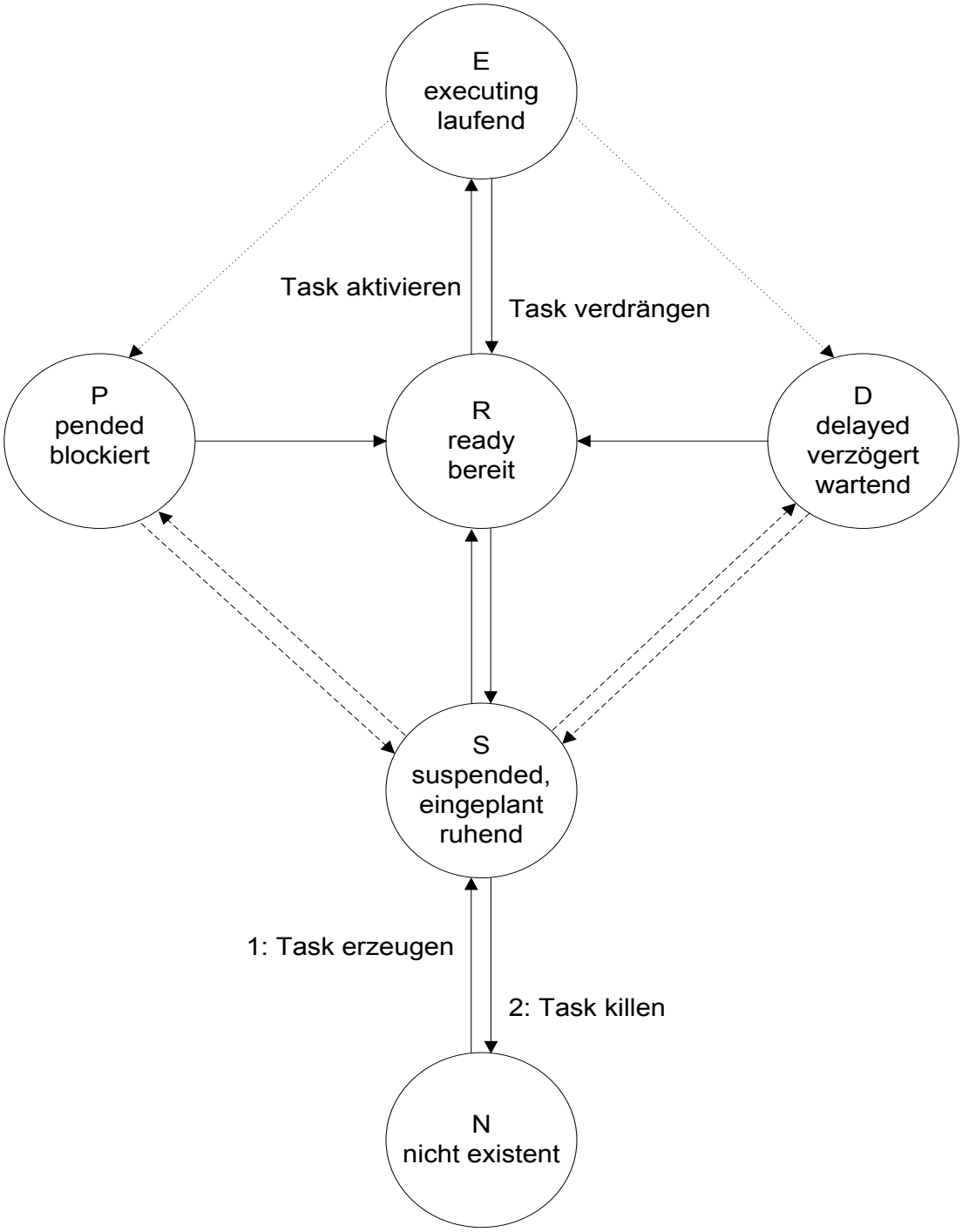


Bild: Zustandsmodell für Rechenprozesse in Echtzeitbetriebssystemen

Zustand wartend (delayed, verzögert):

Diese Tasks haben sich selbst für eine bestimmte Zeit in einen Wartezustand versetzt. Sie werden nach Ablauf ihrer Suspendierzeit (warten auf TimeOut) automatisch durch den Timer-Interrupt-Handler des Echtzeitbetriebssystemkerns wieder in den Zustand „bereit“ versetzt.

Zustand blockiert (pended, blockiert):

Diese Tasks können nicht laufen, weil sie auf ein Ereignis warten, z. B.

- auf ein Semaphore-Signal (zur Synchronisation zweier Tasks),
- oder auf das Eintreffen einer Message in einer Mailbox,
- auf die Fertigstellung einer Analog-Digital-Umsetzung,
- oder auf die Freigabe einer Systemressource, die gerade von einer anderen Task benutzt wird.

Diese Tasks können nur durch eine andere Task oder durch einen Interrupt-Handler wieder in den Zustand „bereit“ gebracht werden. Die Task bleibt solange im Zustand blockiert, bis alle Wartebedingungen erfüllt sind.

Tasks, die gerade nicht laufen, werden von RTKernel in verschiedenen Warteschlangen verwaltet. So gibt es z. B. eine Warteschlange für alle bereiten Tasks. Eine andere Warteschlange enthält alle Tasks, die auf einen bestimmten Zeitpunkt warten. Auch an Semaphoren, Mailboxen oder Tasks können sich Warteschlangen aufbauen, falls sich Tasks an diesen blockieren.

6.3. TaskWechsel (Scheduler-Algorithmen)

6.3.1. Taskübergänge

Taskzustände Kernelfunktionen, die den Übergang auslösen

von Zustand		**in Zustand**		**mit Kernelfunktion**
Laufend	->	Bereit	->	höherpriore Task kommt zur Ausführung
Laufend	->	Blockiert	->	semTake()
Laufend	->	Verzögert	->	taskDelay()
Bereit	->	Laufend	->	Task besitzt höchste Priorität
Bereit	->	Blockiert	->	semTake()
Bereit	->	Verzögert	->	taskDelay()
Bereit	->	Eingeplant	->	taskSuspend()
Pended	->	Bereit	->	semGive()
Pended	->	Eingeplant	->	taskSuspend()
Verzögert	->	Bereit	->	Verzögerungszeit verstrichen
Verzögert	->	Eingeplant	->	taskSuspend()
Eingeplant	->	Bereit	->	taskResume()
Eingeplant	->	Blockiert	->	taskResume()
Eingeplant	->	Verzögert	->	taskResume()

6.3.2. Scheduling Algorithmen

Kern des Taskmanagements ist der Scheduler mit seiner "Scheduling-Policy" d. h. der Taskaktivierungsstrategie. Das smit ist die Art und Weise gemeint, wie die "bereiten" Tasks in den Zustand "laufend" überführt werden. Es haben sich hier zwei Strategien etabliert:

- Prioritätsbasiertes preemptives Scheduling
- Round-Robin Scheduling

Prioritätsbasiertes preemptives Scheduling

Diese Strategie basiert auf den benutzerdefinierten Taskprioritäten. Die Priorität einer Task repräsentiert dabei die Wichtigkeit der jeweiligen Aufgabe, die mit der Task gelöst wird. Grundsätzlich kommt bei prioritätsbasierten Systemen immer die Task zur Ausführung, die zum Zeitpunkt der Einplanung (Scheduling) die aktuelle höchste Priorität besitzt. Der Aufruf des Schedulers erfolgt entweder nach einem Kernel-Funktionsaufruf oder nach Eintritt von Interrupts.

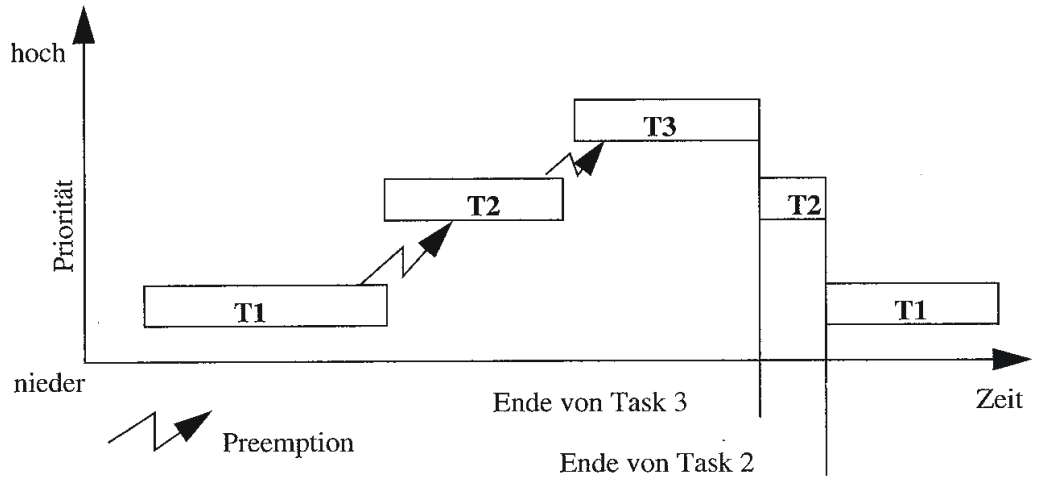


Bild: Prioritätsbasiertes preemptives Scheduling

Round-Robin Scheduling

Bei dieser Strategie ist das Ziel, die CPU Leistung möglichst gleichmäßig auf alle Tasks im System zu verteilen. Alle Tasks besitzen die gleiche Priorität. Grundlage der Round-Robin Technologie ist ein Zeitscheibensystem. Dabei wird jeder Tasks eine Zeitscheibe von definierter Länge zugeteilt. Erst wenn alle Tasks abgearbeitet sind kommt dieselbe Task im System wieder an die Reihe.

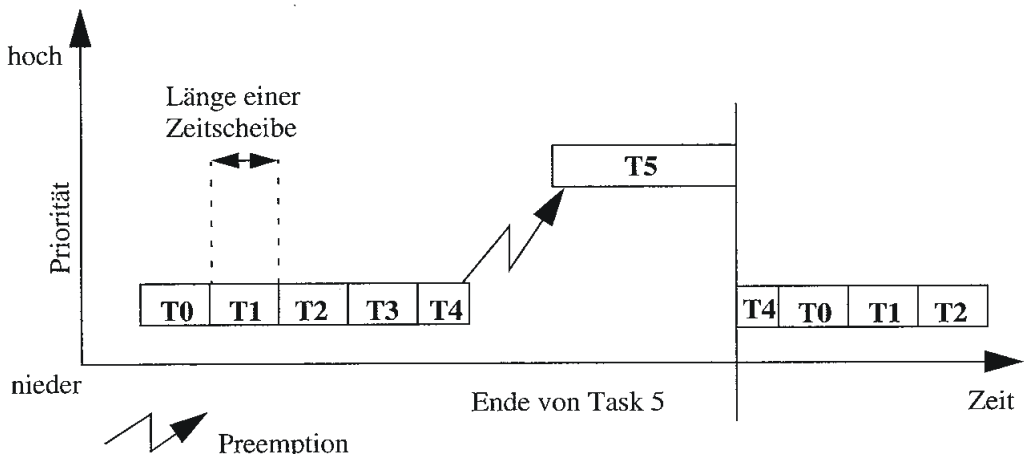


Bild: Round-Robin Scheduling

Bestimmte Systeme wie z. B. VxWorks lassen einen gemischten Betrieb zu. Dabei werden Gruppen von Tasks mit derselben Priorität deklariert. Über einen Kemelaufruf kann für diese Gruppe von Tasks die Round Robin Strategie aktiviert werden.

Wird eine Round Robin Tasks von einer höher prioren Task "preempted", wird die Round-Robin Task (T4) nach der Beendigung der höher prioren Tasks (T5) vollends abgearbeitet, bis die Zeitscheibe aufgebraucht ist.

Die Behandlung von Tasks innerhalb von Echtzeitsystemen hängt sehr stark vom jeweiligen Echtzeitbetriebssystem ab. Je nach Komfort stehen mehr oder weniger Funktionsaufrufe zur Verfügung, die im System vorhanden Tasks zu administrieren. In kompletten Systemen wie z.B. VxWorks steht ein recht umfassendes Instrumentarium zur Verfügung.

6.3.3. Kernelfunktionen zum Taskmanagement unter VxWorks

Tasks besitzen Namen und interne Identifier, auf die wie folgt zugegriffen werden kann:

taskN ame(taskID) // liefert den Namen zurück, der zu der entsprechenden ID gehört

taskNameToId(taskName) // liefert die TaskId, die zu dem Namen gehört

taskIdSelf(taskName) // liefert die taskId der aufrufenden Task zurück

taskIdVerify(taskID) // prüft, ob die Task noch im System vorhanden ist

Obige Funktionsaufrufe dienen der Taskadministration.

Beispielszenario:

Stellt man sich vor, dass eine Task temporär im System erzeugt wird, um eine bestimmte Aufgabe wie z. B. Initialisierung von Systemressourcen auszuführen, so ist

es durchaus üblich, diese Task nach Erledigung ihrer Aufgabe wieder aus dem System zu nehmen. Dies ist bei Systemen mit beschränktem Speicher oft sogar notwendig.

Die Realisierung wird so aussehen, dass eine Steuertask diese Initialisierungstask startet. Da es sich bei der Taskerzeugung um einen dynamischen Vorgang handelt, wird jeder neu erzeugten Task eine fortlaufende ID zugeordnet, die erst nach der Erzeugung bekannt ist. Soll sich die Initialisierungstask nach getaner Arbeit selbst beenden, so besorgt sie sich Ihre ID und setzt damit den Befehl zur Taskbeendigung ab. Die übergeordnete Steuertask kann über den Verifikationsaufruf prüfen, ob die Initialisierung ordnungsgemäß durchgelaufen ist.

```
steuerTask(void)
{
   ……
   taskIniID = taskSpawn (name, 100, 0, 20000, myInitTask, 0,. .0)    // Task einplanen
   ……
   status = taskIdVerify(taskIniID);
}

myInitTask(void)
{
  // initialisiere System
  .......
  // loesche dich selbst
  id = taskIdSelf();
  taskDelete(id) ;
}
```

Die Bibliotheksfunktionen zur Überführung von Tasks zwischen den verschiedenen Zuständen sind u. a.

taskDelay(numberOfTicks)

führt dazu, dass die rufende Funktion für die Dauer der Anzahl Systemtakte (Ticks) blockiert wird. Die Ausplanung der Tasks führt zwangsweise zum Aufruf des Schedulers. Damit ist es möglich, gezielt niederprioren Tasks für eine bestimmte Dauer Rechenzeit zu gewähren.

taskLock()

Wird dieser Befehl innerhalb einer Task abgesetzt, so kann diese Task von keiner anderen Task im gesamten System preempted werden. Dieser Befehl kommt einer Anhebung der Priorität auf die höchste Priorität gleich. Einzig und allein Interrupts können diese Task unterbrechen.

Eine interessante Hilfe zur Realisierung eines vorgegebenen Taskablaufes ist die gezielte Beeinflussung der Taskpriorität. Grundsätzlich wird die Priorität zum Erzeugungszeitpunkt mit dem Befehl taskSpawn(,.. ,..) fest vergeben. Wird zur Laufzeit z.

B. festgestellt, dass die Task eine zu geringe Priorität besitzt, d. h. zu wenig CPU Zeit zugeteilt bekommt, kann die Priorität jederzeit verändert werden.

Die Befehle lauten:

taskPriorityGet() // holt sich die aktuelle Priorität

taskPrioritySet() // setzt eine neue Priorität

6.3.4. Analyse der Abläufe von Kernelfunktionen und Tasks

Zur Veranschaulichung des Zusammenwirkens von Tasks und Kernelfunktionen (sog. Supervisorprozessen) dient das Task/Zeitdiagramm in dem folgenden Bild.

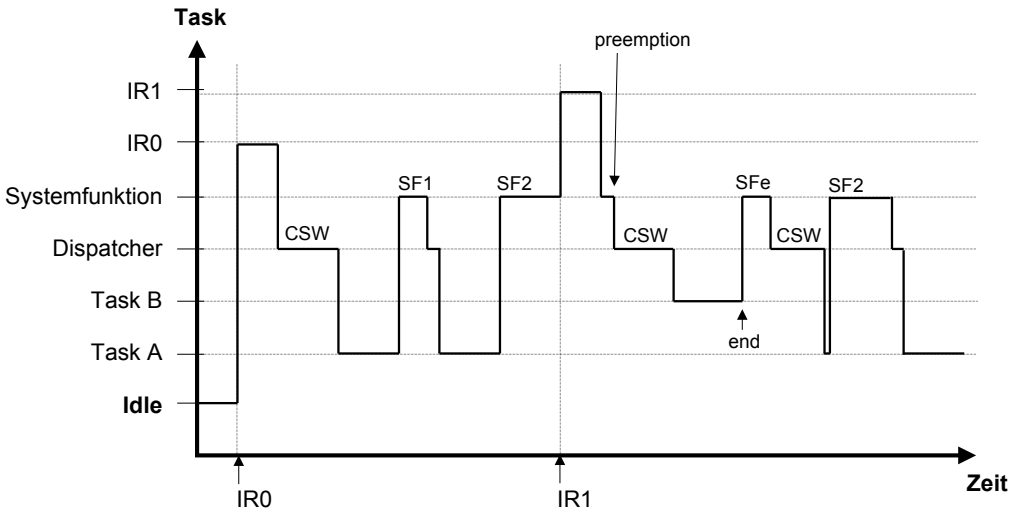


Bild: Task / Zeitdiagramm

Dargestellt ist ein denkbarer Ablauf verschiedener Tasks über der Zeit. Auf der vertikalen Achse sind die einzelnen Prozesse sortiert nach ihrer Priorität aufgeführt. Ein horizontaler Strich in dem Diagramm gibt zu jedem Zeitpunkt an, welche Task der Prozessor gerade bearbeitet.

Zu Anfang der Darstellung hat das System gar keine Aufgabe zu erfüllen. Es stellt den Prozessor dann der stets lauffähigen Task "Idle" zur Verfügung. Sie besitzt die kleinste vorkommende Priorität und kann daher nie eine der anderen Tasks behindern.

Tritt der Interrupt IRO auf, wird die zugehörige Serviceroutine, hier Task A, lauffahig gemacht und der Dispatcher aktiviert. Dieser stellt fest, dass eine andere Task als bisher den Prozessor erhalten soll und führt daher einen Kontextswitch (csw) aus. Die nun laufende Task A ruft nun eine Systemfunktion SF1 auf. SF1 übergibt nach Erledigung ihrer Aufgabe erneut an den Dispatcher, der in diesem Falle feststellt, dass Task A weiterlaufen kann und deshalb kein Kontextswitch notwendig ist.

Zu beachten ist, dass die Systemfunktion den Auftrag von Task A mit einer Priorität verrichtet, die einer Task auf anderem Wege nicht zugänglich ist. Scheinbar erhält Task A damit für die Dauer von SF1 eine höhere Priorität als alle anderen im System befindlichen Tasks.

Eine Systemfunktion kann allerdings viel umfangreicher und damit zeitaufwendiger ausfallen als die in Echtzeitbetriebssystemen im allgemeinen sehr kurzen Interruptroutinen. Ein Problem tritt dann auf, wenn, wie in der Abbildung bei SF2 dargestellt, ein Interrupt IR1 auftritt, der eine Task B aktiviert, die aufgrund ihrer Priorität den Vortritt vor Task A hat.

Preemption

Der Vorgang, mit dem das System auch in dieser Situation eine prioritätsgerechte Bearbeitung der einzelnen Tasks ermöglicht, nennt sich "Preemption". Das bedeutet, dass jede Systemfunktion so zu gestalten ist, dass sie in regelmäßigen Abständen prüft, ob in der Zwischenzeit ein Interrupt aufgetreten ist. War dies der Fall, so muss die Systemfunktion selbst unterbrochen werden und so den Prozessor für die dringlichere Task B frei machen. Da SF2 ihre Aufgabe zu diesem Zeitpunkt noch nicht erledigt hat, die ordnungsgemäße Funktion von SF2 aber nur dann gesichert ist, wenn SF2 in einem Durchlauf durchgeführt wird, muss Task A in diesem Moment in den Zustand zurückversetzt werden, den sie vor dem Aufruf von SF2 hatte.

Anschließend führt wieder der Dispatcher einen Kontextswitch aus und teilt den Prozessor der Task B zu. Am Ende von Task B führt dieser den Befehl "end" aus und aktiviert dadurch die Systemfunktion SFe, welche schließlich in den Dispatcher mündet.

Der Dispatcher lädt jetzt den gespeicherten Kontext von Task A zurück und teilt ihr den Prozessor zu. Task A ruft die Systemfunktion SF2 dann ein zweites Mal auf, die in diesem Beispiel sodann ungestört durchläuft.

Die Zeitspanne, die ein Echtzeitbetriebssystem bei Auftreten eines Interrupts auch im ungünstigsten Fall bis zur prioritätsgerechten Verteilung des Prozessors benötigt, bestimmt entscheidend die Leistungsfähigkeit des Systems. Um den Preemption-Mechanismus vollständig zu realisieren, muss jede einzelne Systemfunktion, deren Umfang einige wenige Befehle überschreitet, an geeigneten Stellen mit entsprechenden Abfragen nach erfolgten Interrupts ausgestattet sein.

Im Detail geht das System bei der Verwirklichung der Preemption sogar noch einen Schritt weiter: stellt der Dispatcher fest, dass der Interrupt, der den Abbruch einer Systemfunktion hervorgerufen hat, nicht zu einem Taskwechsel geführt hat, so ist das System in der Lage, die unterbrochene Funktion an der gleichen Stelle wieder aufzunehmen. Ein Verfall der bereits erledigten Teilaufgabe findet dann nicht statt.

6.4. Übung 1: Zeitliches Sollverhalten von Rechenprozessen

Gegeben sind die zyklischen Rechenprozesse RP1 - RP4. Die Rechenprozesse werden auf einem Rechner abgearbeitet, dessen Uhrimpuls-Takte im Zeitabstand T eintreffen. Weitere Daten zu den Rechenprozessen sind der nachstehenden Tabelle zu entnehmen.

Rechenprozess	Ausführungsdauer	Zykluszeit	Priorität
RP1	T/2	T	1
RP2	T/4	2T	2
RP3	T/2	3T	3
RP4	3T/4	2T	4

Tabelle 1: Charakteristische Daten der Rechenprozesse RP1 - RP4

a) Fertigen Sie ein Zeitdiagramm des Sollverhaltens und des tatsächlichen Ablaufs der Rechenprozesse an (Darstellung vom normierten Zeitpunkt t/T=0 bis t/T=11). Verwenden Sie das vorbereitete Lösungsblatt (Bild 1 und 2).

b) Tragen Sie den zeitlichen Verlauf der Taskzustände von Rechenprozess 4 in Bild 3 ein.

c) Wie beurteilen Sie aufgrund von Frage a) die Ausführung von RP4 unter dem Gesichtspunkt der Rechtzeitigkeit? Begründung.

d) Wie beurteilen Sie die Ausführung eines zusätzlichen Rechenprozesses RP5 (Ausführungsdauer = T/10, Zykluszeit = 5T, Priorität = 5) unter dem Gesichtspunkt der Rechtzeitigkeit?

e) Worin liegt der Grund für Ihre unter Frage c) und d) gegebenen Beurteilungen? Berechnen Sie die durchschnittliche Summe aller Taskausführungszeiten nach folgender Formel

$$\text{Summe aller Taskausführungszeiten} = \sum_{i=1}^{n} \frac{Ausführungsdauer}{Zykluszeit} = \sum_{i=1}^{n} \frac{t_i}{T_{Zyklus}}$$

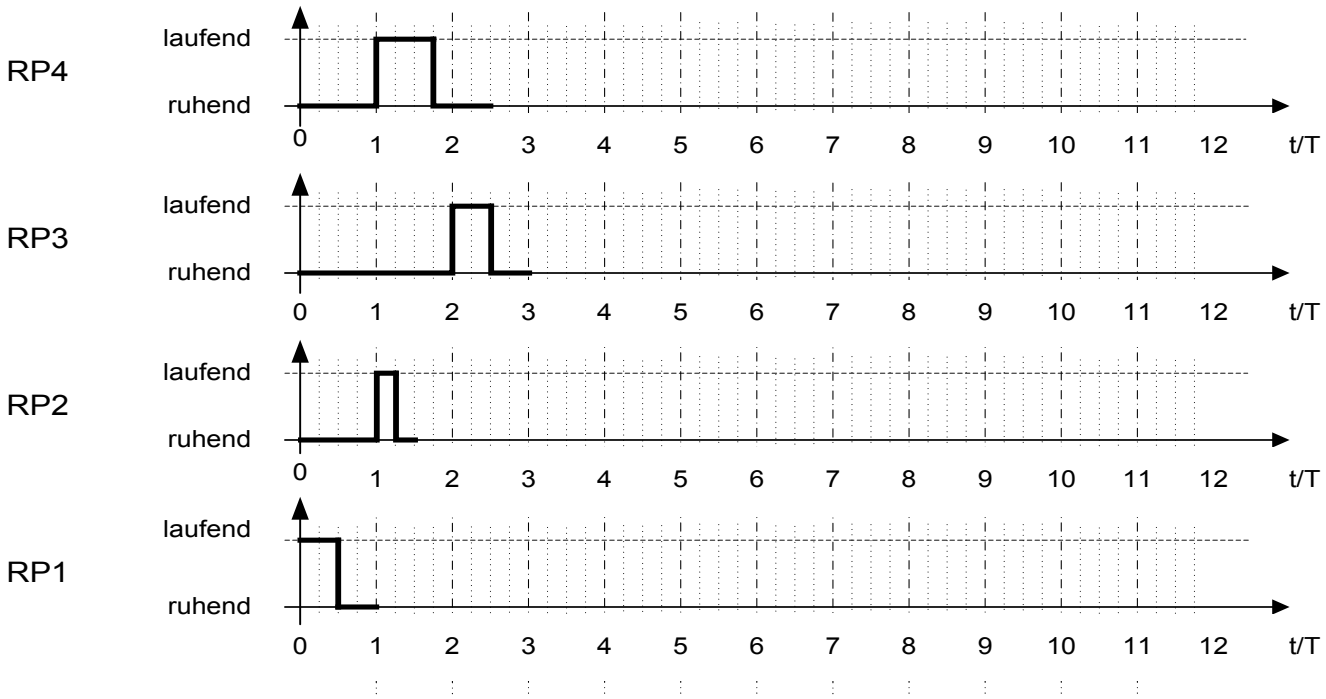


Bild 1: Zeitliches Sollverhalten der Rechenprozesse

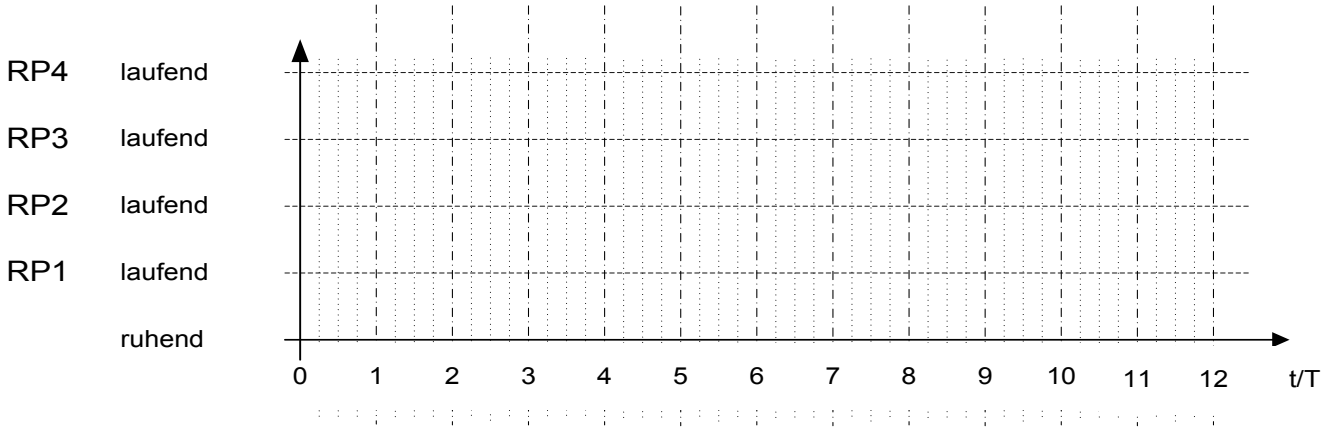


Bild 2: Tatsächlicher Ablauf der Rechenprozesse

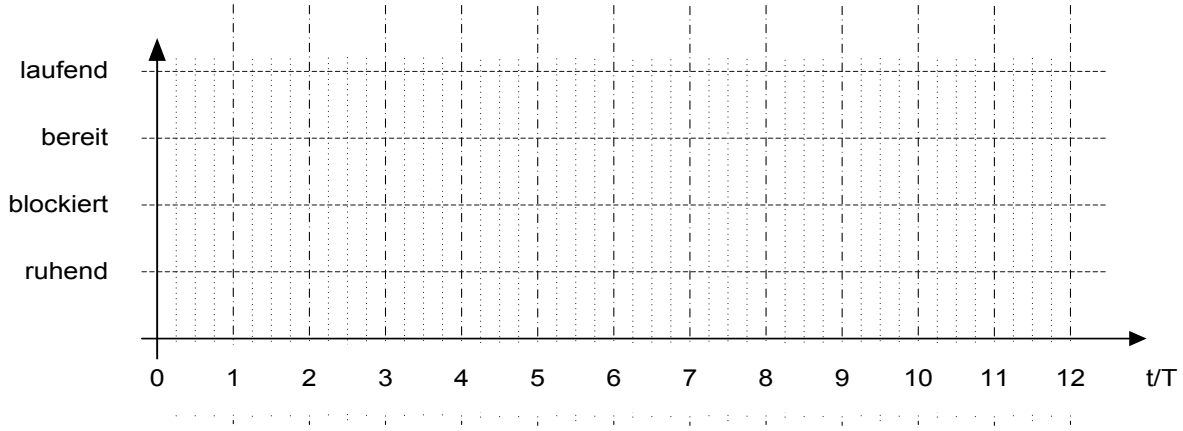


Bild 3: Taskzustände des Rechenprozesses RP4

6.5. Übung 2: Multitasking - Taskzustände - Taskplanung

In dem folgenden Bild 2.1 ist das zeitliche Soll-Verhalten von 4 Rechenprozessen RP1 - 4 dargestellt.

a) Tragen Sie in Bild 2.2 die tatsächliche Abarbeitung der Rechenprozesse ein (bis zum normierten Zeitpunkt t/T=10).

b) Tragen Sie den zeitlichen Verlauf der Taskzustände von Rechenprozess 4 in Bild 2.3 ein.

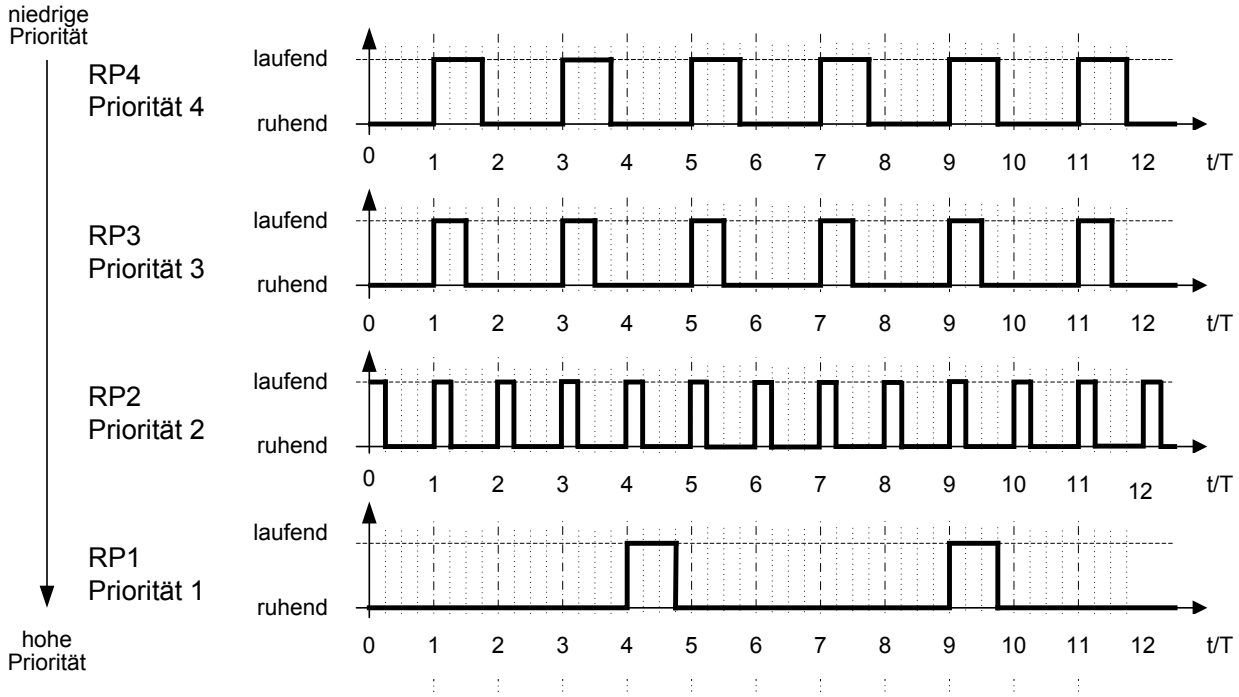


Bild 2.1: Zeitliches Sollverhalten der Rechenprozesse

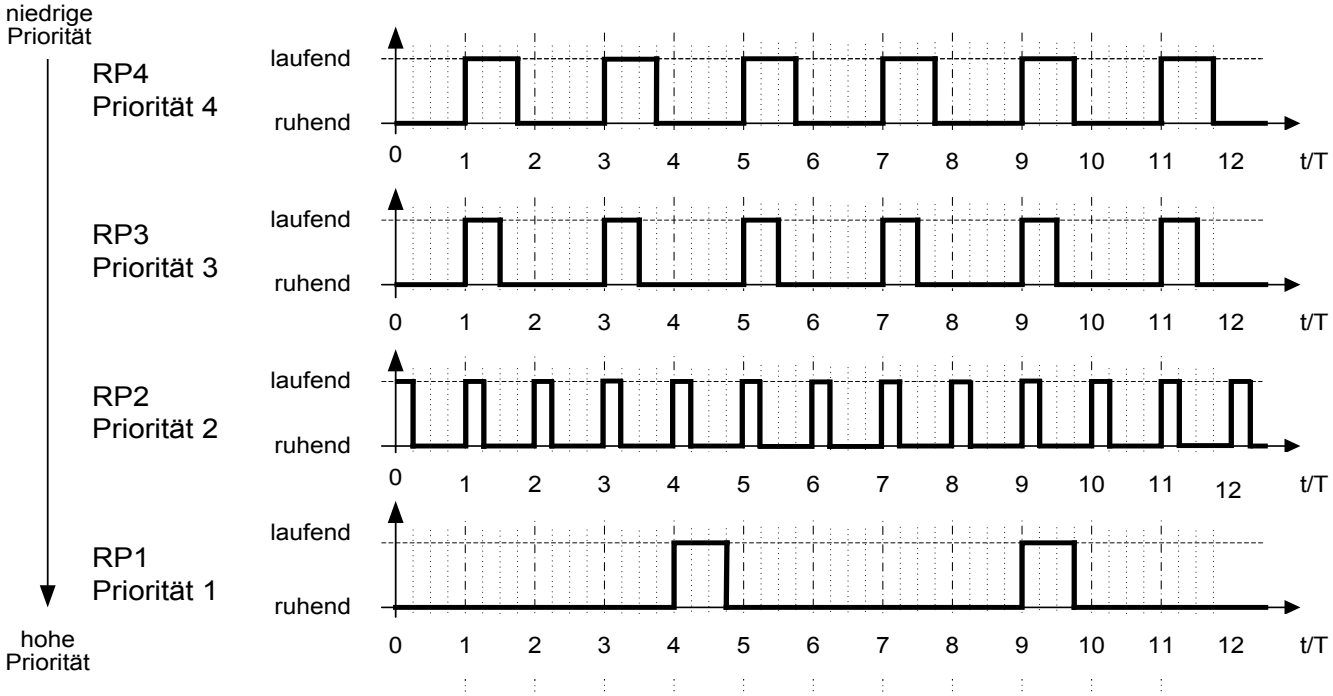


Bild 2.1: Zeitliches Sollverhalten der Rechenprozesse

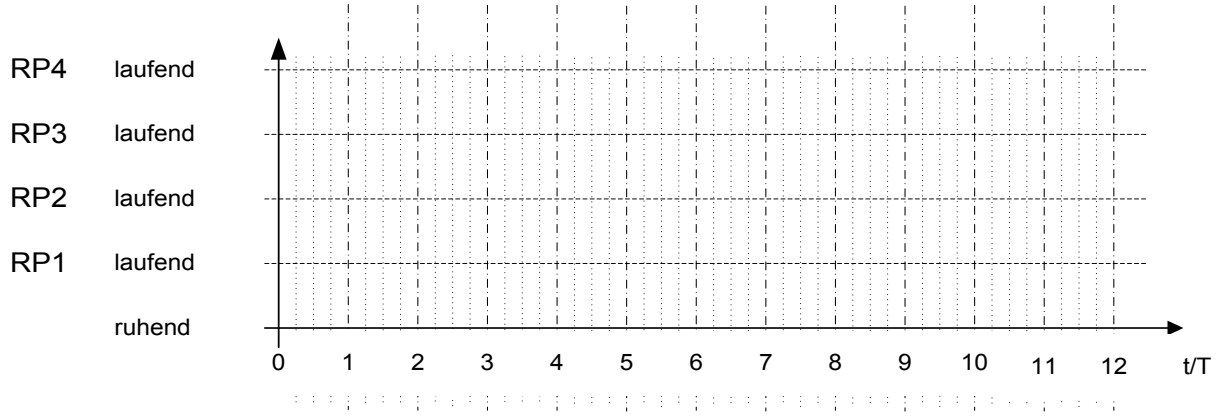


Bild 2.2: Tatsächlicher zeitlicher Ablauf der Rechenprozesse

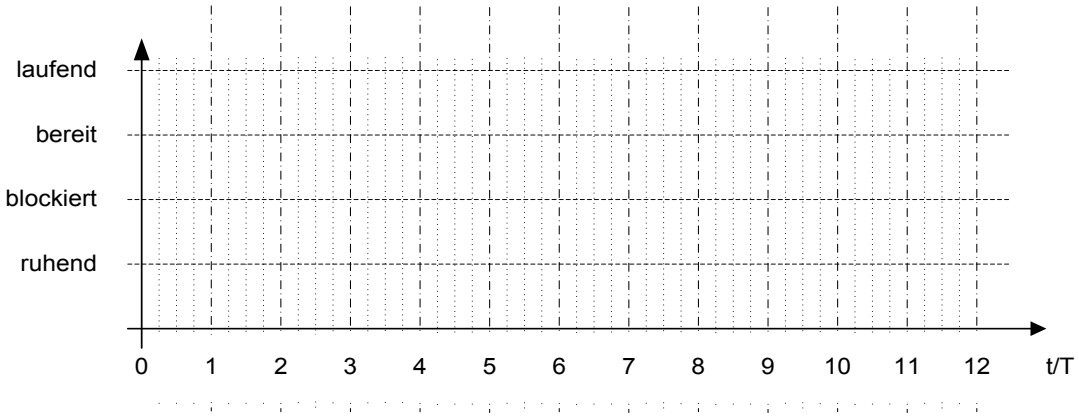


Bild 2.3: Taskzustände des Rechenprozesses RP4

6.6. Übung 3: Überprüfung der Echtzeitfähigkeit

Eine numerische Bahnsteuerung soll jede Millisekunde eine Koordinate $f(x_j)$ ausgeben. Als Steuerdaten erhält sie alle 10 ms eine Koordinate x_j als Stützstelle für ein Polynom zweiten Grades mit $f(x) = ax^2 + bx + c$.

Die Bestimmung eines Zwischenwertes (jede 1 ms) benötigt 0,2 ms.

Die Bestimmung neuer Polynomkoeffizienten (alle 10 ms) benötigt 2,0 ms.

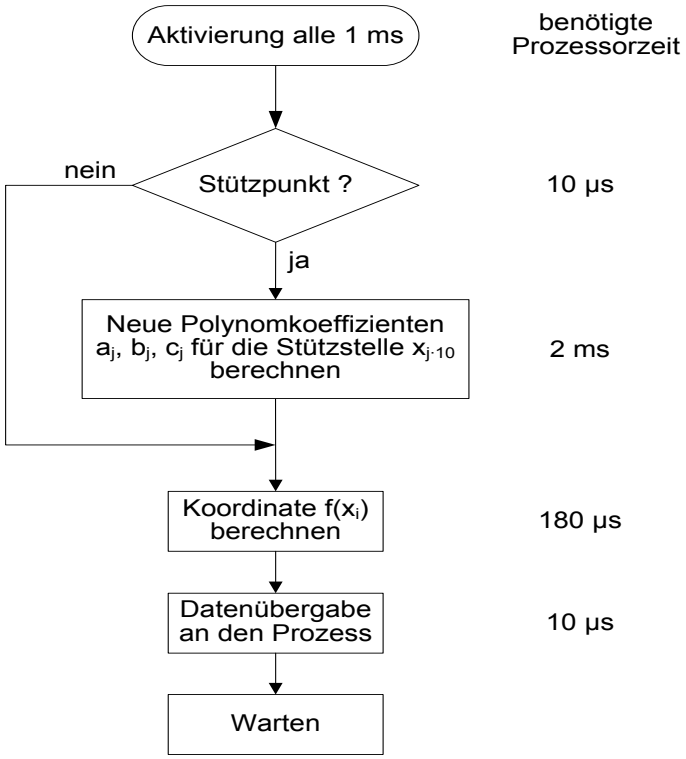


Bild: Ablaufdiagramm

Aufgaben:

a) Stellen Sie obiges Ablaufdiagramm an einem Zeitstrahl dar.

b) Wie groß ist die Auslastung des Prozessors?

c) Sind Echtzeitbedingungen des Prozesses verletzt?

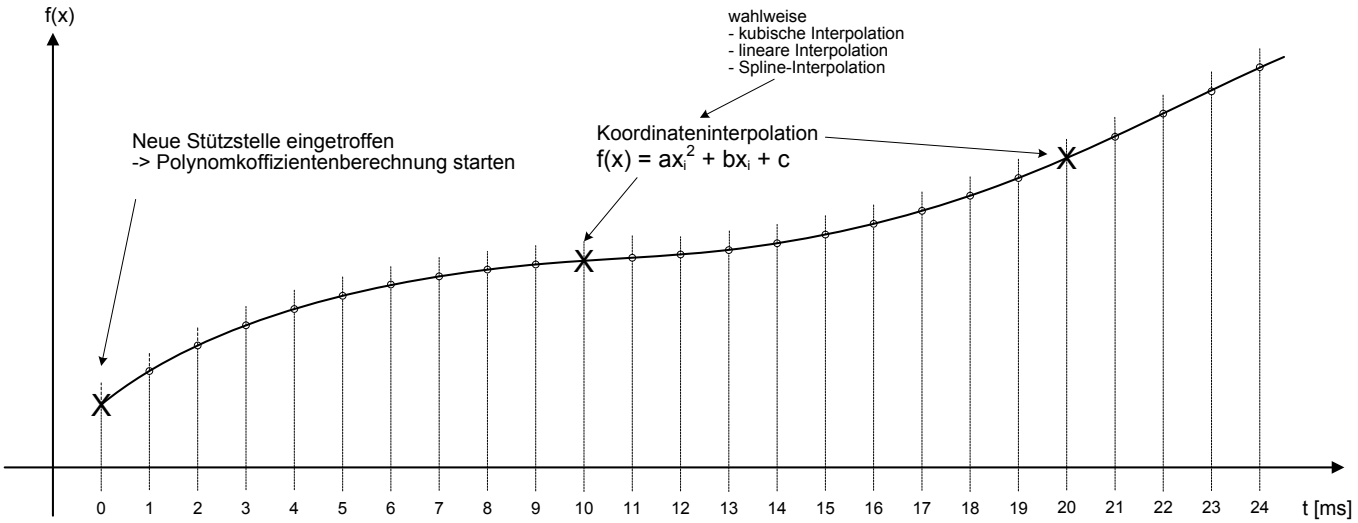
d) Durch welche Maßnahme kann die Situation verbessert werden?

e) Zeichnen Sie die Ablaufdiagramme des verbesserten Programms an einem Zeitstrahl.

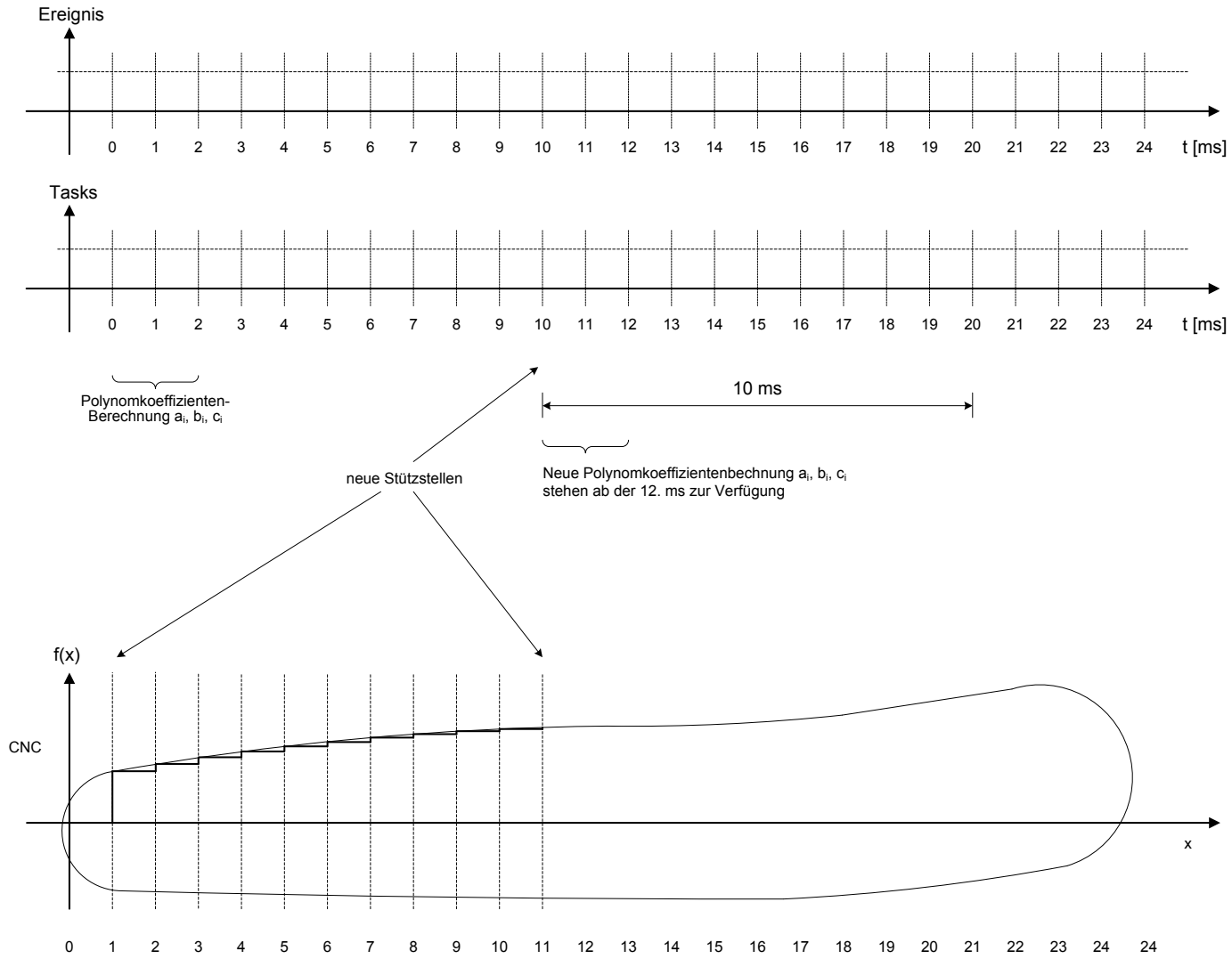
f) Zeichnen Sie die Taskdiagramme des verbesserten Programmes

g) Wie groß ist nun die Auslastung des Prozessors

Übung: Überprüfung der Echtzeitfähigkeit



Ablaufdiagramm (IST-Verhalten) am Zeitstrahl (Anm.: 1-Prozessorsystem)



7. TASKSYNCHRONISATION

Wann wird eine Synchronisation von Tasks benötigt?
3 Arten von Semaphoren

- Binäre Semaphoren
- Mutual Exclusion Semaphoren (Gegenseitiger Ausschluss)
- Counting Semaphore

Binäre Semaphoren

- Anwendungsproblem
- Programmtechnische Lösung des Problems mit Semtake() und SemGive()

Zählende Semaphoren (Counting Semaphoren)

- Anwendungsproblem

Mutual Exclusion Semaphore (Gegenseitiger Ausschluss)

- Anwendungsproblem
- Programmtechnische Lösung des Problems (siehe auch VxWorks Handbuch)

7.1. Problemstellung und Begriffe

Semaphore Control Routines

Call	Description
semBCreate( )	Allocates and initializes a binary semaphore.
semMCreate( )	Allocates and initializes a mutual-exclusion semaphore.
semCCreate( )	Allocates and initializes a counting semaphore.
semDelete( )	Terminates and frees a semaphore.
semTake( )	Takes a semaphore.
semGive( )	Gives a semaphore.
semFlush( )	Unblocks all tasks that are waiting for a semaphore.

7.2. Binäre Semaphoren

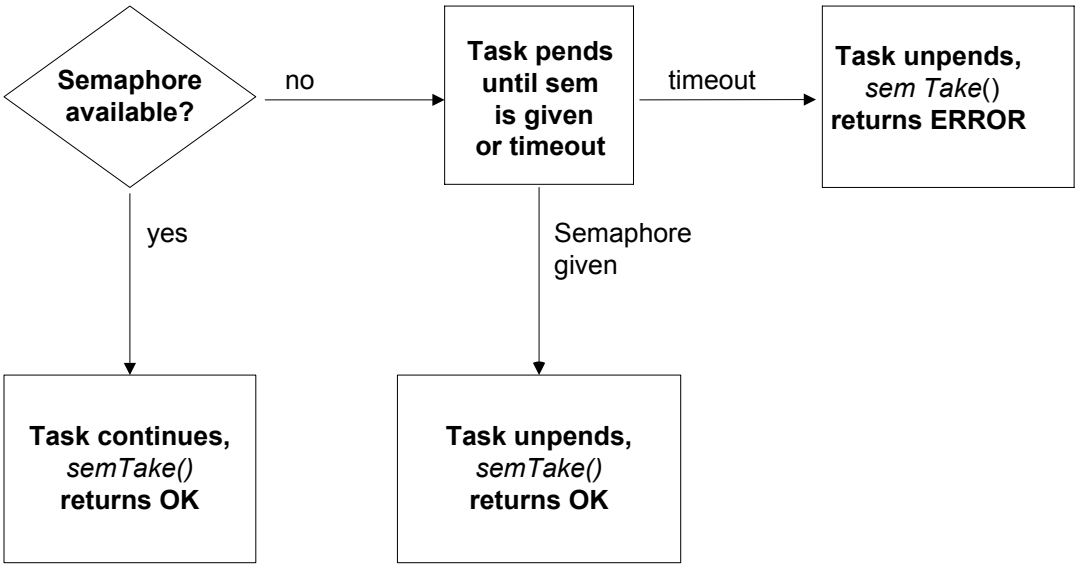


Bild: Taking a Binary Semaphore

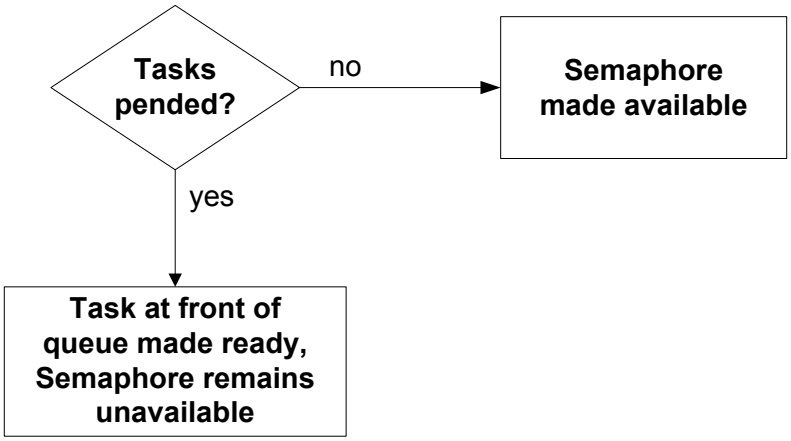


Bild: Giving a Semaphore

7.2.1. Übersicht Befehle zur Tasksynchronisation

SEM_ID = **semBCreate(int *options*, SEM_B_STATE *initialState*)**

// Rückgabewert: Semaphor ID oder NULL

// Options: SEM_Q_PRIORITY: Warteschlange nach Priorität und FIFO-Prinzip,
// SEM_Q_FIFO: Warteschlange nur nach FIFO-Prinzip.

// initalState: SEM_EMPTY oder SEM_FULL.

// Hinweis: Der Semaphor ist nach Aufruf arbeitsbereit.

STATUS = **semGive(SEM_ID *semId*)**

// signalisiert die Semaphore semId

// Rückgabewert: OK oder ERROR, wenn Fehler bei der Signalisierung

STATUS = **semTake(SEM_ID *semId*, int *timeout*)**
// nimmt die Semaphore semId
// Rückgabewert: OK oder ERROR, wenn Semaphore *semId* nicht vorhanden ist oder wenn
// eine Wartezeitüberschreitung aufgetreten ist.
// Timeout: NOWAIT, WAIT_FOREVER oder Anzahl Ticks warten
// Hinweis: Diese Funktion kann nicht in einer ISR aufgerufen werden.

STATUS = **semDelete(SEM_ID *semId*)**
// Semaphoren werden durch semDelete() gelöscht.
// Rückgabewert: OK oder ERROR
// Hinweis: Der Semaphor wird gelöscht.

STATUS = **semFlush(SEM_ID semId)**
// Rückgabewert: OK oder ERROR
// Hinweis: Alle Tasks, die auf das Semaphor warten, werden in den Ready-Zustand versetzt

Befehle zur Taskgenerierung

Id = **taskSpawn** (name, priority, options, stacksize, entrypoint, arg1, .. arg10);

// Task kreieren und aktivieren
name: Name der Task, bestehend aus ASCI Zeichen; "tMeine_Task"
priority: in VxWorks wählbar zwischen 0 (höchste) bis 255 (niedrigste) Priorität
options: Optionen z. B. für das "Task-Debugging"
stacksize: Größe des Stackspeichers für eine Task in Bytes
entrypoint: Hier wird der Name der C-Funktion eingetragen, die im Taskkontext ablaufen soll.
arguments: Diese Argumente sind die C-Funktionsübergabeparameter der entrypoint-Funktion.

7.2.2. Programmbeispiel zu binären Semaphoren

=> Lösung des Synchronisationsproblems mit Hilfe von binären Semaphoren

```
1   #include "vxWorks.h"
2   #include "semLib.h"
3   #include "intLib.h"
4
5   LOCAL SEM_ID mySemId;
6
7   void myInit()
8       {
9           // Initialize software/hardware for device
10      ...
11      mySemId = semBCreate(SEM_Q_FIFO, SEM_EMPTY);
12      intConnect(MY_INT_VEC, myIsr, 0);
13      }
```

```
14
15
16  void myGetData()
17      {
18       // Tweak device registers to start I/O
19       ...
20       // Wait for interrupt indicating data is ready
21       semTake (mySemId, WAIT_FOREVER);
22       // use the data
23       ...
24      }
25
26
27  LOCAL void myIsr()
28      {
29       // Interrupt generated from external device
30      ...
31      semGive (mySemId);
32      }
```

Beispiel: Tasksynchronisation (VxWorks)

```
1   #include "VxWorks.h"
2   #include "semLib.h"
3
4   SEM_ID syncSem;
5
6   Initialization (int someIntNum)
7    {
8     ...
9     // connect interrupt service routine
10    intConnect(INUM_TO_IVEC(someIntNum), eventInterruptSvcRout, 0);
11
12    // create semaphore
13    syncSem = semBCreate(SEM_Q_FIFO, SEM_EMPTY);
14
15    // spawn task used for synchronization
16    taskSpawn("sample", 100, 0, 20000, Task1,0,0,0,0,0,0,0,0,0,0);
17   ...
18   }
19
20
21  Task1()
22   {
23    ...
24    // wait for event to occur
25    semTake(syncSem, WAIT_FOREVER);
26    ...      // process event
27    ...
28   }
29
```

```
30
31  eventInterruptSvcRout()
32   {
33    ...
34    // let Task1 process event
35    semGive(syncSem);
36     ...
37   }
```

Bild: Vollständige Programmlösung zum Synchronisationsproblem

7.2.3. User Services zur Tasksynchronisation beim RTOS Salvo

OSCreateBinSem(): Create a Binary Semaphore

Type:	Function
Prototype:	`OStypeErr OSCreateBinSem ( OStypeEcbP ecbP, OStypeBinSem binSem );`
Callable from:	Anywhere
Contained in:	`binsem.c`
Enabled by:	`OSENABLE_BINARY_SEMAPHORES, OSEVENTS`
Affected by:	`OSCALL_OSCREATEEVENT, OSENABLE_STACK_CHECKING, OSCOMBINE_EVENT_SERVICES, OSLOGGING, OSUSE_EVENT_TYPES`
Description:	Create a binary semaphore with the initial value specified.
Parameters:	`ecbP`: a pointer to the binary semaphore's ecb. `binSem`: the binary semaphore's initial value (0 or 1) .
Returns:	`OSNOERR`
Stack Usage:	1

Notes

Creating a binary semaphore assigns an event control block (ecb) to the semaphore.

A newly-created binary semaphore has no tasks waiting for it.

Signaling or waiting a binary semaphore before it has been created will result in an error if `OSUSE_EVENT_TYPES` is `TRUE`.

You can also implement binary semaphores via messages – see `OSCreateMsg()`.

In the example below, a binary semaphore is used to control access to a shared resource, an I/O port. The port is initially available for use, so the semaphore is initialized to 1.

See Also

`OS_WaitBinSem()`, `OSReadBinSem()`, `OSSignalBinSem()`, `OSTryBinSem()`

Example

```
/* PORTB is a general-purpose I/O port.        */
#define BINSEM_PORTB_P OSECBP(6)
…
/* PORTB is initially available to task that   */

/*  wants to use it.                           */
OSCreateBinSem(BINSEM_PORTB_P, 1);
…
```

OSSignalBinSem(): Signal a Binary Semaphore

Type:	Macro or Function
Prototype:	`OStypeErr OSSignalBinSem ( OStypeEcbP ecbP );`
Callable from:	Anywhere
Contained in:	`binsem.c`
Enabled by:	`OSENABLE_BINARY_SEMAPHORES`, `OSEVENTS`
Affected by:	`OSCALL_OSSIGNALEVENT`, `OSENABLE_STACK_CHECKING`, `OSCOMBINE_EVENT_SERVICES`, `OSLOGGING`, `OSUSE_EVENT_TYPES`
Description:	Signal a binary semaphore. If one or more tasks are waiting for the semaphore, the highest-priority task is made eligible.
Parameters:	`ecbP`: a pointer to the semaphore's ecb.
Returns:	`OSERR_BAD_P` if binary semaphore pointer is incorrectly specified. `OSERR_EVENT_BAD_TYPE` if specified event is not a binary semaphore. `OSERR_EVENT_FULL` if binary semaphore is already 1. `OSNOERR` on success.
Stack Usage:	1

Notes

No more than one task can be made eligible by signaling a binary semaphore.

In the example below, a binary semaphore is used to signal a waiting task. `TaskWaveformGenerator()` outputs an 8-bit waveform to a DAC whenever it receives a signal to do so. The binary semaphore is initialized to 0, so `TaskWaveformGenerator()` remains in the waiting state until the `BINSEM_GEN_WAVEFORM` is signaled elsewhere in the program, whereupon it outputs an array of 8-bit values to a port. It then resumes waiting until `BINSEM_GEN_WAVEFORM` is signaled again.

See Also

`OS_WaitBinSem()`, `OSCreateBinSem()`, `OSReadBinSem()`, `OSTryBinSem()`

Example

```
…

#define BINSEM_GEN_WAVEFORM_P OSECBP(5)

…

OSCreateBinSem(BINSEM_GEN_WAVEFORM_P, 0);

…

/* tell waveform-generating task to create a   */
/*  single waveform.                           */
OSSignalBinSem(BINSEM_GEN_WAVEFORM_P);

…

void TaskWaveformGenerator(void)
{
  char i;

  for (;;)
  {
    /* wait forever for signal to generate     */
    /*  waveform.                              */
    OS_WaitBinSem(BINSEM_GEN_WAVEFORM_P,
      OSNO_TIMEOUT, TaskWaveformGenerator1);

    /* output waveform to DAC.                 */
    for ( i = 0 ; i < 256 ; i ++ )
      DACPORT = WAVEFORM_TABLE[i];
  }
}
```

OS_WaitBinSem(): Context-switch and Wait the Current Task on a Binary Semaphore

Type:	Macro (invokes `OSWaitEvent()`)
Declaration:	`OS_WaitBinSem ( OStypeEcbP ecbP, OStypeDelay timeout, label );`
Callable from:	Task only
Contained in:	`salvo.h`
Enabled by:	`OSENABLE_BINARY_SEMAPHORES`, `OSEVENTS`
Affected by:	`OSENABLE_STACK_CHECKING`, `OSENABLE_TIMEOUTS`, `OSLOGGING`
Description:	Wait the current task on a binary semaphore, with a timeout. If the semaphore is 0, return to the scheduler and continue waiting. If the semaphore is 1, reset it to 0 and continue. If the timeout expires before the semaphore becomes 1, continue execution of the task, with the timeout flag set.
Parameters:	`ecbP`: a pointer the binary semaphore's ecb. `timeout`: an integer (>=0) specifying the desired timeout in system ticks. `label`: a unique label.
Returns:	–
Stack Usage:	2

Notes

Specify a timeout of `OSNO_TIMEOUT` if the task is to wait the binary semaphore indefinitely.

Do not call `OS_WaitBinSem()` from within an ISR!

After a timeout occurs the binary semaphore is undefined.

In the example below for a rocket launching system, a rocket is launched via a binary semaphore `BINSEM_LAUNCH_ROCKET` used as a flag. The semaphore is initialized to zero so that the rocket does not launch on system power-up.[76] Once the rocket is ready and the order has been given to launch (via `OSSignalBinSem()` elsewhere in the code), `TaskLaunchRocket()` starts the rocket on its journey.

---

76 That would be undesirable.

Since the rocket cannot be recalled, there is no need to continue running `TaskLaunchRocket()`, and it simply stops itself. Therefore in order to launch a second rocket, the system must be re-started.

See Also

`OSCreateBinSem()`, `OSReadBinSem()`, `OSSignalBinSem()`, `OSTryBinSem()`

Example

```
#define BINSEM_LAUNCH_ROCKET_P OSECBP(2)

...

/* startup code: no clearance given to launch  */
/*  rocket.                                    */
OSCreateBinSem(BINSEM_LAUNCH_ROCKET_P, 0);

...

void TaskLaunchRocket(void)
{
  /* wait here forever until the order is      */
  /* given to launch the rocket.               */
  OS_WaitBinSem(BINSEM_LAUNCH_ROCKET_P,
    OSNO_TIMEOUT, TaskLaunchRocket1);

  /* launch rocket.                            */
  IgniteRocketEngines();
  ...

  /* rocket is on its way, therefore task is   */
  /*  no longer needed.                        */
  OS_Stop(TaskLaunchRocket2);
}
```

OSReadBinSem(): Obtain a Binary Semaphore Unconditionally

Type:	Function
Prototype:	`OStypeBinSem OSReadBinSem ( OStypeEcbP ecbP );`
Callable from:	Anywhere
Contained in:	`binsem.c`
Enabled by:	`OSENABLE_BINARY_SEMAPHORES`, `OSENABLE_EVENT_READING`, `OSEVENTS`
Affected by:	`OSCALL_OSRETURNEVENT`
Description:	Returns the binary semaphore specified by `ecbP`.
Parameters:	`ecbP`: a pointer to the binary semaphore's ecb.
Returns:	Binary semaphore (0 or 1).
Stack Usage:	1

Notes

`OSReadBinSem()` has no effect on the specified binary semaphore. Therefore it can be used to obtain the binary semaphore's value without affecting the state(s) of any task(s).

No error checking is performed on the `ecbP` parameter. Calling `OSReadBinSem()` with an invalid `ecbP`, or an `ecbP` belonging to an event other than a binary semaphore, will return an erroneous result.

In the example below, a binary semaphore employed as a resource is tested before making a decision to delay a task.

See Also

`OS_WaitBinSem()`, `OSCreateBinSem()`, `OSTryBinSem()`, `OSSignalBinSem()`

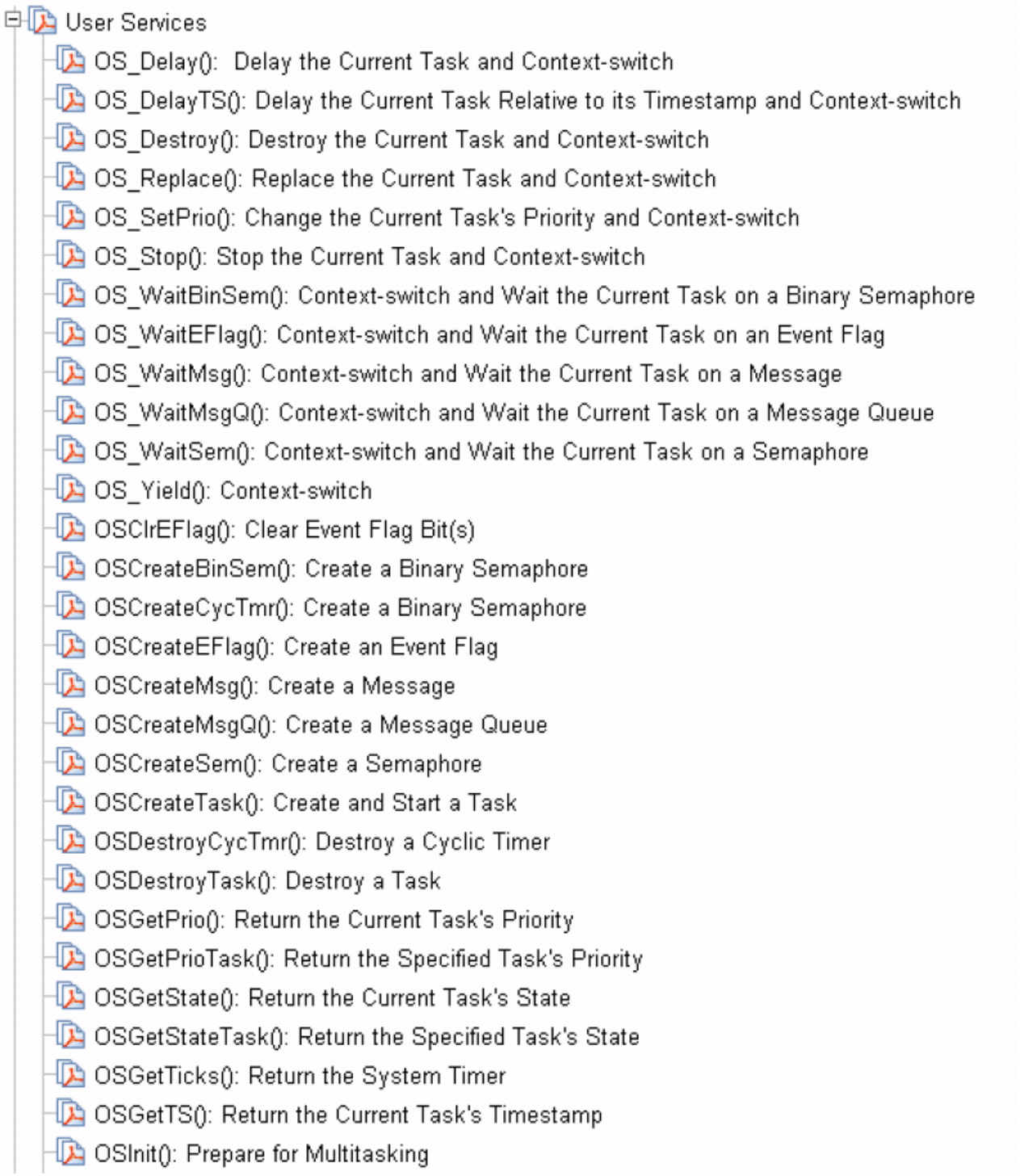
Example

```
…
/* initially, resource #2 is available.        */
OSCreateBinSem(BINSEM_RSRC2_P, 1);

void TaskD (void)
{
  for (;;)
  {
    …
    if ( OSReadBinSem(BINSEM_RSRC2_P) )
      MyFn();
    else
      OS_Delay(100, TaskD1);
  }
}
```

7.2.4. Übung der User Services unter Salvo

This section describes the Salvo user services that you will use to build your mult tasking application. Each user service description includes information on:



- OSMsgQCount(): Return Number of Messages in Message Queue
- OSMsgQEmpty(): Check for Available Space in Message Queue
- OSReadBinSem(): Obtain a Binary Semaphore Unconditionally
- OSReadEFlag(): Obtain an Event Flag Unconditionally
- OSReadMsg():Obtain a Message's Message Pointer Unconditionally
- OSReadMsgQ(): Obtain a Message Queue's Message Pointer Unconditionally
- OSReadSem(): Obtain a Semaphore Unconditionally
- OSResetCycTmr(): Reset a Cyclic Timer
- OSRpt(): Display the Status of all Tasks, Events, Queues and Counters
- OSSched(): Run the Highest-Priority Eligible Task
- OSSetCycTmrPeriod(): Set a Cyclic Timer's Period
- OSSetEFlag(): Set Event Flag Bit(s)
- OSSetPrio(): Change the Current Task's Priority
- OSSetPrioTask(): Change a Task's Priority
- OSSetTicks(): Initialize the System Timer
- OSSetTS(): Initialize the Current Task's Timestamp
- OSSignalBinSem(): Signal a Binary Semaphore
- OSSignalMsg(): Send a Message
- OSSignalMsgQ(): Send a Message via a Message Queue
- OSSignalSem(): Signal a Semaphore
- OSStartCycTmr(): Start a Cyclic Timer
- OSStartTask(): Make a Task Eligible To Run
- OSStopCycTmr(): Stop a Cyclic Timer
- OSStopTask(): Stop a Task
- OSSyncTS(): Synchronize the Current Task's Timestamp
- OSTimer(): Run the Timer
- OSTryBinSem(): Obtain a Binary Semaphore if Available
- OSTryMsg(): Obtain a Message if Available
- OSTryMsgQ(): Obtain a Message from a Message Queue if Available
- OSTrySem(): Obtain a Semaphore if Available

- Additional User Services
  - OSAnyEligibleTasks (): Check for Eligible Tasks
  - OScTcbExt0|1|2|3|4|5, OStcbExt0|1|2|3|4|5(): Return a Tcb Extension
  - OSCycTmrRunning(): Check Cyclic Timer for Running
  - OSDi(), OSEi(): Control Interrupts
  - OSProtect(), OSUnprotect(): Protect Services Against Corruption by ISR
  - OSTimedOut(): Check for Timeout
  - OSVersion(), OSVERSION: Return Version as Integer
- User Macros
  - _OSLabel(): Define Label for Context Switch
  - OSECBP(), OSEFCBP(),OSMQCBP(), OSTCBP(): Return a Control Block Pointer
- User-Defined Services
  - OSDisableIntsHook(), OSEnableIntsHook(): Interrupt-control Hooks
  - OSIdlingHook(): Idle Function Hook
  - OSSchedDispatchHook(), OSSchedEntryHook(), OSSchedReturnHook(): Scheduler Hooks

7.2.5. Übung 1 zur Tasksynchronisation

Gegeben seien zwei zyklische Tasks. Die eine Task soll Vorgänger (v) und die andere Nachfolger (n) genannt werden. Beide Tasks haben einen Programmabschnitt Kv und Kn. Die Synchchronisationsaufgabe lautet:

1. Der Abschnitt Kn des Nachfolgers darf erst dann durchlaufen werden, wenn der Vorgänger den Abschnitt Kv mindestens einmal duchlaufen hat.
2. Die Aufgabe soll mit einem binären Semaphor gelöst werden. Das Semaphor soll global vereinbart werden.
3. Die Tasks sollen beliebig lange existieren.

Bitte vervollständigen Sie das Programm durch Eintragen geeigneter Synchronisierungsbefehle.

```
SEM_ID  semID;       // Deklaration der Semapohre(n)
INT  Tv,  Tn;

void start()
{
 // Semapohre semID erzeugen
 semID =  semBCreate(SEM_Q_PRIORITY, SEM_EMPTY);

 // Tasks generieren
 Tv = taskSpawn ("Task_Tv", 20, 0, 1000, Vorgaenger, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
 Tn = taskSpawn ("Task_Tn", 20, 0, 1000, Nachfolger, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
}


STATUS Vorgaenger (void)
{
   for ( ; ; )
     {

       ......................

       //  Programmabschnitt Kv


       ......................
    }
}



STATUS Nachfolger (void)
{
   for ( ; ; )
     {

       ......................

       //  Programmabschnitt Kn


       ......................
```

```
        }
}
```

Σ Punkte 3 / 3

7.2.6. Übung 2 zur Tasksynchronisation

Aufgabe 6: Tasksynchronisation - Lösung

Bitte vervollständigen Sie das Programm durch Eintragen geeigneter Synchronisierungsbefehle.

```
SEM_ID  semaphoreID;                  // Deklaration der Semapohre(n)
INT  Tv,  Tn;

void start()
{
 semaphoreID =  semBCreate(SEM_Q_PRIORITY, SEM_EMPTY).....    // Semapohre erzeugen

 Tv = taskSpawn ("Task_Tv", 20, 0, 1000, Vorgaenger, 0, 0, 0, 0, 0, 0, 0, 0, 0);   // Tasks generieren
 Tn = taskSpawn ("Task_Tn", 20, 0, 1000, Nachfolger, 0, 0, 0, 0, 0, 0, 0, 0, 0);
}

STATUS Vorgaenger (void)
{
  for ( ; ; )
    {
      .....

      .....

      //  Programmabschnitt Kv

      .....

      .....semGive(semaphoreID).....

    }
}
```

```
STATUS Nachfolger (void)
{
  for ( ; ; )
    {
      .....................

      ..... semTake(semaphoreID).....................

      // Programmabschnitt Kn

      .....................

      .....................
    }
}
```

Aufbauend auf dem vorherigen Programm sollen nun weitere Bedingungen berücksichtigt werden. Die zusätzlichen Synchchronisationsaufgaben lauten:

1. Dem Durchlauf des Vorgängers muss **stets genau ein** Durchlauf des Nachfolgers folgen.
2. Der Nachfolger kann erst wieder ablaufen, wenn der Vorgänger durchlaufen wurde.
3. Der Vorgänger beginnt.

Bitte vervollständigen Sie das Programm durch Eintragen geeigneter Synchronisierungsbefehle.

```
SEM_ID se;       // Deklaration der Semapohre(n)
SEM_ID sv;

INT  Tv, Tn;

void start()

{
 // Semapohre se erzeugen
 se =  semBCreate (SEM_Q_PRIORITY, SEM_FULL);

 // Semapohre sv erzeugen
 sv =  semBCreate (SEM_Q_PRIORITY,  SEM_EMPTY);

 // Tasks generieren
 Tv = taskSpawn ("Task_Tv", 20, 0, 1000, Vorgaenger, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
 Tn = taskSpawn ("Task_Tn", 20, 0, 1000, Nachfolger, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
}


STATUS Vorgaenger (void)

{ for ( ; ; )
```

```
      {

        ......................

        //  Programmabschnitt Kv


        ......................
     }
}


STATUS Nachfolger (void)
{
 for ( ; ; )
      {
        ......................
        //  Programmabschnitt Kn

        ......................
     }
}
```

7.2.7. Übung 3 zur Tasksynchronisation

In einem Echtzeit-Programm zur Automatisierung eines Produktionsprozesses werden 3 Teilprozesse durch jeweils eine Task überwacht. Die Tasks sollen im Folgenden mit ÜBERWACHENi bezeichnet werden, wobei i=1, 2, 3 sein soll.

Jede dieser Tasks liest spezifische Werte von dem jeweiligen Teilprozess ein. Anschließend werden diese Messwerte durch eine universelle Protokolliertask (die Protokolliertask trägt die Bezeichnung PROTOKOLL) in Form von Tabellen und Diagrammen auf einem Drucker ausgegeben.

Durch eine geeignete Synchronisierung soll folgender Ablauf der angeführten Tasks erzwungen werden:

- Nach jeder Überwachungstask soll einmal die Protokolliertask ablaufen.
- Die Reihenfolge der Überwachungstasks untereinander ist durch die Zahl in ihrem Namen festgelegt, d. h. zuerst ÜBERWACHEN1, ÜBERWACHEN2, ... Das Programm einer Task soll vollständig abgearbeitet sein, bevor das Programm einer

anderen Task abgearbeitet wird.

- Der oben vorgegebene Ablauf soll zyklisch ablaufen können.

Aufgabe 1:

Schreiben Sie die Namen der Tasks in der oben geforderten Ablaufreihenfolge auf.

Aufgabe 2:

Fügen Sie in jeder Task an der geeignenten Stelle die erforderlichen Synchronisierungsoperationen zur Sicherstellung der Ablaufreihenfolge ein. Verwenden Sie dabei ausschließlich binäre Semaphoren und die Befehle SemGive(SemID) und SemTake(SemID), wobei für SemID (=SemaphoreID) beliebige Namen verwendet werden können, z. B. SemID = S1 für die Tasksynchronisation von Task ÜBERWACHEN1, SemID = S2 für die Tasksynchronisation von Task ÜBERWACHEN2 usw..

Tasks

ÜBERWACHEN1	ÜBERWACHEN2	ÜBERWACHEN3	PROTOKOLL
BEGIN	BEGIN	BEGIN	BEGIN
....	....	....	....
// C-Source	// C-Source	// C-Source	// C-Source
// Überwachen1	// Überwachen2	// Überwachen3	// Protokollieren
....	....	....	....
END.	END.	END.	END.

Aufgabe 3:

Die Semaphoren stellen unter VxWorks Kernelobjekte dar und müssen deshalb vor Ihrer Verwendung mit SemBCreate() einmalig erzeugt und mit einem Anfangswert (SEM_EMPTY = 0, SEM_FULL = 1) initialisiert werden. Mit welchen Anfangswerten müssen die von Ihnen verwendeten Semaphore-Variablen (SemID1, ..) initialisiert werden?

7.2.8. Übung 4 zur Tasksynchronisation

2 Tasks sollen abwechselnd hintereinander ablaufen. Stellen Sie unter Zuhilfenahme von binären Semaphoren den gewünschten Ablauf sicher.

7.2.9. Übung 5 zur Tasksynchronisation

Im nachstehenden Programm sind die Operationen zur Beeinflussung der binären Semaphoren jeweils am Anfang und am Ende der Tasks angeordnet.

Der RTOS-Scheduler arbeitet prioritätsorientiert und nach dem „First Come First Served“-Prinzip, der Kernel ist non-preemptiv. Die Tasks werden in der Reihenfolge TaskA, TaskB und TaskC initialisiert. Alle Tasks besitzen die Priorität 90.

```
TaskA                          TaskB                          TaskC
{                              {                              {
 while(1)                      while(1)                        while(1)
   {                               {                            {
     SemTake(SA);                    SemTake (SB) ;               SemTake(SC) ;
     SemTake(SA);                    SemTake (SB) ;               SemTake(SC) ;
     SemTake(SA) ;                                                SemTake(SC) ;


     SetP1.0_LowHigh();              SetP1.1_LowHigh();           SetP1.2_LowHigh();
     taskDelay(50);                  taskDelay(100);              taskDelay(200);
     SetP1.0_HighLow();              SetP1.1_HighLow();           SetP1.2_HighLow();


                                     SemGive(SA);
                                     SemGive(SB);                 SemGive(SB);
     SemGive(SB);                    SemGive(SC);                 SemGive(SB);
   }                               }                            }
}                                }                              }
```

Bild: Programm Impulsmustergenerierung

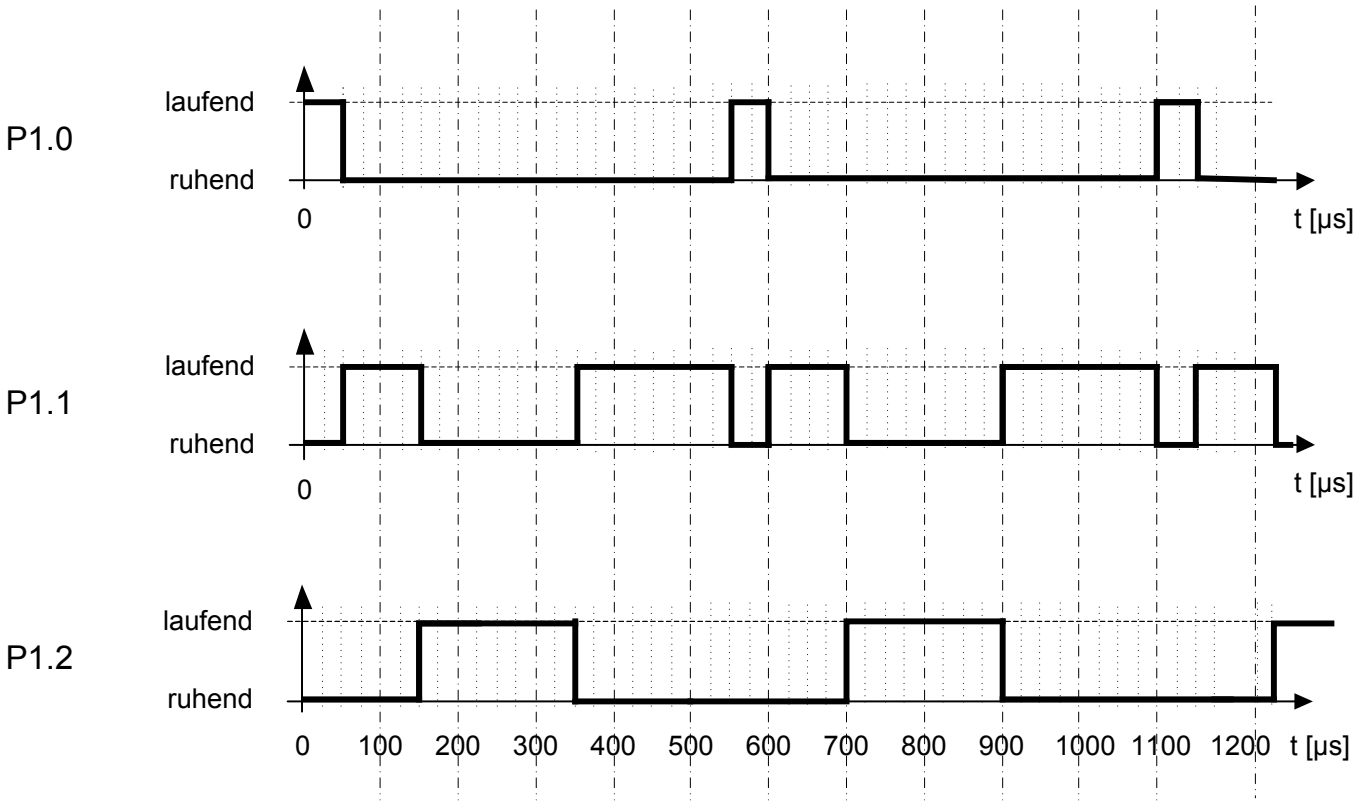
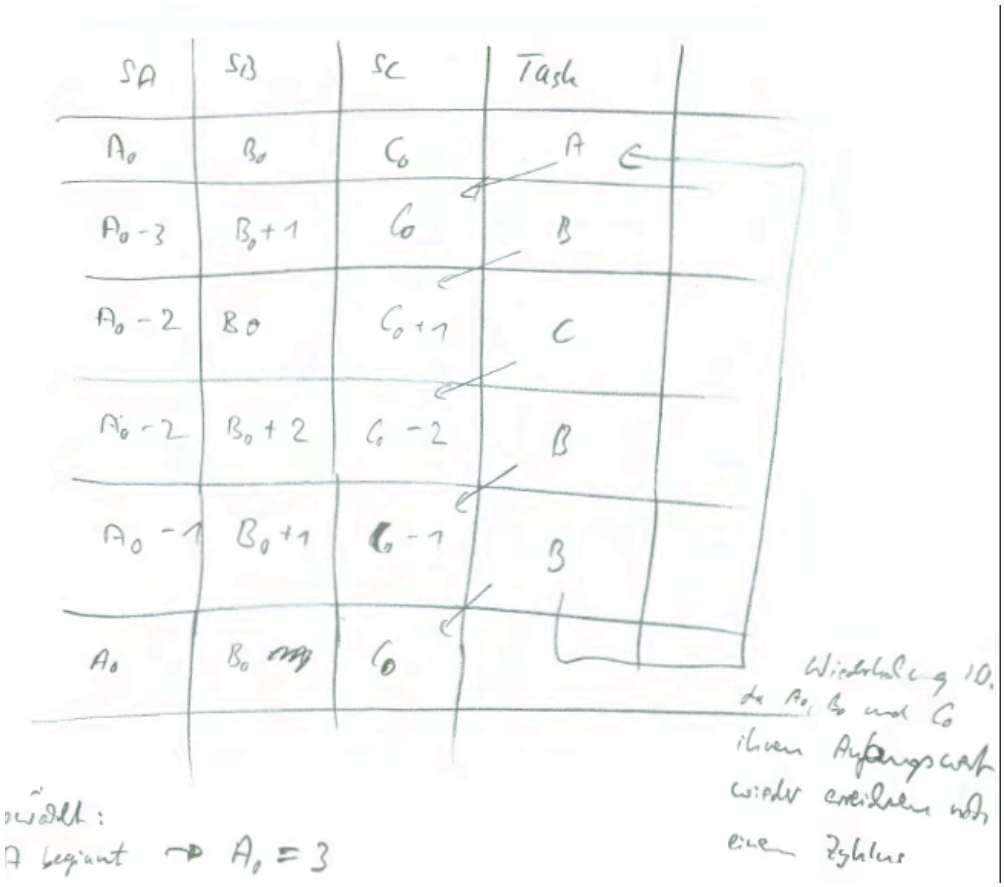


Bild: Impulsdiagramm

a) In welcher Reihenfolge müssen die Tasks ablaufen, um das obige Impulsdiagramm zu realisieren.

=> Ablauf A → B → C → B → B

b) Ist der unter a) ermittelte Taskablauf mit dem Programm Impulsmustergenerierung prinzipiell realisierbar? Weisen Sie dies allgemein nach.



c) Ermitteln Sie die Anfangswerte für SA, SB und SC so, dass das nachstehende periodische Impulsmuster erzeugt wird. Verwenden Sie dabei die RTOS-Funktionen im Anhang dieser Aufgabenstellung.

gewählt:
Task A beginnt → $A_0 = 3$
B macht weiter → $B_0 = 1$
C folgt nach B → $C_0 = 2$

SA	SB	SC	Task
3	1	2	A
0	2	2	B
1	1	1	C
1	1	0	B
2	2	1	B
3	1	2	L

Wiederholung

Aufgabe 6:

Im nachstehenden Programm sind die Semaphoroperationen jeweils am Anfang und am Ende der Tasks angeordnet.

a) Ermitteln Sie die Anfangswerte für SA, SB und SC, um so das nachstehende Impulsdiagramm generieren zu können.

Anm.: Der RTOS-Scheduler arbeitet prioritätsorientiert und nach dem „First Come First Served"-Prinzip, der Kernel ist non-preemptiv. Die Tasks werden in der Reihenfolge TaskA, TaskB und TaskC initialisiert. Alle Tasks besitzen die Priorität 90.

TaskA

```
{
while(1)
 {
  SemTake(SA);
  SemTake(SA);
  SemTake(SA) ;

  SetP1.0_LH();
  taskDelay(50);
  SetP1.0_HL();

  SemGive(SB);
 }
}
```

TaskB

```
{
while(1)
 {
  SemTake (SB) ;
  SemTake (SB) ;

  SetP1.1_LH();
  taskDelay(100);
  SetP1.1_HL();

  SemGive(SC) ;
  SemGive(SA) ;
 }
}
```

TaskC

```
{
while(1)
 {
  SemTake(SC) ;
  SemTake(SC) ;
  SemTake(SC) ;

  SetP1.2_LH();
  taskDelay(200);
  SetP1.2_HL();

  SemGive(SB) ;
  SemGive(SB) ;
 }
}
```

SA	SB	C	ist läuft
A_0	B_0	C_0	A
A_0-3	B_0+1	C_0	B
A_0-2	B_0-1	C_0+1	C
A_0-2	B_0+1	C_0-2	B
A_0-1	B_0-1	C_0-1	B
A_0	B_0-3	C_0	

B_0 erreicht nicht wieder Anfangswert nach einem Durchlauf

Bild: Anordnung der Semaphoren-Operationen in den Tasks

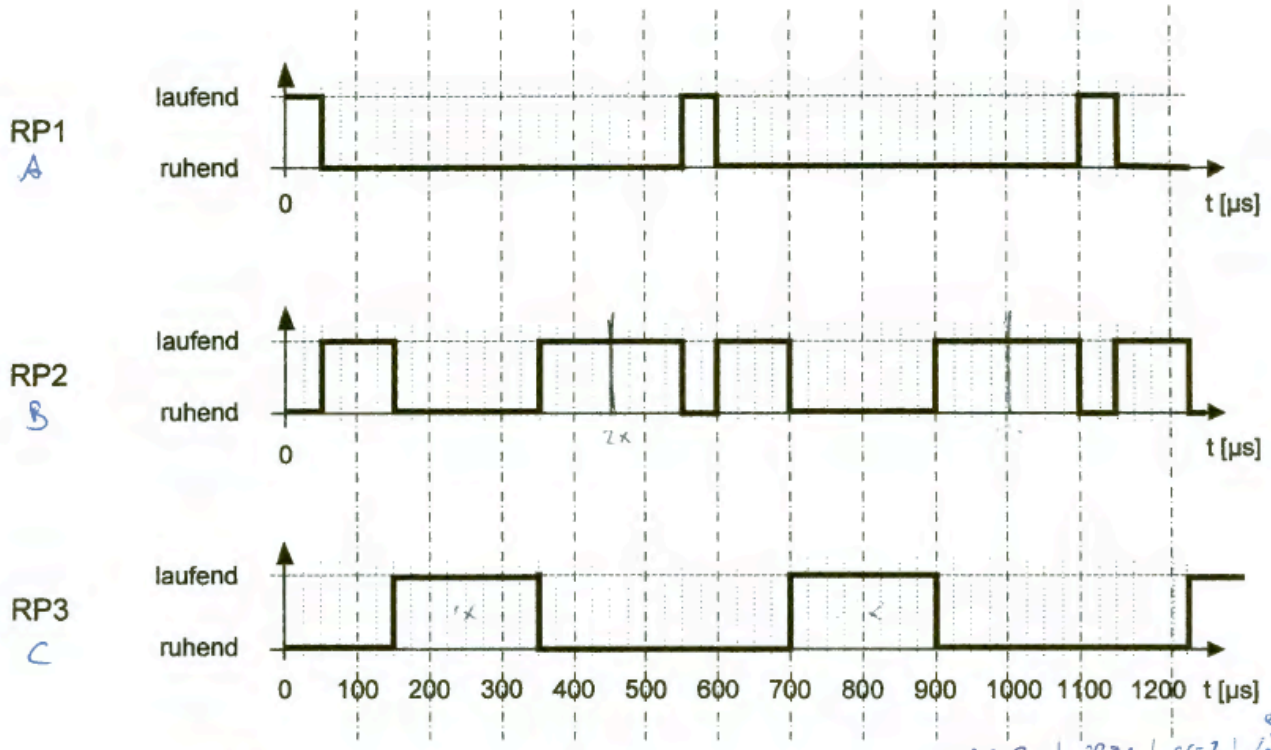
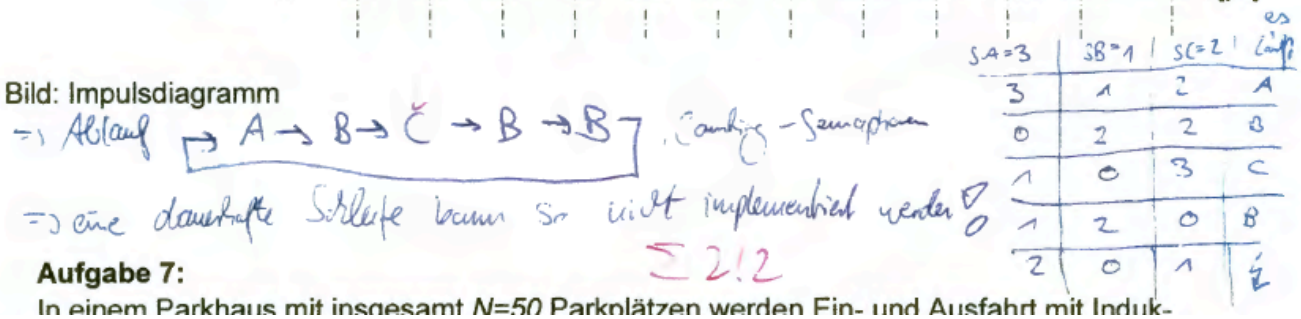


Bild: Impulsdiagramm



SA=3	SB=1	SC=2	Lauf
3	1	2	A
0	2	2	B
1	0	3	C
1	2	0	B
2	0	1	

Aufgabe 7:

In einem Parkhaus mit insgesamt *N=50* Parkplätzen werden Ein- und Ausfahrt mit Induk-

7.3. Zählende Semaphoren (Counting Semaphores)

Zählende Semaphoren werden bei der Erzeugung mit einem Anfangswert initialisiert. Durch *semTake()* wird dieser Wert dekrementiert, mit *semGive()* inkrementiert. Hat die Semaphore beim Nehmen einen Wert von >0, läuft die Task weiter, im anderen Fall muss die Task warten.

Zählende Semaphoren werden in VxWorks durch den Funktionsaufruf *semCCreate()* erzeugt.

Übersicht der in VxWorks zur Verfügung stehenden Funktionsaufrufe zur Handhabung von zählenden Semaphoren.

```
semBCreate(int options, SEM_B_STATE initialState )   // Allocate and intialize a binary semaphore.
semMCreate(int options)                              // Allocate and intialize a mutual exclusion semaphore.
semCCreate(int options, int initialCount)            // Allocate and intialize a counting semaphore.
semDelete(SEM_ID semId)                              // Terminate and free a semaphore.
semTake(SEM_ID semId, int timeout)                   // Take a semaphore.
```

```
semGive(SEM_ID semId)          // Give a semaphore.
semFlush(SEM_ID semId)         // Unblock all tasks waiting for a semaphore.
```

Counting semaphores are another means to implement task synchronization and mutual exclusion. The counting semaphore works like the binary semaphore except that it keeps track of the number of times a semaphore is given.

Every time a semaphore is given, the count is incremented; every time a semaphore is taken, the count is decremented. When the count reaches zero, a task that tries to take the semaphore is blocked.

As with the binary semaphore, if a semaphore is given and a task is blocked, it becomes unblocked. However, unlike the binary semaphore, if a semaphore is given and no tasks are blocked, then the count is incremented. This means that a semaphore that is given twice can be taken twice without blocking.

The next figure shows an example time sequence of tasks taking and giving a counting semaphore that was initialized to a count of 3.

Semaphore Call	Count after Call	Resulting Behavior
semCCreate( )	3	Semaphore initialized with an initial count of 3.
semTake( )	2	Semaphore taken.
semTake( )	1	Semaphore taken.
semTake( )	0	Semaphore taken.
semTake( )	0	Task blocks waiting for semaphore to be available.
semGive( )	0	Task waiting is given semaphore.
semGive( )	1	No task waiting for semaphore; count incremented.

Bild: Funktionsweise von Counting Semaphoren - Beispiel

Counting semaphores are useful for guarding multiple copies of resources. For example, the use of five tape drives might be coordinated using a counting semaphore with an initial count of 5, or a ring buffer with 256 entries might be implemented using a counting semaphore with an initial count of 256.

The initial count is specified as an argument to the semCCreate( ) routine.

7.3.1. Beispiel Produzent-Konsument Problem

Ein Produzent legt seine Daten in einem endlichen Puffer ab, der Konsument soll diese dort abholen. Für die Implementierung ist folgendes Synchronisationsproblem zu lösen:

- Der Produzent kann nur dann neue Daten in den Puffer ablegen, solange mindestens ein freier Platz verfügbar ist. Andernfalls muss er warten, bis ein freier Platz verfügbar ist.

- Der Konsument hingegen kann solange Daten entnehmen, solange mindestens ein Pufferplatz Daten enthält, ansonsten muss er warten.
- Wenn der Produzent den Puffer beschreibt, darf der Konsument keine Daten abholen und umgekehrt.

Lösung:

Für das Produzent-Konsumenten Problem sind zwei zählende und eine binäre Semaphore notwendig.

7.3.2. Übung 1 zu Counting Semaphore

Im nachstehenden Programm wurden Semaphoroperationen am Anfang und am Ende der Task_A, Task_B und Task_C eingefügt. Allen Tasks soll dieselbe Priorität zugrunde gelegt werden.

Ermitteln Sie, ob und in welcher Reihenfolge diese Tasks bei einer Initialisierung nach Fall a), b), c) und d) ablaufen.

```
Task_A          Task_B          Task_C
{               {               {
 SemTake(SA)     SemTake (SB)    SemTake(SC)
 SemTake(SA)     .               SemTake(SC)
 SemTake(SA)     .               SemTake(SC)
      .          .               .
      .          .               .
      .          .               .
                 SemGive(SC)     SemGive(SB)
 SemGive(SB)     SemGive(SA)     SemGive(SB)

}               }               }
```

Bild: Anordnung der Semaphoroperationen in den Tasks A, B und C

1a) Anfangswerte:
SA = 2
SB = 0
SC = 2

1b) Anfangswerte:
SA = 3
SB = 0
SC = 2

1c) Anfangswerte:
SA = 2
SB = 1
SC = 1

1d) Anfangswerte:
SA = 0
SB = 0
SC = 3

Lösung:

a)	Werte der Semaphorvariablen			Reihenfolge der Tasks
	SA	SB	SC	
Initialisierung	2	0	2	Keine der Tasks ist ablauffähig

b)	Werte der Semaphorvariablen			Reihenfolge der Tasks
	SA	SB	SC	ABCBB ABCBB …
Initialisierung:	3	0	2	->A
	0	1	2	->B
	1	0	3	->C
	1	2	0	->B
	2	1	1	->B
	3	0	2	-> siehe Initialisierung

c)	Werte der Semaphorvariablen			Reihenfolge der Tasks
Initialisierung:	SA	SB	SC	BABCB BABCB …
	2	1	1	
	3	0	2	Weitere Reihenfolge wie bei b), also ABCBB ABCBB …

d)	Werte der Semaphorvariablen			Reihenfolge der Tasks
Initialisierung:	SA	SB	SC	CBB, dann ist keine Task mehr ablauffähig
	0	0	3	->C
	0	2	0	->B
	1	1	1	->B

2 0 2 ->|

7.3.3. Übung 2 zu Counting Semaphore

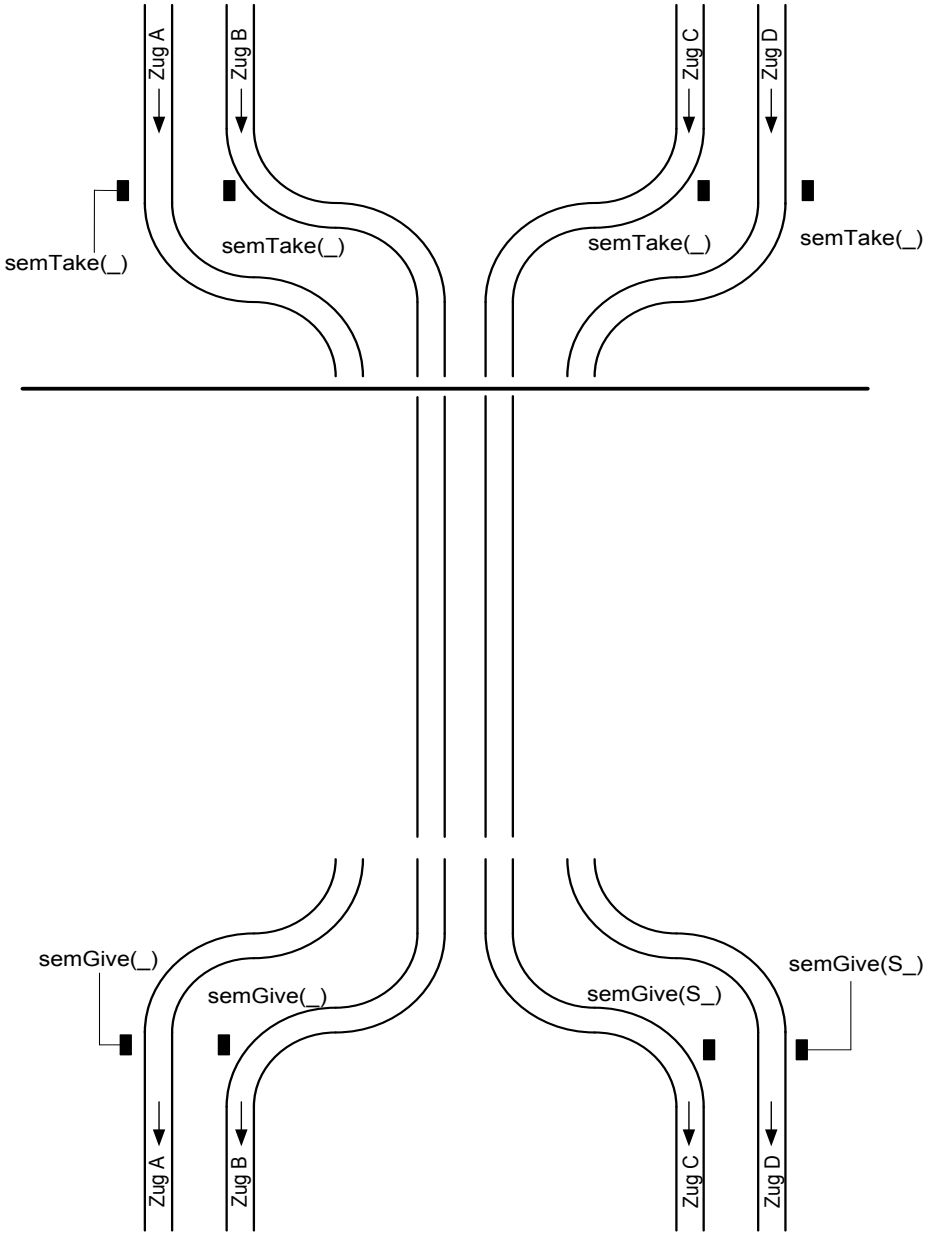


Bild: Bsp: Synchronisation von Zügen mittels Couting Semaphoren

7.4. Mutual Exclusion Semaphoren

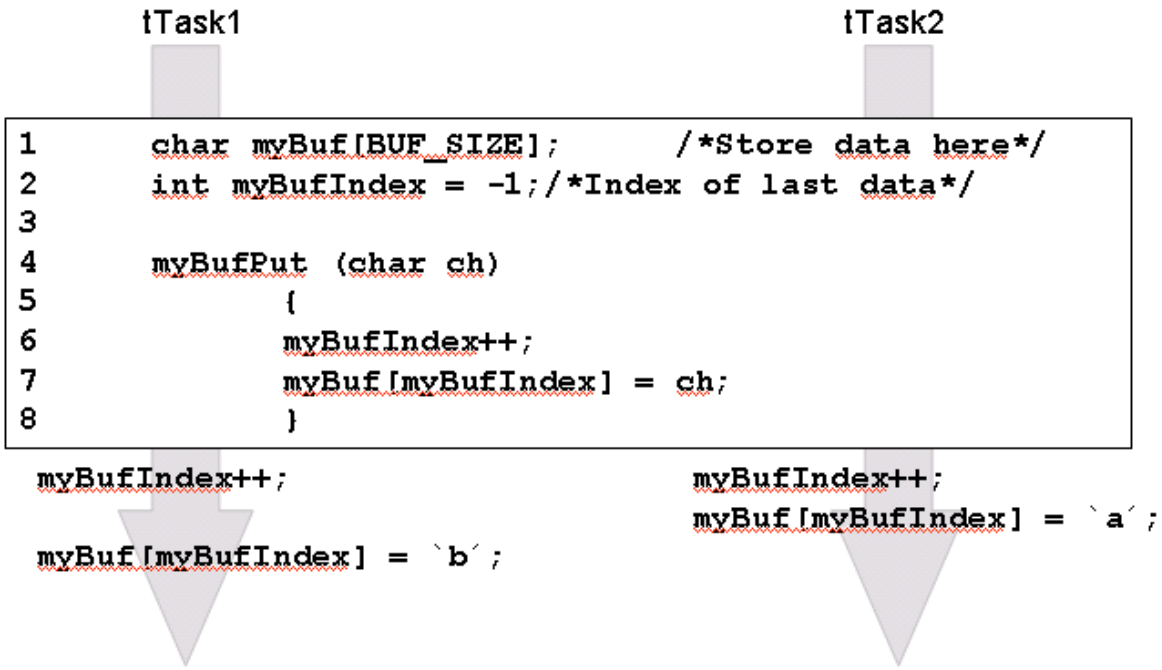
The Mutual Exclusion Problem:

- Some resources may be left inconsistent if accessed by more than one task simultaneously.
  - Shared data structures
  - Shared files

- Shared hardware devices

- Must obtain exclusive access to such a resource before using it.
- If exclusive access is not obtained, then the order in which tasks execute affects correctness.
  - We say a race condition exists.
  - Very difficult to detect during testing.
- Mutual exclusion cannot be compromised by priority.

Beispiel: Race Condition



Vorgehen zur Lösung des Problems:

- Create a mutual exclusion semaphore to guard the resource.
- Call semTake() before accessing the resource; call semGive() when done.
  - **semTake( )** will block until the semaphore (and hence the resource) becomes available.
  - **semGive( )** releases the semaphore (and hence access to the resource).

Creating Mutual Exclusion Semaphores

SEM_ID semMCreate (*options*)

- *options* can be:
  - queue specification SEM_Q_FIFO or SEM_Q_PRIORITY
  - deletion safety SEM_DELETE_SAFE
  - priority inheritance SEM_INVERSION_SAFE
- Initial state of semaphore is available

Mutex Ownership

- A task which takes a mutex semaphore „owns“ it, so that no other task can give this semaphore.
- Mutex semaphores can be taken recursively.
  - The task which owns the semaphore may take it more than once.
  - Must be given same number of times as taken before it will be released.
- Mutual exclusion semaphores cannot be used in an interrupt service routine.

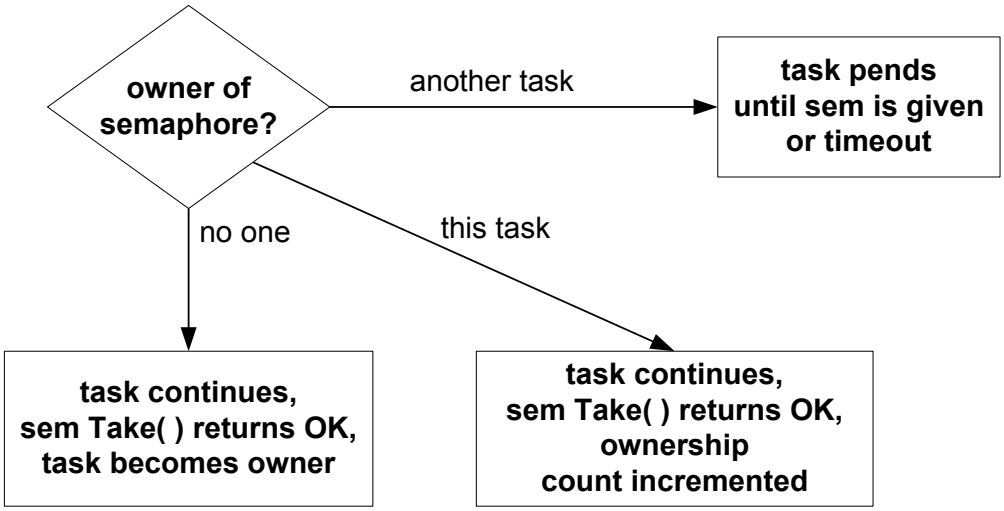
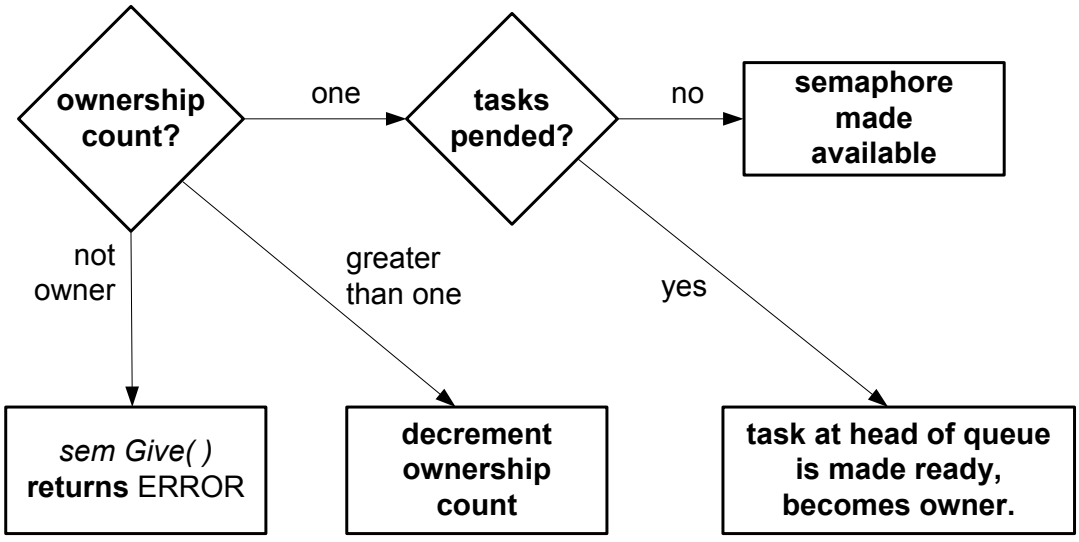


Bild: Taking a Mutex Semaphore (= Gegenseitiger Ausschluss)



Difference from binary semaphore: Only the owner of a semaphore can give it.

Bild: Giving a Mutex Semaphore

7.4.1. Programmbeispiel zu Mutual Exclusion Semaphoren

```
1   #include "vxWorks.h"
2   #include "semLib.h"
3
4   LOCAL char myBuf[BUF_SIZE];     // Store data here
5   LOCAL int myBufIndex = -1;      // Index of last data
6   LOCAL SEM_ID mySemId;
7
8   void myBufInit()
9       {
10       mySemId = semMCreate( SEM_Q_PRIORITY |
11                             SEM_INVERSION_SAVE |
12                             SEM_DELETE_SAVE );
13      }
14
15  void myBufPut(char ch)
16      {
17       semTake (mySemId, WAIT_FOREVER);
18       myBufIndex++;
19       myBuf[myBufIndex] = ch;
20       semGive (mySemId);
21      }
```

Listing: Programmlösung mit Mutual Exclusion Semaphore

Deletion Safety

- Deleting a task which owns a semaphore can be catastrophic.
  - data structures left inconsitent.
  - semaphore left permanently unavailable.
- The deletion safety option prevents a task from being deleted while it owns the sephamore.
- Enabled for mutex semaphores by specifying the SEM_DELETE_SAFE option during semMCreate( ).

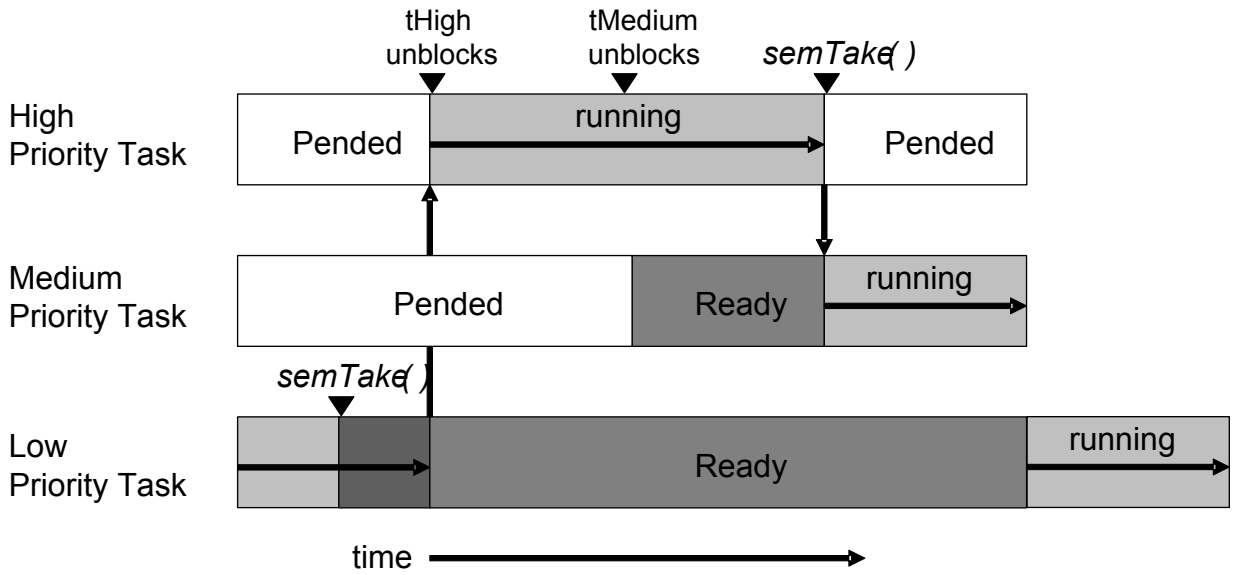


Bild: Unbegrenzte Prioritätsumkehrung

Priority Inheritance

- Priority inheritance algorithm solves the unbounded priority inversion problem.
- Task owning a mutex semaphore is elevated to priority of highest priority task waiting for that semaphore.
- Enabled on mutex semaphores by specifying the SEM_INVERSION_SAFE option during semMCreate( ).
- Must also specify SEM_Q_PRIORITY(SEM_Q_FIFO is incompatible with SEM_INVERSION_SAFE).

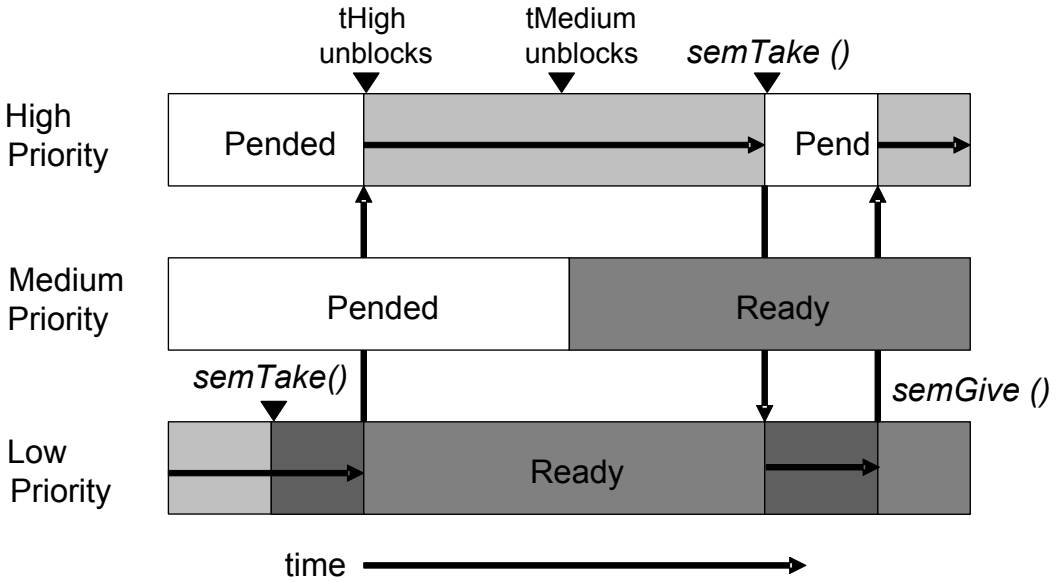


Bild: Priority Inversion Safety

Common Routines

Additional semaphore routines:

- semDelete( ) Destroy the semaphore. All tasks pended on the semaphore return ERROR from semTake( ).
- show( ) Display semaphore information.

To inspect the properties of a specific semaphore, insert the semaphoreID in the Browser's Show box, and click on Show.

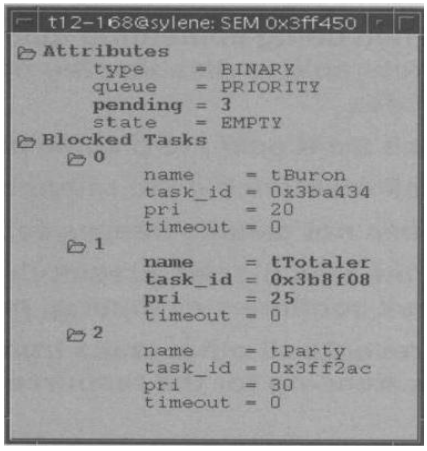


Bild: Semaphore Browser

Locking Out Preemption

- When doing something quick frequently, it may be preferable to disable preemption instead of taking a mutex.

- Call taskLock( ) to disable preemption.
- Call taskUnlock( ) to re-enable preemption.
- Does not disable interrupts.
- If the tasks blocks, preemption is re-enabled. When the task continues executing, preemption is again locked.
- Prevents all other tasks from running, not just the tasks contending for the resource.

ISRs and Mutual Exclusion

- ISRs can´t use mutex semaphores.
- Task sharing a resource with an ISR need to disable interrupts.
- To disable/re-enable interrupts:
  - int intLock( )
  - void intUnlock (lockKey)

    lockKey is return value from intLock( )
- Keep interrupt lock-out time short (e.g., long enough to set a flag)!
- Making kernel calls at task level can reenable interrupts!

Zusammenfassung

- Binary Semaphores allow tasks to pend until some event occurs.
  - Create a binary semaphore for the given event.
  - Tasks waiting for the event blocks on a semTake( ).
  - Task or ISR detecting the event calls semGive( ) or semFlush( ).
- Caveat: if the event repeats too quickly, information may be lost.

- Mutual Exclusion Semaphores are appropriate for obtaining exclusive access to a resource.
  - Create a mutual exclusion semaphore to guard the resource.
  - Before accessing the resource, call semTake( ).
  - To release the resource, call semGive( ).
- Mutex semaphores have owners.

- Vorsichtsmaßnahmen::
  - Keep critical regions short.
  - Make all accesses to the resource through a library of routines.
  - Can´t be used at interrupt time.
  - Deadlocks.

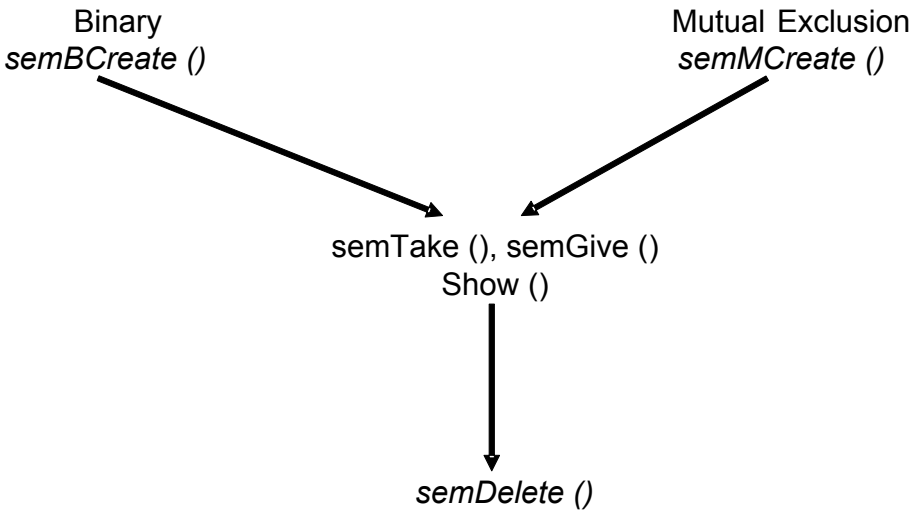


Bild: Summary

- taskLock( ) / taskUnlock( ):
  - Prevents other tasks from running.
  - Use when doing something quick frequently.
  - Caveat: keep critical region short.

- intLock( ) / intUnlock( ):
  - Disables interrupts.
  - Use to protect resources used by tasks and interrupt service routines.
  - Caveat: keep critical region short.

7.4.2. Übung 1 zu Mutual Exclusion Semaphoren

Aufgabe:

Zwei Züge sollen ein Stück weit über eine eingleisige Strecke (Tunnelstrecke) fahren. Die eingleisige Strecke stellt hier ein „gemeinsames Betriebsmittel“dar. Das nachstehende Bild zeigt die Gleisanordnung und die verwendeten Signale.

Die Signalgebung ist wie folgt: Vor der Zusammenführung der beiden Signale befinden sich die beiden Lichtsignale S1 bzw. S2. Die Lichtsignale werden von Gleismagneten bei der Ein- und Ausfahrt der Züge A und B wie folgt geschaltet:

- Bei der Anfahrt eines Zuges wird über den im Bild oben eingezeichneten Gleismagneten die Operation semTake(Si) ausgelöst. Sie bedeutet: Falls das Signal auf grün steht, soll der Zug weiterfahren und das Signal auf rot umschalten. Falls das Signal auf rot steht, soll der Zug angehalten werden und

warten, bis das Signal auf grün steht.

- Bei der Ausfahrt eines Zuges wird über den im Bild unten eingezeichneten Gleismagneten die Operation semGive(Si) ausgelöst, Sie bewirkt, dass das angeschlossene Signal des jeweils anderen Gleises von rot auf grün geschaltet wird.
- In dem Beispiel besteht eine Anfangsbedingung. Diese legt fest, welcher Zug zuerst über die eingleisige Strecke fahren kann, es ist hier der Zug A. Anschließend wechseln sich die Züge in der Reihenfolge ABABAB...ab.

Übertragen Sie die Aufgabenstellung auf die Synchronisierung von zwei Rechenprozessen, hier TaskA und TaskB.

Geben Sie an, wieviele und welche Semaphoren benötigt werden und geben Sie die Tasksabläufe hierzu an. Beschreiben Sie dabei an welcher Stelle die Semaphoroperationen SemTake() und semGive() in die Anweisungsfolge der Rechenprozesse einzufügen sind.

Lösung der Aufgabe:

Zur Lösung dieser Aufgabe werden zwei Semaphorvariablen S1 und S2 benötigt.

Zu Beginn werden die Semaphorvariablen S1=1 und S2=0 gesetzt.

In die Anweisungsfolge des Rechenprozesses A wird zu Beginn die Operation semTake(S1) und am Ende die Operation semGive(S2) eingefügt.

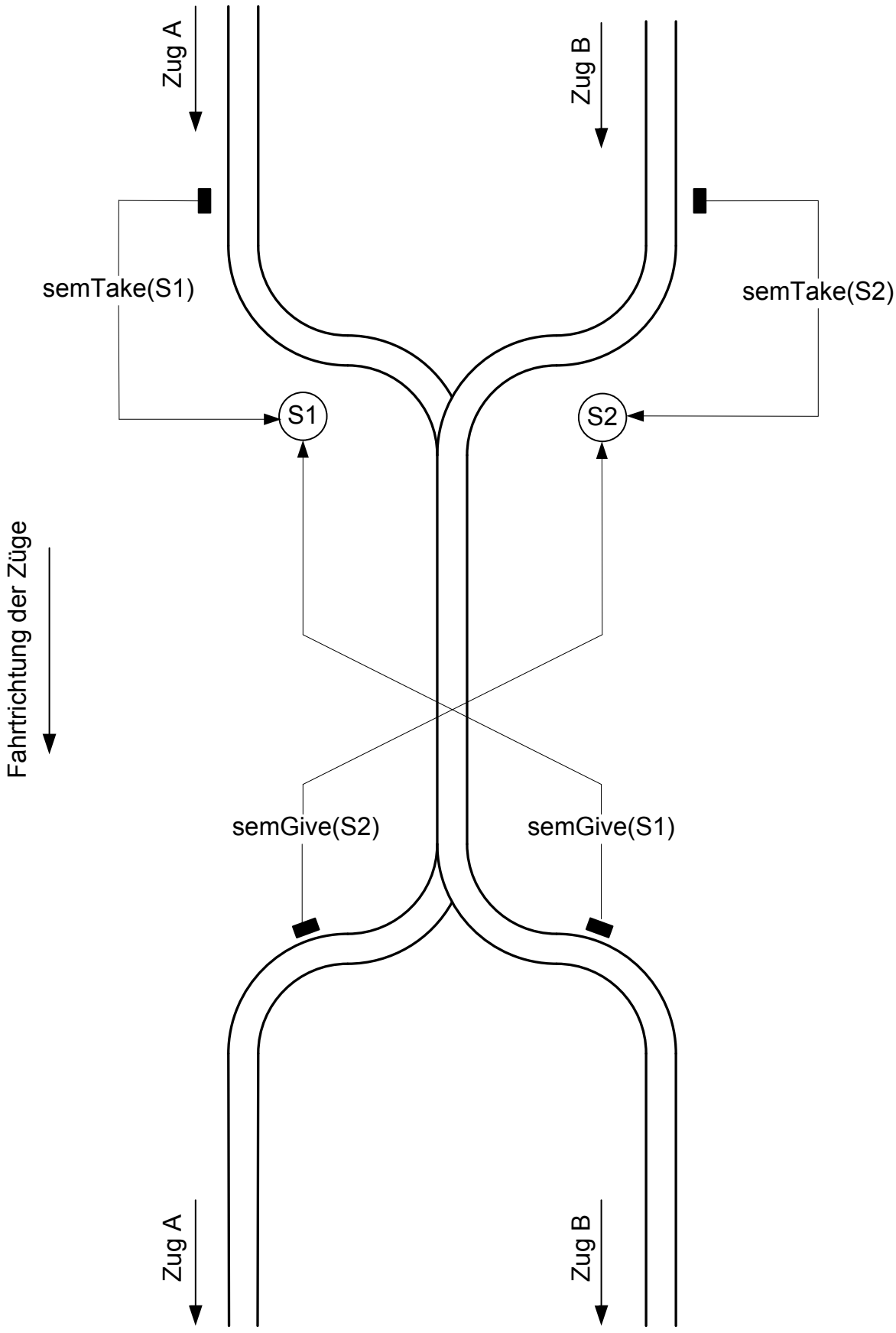


Bild: Analogbeispiel „Zugverkehr über eine eingleisige Strecke“

7.5. Weitere Übungen zur Task-Synchronisation

7.5.1. Übung 1

Lösen Sie das Synchronisationsproblem.

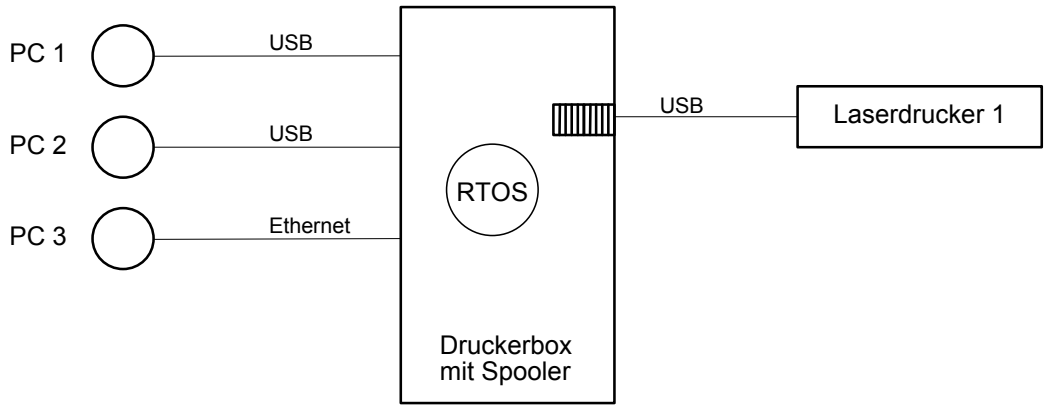


Bild: Synchronisation der Druckerausgabe

7.5.2. Übung 2

Lösen Sie das Synchronisationsproblem.

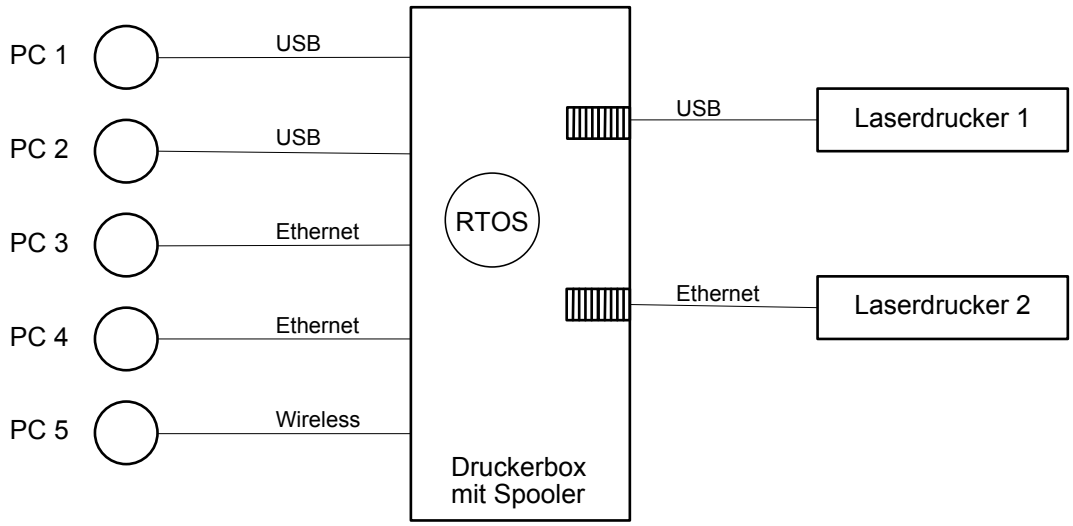
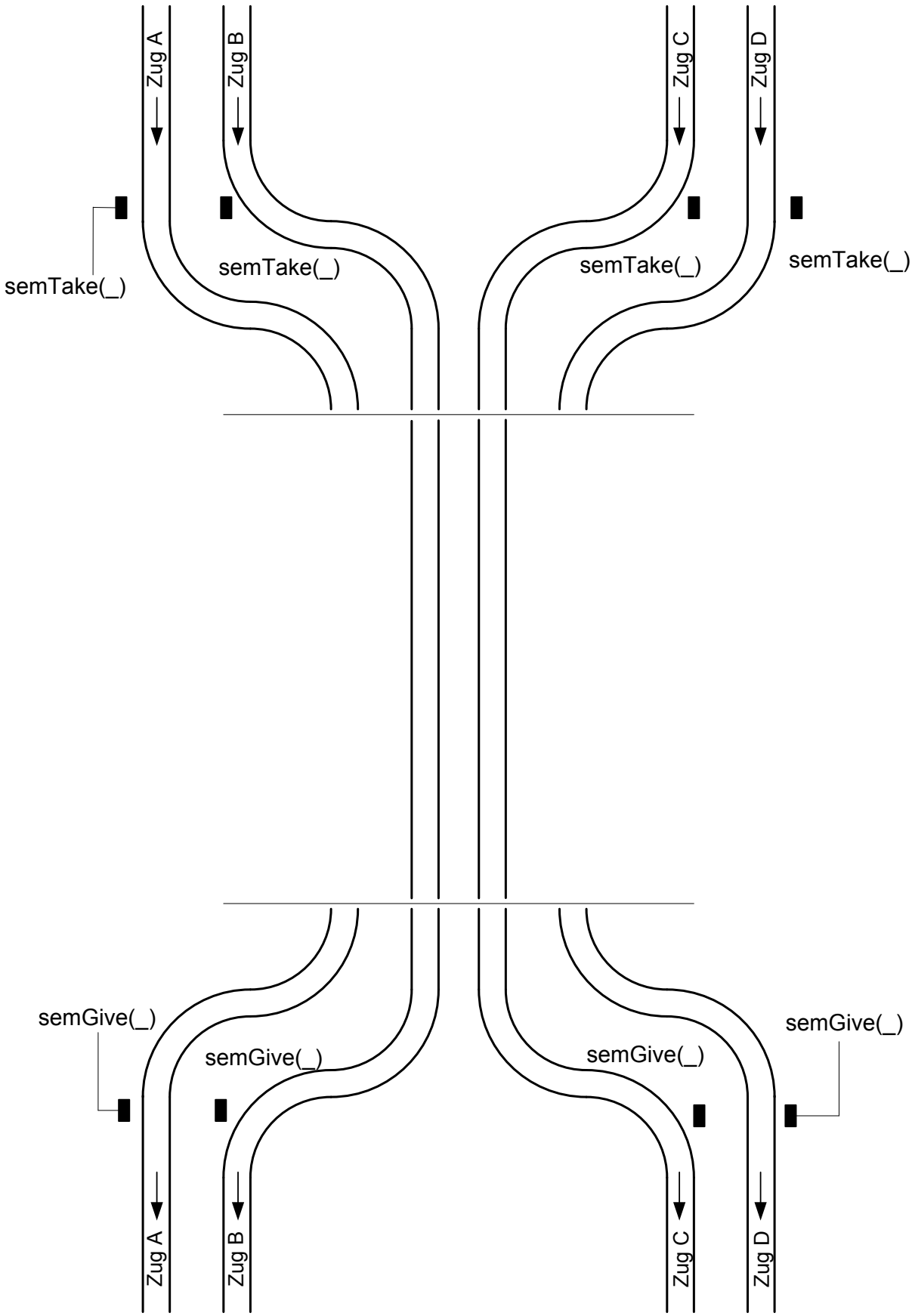


Bild: Synchronisation der Druckerausgabe (Erweiterung)

7.5.3. Übung 3

Lösen Sie das Synchronisationsproblem.



8. TASKKOMMUNIKATION

8.1. Problemstellung und Begriffe

8.2. Verfahren zum Nachrichtenaustausch zwischen Tasks

- **Gemeinsamer Speicher (shared memory)**
- **Warteschlangen (pipes)**
- **Message Passing (Mailbox)**

8.3. Befehlsübersichten

Um Nachrichten versenden zu können, müssen bei der Initialisierung die gewünschten Warteschlangen eingerichtet werden. Bei Beenden einer Task kann reservierter Speicher freigegeben werden, in dem die Warteschlange gelöscht wird.

8.3.1. VxWorks-Funktionen zur Taskkommunikation

Bei VxWorks stehen hierfür die Funktionen msgQCreate() und msgQDelete() zur Verfügung:

MSG_Q_ID = msgQCreate (int maxMsgs, int maxMsgLength, int options)

```
// int maxMsgs          Maximale Anzahl von Nachrichten in der Queue
// int maxMsgLength,    Anzahl Bytes in einer Nachricht
// int options          Sortieralgorithmus, siehe übernächste Zeile
    // MSG_Q_FIFO       Queue wird nach dem FIFO-Prinzip organisiert
    // MSG_Q_PRIORITY   Nachrichten werden nach der Priorität der
    //                  sendenden Task eingeordnet.
// Rückgabewert:        ID der Queue (Pointer) oder NULL
// Hinweis: Funktion kann nicht in einer ISR verwendet werden.
```

MSG_Q_ID = msgQDelete (MSG_Q_ID msgQId)

```
// MSG_Q_ID msgQId      ID der zu löschenden Queue
// Rückgabewert         OK oder ERROR
// Hinweis: Funktion kann nicht in einer ISR verwendet werden.
```

Eine Task sendet bzw. empfängt Nachrichten einer bestimmten Message-Queue mittels des Aufrufes von Funktionen. Als VxWorks-APIs stehen hierfür die Funktionen msgQSendQ und msgQReceiveQ zur Verfügung. Bei einer POSIX-Implemtierung (ebenfalls unter VxWorks verfügbar; mpPxLib) werden u. a. die Funktionen mq_open(), mq_close(), mq_send() und mq_receive() verwendet.

STATUS = msgQSend (MSG_Q_ID msgQId, char* buffer, UINT nBytes,

int timeout, int priority)

// MSG Q_ID msgQId ID der zu benutzenden Queue
// char* buffer Zu sendende Nachricht
// UINT nBytes Länge der Nachricht
// int timeout
// NO_WAIT: Kein Warten, auch wenn die Queue voll ist und die Nachricht nicht gesendet werden kann.
// WAIT_FOREVER: Warten bis Senden möglich ist
// Mit Timeout: Anzahl von Ticks auf Sendebereitschaft der Queue warten.
// int priority
// MSG_PRI_NORMAL: Nachricht an das Ende der Queue hängen
// MSG_PRI_URGENT: Nachricht am Anfang der Queue eintragen
// Rückgabewert: OK oder ERROR, Error wird gesetzt.

STATUS = msgQReceive (MSG_Q_ID msgQId, char* buffer, int timeout)

// MSG_Q_ID msgQId ID der zu benutzenden Queue
// char* buffer Puffer, in den Nachricht geschrieben wird
// UINT maxNBytes Pufferlänge
// int timeout siehe unten
// Timeout:
// NO_WAIT Kein Warten, auch wenn die Queue leer ist
// WAIT_FOREVER Warten bis Empfang möglich ist.
// Mit *timeout* Anzahl von Ticks auf Empfang warten.
// Rückgabewert; Anzahl der empf. Bytes oder ERROR, errno wird gesetzt.

// Hinweis: Funktion kann nicht in einer ISR verwendet werden.

8.4. Programmbeispiele zur Verwendung von Message Queues

8.4.1. Programmbeispiele unter RTOS VxWorks

Beispiel: Aufruf von Nachrichten-Diensten (VxWorks)

```
1   #include "vxWorks.h"
2   #include "msgQLib.h"
3
4   #define MAX_MSGS    10
5   #define MAX_MLEN    100
6   #define MESSAGE     "Greetings form Task1"
7   MSG_Q_ID            myMsgQId
8
9   Initialization()
10   {
11    ...
12    // create message queue
```

```
13    myMsgQId = msgQCreate (MAX_MSGS, MAX_MLEN, MQ_PRIORITY);
14    ...
15   }
16
17
18  Task1()
19   {
20    ...
21    // Send mormal prio message, waiting if queue is full
22    msgQSend (myMsgQId, MESSAGE, sizeof(MESSAGE), WAIT_FOREVER,
23              MSG_PRI_NORMAL);
24    ...
25   }
26
27
28  Task2()
29   {
30    char msgBuf[MAX_LEN];
31    ...
32    // Get message from queue, wait until it's available
33    msgQReceive(myMsgQId, msgBuf, MAX_LEN, WAIT_FOREVER);
34    // printf("Message received: %s\r\n", msgBuf);
35   }
```

```
1 // Beispiel zur Einrichtung und Betreibung von Message Queue
2 #include „msgQLib.h“            // Include Dateien
3
4 #define MAX_MSGS (10)           // Konstantenliste
5 #define MAX_MSG_LEN (100)
6 #define MESSAGE   “Meldung von Task 1“
7 MSG_Q_ID   myMSGQId;
8
9  Task2 (void)
10 {
11  char msgBuf [MAX_MSG_LEN];
12
13  // Hole die Message von der Warteschlange, warten
14  if (msgQReceive (myMSGQId, msgBuf, MAX_MSG_LEN, WAIT_FOREVER) == ERROR)
15     return (ERROR);
16
17  printf (”Message from task 1:\n%s\n”, msgBuf); // Zeige übermitteltes Telegramm
18 }
19
```

```
20
21 Task 1 (void)
22 {
23  // Erzeuge die Message Queue
24  if ((myMsgQId=msgQCreate(MAX_MSGS, MAX_MSG_LEN, MSG_Q_PRIORITY))==NULL)
25       return (ERROR);
26
27  // Sende ein normal priorisiertes Telegramm, und blockiere wenn
28  // die Warteschlange voll ist
29  if (msgQSend(MESSAGE, sizeof(MESSAGE), WAIT_FOREVER, MSG_PRI_NORMAL)==ERROR)
30     return (ERROR);
31 }
```

8.4.2. User Services zur Taskkommunikation beim RTOS Salvo

OSCreateMsg(): Create a Message

Type:	Function
Prototype:	`OStypeErr OSCreateMsg ( OStypeEcbP ecbP, OStypeMsgP msgP );`
Callable from:	Anywhere
Contained in:	`msg.c`
Enabled by:	`OSENABLE_MESSAGE`, `OSEVENTS`
Affected by:	`OSCALL_OSCREATEEVENT`, `OSENABLE_STACK_CHECKING`, `OSCOMBINE_EVENT_SERVICES`, `OSLOGGING`, `OSUSE_EVENT_TYPES`
Description:	Create a message with the initial value specified.
Parameters:	`ecbP`: a pointer to the message's ecb. `msgP`: a pointer to a message.
Returns:	`OSNOERR`
Stack Usage:	1

Notes

Creating a message assigns an event control block (ecb) to the message. A newly-created message has no tasks waiting for it. Messages are passed via pointer so that a message can point to anything.

Signaling or waiting a message before it has been created will result in an error if `OSUSE_EVENT_TYPES` is `TRUE`.

Binary semaphores and resource locking can be implemented via messages using the values `(OStypeMsgP) 0` and `(OStypeMsgP) 1` for the messages.

In the example below, a message is created to pass the key pressed (which is detected by the task `TaskReadKey()`) to the task `TaskHandleKey()`, which acts on the keypress. The message is initialized to zero because no keypress is initially detected. If, due to task priorities and timing, `TaskReadKey()` signals a new message before `TaskHandleKey()` reads the existing message, the new key will be lost.

See Also

`OS_WaitMsg()`, `OSReadMsg()`, `OSSignalMsg()`, `OSTryMsg()`

Example

```
/* pass key via a message. */
#define MSG_KEY_PRESSED_P OSECBP(4)
…
/* this task reads key presses from a keypad   */
/*  and sends them to TaskHandleKey via a      */
/*  message.                                   */
void TaskReadKey(void)
{
  static char key;     /* holds key pressed    */

  /* initially no key has been pressed.        */
  OSCreateMsg(MSG_KEY_PRESSED_P, (OStypeMsgP) 0);

  for (;;)
  {
    if ( kbhit() )
    {
      key = getch();

      /* do debouncing, key-repeat, etc.       */

      /* send new key via message.             */
      OSSignalMsg(MSG_KEY_PRESSED_P,
        (OStypeMsgP) &key);
    }

    /* wait 10msec, then test for keypress     */
    /*  again.                                 */
    OS_Delay(TEN_MSEC, TaskReadKey1);
  }
}

/* this task acts upon keypresses.             */
void TaskHandleKey(void)
{
  static char key;          /* holds new key    */
  static OStypeMsgP msgP;  /* get msg via ptr  */

  for (;;)
  {
    /* do nothing until a key is pressed.      */
    OS_WaitMsg(MSG_KEY_PRESSED_P, &msgP,
      OSNO_TIMEOUT, TaskHandleKey1);

    /* then get the new key and act on it.     */
    key = *(char *)msgP;
    switch ( tolower(key) )
    {
      case KEY_UP:
        MoveUp();
        break;
      …
    }
  }
}
```

OSSignalMsg(): Send a Message

Type:	Macro or Function
Prototype:	`OStypeErr OSSignalMsg ( OStypeEcbP ecbP, OStypeMsgP msgP );`
Callable from:	Anywhere
Contained in:	`msg.c`
Enabled by:	`OSENABLE_MESSAGES`, `OSEVENTS`
Affected by:	`OSCALL_OSSIGNALEVENT`, `OSENABLE_STACK_CHECKING`, `OSCOMBINE_EVENT_SERVICES`, `OSLOGGING`, `OSUSE_EVENT_TYPES`
Description:	Signal a message with the value specified. If one or more tasks are waiting for the message, the highest-priority task is made eligible.
Parameters:	`ecbP`: a pointer to the message's ecb. `msgP`: a pointer to a message.
Returns:	`OSERR_BAD_P` if message pointer is incorrectly specified. `OSERR_EVENT_BAD_TYPE` if specified event is not a message. `OSERR_EVENT_FULL` if message is already defined. `OSNOERR` on success.
Stack Usage:	1

Notes

No more than one task can be made eligible by signaling a message.

In the example below, a message is used (in place of a binary semaphore) to control access to a shared resource, an LCD. When either `TaskDisplay()` or `TaskFlashWarning()` needs to write to the display, it must first acquire the display by successfully waiting on the message `MSG_LCD_RSRC`. Once obtained, the task can write to the LCD. When finished, it must release the resource by signaling the message.

`TaskFlashWarning()` displays a warning message for five seconds by writing to the display and then delaying itself for five seconds before releasing the resource. The use of a message to control access to the LCD prevents `TaskDisplay()` from overwriting the LCD while the warning message is displayed.

See Also

OS_WaitMsg(), OSCreateMsg(), OSReadMsg(), OSTryMsg()

Example

```
#define MSG_DISP_UPDATE_P OSECBP(2)     /* flag */
#define MSG_LCD_RSRC_P    OSECBP(3)     /* rsrc */
#define MSG_WARNING_P     OSECBP(4)     /* flag */
char strLCD[LCD_LENGTH+1]; /* 1 row chars + \0 */

void TaskDisplay(void)
{
  static OStypeMsgP msgP;

  /* display is initially available to all.    */
  OSCreateMsg(MSG_LCD_RSRC_P, (OStypeMsgP) 1);

  for (;;)
  {
    /* wait until display update is required   */
    OS_WaitMsg(MSG_DISP_UPDATE_P, &msgP,
      OSNO_TIMEOUT, TaskDisplay1);

    /* wait if we can't acquire the resource.  */
    OS_WaitMsg(MSG_LCD_RSRC_P, &msgP,
      OSNO_TIMEOUT, TaskDisplay2);

    /* write global string to display.         */
    WriteLCD(strLCD);

    /* free display for others to use.         */
    OSSignalMsg(MSG_LCD_RSRC_P, (OStypeMsgP) 1);
  }
}

void TaskFlashWarning(void)
{
  static OStypeMsgP msgP, msgP2;

  for (;;)
  {
    /* wait for the warning ...                */
    OS_WaitMsg(MSG_WARNING_P, &msgP,
      OSNO_TIMEOUT, TaskFlashWarning1);

    /* grab the LCD, locking others out.       */
    OS_WaitMsg(MSG_LCD_RSRC_P, &msgP2,
      OSNO_TIMEOUT, TaskFlashWarning2);

    /* Flash warning on LCD for 5 seconds.     */
    WriteLCD((char *)msgP);
    OS_Delay(FIVE_SEC, TaskFlashWarning3);

    /* refresh / restore LCD, and free it.     */
    WriteLCD(strLCD);
    OSSignalMsg(MSG_LCD_RSRC_P, (OStypeMsgP) 1);
  }
}
```

OSReadMsg():Obtain a Message's Message Pointer Unconditionally

Type:	Function
Prototype:	`OStypeMsgP OSReadMsg ( OStypeEcbP ecbP );`
Callable from:	Anywhere
Contained in:	`msg.c`
Enabled by:	`OSENABLE_MESSAGES`, `OSENABLE_EVENT_READING`, `OSEVENTS`
Affected by:	`OSCALL_OSRETURNEVENT`
Description:	Returns a pointer to the message specified in `ecbP`.
Parameters:	`ecbP`: a pointer to the message's ecb.
Returns:	Message pointer.
Stack Usage:	1

Notes

`OSReadMsg()` has no effect on the specified message. Therefore it can be used to obtain the message's message pointer without affecting the state(s) of any task(s).

No error checking is performed on the `ecbP` parameter. Calling `OSReadMsg()` with an invalid `ecbP`, or an `ecbP` belonging to an event other than a message, will return an erroneous result.

In the example below, a task checks to see if a message is non-empty before signaling the message.[82] Thus it avoids losing the message.

See Also

`OS_WaitMsg()`, `OSCreateMsg()`, `OSSignalMsg()`, `OSTryMsg()`

Example

```
/* send this when there's a problem.           */
const char strImpMsg[] = "Important Message!\n";

void TaskC (void)
{
  for (;;)
  {
    …
    /* delay one system tick as long as MSG    */
    /*  has a message in it.                   */
    while ( OSReadMsg(MSG_P) )
      OS_Delay(1, TaskC1);

    /* now that MSG is empty, we can send our  */
    /*  important message.                     */
    OSSignalMsg (MSG_P, (OStypeMsgP) &strImpMsg);
  }
}
```

8.4.3. Übung zur Taskkommunikation

Bitte erläutern Sie die Wirkungsweise der nachstehenden Services unter Salvo RTOS.

- OS_CreateMsgQ()
- OSSignalMsgQ()
- OSReadMsgQ()
- OSTryMsg()
- OSTryMsgQ()

8.5. VxWorks Library-Funktionen zu Message Queues

msgQLib - message queue library

ROUTINES

***msgQCreate*( )** - create and initialize a message queue
***msgQDelete*( )** - delete a message queue
***msgQSend*( )** - send a message to a message queue
***msgQReceive*( )** - receive a message from a message queue
***msgQNumMsgs*( )** - get the number of messages queued to a message queue

DESCRIPTION

This library contains routines for creating and using message queues, the primary intertask communication mechanism within a single CPU. Message queues allow a variable number of messages (varying in length) to be queued in first-in-first-out (FIFO) order. Any task or interrupt service routine can send messages to a message queue. Any task can receive messages from a message queue. Multiple tasks can send to and receive from the same message queue. Full-duplex communication between two tasks generally requires two message queues, one for each direction.

CREATING AND USING MESSAGE QUEUES

A message queue is created with ***msgQCreate*( )**. Its parameters specify the maximum number of messages that can be queued to that message queue and the maximum length in bytes of each message. Enough buffer space will be pre-allocated to accommodate the specified number of messages of specified length.

A task or interrupt service routine sends a message to a message queue with ***msgQSend*( )**. If no tasks are waiting for messages on the message queue, the message is simply added to the buffer of messages for that queue. If any tasks are already waiting to receive a message from the message queue, the message is immediately delivered to the first waiting task.

A task receives a message from a message queue with ***msgQReceive*( )**. If any messages are already available in the message queue's buffer, the first message is immediately dequeued and returned to the caller. If no messages are available, the calling task will block and be added to a queue of tasks waiting for messages. This queue of waiting tasks can be ordered either by task priority or FIFO, as specified in an option parameter when the queue is created.

TIMEOUTS

Both *msgQSend*( ) and *msgQReceive*( ) take timeout parameters. When sending a message, if no buffer space is available to queue the message, the timeout specifies how many ticks to wait for space to become available. When receiving a message, the timeout specifies how many ticks to wait if no message is immediately available. The *timeout* parameter can have the special values NO_WAIT (0) or WAIT_FOREVER (-1). NO_WAIT means the routine should return immediately; WAIT_FOREVER means the routine should never time out.

URGENT MESSAGES

The *msgQSend*( ) routine allows the priority of a message to be specified as either normal or urgent, MSG_PRI_NORMAL (0) and MSG_PRI_URGENT (1), respectively. Normal priority messages are added to the tail of the list of queued messages, while urgent priority messages are added to the head of the list.

INCLUDE FILES

msgQLib.h

msgQShow - message queue show routines

ROUTINES

msgQShowInit( ) - initialize the message queue show facility
msgQInfoGet( ) - get information about a message queue
msgQShow( ) - show information about a message queue

DESCRIPTION

This library provides routines to show message queue statistics, such as the task queuing method, messages queued, receivers blocked, etc.

The routine *msgQshowInit*( ) links the message queue show facility into the VxWorks system. It is called automatically when the message queue show facility is configured into VxWorks using either of the following methods:

- If you use the configuration header files, define **INCLUDE_SHOW_ROUTINES** in **config.h**.
- If you use the Tornado project facility, select **INCLUDE_MSG_Q_SHOW**.

INCLUDE FILES

msgQLib.h

9. INTERRUPTSTEUERUNG

Interrupt-Behandlung

- Wozu sind Interrupts nützlich?
- Reaktionsstruktur (Interrupt-Controller)
- Interrupt Service Routine (zur Signalisierung)
- Interrupt Service Task (oder Thread)
- Latenzzeiten

Überblick Interrupt-Behandlung
Zweistufige Reaktionsstruktur

Die Sprachmittel zur Reaktion und Nutung von sogenannten asynchronen Ereigenissen wie z. B. Interrupt, Signale, spezielle Elemente zur zeitlichen Steuerung und Behandlung von Ausnahmen, zeichnen Echtzeitsysteme aus. In der gezielten Nutzung dieser Sprachmittel besteht wohl der Hauptunterschied zur Programmierung in gewöhnlichen „Betriebssystemen“.

Ja nachdem um welches Sprachmittel es sich handelt wird der reguläre Programm-ablauf von externen (Interrupts) oder internen Ereignissen (Signalen) unterbrochen. Interrupts sind im Prinzip elektrische Signale, die direkt oder über sogenannte Interrupt Controller an die CPU gemeldet werden. Setzt man voraus, dass Echtzeitsysteme für kürzeste Reaktionszeiten konstruiert sind, so muss jeder externe Interrupt in der Lage sein, die aktuell ablaufenden Task zu unterbrechen.

Die Zeit, die vom Anlegen des Interrupts bis zum Start der entsprechenden Interrupt-Serviceroutine vergeht, nennt man „Interrupt-Laufzeit“. Sie wird oftmals zum Vergleich der verschiedenen Echtzeitsysteme herangezogen.

Typischerweise Interrupt-Latenzzeiten liegen bei heutigen Echtzeitbetriebssystemen im Bereich um 10 Millisekunden. Sie hängen natürlich neben der Effizienz des Betriebssystems wesentlich von der eingesetzten CPU ab.

9.1. Signale und Ausnahmebehandlung

Signale sind interne Programmunterbrechungen, die grundsätzlich ebenfalls Tasks unterbrechen. Damit Signale Tasks unterbrechen können, muss innerhalb der Task eine Art „Signalservice Routine“ installiert sein, die beim Auftreten eines passenden Signals aktiviert wird.

Das folgenden Bild zeigt die Wirkung von Signalen.

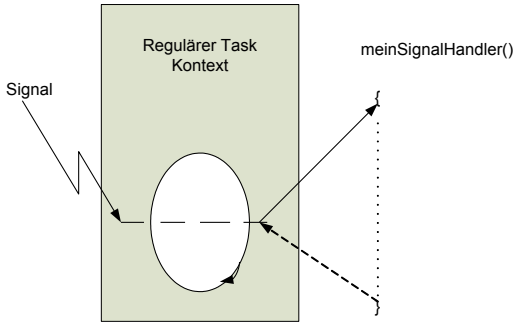


Bild: Zusammenwirken von Signalen und Tasks

Signale sind im Prinzip „Software Interrupts“. Ein Signal zeigt ein asynchrones Ereignis an. Innerhalb VxWorks können 31 vordefinierte Ereignisse verwendet werden. Um auf ein Signal reagieren zu können wird der Task ein sogenannter „Signal Handler“ installiert. Dieser Handler wird bei Auftreten des Signals ausgeführt. Nachdem die Signalroutine abgelaufen ist, wird die Task an der Stelle der Unterbrechung fortgesetzt. Ist keine Signalroutine installiert, wird ein aktiviertes Signal ignoriert. Eine Besonderheit tritt dann auf, wenn sich eine Task im blockiertem Zustand befindet. In diesem Fall wird die Task aus der regulären Warteschlange der Task herausgenommen. Danach wird die Signalroutine aktiviert und kommt als nächste Task zum Ablaufen. Ist die Signalroutine abgearbeitet wird die Task wieder in die Warteschlange eingereiht. Die Task befindet sich wieder im „blockierten“ Zustand.

Grundsätzlich stellen Signale eine Möglichkeit dar, wichtige Operationen asynchron mit hoher Priorität zur Ausführung zu bringen. Signale sollten aufgrund der Prioritätsverschiebung und der asynchronen Ereignisse sehr sparsam verwendet werden. Es kann unter bestimmten Umständen zu schwerwiegenden Ablaufproblemen kommen. Ein Beispiel ist ein Ressourcenkonflikt mit Beteiligung einer Signalroutine. Nimmt man an, dass eine Task eine Semaphore nimmt, die z. B. eine Systemressource sicher. Wird im Anschluss eine Signalroutine aktiviert, die ebenfalls diese Semaphore nehmen will, kommt es zu einer Systemblockierung, einem sogenannten „Dead Lock“.

Eine verbreitete Anwendung von Signalen ist die sogenannte „Shut down“ Funktion für Tasks. Hierbei nützt man die hohe dynamische Priorisierung von Signalroutinen aus, die prinzipiell unabhängig vom Taskzustand zu jeder Zeit aufgerufen werden können. Damit ist es möglich eine Task zu jedem Zeitpunkt quasi augenblicklich aus dem System zu entfernen. Unter Unix ist dieser Mechanismus als „kill“ Anweisung bekannt. Die Syntax zur Registrierung einer Signalroutine würde unter VxWorks wie folgt aussehen:

Registrierung im System

Signal (signo, handler);

Signo	Signalnummer (1..32)
Handler	Funktion, die aufgerufen wir, wenn das Signal aktiviert wird

Deklaration der Signalroutine

void sigHandler (int sig);

Eine weitere Besonderheit im Umgang mit Signalen ist die Verknüpfung von Signalen mit sogenannten Ausnahmebehandlungen (Exceptions). In VxWorks werden z. B. Signale benutzt, um bei Systemfehlern sogenannte Ausnahmebehandlungen anzustoßen. Geläufig ist z. B. ein Fehler namens „bus error“, der auftritt wenn auf einen nicht vorhandenen Speicher zugegriffen wird. Falls in diesem Fall eine Signalroutine installiert ist, wird diese aufgerufen. Wenn keine Routine installiert ist, wird die Task, die den Fehler provozierte, angehalten und aus dem System genommen und eine Fehlermeldung (log error message) am Bildschirm produziert.

9.2. Interrupts (Hardware Systemunterbrechungen)

Interrupts bezeichnen Systemberechnungen, die direkt via Hardwareleitung auf den Systemprozessor wirken.

Die CPU reagiert dabei konkret auf die Veränderung eines Spannungspegels an ihrem Eingang. Je nach CPU und Ausstattung unterliegt die Behandlung eines Interrupts zuerst festen Abläufen, die in der Hardware und der Systemsoftware festgelegt werden.

Zur programmseitigen Behandlung kann eine Anwenderroutine mit einem Interrupt verknüpft werden. Diese Routine, die sogenannte Interrupt-Serviceroutine läuft im sogenannten Interrupt-Kontext. Interruptkontext bedeutet vor allem ein von dem Tasks unabhängiges Prioritätsschema und einen speziellen Speicherbereich.

Ganz typisch für Echtzeitsysteme sind Timerbausteine auf Rechnerbords. Diese Timerbausteine erlauben es i.d.R. mittels Programmierung einen Hardwareinterrupt auf der CPU zu generieren.

Das nachstehende Bild zeigt die Wirkungsweise und die Behandlung von Interrupts in Echtzeitsystemen.

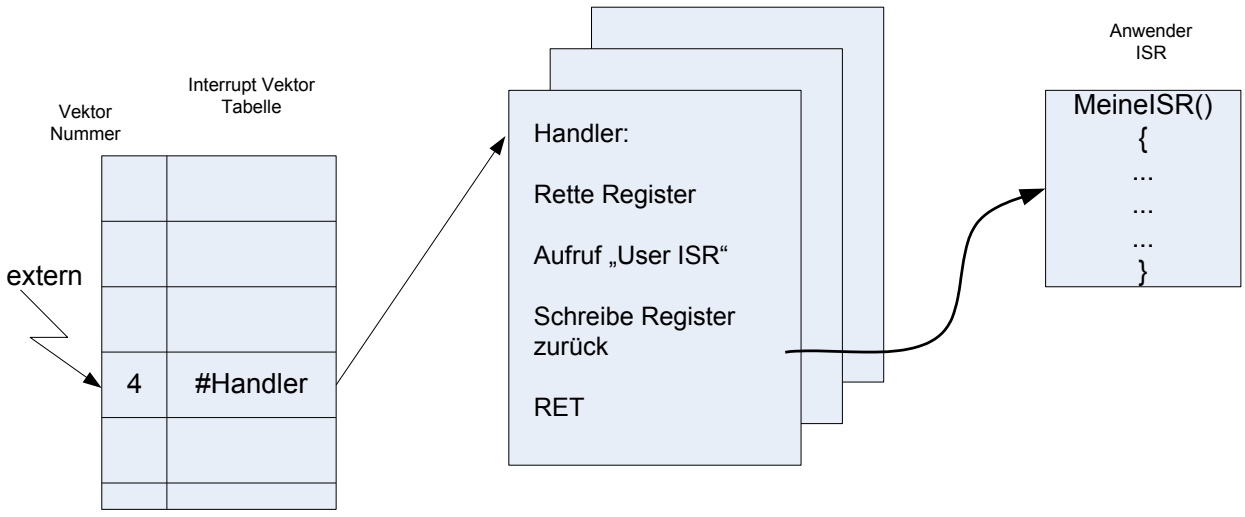


Bild: Bearbeitung eines Interrupts

Eine Interruptbehandlung läuft folgendermaßen ab:

1. Interrupt wird an den Interruptanschluss des Prozessors gelegt
2. Entsprechend der Interrupt-Vektortabelle wird hierzu der entsprechende Interrupt-Handler aktiviert.

3. Der Prozessor wechselt zu einem speziellen Interruptstask oder benutzt den Stack der aktuellen Task.
4. Im Interrupt-Handler, der zur Systemsoftware des Betriebssystems gehört, werden die wichtigsten Prozessorregister (architekturabhängig) gesichert.
5. Jetzt wird die Anwender-Interrupt-Serviceroutine (ISR) aufgerufen. Dort wird die vom Anwender vorgesehene Aufgabe im sogenannten Interrupt-kontext ausgeführt.
6. Nach Beendigung der ISR wird der Handler aktiviert. Dort werden die ursprünglichen Prozessorregister wieder hergestellt. Nach Beendigung der Handlerroutine ist die Interruptbearbeitung abgeschlossen.

Eigenschaften von Interrupts:

- **Interrupt-Serviceroutinen sind keine Tasks**.
- Eine sehr wichtige Eigenschaft von Interrupts ist die asynchrone Unterbrechungs-möglichkeit des Taskablaufs. **Dies ist möglich, weil Interrupts grundsätzlich eine höhere Priorität als alle im System vorhandenen Tasks haben.** Bild 3.16 zeig das Prioritätsschema in Echtzeitsystemen wie z. B. VxWorks.

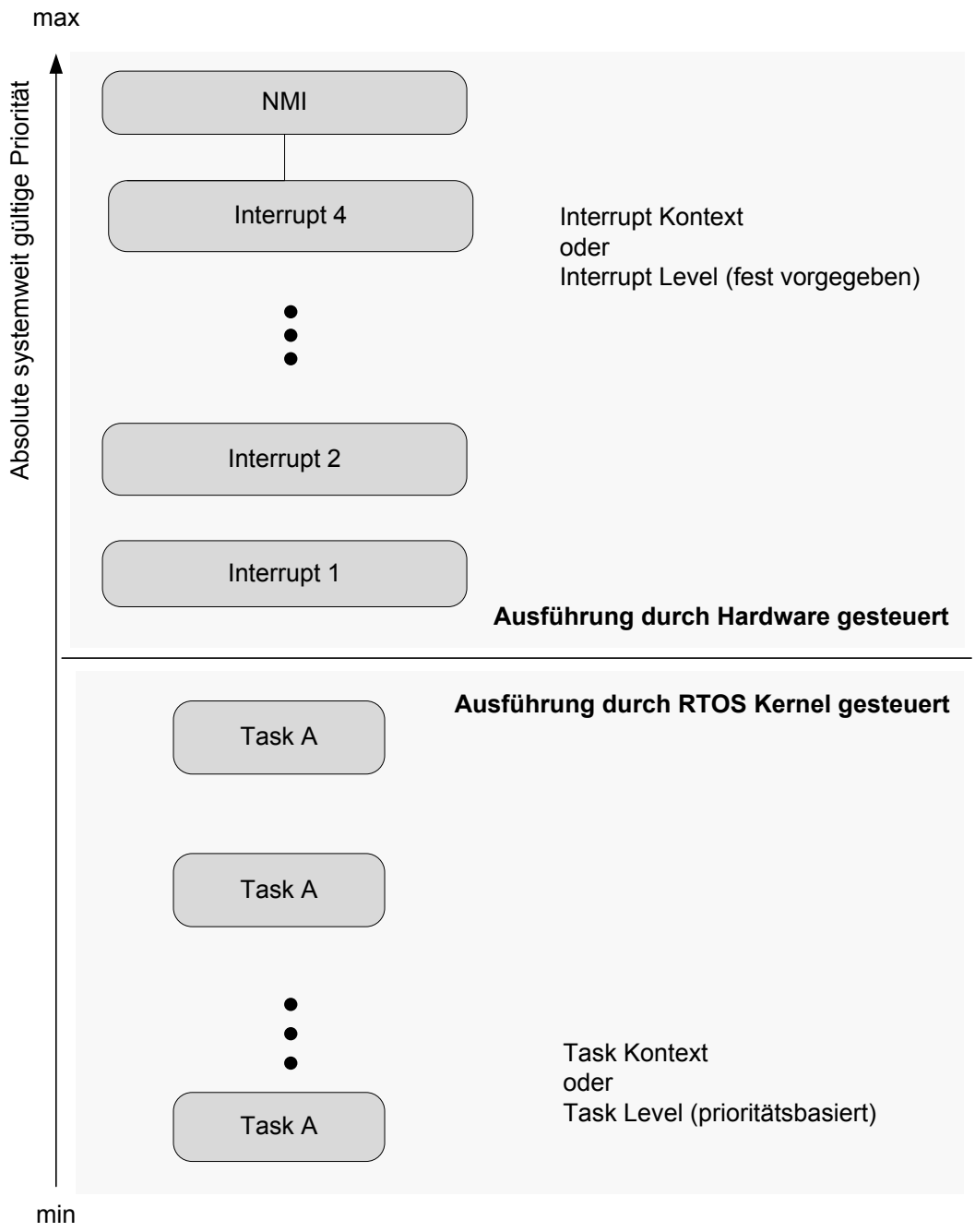


Bild 3.16 Prioritätsschema in Echtzeitsystemen

Für den Fall, dass für die auftretenden Interrupts ein Systemspeicherbereich vorhanden ist, muss dieser groß genug sein, alle Interrupts aufzunehmen. Je nach CPU und Echtzeitsystem kann der Interrupttask auch auf Task verteilt sein. Auch hier ist genügend Platz für aufeinanderfolgende Interrupts vorzusehen.

In Echtzeitsystemen wie z. B. VxWorks gelten für Interrupt-Serviceroutinen besondere Einschränkungen, da bestimmte Funktionsaufrufe gelten:

- Da Interrupts eine sehr hohe Priorität besitzen, würde ein blockierter Interrupt, z. B. aufgrund einer Semaphore, zu erheblichen Problemen bis hin zum Systemabsturz führen. Aus diesem Grund sind alle Funktionsaufrufe, die blockieren können, im Interruptkontext verboten. Dazu gehören z. B. semTake(), MessageQRead().

- Aufgrund des Interruptkontextes kann i.d.R. kein Systemspeicher dynamisch allokiert werden, d.h. malloc() ist nicht aufrufbar.
- Ferner sind Funktionen, die den Standard-I/O Funktionsaufrufe wie printf() beinhalten, nicht erlaubt. Innerhalb einer ISR wird typischerweise in Register geschrieben, die direkt im Systemspeicher deklariert sind. Für Datenübergaben kann z. B. nicht blockierend auf Message Queues geschrieben werden. **In vielen Fällen wird auch einfach eine Semaphore gegeben, die ihrerseits eine Task zum Ablaufen bringt.** Diese Task erledigt dann die Aufgaben, die an das asynchrone Ereignis angebunden sind. **Damit wird erreicht, dass die Serviceroutine von kurzer Dauer ist.**

Für den Gebrauch von Interrupt-Serviceroutinen gelten eine Reihe von Regeln:

- ISRs sind möglichst kurz zu halten, da
  - andere Interrupts verzögert werden
  - alle Tasks im System während der Ausführung der ISR blockiert sind.
  - Fehlerzustände im System während einer ISR schwer feststellbar sind.

- Generell sollten Fließkommaoperationen vermieden werden, da
  - Operationen mit Fließkommaanteilen relativ langsam sind
  - Es oftmals notwendig ist, Fließkommaregister der CPU manuell zu sichern.

- Es sollten möglichst viele Aufgaben an eine angehängte Task übertragen werden, weil
  - Tasks unterbrochen werden könne, und somit das Gesamtsystem reaktionsfähig bleibt.
  - der Absturz oder der Fehler einer einzelnen Task weniger kritisch für das System ist.

Kommt es während der Bearbeitung eines Interrupts zu einem weiteren Interrupt, der eine höhere Priorität hat als der aktuelle, so wird der aktuelle Interrupt unterbrochen und der höher priore Interrupt kommt zur Bearbeitung. Während dieser Zeit bleiben alle Tasks blockiert.

Kommt es zu einer Ausnahmebehandlung während eines Interrupt werden je nach Echtzeitsystem verschiedene Strategien angewendet. In VxWorks führt ein Ausnahmefehler zu einem Warmstart des Systems. Das bedeutet, der Programmablauf verzweigt in eine Systemunterbrechung, kurz „trap“, die unmittelbar einen Bootvorgang im Boot Eprom startet. Dies hat einen sogenannten Warmstart zu Folge. Ein Warmstart ist ein Systemneustart, ohne dass der Arbeitsspeicher mit null überschrieben wird.

Priorisierung von Interrupts

Eine weitere Besonderheit von Interrupts betrifft die Priorisierung. Hier spielt zum einen das Echtzeitsystem als auch die verwendete Hardware eine wesentliche Rolle. Moderne Systeme besitzen einen sogenannten Interrupt Controller. Das ist eine Hardwareeinheit, die den Prozessor in der Behandlung von Interrupts unterstützt. Im Controller wird meist per Hardware bereits eine Prioritätsverteilung über die Leitung realisiert. Zudem bieten manche Controller die Möglichkeit, gezielt Prioritäten zu programmieren. I.d.R. werden die Interrupts auch über Warteschlangen gespeichert und gezielt an die CPU zur Verarbeitung mitgeteilt. Wird der Interruptcontroller vom entsprechenden Echtzeitsystem unterstützt steht es dem Anwender offen, die Programmiermöglichkeiten der Hardware zu nutzen. Im anderen Fall werden Standardwerte (Default) eingestellt, und die Verarbeitung erfolgt über die CPU.

Grundsätzlich ist die Interruptbehandlung sehr schnell, da Interrupts die höchste Systempriorität besitzen. Gleichwohl gibt es Fälle, wie z. B. bestimmte Steuerungsaufgaben, die jeglichen Overhead bei der Interruptverarbeitung verbieten. Betrachtet man z. B. ein Echtzeitsystem, das unter anderem die Regelung eines elektrischen Antriebs übernimmt. Genügt es für den Prozessteil Interruptlatenzzeiten im Bereich von 10 bis 50 Mikrosekunden zu garantieren, erfordert eine hochdynamische Motorregelung einen Abtastjitter kleiner als 5 Mikrosekunden. Diese Zeitanforderungen können heute mit „Standard-Echtzeitbetriebssystemen“ nicht realisiert werden. Dafür bedarf es je nach System einger Tricks, um quasi am System vorbei eine Bearbeitung zu realisieren, die nur von der Rechenleistung der eingesetzten Hardware abhängt. Dazu existiert z. B. in VxWorks eine Einrichtung, die es erlaubt, Systeminterrupts das Echtzeitbetriebssystem zu sperren:

Int LockLevelSet() // Befehl für den Ausschluss von Interruptprioritäten

Angenommen eine CPU besitzt 7 Prioritätsebenen für Interrupts, dann würde man z. B. die Interruptpriorität auf maximal 5 setzen. Das bedeutet die Interrupts von 1 (niedrigste Priorität) bis Ebene 5 würden als reguläre Systeminterrupts abgearbeitet. Wird ein Interrupt auf Leitung 6 oder 7 gelegt, so reagiert das system nicht mit den regulären Routinen sondern verzweigt direkt zu einer Anwenderroutine, die außerhalb des Betriebssystemskontextes liegt. Mit diesem Mechanismus wird im Prinzip eine Null-Interruptlatenzzeit erreicht. Hierbei sollte man sehr vorsichtig sein, da der Systemkontext bei der Systemunterbrechung und anschließenden Sonderbehandlung nicht Schaden nehmen darf, da sonst ein Systemabsturz nach Bearbeitung der Unterbrechung unweigerlich ist. Um zu klären, welche Interruptleitung für diesen Sonderfall verwendet werden können, muss in jedem Fall der Prozessor angeschaut werden.

Zusammengefasst kann man sagen, dass asynchrone Systemunterbrechungen (Interrupts), eine Grundlage für Echtzeitfähigkeit eines Systems liefern. Damit kann in vorhersagbaren Zeiten auf externe Ereignisse reagiert werden. Die Sprachmittel zur Einrichtung, Verwaltung und Aufgabenerledigung hängen von der Hardware und dem Echtzeitsystem ab. Grundsätzlich besitzen Interrupts stets eine höhere Priorität als die im System vorhandenen Tasks, die Systemaufrufe sind eingeschränkt und sie sollten

von möglichst kurzer Dauer sein, um die Reaktionsfähigkeit das Gesamtsystems möglichst hoch zu behalten. Da es auch hier unterschiedliche Prioritäten gibt, muss der Ablauf und insbesondere die Interaktion mit den Tasks im System sehr genau im Voraus überlegt werden um Überraschungen zu vermeiden, die zu Systemfehlern führen können.

10. ZEITDIENSTE

- Was sind Zeitdienste?
- Anwendungsbeispiele/Funktionen der Zeitverwaltung

- Realisierung durch Timer (meist Software-Timer, jedoch auch Hardware-Timer möglich)

- VxWorks Zeitdienste:
  Systemzeit
  Kalenderzeit

-Anwendung:Takterzeugung im Praktikum
  Überblick Zeitdienste

10.1. Einsatzgebiete

In Embedded Systemen sind Funktionalitäten notwendig, die es erlauben zeitlich abhängige Aktivitäten zu starten, zu überwachen, oder zu stoppen. Aus diesem Grund sind in Echtzeit-Betriebssystemen wesentlich mächtigere Zeit-Funktionen zu finden als in Desktop-Betriebssystemen. Diese Zeit-Funktionalitäten sind teilweise mit anderen Funktionen gekoppelt (z.B. Timeout-Mechanismen bei Nachrichtendiensten).

Die Zeitverwaltung erlaubt das

- zeitgesteuerte Senden von Ereignissen zum Aufwecken von Tasks nach einem *relativen* oder nach einem *absoluten* Zeitraum,
- die Realisierung von Timeout-Mechanismen beim Warten auf Threadgenerierung, Speicherallokierung, der Semaphoren-Freigabe, Ereignissen und Nachrichten,
- die Bestimmung von Zeitscheiben bei bestimmten Scheduling-Algorithmen ermöglichen (Round Robin- Scheduling-Verfahren),
- die Verwaltung von Datum und Uhrzeit (Kalender),
- ermöglicht Zeit- und Performancemessungen (Profiling, Instrumentierung).

Funktionen zum Senden von zeitlich gebundenen Ereignissen erlauben der Sende-Task nach dem Senden eines Zeit-Ereignisses beliebig weiter zuarbeiten (z. B. Starten des Timers) oder bei Wakeup-Mechanismen sich verdrängen zu lassen, um sich nach

Ablauf der eingestellten Zeit vom Scheduler aktivieren zu lassen.

Das gesendete Ereignis wird nach Ablauf der gewählten Zeit (relativ, absolut) in die Ergeignisqueue der Empfangstask eingereiht. Abhängig von der gewählten Funktionalität können ein oder mehrere Aktionen vom Betriebssystem oder Task ausgeführt werden. Innerhalb des Zeitablaufes kann die Zustellung durch weitere Funktionen verhindert werden (z. B. zur Realisierung eines WatchDog-Mechanismus).

> **Definition: WatchDog**
>
> Ein WatchDog ist eine unabhängige Überwachungseinheit, die durch ein Signal regelmäßig zurückgesetzt werden muss, damit durch die Einheit kein Alarm oder Ausnahme-Behandlung ausgelöst wird.

Realisierung der Timer

Um eine gewisse Unabhängigkeit von der Hardware zu gewährleisten, werden Zeitdienste in Echtzeit-Betriebssystemen durch Software-Timer realisiert. Für spezielle Aufgaben können aber auch Hardware-basierte Timerfunktionalitäten von Betriebssystemen unterstützt werden.

Zur Realisierung von Software-Timern wird mindestens ein durch die Hardware aktivierter periodischer Interrupt benötigt (i.d.R. der System-Tick). Die Frequenz des periodischen Interrupts legt die maximal verwendbare Zeitauflösung fest. Je kürzer die Periode der Interruptaktivierung ist, um so stärker wird das System belastet (typische Perioden liegen im Millisekundenbereich).

Alle aktiven Timer werden in einer zeitlich geordneten Liste vom zentralen Zeitdienst verwaltet. Bei jedem Auftreten des periodischen Interrupts werden die in der Liste stehenden Timer-Strukturen auf Ablauf ihrer eingestellten Zeit überprüft. Gegebenenfalls werden diese aus der Liste entfernt und die vereinbarte Funktionalität vom Betriebssystem aktiviert (Ereignis senden, Task aufwecken, etc.).

Durch die Verwaltung der Timer in einer zeitlich geordneten Liste, entstehen bei Starten und Stoppen von Timern bei der Listen-Sortierung der Timer-Strukturen höhere Verwaltunksaufwände. Die einfache Überprüfung der Timer-Strukturen bei jedem periodisch auftretenden Hardware-Interrupt rechtfertigt jedoch diese Realisierung.

Ein periodisch auftretender Timer (der des Hardware-Interrupts) ermöglicht es, dass auch andere Betriebssystemdienste implizit Zeitdienste nutzen können:

- Timeout: Auf Ereignisse, etc. kann eine gewisse Zeit gewartet werden. Tritt innerhalb dieser Wartezeit das Ereignis nicht ein, so wird der Task nach der Wartezeit vom Zustand „wartend" in den Zustand „lauffähig" gebracht. Die Anwendung wird über den Ablauf der Wartezeit informiert (Rückgabewert der Funktion).
- Scheduling: Bestimmte Scheduling-Algorithmen benötigen einen Zeitgeber (Zeitscheibenverfahren). Die Aktivierung einer neuen Zeitscheibe ist nur ein Ereignis! Hierzu wird in der Regel ebenfalls der Timer-Interrupt genutzt.

- Echtzeituhr: Setzen und verwalten einer Echtzeituhr (Uhrzeit, Datum) erfolgt über Zeitdienste.

10.2. VxWorks Zeitdienste - Übersicht

10.2.1. Systemzeit

Funktionen zur Ausführung des Scheduling und zur Intertask-Kommunikation benötigen ein zeitliches Raster. Hierzu wird ein Timer des Prozessors so initialisiert, dass er in einem bestimmten Takt Interrupts auslöst, die ihrerseits bestimmte Funktionen des Systems bewirken.

Die Zeit wird üblicherweise in Ticks (Anzahl System-Interrupts) gemessen. Die Tickfrequenz kann gesetzt und abgefragt werden durch:

```
STATUS  sysClkRateSet (Int ticks)      // Setzt Ticks pro Sekunde (default 60)
                                       // Rückgabewert: OK oder ERROR

int sysClkRateGet (void)               // Ticks pro Sekunde
                                       // Rückgabewert: Tickfrequenz
```

In einem Tickcounter, er wird bei Start auf Null gesetzt, werden die Ticks akkumuliert. Diesen Zähler kann man ebenfalls setzen und abfragen. Weiterhin kann die seit dem Systemstart vergangene Zeit ermittelt und umgerechnet werden:

```
void    tickSet (ULONG ticks)   // Rückgabewert: Neuer Zählerstand gesetzt

ULONG   tickGet (void)          // Rückgabewert: Aktueller Zählerstand

int     timeGet (void)          // Rückgabewert: Zeit in Millisekunden

ULONG   tickToTime (ULONG ticks)
                                // Rückgabewert: Ticks in Millisekunden umgerechnet
```

Weitere Funktionen sind

```
STATUS  timeGet()
```

oder

```
STATUS rateSetCl (int ticks)    // Kombination von sysClkRateSetQ und tickSet()
                                // Rückgabewert: OK oder ERROR
```

Zur Erzeugung einer zufälligen Zeitspanne At dient folgende Funktion:

```
ULONG    tickRand (ULONG t)      //Ticks
                                 // Rückgabewert: Es gilt 2 < At < t
```

Eine Task eine festgelegte Zeitspanne rechnen zu lassen, wird realisiert durch:

```
void  tickToCompute (int t)      // t = Anzahl Ticks
                                 // Hinweis: Die rufende Task rechnet in dieser
                                 // Funktion die gegebene Anzahl von Ticks
```

10.2.2. Kalenderzeit

Die Echtzeituhr kann gesetzt werden durch:

```
void     dateSet ( unsigned char tag,
                   unsigned char datum,
                   unsigned char monat,
                   unsigned char jahr,
                   unsigned char std,
                   unsigned char min)
                         // Rückgabewert: Echtzeituhr wurde
```

Die Kalenderzeit wird aus der Echtzeituhr des Targetrechners als String gebildet. Er hat das Format: Mo 16.1.96 3:52:13

```
CHAR     *dateAndTime (void)
                   // Rückgabewert: Pointer auf die Zeichenkette
```

10.2.3. Takterzeugung im Praktikum

Der eingesetzte Prozessor besitzt neben dem für das Systemtiming eingesetzten Zähler einen weiteren Timer. der für das Praktikum eingesetzt wird. Er löst, solange er aktiviert ist, periodisch einen Interrupt aus. In der zugehörigen Interrupt Service Routine (ISR) wird ein Semaphor signalisiert.

Der Zusatztimer kann im Bereich von

$$3[\text{ticks/s}] < f_b < 5000\ [\text{ticks/s}]$$

eingestellt werden. Weiter können für das Semaphor Signalisierungsfrequenzen bzw. Taktzeiten eingestellt werden:

$$F = f_b / \omega \text{ und } T = \omega / f_b \quad \text{mit } \omega > 0$$

Für das Verbinden des Interrupts mit einem Semaphor, dem Einstellen des Basistaktes

und dessen Abfrage, sowie Aktivieren und Deaktivieren des Timers dienen die Funktionen:

```
SEMID    taktConnect(void)
        // Rückgabewert: ID einer binären Semaphore oder NULL
        // Hinweis:      Das Semaphor wird mit SEM_Q_PRIORITY und SEM_EMPTY
        //               erzeugt. Der Zusatztimer ist deaktiviert.

STATUS   taktSet(int fb)  // Interrupts pro Sekunde
        // Rückgabewert: OK oder ERROR falls Rate unzulässig oder Timer defekt ist
        // Hinweis: Timer wird nicht aktiviert

int      taktGet(void)            // Rückgabewert: Interrupts pro Sekunde

void     taktEnable(void)         // Hinweis: Timer (Takt) wird aktiviert

void     taktSetEnable(int fb)    // Hinweis: Timer (Takt) wird gesetzt und aktiviert

void     taktDisable(void)        // Hinweis: Timer (Takt) wird deaktiviert
```

Auf die Signalisierung des Taktsemaphors wartet man mit:

```
void  taktWait (SEMID sem, int  ω)
                    // SEMID sem   = zu signalisierendes Semaphor
                    // int  ω =  bei jedem ω -ten Interrupt wird das Semaphor signalisiert
                              // Hinweis: Semaphorsignalisierung
                              // Die Taktfreuquenz ist fb / ω ;  ω > = 0 ist nicht erlaubt.
```

Ein typisches Einsatzbeispiel hat folgende Struktur:

```
for(;;)
  {                   // Echtzeitschleife
    taktWait(sem, w)      // Auf Signalisierung warten
    /* Do work */
  }
```

Solange die Schleife durchlaufen wird, ist der Ausgang von DAU7 gleich 5V. Während der Wartezeit in taktWait() ist er 0V.

Hinweis: Es muss vorher die Funktion initVdot() aufgerufen werden!

Die folgenden Funktionen dienen der Umrechnung:

double **taktFreq(int f_b, int w)** // Ergebnis in [Hz]

double **taktZeit(int f_b, int w)** // Ergebnis in [s]

Die beiden folgenden Funktionen dienen der Ausgabe über shprintf():

void **taktFreqOut(int f_b, int w)**

void **taktZeitOut(int f_b, int w)**

Hinweis: Zur Ermittlung der Einstellwerte f_b und ω existiert in der MATLAB-PDV-TOOLBOX die Funktion:

[fb, ω] = ratiot (T); Umrechnung einer Tastzeit

Ergebnis: Einstellwerte für den Zusatztimer und die Funktionen taktWait() und taktSet()

10.3. Systemuhr und Zeitgeber

Dieser letzte Abschnitt beschreibt Funktionen und Sprachmittel, die spezifisch für Echtzeitbetriebssysteme sind. Andere Systeme nutzen Zeitgeber (system clock) i.d.R. nur, um Dateien mit dem Erstellungszeitpunkt zu versehen, oder zyklisch Stundenplanungsfunktionen für den Anwender bereitzustellen. In Echtzeitsystemen hingegen existieren Sprechmittel zur Einbindung der Zeitgeber in Anwendungsprogramme. Ferner sind Mechanismen vorhanden, um den Programmablauf zeitlich zu steuern. Wie bereits dargestellt, hängt die Richtigkeit des Ergebnisses bei der Echtzeitdatenverarbeitung neben der logischen Richtigkeit ganz wesentlich von der „Rechtzeitigkeit“ des Ereignisses ab.

Grundsätzlich kann man folgende unterscheiden:

- Systemuhr oder Systemzeitgeber (System Clock)
- Hilfsuhr (Auxiliary Clock)
- Alarmzeitgeber (Watchdog Timer)

Je nach Hardware findet man tatsächlich drei verschiedene Zeitgeber (Timer) oder man bedient zwei oder mehrere Zeitgeber mit einem Timerbaustein. Ein Timerbaustein besteht aus einem Quarz, der mit einer definierten Frequenz schwingt. Nachgeordnet findet man einen programmierbaren Frequenzteiler, der es erlaubt, quasi jede beliebige Frequenz einstellen. Über entsprechende Systemroutinen, die meist in einer Timer-Bibliothek zusammengefasst sind, kann der Timerbaustein programmiert und abgefragt werden.

Systemuhr und Hilfsuhr (System Clock, Auxiliary Clock)

Typischerweise besitz eine Rechnerkarte einen oder zwei Timerbausteine, die die CPU periodisch in Form eines „Timer-Interrupts“ unterbrechen. Der Anwender kann sine Applikation über eine reguläre intConnect()-Routine an den Timer Interrupt anbinden.

Dabei kann er beispielsweise folgendes erledigen:

- Periodisches Abfragen von Eingängen oder Hardwareressourcen
- Anstoßen periodischer Prüfvorgänge im System
- Stoppen von Endlosprozessen oder „aufgehängten“ Tasks

Typische Bibliotheks-Zugriffsroutinen auf die Zeitgeber sind z. B. in VxWorks:

Tick(Get):

Ermittelt den aktuellen Zählerstand der Systemuhr in sogenannten Ticks. Ticks sind Bruchteile von Sekunden, deren Dauer einstellbar ist.

Tick(Set):

Setzt den Zähler auf null. Im Programmtext kann dann direkt z. B. in einer if-Schleife der Zählerstand mit einem festen Wert verglichen werden. Damit lassen sich auf einfache Art und Weise sehr präzise Abläufe programmieren.

Zur Verwaltung und Programmierung des Timerbausteins stehen unter VxWorks folgende Bibliotheksfunktionen zur Verfügung:

sysClkRateSet() // Setzt die Zeitgeberfrequenz ACHTUNG

sysClkRateGet() // Gibt die Zeitgeberrate zurück

Je nach Anpassung des Echtzeitsystems oder Ergänzungen durch den Anwender können weitere Routinen zur Manipulation des Timerbausteins zur Verfügung stehen.

Alarmzeitgeber (Watchdog Timer)

Diese Einrichtung ist eine spezialisierten Schnittstelle zur Systemuhr. Grundsätzlich kann damit die zeitliche Ablaufsteuerung von Tasks realisiert werden. Der Mechanismus beruht darauf, dass nach einer durch den Anwender vorgegebenen Zeit ein Timerinterrupt ausgelöst wird, der die Ausführung einer Funktion zur Folge hat. Ähnlich anderer Programmressourcen wie z. B. Message Queues muss ein Watchdog erzeugt, gestartet und nach Gebrauch wieder gelöscht werden.

Gebrauch von Watchdogs

WDOG_ID wdCreate()

// Liefert einen Watchdog Bezeichner (Identifier) zurück

wdStart(wdId, delay, pRoutine, parameter)

// wdID	Watchdog Identifier; ist der Rückgabewert des Erzeugungsbefehls wdCreate()
// pRoutine	Funktion (hier C-Funktion), die nach Ablauf der Verzögerung delay aufgerufen werden soll
// parameter	Funktionsparameter zur Übergabe beim Aufruf der Funktion pRoutine

Wird der Watchdog an einer Stelle im System wie oben beschrieben gestartet, läuft der Zähler ab und startet nach der eingestellten Zeit die im Aufruf deklarierte Funktion. Für den Fall, dass der Watchdog periodisch arbeitet, muss sich der Watchdog selbst aktivieren.

wdCancel (wdId)	// hält den Watchdog an
wdDelete (wdId)	// hält an und nimmt den Watchdog aus dem System

Folgendes Anwendungsbeispiel zeigt die Verwendung von Watchdogs zur Überwachung von Funktionsdauern.

```
WDOG_ID wdId;

void laufzeit_kontrolle (void)
     {
      wdId = wdCreate ();
      int check = 0;

         // Zu überwachenden Funktion, Zeitlimit 10 Sekunden die im
         // Konstante wdSart Aufruf entspricht der Anzahl Ticks
       FOREVER
          {
           check++;
           wdStart (wdId, DELAY_10_SEC, myWdFunktion, check);
          }
      }

 void myWdFunktion ( int durchlauf_nummer)
      {
       // Bearbeitung des zeitüberlaufes der applikations_funktion
       logMessage ( „Achtung Zeitüberlauf nach %d. Aufruf“);

        // Weitere Maßnahmen
        ...
     }
```

Obiges Beispiel zeigt auf sehr einfach Art und Weise, wie die tatsächliche Ausführungszeit, die einer Funktion benötigt, überwacht werden kann. Bleibt die Applikati-

onsfunktion innerhalb des Zeitrahmens von 10 Sekunden, so wird der Watchdog erneut gestartet und der Zähler entsprechend zurückgesetzt. Bei einer Verzögerung springt die Funktion myWdFunktion() an. Diese Alarmfunktion zeigt den Überlauf mit einer Meldung an einer Konsole an und stoppt eventuell die Applikation. Dieser simple Mechanismus erlaubt die Überwachung von Taskausführungszeiten und die zeitliche Synchronisation von Tasks. Letzteres erreicht man durch einen Watchdog und eine binäre Semaphore. Dabei wird eine beliebige Task auf eine Semaphore blockiert. Eine andere Task kann zu einem beliebigen Zeitpunkt den Watchdog aktivieren. Damit ist sichergestellt, dass durch ein semGive() in der Alarmroutine die wartende Funktion nach einer definierten Zeit bei entsprechender Priorität an die Reihe kommt. Eine andere Möglichkeit zur Taskverzögerung, die ebenfalls die Systemuhr benutzt, ist der bereits erwähnte taskDelay(AnzalTicks) Aufruf.

Zusammengefasst kann man sagen, dass Echtzeitsysteme im Gegensatz zu klassischen Betriebssystemen ein reichhaltiges Instrumentarium zur Nutzung von Timerbausteinen zur Verfügung stellen. Insgesamt ergeben sich daraus zahlreiche Möglichkeiten, das Applikationsprogramm in den rechten zeitlichen Einklang mit der Außenwelt, d.h. mit dem technischen Prozess zu bringen. Ein Haupthilfsmittel sind Interruptws, d.h. asynchrone externe Programmunterbrechungen. Zusätzlich bieten programmierbare Timerbausteine mit entsprechenden Schnittstellenbibliotheken umfangreiche Unterstützung. Die Ausstattung bzw. der Komfort variiert von Echtzeitsystem zu Echtzeitsystem und hängt zusätzlich von den Möglichkeiten der verwendeten Hardware ab.