
Betriebssysteme für eingebettete Systeme und Echtzeitsysteme



Betriebssystem-Kerne

Grundlegende Funktionalität:

High-Level:

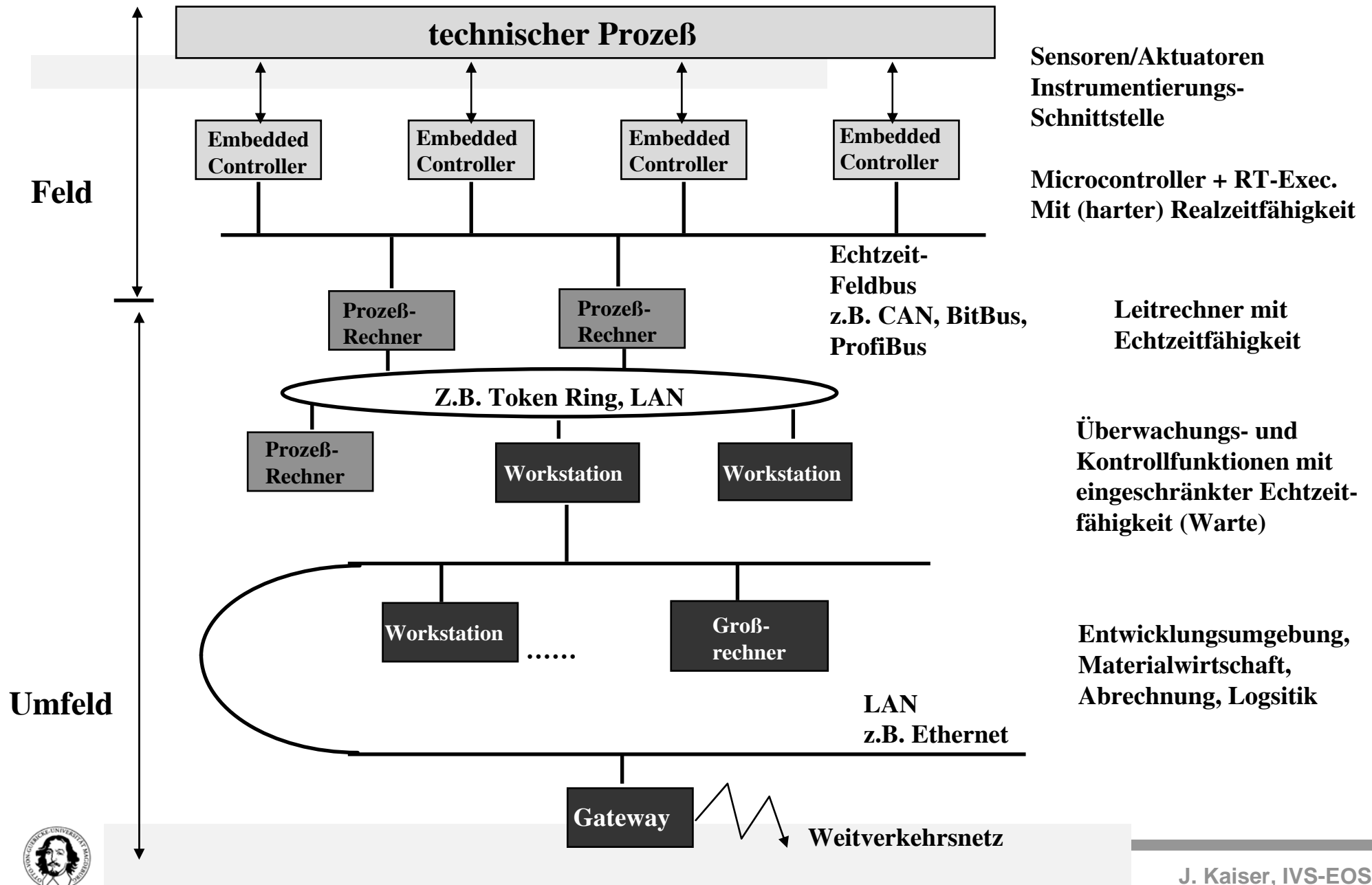
- Scheduling
- Zeitbehandlung
- Ein-Ausgabe, Ereignisbehandlung
- Kommunikation
- Ressourcenverwaltung

Low-Level:

- Task Management
- Task-Synchronisation
- Inter-Task-Kommunikation
- Zeitgeber und Uhren
- Unterbrechungsbehandlung
- Statische/Dynamische Speicherallokation



Einbettung von Echtzeitsystemen in eine CIM-Umgebung



Anforderungen und Technologie

Ebene:	Charakteristische Anforderungen:	Technologien/Betriebssysteme:
Embedded Controller, einfache Steueraufgaben	Harte Echtzeitbedingungen, kurze Reaktionszeiten	Microcontroller, Echtzeitkerne mit eingeschränkter Funktionalität PXROS, RTkernel, VRTX, VXworks EUROS, iRMX, OSEK. . .
Leitrechnerebene	Harte Echtzeitbedingungen, kurze Reaktionszeiten, Realisierung konsistenter globaler Zustände, globales, verteiltes Scheduling, Erkennung globaler (kritischer) Zustände	Prozeßrechner oder Workstations, Echtzeitbetriebssysteme für komplexe verteilte Steuerungsaufgaben, Beisp.: OpenVMS, Posix IEEE 1003.13, LynxOS, QNX, RT-Linux, Maruti, SPRING, MARS, RT-Mach,
Warte	Eingeschränkte Echtzeitfähigkeit, graphische Bedienoberfläche	Workstationstechnologie, Touchscreen, GUI, Visualisierungswerkzeuge (z.B. Matlab/Simulink, LabView)
Entwicklungsumgebung	Keine Echtzeitanforderungen, Leistungsfähige, Multi-User-Umgebung zur Programm-Entwicklung, Simulation, Verifikation, Test	High-End-Workstation, hoher Leistungsbedarf, Compiler: ADA, PEARL, C, C++, JAVA Off-line Scheduling Werkzeuge, Simulationsumgebung



Kategorien von eingebetteten Betriebssystemen

- **Echtzeiterweiterungen für “Timesharing” Betriebssysteme**
Beispiele: RT-Mach, RT-Erweiterungen in POSIX, RED-Linux, RT-Linux,
- **Echtzeitbetriebssysteme in der Forschung**
Beispiele: MARS, CHAOS, Spring, ARTS, MARUTI,
- **Prioritätsbasierte Betriebssystemkerne für eingebettete Anwendungen**
Beispiele: VRTX, VxWorks, QNX, OSEK, PXROS, RTkernel, OS9, RTEMS, u8os.
- **Betriebssystemkerne für stark ressourcenbeschränkte Anwendungen**
Beispiele: TinyOS, Contiki, Mantis,



VRTX
u8os

PXROS,
OSEK,

RTKernel,
EUROS, RTOS

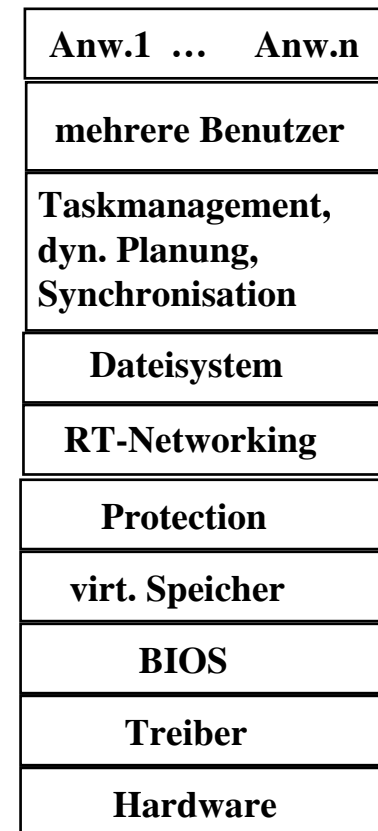
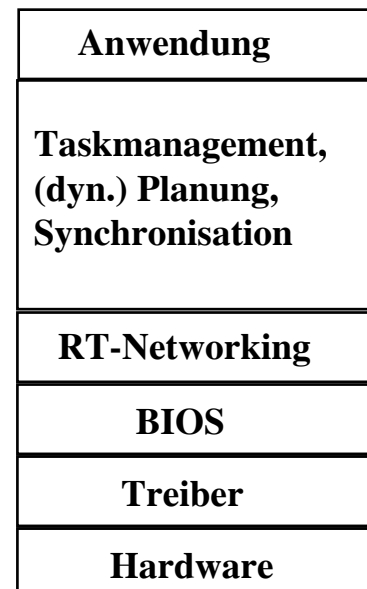
Maruti, MARS,
Spring, PS/9

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



Anpassen der Betriebssystemkerne an die Anwendungsanforderungen

Profile in POSIX (1003.13) :

PSE50 - Minimales Realzeit System Profil:

Geeignet für eingebettete Ein- oder Mehrprozessorsysteme, die ein oder mehrere externe Geräte kontrollieren. Es wird nur ein einzelner Prozeß (nur 1 Adressraum!) mit mehreren Threads unterstützt. Kein Dateisystem, keine Operator-Schnittstelle.

PSE51- Systemprofil für „Real-Time Controller“:

Erweiterung zu PSE50. Dateisystem und asynchrone Ein-/Ausgabe.

PSE52 – Spezielles Reazzeit-Profil:

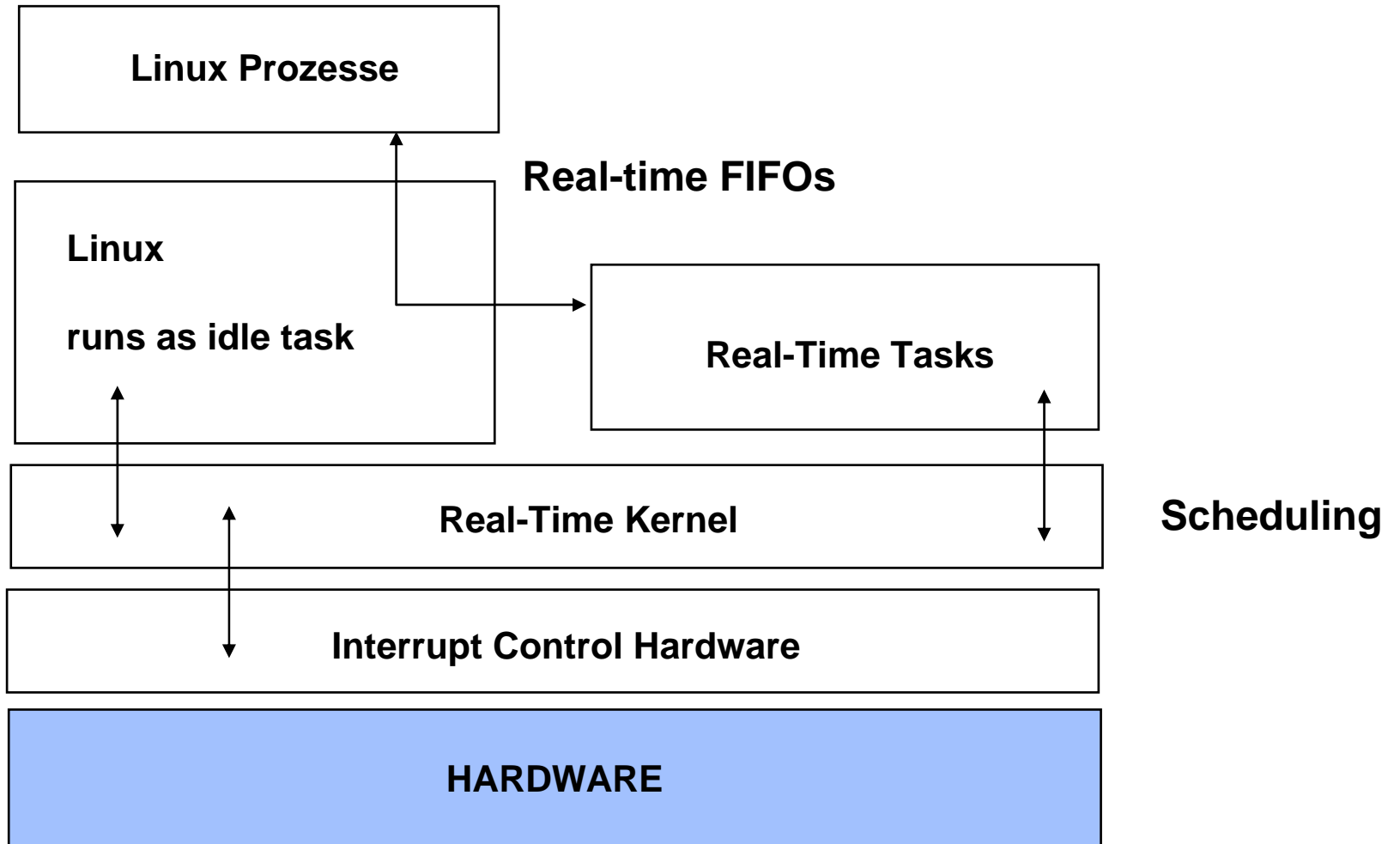
Erweiterung zu PSE50 um Systeme mit Speicherverwaltungseinheit. Unterstützt mehrere Prozesse aber kein Dateisystem.

PSE53 – Multi-purpose Realzeit-Profil:

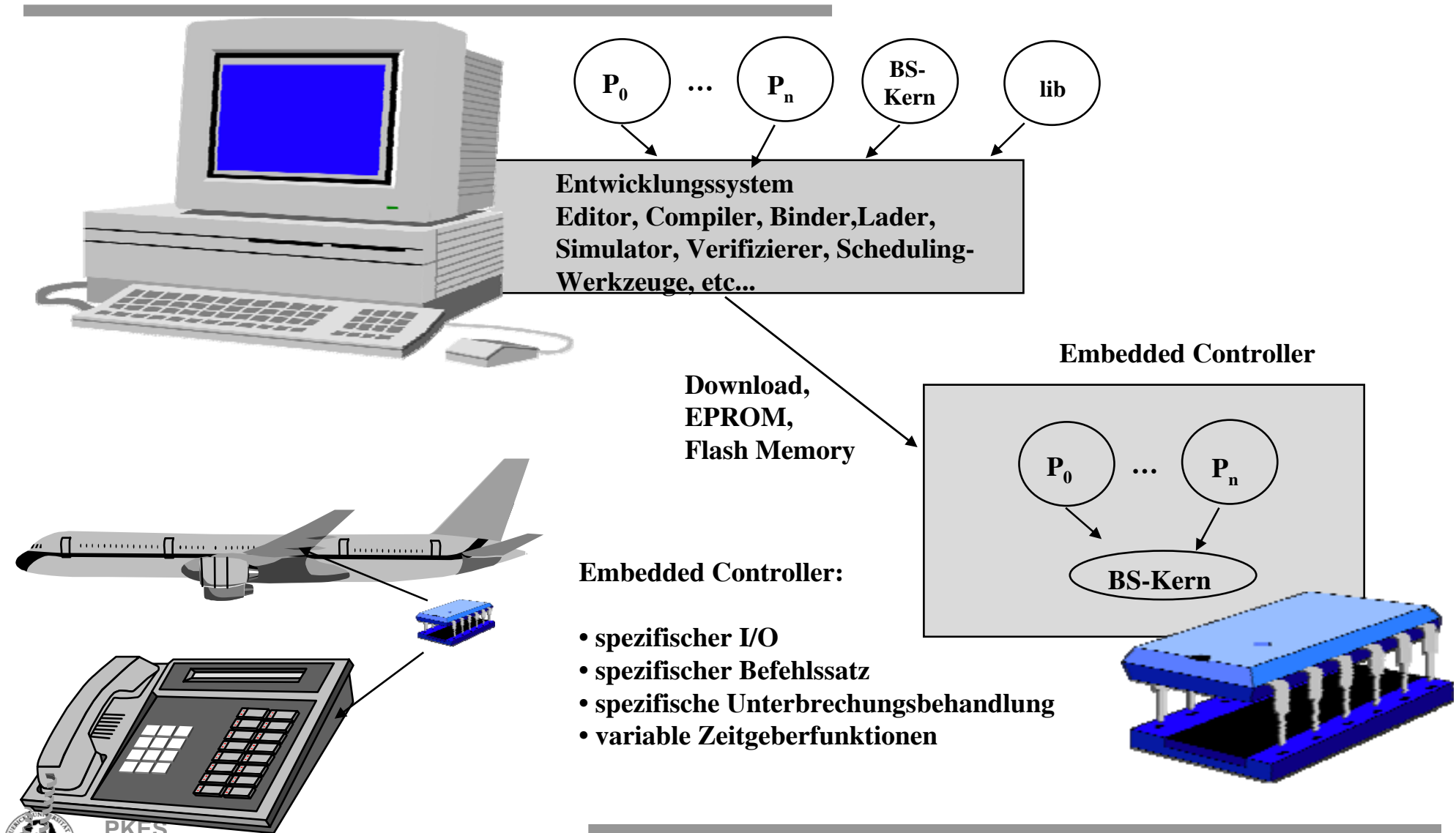
Unterstützt Koexistenz von Echtzeitsystemen und Nicht-Echtzeitsystemen. Für Ein-/ Mehrprozessorsysteme mit MMU, Massenspeicher, Netzwerk etc..



Architektur und Einbettung von RTLinux



Schema einer Cross-Entwicklungsumgebung: Konfigurierbarer BS-Kern



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

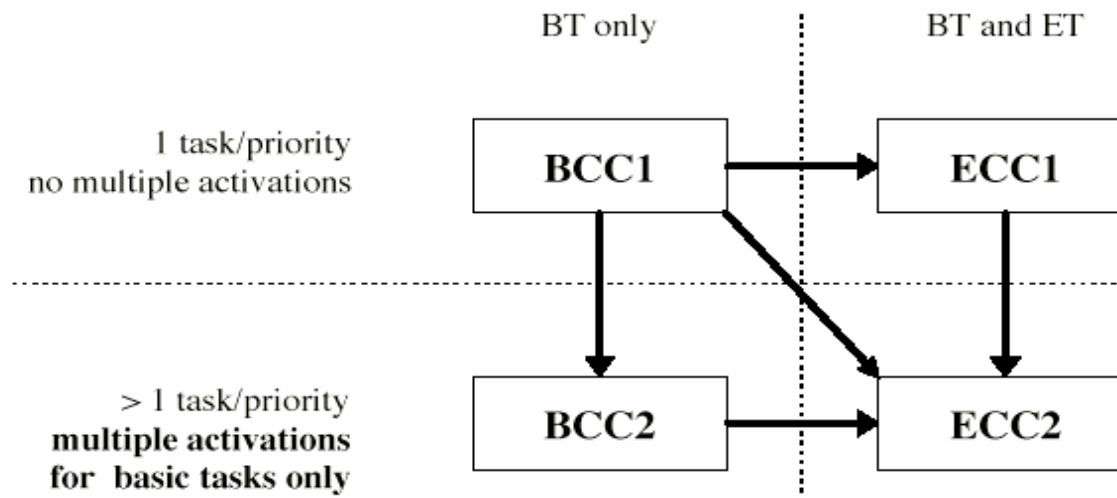
„Vehicle Distributed eXecutive“.

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



OSEK OS - Conformance Classes (CC)

- Generell zwei Gruppen von CC
 - Basic (BCC) erlaubt nur Basic Tasks
(Basic Tasks haben keinen *waiting*-Zustand, sie können unterbrechbar oder nicht unterbrechbar sein)
 - Extended (ECC) erlaubt auch extended Tasks
(Extended Tasks können im Wait-Zustand auf Events warten)
- Aufwärts-kompatible CC's

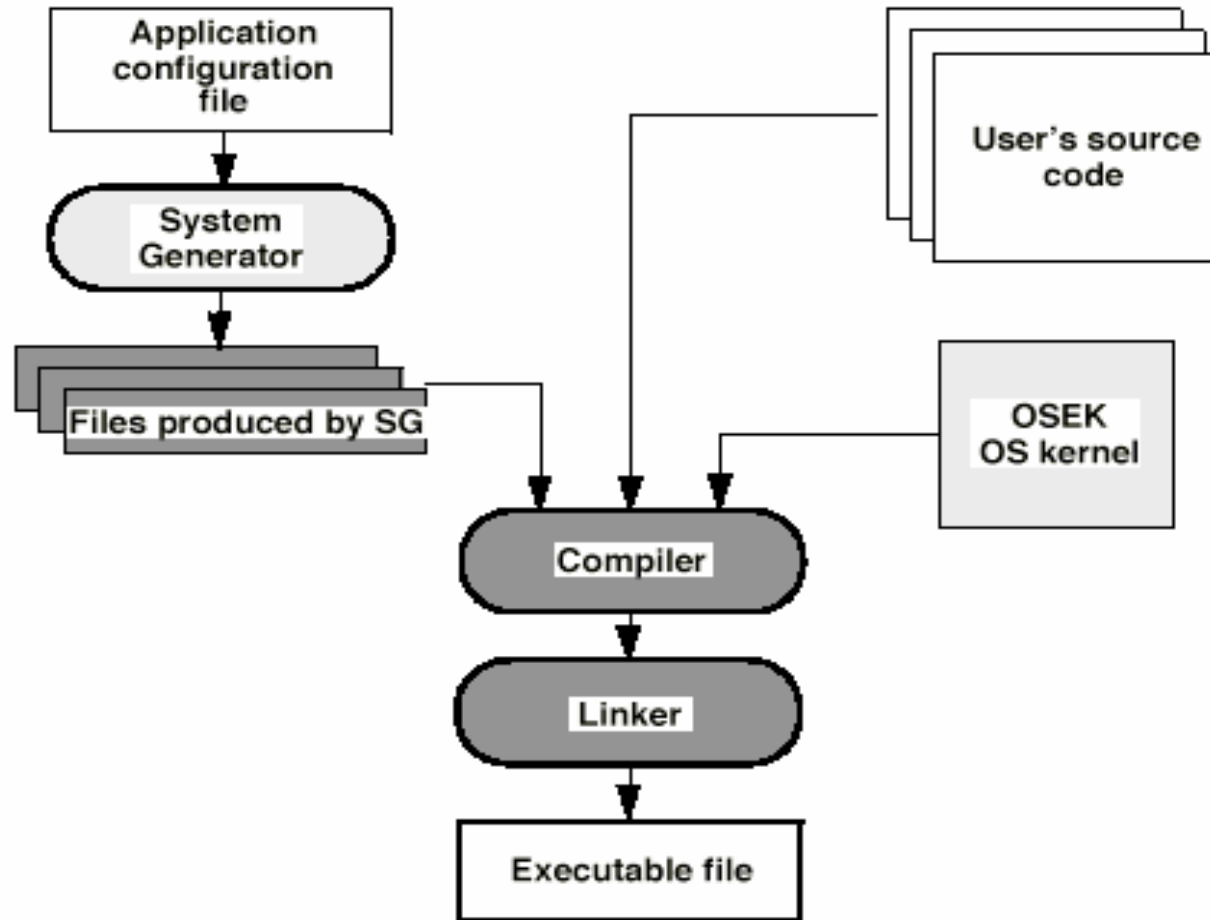


Minimal parameters of implementations

	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 <i>suspended</i> 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			



Application building process in OSEK



-  - user defined files
-  - OSEK Operating System components
-  - files produced by SG
-  - development software



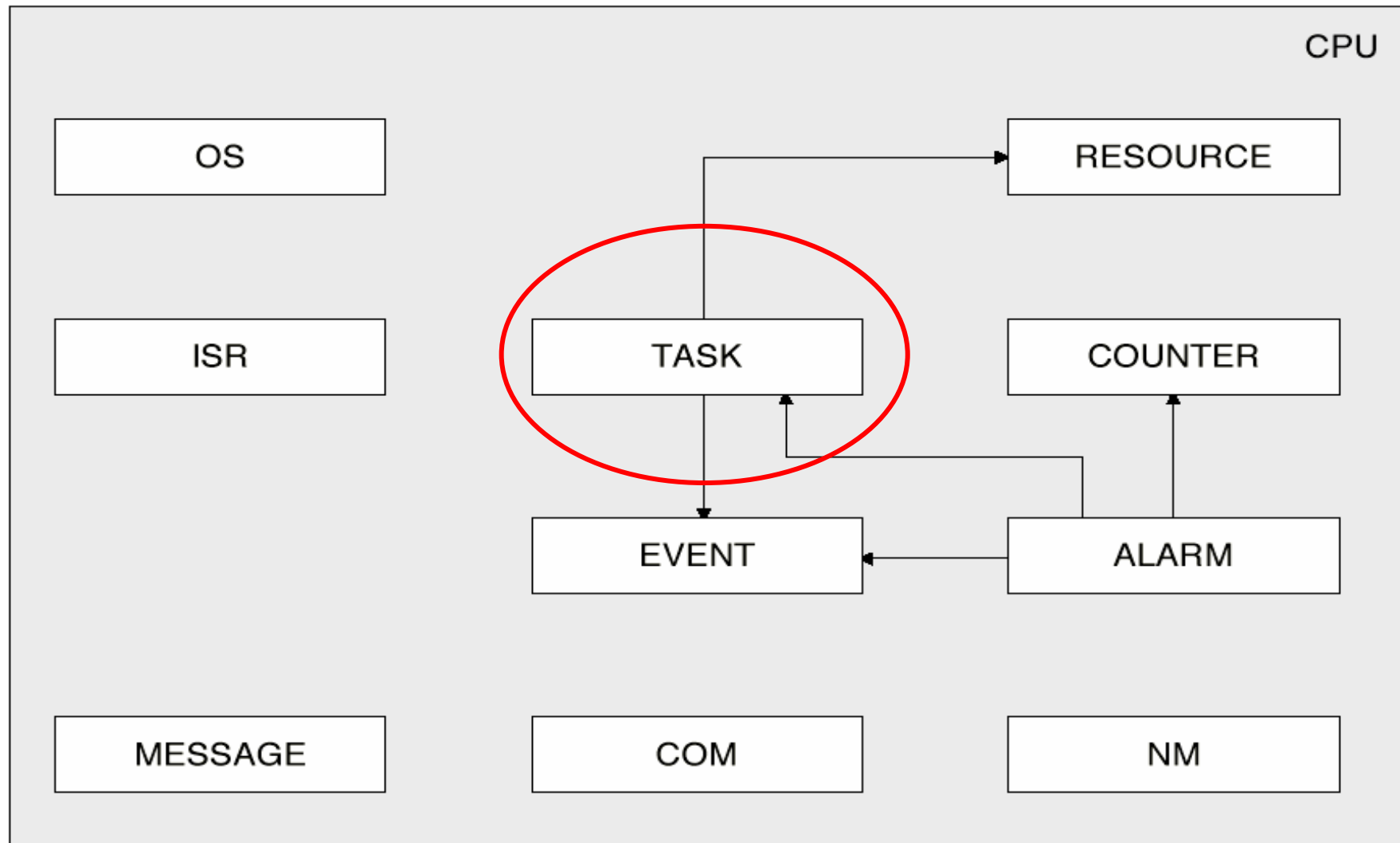
Beispiel: OSEK Konfigurationsdatei

```
[Property]
TargetMCU = HC08;
ConformanceClass =ECC1;
SimpleScheduler = OFF;
ExtendedStatus = ON;
HookRoutines = ON;
ErrorHandler = ON;
ContextSwitchRoutine = ON;
InterruptMaskControl = OFF;
TaskIndexMethod = OFF;
PersistentNode = OFF;
TaskOwnStack = ON;
TaskBasePage = ON;
StackPool = ON;
SchedulerPolicy = MIXPREEMPT;
CounterSize = 16;
Alarms = ON;
AlarmList = ON;
Resources = ON;
FastResource = OFF;
Events = ON;
StateMessage = ON;
StateMsgDefaultValue = ON;
StateMsgTimeStamp = ON;
EventMessage = ON;
EventMsgTimeStamp = ON;
EventMsgOneToN = OFF;
ActivateOnMsg = ON; /* Message signalling mechanism */
AlarmOnMsg = ON;
SetEventOnMsg = ON;
```

```
[Scheduler]
DefineScheduler( 4, 5, 128, , 64 );
[Interrupt management]
DefineInterrupts( 0x8, 0, 0, 64 );
[User's hook]
DefineHooks ( OSError, OSPreTask, OSPostTask);
[Tasks]
DefineTask ( TASKSND, BASIC|PREEMPT|ACTIVATE, 3, TaskSND );
DefineTask ( TASKRCV, PREEMPT| BASIC|OWNSTACK, 1,
TaskRCV,,, TaskStack, TASKSTACKSIZE );
DefineTask ( TASKPROD, PREEMPT| EXTENDED|ACTIVATE|POOLSTACK, 2,
TaskProd,,, POOL );
DefineTask ( TASKCONS, NONPREEMPT|BASIC|POOLSTACK, 4,
TaskCons,,, POOL );
DefineStackPool( POOL, 70, 2 );
[Resources]
DefineResource( MSGACCESS, 1);
[Counters]
DefineSystemTimer( SYSTEMTIMER, -1, 10, 1000000 );
DefineCounter( MSGCOUNTER, 6, 1 );
[Alarms]
DefineAlarm( MSGALARM, SYSTEMTIMER, TASKSND );
DefineAlarm( PRODALARM, SYSTEMTIMER, TASKPROD, TIMEVENT );
DefineAlarm( EVMSGALARM, MSGCOUNTER, TASKCONS );
[State Messages]
DefineStateMessage( MsgA, MSGATYPE, sizeof(MSGATYPE),
WithoutTimeStamp );
ActivateOnMessage( MsgA, TASKRCV );
DefineMessageAlarm( MsgA, MSGALARM, 100 );
[Event Messages]
DefineEventMessage( MsgB, MSGBTYPE, sizeof(irt), 5, 1,
WithOverwriteCheck, WithTimeStamp );
```



OSEK Komponenten



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



Basic Tasks

- **running**

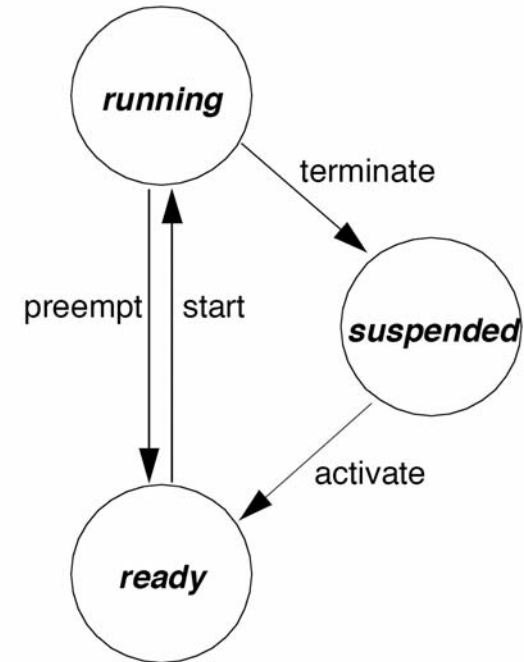
In the running state, the CPU is assigned to the task, so that its instructions can be executed. Only one task can be in this state at any point in time, while all the other states can be adopted simultaneously by several tasks.

- **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**

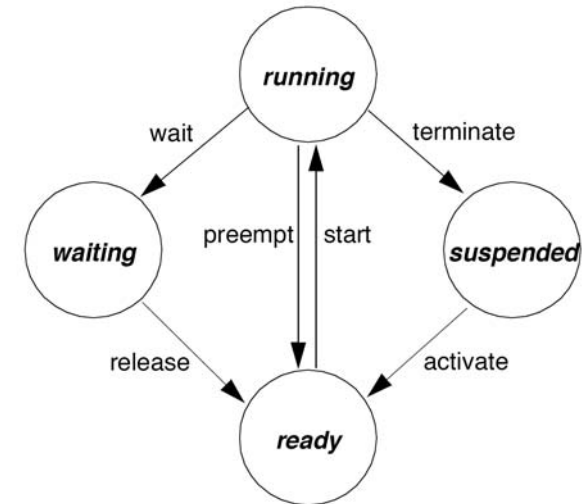
In the suspended state the task is passive and can be activated.



Transition	Former state	New state	Description
activate	suspended	ready	A new task is entered into the <i>ready</i> list by a system service.
start	ready	running	A <i>ready</i> task selected by the scheduler is executed.
preempt	running	ready	The scheduler decides to start another task. The <i>running</i> task is put into the <i>ready</i> state.
terminate	running	suspended	The <i>running</i> task causes its transition into the <i>suspended</i> state by a system service.



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 <i>ready</i> list by a system service.
start	ready	running	A <i>ready</i> task selected by the scheduler is executed.
wait	running	waiting	To be able to continue operation, the <i>running</i> task requires an event. It causes its transition into the <i>waiting</i> state by using a system service.
release	waiting	ready	Events have occurred which a task has waited on.
preempt	running	ready	The scheduler decides to start another task. The <i>running</i> task is put into the <i>ready</i> state.
terminate	running	suspended	The <i>running</i> task causes its transition into the <i>suspended</i> state by a system service.



Task Properties

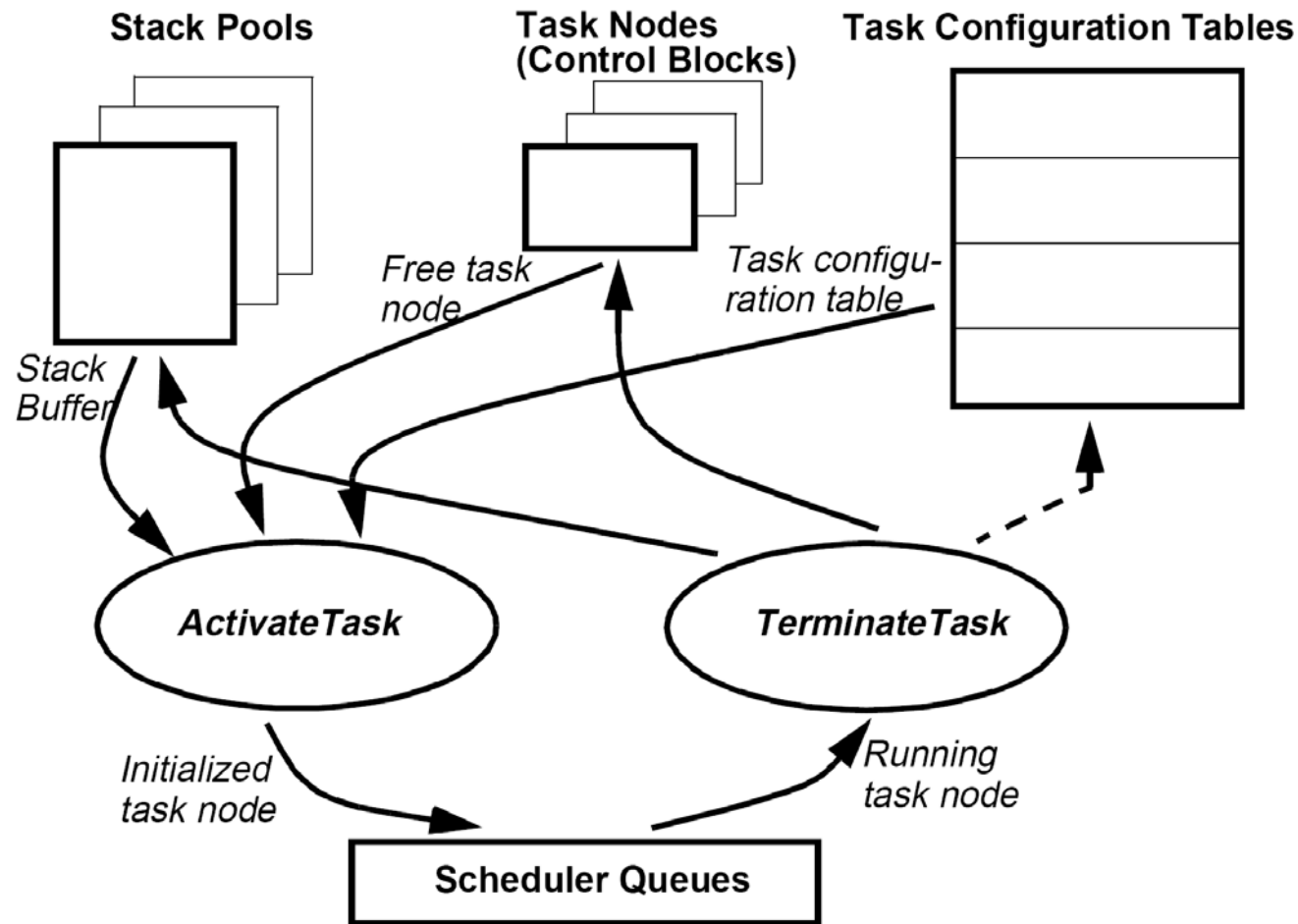
Property Name	Description
BASIC	Basic task
EXTENDED	Extended task
NONPREEMPT	Non-preemptive task
PREEMPT	Preemptive task
ACTIVATE	Activate the task at system start-up
ASSIGNNODE	A persistent task control block is assigned (linked with the task configuration tables)
POOLSTACK	A stack should be allocated to the task during task activation - dynamic stack allocation from the stack pool
OWNSTACK	A stack is explicitly assigned to the task (address of the top of the stack is included into the task configuration table)
NODESTACK	A task node stack is implicitly assigned to be used by the task (during the task activation)
ASSIGNSTK	A persistent stack from the stack pool (the pool identifier is included into the task configuration table)



System services related to tasks:

Service Name	Description
ActivateTask	Activates the task, i.e. put it from the <i>suspended</i> into the <i>ready</i> state
TerminateTask	Terminates the task, i.e. put it from the <i>ready</i> into the <i>suspended</i> state
ChainTask	Terminates the task and activates a new one immediately
Schedule	Yields control to a higher-priority ready task (if any exists)
GetTaskId	Gets the identifier of the running task
GetTaskState	Gets the status of the specified task



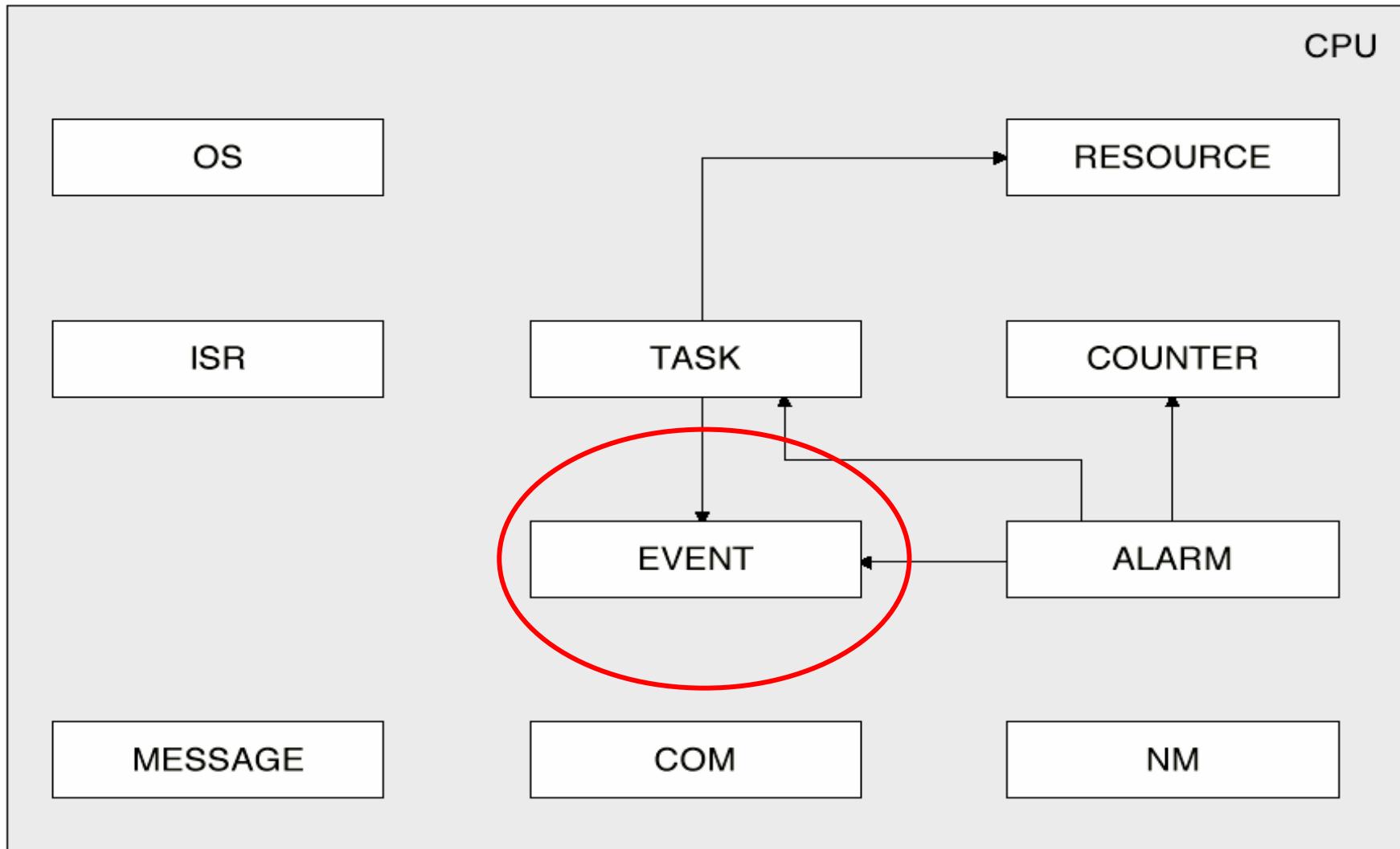


```

DefineTask( <TaskName>, <TaskProperties>, <TaskPriority>,
<EntryPoint> [, <TaskBank>] [, <InterruptMask>]
[, <TaskStack>] [, <TaskStackSize>] );

```





OSEK Komponenten



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

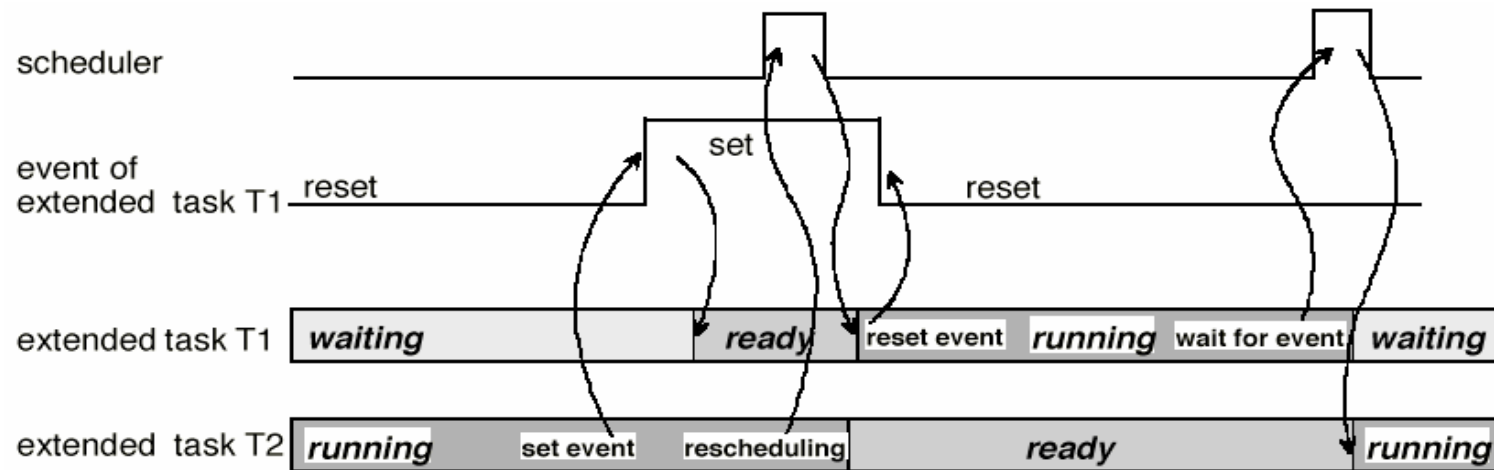


System services related to Event Processing:

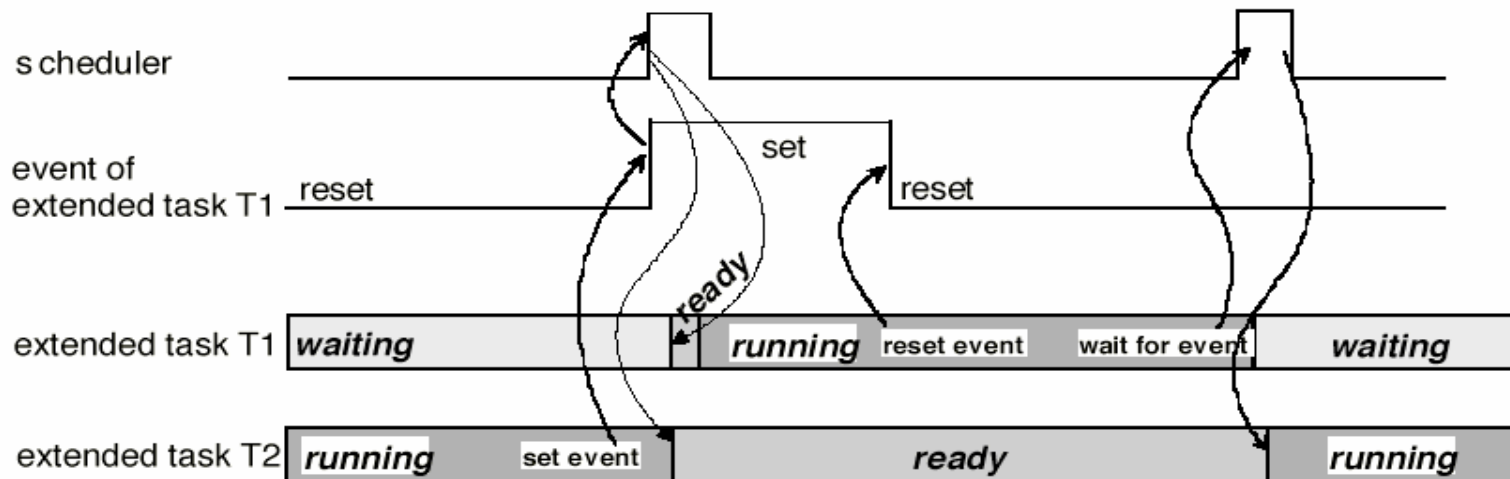
Service Name	Description
SetEvent	Sets events of the given task according to the event mask
ClearEvent	Clears events of the calling task according to the event mask
GetEvent	Gets the current event setting of the given task
WaitEvent	Transfers the calling task into the waiting state until specified events are set



Synchronisation of non-preemptive tasks



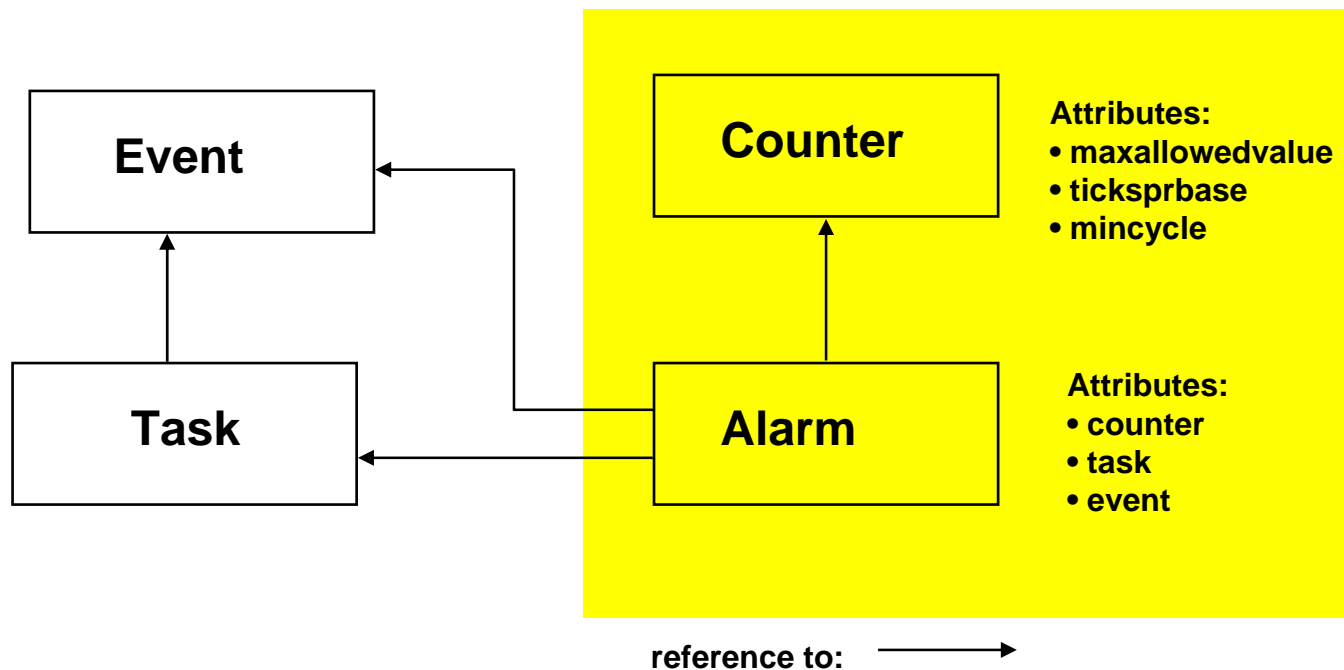
Synchronisation of fully-preemptive tasks



An Alarm may start a task or set an event.

An alarm is statically assigned at system generation time to:

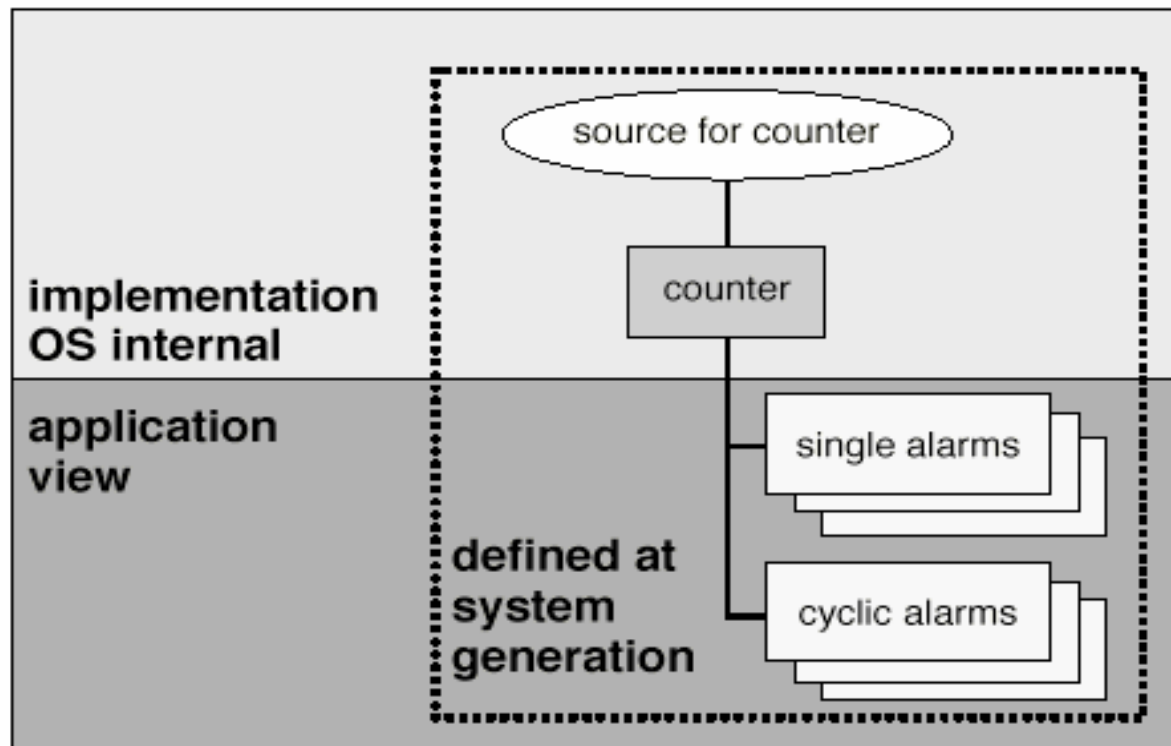
- one counter
- one task
- a notation if the task is to be activated or the event is to be set



Alarms

- Processing of recurring 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

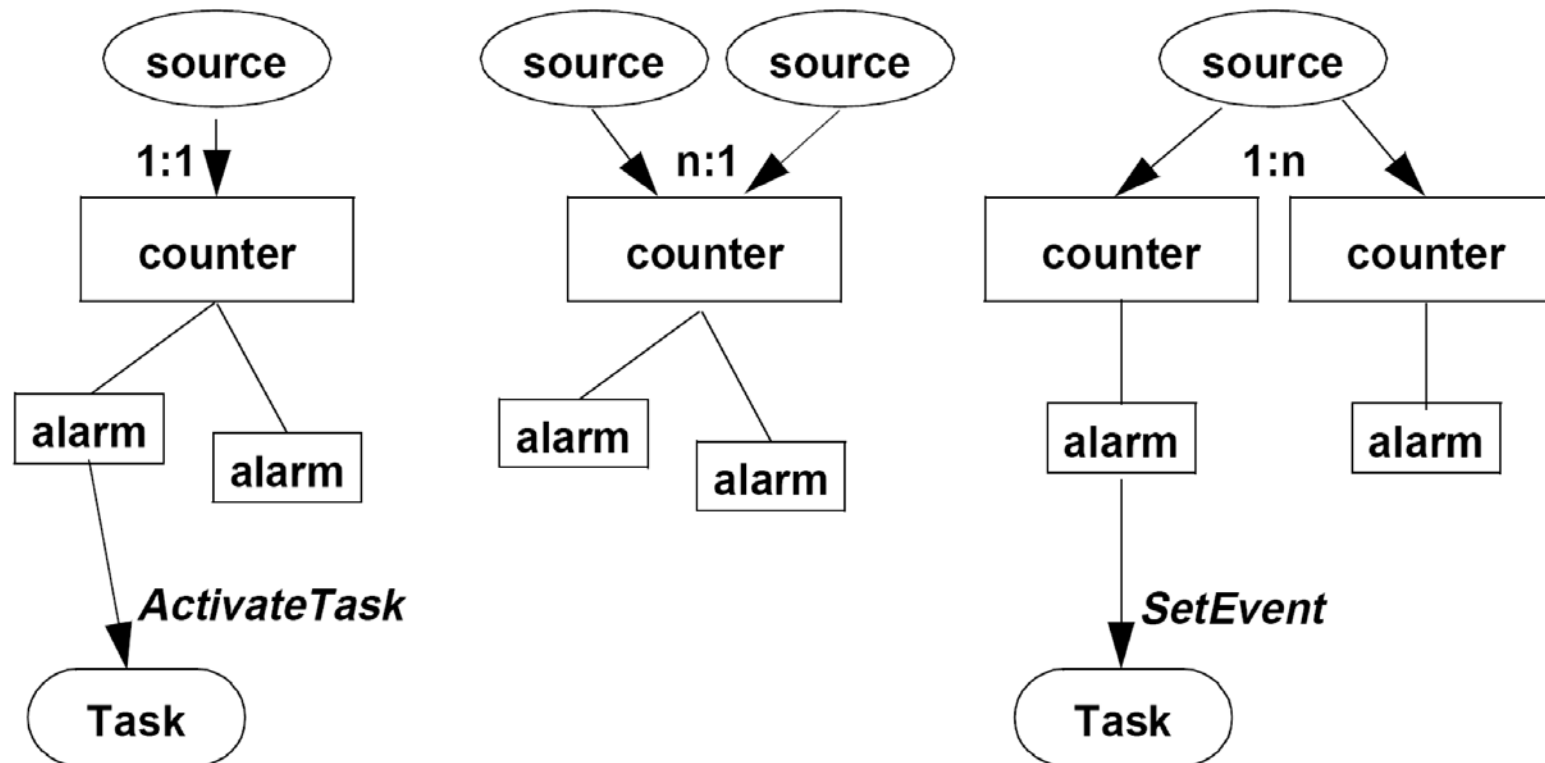


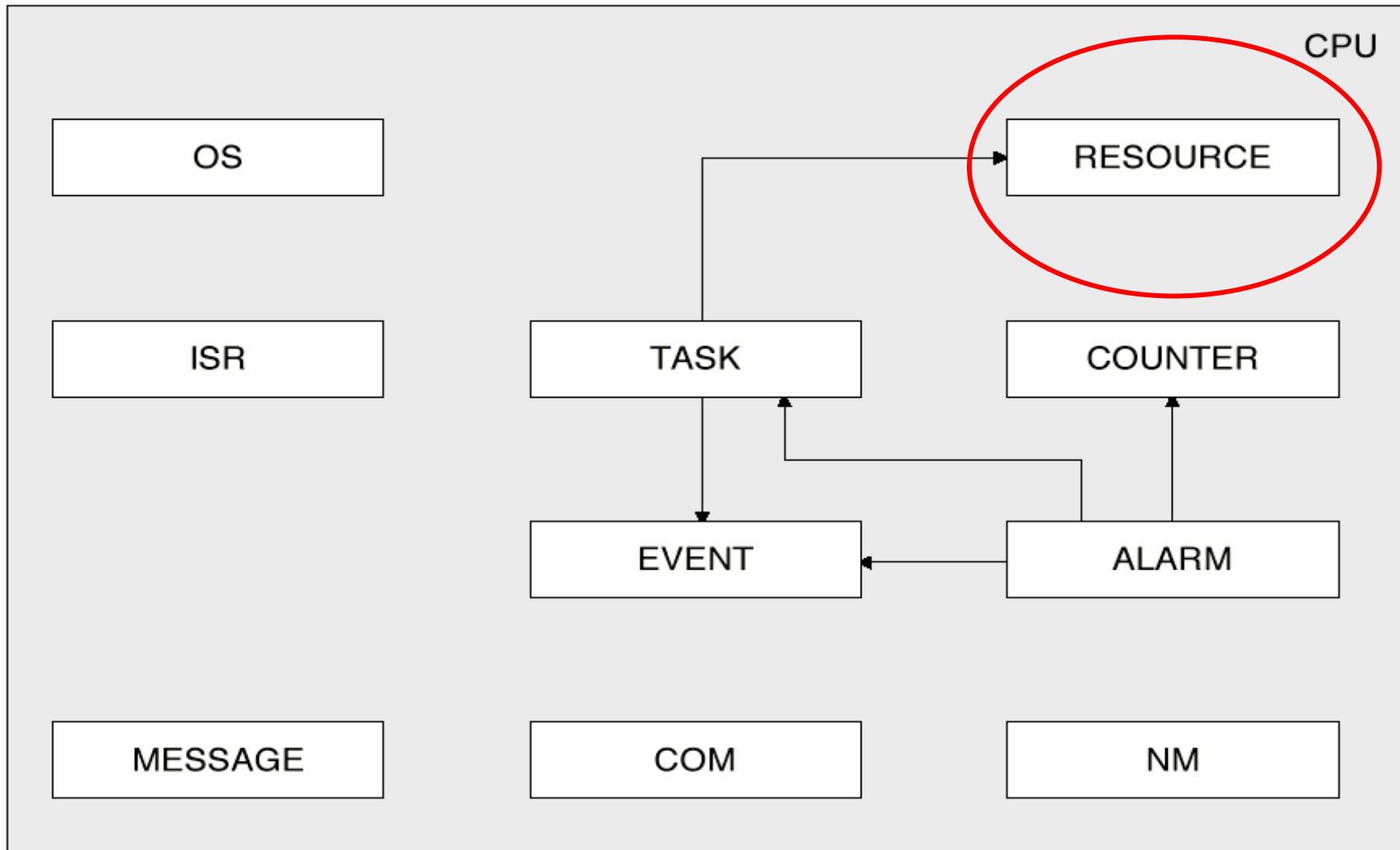
Counter and Alarm Management Run-time Services

Service Name	Description
InitCounter	Sets the initial value of the counter
CounterTrigger	Increments the counter value
GetCounterValue	Gets the counter current value
GetCounterInfo	Gets counter parameters
SetRelAlarm	Sets the alarm with a relative start value
SetAbsAlarm	Sets the alarm with an absolute start value
CancelAlarm	Cancels the alarm: the alarm is transferred into the STOP state
GetAlarm	Gets the time left before the alarm expires



Counters and Alarms





OSEK Komponenten



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

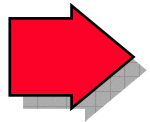
The resource management is mandatory for all conformance classes.



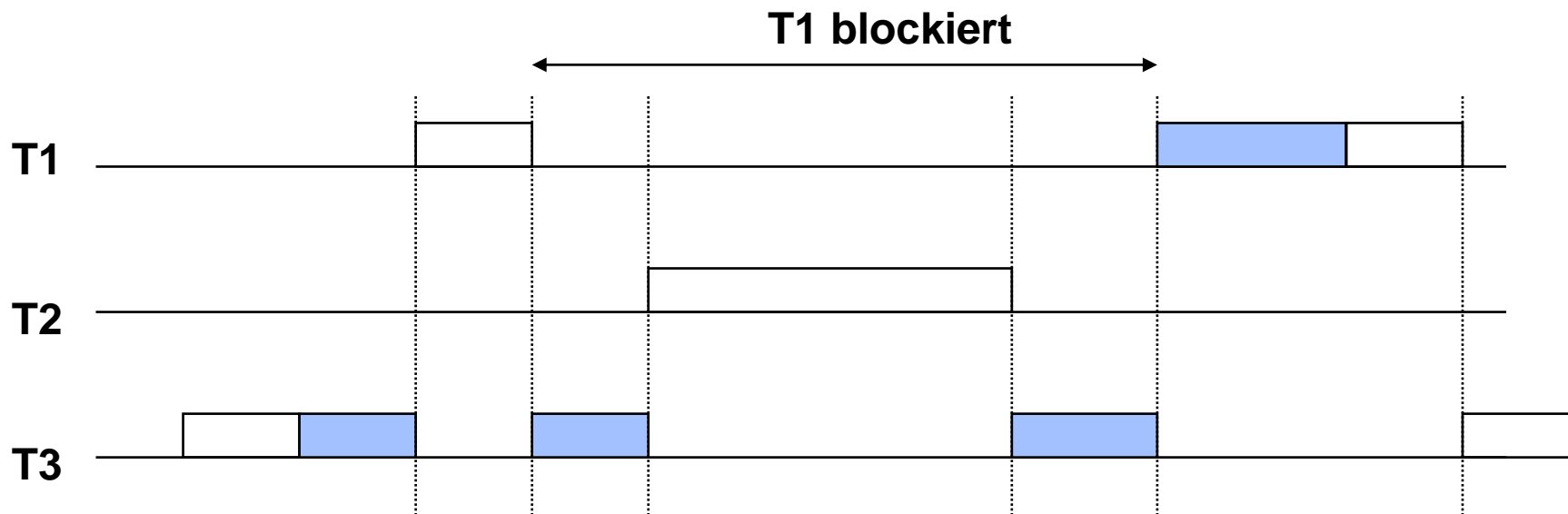
Das Problem: Prioritätsumkehrung (Priority Inversion)

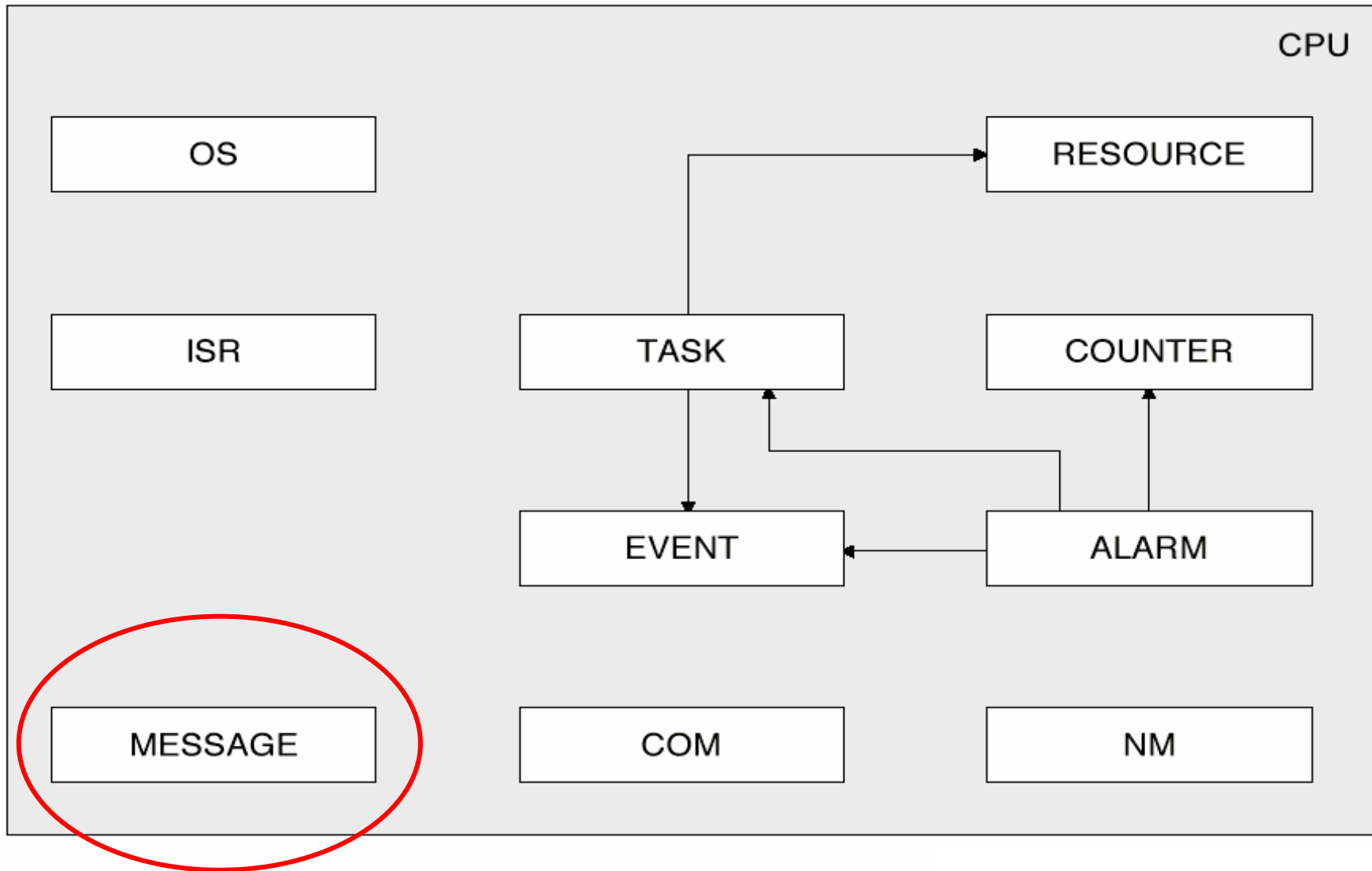
Annahme: Prioritätsbasiertes, unterbrechbares Scheduling.
Standard für die meisten RT-Kerne

Problem: nicht vorhersagbare Blockierung einer Task mit hoher Priorität durch
Tasks mit niedrigerer Priorität.



Prioritätsumkehrung !





OSEK Komponenten



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 unique 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

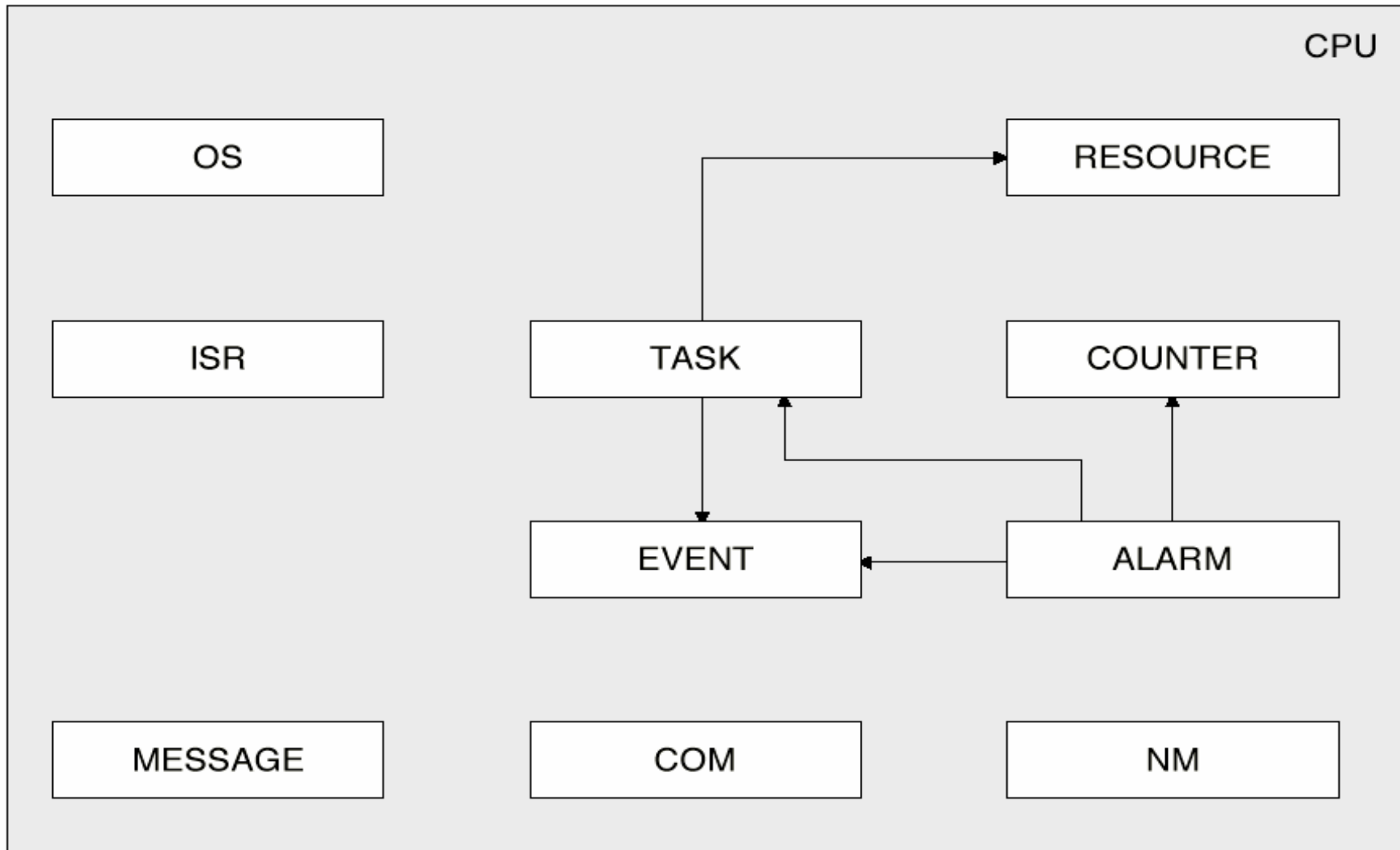
State Message	Event Message
No buffering	Buffering in a FIFO-queue
No consumption of message	Consumption of messages
Direct access possible in non-preemptive systems (no copies)	No direct access possible (always copies)
Static definition of task activation or event signalling	
Static definition of a 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 Systemunterstützung für das Warten auf Nachrichten zur Verfügung. Hierzu kann der allgemeine „Event-“ Mechanismus verwendet werden.**





OSEK Komponenten

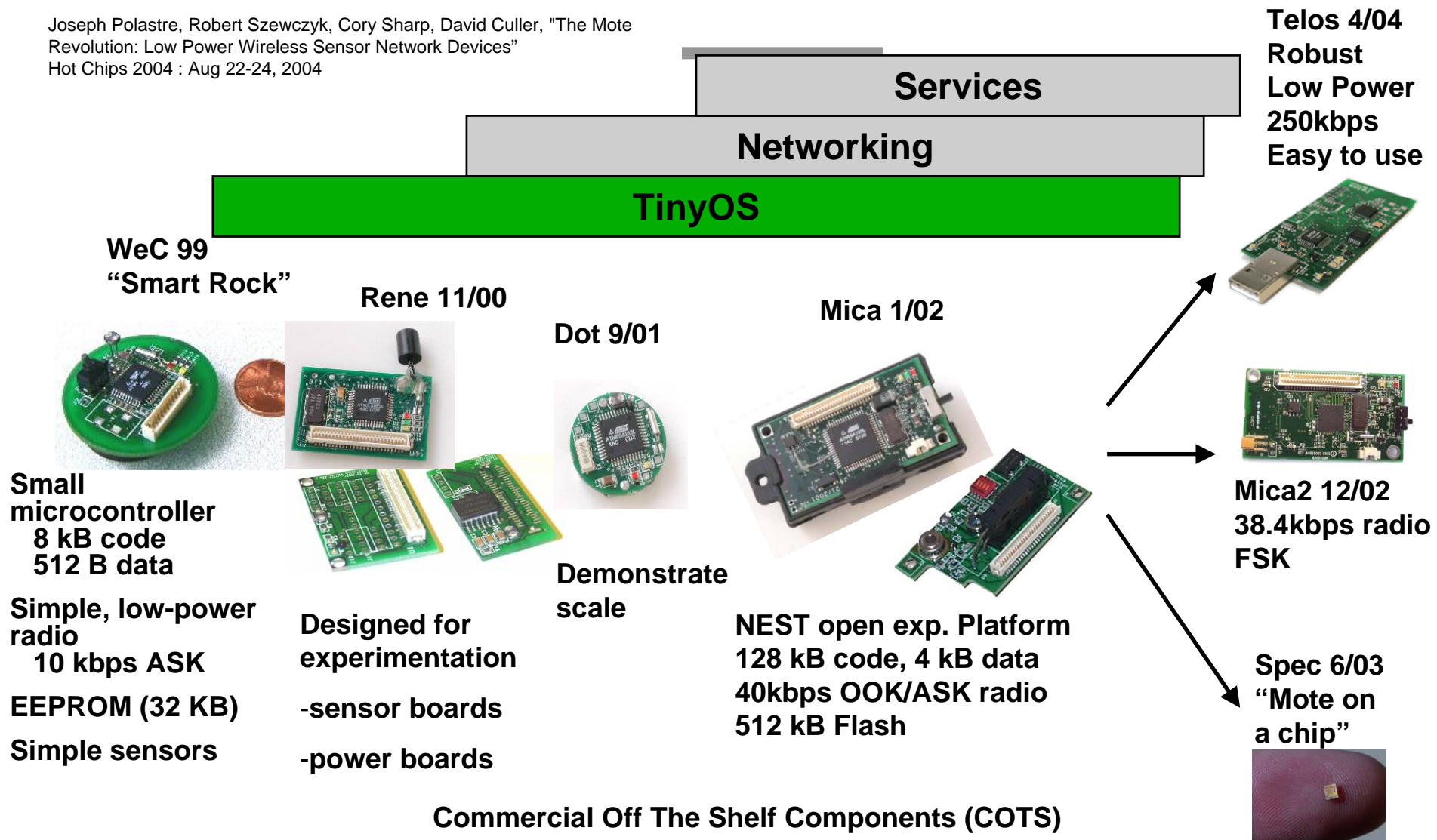


Operating Systems for networked resource constrained embedded systems



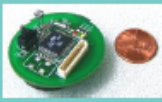






Open Experimental Platform

Joseph Polastre, Robert Szewczyk, Cory Sharp, David Culler, "The Mote Revolution: Low Power Wireless Sensor Network Devices"
Hot Chips 2004 : Aug 22-24, 2004



Mote Evolution

Joseph Polastre, Robert Szewczyk, Cory Sharp, David Culler, "The Mote Revolution: Low Power Wireless Sensor Network Devices"
Hot Chips 2004 : Aug 22-24, 2004

Mote Type Year	<i>WeC</i> 1998	<i>René</i> 1999	<i>René 2</i> 2000	<i>Dot</i> 2000	<i>Mica</i> 2001	<i>Mica2Dot</i> 2002	<i>Mica 2</i> 2002	<i>Telos</i> 2004
								

Microcontroller

Type	AT90LS8535	ATmega163	ATmega128		TI MSP430
Program memory (KB)	8	16	128		60
RAM (KB)	0.5	1	4		2
Active Power (mW)	15	15	8	33	3
Sleep Power (μ W)	45	45	75	75	6
Wakeup Time (μ s)	1000	36	180	180	6

Nonvolatile storage

Chip	24LC256	AT45DB041B		ST M24M01S
Connection type	I ² C	SPI		I ² C
Size (KB)	32	512		128

Communication

Radio	TR1000	TR1000	CC1000	CC2420
Data rate (kbps)	10	40	38.4	250
Modulation type	OOK	ASK	FSK	O-QPSK
Receive Power (mW)	9	12	29	38
Transmit Power at 0dBm (mW)	36	36	42	35

Power Consumption

Minimum Operation (V)	2.7	2.7	2.7	1.8
Total Active Power (mW)	24	27	44	41

Programming and Sensor Interface

Expansion	none	51-pin	51-pin	none	51-pin	19-pin	51-pin	10-pin
Communication	IEEE 1284 (programming) and RS232 (requires additional hardware)							USB
Integrated Sensors	no	no	no	yes	no	no	no	yes



TinyOS

**Ein Betriebssystem für "sehr kleine" Systeme.
Aus der Forschung über Sensornetze in Berkeley entstanden.**

Unterstützung von Nebenläufigkeit

→ Strukturierung von Aktivitäten in unabhängige Tasks

Extreme Beschränkung der Ressourcen

**→ Typisch: RAM < 10k, Flash < 100k, Batterielebensdauer > 1Jahr,
Kommunikation mit geringer Bandbreite und sehr wenig Energie.**

Adaptierung an Hardware-Änderungen

**→ Hardwareunabhängige Abstraktionen zur Programmierung von
Anwendungen.**

Robuste Software

→ Modularisierung, Komponentenorientierung



Literatur

D. Gay, M. Welsh, P. Lewis, E. Brewer, R.v. Behren, D. Culler, "The nesC Language: A Holistic Approach to Networked Embedded Systems", in Proceedings of Programming Language Design and Implementation (PLDI, 2003 (available from <http://nesc.sourceforge.net>)

P. Lewis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woll, E. Brewer, D. Culler, "The Emergence of Networking Abstractions and Techniques in TinyOS", in Proceedings of 1st conference on Symposium on Networked Systems Design and Implementation –vol.1, San Francisco, CA, USA, 2004

Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. 2000. System architecture directions for networked sensors. SIGPLAN Not. 35, 11 (Nov. 2000), 93-104

William Stallings "Operating Systems, Internals and Design Principles, 6. Edition, Pearson Education, 2009

A. Dunkels, O. Schmidt, T. Voigt, "Using Protothreads for Sensor Node Programming", in Proceedings of the REALWSN 2005 Workshop on RealWorld Wireless Sensor Networks

E.A. Lee, Y. Zhao "Reinventing Real Time", in F.Kordon, J. Sztipanovits (Eds.): Monterey Workshop 2006, LNCS 4322, pp. 1-25, Springer-Verlag Berlin heidelberg, 2007



TinyOS Abstraktionen

Interagierende Komponenten

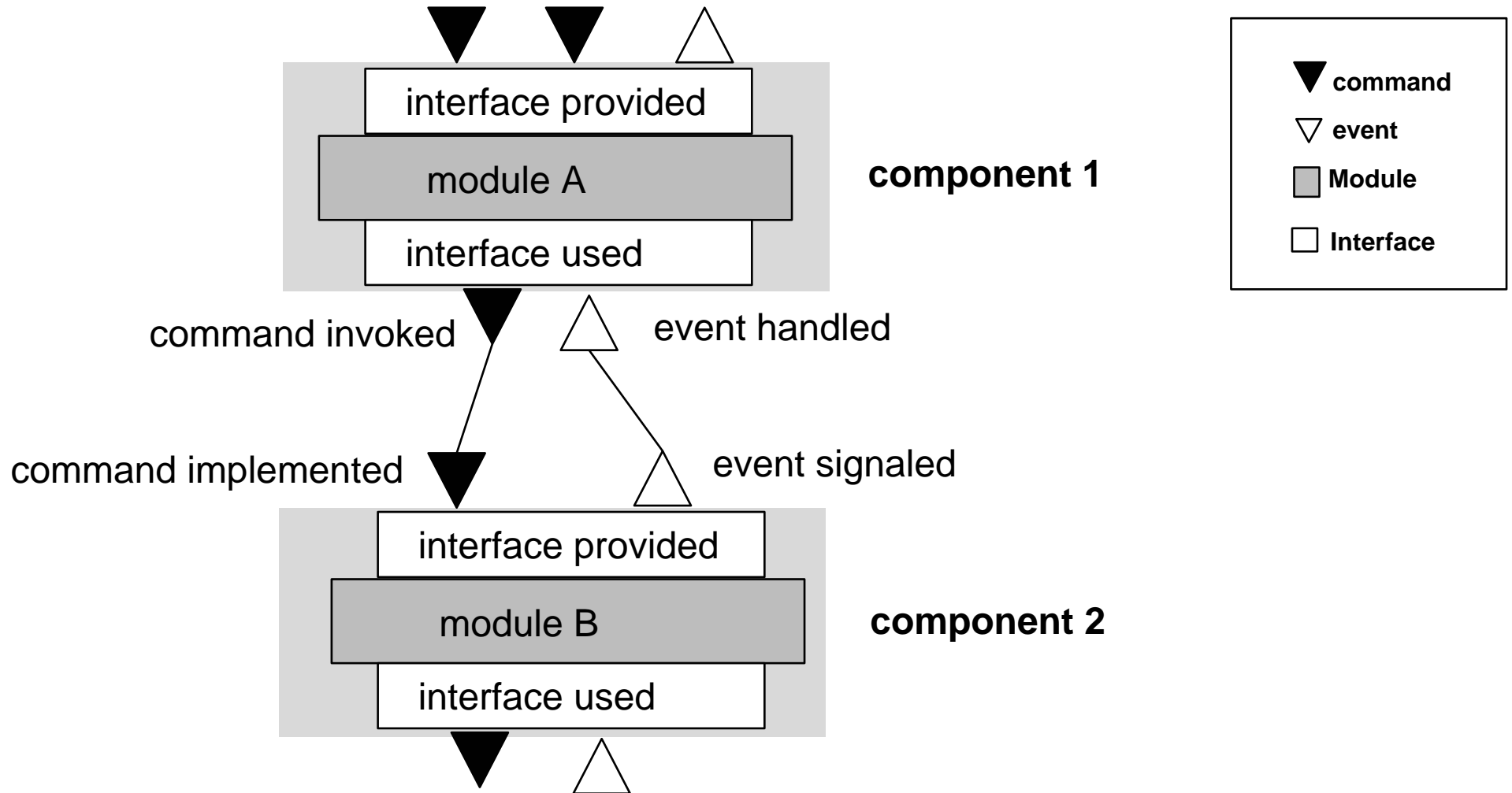
Aktivität:	Struktur:	Zustand:
Events Commands Tasks Handlers	Components Modules Configurations	frame (static memory) Resources (T2)

Ausführungsmodell:
Ausführungskomponente:

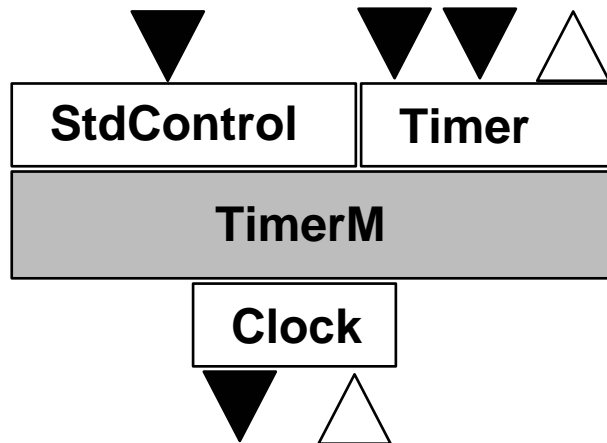
Zustandsmaschinen
(FIFO-) Scheduler



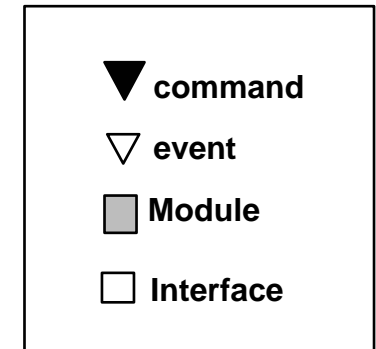
TinyOS Komponenten



TinyOS Module



```
Module TimerM {  
  provides {  
    interface StdControl  
    interface Timer;  
  }  
  use interface Clock as Clock;  
} ...
```

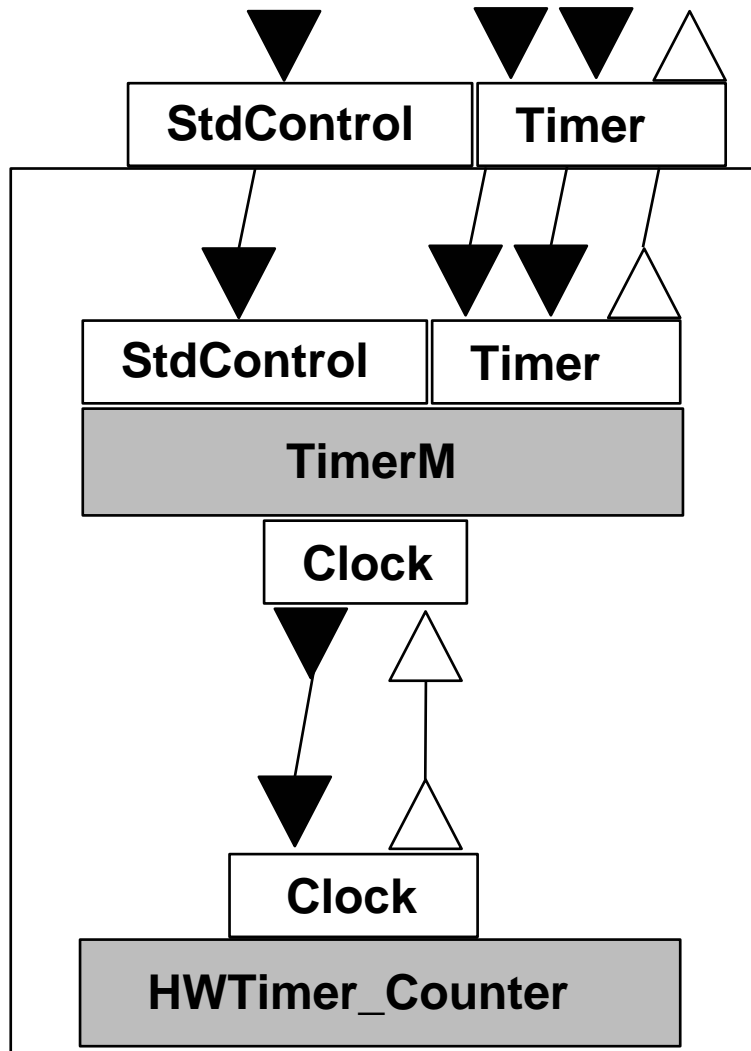


```
interface StdControl {  
  command result_t init ();  
  command result_t start();  
  command result_t stop();  
}
```

```
interface Timer {  
  command result_t start(char type, uint32_t, interval);  
  command result_t stop();  
  event result_t fired();  
}
```

```
interface Clock {  
  command result_t setRate(char interval, char scale);  
  event result_t fire();  
}
```

TinyOS Configuration

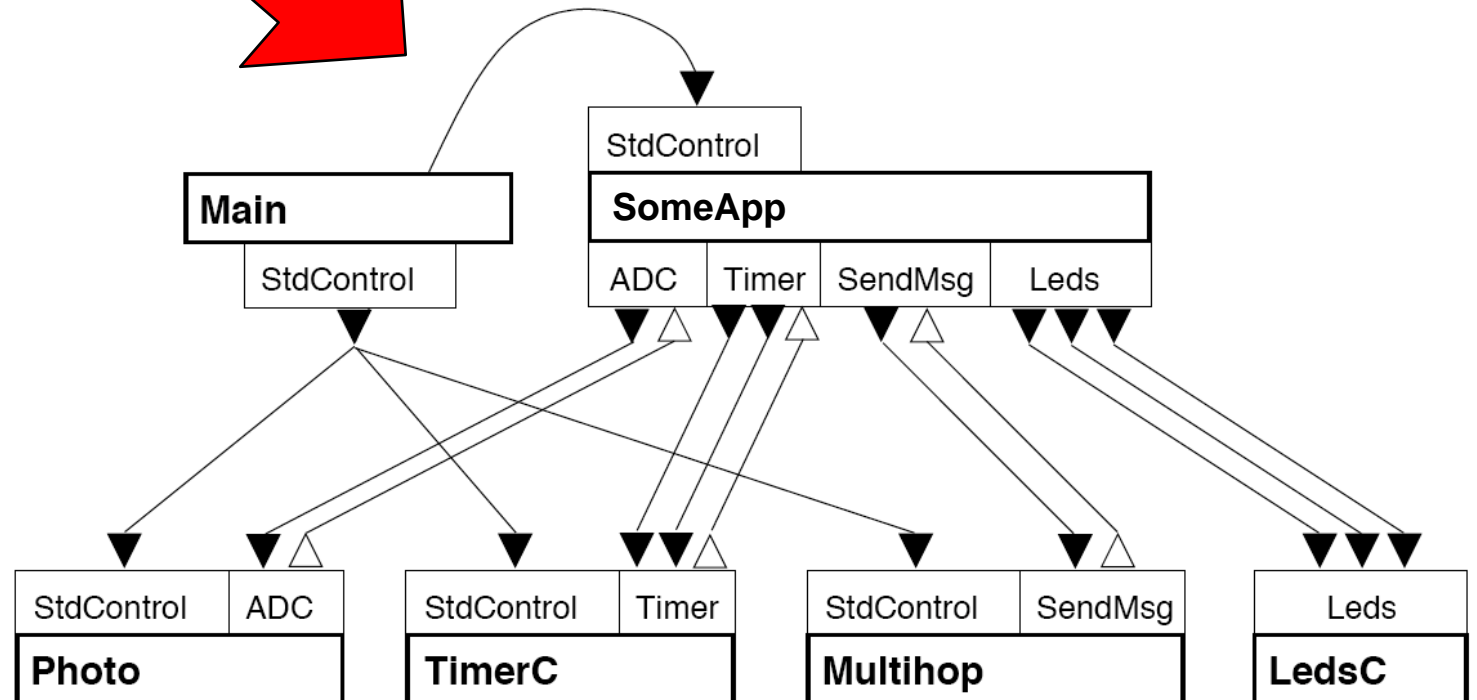
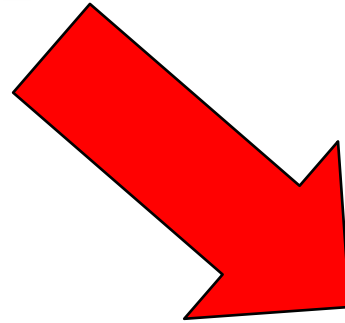
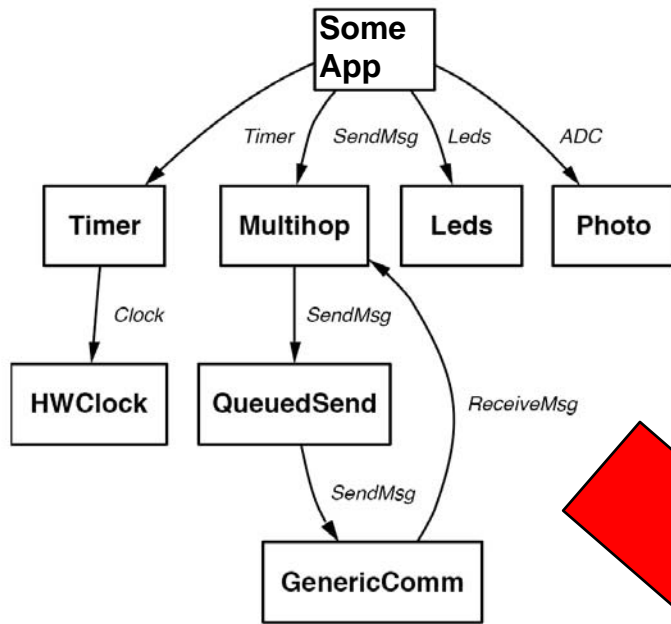


```
configuration TimerC {
  provides {
    interface StdControl;
    interface Timer;
  }
}
```

```
implementation {
  components TimerM, HWTimer_Counter;
  StdControl = TimerM.StdControl;
  Timer = TimerM.Timer;
  TimerM.Clk → HWTimer_Counter.Clock
}
```



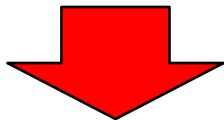
Beispielanwendung: "SomeApp"



Interface:

is bi-directional
contains:

commands
events



split phase operations

```
interface StdControl {  
    command result_t init();  
}  
  
interface Timer {  
    command result_t start(char type, uint32_t interval);  
    command result_t stop();  
    event result_t fired();  
}  
  
interface Clock {  
    command result_t setRate(char interval, char scale);  
    event result_t fire();  
}  
  
interface Send {  
    command result_t send(TOS_Msg *msg, uint16_t length);  
    event result_t sendDone(TOS_Msg *msg, result_t success);  
}  
  
interface ADC {  
    command result_t getData();  
    event result_t dataReady(uint16_t data);  
}
```

Component definitions:



split phase operations

application example

-commands are
non-blocking

-effect may be signalled
by event

```
module SomeAppM {
    provides interface StdControl;
    uses interface ADC ;
    uses interface Timer;
    uses interface Send;
}

implementation {

    uint16_t sensorReading;

    command result_t StdControl.init() {
        return call Timer.start(TIMER_REPEAT, 1000);
    }

    event result_t Timer.fired() {
        call ADC.getData();
        return SUCCESS;
    }

    event result_t ADC.dataReady(uint16_t data) {
        sensorReading = data;
        ... send message with data in it ...
        return SUCCESS;
    }

    ...
}
```



split phase operations

Blocking

```
if (send() == SUCCESS) {  
    sendCount++;  
}
```

Split-Phase

```
// start phase  
send();  
-----  
//completion phase  
void sendDone(error_t err) {  
    if (err == SUCCESS) {  
        sendCount++;  
    }  
}
```



TinyOS Concurrency Model

Tasks: represent deferred execution of code. Tasks run to completion and do not preempt each other.

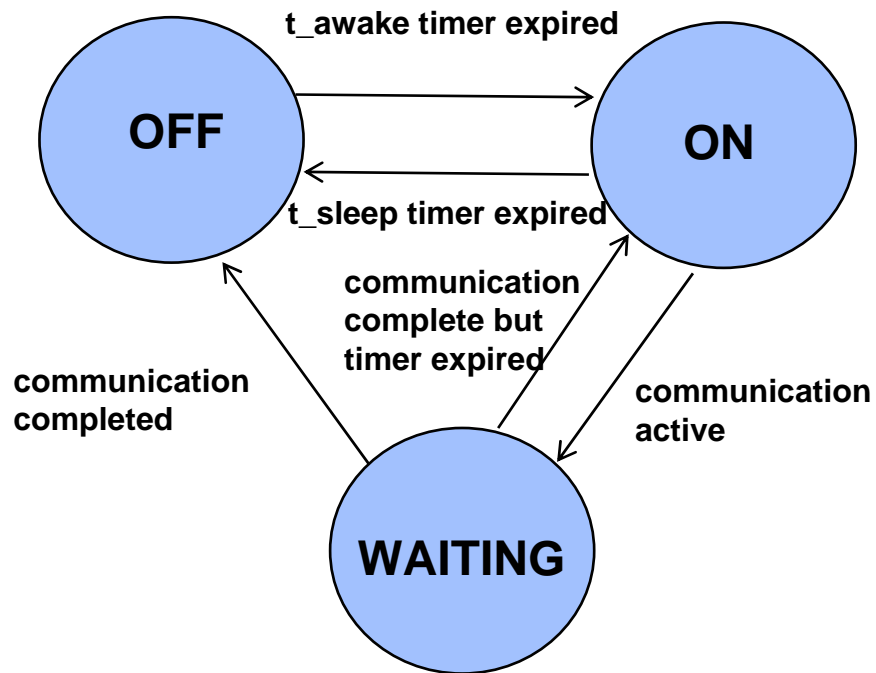
Event Handlers: are executed on events including asynchronous interrupts. Event Handlers run to completion but may preempt a task and can be preempted by another Event Handler. Event Handlers may post a task.

Consequences:

- Tasks cannot block and do not (busy) wait.
- Only a single stack needed.
- (nested) preemption of Event Handlers may cause inconsistencies of shared resources.



State Machine Modell



A. Dunkels, O. Schmidt, T. Voigt, "Using Protothreads for Sensor Node Programming", in Proceedings of the REALWSN 2005 Workshop on RealWorld Wireless Sensor Networks

```
enum {
    ON,
    WAITING,
    OFF
} state;

void radio_wake_eventhandler() {
    switch(state) {

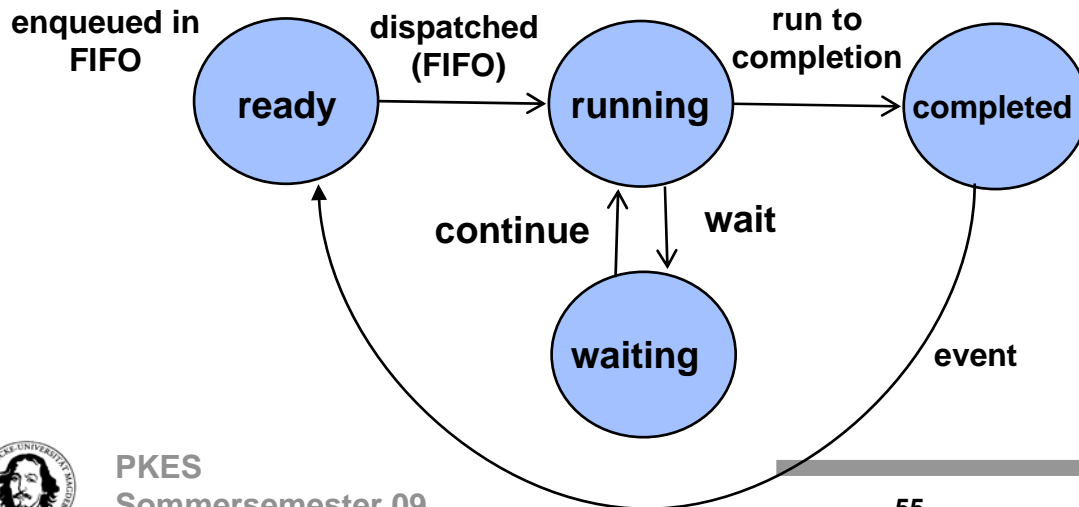
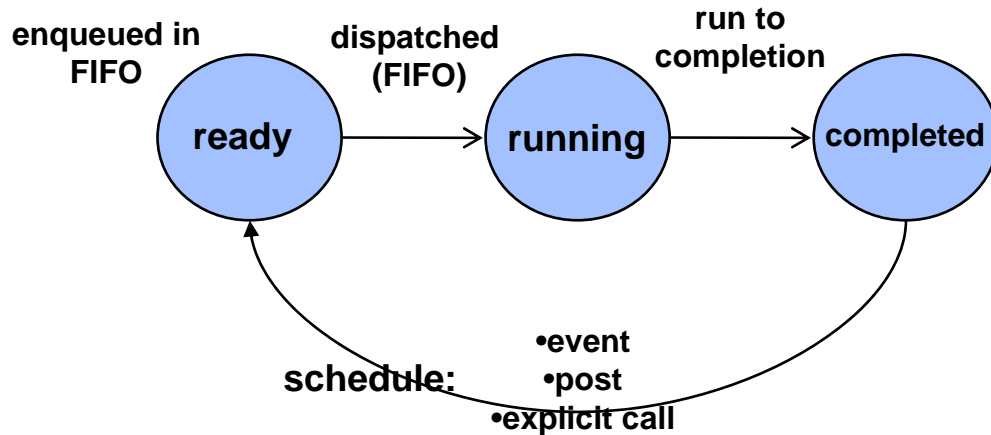
        case OFF:
            if(timer_expired(&timer)) {
                radio_on();
                state = ON;
                timer_set(&timer, T_AWAKE);
            }
            break;

        case ON:
            if(timer_expired(&timer)) {
                timer_set(&timer, T_SLEEP);
                if(!communication_complete()) {
                    state = WAITING;
                }
            }
            else {
                radio_off();
                state = OFF;
            }
        }
        break;

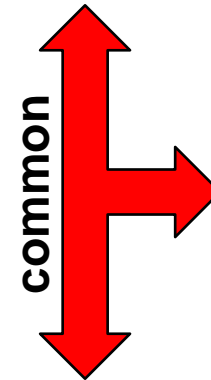
        case WAITING:
            if(communication_complete() & !timer_expired(&timer)) {
                state = ON;
                timer_set(&timer, T_AWAKE);
            }
            else {
                radio_off();
                state = OFF;
            }
        }
        break;
    }
}
```



Task Modelle

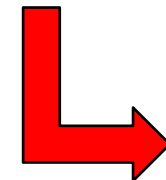


Tiny-OS



- no preemption
- event based
- single stack only

Contiki



- wait state
- global variables only
- no switch inside a wait



TinyOS Concurrency Model

Synchronous Code (SC): code that is only reachable from tasks

Asynchronous Code (AC): code that is reachable from at least one event handler

Any update to shared state in which at least one AC is involved (AC-AC or AC-SC) may result in an inconsistency.

Consistency problems are handled by "atomic" code sequences that disable interrupts. The "atomic" keyword is the only means for defining mutual exclusion (e.g. no atomic test-and-set mechanism available)



Example: Atomic section

"atomic statements are not allowed to call commands or signal events either directly or in a called function"

```
module SomeAppM { ... }
implementation {
    bool busy;
    norace uint16_t sensorReading;

    event result_t Timer.fired() {
        bool localBusy;
        atomic {
            localBusy = busy;
            busy = TRUE;
        }
        if (!localBusy)
            call ADC.getData();
        return SUCCESS;
    }

    task void sendData() { // send sensorReading
        adcPacket.data = sensorReading;
        call Send.send(&adcPacket, sizeof adcPacket.data);
        return SUCCESS;
    }

    event result_t ADC.dataReady(uint16_t data) {
        sensorReading = data;
        post sendData();
        return SUCCESS;
    }...
}
```



TinyOS

Minimales OS für ressourcenbeschränkte Systeme

Kern benötigt 400 Bytes (Code UND Daten zusammen)

Kein Echtzeitbetriebssystem (lediglich Timer Abstraktionen)

Ereignisgesteuert (Handler werden sofort ausgeführt und können sich gegenseitig unterbrechen).

"State-Machine" Ansatz (

Nicht-blockierende Funktionsaufrufe → Split Phase Ansatz (Callbacks)

Nicht-unterbrechbares Taskmodell (single stack, Tasks zueinander synchron)

Explizite Synchronisation und Compilerunterstützung bei der Erkennung und Behandlung von "Race Conditions".



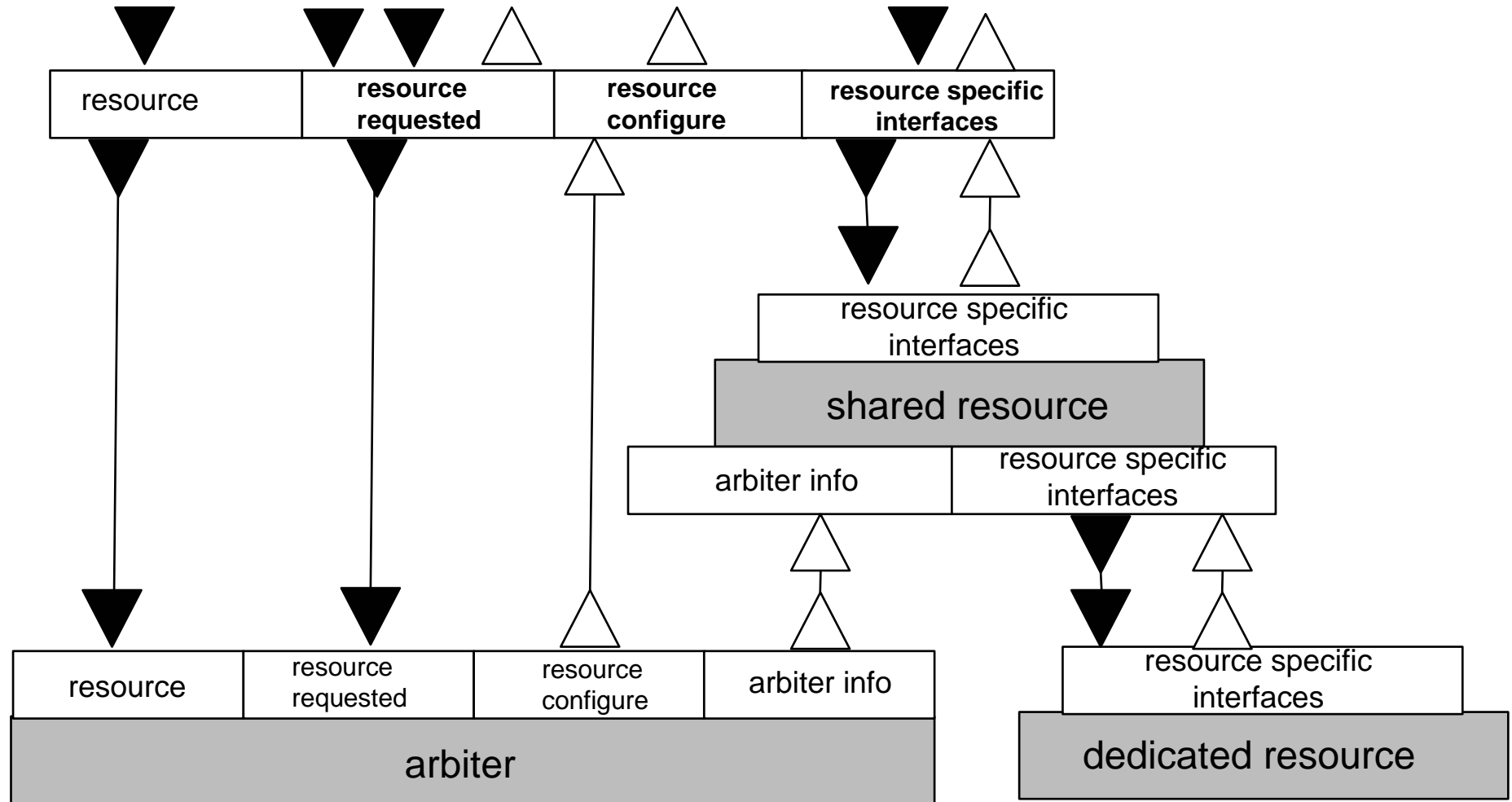
Inhalt

- ➔ **Einführung: Was ist ein eingebettetes System?
Was macht den Unterschied aus?
Was muss man können?**
- ➔ **Mikrocontroller: Architektur und Besonderheiten**
- ➔ **Sensoren, Aktoren und ihre
Unterstützung durch Funktionseinheiten
in Microcontrollern:
Analoge Schnittstellen
Zeitgeber und Zähler**
- ➔ **Zuverlässigkeit und Fehlertoleranz**
- ➔ **Zeitgerechte Ausführung**
- ➔ **Betriebssysteme für eingebettete Systeme**





Gemeinsam genutzte Ressourcen in T2





Beispiel: Nachrichtendeklaration

[Tasks]

```
DefineTask( TASK_A, EXTENDED|POOLSTACK,1, task_a,, ,POOL1 );  
DefineTask( TASK_B, BASIC|ACTIVATE|OWNSTACK,2, task_b,, ,64 );  
DefineTask( TASK_X, BASIC|ACTIVATE|NODESTACK,3, task_x,, , );
```

...

[Counters]

```
DefineCounter(Post, 24, 1);
```

...

[State Messages]

```
DefineStateMessage( msgAAst, int, sizeof(int), WithoutTimeStamp);  
DefineStateMessage( msgBAst, MSGTS, sizeof(int), WithTimeStamp);  
DefineStateMessage( msgCBst, int, sizeof(int), WithoutTimeStamp);
```

[Event Messages]

```
DefineEventMessage( msgDBev, int, sizeof(int), 6,3,WithOverwriteCheck, WithoutTimeStamp);
```



Interrupt Categories

ISR category 1

ISRs of this type does not use any operating system service. These ISR does not use OS services EnterISR and LeaveISR, consequently, ISR of these type are executed on the current stack. In this case, if ISR uses the stack space for its execution, the user is responsible for the appropriate stack size.

ISR category 2

In ISR category 2 the OSEK Operating System provides the ISR frame to execute more complicated user code. The ISR frame are instructions between OS services EnterISR and LeaveISR. Such ISRs must have the EnterISR call at their beginning to switch to the ISR stack and save the initial interrupt mask. At the end of the ISR the LeaveISR service must be executed to switch back to the task stack and restore the interrupt mask.

ISR category 3

Such ISR's are similar to those of category 2. But the location of the ISR-frame in the code segment is application dependent and user defined.



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)

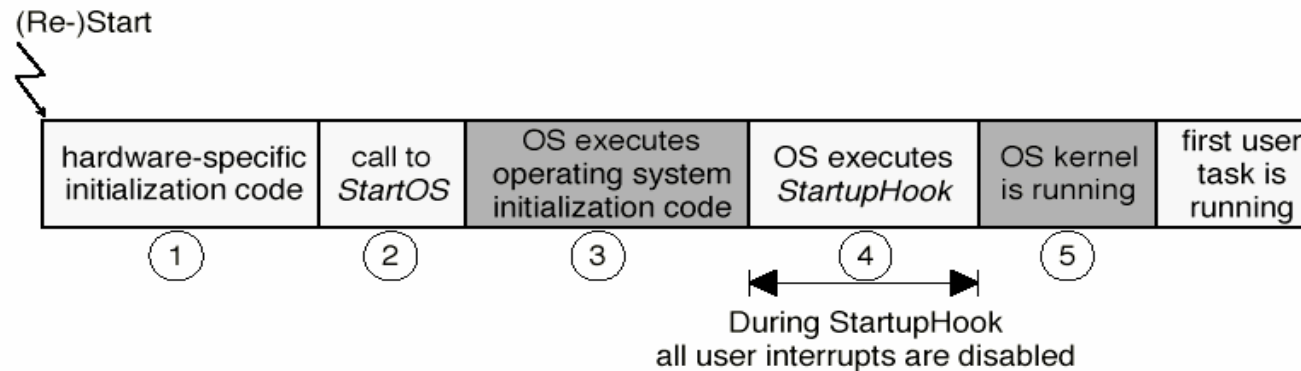
In the OSEK operating system hook routines may be used for:

- 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.



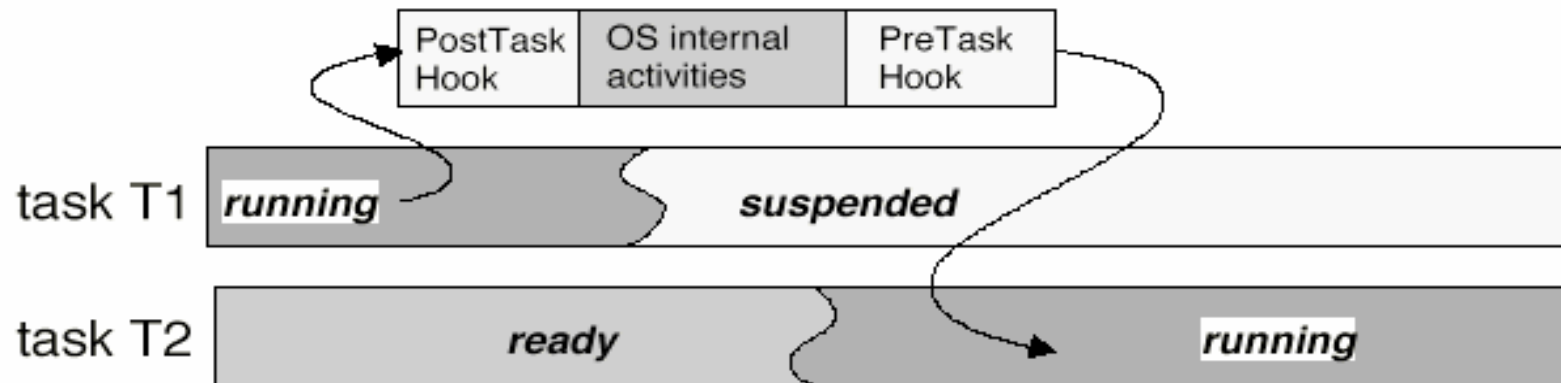
Example: StartupHook



- 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.



Counters and Alarm Generation

```
DefineCounter( <CounterID>, <maxallowedvalue>, <ticksperbase> [,<mincycle>] );
```

```
DefineSystemTimer( <CounterID>, <maxallowedvalue>, <ticksperbase>,  
                  <tickduration> [,<mincycle>] [, <HardwareType> [, <HardwareParams>]] );
```

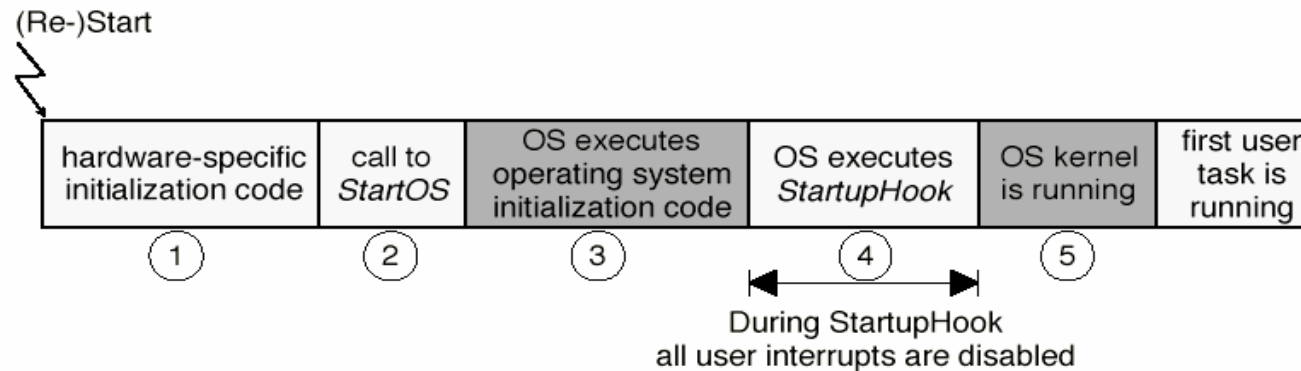
```
DefineAlarm( <AlarmID>, <CounterID>, <TaskID> [,<Event>]);
```

```
DeclareCounter( CtrRefType <CounterID> );
```

```
DeclareAlarm( AlarmRefType <AlarmID> );
```



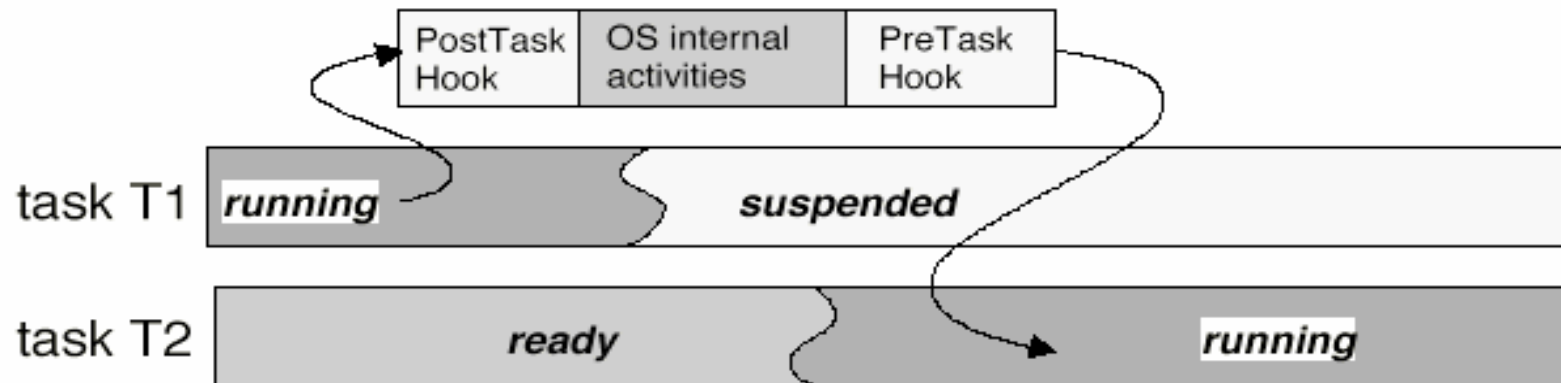
Example: StartupHook



- 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”

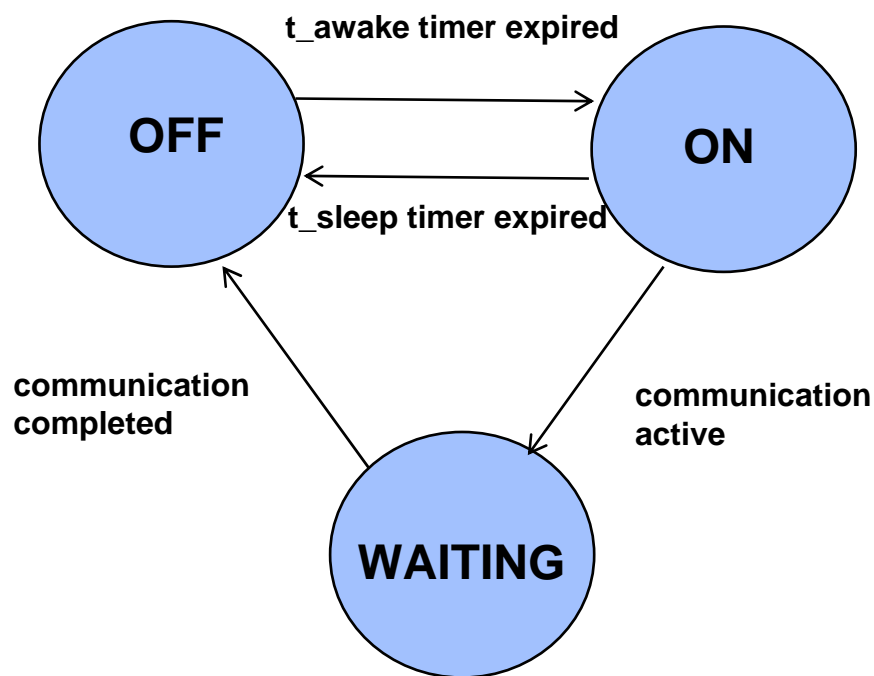


Two hook routines (PreTaskHook and PostTaskHook) are called on task context switches.

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.



State Machine Modell

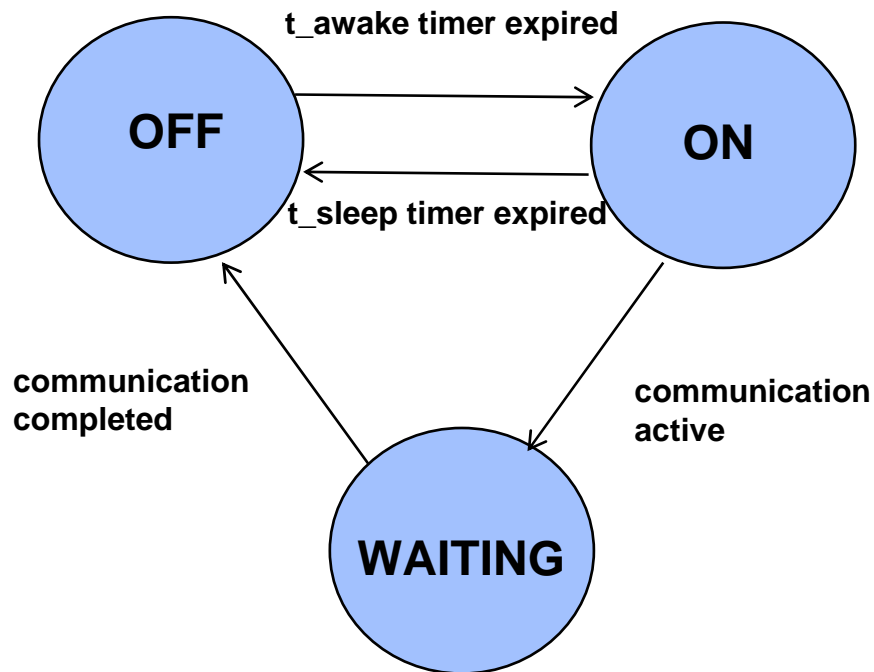


```
state: {ON, WAITING, OFF}
radio_wake_eventhandler:
  if (state = ON)
    if (expired(timer))
      timer ← t_sleep
  if (not communication_complete())
    state ← WAITING
    wait_timer ← t_wait_max
  else
    radio_off()
    state ← OFF
  elseif (state = WAITING)
    if (communication_complete() or
        expired(wait_timer))
      state ← OFF
      radio_off()
    elseif (state = OFF)
      if (expired(timer))
        radio_on()
        state ← ON
        timer ← t_awake
```

Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006), Boulder, Colorado, USA, November 2006



Eine Alternative: Protothreads (Contiki)



radio_wake_protothread:

```
PT_BEGIN
```

```
while (true)
```

```
    radio_on()
```

```
    timer ←  $t_{awake}$ 
```

```
PT_WAIT_UNTIL(expired(timer))
```

```
timer ←  $t_{sleep}$ 
```

```
if (not communication_complete())
```

```
    wait timer  $t_{wait\_max}$ 
```

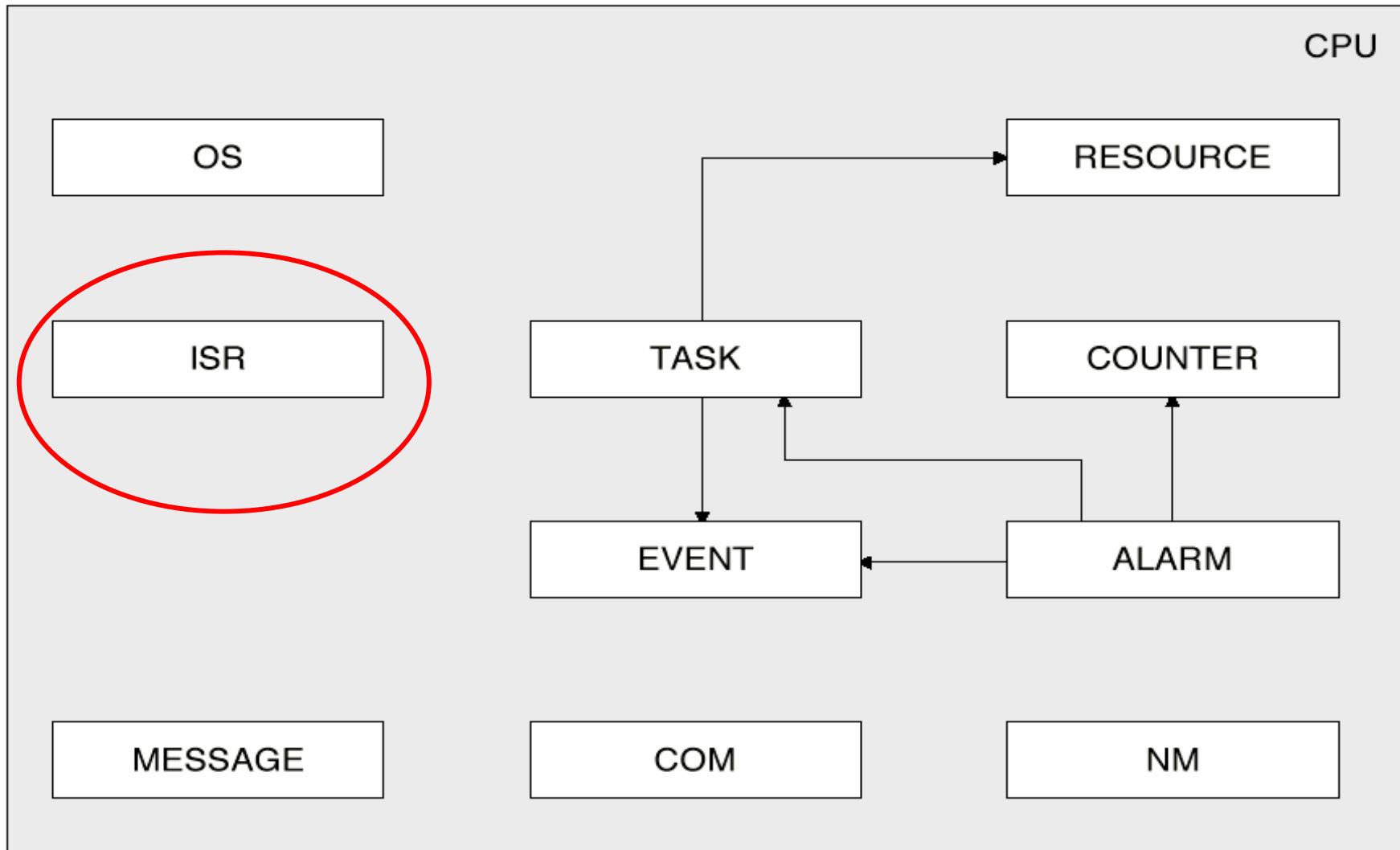
```
PT_WAIT_UNTIL(communication complete() or  
expired(wait timer))
```

```
radio off()
```

```
PT_WAIT_UNTIL(expired(timer))
```

Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006), Boulder, Colorado, USA, November 2006





OSEK Komponenten



Interrupt Management

Service Name	Description
Interrupt Management Services	
EnterISR	Registers the switching to the interrupt level and switch context to the ISR stack
LeaveISR	Registers the leaving of the ISR level
EnableInterrupt	Enables interrupts in accordance with the given mask
DisableInterrupt	Disables interrupts in accordance with the given mask
GetInterruptMask	Returns the current state of interrupts
Services allowed for use in ISR	
ActivateTask	Activates the specified task (puts it into the <i>ready</i> state)
SendStateMessage	Sends a state message to the specified task
SendEventMessage	Sends an event message to the specified task
CounterTrigger	Increments a counter value



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.

