Fallbeispiele

Fallbeispiele

- Einbettung von Echtzeitkernen in Standardbetriebssysteme

- Fallbeispiele zu prioritätsgesteuerten Echtzeitkernen
  - PXROS
  - OSEK

- Fallbeispiel für zeitgesteuerte Echtzeitkerne
  - Mars

Betrachtete Funktionen:
  - Taskverwaltung
  - Inter-Task-Kommunikationsmechanismen
  - Zeitverwaltung

VRTX
u8os

PXROS,
OSEK,

RTKernel,
EUROS, RTOS

Maruti, MARS,
Spring, PS/9

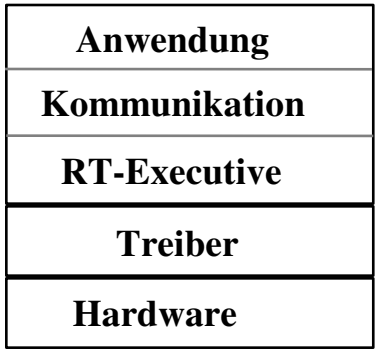
LynxOS, Mach, QNX,
Solaris, RTLinux, NT…

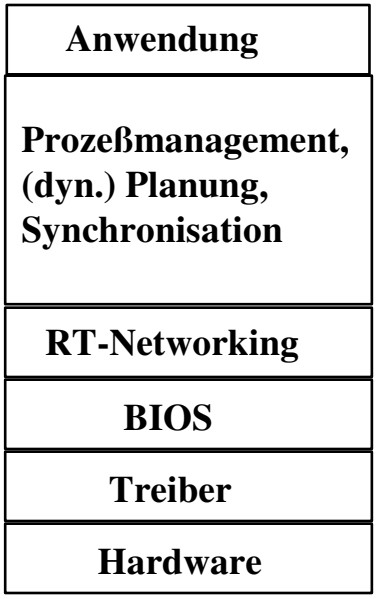
zunehmende
Systemkomplexität

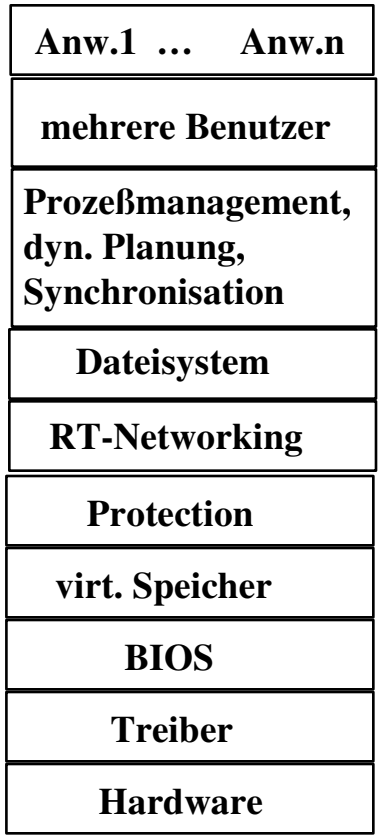
SimpleEmbedded Systems

+ Minimale System-Unterstützung
+ Minimaler Platzbedarf
+ Minimaler Overhead

- Aufwendige Off-line Analyse
- Statisches Systemverhalten
- kein Dateisystem
- kein BIOS

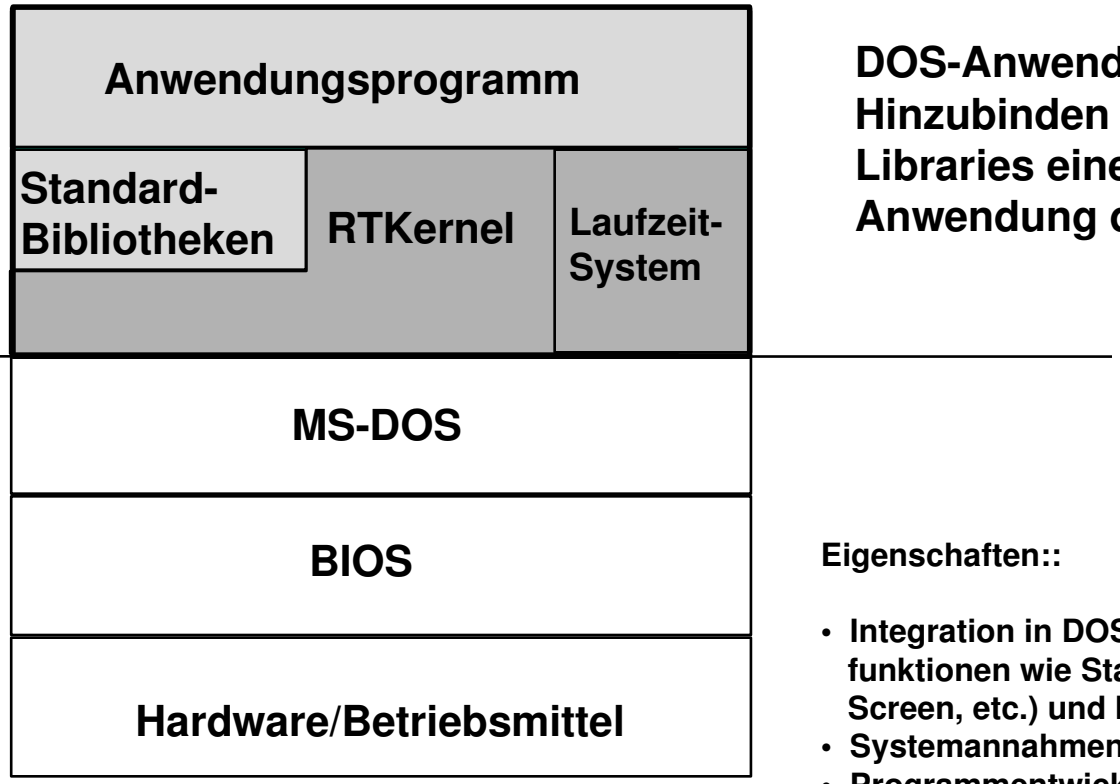






Einbettung von Echzeitkernen in die Systemarchitektur

Beispiel: RTKernel

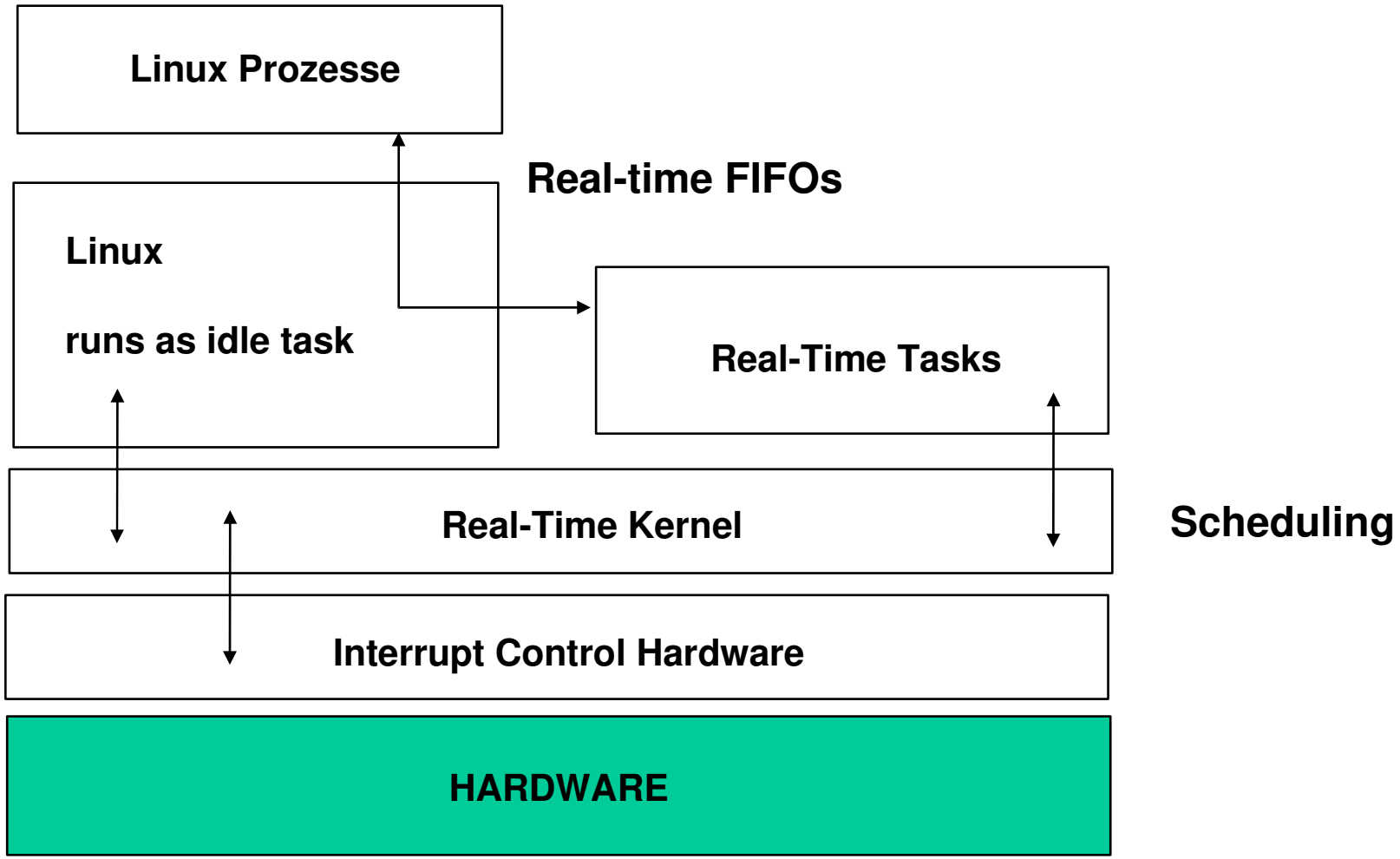


DOS-Anwendung, die durch Hinzubinden der RTKernel Libraries einen echzeitfähige Anwendung darstellt

Eigenschaften::

- **Integration in DOS /BIOS , dadurch sind Basisbetriebssystem-funktionen wie Standard I/O Funktionen (Platte, Floppy, Screen, etc.) und Interruptverarbeitung verfügbar.**
- **Systemannahmen 80x86 Architektur + 8059 Interruptbaustein**
- **Programmentwicklung, Debugging, Simulation auf einem System**

Architektur und Einbettung von RTLinux



Fallbeispiel:
Echtzeitkern PXROS

HighTec, Saarbrücken

PXROS Komponenten :

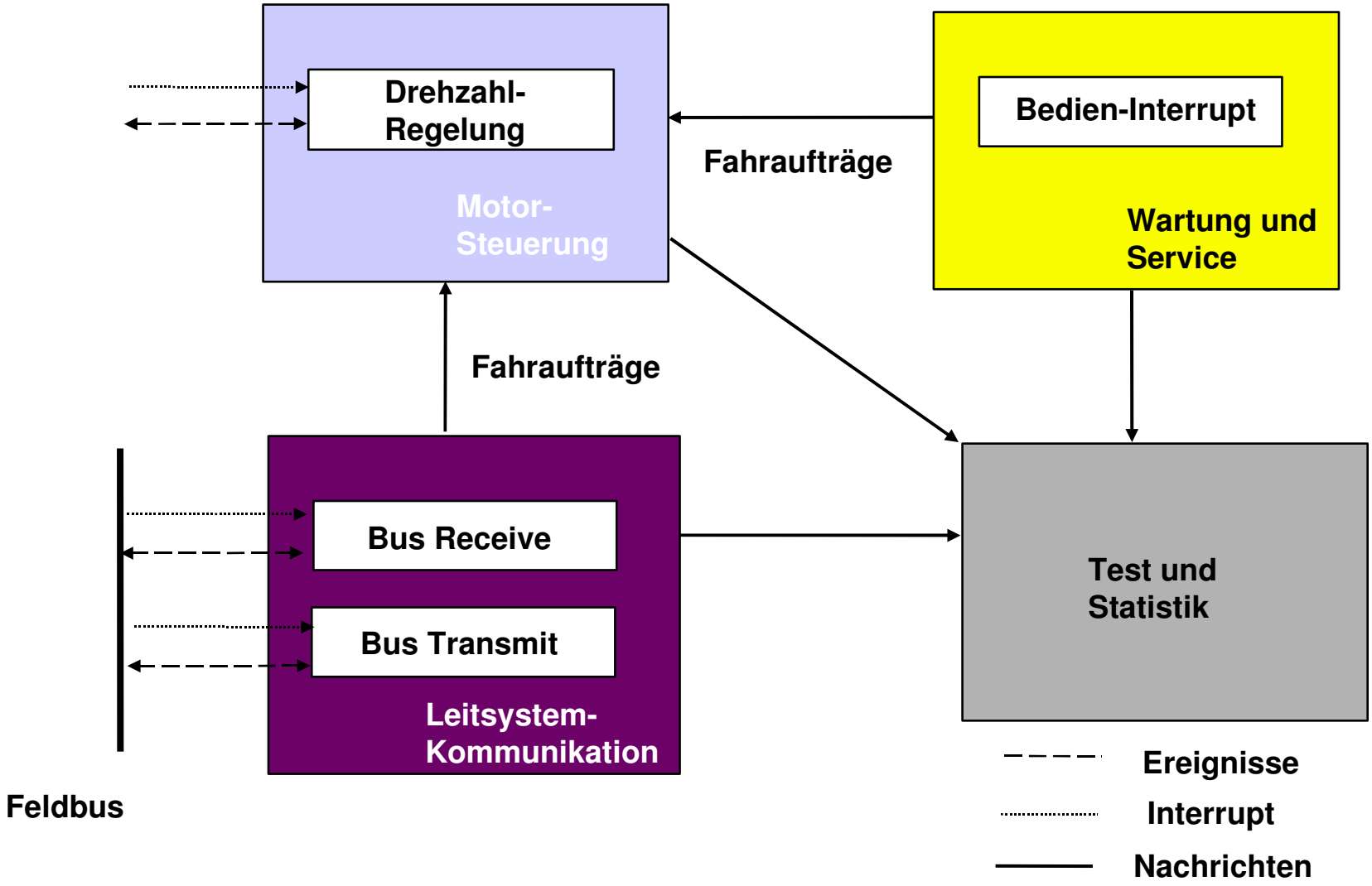
- **Tasks:**
  sind Aktivitätsträger, die eigene Ausführungskontexte besitzen und die Einheiten für das prioritätsbasierte, unterbrechbare Scheduling darstellen. 32 Prioritätsebenen.

- **Handler**
  sind spezielle Systemtasks, die gegenüber Anwendungstasks immer höhere Priorität besitzen und zur Bearbeitung besonders zeitkritischer Aufgaben wie z.B. Unterbrechungs-bearbeitung und periodisch zu aktivierende Regler-Routinen eingesetzt werden.

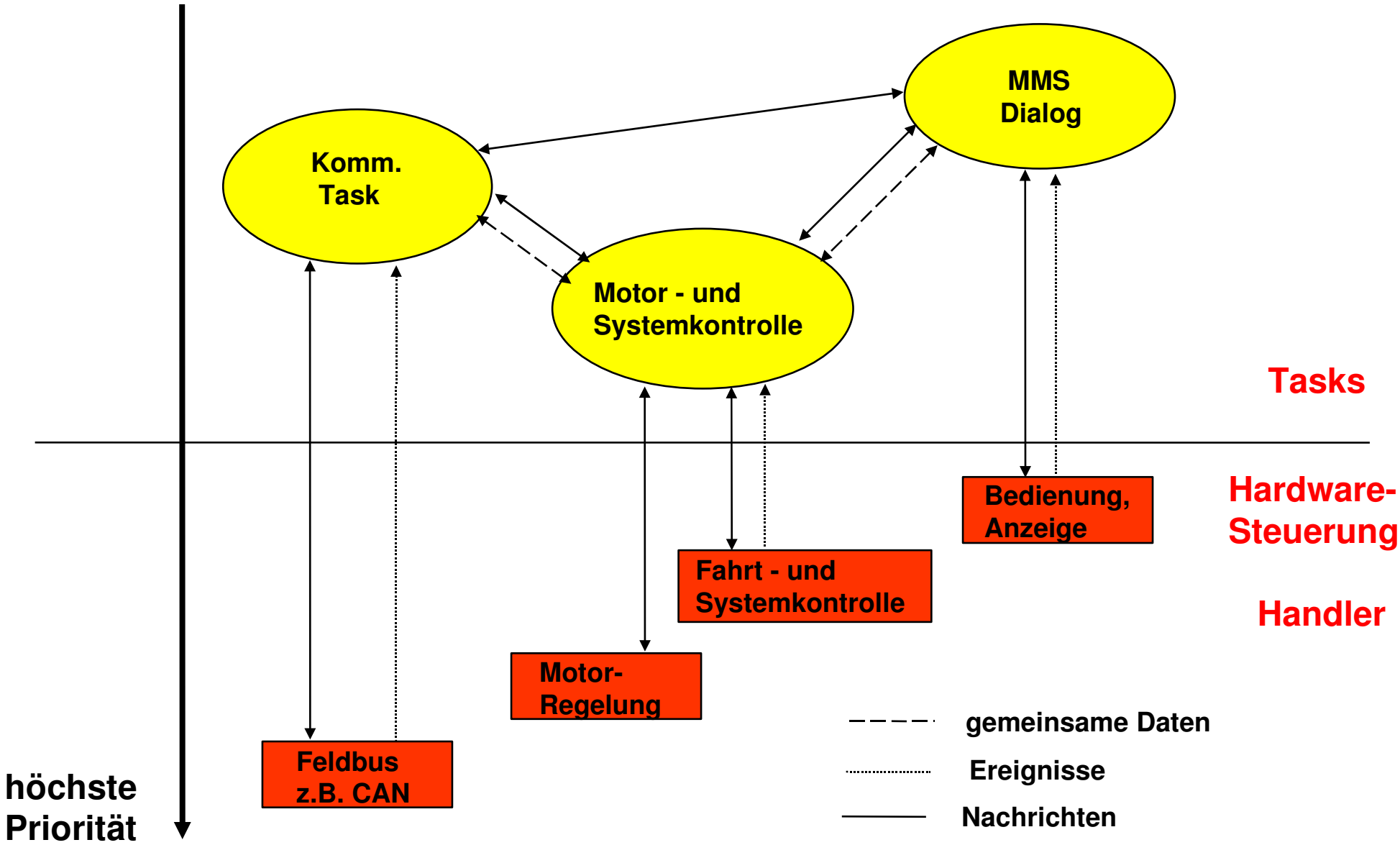
- **Objekte:**
  PXROS stellt folgende Objektklassen systemseitig zur Verfügung:
  - Nachrichten
  - Mailboxen
  - Speicherobjekte
  - Objekt-Pools
  - Delay-Objekte

- **Services**
  für die angegebenen Objektklassen werden jeweils bestimmte Services zur Verfügung gestellt

Architektur der Anwendung unter PXROS



Tasks und Handler



Beispiel Echtzeitkern PXROS(cont.): - Taskverwaltung -

Tasks

Eine Anwendung besteht i.a. aus mehreren verschiedenen Teilaufgaben. PXROS bietet die Moeglichkeit, diese Aufgaben in eigenen Programmen mit jeweils eigenem Ausfuehrungskontext innerhalb des Gesamtsystems zu realisieren. Diese Systemteile sind die PXROS-Tasks. Die zur Task gehoerende Funktion wird als Task-Code bezeichnet. PXROS teilt die Zugriffszeit auf den Prozessor zur Ausfuehrung des Task-Codes den einzelnen Tasks gemaess ihrerjeweiligen Task-Prioritaet zu. Die Tasks koennen sich gegenseitig unterbrechen, daher stellt PXROS den Kontext einer Task wieder her, bevor sie wieder aktiv wird. Der Task-Code wird auf einer eigenen Registerbank mit eigenem Stack ausgefuehrt und ander Stelle fortgesetzt, an der die Task zuvor unterbrochen wurde.

Services zur Manipulation von Tasks (PXROS^{COMPACT}):

PxGetId	Lesen des Task-Id der aufrufenden Task
PxTaskGetMbx	Lesen der Mailbox-Adresse einer Task
PxTaskGetPrio	Lesen der Task-Priorität
PxEnterHndlvl	Gehe auf die Handler-Ebene
PxExitHndlvl	Verlasse die Handler-Ebene

Services zur Manipulation des Modus von Tasks (PXROS^{COMPACT}):

Modus-Bits werden benutzt, um bestimmte Eigenschaften von Tasks zu maskieren, z.B. das Zeitscheibenverfahren oder den Abbruch-Mechanismus für diese Task außer Kraft zu setzen.

PxSetModebits	Setze die Modus-Bits der aufrufenden Task
PxClearModebits	Setze die Modus-Bits der aufrufenden Task zurück

Allgemeine Struktur einer Task in PXROS

```
void TaskFunc        ( PxTask_t myid,
                       PxMbx_t mymbx,
                       PxEvents_t myevents )
{
      INITIALIZE_TASK;
            for (;;)
            {
            WAIT_FOR_MESSAGES_AND_EVENTS;
            PROCESS_EVENTS;
            PROCESS_MESSAGE;
            CLEANUP;
            }
}
```

Beispiel Echtzeitkern PXROS (cont.):
- Inter-Task-Kommunikation -

Ereignisse:

Das Signalisieren von Ereignissen wird zur Informationsübermittlung durch einen Handler oder eine Task verwendet. Auf diese Art wird einer Task das Eintreten einer für sie wichtigen Begebenheit angezeigt, z.B. das Ende eines Timeouts oder das Auftreten eines Fehlers. Ereignisse können nicht an Handler signalisiert werden.

Fuer Handler ist dies die übliche Art, mit Tasks zu kommunizieren, da sie i.a. keine Nachrichten verschicken können. Aber auch Tasks signalisieren u.U. anderen Tasks Ereignisse, anstatt Nachrichten zu schicken. Dies ist z.B. dann der Fall, wenn nicht umfangreiche Daten zu übergeben sind, sondern nur eine Zustandsänderung angezeigt werden soll. Ausserdem benötigt das Signalisieren eines Ereignisses keine Ressourcen.

Die Bedeutung von Ereignissen ist nicht festgelegt. Jeder Task können 32 verschiedene Ereignisse signalisiert werden, die von 0 bis 31 durchnumeriert sind. Das Ereignis mit der Nummer i wird dadurch dargestellt, dass in einer 32-Bit-Maske Bit i gesetzt ist.

Services für die Ereignisbehandlung (PXROS^{COMPACT}):

Ereignisbehandlung:

PxAwaitEvents	**Erwarte spezifiziertes Ereignis.**
PxResetEvents	**Setze die Ereignismaske für spezifiziertes Ereignis zurück.**
PxTaskSignalEvents	**Signalisiere das spezifizierte Ereignis an eine Task (Task Service).**
PxTaskSignalEvents_Hnd	**Signalisiere das spezifizierte Ereignis an eine Task (HandlerService).**
PxGetSavedEvents	**Lies die Ereignisse, die für die aufrufende Task gespeichert wurden.**
PxGetAbortingEvents	**Lies die Abbruch-Ereignisse für die aufrufende Task.**
PxExpectAbort	**Aufruf einer Funktion mit Erwartung eines Abbruchs**

Beispiel Echtzeitkern PXROS(cont.):
- Inter-Task-Kommunikation -

Nachrichten

PXROS-Anwendungen benutzen Nachrichten zur Kommunikation zwischen den Tasks. Eine Nachricht ist ein Nachrichten-Objekt mit einem Datenbereich, der von der versendenden Task belegt und von der empfangenden Task ausgewertet wird. Nachrichten koennen bei Bedarf dynamisch erzeugt oder wiederverwendet werden. Es gibt die Moeglichkeit, vorab Nachrichten zu erzeugen und in einer eigenen Mailbox aufzubewahren. Bei Bedarf kann dann eine Task eine der vorbereiteten Nachrichten aus der Mailbox entnehmen. Dies ist u.U. erheblich schneller als die komplette Erzeugung einer Nachricht. So kann die Laufzeit der Anwendung verringert werden.

Services für Nachrichten (PXROS^{COMPACT}):

PxMsgSend	Sende normale Nachricht
PxMsgSend_Prio	Sende priorisierte Nachricht
PxMsgReceive	Task nimmt Nachricht aus einer Mailbox. Falls die Mailbox leer ist, wartet die Task.
PxMsgReceive_NoWait	Task nimmt Nachricht aus einer Mailbox. Falls die Mailbox leer ist, kehrt Task mit Fehlercode zurück.
PxMsgReceive_EvWait	Task nimmt Nachricht aus einer Mailbox. Falls die Mailbox leer ist, wartet Task bis Ereignis Ev eintritt oder die Nachricht ankommt.
PxMsgRelease	Ein Nachrichteobjekt wird freigegeben, d.h. die Nachricht wird gelöscht und die belegten Ressourcen freigegeben.
PxMsgGetData	Lesen des Datenteils der Nachricht

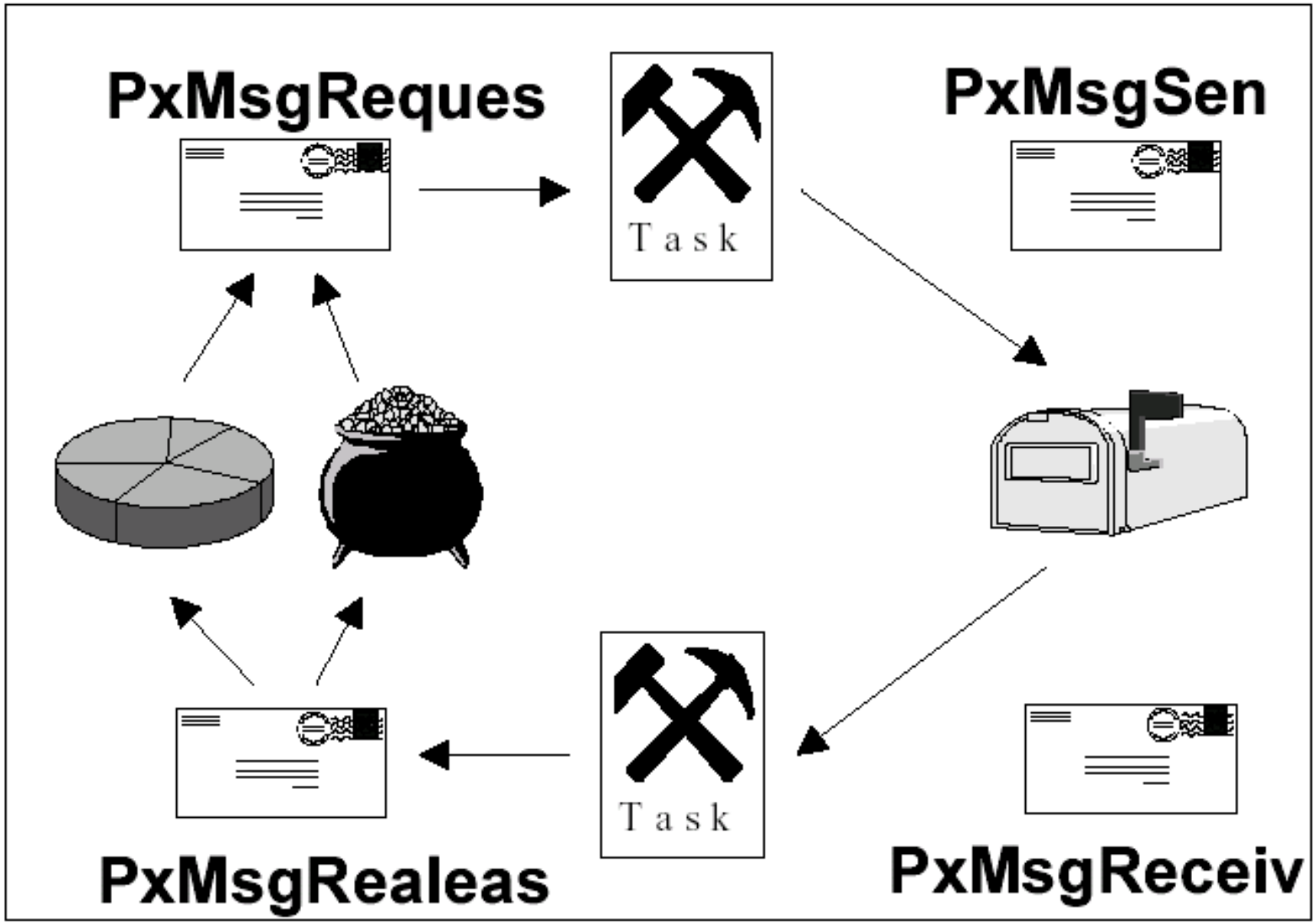
Beispiel Echtzeitkern PXROS(cont.):
- Inter-Task-Kommunikation -

Mailboxen

Eine Mailbox bewahrt Nachrichten in der Reihenfolge ihres Eintreffens auf, bis diese von Tasks entnommen und ausgewertet werden. Eine Task erhaelt immer die Nachricht, die am laengsten in der Mailbox liegt. (Ausnahme: priorisierte Nachrichten werden vor anderen Nachrichten ausgegeben). Ist die Mailbox leer, kann die Task ggfs. zusammen mit anderen Tasks auf eine Nachricht warten. Eine eintreffende Nachricht wird immer der Task uebergeben, die am laengsten gewartet hat. Es ist einer Task nicht moeglich, eine bestimmte Nachricht aus einer Mailbox zu nehmen.

Services für Mailboxen (PXROS^{COMPACT}):

Keine Mailbox-Services in PXROS^{COMPACT}. Hier werden Mailboxen Tasks zugeordnet und werden nicht explizit erzeugt oder zerstört. Eine Task hat eine Mailbox zur Kommunikation mit anderen Tasks.
Im Standard PXROS sind Services zur Erzeugung und Behandlung zusätzlicher Mailboxen vorhanden.


PxMsgReques
Task
PxMsgSen
Task
PxMsgRealeas
PxMsgReceiv

Beispiel Echtzeitkern PXROS(cont.):
- Ressourcenverwaltung -

Speicherklassen

PXROS verwaltet dynamisch allokierbaren Speicher fuer die Applikation. Es ist moeglich, den Speicher in einzelne Speicher klassen aufzuteilen, die bestimmten Teilen der Anwendung zugeordnet werden. Dadurch koennen temporaere Speicher-Engpaesse lokal begrenzt werden. In jedem PXROS-System existiert die Systemspeicherklasse, aus der ggfs. der Speicher fuer andere Speicherklassen entnommen wird. Hauptsaechlich stellt sie Speicher fuer PXROS-Systemfunktionen zur Verfuegung, z.B. bei der Erzeugung von Tasks.

Speicherklassen werden danach unterschieden, ob daraus Bloecke unterschiedlicher oder fester Groesse entnommen werden koennen. Die Nutzung einer Speicherklasse fester Blockgroesse erhoeht die Bearbeitungsgeschwindigkeit der Speicherverwaltung. Dagegen kann eine Speicherklasse variabler Blockgroesse flexibler genutzt werden.

Wird von der Applikation dynamisch Speicher angefordert, so muss beim Funktionsaufruf immer die Speicherklasse angegeben werden, aus der der Speicher genommen werden soll. Die Systemspeicherklasse wird durch das Makro PXMcSystemdefault bezeichnet.

Services für Speicherverwaltung (PXROS^{COMPACT}):

PxMcTakeBlk	**Nimm einen Speicherblock aus einer Speicherklasse.**
PxMcReturnBlk	**Freigeben eines Speicherblocks, der wieder in die Speicherklasse als freie Ressource eingegliedert wird.**
PxMcGetSize	**Größe der Speicherklasse (freie Ressourcen) wird angegeben.**

Beispiel Echtzeitkern PXROS(cont.):
- Ressourcenverwaltung -

Objekt-Pools

Alle unbenutzten PXROS-Objekte werden in Objekt-Pools aufbewahrt. Die Objekte koennen auf verschiedene Objekt-Pools verteilt werden, die bestimmten Teilen der Anwendung zugeordnet sind. Dadurch koennen temporaere Ressourcen-Engpaesse lokal begrenzt werden. In jedem PXROS-System existiert der System Pool, der zunaechst alle erzeugten Objekte enthaelt und aus dem ggfs. die Objekte fuer andere Objekt-Pools entnommen werden.

Es existieren reale und virtuelle Objekt-Pools. Reale Objekt Pools dienen zur echten Aufteilung der Objekte in verschiedene Pools. Dagegen werden virtuelle Objekt-Pools lediglich zur Nutzungsbeschraenkung und -kontrolle verwendet.

Werden von der Applikation Objekte angefordert, so muss immer der Objekt-Pool angegeben werden, aus der das Objekt genommen werden soll. Der System-Pool wird durch das Makro PXOpoolSystem default bezeichnet.

Services für Speicherverwaltung (PXROS^{COMPACT}):

Keine. Alle Objekte liegen im System-Pool.
Im Standard PXROS gibt es Services zur Erzeugung, Zerstörung, Anforderung und Freigabe von Objekt-Pools.

Beispiel Echtzeitkern PXROS (cont.):
- Zeitverwaltung -

Delay-Objekte

**Delay-Objekte dienen zur Realisierung zeitbasierter Routinen.
Mit einem Delay-Objekt wird der Auftrag fuer PXROS verbunden,
eine bestimmte Funktion nach Ablauf einer angegebenen Zeitspanne
aufzurufen. Diese Funktion wird auf Handler-Ebene ausgefuehrt,
d.h. sie unterbricht die laufende Task.**

Services für die Zeitbehandlung (PXROS^{COMPACT}):

PxTickDefine_Hnd	**Definiere einen Tick der PXROS-Uhr**
PxDelaySched	**Plane eine Verzögerung für eine Task ein**
PxDelaySched_Hnd	**Plane eine Verzögerung für einen Handler ein**
PxPeStart	**Starte periodisches Ereignis**
PxPeStop	**Stoppe periodisches Ereignis**
PxToStart	**Starte Timeout**
PxToStop	**Stoppe Timeout**

Fallbeispiel:
OSEK/VDX

OSEK
Offene Systeme und deren Schnittstellen für die Elektronik im Kraft-fahrzeug

VDX
Vehicle Distributed eXecutive

What is OSEK/VDX?

OSEK/VDX is a joint project of the automotive industry. It aims at an industry standard for an open-ended architecture for distributed control units in vehicles.

A real-time operating system, software interfaces and functions for communication and network management tasks are thus jointly specified.

The term OSEK means:

”Offene Systeme und deren Schnittstellen für die Elektronik im Kraft-fahrzeug”

(Open systems and the corresponding interfaces for automotive electronics).

The term VDX means

„Vehicle Distributed eXecutive“.

The functionality of OSEK operating system was harmonised with VDX. For simplicity OSEK will be used instead of OSEK/VDX in the document.

OSEK/VDX partners

The following companies attended and contributed to the OSEK/VDX Technical Committee:

Adam Opel AG,
AFT GmbH,
ATM Computer GmbH,
BMW AG,
Blaupunkt,
Centro Ricerche Fiat,
Denso,
Cummins Engine Company,
Daimler-Benz AG,
Dassault Electronique,
Delco Electronics,
ETAS GmbH & Co KG,
Hella KG Hueck & Co.,
Hewlett Packard France,
IIIT University of Karlsruhe,
ITT Automotive,
Integrated Systems,
LucasVarity,
Magneti Marelli,
Mecel,
Motorola,
National Semiconductor,
NEC Electronics,
NRTT,
Philips Car Systems,
PSA,
Renault,
Robert Bosch GmbH,
Sagem Electronic Division,
SGS Thomson,
Siemens Automotive,
Siemens Semiconductors,
Softing GmbH,
TEMIC,
Texas Instruments,
UTA - United Technologies Automotive,
Valeo Electronique,
Vector Informatik,
Volkswagen AG,
Volvo Car Corporation,
Wind River Systems.

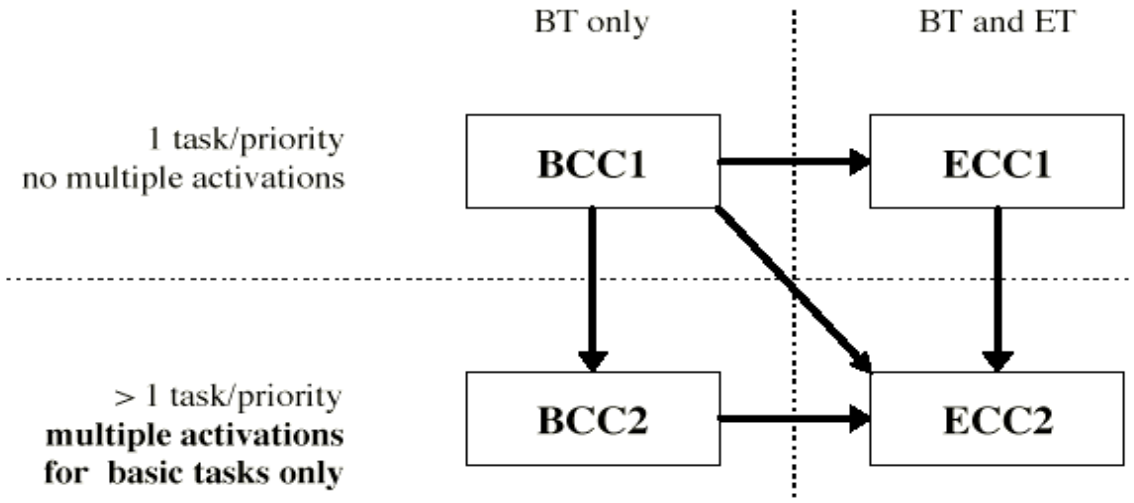
Special support for automotive requirements

Specific requirements for an OSEK operating system arise in the application context of software development for automotive control units. Requirements such as reliability, real-time capability, and cost sensitivity are addressed by the following features:

- **The OSEK operating system is configured and scaled statically. The number of tasks, resources, and services required is statically specified by the user.**
- **The specification of the OSEK operating system supports implementations capable of running on ROM, i.e. the code could be executed from Read-Only-Memory.**
- **The OSEK operating system supports portability of application tasks.**
- **The specification of the OSEK operating system provides a predictable and documented behaviour to enable operating system implementations, which meet automotive real-time requirements.**
- **For each operating system implementation performance parameters must be known.**

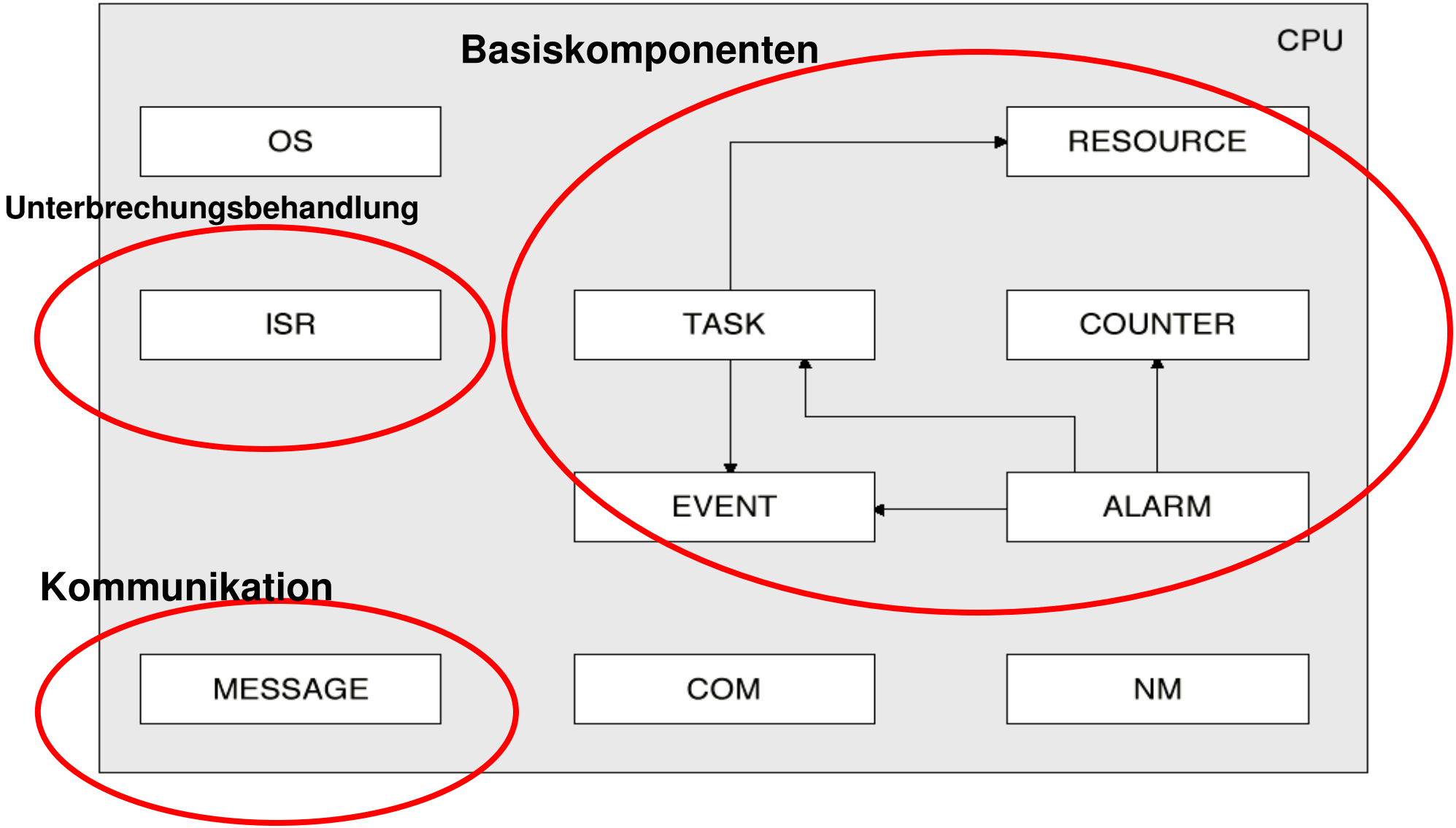
OSEK OS - Conformance Classes (CC)

- Generell zwei Gruppen von CC
  - Basic (BCC) erlaubt nur basic Tasks
    (Basic Tasks haben keinen *waiting*-Zustand)
  - Extended (ECC) erlaubt auch extended Tasks
    (Extended Tasks können auf Events warten)
- Aufwärts-kompatible CC's:
  BCC1, BCC2, ECC1, ECC2



Minimal parameters of implementations

	BCC1	BCC2	ECC1	ECC2
Multiple requesting of task activation	no	yes	BT[9]: no ET: no	BT: yes ET: no
Number of tasks which are not in the *suspended* state	≥ 8		≥ 16 (any combination of BT/ET)	
Number of tasks per priority	1	> 1	1 (both BT/ET)	> 1 (both BT/ET)
Number of events per task	—		≥ 8	
Number of priority classes	≥ 8			
Resources	only scheduler	≥ 8 (including scheduler)		
Alarm	≥ 1 (single or cyclic alarm)			
Application Mode	≥ 1			



OSEK Komponenten

Beschreibung der Komponenten

CPU: the CPU on which the application runs under the OSEK OS control.

OS: the OSEK OS which runs on the CPU. No standard references are defined in OIL from OS to other system objects but, of course, all system objects are controlled by OS.

ISR: interrupt service routines supported by OS.

RESOURCE: the resource which can be occupied by a task.

TASK: the task handled by the OS.

COUNTER: the counter represents hardware/software tick source for alarms.

EVENT: the event owned by a extended task. Up to 8 events can be assigned to an extended task.

ALARM: the alarm is based on a counter and can activate a task or set an event.

MESSAGE: the message which provides local or network communication.

COM: the communication subsystem. This subsystem is out of the scope of the OSEK OS specification.

NM: the network management subsystem. This subsystem is out of the scope of the OSEK OS specification.

Task management

•Activation and termination of tasks
• Management of task states, task switch

Synchronisation

The operating system supports two means of synchronisation effective on tasks:

- Resource management
  Access control for inseparable operations to jointly used (logic) resources or devices, or for control of a program flow.
- Event control
  Event management for task synchronisation.

Inter-Task Kommunikation

Interrupt management

- Services for interrupt processing

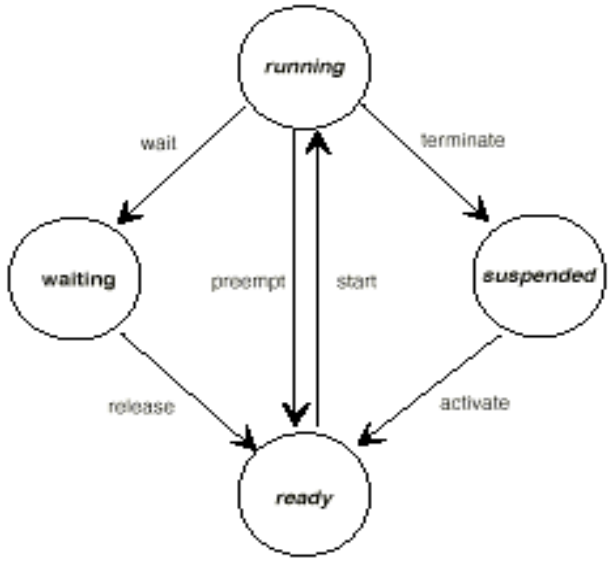
Alarms

• Relative and absolute alarms
• Static (defined at compile-time) and dynamic (defined at run-time) alarms

Error treatment

• Mechanisms supporting the user in case of various errors

Extended Tasks



- **running**

In the running state, the CPU is assigned to the task, so that its instructions can be executed. Only one task can be in this state at any point in time, while all the other states can be adopted simultaneously by several tasks.

- **ready**

All functional prerequisites for a transition into the running state exist, and the task only waits for allocation of the processor. The scheduler decides which ready task is executed next.

- **waiting**

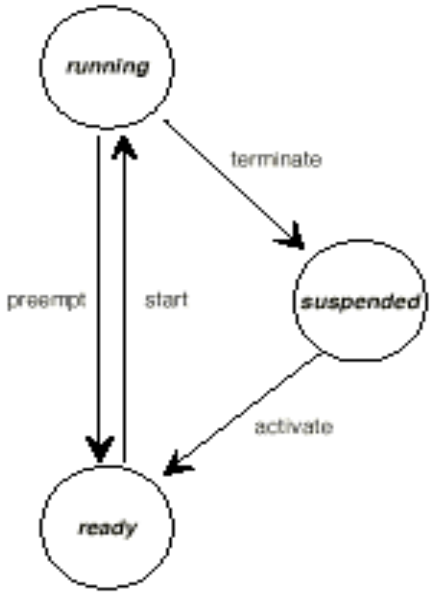
A task cannot continue execution because it has to wait for at least one event (see chapter 6, Event mechanism).

- **suspended**

In the suspended state the task is passive and can be activated.

Transition	Former state	New state	Description
activate	suspended	ready	A new task is entered into the *ready* queue by a system service. The OSEK operating system ensures that the execution of the task will start with the first instruction.
start	ready	running	A *ready* task selected by the scheduler is executed.
wait	running	waiting	To be able to continue operation, the *running* task requires an event. It causes its transition into the *waiting* state by using a system service.
release	waiting	ready	At least one event has occurred which a task has *waited* on.
preempt	running	ready	The scheduler decides to start another task. The *running* task is put into the *ready* state.
terminate	running	suspended	The *running* task causes its transition into the *suspended* state by a system service.

Basic Tasks



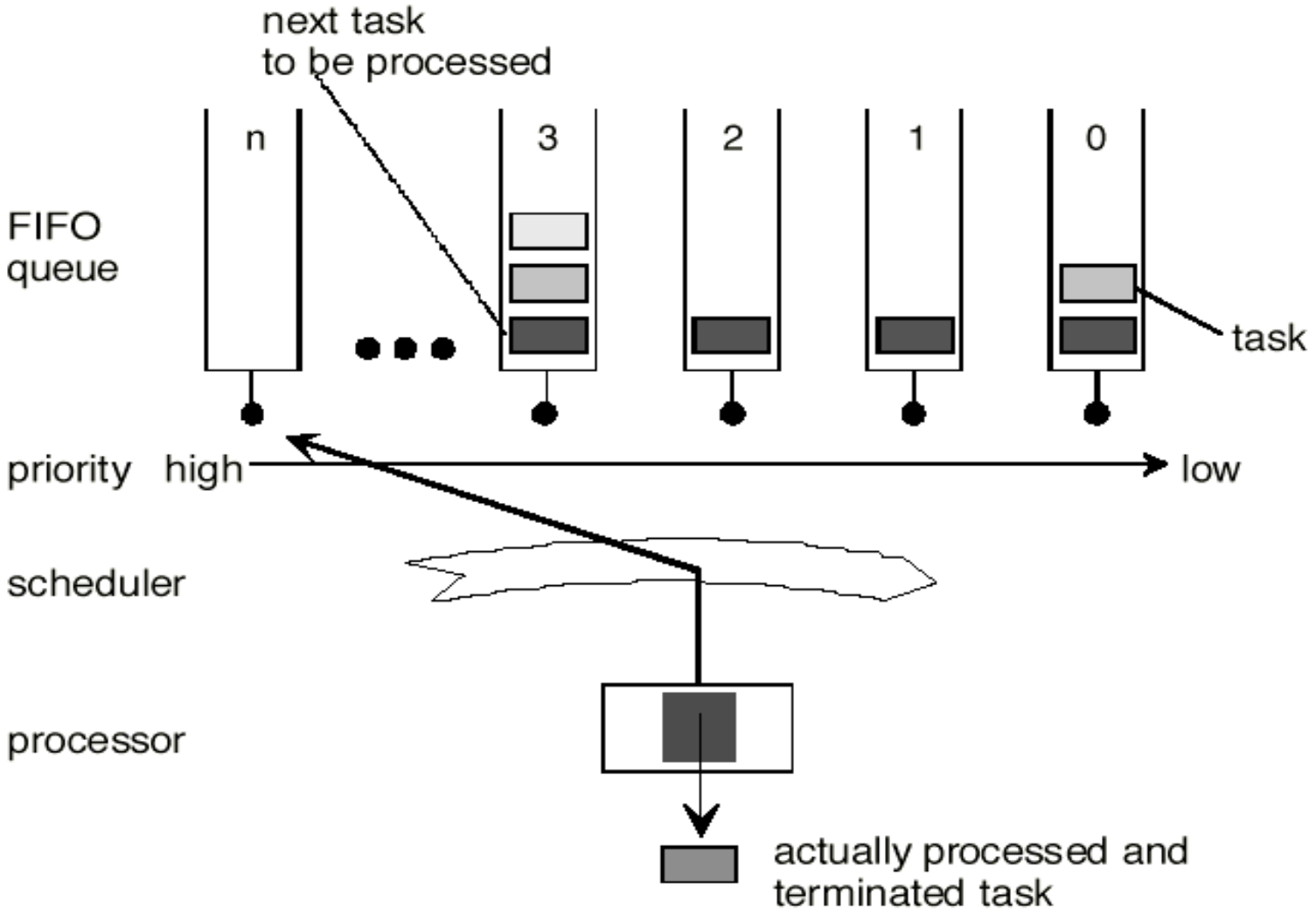
- **running**
  In the running state, the CPU is assigned to the task, so that its instructions can be executed. Only one task can be in this state at any point in time, while all the other states can be adopted simultaneously by several tasks.

- **ready**
  All functional prerequisites for a transition into the running state exist, and the task only waits for allocation of the processor. The scheduler decides which ready task is executed next.

- **suspended**
  In the suspended state the task is passive and can be activated.

Transition	Former state	New state	Description
activate	suspended	ready[2]	A new task is entered into the *ready* queue by the a system service. The OSEK operating system ensures that the execution of the task will start with the first instruction.
start	ready	running	A *ready* task selected by the scheduler is executed.
preempt	running	ready	The scheduler decides to start another task. The *running* task is put into the *ready* state.
terminate	running	suspended	The *running* task causes its transition into the *suspended* state by a system service.

Scheduler


next task
to be processed
n
3
2
1
0
FIFO
queue
task
priority
high
low
scheduler
processor
actually processed and
terminated task

System services related to tasks:

- DeclareTask
- ActivateTask
- TerminateTask
- ChainTask
- Schedule
- GetTaskID
- GetTaskState

The Event mechanism

- **is a means of synchronisation**

  in conjunction with the wait-state of an extended task

- **is only provided for extended tasks**

  Each extended task has a definite number of events

- **initiates state transitions of tasks to and from the waiting state.**

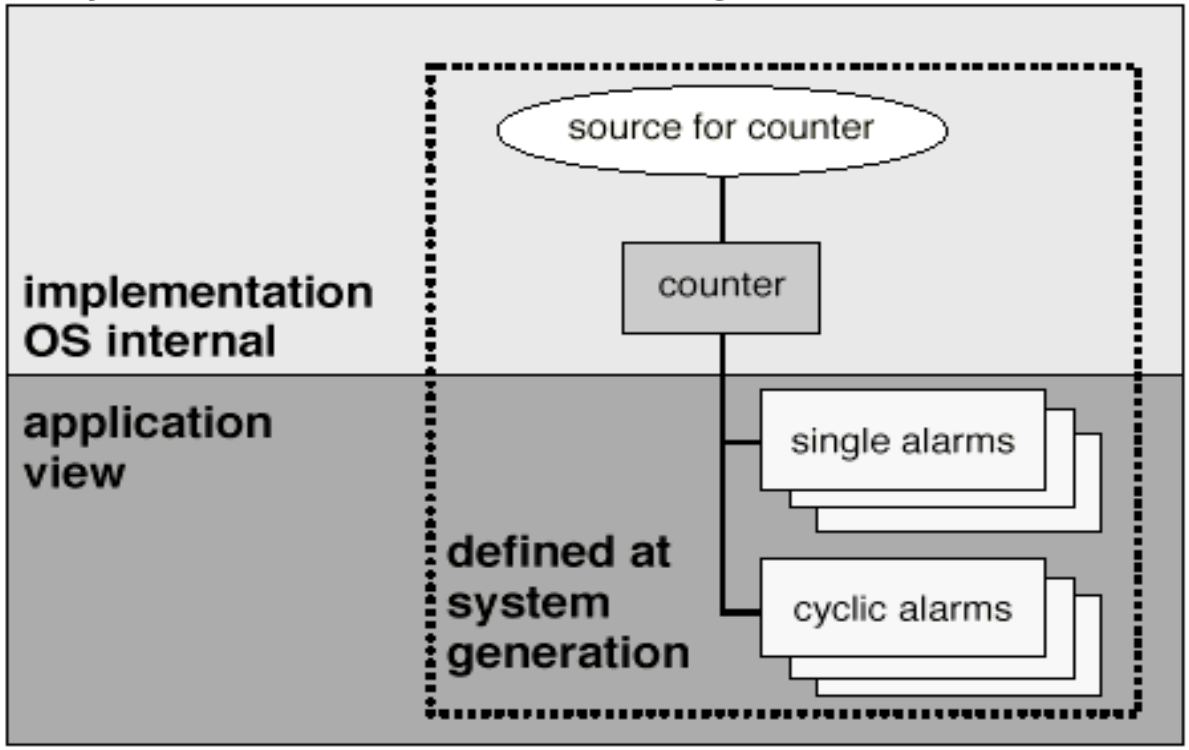
System services related to Event Processing:

- **DeclareEvent**
- **SetEvent**
- **ClearEvent** (reset event)
- **GetEvent** returns the current state of all event bits the task is waiting for
- **WaitEvent**

Alarms

- **Processing of reccuring events**
  - **single alarm and cyclic alarm**
  - **relative (to a timer value) and absolute points in time**
  - **time (clock ticks) and event counting**

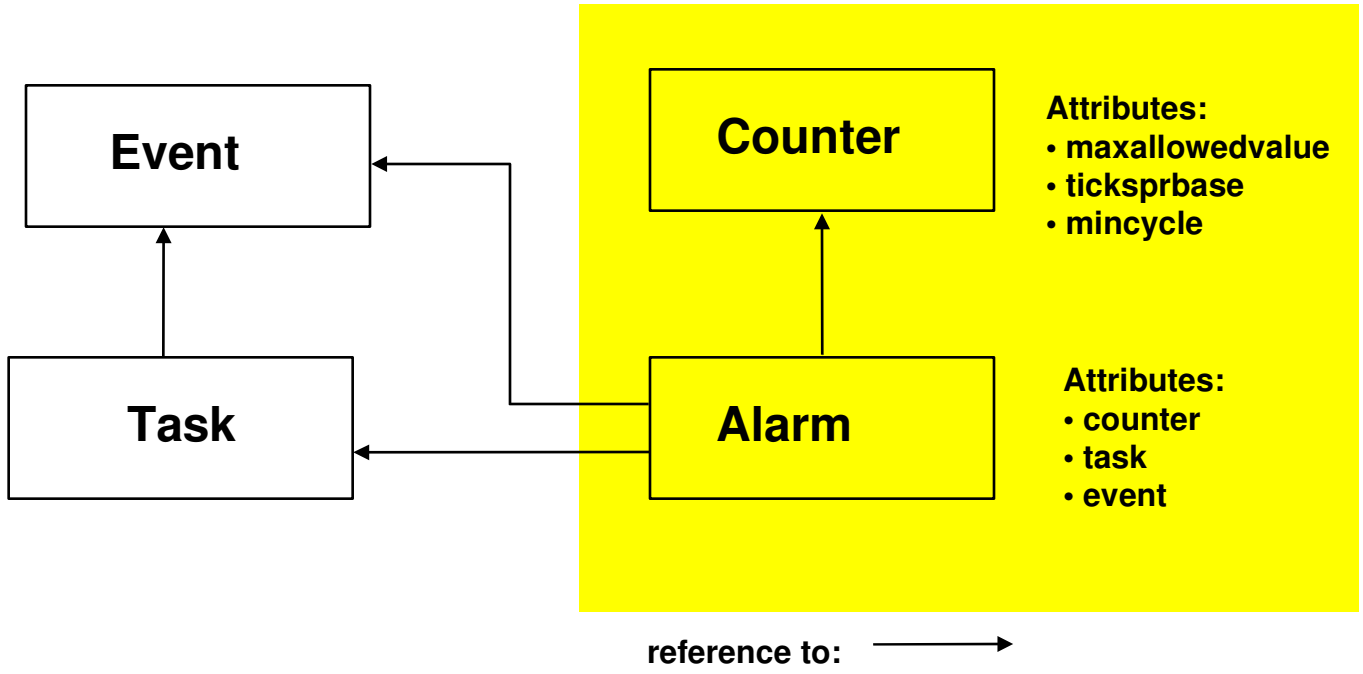
Layered model of alarm management



An Alarm may start a task or set an event.

An alarm is statically assigned at system generation time to:

- **one counter**
- **one task**
- **a notation if the task is to be activated or the event is to be set**



System services related to Alarms:

- DeclareAlarm
- GetAlarmBase
- GetAlarm
- SetRelAlarm
- SetAbsAlarm
- CancelAlarm

Resource Management

The resource management is used to coordinate concurrent accesses of several full preemptive tasks with different priorities to shared resources, e.g. management entities (scheduler), program sequences, memory or hardware areas.

Resource management ensures that:
- **two tasks cannot occupy the same resource at the same time.**
- **priority inversion can not occur.**
- **deadlocks do not occur by use of these resources.**
- **access to resources never results in a waiting state.**

The functionality of resource management is only required in the following cases:
- **full-preemptive tasks**
- **non-preemptive scheduling, if resources are also to remain occupied beyond a scheduling point (in OSEK this only applies to the system service "*Schedule( )*")**
- **non-preemptive scheduling, if the user intends to have the application code executed under other scheduling policies, too**

The resource management is mandatory for all conformance classes.

Inter-Task Kommunikation

- Transparent local and network communication by messages.
- Messages are encapsulated in message objects which are handled by the OS.
- Message objects are defined at system configuration.
- A uinique identifier (UID) is assigned to message objects.
- Tasks reference messages by this UID.
- OSEK distinguishes between queued (event) and unqueued (state) messages.
- Task activation and event signalling can be statically defined to be performed at message arrival for notification.

Inter-Task Kommunikation

OSEK unterscheidet:

- **State Messages und**
- **Event Messages**

State Messages repräsentieren den Wert einer Echtzeitvariablen. Die Empfangsoperation liest die Nachricht, ohne sie zu zerstören. Eine State Message enthält stets den aktuellen Wert der Variablen, d.h. ein alter Wert wird von einem neuen Wert beim „send“ überschrieben.

Event Messages werden gepuffert, d.h. ein neuer Wert wird in einen neuen Puffer geschrieben. Event Messages werden beim „receive“ konsumiert, d.h. zerstörend gelesen.

Zusammenfassung: Features of the Message Concept

State Message	Event Message
No buffering	Buffering in a FIFO-queue
No consumption of message	Consumption of messages
Direct access possible in non-preemptive systems (no copies)	No direct access possible (always copies)
Static definition of task activation or event signalling	
Static definition of a message alarm	

Inter-Task Kommunikation

- **OSEK unterstützt 1:1 und 1:N Kommunikation für State- und Event-messages.**
- **Optional kann statisch festgelegt werden, dass eine Task aktiviert oder notifiziert wird, wenn eine neue Nachricht ankommt.**
- **OSEK stellt keine spezielle Systemunterstüzung für das Warten auf Nachrichten zur Verfügung. Hierzu kann der allgemeine „Event-“ Mechanismus verwendet werden.**

Error Handling, Tracing, Debugging

Hook Routines

The OSEK operating system provides system specific hook routines to allow user-defined actions within the OS internal processing.

Those hook routines are:

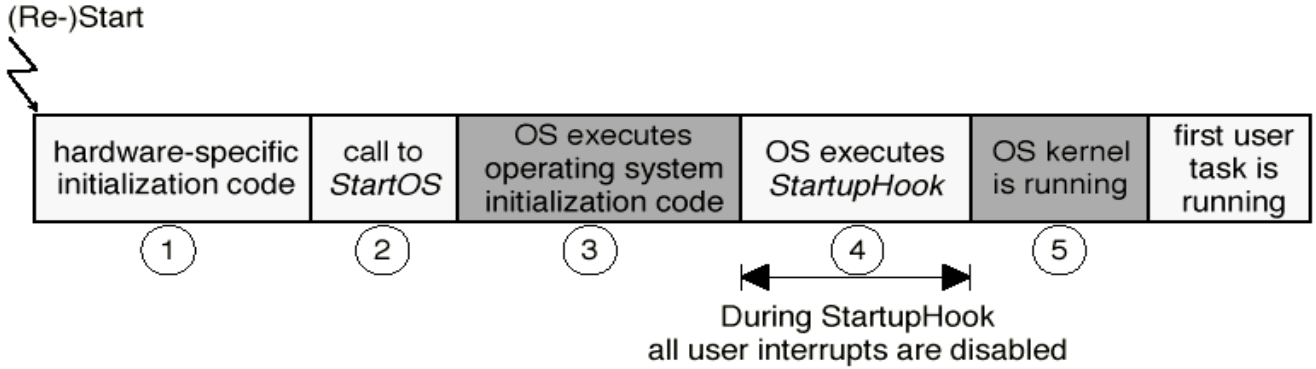
- **called by the operating system, in a special context depending on the implementation of the operating system**
- **higher priority than all tasks**
- **using an implementation dependent calling interface.**
- **part of the operating system but user defined**
- **implemented by the user**
- **standardised in interface per OSEK OS implementation, but not standardised in functionality (environment and behaviour of the hook routine itself), therefore usually hook routines are not portable**
- **are only allowed to use a subset of API functions**
- **optional (the implementation should omit calls to hook routines which do not exist)**

In the OSEK operating system hook routines may be used for:

- **system start-up. The corresponding hook routine (StartupHook) is called after the operating system start-up and before the scheduler is running.**
- **system shutdown. The corresponding hook routine (ShutdownHook) is called to a) request a shutdown by the application and b) force a system shutdown in case of a severe error.**
- **tracing or application dependent debugging purposes as well as user defined extensions of the context switch.**
- **error handling. The corresponding hook routine (ErrorHook) is called if a system call returns a value not equal to E_OK.**

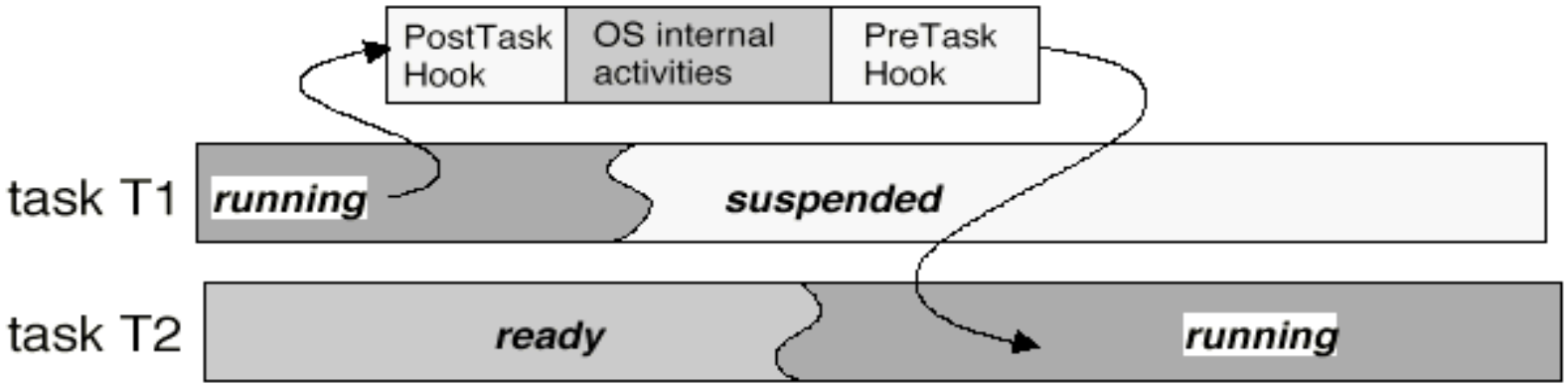
Each implementation of OSEK has to describe the interfaces and conventions for the hook routines, e.g. interfaces of system functions that may be called by the hook routines.

Example: StartupHook



1) **After a reset, the user is free to execute (non-portable) hardware specific code. The non-portable section ends by detection of the application mode.**
2) **Call StartOS with the application mode as a parameter. This call starts the operating system.**
3) **The operating system performs internal start-up functions and**
4) **calls the hook routine StartupHook, where the user may place initialisation procedures. During this hook routine, all user interrupts are disabled.**
5) **The operating system enables user interrupt according to the INITIAL_INTERRUPT_DESCRIPTOR, and starts the scheduler.**

Example: Debugging with “OSPreTask” and “OSPostTask”



Two hook routines (PreTaskHook and PostTaskHook) are called on task context switches.

These two hook routines may be used for debugging or time measurement (including context switch time). Therefore PostTaskHook is called after leaving the context of the old task, PreTaskHook is called before entering the context of a new task.

MARS
MAintanable Real-time System
+
TTA: Time Triggered Architecture

Ein zeitgesteuertes verteiltes Echtzeitbetriebssystem

H. Kopetz, W. Merker: „The Architecture of MARS“, Proc. 15th FTCS, June 1985

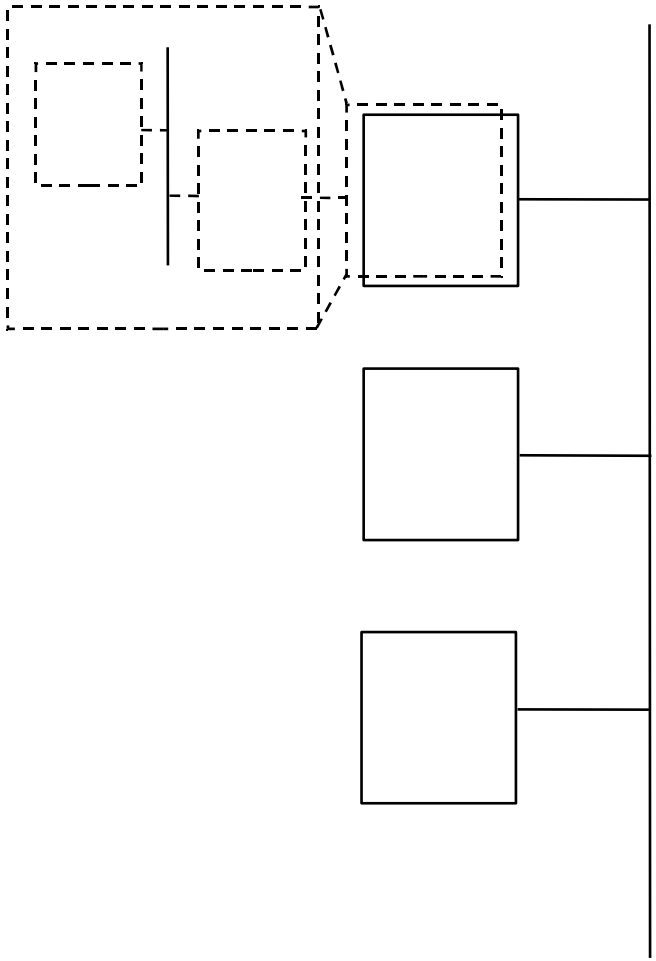
H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, R. Zailinger: „Distributed Fault-Tolerant Systems: The MARS approach“, IEEE MICRO, vol. 9, No. 1, February 1989

H. Kopetz: „Real-Time Systems: Design Principles for Distributed Embedded Applications“, Kluwer 1997

Entwurfsziele und Eigenschaften in MARS:

- **Berücksichtigung der zeitlichen Gültigkeit von Information**
- **Unterstützung periodischer Systeme**
- **Berücksichtung von Höchstlastsituationen**
- **(Echtzeit-) Transaktionsorientiert**
- **Unterstützung globaler Zeit durch Synchronisation lokaler Uhren**
- **Interprozesskommunikation über Zustandsvariablen**
- **Netzwerk- und Cluster-orientiert**
- **TDMA Netzwerk MAC-Protokoll**
- **Vorwärtsfehlerbehebung und aktive Redundanz**

Hardwarestruktur



Komponenten-orientierte Systemsicht

Eine Komponente definiert die Granularität der Fehlertoleranz

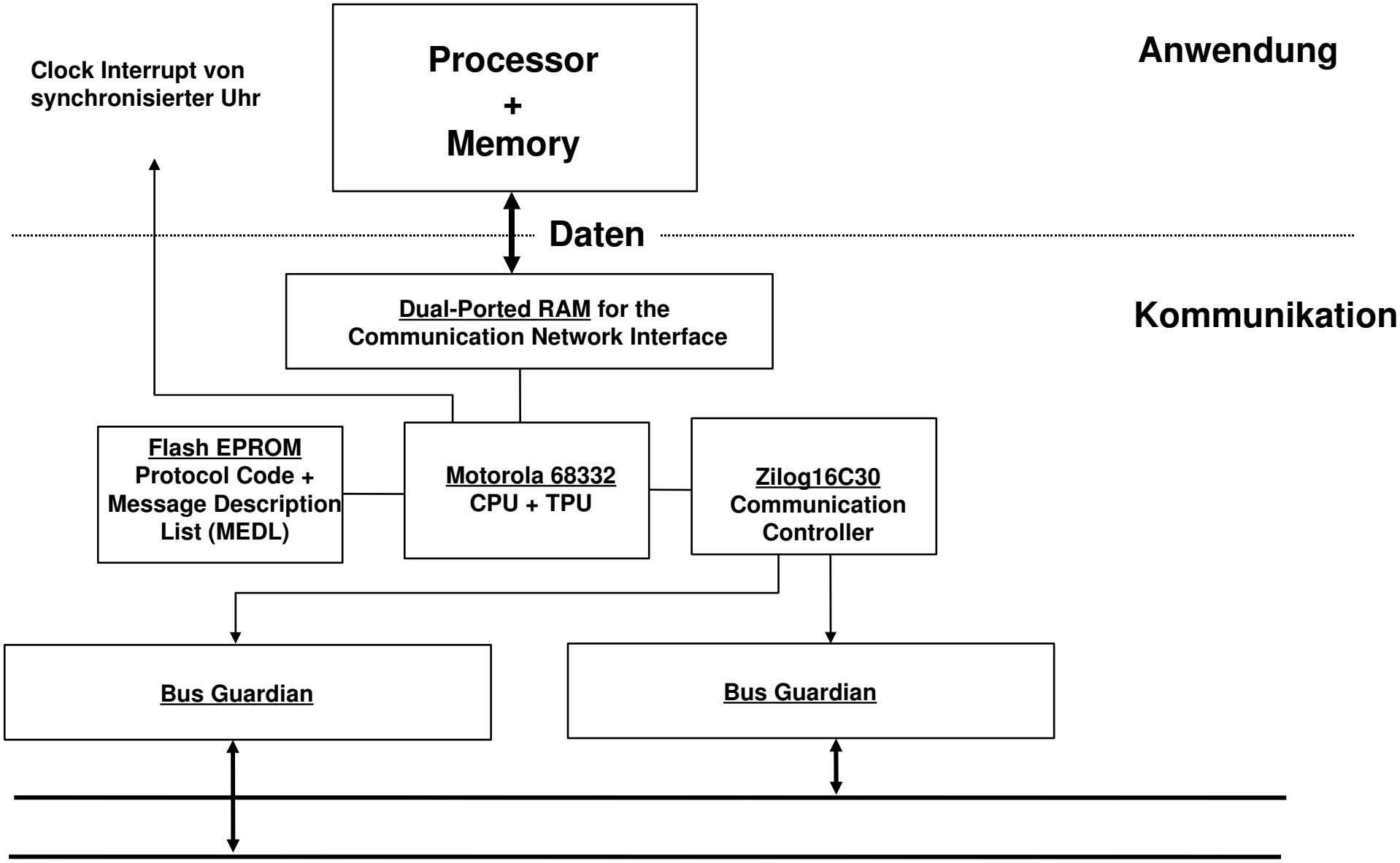
Die Kommunikation über den Cluster-Bus muss vorhersagbar sein

Fail-Silent Eigenschaften der Komponente muss gewährleistet sein

Strikte off-line Analyse des Zeitverhaltens

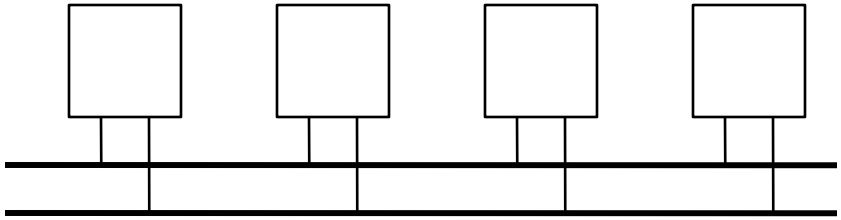
Vollständige Kenntnis des Zeitverhaltens ist der Schlüssel zur:

- **Erkennung von Fehlern**
- **Durchsetzung des Fail-Silent Verhaltens**
- **Erweiterbarkeit des Systems**

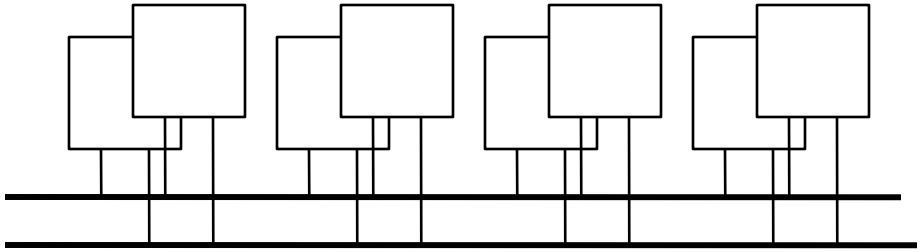

Clock Interrupt von synchronisierter Uhr
Processor + Memory
Anwendung
Daten
Dual-Ported RAM for the Communication Network Interface
Kommunikation
Flash EPROM Protocol Code + Message Description List (MEDL)
Motorola 68332 CPU + TPU
Zilog16C30 Communication Controller
Bus Guardian
Bus Guardian

Fehlertolerante Netzwerk-Konfigurationen

Class 1:
1 Knoten/FTU
2 Frames/FTU

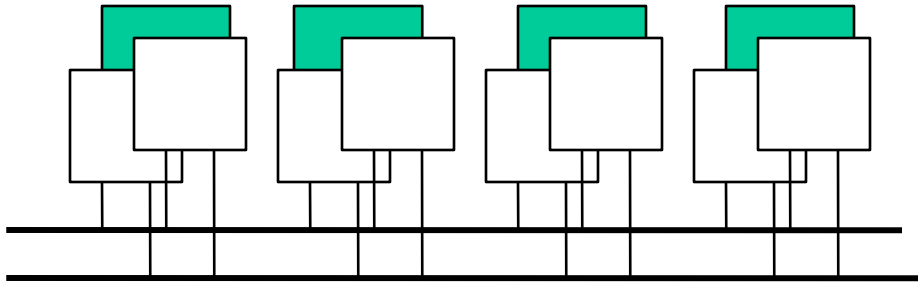


Class 2:
2 aktive Knoten/FTU
2 Frames/FTU



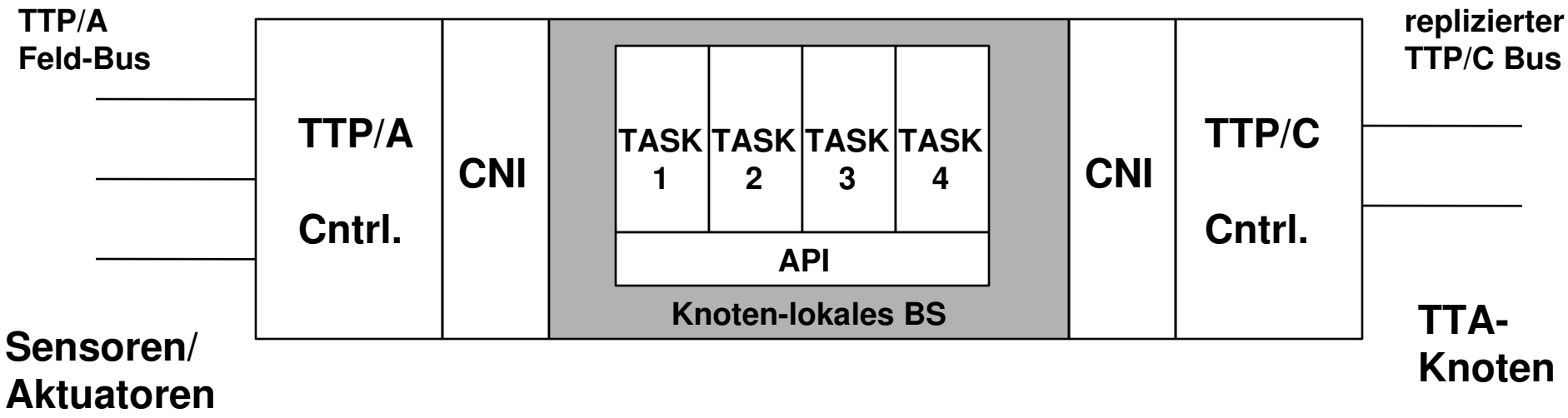
Class 3:
2 aktive Knoten/FTU
4 Frames/FTU

Class 4:
2 aktive Knoten/FTU
+ 1 Ersatz/FTU
4 Frames/FTU



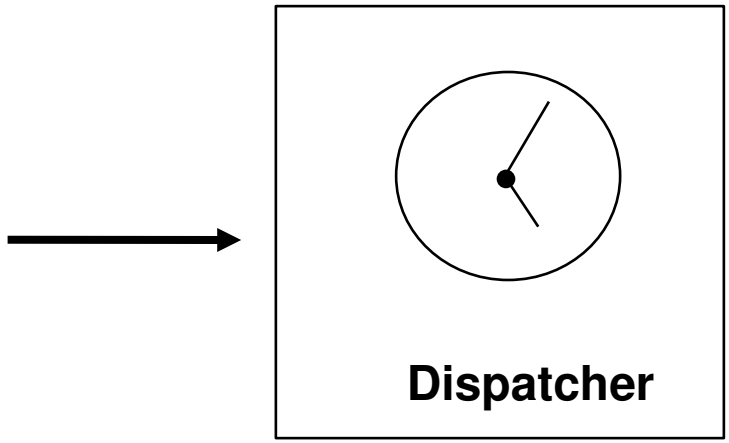
Komponentenredundanz + Zeitredundanz

Einbettung des Betriebssystems in einer TT-Architektur



Task Descriptor List in einer TT-Architektur

Zeit	Aktion	WCET
10	Start T1	12
17	Send M7	
22	Stop T1	
38	Start T3	20
47	Send M2	



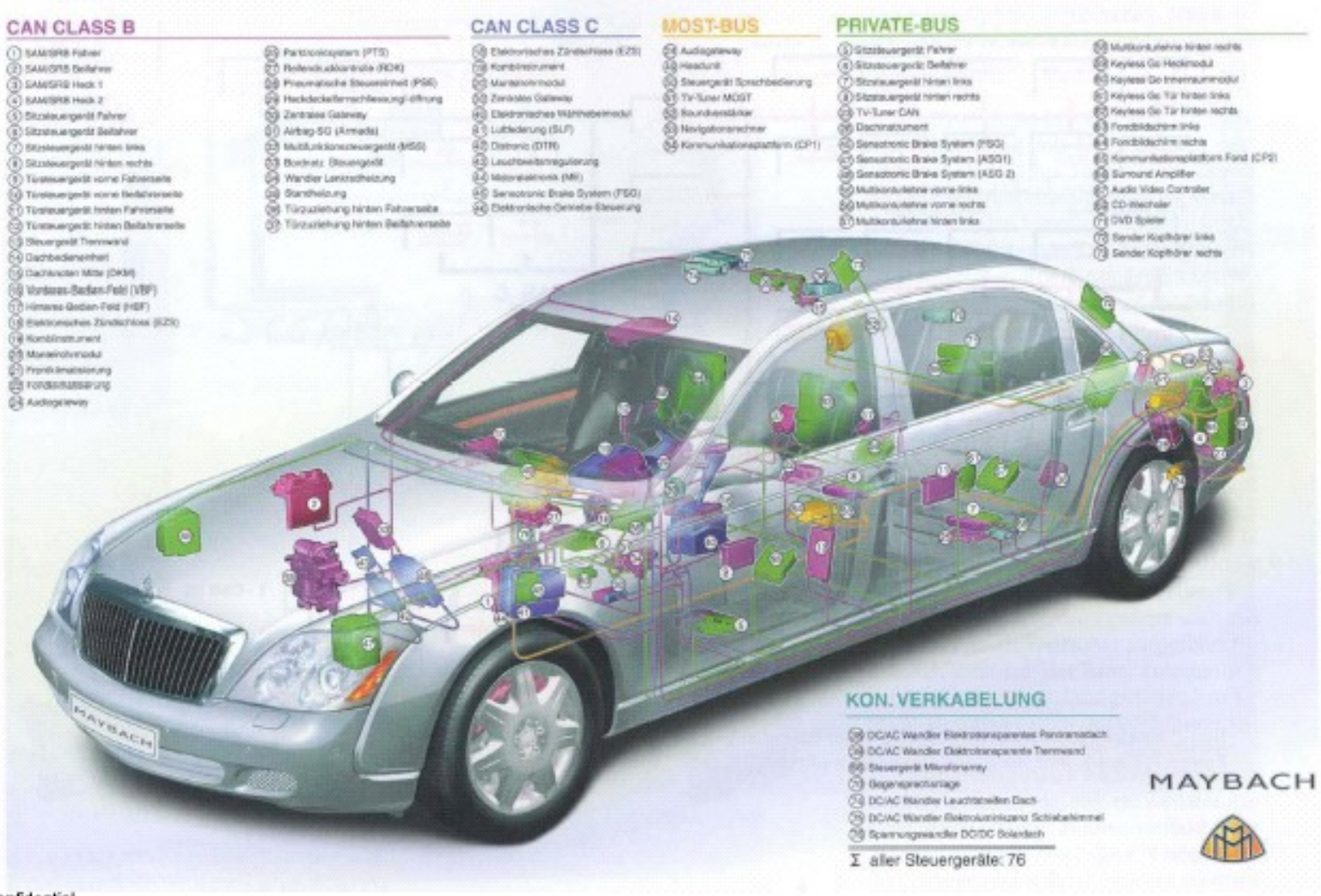
- Das zeitliche Verhalten wird off-line analysiert und ein TADL wird off-line erstellt.

- Ressourcenkonflikte werden durch off-line Planung verhindert.

- Replizierte Tasks werden zeitlich deterministisch ausgeführt.

- Knoten, die immer denselben extern sichtbaren Zustand enthalten heissen „Replica deterministisch“.

... very complex software-intensive systems ...



Ausblick: Verteilte-Echtzeitsysteme

- Protokolle für Echzeitsysteme
  - * Feldbusse (CAN, ProfiBus, WorldFip,Lon (Echelon))
  - * Time Triggered Protokoll (TTP/C)
  - * Token-Ring
  - * Echtzeitfähige CSMA-Netze (VTCSMA)
  - * Echtzeitfähigkeit drahtloser Netze

- Zeit und Ordnung in verteilten Echzeitsystemen
  - * Ordnung und Konsens in verteilten Systemen
  - * Protokolle zur Uhrensynchronisation

- Zuverlässigkeit und Fehlertoleranz
  - * Attribute und Maße der Zuverlässigkeit
  - * grundlegende Techniken der Fehlertoleranz
  - * Software- Fehlertoleranz (DRB, NVP)

- Zuverlässige, zeitbeschränkte Konsensprotokolle
  - * "Reliable Broadcast "