OSEK/OSEKtime OS

Ausgewählte Kapitel eingebetteter Systeme

Friedrich Alexander Universität Erlangen-Nürnberg

Wilhelm Haas

15. Juli 2006

1 Einführung

Die Abkürzung OSEK steht für ein im Jahre 1993 gegründetes industrielles Standardisierungsgremium, das vor allem aus Automobilherstellern bestand, und ausgeschrieben „Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug" bedeutet. Zu den Gründungsmitgliedern gehört die BMW AG, Daimler-Benz AG, Siemens AG und einige andere. Im Jahre 1994 schloss man sich mit der 1988 gegründeten französischen VDX[1]-Initiative zusammen und ist seitdem unter der offiziellen Bezeichnung OSEK/VDX bekannt.

Der bedeutenste geschaffene Standard des Gremiums ist das OSEK-OS. OSEK-OS ist ein spezifiziertes Betriebssystem für eingebettete Systeme. In der Wirtschaft wird üblicherweise OSEK als Synonym für OSEK-OS verwendet. OSEK-OS gibt es in zwei Variationen. Zu einem gibt es das OSEK, das ein ereignisgesteuertes System ist, und das OSEKtime-OS, das ein zeitgesteuertes System ist. Beide Systeme sind für eine Monoprozessorarchitektur ausgelegt. Durch die hohe Flexibilität, die durch flexible Konfiguration und den hohen Modularisierungsgrad möglich ist, sind die Systeme auf sehr vielen verschiedenen Hardwarearchitekturen einsetzbar.

Weitere vom Gremium geschaffene Standards sind:

- **OSEK-OIL**[2] definiert eine Beschreibungssprache, mit der OSEK-OS konfiguriert wird.
- **OSEK-ORTI**[3] definiert das Kommunikationsprotokoll des OSEK-OS mit dem Debugger
- **OSEK-COM** definiert das Interprozesskommunikationsprotokoll.
- **OSEK-NM**[4] definiert den Mechanismus des Power-Managements.

2 OSEK-OS

2.1 Architektur

OSEK ist ein statisches Betriebssystem, d. h. zur Laufzeit ist eine Konfigurationsänderung nicht möglich. Noch während der Entwicklung der Anwendungen müssen alle Betriebsmittel und Tasks definiert und konfiguriert werden. Zur Festlegung der Eigenschaften eines Systems existieren vier definierte Konformitätsklassen. Über diese lassen sich die verwendeten Task-Typen, die Anzahl der Aktivierungen eines Tasks und Anzahl Tasks pro Prioritätsebene festlegen. Die Konformitätsklassen sind vor allem zum schnellen und leichteren Verständnis des Systems gedacht. OSEK definiert zwei Prozesstypen. Der eine Prozesstyp entspricht den Interruptroutinen, die vom Betriebssystem verwaltet werden, der Andere den

[1] Vehicle Distributed Executive
[2] OSEK Implementation Language
[3] OSEK Runtime Interface
[4] OSEK Network Management

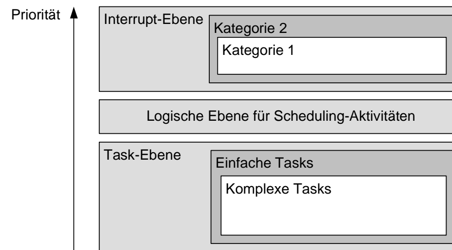

Abbildung 1: Architektur des OSEK-OS

Tasks. Zusätzlich gibt es noch einen Prozess, der exklusiv nur dem Scheduler zu steht. Jeder der drei Prozesse bilden eine eigene Prioritätsklasse (Abbildung 6), welche die Reihenfolge der Prozessorbelegung bestimmt.

Interruptroutinen (ISR) haben grundsätzlich Vorrang vor den Tasks. Die Interrupt-Ebene kann aus einer oder mehreren Interrupt-Prioritäsebenen bestehen. Die Anzahl der Prioritäsebenen und die Zuordnung einer Priorität an eine ISR ist von der Hardware abhängig. Somit haben ISRs eine statisch zugewiesene Priorität. Die Vergabe der Prioritäten an die Tasks muss ebenfalls statisch vorgenommen werden.

2.2 Task-Management

Das OSEK Task-Kozept unterscheidet zwischen zwei Task-Typen, den komplexen Tasks und den einfachen Tasks.

2.2.1 Komplexe Tasks

Komplexe Tasks erwarten während der Ausführung die Zuteilung von Ressourcen, ausserdem haben sie die Möglichkeit mit *WaitEvent* auf ein bestimmtes Ereignis zu warten. Somit ist diese Art der Tasks von anderen Tasks abhängig und können im laufendem Betrieb blockiert werden.

Das Zustandsmodell der komplexen Tasks (Abbildung 2) lässt sich an Hand vier Zustände verdeutlichen:

- **laufend**
  Im Zustand *laufend* darf sich zu jedem Zeitpunkt maximal nur ein Task befinden. Tasks, in diesem Zustand, bekommen die CPU zugewiesen und werden somit ausgeführt.
- **bereit**
  Befindet sich ein Task in diesem Zustand, dann hat dieser alle nötigen Bedingungen erfüllt um in den Zustand *laufend* zu wechseln. In diesem Zustand befindliche Tasks warten auf die Zuteilung der CPU. In welcher Reihenfolge diese die CPU zugewiesen bekommen entscheidet der Scheduler.
- **warten**
  In diesem Zustand befindliche Tasks warten auf das Eintreten eines Ereignis.
- **suspendiert**
  Im Zustand *suspendiert* warten die Tasks passiv auf ihre Aktivierung.

2.2.2 Einfache Tasks

Einfache Tasks laufen durch ohne während der Ausführung blockiert zu werden oder die Möglichkeit zu haben auf ein Ereignis mit *WaitEvent* zu warten. Sie geben den Prozessor erst frei, wenn

- sie sich terminieren,
- sie von einem höher priorisierten Task verdrängt werden,

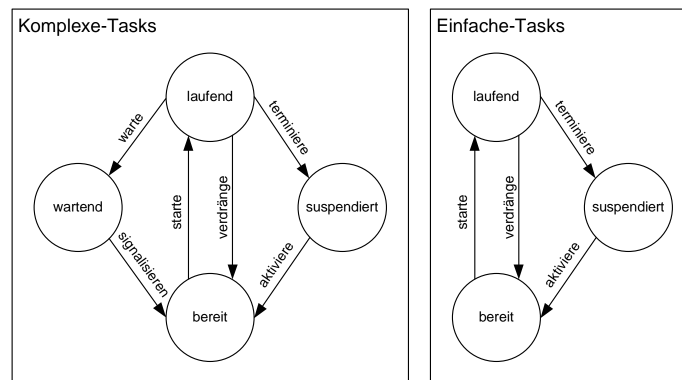

Abbildung 2: Zustandsmodelle der Komplexen-Tasks und Einfachen-Tasks

- eine Unterbrechung aufgetretten ist und der Prozessor zu der Unterbrechungsbehandlung schaltet.

Das Zustandsmodell ähnelt dem der komplexen Tasks. Der Unterschied liegt darin, dass einfache Tasks nicht blockieren können, dadurch ist der Zustand *warten* überflüssig.

2.2.3 Scheduling

Der OSEK-Scheduler plant die Tasks in der Reihenfolge ihrer Prioritäten ein. Wie schon erwähnt, werden die Prioritäten den Tasks statisch zugewiesen, d. h. noch während der Entwicklungszeit. Im Betrieb ändert sich die Priorität der Tasks nicht, ausser wenn das Priority-Ceiling-Protokoll (Seite 5) zum Einsatz kommt. Tasks in der selben Prioritätsebene werden in der Reihenfolge der Aktivierungen eingeplant.

Je nach Konfiguration, kann der Scheduler auf zwei Weisen arbeiten.

- **Verdrängendes Scheduling**
  Ein Task im Zustand *laufend*, kann durch das Aktivieren eines höher priorisierten Tasks in den Zustand *bereit* gesetzt werden. Zusätzlich ist es auch möglich den Scheduler für eine Weile zu blockieren (*GetResource*). So erreicht man, das der Task eine gewissen Zeit nicht verdrängt wird.
- **Nichtverdrängendes Scheduling**
  Beim nichtverdrängendem Scheduling wird das Scheduling nur an bestimmten Stellen durchgeführt. Diese wären:
  - das erfolgreiche Beenden eines Tasks
  - expliziter Aufruf des Schedulers (Systemfunktion *Schedule*)
  - wenn mit *WaitEvent* in den Zustand *warten* gewechselt wird

  Nachteil dieses Verfahrens ist, dass die Latenzzeit der Tasks mit höherer Priorität von den niedriger priorisierten Tasks abhängt. Das kann dazu führen, dass die Deadline eines Tasks nicht eingehalten wird.

Man hat auch die Möglichkeit einen Mischbetrieb zu kreieren. Dabei hängt die Schedulingstrategie von der Verdrängbarkeit der einzelnen Tasks ab. Dies kann über Attribute der Tasks eingestellt werden.

Ein Task kann nur durch sich selbst beendet werden. Bei der Terminierung kann ein Nachfolger Task angegeben werden, dies geschieht über *ChainTask*. Jeder Task muss als letzte Anweisung entweder ein *TerminateTask* oder *ChainTask* stehen haben. Ein Beenden ohne diese Anweisungen ist strengstens verboten und führt zu undefiniertem Verhalten.

2.3 Interrupt-Behandlung

In der Interrupt-Behandlung (ISR) wird zwischen zwei Kategorien unterschieden, denn

- **ISR Kategorie 1**
  Verwendet keine Systemfunktionen. Nachdem die ISR beendet wurde, wird die Ausführung des unterbrochenen Tasks wieder an der Unterbrechungsstelle fort gesetzt, d. h. eine ISR der Kategorie 1 hat keinen Einfluss auf die unterbrochenen Tasks.
- **ISR Kategorie 2**
  Dürfen nur eingeschränkt die Systemfunktionen nutzen.

Wie Anfangs erwähnt, ist die Anzahl der Unterbrechungsprioritäten von der Hardware abhängig, genauso das Einplanen der Interrupts.
Zusätzlich ist es möglich alle Interrupts abzuschalten bzw. wieder zu ermöglichen (*DisableAllInterrupts*, *EnableAllInterrupts*) oder aber auch nur die der Kategorie 2 (*SuspendOSInterrupts*, *ResumeOSInterrupts*). Typischerweise verwendet um kurze kritische Bereiche zu schützen.

2.4 Ereignismechanismus

OSEK bietet den komplexen Tasks die Möglichkeit auf bestimmte Ereignisse zu warten. Dies wird vor allem für Synchronisationszwecke verwendet. Einfache Tasks dürfen diesen Mechanismus nicht nutzen. Das Warten auf ein Ereignis initiiert einen Wechsel vom Zustand *laufend* in den Zustand *wartend*.

Ereignisse sind Objekte die vom Betriebssystem verwaltet werden und einem bestimmten Task zugeordnet sind. Komplexe Tasks besitzen eine gewisse Anzahl an Ereignisobjekten und sind somit ihre *Besitzer*. Über Ereignissobjekte werden binäre Informationen verschickt bzw. empfangen. Die Bedeutung dieser wird von der Anwendung definiert, wie z. B. die Verfügbarkeit eines bestimmten Betriebsmittels oder Empfang einer neuen Nachricht.

Jenachdem, ob ein Task ein Besitzer eines Ereignisses ist, gibt es verschiedene Operationen mit denen die Ereignisobjekte modifiziert werden können. Alle Tasks, seien diese einfach oder komplex, und auch ISRs der Kategorie 2 dürfen ein Ereignis auslösen. Nur die Besitzer haben das Recht und die Möglichkeit ein Ereignis zurückzusetzen und auf dieses zu warten. In allen Fällen ist der Empfänger eines Ereignisses ein komplexer Task und somit können einfache Tasks und ISR der Kategorie 2 nicht blockieren. Will ein momentan laufender Task auf ein schon eingetretenes Ereignis warten, dann verbleibt dieser im Zustand *laufend*.

2.5 Betriebsmittelverwaltung

Die Betriebsmittelverwaltung wird zur Koordinierung des Zugriffs konkurierender Tasks verschiedener Prioritäten auf gemeinsame Betriebsmittel eingesetzt. Je nach Wunsch kann diese auf die Koordinierung des Zugriffs von Tasks und ISR auf Betriebsmittel erweitert werden.

Die Betriebsmittelverwaltung stellt sicher, dass keine zwei Tasks zur selben Zeit die selbe Ressource besitzen. Ist die Verwaltung auch auf ISRs erweitert, muss zustätzlich sichergestellt werden, dass keine zwei Tasks oder ISRs die selbe Ressource zur selben Zeit besitzen. Die Funkionalität der Betriebsmittelverwaltung ist nur in folgenden Fällen nützlich:

- bei verdrängbaren Tasks
- bei gemeinsamer Betriebsmittelnutzung zwischen Tasks und ISRs
- bei gemeinsamer Betriebsmittelnutzung zwischen ISRs

2.5.1 Probleme blockierender Synchronisation

Ein Mittel zur Koordinierung der Zugriffe könnte z. B. die blockierende Synchronisation sein. Diese hat aber als Nachteil, dass es sehr leicht zum Deadlock bzw. zur (unkontrollierten) Prioritätumkehr kommen kann.

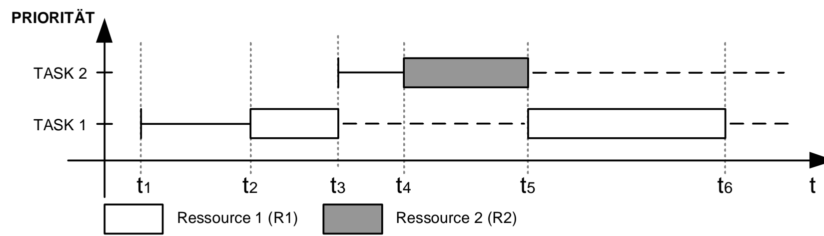

Abbildung 3: Beispiel für einen Deadlock

Beispiel für einen Deadlock (Abbildung 3)

Zeit (t)	TASK 1	TASK 2
1	wird aktiviert und gestartet	
2	belegt Ressource 1 (R1)	
3	wird verdrängt	wird aktiviert und gestartet
4		belegt Ressource 2 (R2)
5	wird gestartet	möchte R1, wird blockiert
6	möchte R2, wird blockiert	

Beispiel für eine Prioritätsumkehr (Abbildung 4)

Prioritätsumkehr bedeutet, dass ein aktiver höher priorisierter Task auf einen Niedrigprioren warten muss.

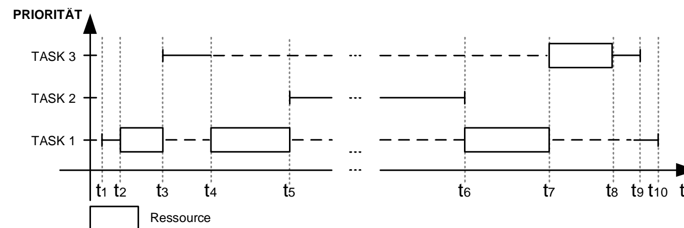

Abbildung 4: Beispiel zur (unkontrollierten) Prioritätsumkehr

Zeit (t)	TASK 1	TASK 2	TASK 3
1	wird aktiviert und gestartet		
2	belegt Ressource 1 (R1)		
3	wird verdrängt		wird aktiviert und gestartet
4	wird gestartet		möchte R1, wird blockiert
5		wird aktiviert und gestartet	
6	wird gestartet	terminiert	
7	gibt R1 frei, wird verdrängt		wird gestartet und belegt R1
8			gibt R1 frei
9	wird gestartet		terminiert
10	terminiert		

2.5.2 OSEK Priority-Ceiling-Protokoll

Die Lösung zu diesen beiden Problemen liefert das Zugriffsprotokoll mit dem Namen OSEK Priority-Ceiling-Protokoll (OSEK Prioritätsobergrenzenprotokoll). Vier einfache Regeln müssen befolgt werden:

- Während der Systementwicklung wird jedem Betriebsmittel eine statische Prioritätsobergrenze zugewiessen.
  $Prioritätsobergrenze\,(r) = max\,(T.Priorität : T.belegt\,(r))\ \forall T \in Tasks, \forall r \in Betriebsmittel$
- Möchte ein Task eine Ressource belegen, deren Prioritätsobergrenze größer oder gleich der Priorität des Tasks ist, dann wird die Priorität des Task auf die Prioritätsobergrenze angehoben.
- Gibt ein Task ein Betriebsmittel wieder frei, so wird seine Priorität auf die ursprüngliche Priorität gesetzt, die er vor dem Belegen dieser Ressource hatte.
- Ressourcen müssen im LIFO-Prinzip wieder frei gegeben werden.

Das Priority-Ceiling-Protokoll enthält implizit die Deadlockvermeidung und sorgt dafür, dass unkontrollierte Prioritätsumkehr nicht entsteht.

Beispiel für Priority-Ceiling-Protokoll (Abbildung 5)

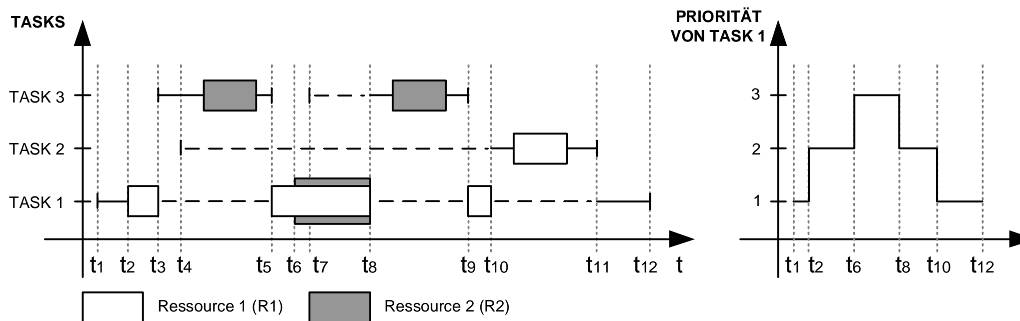

Abbildung 5: Beispiel zum OSEK Priority-Ceiling-Protokoll

Zeit (t)	TASK 1	TASK 2	TASK 3
1	wird aktiviert und gestartet		
2	belegt R1, hat Priorität 2		
3	wird verdrängt		wird aktiviert und gestartet
4		wird aktiviert	
5	startet		wird terminiert
6	belegt R2, bekommt Priorität 3		
7			wird aktiviert
8	gibt R2 frei, hat Priorität 2		startet
9	startet		terminiert
10	gibt R1 frei, hat Priorität 1	wird gestartet	
11		terminiert	
12	terminiert		

2.6 Hooks

Das OSEK-OS stellt dem Benutzer eine Zahl von Hook-Routinen zur Verfügung. Damit wird dem Benutzer die Möglichkeit gegeben bestimmte Aktionen bei bestimmten Verarbeitungsschritten des Betriebssystems aufzurufen.
Die Hook-Routinen werden vom Betriebssystem ausgeführt. Dabei haben diese eine höhere Priorität als alles Tasks des Systems und werden auch nicht von den ISR der Katorie 2 unterbrochen. Wie bei den ISRs ist die Benutzung der Systemfunktionen eingeschränkt.

Im OSEK Betriebssystem können die Hook-Routinen für folgende Zwecke genutzt werden:

- **Beim Systemstart**
  Nach dem Start des Systems und noch vor der Aktivierung des Schedulers wird die *StartupHook* aktiviert. Dieser Hook kann für Initialisierungen genutzt werden.

- **Vor dem Herunterfahren des Systems**
  Bei gravierenden Fehlern wird das System zum Herunterfahren gezwungen. Durch den Aufruf von *ShutdownOS* wird die *ShutdownHook*-Routine ausgeführt. Es ist dem Benutzer überlassen was er in diesem Fall macht. Möglich ist auch, dass in einem System, in dem OSEKtime und OSEK-OS koexistieren, nur das OSEK-OS heruntergefahren wird.

- **Zum Debuggen**
  Bei jedem Kontextwechsel werden zwei Hook-Routinen aufgerufen, die *PreTaskHook* und *PostTaskHook*. *PreTaskHook* wird direkt, nachdem ein Task in den Zustand *laufend* gewechselt ist, ausgeführt und *PostTaskHook* kurz vor dem verlassen des Zustandes *laufend*. Beide Routinen können zum Debuggen oder für Zeitmessungen eingesetzt werden.

- **Im Fehlerfall**
  Die Fehler-Hook-Routine *ErrorHook* wird jedesmal aufgerufen wenn eine Systemfunktion einen Wert ungleich *E_OK* zurück liefert. Der Wert *E_OK* weisst hin, dass die Funktion, die diesen Wert geliefert hat, problemlos ausgeführt wurde. Ein rekursiver Aufruf von *ErrorHook* ist nicht möglich, d. h. Fehler die während der Abarbeitung der *ErrorHook*-Routine entstehen, können nur über den Rückgabewert erkannt werden. Generäl wird zwischen zwei Fehlerarten unterschieden:

  - **Softwarefehler**
    Das Betriebssystem konnte eine Systemfunktion nicht ausführen, nimmt aber an, dass der interne Zustand korrekt ist. In diesem Fall überlässt man dem Benutzer die Entscheidung was in diesem Fall geschehen soll.
  - **Gravierende Fehler**
    In diesem Fall ist das weitere Ausführen des Betriebssystems nicht mehr möglich. Das System ruft *ShutdownOS* auf, mit dem das Herunterfahren des Systems beginnt.

3 OSEKtime-OS

3.1 Architektur

OSEKtime-OS ist ein zeitgesteuertes Echtzeitsystem. Schon vor dem Betrieb sind alle Parameter bekannt, dazu gehören die Aktivierungszeitpunkte der Tasks, ihre Deadlines und auch die Ausführungszeiten. Während der Entwicklung wird ein statischer Ablaufplan erstellt der zur Laufzeit von dem OSEKtime Dispatcher umgesetzt wird. Die Einlastung der Arbeitsaufträge geschieht streng nach Fahrplan. Weil die Einplanung off-line passiert, können Algorithmen mit hoher Berechnungskomplexität zum Einsatz kommen. Das erfordert, dass alle Paramater noch vor dem Betrieb bekannt sind. Mit speziellen Algorithmen können somit Ablaufpläne generiert werden die keine Synchronisation zwischen Tasks erfordern. Somit hat das OSEKtime-OS keine Mechanismen zur Synchronisation von Tasks definiert.
In OSEKtime gibt es die Möglichkeit das komplette OSEK-OS als Subsystem in den Kern einzubauen. Dies ist aber nur mit folgenden Bedingungen möglich:

- Das komplette OSEKtime muss eine höhere Priorität haben als das OSEK-OS Subsystem.
- Die Hardware muss genügend viele Interrupt-Ebenen zur Verfügung stellen.
- Speicherschutzmechanismen müssen eingesetzt werden.

Nur wenn die drei Punkte erfüllt sind, kann das OSEK als Subsystem eingebaut werden ohne dass es OSEKtime in irgendeiner Weise im Betrieb stört. Nichverdrängbare OSEK-Tasks sind erlaubt, sie sind aber nur in der Umgebung des OSEKs nichtverdrängbar. Ein in Ausführung befindlicher OSEK-Task wird bei Aktivierung eines OSEKtime-Tasks verdrängt. Zusätzlich muss noch gesagt werden, dass OSEK- und OSEKtime-Tasks keine Betriebsmittel teilen dürfen.

Wird das Subsystem nicht benötigt, so muss es durch einen Idle-Task ersetzt werden.

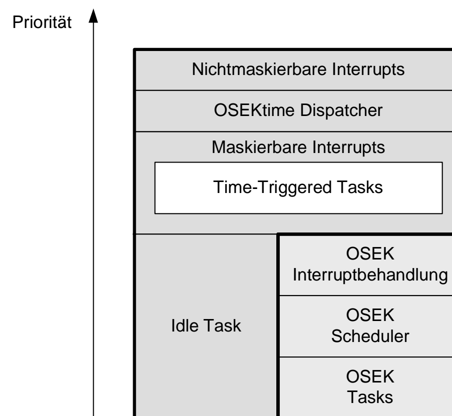

Abbildung 6: Architektur des OSEK-OS

3.2 Task-Management

Tasks werden sequenziell, nach dem statischen Ablaufplan, gestartet und laufen bis zum Terminierungszeitpunkt. Um ein deterministisches System gewährleisten zu können, muss es zu jeder Zeit möglich sein die Ausführungszeit des schlimmsten Falls (Worst Case Execution Time) zu bestimmen. Blockierungen durch Ereignise oder durch den Betriebsmittelverwalter finden nicht statt.
Ein Task kann drei Zustände (Abbildung 7) einnehmen:

- **laufend**
- **verdrängt**
- **suspendiert**

OSEKtime-OS benutzt verdrängbares Scheduling. Jeder Task kann jeden anderen Task verdrängen. Ein Durchlauf der Ablauftabelle wird eine Dispatcherrunde genannt. Der Dispatcher selbst wird durch einen Interrupt aktiviert. Die Quelle des Interrupts ist ein lokaler, logischer Zeitgeber, der mit einem globalen Zeitgeber, sofern einer existiert, synchronisiert ist. Wie auch OSEK, darf ein OSEKtime-Task nur durch sich selbst terminiert werden.
So genannte Application Modes (Anwendungsverfahren) erlauben ein effizientes Management verschiedener Zustände in der Anwendungssoftware. Application Modes werden über die Dispatcher-Tabelle definiert. Alle Dispatcherrunden haben jeweils die gleiche Länge, ein Wechsel zwischen den Modi ist nur am Ende einer Dispatcherrunde möglich. Typische Application Modes sind beispielsweise Initialisierung, normaler Betrieb, Herunterfahren / Nachlauf.

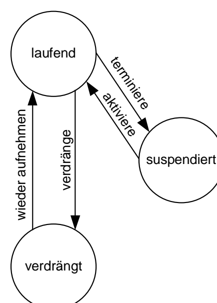

Abbildung 7: Zustandsmodell der OSEKtime-Tasks

3.2.1 Deadline-Beobachtung

Ein sehr wichtige Eigenschaft eines Tasks, in einem Echtzeitsystem, ist seine Deadline. Deadline ist ein Zeitpunkt, bis zu dem die Ausführung des Tasks beendet sein muss. Auf Verletzungen der Deadlines muss zur Ausführungszeit geprüft werden. Die Beobachtung der Deadlines wird duch den Dispatcher vorgenommen. Ein Spezieller Eintrag in der Ablauftabelle gibt dem Dispatcher ein Zeichen wann dieser die Prüfung durchführen soll. Wird eine Deadline verletzt, so muss eine Fehlerbehandlung durchgeführt werden.

OSEKtime definiert zwei Mechanismen, mit denen die Deadlines überprüft werden:

- **Strenge-Task-Deadline-Beobachtung**
  Ein Beobachter wird direkt an der Stelle, an der sich die Deadline befindet, positioniert.
- **Keine-Strenge-Task-Deadline-Beobachtung**
  Ein Beobachter kann an eine Stelle nach der Deadline aber noch vor dem Dispatcherrundenende gesetzt werden.

Welcher Mechanismus verwendet wird kann für jeden Task einzeln eingestellt werden.

3.3 Interruptbehandlung

In zeitgesteuerten Systemen muss mit Interrupts vorsichtig umgegangen werden. Für die Tasks sind definierte Zeitpunkte vorgegeben an denen diese starten müssen. Asynchrone Unterbrechungen verursachen immer Verzögerungen. Diese Verzögerungen können dazu führen, dass der ganze Zeitplan durcheinander kommt und im schlimmsten Fall die Deadlines nicht mehr eingehalten werden können. Deshalb unterscheidet sich die Interruptbehandlung des OSEKtime im Gegensatz zu OSEK.

OSEKtime definiert für jeden Interrupt, der eintreten kann, ein Intervall, indem dieser Interrupt maximal nur einmal zum Zuge kommen kann. Wird ein Interrupt erkannt so wird die dazu gehörige Interruptbehandlung durchgeführt und der Interrupt wird deaktiviert. Das Reaktivieren des Interrupts erfolg an bestimmten ausgewiesenen Stellen. Diese Stellen entsprechen Einträgen in der Ablauftabelle, die offline definiert werden. Für jeden Interrupt, der während des Betriebs eintreten kann, gibt es einen Aktivierungseintrag bzw. mehrere Aktivierungseinträge in der Ablauftabelle.

Besondere Vorsicht gilt den nichtmaskierbaren Interrupts. Diese werden, wenn sie eintreten, bearbeitet. Ist damit zu rechnen, dass eine Deadline nicht eingehalten werden kann, dann sollte man auf nichtmaskierbare Interrupts verzichten.

In OSEKtime haben Tasks keine Möglichkeit Interrupts selbst zu aktivieren bzw. deaktivieren. Alle möglichen Interrupts müssen während der Entwicklung in der Konfiguration spezifiziert werden. Diese werden vom Betriebssystem beim Systemstart aktiviert, alle anderen werden deaktiviert.

Beispiel (Abbildung 8)

Zeit (t)	Interrupt 1	Interrupt 2
$IE_{1,1}$	wird aktiviert	
t_n	Behandlung wird gestartet	
t_{n+1}	wird deaktiviert	
$IE_{2,1}$		wird aktiviert
t_{n+2}		Behandlung wird gestartet
t_{n+3}		wird deaktivier
$IE_{1,2}$	wird reaktiviert	
t_{n+4}	Behandlung wird gestartet	
t_{n+5}	wird deaktiviert	
$IE_{2,2}$		wird reaktiviert
$IE_{1,3}$	wird reaktiviert	

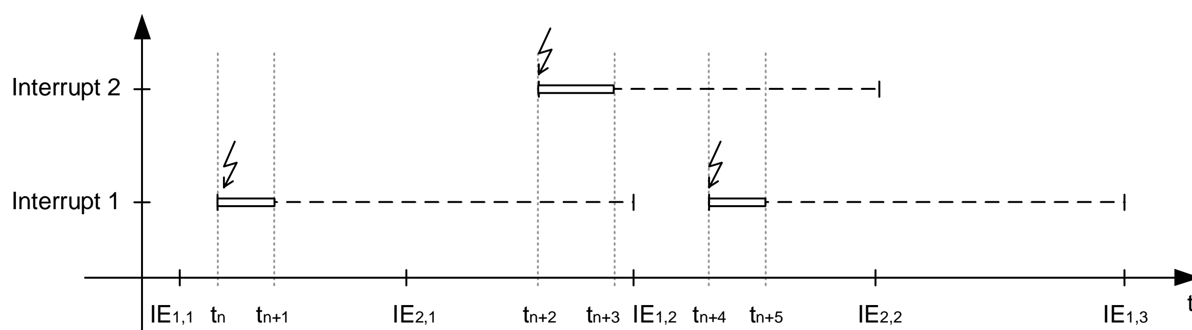

Abbildung 8: Beispiel für die Interruptbehandlung im OSEKtime

3.4 Synchronisation der Zeit

Jede ECU[5] besitzt eine eigene lokale Zeit, die durch einen lokalen Zeitgeber inkrementiert wird. Existiert eine globale Zeit, so wird beim Systemstart oder aber auch während des Betriebs die lokale Zeit mit der globalen Zeit synchronisiert. Die Synchronisation übernimmt dabei die Synchronisationsschicht[6]. Die Synchronisationsschicht hat das Wissen über den Startzeitpunkt jeder Dispatcherrunde.

Während des Betriebs gibt es keine explizite Warnung bzw. Hinweis wenn die lokale Zeit nicht mehr synchron zur Globalen läuft.Um herauszubekommen ob die lokale Zeit synchron zur Globalen läuft kann der Systemaufruf *ttGetOSSyncStatus* verwendet werden. Ist eine Synchronisation nötig, so muss zum Abgleich die Funktion *ttSyncTimes* aufgerufen werden.

Es können drei Synchronisationsarten verwendet werden, um den Abgleich der Uhren beim Systemstart durchzuführen:

- **Synchroner Start**
  Die ECU wird so lange keine Tasks ausführen, so lange keine Synchronität herscht.
- **Asynchroner Start - harte Synchronisation**
  Die Ablauftabelle wird gemäß der lokalen Zeit, ohne zu synchronisieren, einmal komplett durchlaufen. Am Ende der Dispatcherrunde wird die nächstfolgende Runde so lange verzögert bis die Synchronität geschaffen wurde.
- **Asynchroner Start - weiche Synchronisation**
  Wie auch bei der harten Synchronisation, wird die Ablauftabelle gemäß der lokalen Zeit durchlaufen. Die Synchronität wird über mehrere Dispatcherrunden hinweg geschaffen. Dabei werden die Runden um einen bestimmten Zeitwert verzögert, der als Konfigurationsparameter vorher definiert wird.

4 Zusammenfassung

Nicht zu vergessen ist, dass OSEK-OS und OSEKtime-OS statische Betriebssysteme sind. Damit ist z. B. eine dynamische Erzeugung von Tasks während des Betriebs nicht möglich. OSEK-OS ist ein ereignisgesteuertes, OSEKtime-OS ein zeitgesteuertes Betriebssystem. Mit OSEKtime ist es auch möglich OSEK-OS parallel zu betreiben. Unterschiede finden sich auch bei den Tasks. OSEK-OS definiert zwei verschiedene Task-Typen, die einfachen und die komplexen Tasks, während OSEKtime-OS nur einen Task-Typ kennt, den TimeTriggered-Task. Während in OSEK-OS Interrupts sofort bearbeitet werden, wenn diese nicht vorher deaktiviert wurden, wird ein Interrupt in OSEKtime-OS maximal nur ein mal in bestimmten Intervallen abgebearbeitet.

[5]Elektronische Kontrolleinheit
[6]Näheres zur Synchronisationsschicht findet man in der FTCom-Spezifikation.

Literatur

[1] **OSEK/OSEKtime Specification**
Quelle: *http://www.osek-vdx.org/*
Version: OSEK-OS V 2.3.3 und OSEKtime V 1.0

[2] **Echtzeitsysteme, WS 2005 / 06**
Author: *Prof. Dr. Wolfgang Schröder-Preikschat* Quelle: *http://www4.informatik.uni-erlangen.de*

[3] **OSEK-Gremium**
Quelle: *http://de.wikipedia.org/wiki/OSEK*

[4] **OSEK-OS**
Quelle: *http://de.wikipedia.org/wiki/OSEK-OS*

[5] **OSEKtime - ein zeitgesteuertes Betriebssystem**
Quelle: *http://www.elektroniknet.de/topics/kommunikation/fachthemen/artikel/02025.htm*