Ausarbeitung im Rahmen der PG Autolab zum Thema: OSEK[1]-OS

geschrieben von Oliver Botschkowski

1 Offene Systeme und deren Schnittstelle für die Elektronik im Kraftfahrzeug

Inhaltsverzeichniss

Kapitel 1: Einleitung.....03

Kapitel 2: Einführung in OSEK-OS.....03

2.1: Architektur.....03

2.2: Task Management.....04

2.3: Interrupt Processing.....07

2.4: Event Mechanism.....07

2.5: Resource Management.....08

2.6: Conformance Classes.....09

Quellenangaben

Kapitel 1: Einleitung

Einleitend möchte ich erwähnen warum das OSEK Konsortium ein neues Betriebssystem entworfen hat und welche Vorteile es bietet. In der Automobilindustrie wurde viel Geld für die Entwicklung oder Anpassung von Software für die verschiedenen Control Units der verschiedenen Firmen ausgegeben. Um die Kosten und die Entwicklungszeit einzusparen wurde OSEK-OS entwickelt. Es sollte Interfaces bieten, die in allen Bereichen so anwendungsunabhängig wie möglich gestaltet sind. Außerdem sollte es effizient auf die vorhandene Anwendung anzupassen sein, also konfigurierbar auf die darunterliegende Hardware und skalierbar auf die Größe der Anwendung. Weiterhin stand die Verifikation der Funktionen und die Möglichkeit der Prototypenentwicklung im Vordergrund.
Ich konnte im Rahmen meines Vortrags und der Ausarbeitung natürlich nicht jedes Feature, das OSEK-OS bietet, ansprechen. Für alle anderen Informationen sollte die Spezifikation, die auf www.osek-vdx.org erhältlich ist, erste Quelle sein.

Kapitel 2: Einführung in OSEK-OS

2.1 Architektur

OSEK-OS wird bei Erzeugung einmal statisch skaliert und konfiguriert, d.h. die Anzahl der Tasks, die Interrupts, die benötigten Services und die Prioritäten werden einmalig bei Erzeugung festgelegt.

Es werden in Realzeit parallel mehrere Prozesse ausgeführt, wobei folgende Prozesslevel vorhanden sind:

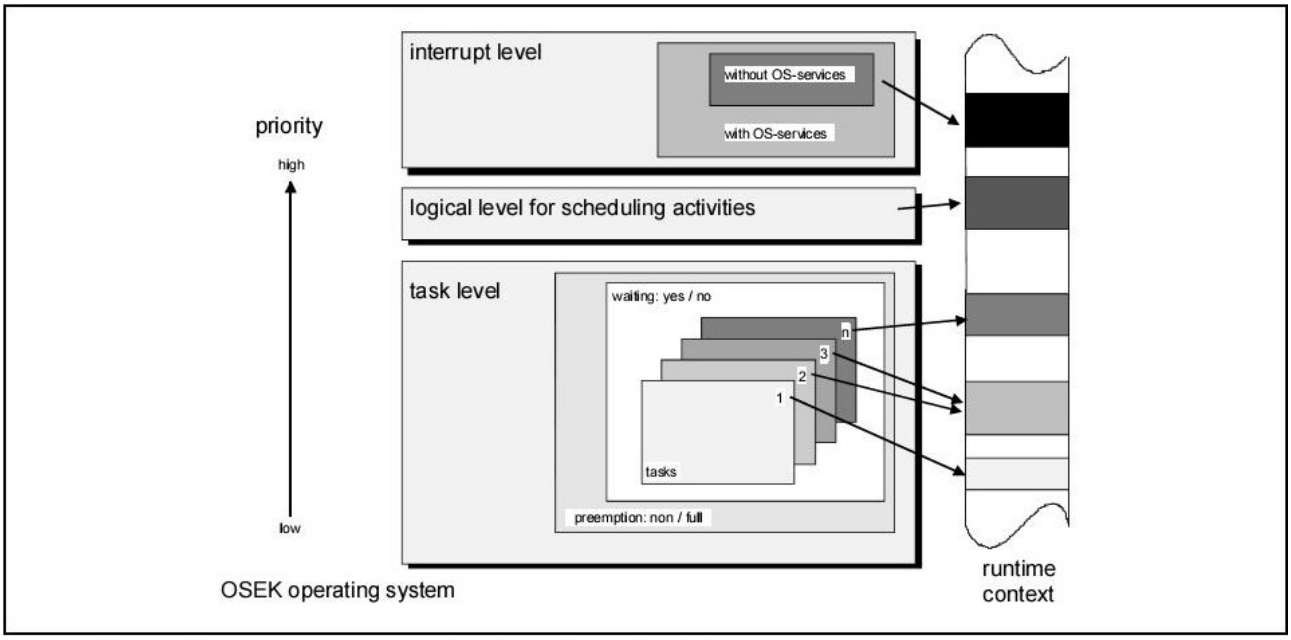

Figure 3-1 Processing levels of the OSEK operating system

Auf dem obigen Schaubild sind die drei Prozessebenen von OSEK-OS dargestellt, wobei die höheren Ebenen, höhere Prioritäten besitzen. Auf der untersten Ebene sind die Tasks, die in einem speziellen Zustand sind und full oder non preemptive gescheduled werden (siehe Kapitel 2.2). Eine Ebene höher befindet sich der Scheduler, der die Tasks dem Prozessor je nach Priorität zuweist. Auf der höchsten Ebene befindet sich das Interrupt Level auf dem sowohl Interrupts mit als auch ohne OS-Services behandelt werden (siehe Kapitel 2.3). Diese Aufteilung in die Prioritätsebenen ist nur

logisch, da es dem Scheduler ermöglicht werden muss, die Tasks zu unterbrechen. Desweiteren muss es den Interrupts ermöglicht werden, den Scheduler zu unterbrechen.

2.2 Task Management

Innerhalb des Task Managements gilt es zu klären, welche Task-Arten es innerhalb des Betriebssystems gibt, wie und zu welchen Zeitpunkten gescheduled wird.
Ich beginne damit Zustandsautomaten zu den beiden Task-Arten vorzustellen, nämlich den Basic und Extended Tasks.
Basic Tasks:

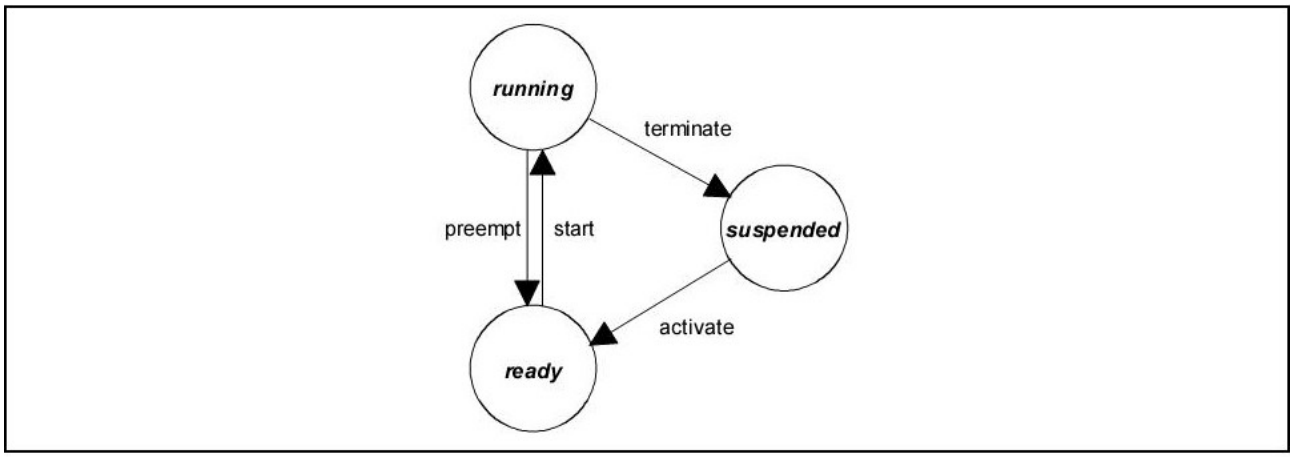

Figure 4-3 Basic task state model

Im obigen Zustandsautomaten sieht man 3 Zustände und die jeweiligen Übergänge. Im „suspended“ State ist der Task wenn er gerade nicht darauf wartet, dem Prozessor zugewiesen zu werden. Wenn der Task aktiviert wird, wechselt er in den „ready“ State. Dort wartet der Task bis er durch den Scheduler dem Prozessor zugewiesen wird. Wenn er dem Prozessor zugewiesen ist, wechselt der Task in den „running“ State. Nun kann er nur noch terminieren, um damit in den „suspended“ State überzugehen oder von einem andern Task preempted wird (siehe Abschnitt über full preemptive scheduling in diesem Kapitel).

Extended Tasks:

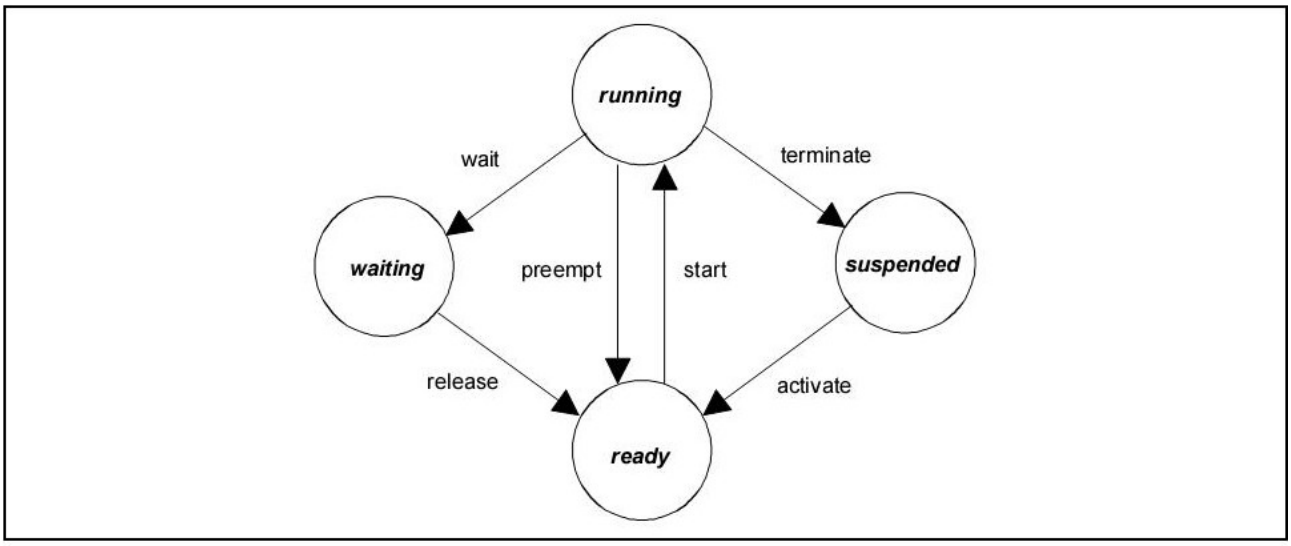

Figure 4-1 Extended task state model

Die Extended Tasks sehen bis auf einen Zustand genauso aus wie die Basic Tasks. Es kommt neben den Zuständen „running“, „suspended“ und „ready“ noch der „waiting“ State dazu. In den „waiting“ State wechselt ein Task wenn er dem Prozessor zugewiesen ist, also im „running“ State ist und dann auf ein Event wartet (zu Events siehe Kapitel 2.4). Wenn das Event eintritt, wird der Task released und kommt in den „ready“ State, womit er dann dem Prozessor wieder zugewiesen werden kann.

Nun komme ich dazu wie der Scheduler die Tasks dem Prozessor zuteilt. Wie schon am Anfang erwähnt, hat jeder Task eine Priorität statisch zugewiesen bekommen. Es können jedoch mehrere Tasks die gleiche Priorität besitzen. Folgendes Schaubild zeigt, wie der Scheduler die Tasks dem Prozessor zuweist:

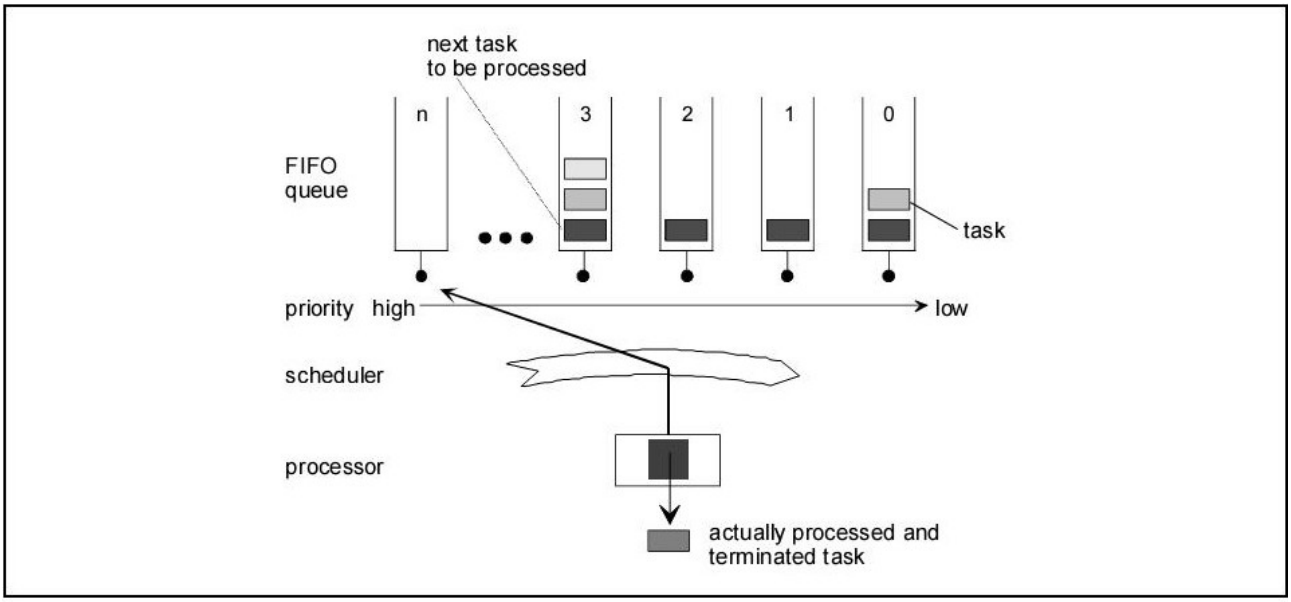

Figure 4-5 Scheduler: order of events

Für jede Priorität hält der Scheduler eine FIFO-Queue. Dort werden Tasks mit der gleichen Priorität nach dem FIFO-Prinzip einsortiert. Wenn der Scheduler aktiv wird, schaut er auf die FIFO Queue mit der höchsten Priorität, die einen Task enthält. Der vorderste Task wird nun dem Prozessor zugewiesen.

Nachdem wir nun wissen wie der Scheduler die Tasks dem Prozessor zuweist, ist noch zu klären wann er aktiv wird. Hierbei gibt es bei OSEK-OS drei Möglichkeiten: das full-, non-, und mixed-

preemptive Scheduling.
Beim full-preemptive Scheduling wird der Scheduler nach folgenden Aktivitäten aufgerufen:

- Terminierung eines Tasks
- Aktivierung eines Tasks auf Task-Level
- Transition in den Waiting State
- Event bei einem wartenden Task (siehe Kapitel 2.4)
- Freigabe von Ressourcen auf Task Level (siehe Kapitel 2.5)
- Rückkehr von Interrupt auf Task Level (siehe Kapitel 2.3)

Hierzu ein Beispiel:

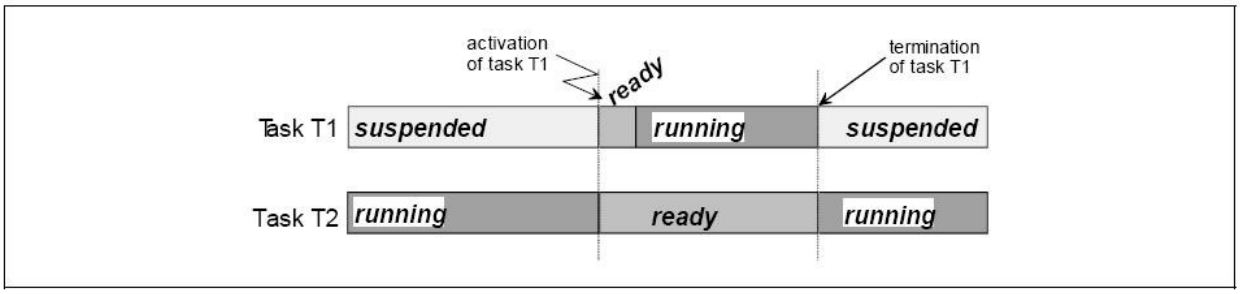

Figure 4-6 Full preemptive scheduling

Im Schaubild sieht man die Tasks T1 und T2, wobei T1 die höhere Priorität hat. T1 ist gerade dem Prozessor zugewiesen, T2 befindet sich im „suspended“ State. Wenn nun T1 aktiviert wird und in den „ready“ State wechselt, wird beim full preemptive Scheduling der Scheduler aufgerufen. Da T1 die höhere Priorität hat, wird er nun dem Prozessor zugewiesen. Nachdem T1 terminiert, wird wiederum der Scheduler aktiv und T2 wird dem Prozessor zugewiesen.
Beim non-preemptive Scheduling wird der Scheduler nur noch nach folgenden Aktionen aufgerufen:

- Terminierung eines Tasks
- Expliziter Aufruf des Schedulers
- Transition in den Waiting State

Hierzu ein weiteres Beispiel:

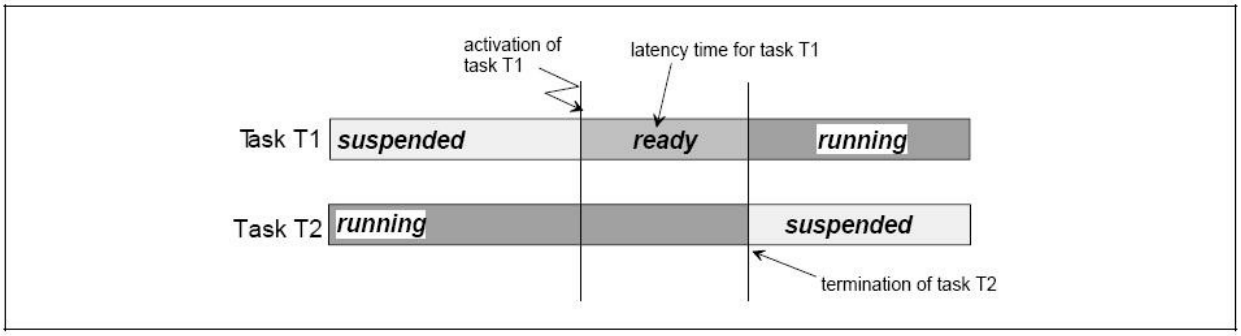

Figure 4-7 Non preemptive scheduling

Wir haben die gleiche Situation wie beim letzten Beispiel. Nachdem T1 aktiviert wird, läuft T2 jedoch weiter, da beim non-preemptive Scheduling nach Aktivierung eines Taks nicht gescheduled wird. Task T1 hat nun Wartezeit bis T2 terminiert und somit der Scheduler aufgerufen wird.

Als letztes der drei Verfahren gibt es das mixed-preemptive Scheduling. Hierbei kann jeder Task preemptive oder non-preemptive gescheduled werden.

2.3 Interrupt Processing

Interrupts sind Unterbrechungen des fortlaufenden Programms durch externe Signale. Nach einem Interrupt wird eine Interrupt Service Routine (ISR) ausgeführt, wonach der Programmablauf fortgeführt wird. Es können auch mehrere Interrupts gleichzeitig auftreten, weshalb sie genauso wie Tasks statische Prioritäten besitzen. Es können auch innerhalb von ISRs weitere Interrupts auftreten, daher kann es passieren, dass die ISRs durch weitere ISRs geschachtelt werden. Es gibt zwei Kategorien von Interrupt Service Routinen: Kategorie 1 benutzt keine OS-Services und nach Abarbeitung der ISR wird am Punkt an dem das Programm unterbrochen wurde fortgesetzt. Kategorie 1 besitzt daher weniger Overhead und verbraucht deswegen weniger Speicher (gerade für kleine Anwendungen interresant).
Kategorie 2 der ISRs darf einige OS-Services benutzen. Welche OS-Services benutzt werden dürfen, kann man in einer Tabelle in der Spezifikation nachlesen. Nach Abarbeitung einer ISR der Kategorie 2 kann der Scheduler aufgerufen werden.

2.4 Event Mechanism

Events sind Objekte, die durch das OS gehandhabt werden und jeweils zu einem Extended Task gehören. Events erwirken Zustandsübergänge vom „waiting“ zum „ready“ State und werden zur Kommunikation zwischen Tasks benutzt. Hierbei kann man sogar soweit gehen eine eigene Scheduling Policy zu implementieren, indem man einen Task hat, der die anderen jeweils mit Events aufweckt und somit Scheduled. Nun zeige ich an zwei Beispielen wie man Tasks bei preemptive und non-preemptive Scheduling synchronisieren kann:

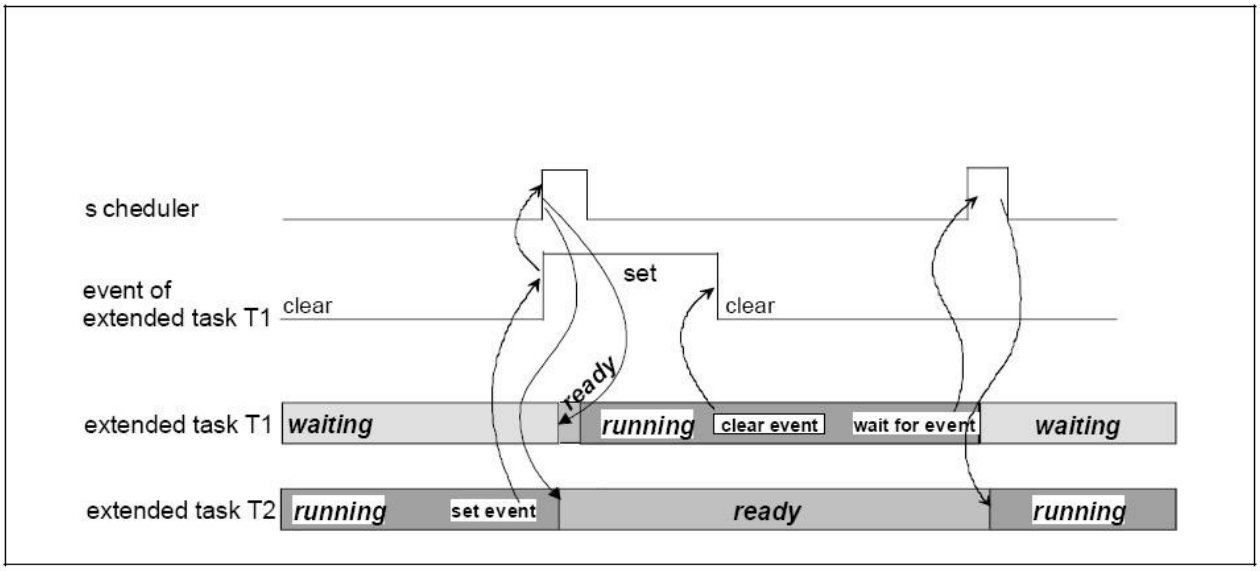

Figure 7-1 Synchronisation of preemptable extended tasks

Im Beispiel gibt es die Tasks T1 und T2, wobei T1 eine höhere Priorität als T2 hat.. Weiterhin gibt es eine Zeile die anzeigt, wann der Scheduler aktiv wird und eine weitere die zeigt wann ein Event für T1 aktiv wird. Die Ausgangssituation ist, dass T1 im „waiting“ State ist (wartet auf ein Event) und T2 gerade dem Prozessor zugewiesen ist. Nun setzt T2 ein Event für T1, woraufhin T1 in den „ready“ State geht und aufgrund des full preemtpive Scheduling der Scheduler aktiv wird (siehe Kapitel 2.2). Nun wird T1 dem Prozessor zugewiesen und arbeitet weiter bis er wiederum auf ein Event wartet und in den „waiting“ State übergeht. Daraufhin wird der Scheduler wieder aktiv und übergibt T2 an den Prozessor. Im Beispiel wird gezeigt wie T1 und T2 über Events synchronisiert werden. T2 entscheidet hierbei wann T1 laufen darf, indem er für T1 ein Event setzt.
Am zweiten Beispiel zeige ich wie der Synchronisationsprozess beim non preemptive Scheduling funktioniert:

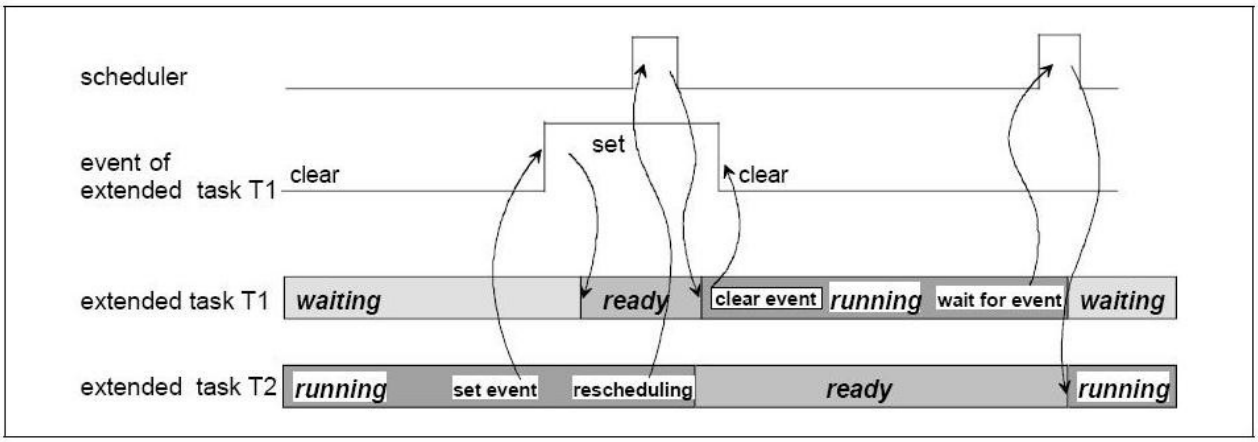

Figure 7-2 Synchronisation of non preemptable extended tasks

Die Ausgangslage ist die gleiche wie beim vorherigen Beispiel. T2 setzt ein Event für den wartenden Task T1. Da hier jedoch non preemptive gescheduled wird, wird der Scheduler erst aktiv wenn er explizit von T2 aufgerufen wird. Hier reicht es also nicht ein Event zu setzen, sondern der Scheduler muss aktiv aufgefordert werden zu schedulen, wenn man den Prozessor an den anderen Task abgeben will.

2.5 Ressource Management

Beim Ressource Management geht es darum den konkurrierenden Zugriff unterschiedlicher Tasks auf Ressourcen zu steuern. Es wird beim Ressource Management folgendes zugesichert:

- Zwei Tasks können nicht die gleiche Ressource zur gleichen Zeit haben
- Keine Priority Inversion
- Kein Deadlock
- Der Zugriff auf Ressourcen endet nicht im Waiting State

Den Begriff des Ressource Managements kann man auch auf die Interrupt Service Routinen erweitern. Dann gilt als weitere Zusicherung:

- Zwei Tasks oder ISR´s können nicht die gleiche Ressource zur selben Zeit nutzen

Um die oben genannten Zusicherungen zu erfüllen, wird das Priority Ceiling Protokoll angewendet, dies werde ich anhand eines Beispiels erklären:
Beim Priority Ceiling Protokoll wird jeder Ressource eine statische Priorität zugeordnet. Diese Priorität ist mindestens so hoch wie die des Tasks mit der höchsten Priorität, die die Ressource anfordert. Weiterhin ist die Priorität der Ressource kleiner als die des Tasks mit der nächsthöheren Priorität, die die Ressource nicht benötigt. Nun wird einem Task, der die Ressource benutzt, die Priorität der Ressource zugeordnet. So wird verhindert, dass andere Tasks, die die selbe Ressource benötigen, nicht in den „running“ State übergehen können.

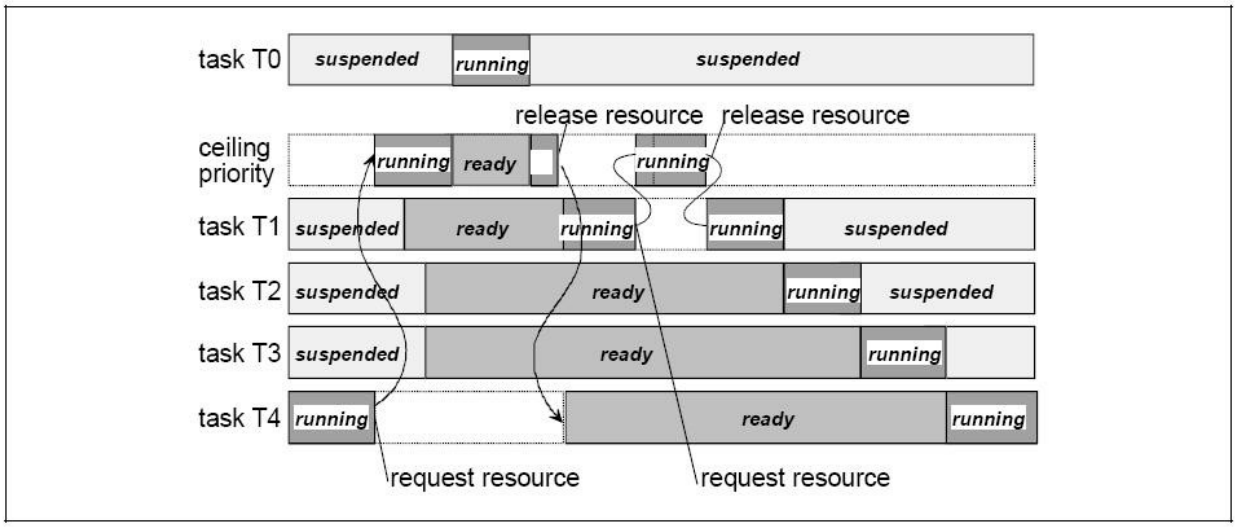

Figure 8-3 Resource assignment with priority ceiling between preemptable tasks.

Im obigen Beispiel sieht man die Funktionalität des Priority Ceiling Protokolls. T0 hat die höchste und T4 die niedrigste Priorität. T1 und T4 wollen die Resource benutzen. Sobald T4 die Resource benutzt, bekommt es die Ceiling Priority und kann, obwohl T1-T3 „ready“ sind, weiterlaufen bis T4 die Resource wieder freigibt. Priority Inversion wird hierbei unterbunden da die maximale Wartezeit von T1, die Zeit ist, die T4 die Resource benutzt. Außerdem kann T4 nur von Tasks unterbrochen werden, die eine größere Priorität als T1 besitzen. Somit wird Priority Inversion durch andere Tasks unterbunden.

2.6 Conformance Classes

Folgende Conformace Classes sind definiert:

- BCC1: nur Basic Tasks unterstützt, nur ein Task pro Priotität möglich, keine multiple activation requests (wenn ein Task einmal läuft darf er nicht noch einmal aktiviert werden um nach Beendigung direkt wieder zu laufen)
- BCC2: nur Basic Tasks, mehrere Tasks pro Priorität möglich, multiple activation requests möglich
- ECC1: wie BCC1 plus extended Tasks möglich
- ECC2: wie ECC1 plus mehrere Tasks pro Priorität möglich und multiple activation requests für basic tasks möglich

Conformance Classes sind definiert, damit OSEK-OS auf die Größe der Anwendung skalierbar wird und man nicht immer alle Funktionen mitbezahlen muss. Weiterhin hat man nach dem Schema im folgenden Bild einen Upgrade-Pfad, in dem man Applikationen, die z.B. auf BCC1 laufen, auch ohne Umbauten auf BCC2, ECC1 und ECC2 laufen lassen kann.

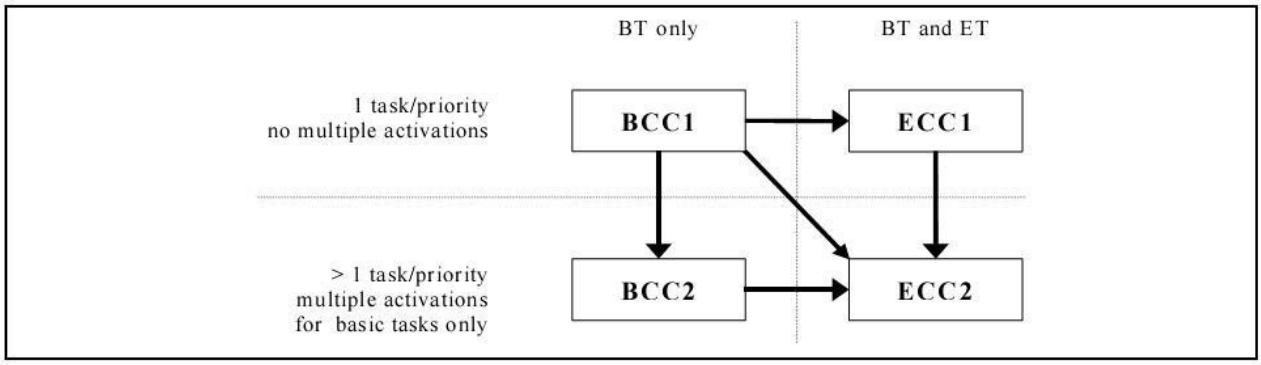

Figure 3-2 Restricted upward compatibility for conformance classes

Quellenverzeichniss:

OSEK/VDX Operating System Specification 2.2.3 (alle Schaubilder hieraus entnommen)
OSEK/VDX Binding Specification 1.4.2
beides erhältlich auf: http://www.osek-vdx.org

http://de.wikipedia.org/wiki/OSEK
http://de.wikipedia.org/wiki/OSEK-OS