

# Das OSEK-Betriebssystem ERCOS<sup>EK</sup>

## Inhaltsverzeichnis

<b>1</b>	<b>Grundlagen</b>	<b>2</b>
1.1	Echtzeitsysteme	2
1.2	OSEK/VDX	2
1.2.1	<i>Das OSEK Betriebssystem</i>	4
1.2.2	<i>Conformance Klassen</i>	6
1.3	ERCOS <sup>EK</sup> als Beispiel für ein OSEK-konformes Betriebssystem	8
1.3.1	<i>Tasks und Prozesse</i>	8
1.3.2	<i>Prioritäten</i>	12
1.3.3	<i>Scheduling-Strategie</i>	14
1.3.3.1	Kooperative Tasks	15
1.3.3.2	Preemptive Tasks	16
1.3.4	<i>Priority Ceiling Protokoll</i>	17
1.3.5	<i>Schutz von Variablen – Ressourcen und Messages</i>	19
1.3.6	<i>Timer</i>	20
1.3.7	<i>Startup und Shutdown des Systems</i>	22
1.4	Die Objekte der OSEK Implementation Language OIL	23
<b>2</b>	<b>Praktische Anmerkungen</b>	<b>25</b>
2.1	Einführung in die Entwicklungsumgebung ERCOS <sup>EK</sup>	25
<b>3</b>	<b>Anhang</b>	<b>28</b>
3.1	Literaturverzeichnis	28

# 1 Grundlagen

## 1.1 Echtzeitsysteme

Bei Echtzeitsystemen treten sehr hohe Anforderungen an das zeitliche Verhalten und die Korrektheit auf. Es gibt keine Standard-Definition, was man unter einem Echtzeitsystem verstehen kann.

Manche Personen verstehen unter einem Echtzeitsystem nur eine Mini-Anwendung in einem 8-Bit-Mikrokontroller mit 256 KByte ROM und 256 Byte RAM, die in Assembler "gehackt" wurde. Mittlerweile sind solche Systeme vielleicht die kleinsten zu verwirklichenden Echtzeitsysteme. Heutzutage werden Echtzeitsysteme sowohl für solche Mini-Anwendungen wie Herzschrittmacher oder Waschmaschinen eingesetzt, aber auch für größere Systeme wie die Motorsteuerung in einem Mittelklassewagen mit mehreren Prozessoren pro Steuergerät und Kommunikation über echtzeitfähige Datenbusse. Schließlich werden Echtzeitsysteme mit Echtzeitbetriebssystemen auch für hochkomplexe Steuerungen wie sie in einem Kraftwerk auftreten eingesetzt. Diese Echtzeitsysteme beinhalten viele Steuergeräte und Computer sowie Hunderte von Sensoren und Aktoren, die alle wochen- bzw. monatelang fehlerfrei funktionieren müssen.

Bruce Powel Douglass unterscheidet bei Echtzeitsystemen die zeitlichen Anforderungen in "**hard deadlines**", "**soft deadlines**" und "**firm deadlines**" [Douglass\_99]. Unter "hard deadlines" versteht er ein zeitliches Verhalten, das **unbedingt eingehalten** werden muss. Ein verspätetes Auftreten des Ergebnisses führt zu einem Fehlverhalten des Systems. Beispiele für harte Echtzeit sind Bremssysteme (ABS) oder sehr sicherheitskritische Prozesse wie die Not-Aus-Funktion in einer Maschinensteuerung. Seine Definition von "soft deadline" und "firm deadline" sind in der Fachwelt umstritten.

In der DIN 44 300 wird **Echtzeit** als **deterministisches Zeitverhalten der in einer vorgegebenen Zeitspanne verfügbaren Ereignisse** definiert. Diese Garantie ist als Standard- oder Teilfunktion jedes Echtzeitbetriebssystems anzusehen. Die DIN 44 300 beschreibt auch die weiche Echtzeit, bei der das Überschreiten von deadlines akzeptiert wird. Dabei reduziert sich die **Echtzeitfähigkeit** auf eine **garantierte maximale Reaktionszeit**.

## 1.2 OSEK/VDX

Das OSEK-Projekt wurde im Mai 1993 als ein Gemeinschaftsprojekt führender Automobilhersteller, deren Zulieferer und dem Institut für industrielle Informationstechnik der Universität Karlsruhe begonnen. OSEK steht für

**"Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug".**

1994 traten die französischen Automobilhersteller PSA und Renault dem Projekt bei und brachten ihre Erfahrungen mit ihrem Standard VDX (**Vehicle Distributed eXecutive**) in das OSEK-Projekt mit ein, das seitdem unter dem Namen OSEK/VDX auftritt. Das Ziel war, eine einheitliche produktunabhängige Architektur für die Fahrzeugelektronik zu definieren. Diese offene Architektur befasst sich mit drei Themen:

- Kommunikation (Datenaustausch zwischen Controllern)
- Netzmanagement
- Betriebssystem (Echtzeitbetrieb für ECU<sup>1</sup> Software).

Im Oktober 1997 wurde die 2. Version der Spezifikation veröffentlicht. Die aktuellsten Versionen dieser drei Bereiche wurden im Sommer 2000 verabschiedet.

Die Dokumente des OSEK/VDX-Projektes sind frei verfügbar und stehen auf dem Webserver von OSEK ([www.osek-vdx.org](http://www.osek-vdx.org)) zum Download bereit. Üblicherweise wird anstatt der Langform OSEK/VDX nur die Abkürzung OSEK verwendet.

Zur Zeit gehören folgende Firmen und Institutionen zum Technischen Komitee:

Accelerated Technology Inc.,	Mecel,
ACTIA,	Motorola,
Adam Opel AG,	National Semiconductor,
AFT GmbH,	NEC Electronics GmbH,
ATM Computer GmbH,	NRTA,
Blaupunkt,	Philips Car Systems,
BMW AG,	Porsche AG,
Borg Instruments GmbH,	PSA,
Cambridge Consultants,	Renault,
Continental Teves,	<b>Robert Bosch GmbH,</b>
Cummins Engine Company,	Sagem Electronic Division,
DaimlerChrysler AG,	Siemens Automotive,
Delco Electronics,	Softing GmbH,
Denso,	ST Microelectronics,
Epsilon GmbH,	Stenkil Systems AB,
ETAS GmbH & Co KG,	Sysgo Real-Time Solutions GmbH,
FIAT- Centro Ricerche,	TECSI,
FZI,	Telelogic GmbH,
GM Europe GmbH,	TEMIC,
Hella KG,	Texas Instruments,
Hewlett Packard France,	Thomson-CSF Detexis,
Hitachi Micro Systems Europe Ltd.,	Trialog,
Hitex,	UTA - United Technologies Automotive,
IBM Deutschland Entwicklung GmbH,	VDO Adolf Schindling GmbH,
IIIT - University of Karlsruhe,	Vector Informatik,
Infineon,	Visteon,
INRIA,	Volkswagen AG,
Integrated Systems Inc.,	Volvo Car Corporation,

<sup>1</sup> Electronic Control Unit

IRISA, LucasVarity, Magneti Marelli,	Wind River Systems, 3Soft GmbH.
--	------------------------------------

*Tabelle 1-1 Mitglieder des Technischen Komitees von OSEK*

Der Hintergrund dieses Projektes war die Tatsache, dass die Entwicklung und die Verwaltung der Variantenvielfalt von Steuergeräte-Software sehr teuer ist. Außerdem waren Steuergeräte der unterschiedlichen Hersteller zueinander inkompatibel, da sie unterschiedliche Schnittstellen und Protokolle verwendeten. Eine Software, die für ein bestimmtes Steuergerät geschrieben wurde, konnte nicht ohne weiteres auf ein anderes portiert werden.

Das Ziel von OSEK ist es, Portabilität und Wiederverwendung der Anwendungssoftware zu unterstützen. Dies soll durch die folgenden Punkte erreicht werden [osek\_os2.1]:

1. Abstrakte Schnittstellen spezifizieren, die so unabhängig wie möglich von der Applikation sind. Wie bereits oben erwähnt, erfolgt dies auf drei Gebieten:
  - Echtzeit-Betriebssystem (RTOS)
  - Kommunikation
  - Netzmanagement
2. Spezifikation von hardware- und netzwerkunabhängigen Schnittstellen
3. Effizienter Entwurf der Architektur. Das bedeutet, dass die Funktionalität **konfigurierbar** und **skalierbar** sein sollte, um die Funktionalität optimal auf die geforderte Anwendung anpassen zu können.

Diese Vorgehensweise bringt mehrere Vorteile: Zum einen werden **Kosten und Entwicklungszeit gespart**. Außerdem wird die **Qualität der Software** von Steuergeräten unterschiedlicher Firmen verbessert. Zuletzt bietet der Standard **Unabhängigkeit** für die einzelnen Implementierungen, da die Spezifikation nicht Aspekte der Implementierung beschreibt.<sup>2</sup>

### 1.2.1 Das OSEK Betriebssystem

Die Spezifikation des OSEK Betriebssystems [osek\_os2.1] stellt eine einheitliche Umgebung dar, welche die effiziente Ausnutzung der Ressourcen von Steuergeräte-Software im Kraftfahrzeug unterstützt. Das Betriebssystem ist für Geräte mit einem Prozessor vorgesehen. Außerdem unterstützt es verteilte Anwendungen<sup>3</sup> von eingebetteten Steuergeräten.

Gerade im Kraftfahrzeug treten sehr hohe Anforderungen an ein Betriebssystem auf: Dies sind zum einen Anforderungen an das Echtzeitverhalten - man stelle sich nur die Auswirkungen eines zu spät reagierenden Airbags oder falsches Timing beim ABS vor -

<sup>2</sup> Die Spezifikation ist nur die theoretische Beschreibung des Standards. Implementiert wird der Standard in kommerzielle Produkte, beispielsweise in das Betriebssystem ERCOS<sup>EK</sup> von ETAS oder in ProOSEK von 3Soft.

<sup>3</sup> Die Steuergeräte kommunizieren untereinander über festgelegte Protokolle, zum Beispiel den CAN-Bus

zum anderen soll das Betriebssystem auch auf Prozessoren und Mikrocontrollern mit wenig Speicher (ROM und RAM) laufen.

Von OSEK selbst existiert keine Implementierung, OSEK gibt nur die Spezifikation vor. Implementiert wird das Betriebssystem von kommerziellen Anbietern wie ETAS, 3Soft oder Sysgo Real-Time Solutions GmbH.

Das gleiche Betriebssystem kann mit unterschiedlicher Konfiguration sowohl auf einem mächtigen Steuergerät als auch in einem Billig-Controller arbeiten. OSEK-Betriebssysteme können daher in "Conformance Classes" (Anpassungs-Klassen) eingeteilt werden. Auf diese Klassen wird in Kapitel 1.2.2 ausführlich eingegangen.

Obwohl die Implementierung eines Betriebssystems abhängig vom eingesetzten Prozessor ist, bietet der OSEK-Standard prozessorunabhängige Schnittstellen zur Anwendungssoftware. Somit kann eine Anwendung wie zum Beispiel die Steuerung eines Blinkers oder eines Radios unabhängig vom verwendeten Prozessor geschrieben werden und wird dann mit unterschiedlichen Compilern für die einzelnen Prozessoren übersetzt.

Diese Unabhängigkeit der Anwendungssoftware ist mit der Portierung des Betriebssystems Linux vergleichbar, welches man mittlerweile auf vielen unterschiedlichen Plattformen, zum Beispiel PC, S/390 oder RS/6000 zum Laufen gebracht hat. Der Quellcode einer Anwendung, zum Beispiel der Apache Webserver, kann hier auf den einzelnen Systeme kompiliert werden und muss nur einmal geschrieben werden.

Ein Ziel von OSEK ist es, portable und wiederverwendbare Software zu ermöglichen. Dies wird durch eine spezifizierte Anwendungsschnittstelle (API) ermöglicht, über die Befehle an den Mikrocontroller gesendet werden. Beim Wechsel des Controllers sorgt das Betriebssystem für die richtige Umsetzung des Programms auf den Mikrocontroller.

Auch wenn das OSEK Betriebssystem als Zwischenschicht zwischen Prozessor und Anwendungssoftware liegt, ist es möglich, direkt die Schnittstellen eines Mikrocontrollers anzusprechen. Hardwaretreiber nutzen beispielsweise diese Möglichkeit. Bei einer Portierung auf einen anderen Chip muss dieser Teil der Anwendung jedoch an den neuen Controller angepasst werden.

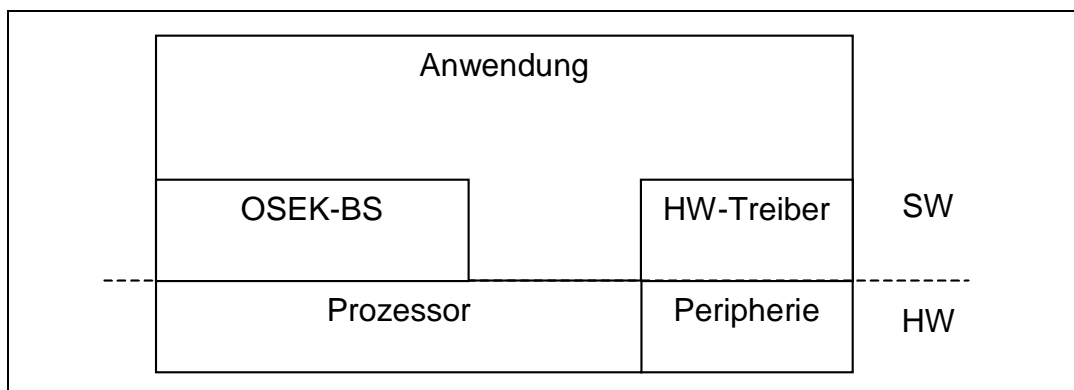


Abbildung 1-1 Schichten im Mikrocontroller7

Im Unterschied zu Unix liegen die Hardwaretreiber in Abbildung 1-1 nicht unterhalb des Betriebssystems, da das OSEK-Betriebssystem nicht auf die Treiber zugreift.

Aufgrund der hohen Anforderungen an Zuverlässigkeit, Echtzeitfähigkeit und Kosten-Nutzen-Verhältnis bildeten sich folgende Merkmale heraus:

- Eine Anwendung, die auf dem OSEK BS aufbaut, muss statisch konfiguriert und skaliert sein. Das bedeutet, dass die Anzahl der Tasks, Ressourcen und Dienste schon zur Compilezeit durch den Benutzer eingestellt wird. Die einzelnen Objekte des Betriebssystems werden später noch genauer beschrieben.
- Die Spezifikation des OSEK BS unterstützt die Ausführung der Software im ROM<sup>4</sup>, das heißt, der Code kann direkt vom ROM aus ausgeführt werden
- Die Spezifikation des OSEK BS beschreibt ein vorhersehbares und dokumentiertes Verhalten für Echtzeit-Anforderungen im Kraftfahrzeug, um entsprechende Betriebssystem-Implementationen zu ermöglichen.
- Die Spezifikation des OSEK BS erlaubt die Implementation von vorhersehbaren Leistungsparametern. Durch die Conformance Klassen werden Mindestanforderungen festgelegt, beispielsweise die minimale Anzahl von Tasks oder Ressourcen.

### 1.2.2 Conformance Klassen

Wie schon oben beschrieben gibt es keine "richtige" OSEK-Betriebssystemimplementierung. Je nach unterstützter Funktionalität werden die Betriebssystemimplementierungen in unterschiedliche Klassen eingeteilt. Daraus ergeben sich vier "Conformance Classes". Deren Leistungsumfang hängt von der verfügbaren Performance des Mikrocontrollers (z.B. CPU, Speicher) ab.

Diese Conformance Klassen helfen, Implementierungs-Ebenen des Betriebssystems zu klassifizieren. Teil-Implementierungen des gesamten Standards sind an bestimmten Punkten möglich und können als OSEK-konform zertifiziert werden. Außerdem wird ein Weg für die Umwandlung leistungsärmerer Klassen zu Klassen mit mehr Leistungsumfang dargestellt, mit dem man die Implementierung des Betriebssystems ändern kann, ohne die darüberliegende Anwendung zu ändern. Es ist also definiert, welche Conformance Klasse in welche andere überführt werden kann, wenn zusätzliche Leistungsmerkmale implementiert werden.

Folgende vier Klassen sind definiert:

---

<sup>4</sup> Read Only Memory: Nur-Lese-Speicher, das Programm ist fest in den Mikrocontroller eingebrannt.

BCC1 (Basic Conformance Class 1)	<ul style="list-style-type: none"> <li>• Nur einfache Tasks</li> <li>• Nur eine Aktivierung<sup>5</sup> pro Task erlaubt</li> <li>• Nur eine Task pro Prioritätslevel</li> </ul>
BCC2 (Basic Conformance Class 2)	<ul style="list-style-type: none"> <li>• Nur einfache Tasks:</li> <li>• Mehrfache Aktivierung pro Task erlaubt</li> <li>• Mehr als eine Task pro Priorität möglich</li> </ul>
ECC1 (Extended Conf. Class 1)	<ul style="list-style-type: none"> <li>• einfache und "extended" Tasks</li> <li>• Nur eine Aktivierung pro Task erlaubt</li> <li>• Nur ein Task pro Prioritätslevel</li> </ul>
ECC2 (Extended Conf. Class 2)	<ul style="list-style-type: none"> <li>• einfache und "extended" Tasks</li> <li>• Mehrfache Aktivierung pro Task erlaubt; extended Tasks dürfen nur einfach aktiviert werden</li> <li>• Mehr als ein Task pro Prioritätslevel möglich</li> </ul>

Im Gegensatz zu einfachen Tasks haben "extended" Tasks noch den zusätzlichen Modus "waiting" oder "blocked". Dieser Modus tritt auf, wenn die Task auf ein äußeres Ereignis warten muss und deshalb die CPU abgibt.

Einfachere Implementierungen, beispielsweise ERCOS<sup>EK</sup> haben nur einfache Tasks. Auf die Zustände von Tasks wird nochmals ausführlicher in Kapitel 1.3.1 eingegangen.

Zwischen den Conformance Classes gibt es Kompatibilitäts-Pfade, es lässt sich also beschreiben, welche Klasse eine andere enthält:

BCC1  $\in$  BCC2  $\in$  ECC2       $\in$  = "ist Element von"

BCC1  $\in$  ECC1  $\in$  ECC2

BCC2 und ECC1 sind nicht pauschal vergleichbar.

Abbildung 1-2 stellt die von OSEK geforderten minimalen Anforderungen an die definierten Conformance Klassen dar.

---

<sup>5</sup> Unter Aktivierung einer Task versteht man einen Zustandsübergang in den Zustand ready (Siehe auch Seite 10)

	BCC1	BCC2	ECC1	ECC2
Mehrfache Taskaktivierung	-	+	-	+
Anzahl der Tasks, die nicht im Zustand <i>suspended</i> sind.	8		16 (Gesamtzahl von Basic Tasks und Extended Tasks)	
Mehr als eine Task pro Priorität	-	+	-	+
Anzahl Events pro Task	-		8	
Anzahl der Task-Prioritäten	8			
Ressourcen	RES_SCHEDULER		8(inklusive RES_SCHEDULER)	
Anzahl Alarme	1			
Anzahl der Application Modes	1			

Abbildung 1-2 Minimale Anforderungen an die Conformance Klassen

### 1.3 ERCOS<sup>EK</sup> als Beispiel für ein OSEK-konformes Betriebssystem

Wie schon oben erwähnt, ist OSEK nur der Standard, nicht die Implementierung. Als Implementierung des Standards wird bei dieser Diplomarbeit das Betriebssystem ERCOS<sup>EK</sup> von ETAS verwendet. ERCOS<sup>EK</sup> implementiert die Conformance-Klasse BCC2 und somit auch BCC1 des OSEK-Standards - aber leider nicht vollständig.

Anhand dieses Betriebssystems sollen die wichtigsten Merkmale von OSEK, der Sprachumfang von OIL<sup>6</sup>, beschrieben werden. Dabei soll auch der grundsätzliche Unterschied zwischen einem Echtzeitbetriebssystem und einem Betriebssystem für Informationssysteme, beispielsweise Unix herausgestellt werden. Es wird jedoch nicht der gesamte Umfang des OSEK-Betriebssystems beschrieben. Eine ausführliche Einführung in das OSEK-Betriebssystem findet man in der OSEK-Definition [osek\_os2.1]. In der OSEK Implementation Language (OIL) sind die Objekte der Sprache beschreiben. Dazu gehören beispielsweise Tasks, Events, Counter oder das OS als Ganzes. Die einzelnen Objekte und ihre Beziehungen sind in Kapitel 1.4 beschrieben.

#### 1.3.1 Tasks und Prozesse

Im Gegensatz zu DOS- oder Unix-Betriebssystemen werden bei ERCOS<sup>EK</sup> und OSEK Tasks und Prozesse anders definiert.

Bei PC-Betriebssystemen stellt ein **Prozess** ein Programm dar, das in einem **eigenen Speicherbereich** relativ unabhängig von den anderen Prozessen ausgeführt wird. Prozesse werden auch "heavy weight processes" genannt. Bei Multitasking-Betriebssystemen können mehrere Prozesse nacheinander quasiparallel – bei Multiprozessormaschinen auch parallel – ausgeführt werden. Ein Prozess kann aus mehreren **Threads** aufgebaut sein. Alle Threads eines Prozesses laufen im **gleichen**

<sup>6</sup> OSEK Implementation Language

**Speicherbereich.** Threads haben beispielsweise die Aufgabe, auf Eingabeereignisse von Tastatur oder Maus zu reagieren, während parallel eine zeitintensive Berechnung durchgeführt wird. Durch mehrere Threads ist es beispielsweise möglich, auch während des Drucks eines Dokuments im selben Prozess weiterzuschreiben. Die Threads werden meistens zyklisch abgearbeitet. Dieses Verfahren wird im nächsten Kapitel bei Scheduling angesprochen.

In Abbildung 1-3 könnte Thread1 beispielsweise für die Verarbeitung der Oberflächeneingaben (Tastatur, Maus) zuständig sein. Thread2 speichert das Dokument, während parallel weitergearbeitet wird. Thread3 könnte dann eine zeitintensive Berechnung durchführen, während Thread4 das Dokument druckt.

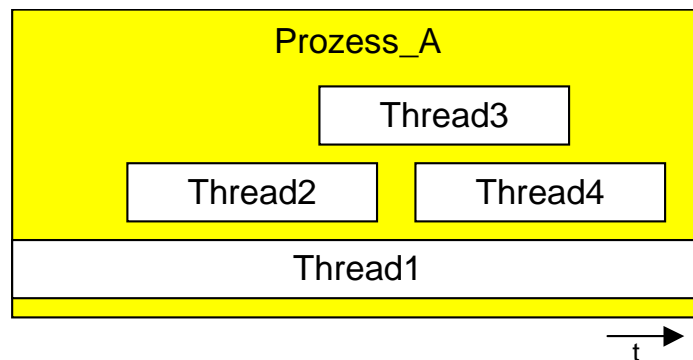


Abbildung 1-3 Mehrere quasiparallel laufende Threads in einem Prozess

Bei **OSEK** wird ein monolithischer Kernel mit einem Programm aus Betriebssystem und Anwendung generiert. Die Anwendung wird in mehrere **Tasks** eingeteilt, die abhängig von ihrer **Priorität** unterschiedlich schnell nach ihrer Aktivierung ausgeführt werden. Diese Tasks können sich je nach Priorität unterbrechen. Auf die Aufruffolge von Tasks durch den Scheduler wird in Kapitel 1.3.3 eingegangen.

In einer Task definiert **ERCOS<sup>EK</sup>** mehrere Prozesse. Ein **Prozess** ist eine **Folge von Anweisungen, die je nach Scheduling-Strategie unterbrechbar oder nicht** sein soll. Dies wird in Kapitel 1.3.3 näher beschrieben. Alle Prozesse einer Task werden **sequentiell** ausgeführt.

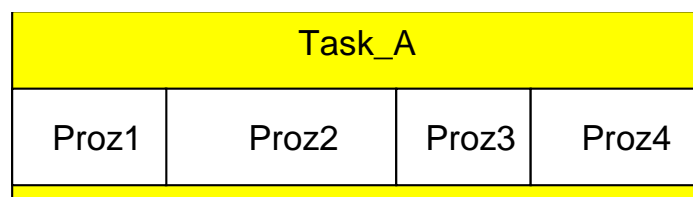


Abbildung 1-4 Tasks und Prozesse bei OSEK/ERCOS<sup>EK</sup>

Tasks können mehrere Zustände haben. Je nach Implementierungsstufe (Conformance Klasse) gibt es drei oder vier Zustände bei den Tasks:

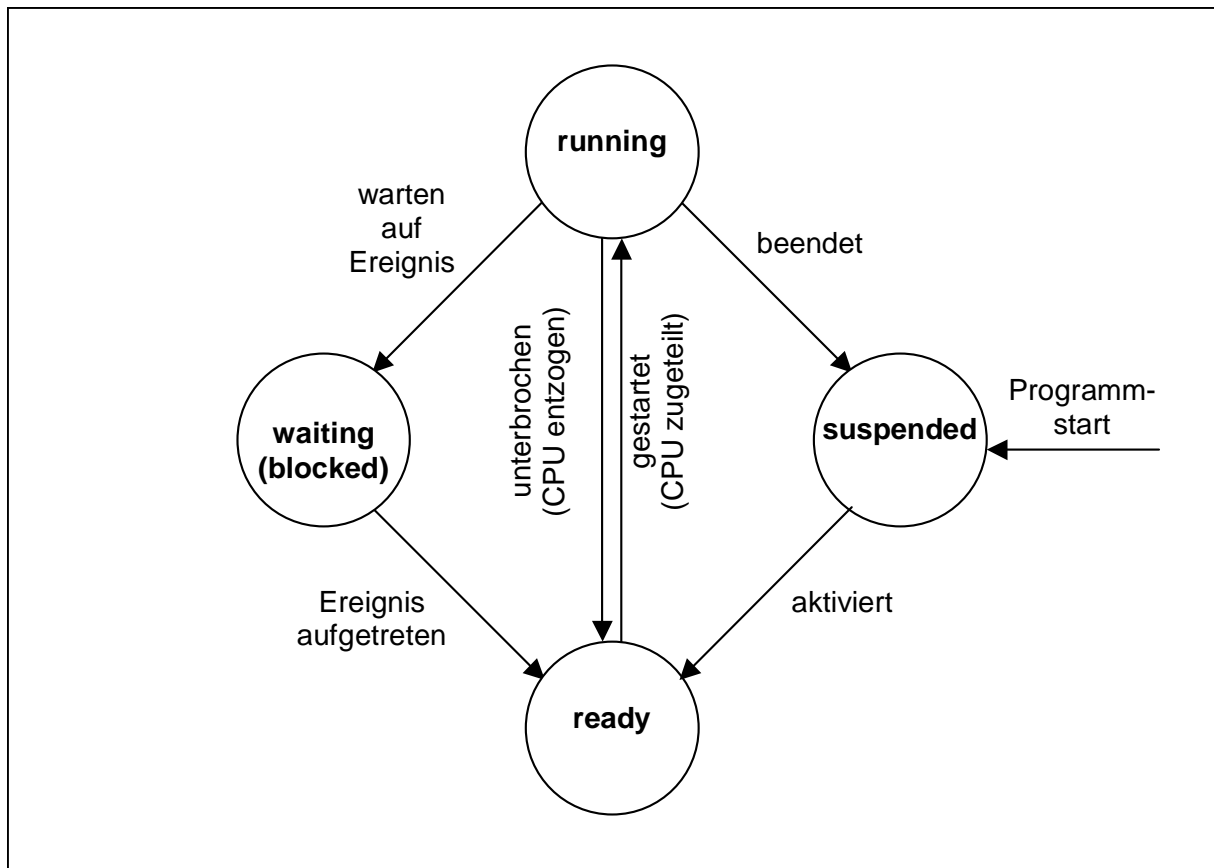


Abbildung 1-5 Zustände und Übergänge mit erweiterten Tasks

Beim Programmstart wird der Code aller Tasks geladen, die Tasks befinden sich im **Ruhezustand**, auf englisch **suspended**. Die Startfunktion `main()` läuft außerhalb des Betriebssystems als einziger Prozess.

Hat eine Task die CPU und kann „arbeiten“, so befindet sie sich im Zustand **running**. Nur in diesem Zustand wird das Programm abgearbeitet. Im Gegensatz zu den anderen Zuständen kann sich hier nur ein einziger Prozess im Zustand **running** befinden.

Wenn eine Task alle Voraussetzungen für den Zustand **running** hat und **bereit** ist, gestartet zu werden, der Prozessor jedoch durch eine andere Task belegt ist, wird die Task in den Zustand **ready** überführt. Der **Scheduler** entscheidet, welche Task nach dem Freiwerden der CPU als nächstes ausgeführt wird.

Bei den Conformance Klassen ECC1 und ECC2 gibt es noch den zusätzlichen Zustand **waiting**<sup>7</sup>. Extended Tasks können in diesen Zustand wechseln, wenn sie den Prozessor haben, aber nicht weiterarbeiten können, weil sie **auf ein Ereignis warten** müssen. Dies kann beispielsweise auftreten, wenn der Controller eine CAN-Nachricht verschickt und auf die Bestätigung warten muss.

Bei Betriebssystemen für Informationssysteme tritt der Zustand **blocked** oft auf, wenn auf I/O-Komponenten, zum Beispiel Festplattencontroller zugegriffen wird.

<sup>7</sup> Der Zustand **waiting** wird auch **blocked** bezeichnet.

Im Gegensatz zu Unix gibt es bei OSEK weniger Prozesszustände. Da das Betriebssystem statisch konfiguriert wird, ist beim Programmstart schon der gesamte Code geladen.

Bei Systemen mit einfachen Tasks gibt es keinen **waiting** Zustand. Dadurch fällt die linke Hälfte des Graphen weg. Muss eine Task auf ein bestimmtes Ereignis warten, kann es dessen Quelle pollend abfragen. Ein Übergang in den Zustand ready kann nur über eine explizite Taskaktivierung durch einen Prozess einer anderen Task erfolgen.

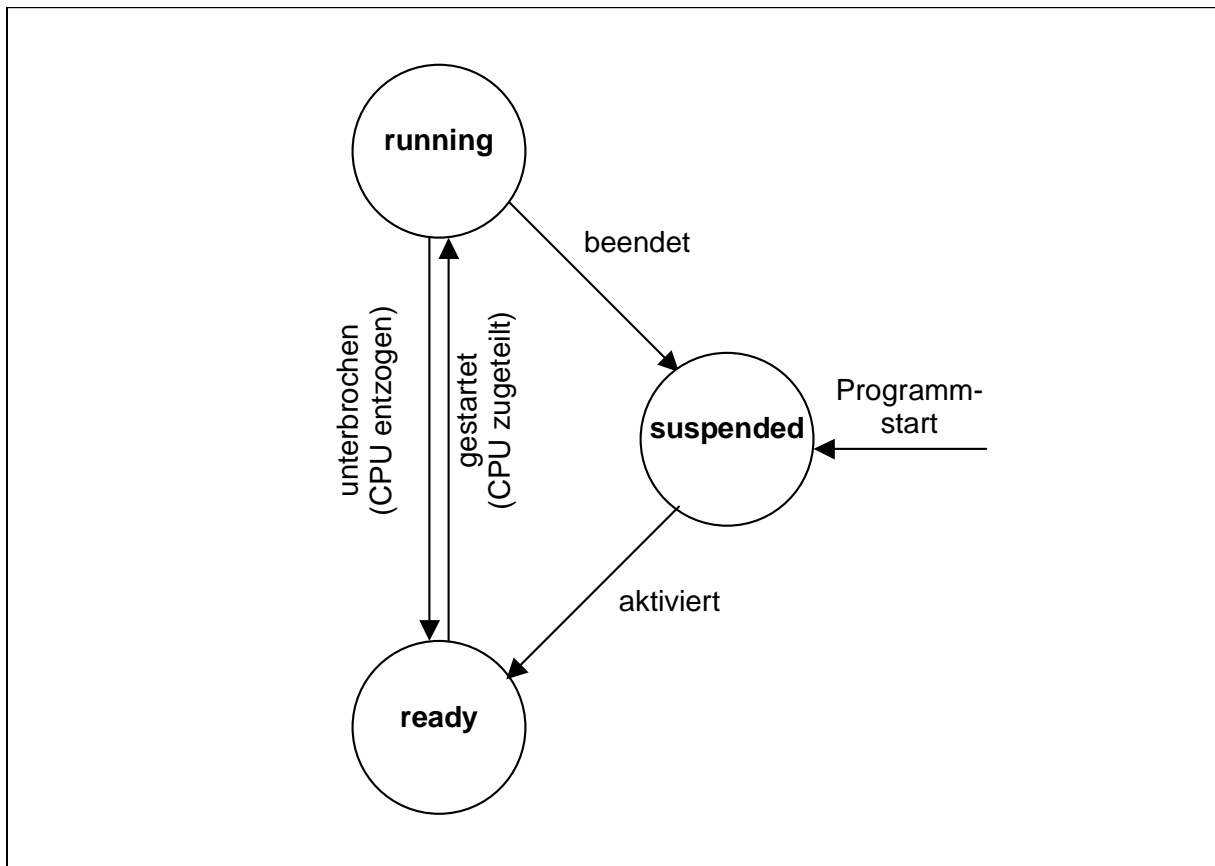


Abbildung 1-6 Zustände und Übergänge bei einfachen Tasks

Wird das Programm (Betriebssystem und Anwendung) **gestartet**, so treten alle Tasks in den Zustand **suspended**. Über das Programm werden die einzelnen Tasks **aktiviert** und treten in den **Bereit-Zustand (ready)**. Die Prozesse im ready-Zustand werden nacheinander vom **Scheduler gestartet** und gelangen damit in den Zustand **running**. Die dahinter liegende Prozedur wird im nächsten Kapitel behandelt. Ein **laufender Prozess** wird so lange ausgeführt, bis er sich **selbst beendet**, vom **Scheduler unterbrochen** wird oder – bei erweiterten Tasks – **sich selbst blockiert** und in den Wartezustand übertritt. Bei OSEK gibt es im Gegensatz zu Unix keine Zeitscheiben, es wird also nicht bestimmt, wie lange ein Prozess maximal im Zustand running verbleiben darf.

**Wartende Tasks** werden durch **Ereignisse geweckt** und gehen dann in den Pool der bereiten Tasks über, in dem sie dann entsprechend ihrer Priorität vom Scheduler gestartet werden.

### 1.3.2 Prioritäten

Im Echtzeitbereich hat jede Task eine bestimmte Priorität. Diese Priorität wird statisch vergeben, kann also während der Ausführung nicht dynamisch verändert werden. Die Priorität legt fest, wie wichtig die Ausführung der in der Task enthaltenen Prozesse ist. Außerdem wird über die Priorität gesteuert, in wieweit sich die Tasks untereinander unterbrechen. Eine Task darf nur von einer anderen Task höherer Priorität unterbrochen werden.

Bei ERCOS<sup>EK</sup> sind drei Gruppen von Tasks definiert, die in vier **Prioritätsgruppen** aufgeteilt werden. In jeder Prioritätsgruppe können mehrere Prioritäten vereinbart werden. Die maximale Anzahl der Prioritäten ist prozessorabhängig.

Die Grenzwerte für die einzelnen Gruppen werden bei ERCOS<sup>EK</sup> durch Attribute in der Struktur OS bestimmt. Die Attribute für die Bereichsgrenzen sind:

MAX_ISR_PRIO	höchste Priorität für Interrupts
MIN_ISR_PRIO	niedrigste Priorität für Interrupts
MAX_TASK_PRIO	höchste Priorität für Softwaretasks
MIN_PREEMP_PRIO	niedrigste Priorität für preemptive Tasks

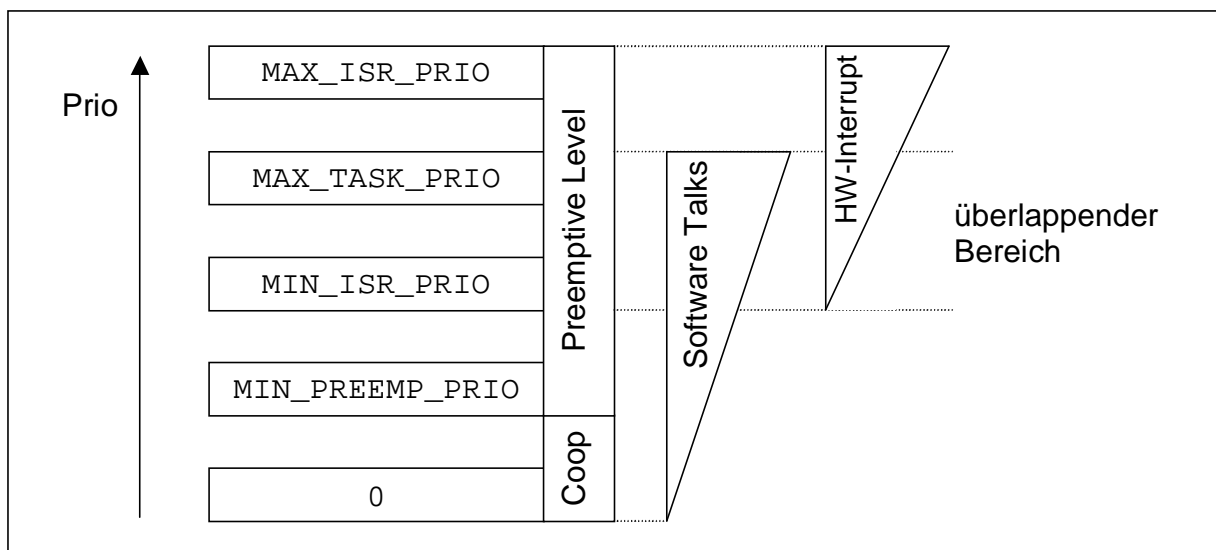


Abbildung 1-7 Prioritätsgruppen bei ERCOS<sup>EK</sup>

ERCOS<sup>EK</sup> versteht unter **Tasks** sowohl reine **Softwaretasks** – also Programmaufrufe mit festgelegter Priorität – als auch **Interrupts**, die hier als **Hardwaretasks** bezeichnet werden. Die Softwaretasks werden noch einmal in kooperative Tasks und in preemptive Tasks aufgeteilt.

**Kooperative Tasks** haben die **geringste Priorität**. Sie werden für Hintergrundprozesse und „unwichtige“ Aufgaben eingesetzt. Als nächstes gibt es einen Bereich für preemptive Softwaretasks. In dieser Gruppe sind die meisten preemptiven Tasks definiert. Preemptive Tasks können andere preemptive Tasks niedrigerer Priorität und alle kooperativen Tasks unterbrechen, da die kooperativen Tasks eine niedrigere Priorität als die preemptiven besitzen.

Die nächst höherpriorie Gruppe ist sowohl für Interrupts (Hardwaretasks) als auch für Softwaretasks reserviert. In dieser Gruppen können höherpriorie Softwaretasks niederpriorere Interrupts unterbrechen! Diese Gruppe ist eine **Erweiterung von ERCOS<sup>EK</sup>** und ist **nicht in OSEK definiert**. In OSEK überschneiden sich Softwaretasks und Interrupts nicht.

Zuletzt gibt es eine Gruppe für reine Interrupts. Bei einem Interrupt sollen kurze Anweisungsfolgen sehr schnell verarbeitet werden. Interrupts werden durch äußere Ereignisse aktiviert.

Durch den Präprozessor werden bei ERCOS<sup>EK</sup> die Prioritäten auf Interrupttabellen abgebildet. Grundsätzlich dürfen mehr Prioritätslevel verwendet werden als Interrupts verfügbar sind. Bei Prioritäten mit reinen Softwaretasks werden dabei mehrere Ebenen auf einen Interrupt abgebildet. So haben alle kooperativen Ebenen den Interrupt 0 (keinen) und alle preemptiven Softwaretasks den gleichen Level. Das Betriebssystem kümmert sich mit anderen Mitteln um die richtige Taskaktivierung. Mit dieser Technik ist es beispielsweise möglich, bei einem Prozessor mit 8 Interruptleveln 15 Prioritäten unterzubringen.

Die Struktur ist in der Datei `<projektpfad>\source\<target>\os_setup.h` definiert. Das folgende Beispiel zeigt einen Ausschnitt dieser Struktur:

```
OS {  
    ...  
    MAX_ISR_PRIO      = 15;  
    MIN_ISR_PRIO      = 10;  
    MAX_TASK_PRIO     = 13;  
    MIN_PREEMP_PRIO   = 5;  
};
```

Insgesamt gibt es hier 15 Prioritätsebenen. 0 bis 4 sind kooperative Softwaretasks, preemptive Softwaretasks sind zwischen 5 und 13 und Interrupts können auf den Prioritätsebenen 10 bis 15 agieren. Die Prioritätsebenen 10 bis 13 teilen sich Softwaretasks und Hardwaretasks.

Die durch den Präprozessor erzeugte C-Datei sieht dann folgendermaßen aus:

```

...
/* search table: SW priorities for interrupt masks
*****/
#pragma ghs section rozdata = ".erc_con"
FAR const uint _intMaskTab[] = {
0x00,      /* Priority 0      (S/W coop) */
0x00,      /* Priority 1      (S/W coop) */
0x00,      /* Priority 2      (S/W coop) */
0x00,      /* Priority 3      (S/W coop) */
0x00,      /* Priority 4      (S/W coop) */
0x87,      /* Priority 5      (pure S/W preemp) */
0x87,      /* Priority 6      (pure S/W preemp) */
0x87,      /* Priority 7      (pure S/W preemp) */
0x87,      /* Priority 8      (pure S/W preemp) */
0x87,      /* Priority 9      (pure S/W preemp) */
0x86,      /* Priority 10     (overlapping S/W, H/W) */
0x85,      /* Priority 11     (overlapping S/W, H/W) */
0x84,      /* Priority 12     (overlapping S/W, H/W) */
0x83,      /* Priority 13     (overlapping S/W, H/W) */
0x82,      /* Priority 14     (pure H/W) */
0x81,      /* Priority 15     (pure H/W) */
};
...

```

### Quellcode 1-1 Tasklevel werden auf Interrupts abgebildet

Die kooperativen Prioritätsebenen 0 bis 4 bekommen alle den gleichen Wert in der Interruptmask-Tabelle. Die reinen preemptiven Softwaretasks der Ebene 5 bis 9 sind haben ebenfalls einen gemeinsamem Wert 0x87 in der Tabelle. Die Prioritätsebenen 10 bis 13 teilen sich Softwaretasks und Interrupts. In dieser Erweiterung von ERCS<sup>EK</sup> könnte eine Task auf Ebene 13 einen Interrupt auf Ebene 12 unterbrechen! Die Prioritätsebenen 14 und 15 sind für Interrupts reserviert. Das Betriebssystem selbst läuft auf der höchsten Taskebene, hier auf Prioritätslevel 13.

### 1.3.3 Scheduling-Strategie

Wie schon oben besprochen, gibt es Probleme, wenn **mehrere Tasks gleichzeitig ausgeführt** werden wollen. PC-Betriebssysteme unterscheiden sich von Echtzeit-Betriebssystemen in der Art und Weise, wie die Tasks nacheinander abgearbeitet werden. Während bei einem Informationssystem alle Prozesse gleichmäßig bedient werden wollen, gibt es im Echtzeitbereich Prioritäten, wie Tasks im ready-Zustand abgearbeitet werden. Es ist offensichtlich, dass im Automobilbereich mehr und weniger kritische Anwendungen auftreten: Bei einem Unfall soll der Airbag nicht durch die Ansteuerung des Scheibenwischers verzögert werden!

Bei Informationssystemen bekommen gleichpriore Tasks die CPU zu gleichen Teilen zugeteilt. Beim Round-Robin-Verfahren veranlasst der Scheduler alle n Millisekunden einen Taskwechsel. Die Bearbeitungszeit ist einstellbar, zum Beispiel 100ms. Zu beachten ist, dass der Scheduler auch Zeit für das Umschalten braucht (1 - 5 ms). Wählt man die Schedulingzeit zu kurz, so ist der Wirkungsgrad schlecht, wählt man sie zu lang, so ergeben sich zu lange Wartezeiten für kurze Transaktionen.

Bei Echtzeitbetriebssystemen ist die Scheduling-Strategie anders: Die wartenden Tasks jeder Priorität werden in FIFO-Queues zwischengespeichert. Wird nun eine Task beendet und eine neue Task soll in den running-Status überführt werden, so nimmt der Scheduler die erste Task aus der Queue mit der höchsten Priorität und führt diese aus.

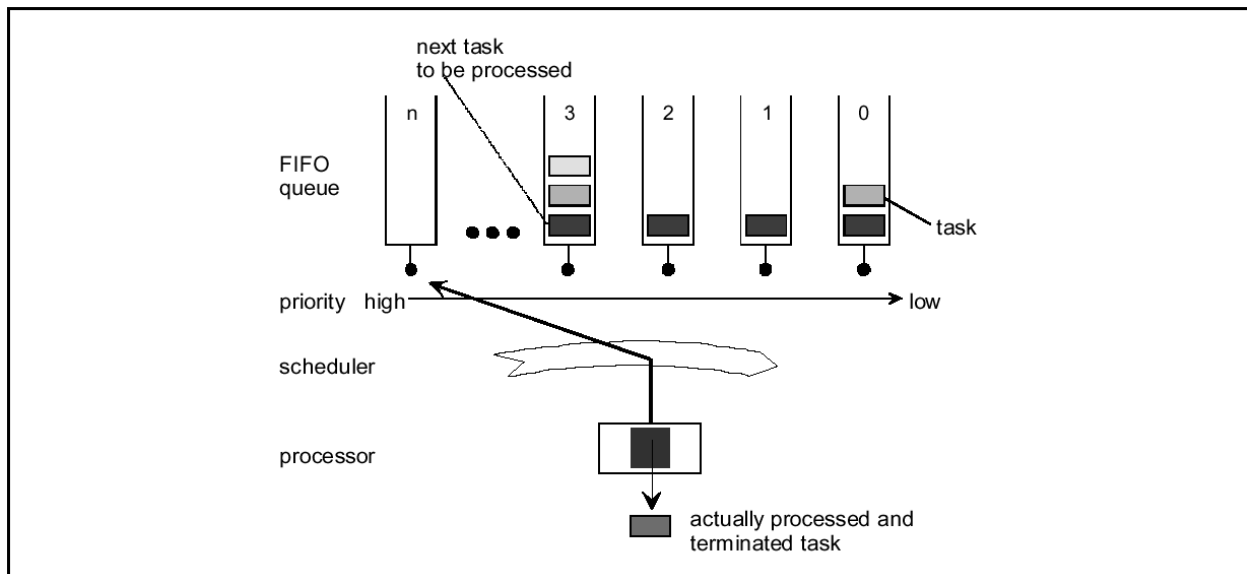


Abbildung 1-8 Scheduling bei OSEK (Quelle: OSEK)

Interessanter wird es, wenn eine Task neu aktiviert wird und in den ready-Zustand kommt. Wenn die Priorität der neuen Task höher als die der ausgeführten ist, muss die laufende Task unterbrochen werden und die neue wird gestartet. Je nach **Zeitpunkt der Unterbrechung** unterscheidet man **kooperative Tasks** und **preemptive Tasks**.

### 1.3.3.1 Kooperative Tasks

Laufende kooperative Tasks können von höherprioritäre kooperativen Tasks nur an bestimmten Stellen unterbrochen werden. **OSEK** definiert folgende „**Points of rescheduling**“ [OSEK\_os2.1]:

- Erfolgreiche **Beendigung** einer Task.
- Erfolgreiche **Beendigung** einer Task mit der **expliziten Aktivierung einer Nachfolgetask**.
- Expliziter **Aufruf des Schedulers** mit der Funktion `Schedule()`.
- Ein **Wechsel** in den **waiting**-Zustand findet statt. (Nur bei Implementierungen, welche die **ECC-Conformance Klassen** unterstützen)

Bei **ERCOS<sup>EK</sup>** wird zusätzlich innerhalb einer Task bei **Prozessgrenzen** der Scheduler aufgerufen.

Bei Kooperativen Tasks können somit größere Wartezeiten zwischen der Aktivierung und dem Start einer Task auftreten. In Abbildung 1-9 hat T2 eine höhere Priorität als T1. Die Task T1 wird hier (bei **ERCOS<sup>EK</sup>**) an der Prozessgrenze von Proz2 unterbrochen, T2 wird ausgeführt und nach der Beendigung von T2 wird T1 mit Proz3 weiterbearbeitet.

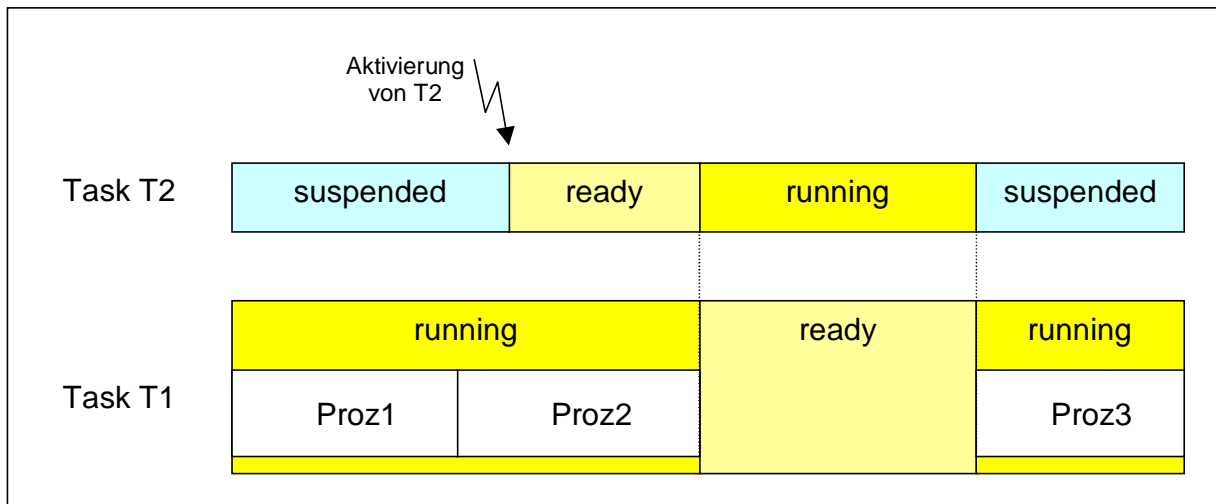


Abbildung 1-9 Aktivieren und Starten von kooperativen Tasks bei ERCOS<sup>EK</sup>

### 1.3.3.2 Preemptive Tasks

Preemptive Tasks können niederpriorere Tasks **an Maschinenbefehlsgrenzen unterbrechen**. Preemptive Taskwechsel brauchen mehr Speicher als kooperative, da der gesamte Kontext gesichert werden muss. Bei kooperativen Taskwechseln ist der Kontext kleiner, da der User-Adressraum kleiner ist und die CPU-Registerinhalte unter Umständen nicht gesichert werden müssen.

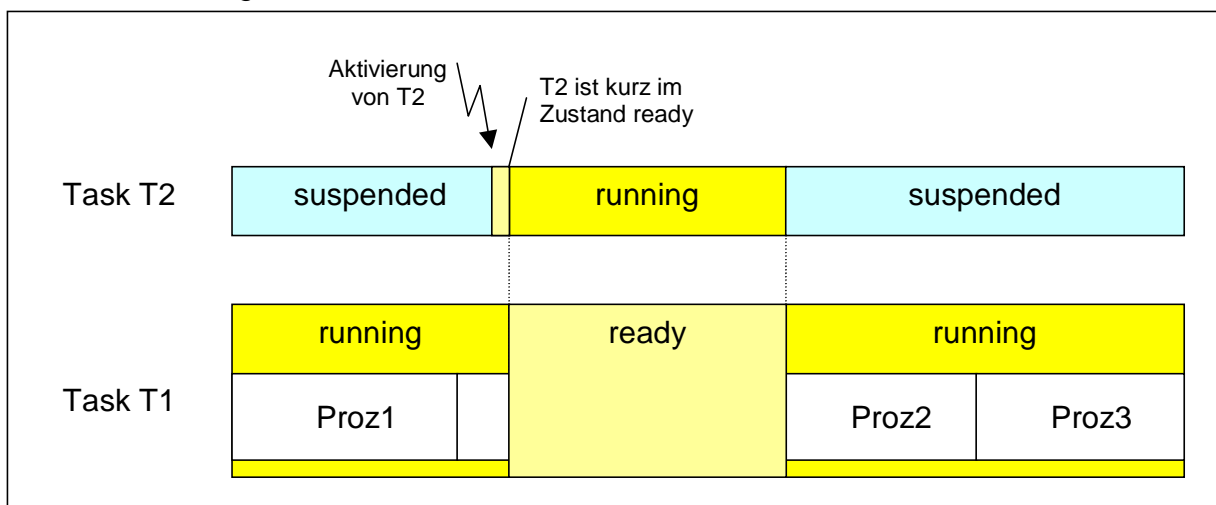


Abbildung 1-10 Preemptive Tasks werden sofort gestartet

Da Taskwechsel hier jederzeit auftreten können, müssen für die **Integrität der Daten** besondere Vorkehrungen getroffen werden. OSEK beschreibt dafür **das Priority Ceiling Protocol**, mit dem Taskwechsel zu höherprioreren Tasks temporär verhindert werden können.

Da der Scheduler immer die höchstprioreren Tasks im ready Zustand abarbeitet, kann es vorkommen, dass niederpriorere Tasks über längere Zeit nicht abgearbeitet werden. Eine deterministische maximale Wartezeit ist hier nicht gegeben.

### 1.3.4 Priority Ceiling Protokoll

Sobald nur eine einzige preemptive Task im System erzeugt wird, ist nicht mehr sichergestellt, dass Befehlsfolgen anderer niederpriorerer Tasks am Stück ausgeführt werden, da preemptive Tasks niederpriorere Task jederzeit unterbrechen können. Wenn "gleichzeitiger" Zugriff auf Ressourcen möglich ist, können diese ungewollt durch höherpriorere Tasks manipuliert werden. Unter Ressourcen versteht man hier Ein- und Ausgangsports oder gemeinsame Speicherbereiche (globale Variablen).

Beispiel:

Es gibt 2 Prozesse, die jeweils aus einer C-Funktion bestehen. Im ersten wird der Betrag einer Variablen `zahl` in die Variable `betrag` abgespeichert. Der zweite Prozess, der eine höhere Priorität als der erste hat, wechselt zyklisch das Vorzeichen der Variablen `zahl`.

<pre>void betrag() {     if (zahl &gt; 0) /*positive zahl*/         betrag = zahl;     else /* negative zahl*/         betrag = - zahl; }</pre>	<pre>void wechsle() {     zahl = -zahl; }</pre>
---	---

Im ungünstigsten Fall wird die Funktion `wechsle()` direkt nach der Abarbeitung der `if`-Selektion in der Funktion `betrag()` abgearbeitet. Dadurch bekommt die Variable `zahl` einen neuen Wert und der Betrag wird falsch. Der folgende Programmfluss demonstriert dieses Verhalten. Der `else`-Zweig wird weggelassen, da er hier aus Programmsicht nicht ausgeführt wird.

<code>betrag()</code>	<code>wechsle()</code>	wert von <code>zahl</code>	wert von <code>betrag</code>
<pre>void betrag() { if (zahl &gt; 0)  betrag = zahl;</pre>	<pre>void wechsle() {     zahl = -zahl; }</pre>	2	
		2	
		-2	<b>-2</b>

Eine bekannte Abhilfe ist hier die Benutzung der von Dijkstra 1986 eingeführten Semaphoren[Weber97], um die globale Variable `zahl` für die Dauer der Betragsrechnung zu sperren.

Im prioritätsgesteuerten Echtzeitbereich haben Semaphoren jedoch Probleme, da sie Deadlocks und Prioritätsumkehrungen zur Folge haben können.

Semaphoren können den Caller blockieren, wenn auf die angeforderte Ressource bereits zugegriffen wurde. Dies kann zu Deadlocks führen, die ernste Softwarefehler nach sich ziehen können. Deadlocks treten auf, wenn zwei Tasks wechselseitig auf zwei durch Semaphoren geschützte Ressourcen zugreifen.

Neben den Deadlocks führen Semaphoren zum Problem der **Prioritäteninversion**. Dies führt wiederum zu Verzögerungen, wenn versucht wird, auf eine Ressource zuzugreifen. Die folgende Abbildung stellt ein Beispiel einer Prioritätsumkehrung dar. Die Task A mit niedriger Priorität bekommt bei  $t_0$  **exklusiven Zugriff** auf die Semaphore S. Task A wird von der Task C mit hoher Priorität zum Zeitpunkt  $t_1$  unterbrochen, welche **versucht, ebenfalls zur Zeit  $t_2$  auf die Semaphore zuzugreifen**. Da die Semaphore bereits von der Task A belegt ist, wird die Task C blockiert und muss warten, bis die Semaphore S frei wird: Es wird also bei  $t_2$  **die Task A** weiterbearbeitet. Bei  $t_3$  **gelangt Task B** in den ready-Zustand und wird sofort gestartet, da sie eine **höhere Priorität als A** hat. Task B mit mittlerer Priorität kann jetzt für eine beliebige Dauer ausgeführt werden und blockiert damit indirekt die Ausführung der höherprioriten Task C. Dieses Phänomen wird **Prioritätsumkehrung** genannt, da die Task C mit hoher Priorität auf die Beendigung der Task B mit mittlerer Priorität warten muss (auch wenn die Task B keinen Zugriff auf die Semaphore S fordert).

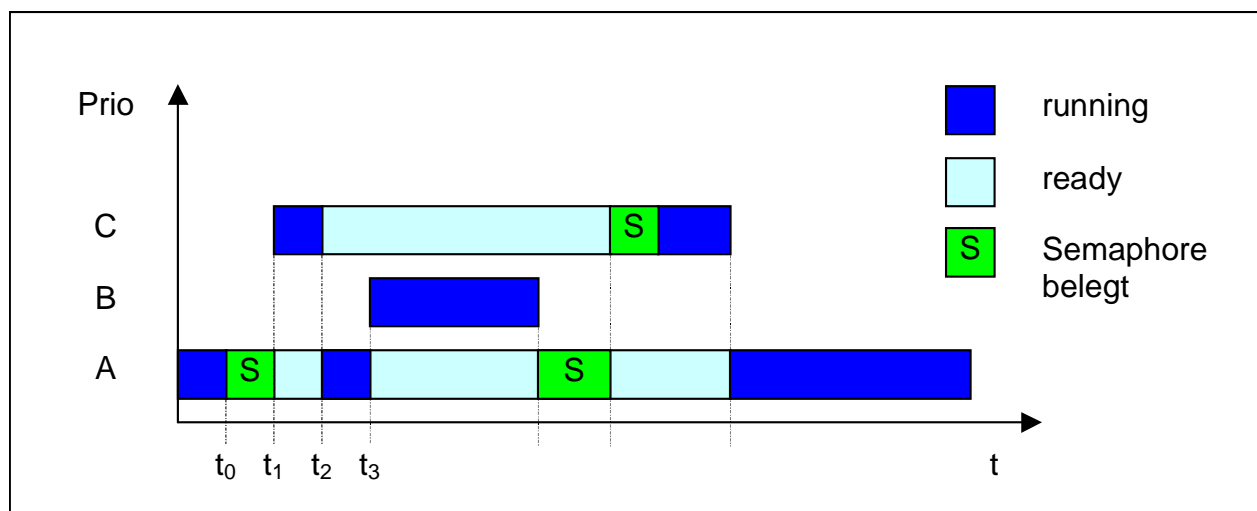


Abbildung 1-11 Prioritätsumkehrung bei Semaphoren

OSEK schlägt anstatt der Verwendung von Semaphoren das Priority Ceiling Protokoll vor, um Ressourcen gegen gleichzeitige Benutzung zu schützen.

Bei dem **Priority Ceiling Protokoll** wird die Task, die eine Ressource anfordert, während der Belegung der Ressource dynamisch auf die höchste Priorität aller Tasks, die auf diese Ressource zugreifen, gesetzt. Diese **Ceiling-Priorität einer Ressource** kann offline zum Compilerzeitpunkt als **Maximum der Prioritäten** aller auf die Ressource zugreifenden Tasks bestimmt werden.

In der folgenden Abbildung wird obiges Beispiel nochmals mit dem Priority Ceiling Protokoll gezeigt:

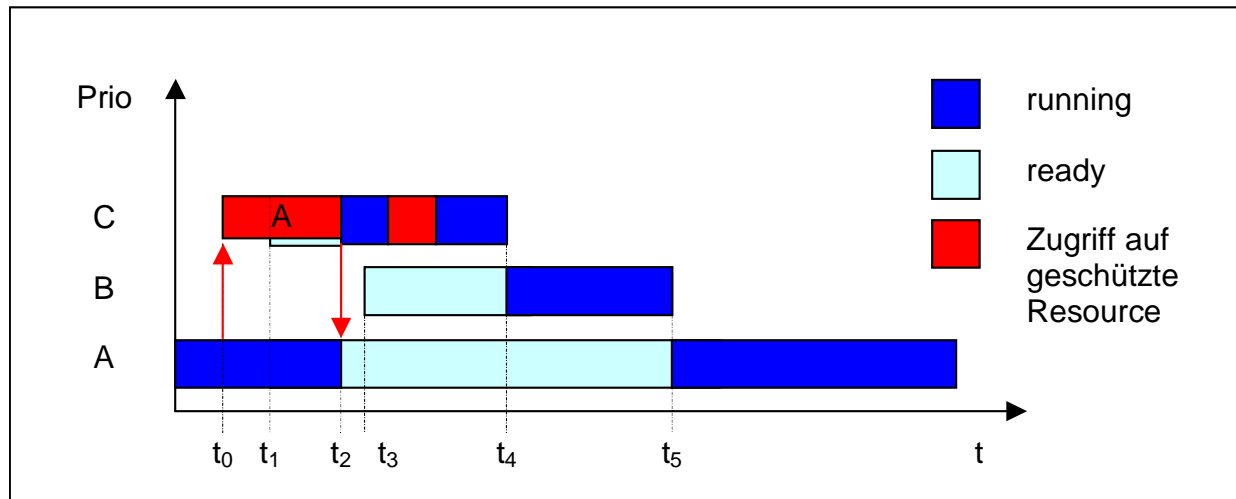


Abbildung 1-12 Prioritätsanhebung beim Priority Ceiling Protokoll

Zum Zeitpunkt  $t_0$  fordert Task A die Ressource an und wird dadurch auf die **Ceiling Priorität**, also auf die Priorität der Task C gesetzt. Task C tritt bei  $t_1$  in den ready-Zustand und muss warten, bis A bei  $t_2$  die Ressource freigibt und wieder auf die Ausgangspriorität sinkt. A wird jetzt durch C unterbrochen und wartet im ready-Zustand, während C ausgeführt wird. Die bei  $t_3$  aktivierte Task B muss bis zur Beendigung von C bei  $t_4$  warten, da C eine höhere Priorität als B hat. Nach der Ausführung von B kann A bei  $t_5$  mit der normalen Ausführung fortfahren. Die Abarbeitung der höherprioreren Task C wird hier nur durch Task A verzögert, während sie auf die geschützten Ressource zugreift.

### 1.3.5 Schutz von Variablen – Ressourcen und Messages

Zwischen Tasks und Prozessen können keine Variablen über Parameter ausgetauscht werden. Man kann nur über globale Variablen kommunizieren. Wie im letzten Kapitel beschrieben, müssen Daten bei gleichzeitigem Zugriff mehrerer Tasks geschützt werden. Eine Möglichkeit dazu sind Ressourcen. Die gewünschten Betriebsmittel - globale Variablen, Ein- und Ausgangsports oder sogar der Scheduler – sind innerhalb des Aufrufs von `GetResource()` und `ReleaseResource()` gegen Zugriff durch andere Tasks geschützt.

Die zweite Möglichkeit des sicheren Datenaustausches sind Messages. Messages sind in der OSEK COM API spezifiziert. Dazu gehören auch Queues, mit denen mehrere Nachrichten nach dem FIFO-Prinzip zwischen Tasks ausgetauscht werden können.

StateMessages ermöglichen - ähnlich wie Ressourcen- den geschützten und gezielten Lese- und/oder Schreibzugriff auf "globale Variablen". Bei StateMessages werden Kopien der ursprünglichen Message in den aufgerufenen Prozess übernommen, auf denen dann gearbeitet werden kann. Bei jeder Prozessdefinition wird im OIL-Header angegeben, welche Messages lesend bzw. schreibend verwendet werden. Da auf Kopien gearbeitet wird, braucht man keine Semaphore bzw. Ressource, wenn bei zwei Prozessen einer schreibend und der andere lesend auf die gleiche Message zugreift. Eine höherpriorere Task wird somit - im Vergleich zum Priority Ceiling Protocol - nicht durch eine niederpriorere Task in ihrer Ausführung angehalten.

### 1.3.6 Timer

Tasks können entweder durch expliziten Aufruf aus einem Prozess mit `ActivateTask()` oder zeitgesteuert über einen Alarm oder eine Timetable aktiviert werden. Diese Timer sind nützlich, wenn Tasks **zeitgesteuert abgearbeitet** werden sollen oder wenn eine Task **nach einer bestimmten Zeitspanne** ein- oder mehrmalig aktiviert werden soll. Es ist jedoch aufgrund der Prioritäten der laufenden Tasks nicht sichergestellt, dass der Zeitplan eingehalten wird.

Während der Alarm im OSEK-Standard definiert wurde, ist die Timetable eine Erweiterung von ERCON<sup>EK</sup>. Es gibt jedoch auch bei den Alarmen zusätzliche, über den OSEK-Standard hinausgehende Zusätze.

Taskaktivierungen mit variabler Wiederholungsperiode und Verzögerungszeit werden durch Alarme ermöglicht.

Ein Alarm löst eine bestimmte Zeit nach seiner Aktivierung einmalig oder periodisch den Start einer Task aus.

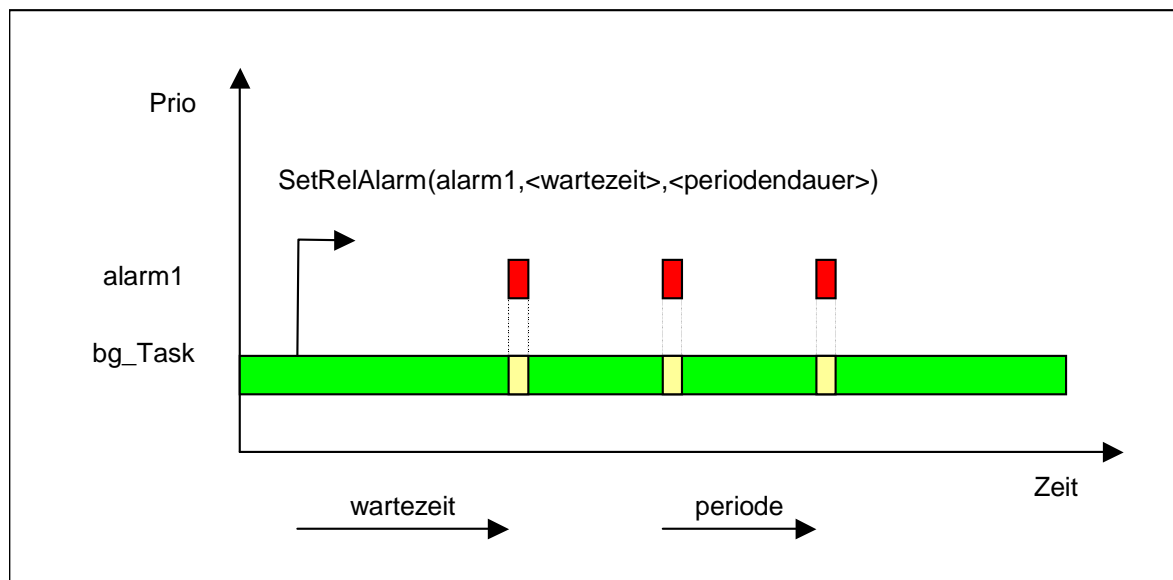


Abbildung 1-13 Aktivierung eines Alarms

Für Alarme gibt es unter anderem folgende Befehle:

DeclareAlarm (<identifizier>)	Bekannt machen des Alarms am Programm-anfang	OSEK
GetAlarm (<identifizier>, <Ticks_ptr>)	Auslesen der Zeit in Ticks, bis der Alarm ausgelöst wird. Gleichzeitig kann über den Rückgabewert geprüft werden, ob der Alarm aktiviert ist.	OSEK
SetRelAlarm (<identifizier>, <wartezeit>, <periodendauer>)	Startet den Alarm mit dem Namen <identifizier>. Der Alarm löst <wartezeit> Ticks nach dem Start aus und wiederholt sich dann alle <periodendauer> Ticks.	OSEK
SetAbsAlarm (<identifizier>, <zeitpunkt>)	Startet den Alarm mit dem Namen <identifizier>. Der Alarm löst zum Zeitpunkt <zeitpunkt> aus.	OSEK

<startzeit>, <periodendauer>)	<startzeit> des dem Alarm zugrunde liegenden Timers aus und wiederholt sich dann alle <periodendauer> Ticks.	
CancelAlarm (<identifizier>)	Löscht den angegebenen Alarm.	OSEK
ChangeAlarmPeriode (<identifizier>, <neuePeriodendauer>)	Ändert die Periodendauer eines aktivierten Alarms	<b>nicht OSEK</b>

Bei einer **Timetable** werden zyklische Taskaktivierungen definiert. Da in einer Timetable meist mehrere Aktionen definiert sind, wird die Ausführungsfolge schon offline zum Compilezeitpunkt berechnet. Timetables brauchen weniger Ressourcen als Alarmer.

Beispiel:

Task\_A soll alle 100 ms aufgerufen werden, erster Zeitpunkt: 0 ms  
 Task\_B soll alle 200 ms aufgerufen werden, erster Zeitpunkt: 40 ms  
 Task\_C soll alle 300 ms aufgerufen werden, erster Zeitpunkt: 150 ms

Diese drei Taskaktivierungen haben als kleinstes Gemeinsames Vielfaches eine Periodendauer von 600 ms. Somit reicht es, eine Tabelle für die Aktivierungen in dieser Zeit zu definieren und diese zyklisch abzuarbeiten: Diese Tabelle wird vom Präprozessor von ERCOS<sup>EK</sup>. erzeugt und liegt dann als komplexer Datentyp im C-Code vor. Die Struktur ist ähnlich nachfolgender Tabelle:

Zeitpunkt in ms	Aktion
0	AktiviereTask(Task_A)
40	AktiviereTask(Task_B)
100	AktiviereTask(Task_A)
150	AktiviereTask(Task_C)
200	AktiviereTask(Task_A)
240	AktiviereTask(Task_B)
300	AktiviereTask(Task_A)
400	AktiviereTask(Task_A)
440	AktiviereTask(Task_B)
450	AktiviereTask(Task_C)
500	AktiviereTask(Task_A)
600	<b>wiederhole Tabelle</b>

Man sollte die Zyklen innerhalb einer Timetable so wählen, dass sich sehr schnell eine Gesamtwiederholung ergibt. Das Kleinste Gemeinsame Vielfache aller Periodendauern sollte relativ klein sein, damit die Tabelle klein bleibt und nicht zu viel Speicherplatz verbraucht.

Abbildung 1-14 zeigt die zeitliche Aktivierung dieser Timetable. Die Priorität der einzelnen Tasks soll hier nicht betrachtet werden. Im Allgemeinen haben häufig wiederholte Tasks eine höhere Priorität als weniger häufig wiederholte.

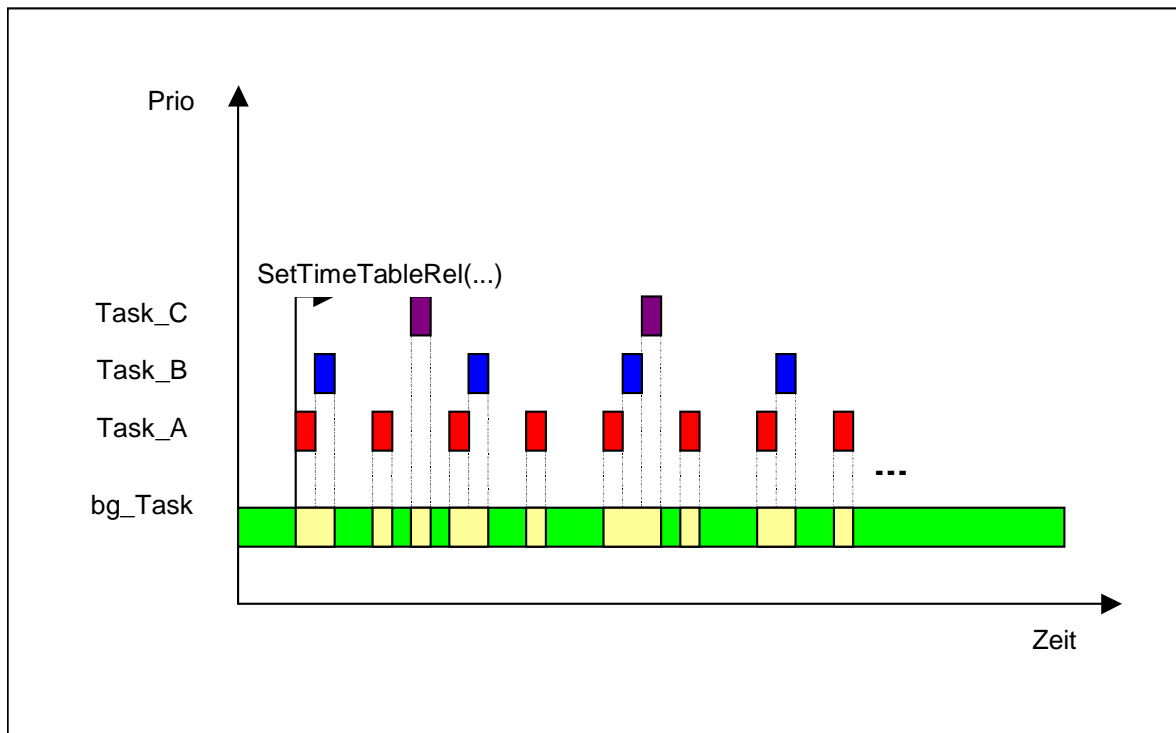


Abbildung 1-14 Zyklische Taskaktivierung durch eine Timetable

### 1.3.7 Startup und Shutdown des Systems

Die gesamte Software für ein Mikrocontrollerprogramm inklusive Betriebssystem wird im ROM des Mikrocontrollers gespeichert. Dadurch gibt es hier weniger Stufen beim "Hochfahren" als beim PC. Bei einem PC werden nacheinander das BIOS, der Bootloader im MBR<sup>8</sup>, der Bootloader der Partition und schließlich das Betriebssystem gestartet.

Bei Mikrocontrollern wird die gesamte Software in **einem** Programm im ROM untergebracht. Das bedeutet, dass die Software wie bei jedem C-Programm mit der Funktion `void main(void)` gestartet wird. In dieser Funktion erfolgt die grundlegende (hardware-spezifische) Initialisierung und der Aufruf des Betriebssystems mit dem OSEK-Befehl `StartOS(<ApplicationMode>)`. Über den `ApplicationMode` können verschiedene Ausführungsmodi gestartet werden. Der `ApplicationMode` beinhaltet einen Descriptor, der angibt, durch welche Task der Modus initialisiert wird und welche Timetables gestartet werden sollen. In der Initialisierungstask wird die Initialisierungsprozedur definiert. Die Initialisierungsprozedur setzt globale Variablen und ruft Initialisierungsroutinen des Betriebssystems auf.

OSEK spezifiziert die Funktion `StartupHook()`, die noch vor der ersten Initialisierungstask aufgeführt werden kann. Wenn diese Funktion eingesetzt werden soll, muss bei ERCOS<sup>EK</sup> die Struktur `OS`, in der ein Teil der Betriebssystemkonfiguration abgelegt ist, entsprechend angepasst werden.

Während der Initialisierung bis hierher sind nur einfache Interrupts zugelassen, die keine Dienste des Betriebssystems aufrufen. Die allgemeinen Interrupts werden erst nach dem Hochfahren des Systems freigegeben.

<sup>8</sup> MBR: Master Boot Record, Boot-Bereich der ersten Festplatte

Das Betriebssystem kann von einem Anwendungsprogramm oder vom Betriebssystem selbst **beendet** werden. OSEK definiert dazu den Befehl `shutdownOS()`, welcher das **Betriebssystem sofort beendet** mit der Konsequenz, dass alle Tasks und Prozesse unterbrochen werden. Nach dem Herunterfahren führt der Mikrokontroller den auf `startOS()`folgenden Befehl aus. Es ist somit möglich, das Betriebssystem öfter nacheinander zu starten.

Neben `ShutdownOS()` gibt es bei ERCOS<sup>EK</sup> noch ähnliche Befehle, um das Betriebssystem nach Abarbeitung aller im ready-Zustand liegenden Tasks herunterzufahren oder es in einem anderen Applikationsmode zu starten.

#### 1.4 Die Objekte der OSEK Implementation Language OIL

Der Sprachumfang von OIL beschreibt ein OSEK-Betriebssystem. Da die einzelnen Komponenten "etwas miteinander zu tun haben", also in einer Relation zu einander stehen, sich gegenseitig benutzen und teilweise aggregieren, kann ein Klassen- oder Objektdiagramm über die Elemente des Betriebssystems erstellt werden.

Das Objektdiagramm von Abbildung 1-15 zeigt den vollen Umfang von OIL für die Conformance Klasse ECC2. Bei einer nach BCC1 oder BCC2 konformen Implementierung fehlt das Event-Objekt. Man sieht deutlich, dass das Betriebssystem von OSEK auch Schnittstellen zur von OSEK definierten Kommunikationsschnittstelle (COM) und zum Netzmanagement (NM) beinhaltet.

Dieses Objektdiagramm zeigt die Standardattribute der Objekte, wie sie von OSEK definiert sind. Jede Implementierung kann **zusätzliche Attribute** zu den Objekten hinzufügen. Dieses Prinzip entspricht im Grunde einer "Ableitung" des OSEK-Frameworks. Außerdem können die Wertebereiche der Attribute eingeschränkt werden.

Beispiel: OSEK beschreibt das Attribut `PRIORITY` des Objektes `TASK` als `UINT32` (32bit-Integer). Die Implementierung kann diesen Wertebereich einschränken. In ERCOS<sup>EK</sup> wird - wie in Kapitel 1.3.2 besprochen – dieser Wertebereich auf die wirklich verwendeten Prioritäts-Ebenen, beispielsweise 15 eingeschränkt. In diesem Punkt wird das Prinzip des Entwurfs durch Verträge (**Design by Contract**) verletzt: Die Implementierung hält die Vorbedingung der abstrakten OSEK-Definition nicht ein, da der Bereich der abgeleiteten Implementierung eingeschränkt wird.

Die meistgenutzte Beziehung zwischen den Objekten ist die "Benutzt"-Beziehung. Man könnte die Richtung dieser Beziehung noch mit spitzen Pfeilen  $\longrightarrow$  angeben, um auszudrücken, wer wen benutzt.

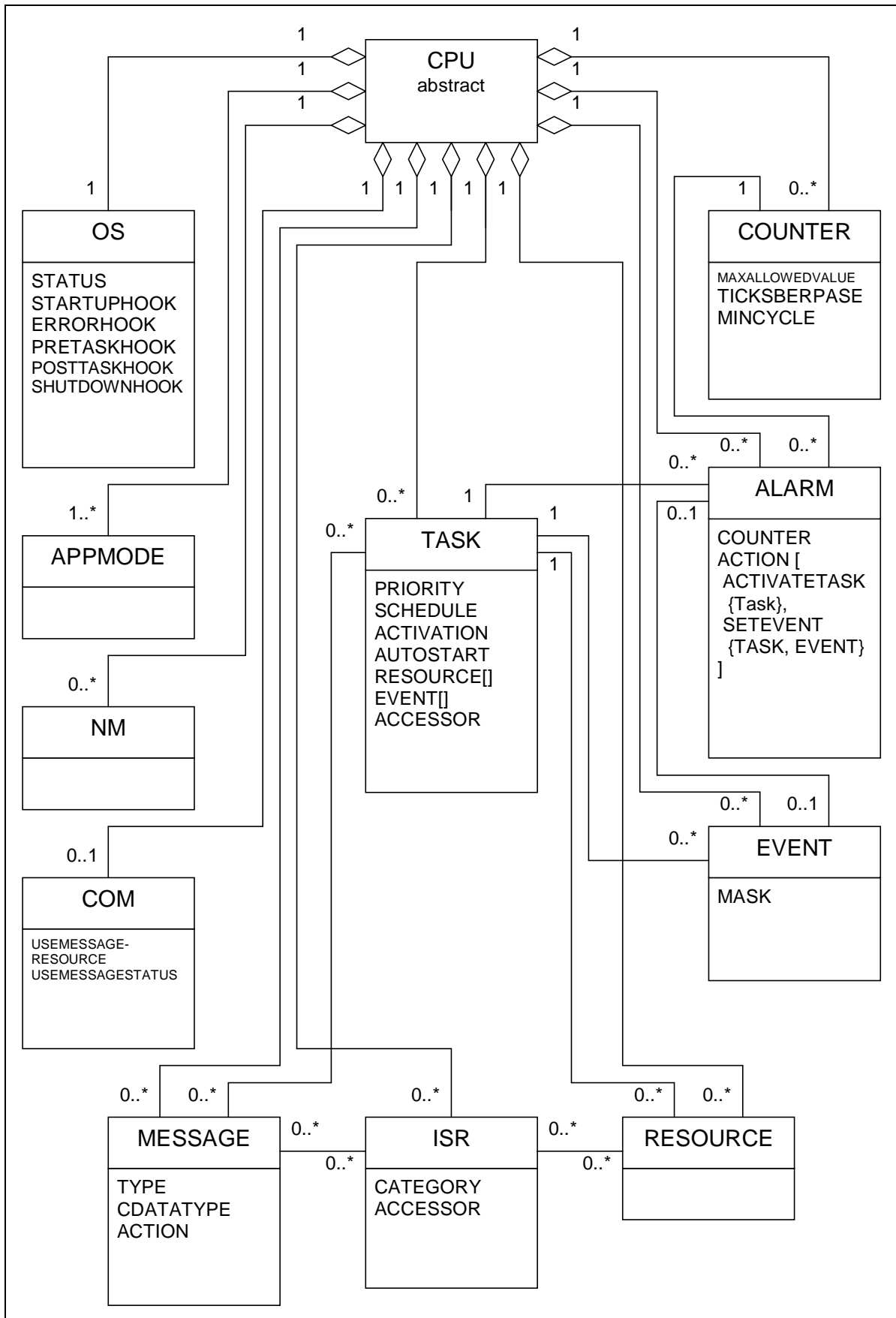


Abbildung 1-15 Objektdiagramm von OIL<sup>9</sup> mit den Standardattributen

<sup>9</sup> In diesem Diagramm fehlen die Beziehungen zwischen Message-ACTION und TASK bzw. EVENT

## 2 Praktische Anmerkungen

### 2.1 Einführung in die Entwicklungsumgebung ERCOS<sup>EK</sup>

Bei einem Steuergerät, dessen Anwendung auf dem Betriebssystem ERCOS<sup>EK</sup> aufbaut, wird die Software in ANSI-C geschrieben. Die Betriebssystemfunktionen von ERCOS<sup>EK</sup> liegen als fertige C-Bibliothek des Herstellers vor und werden beim Linken zum Anwendungsprogramm hinzugebunden.

Um eine in C geschriebene Anwendung in das Betriebssystem ERCOS<sup>EK</sup> einzubinden benutzt man das Programmpaket ESCAPE. Es besteht hauptsächlich aus den C-Bibliotheken und Header-Files des Betriebssystems sowie einer mächtigen Make-Umgebung mit einem auf Perl-Dateien basierendem Präprozessor. Das geschriebene Softwareprojekt wird dann mit einem prozessorabhängigen C-Compiler für das entsprechende Zielsystem in Maschinencode übersetzt.

Da der Quellcode der Anwendungsprogramme noch durch den Präprozessor bearbeitet wird, bevor er kompiliert wird, werden die ERCOS<sup>EK</sup>-Dateien im Allgemeinen mit der Endung `.esc` abgespeichert.

Das kleinste und einfachste C-Programm ist bekanntlich das "HelloWorld!"-Programm von Kernighan/Ritchie. Eine Implementierung mit ESCAPE ist "etwas" aufwendiger als das Originalprogramm und besteht aus mehreren Dateien. Hier sollen zum Verständnis nur die C-Datei mit dem Programmcode `main.esc` und die entsprechende OIL-Konfigurations-Datei `conf.esc` betrachtet werden. Zusätzlich gibt es bei ESCAPE noch Dateien für die Betriebssystemkonfiguration, für selbstdefinierte Datentypen und für das Fehlerhandling. Diese werden jedoch hier nicht gezeigt.

```

/*****
*   FILE: main.esc
*****/

/*-----
*   OIL-Version
*-----
*/

/*OIL  OIL_VERSION      = "2.1";  OIL*/
/*OIL  ESCAPE_VERSION   = "1.0";  OIL*/

/*-----
*   Include files
*-----
*/
#include "erc_def.h"
#include <e_inface.h>

#include "conf.h"

/*-----
*   ESCAPE-Application Mode and Task declarations
*-----
*/
DeclareAppMode(my_appmode_drive); /* Der Applikationsmode legt einen
                                   Anwendungszustand fest */

```

```

DeclareTask(my_inittask); /*einmalige Initialisierung beim Startup*/
DeclareTask(hello_task); /*einzige Task der Anwendung*/

/*Der Initialisierungsprozess wird waehrend des Startens des OS ausgefuehrt*/
/*OIL
    PROCESS my_initproc { };
OIL*/
void my_initproc (void)
{
    ActivateTask(hello_task);
}

/*OIL
    PROCESS hello_process { };
OIL*/
void hello_process (void)
{
    printf("Hello World!");
}

/*-----
 *   Functions
 *-----

*/
void main (void)
{
    StartOS(my_appmode_drive);
}

/*****/

```

*Quellcode 2-1 C-Datei für das HelloWorld! Programm (main.esc)*

```

/*****
 * FILE: conf.esc
 * ESCAPE-ERCOSEK-configuration
 *****/
OIL_VERSION      = "2.1";
ESCAPE_VERSION  = "1.0";

#include "erc_def.h"
#include "conf.h"
#include <e_inface.h>

#define PRIO_A      1      /* Priority my_task_A          */

CPU appExample {

ESCAPE_INCLUDE = "os_setup.h";
#include "os_setup.h"

/* tasks for my_appmode_drive */
    TASK hello_task {
        PRIORITY = PRIO_A;
        PROCESS = hello_process;
    };

/* init tasks */
    INITTASK my_inittask {
        PROCESS = my_initproc;
    };

```

```
/* application modes */
  APPMODE my_appmode_drive {
    INITTASK = my_inittask;
  };
}; /*Ende der Struktur CPU */
```

*Quellcode 2-2 Konfigurations-Datei des HelloWorld Programms (conf.esc)*

In der Konfigurationsdatei sieht man deutlich die Struktur `CPU`, die in OIL definiert ist und die anderen OIL-Objekte wie Tasks oder Application Modes beinhaltet. `INITTASK` ist kein Standard-Attribute von OIL, sondern eine Erweiterung von ESCAPE.

Beim Startup wird das Programm mit der `main`-Methode gestartet. Innerhalb dieser wird mit `StartOS(my_appmode_drive)` das Betriebssystem aktiviert. Im Application Mode `my_appmode_drive` ist als Initialisierungstask die Task `my_inittask` angegeben, die als nächstes ausgeführt wird. In ihr wird als einziger Prozess der Initialisierungsprozess `my_initproc` ausgeführt. Im Initialisierungsprozess `my_initproc` wird hier nur die Task `hello_task` mit dem Befehl `ActivateTask (hello_task)` aufgerufen. Die Task `hello_task` besteht ihrerseits aus dem Prozess `hello_process` und wird mit definierter Priorität ausgeführt. Letztendlich erfolgt im Prozess `hello_process` die erwartete Ausgabe

```
hello World!
```

auf dem Bildschirm.<sup>10</sup>

---

<sup>10</sup> Da Steuergeräte meistens keinen Bildschirm haben, ist diese Ausgabe nur bei einer Emulation möglich.

## 3 Anhang

### 3.1 Literaturverzeichnis

#### Bücher

- [Douglass\_99]** Doing Hard Time  
Bruce Powel Douglass  
Addison-Wessley, 1999
- [etas\_ercb]** ERCOS<sup>EK</sup> Benutzerhandbuch  
ETAS GmbH & Co KG, Stuttgart 1999
- [etas\_ercr]** ERCOS<sup>EK</sup> Referenzhandbuch  
ETAS GmbH & C KG, Stuttgart 1999

#### Online-Dokumente

- [osek\_os2.1]** OSEK/VDX Operating System  
Version 2.1 vom 22. Mai 2000  
[www.osek-vdx.org/mirror/os21.pdf](http://www.osek-vdx.org/mirror/os21.pdf)
- [osek\_web]** [www.osek-vdx.org](http://www.osek-vdx.org)  
Web-Site von OSEK. Spezifikationen und Informationen zum  
OSEK-Standard.