# OSEKtime - Time-Triggered OSEK/OS

## Projektgruppe 522: AutoLab*

***Eine Experimentierplattform für automotive Softwareentwicklung***

**Fachbereich Informatik, Lehrstuhl 12**

**Arbeitsgruppe „Entwicklung und Betrieb eingebetteter und vernetzter Systeme“**

Gregor Kaleta†

15. November 2007

---

*Homepage: http://ls12-www.cs.uni-dortmund.de/autolab/

†E-Mail: gregor.kaleta@udo.edu

# Inhaltsverzeichnis

1 Einleitung 3

2 OSEKtime 3
- 2.1 Task-Zustandsmodell von OSEKtime . . . . . 4
- 2.2 Dispatching in OSEKtime . . . . . 4
- 2.3 Interrupt-Verarbeitung . . . . . 6
- 2.4 Deadline-Überwachung . . . . . 7
- 2.5 OSEK/VDX-OS als Subsystem . . . . . 8

3 OSEKtime FTCom 9
- 3.1 FTCom Schichtenmodell . . . . . 9
- 3.2 Synchronisation der Systemzeit . . . . . 10

4 Toolkette 12

5 TimeCore 13

6 Zusammenfassung 13

# 1 Einleitung

Die Standardisierung wichtiger Elemente einer fahrzeugweiten Elektronik-Infrastruktur gewinnt immer mehr an Bedeutung. So gründeten bereits 1993 Daimler-Benz, Volkswagen, BMW, Opel, Renault, PSA, Siemens und Bosch die OSEK/VDX-Initiative. Es wurden Standards verabschiedet, die das Betriebssystem (OSEK/VDX-OS), Kommunikations-System (OSEK/VDX-COM), Netzmanagement (OSEK/VDX-NM) und die OSEK Implementation Language (OSEK/VDX-OIL) umfassen. Seit Februar 2005 liegt die Spezifikation [1] von OSEK/VDX-OS in der Version 2.2.3 vor. Heute wird der OSEK/VDX-Standard bei zahlreichen Automobilherstellern und Zulieferern genutzt. Die großen Automobilhersteller setzen OSEK-Implementierungen bereits serienmäßig ein [2]. Die gesammelten Erfahrungswerte haben gezeigt, dass sich OSEK/VDX für die typischen Anforderungen im Innenraum und Antriebsstrang bewährt hat. Dabei handelt es sich um lose Verbunde von meist autonomen Steuergeräten.
Jedoch zeichnet sich heute der Trend ab, mechanische und hydraulische Kopplungen durch fehlertolerante elektronische Systeme zu ersetzen. Man spricht von sogenannten X-by-Wire Anwendungen [3]. Prominente Beispiele sind Brake-by-Wire (die vollelektronische Bremse) und Drive-by-Wire (Lenkung). Die Software erhält Zugriff auf das gesamte Fahrverhalten. Dies ermöglicht neben der Unfallerkennung sogar eine aktive Unfallvermeidung. Allein der Wegfall der Lenksäule bedeutet einen Gewinn an Sicherheit. Statistisch gesehen gehört die Lenksäule zu den größten Gefahrenquellen für schwerste Verletzungen [4]. Weiterhin wird durch die Umwandlung der Aktionen des Fahrers in binäre Daten eine Korrektur menschlichen Fehlverhaltens durch Software ermöglicht. Auch die Fertigung wird vereinfacht und die Designfreiheit vergrößert.
DaimlerChrysler hat der Öffentlichkeit ein Drive-by-Wire-System zur Steuerung eines Fahrzeugs über eine Art Joystick vorgestellt. Der Stick übernimmt die Aufgaben von Lenkrad, Gas- und Bremspedal. Dieses System, ebenso wie alle anderen X-by-Wire Anwendungen, besteht aus mehreren Steuergeräten, die ein zuverlässiges, fehlertolerantes und koordiniertes Handeln in hoher zeitlicher Synchronität erfordern.
Diesen Aufgaben ist der OSEK/VDX-OS Standard allein nicht gewachsen. Die unverzichtbare globale Zeit für alle Steuergeräte wird genau so wenig bereitgestellt, wie die Fehlertoleranz und Vorhersehbarkeit sowohl der Kommunikation als auch der Taskausführung. Aus diesem Grund können Aktionen zweier verschiedener Steuergeräte nicht zu identischen Zeitpunkten garantiert werden. Die Folge war die Gründung der OSEKtime Working Group. Sie definierte ein streng zeitgesteuertes Echtzeitbetriebssystem (time triggered OSEK/OS [5, 6, 7]) mit der dazugehörigen fehlertoleranten Kommunikationsschicht (Fault-Tolerant Communication [8]), die unter anderem die globale Zeit zur Verfügung stellt.

# 2 OSEKtime

Unter OSEKtime versteht man einen Standard, der im Wesentlichen aus zwei Teilen besteht. Auf der einen Seite befindet sich das zeitgesteuerte Echtzeitbetriebssystem (time

triggered OSEK/OS), das für die Ausführung von Tasks und Interrupt Service Routinen (ISRs) sowie deren zeitliche Überwachung zuständig ist. Das beinhaltet vor allem das Scheduling und die Deadlineüberwachung. Auf der anderen Seite steht die Kommunikationsschicht FTCom (Fault-Tolerant Communication), die für die fehlertolerante Kommunikation zwischen den einzelnen Steuergeräten zuständig ist. Zu ihrer Verantwortung gehört aber auch das Bereitstellen der globalen Zeit. Offensichtlich werden erst durch das Zusammenwirken beider Teile die zuvor genannten Anforderungen erfüllt.

## 2.1 Task-Zustandsmodell von OSEKtime

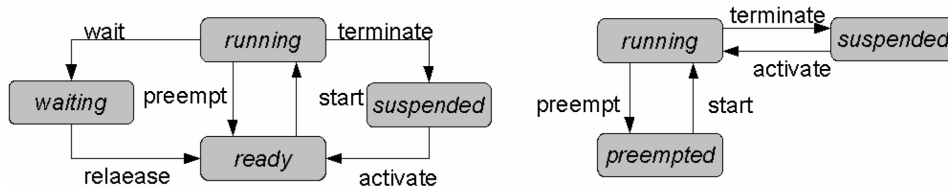



Abbildung 1: OSEK/VDX-OS und OSEKtime Task-Zustandsmodell

In OSEKtime können Tasks genau einen von drei Zuständen einnehmen. Alle inaktiven Tasks befinden sich im Zustand *suspended*. Wird eine Task aktiviert, so wechselt sie ohne Verzögerung direkt in den Zustand *running*. Dies ist bereits der erste Unterschied zu OSEK/VDX-OS, in dem die Task in den Zustand *ready* (hier *preempted*) wechselte. OSEKtime kennt aber keine Prioritäten. Tasks werden zu fest definierten Zeitpunkten gestartet und wechseln daher sofort in den Zustand *running*. Dies bedeutet natürlich, dass eine gerade laufende Task verdrängt (preempt) wird und somit in den Zustand *preempted* wechseln muss. Beendet sich eine im Zustand *running* befindliche Task selbst, so wechselt sie in den Zustand *suspended*. Wird keine andere Task aktiviert, so kann die zuvor verdrängte Task aus dem Zustand *preempted* in den Zustand *running* wechseln und ihre Ausführung fortsetzen. OSEKtime bietet keine Events, Counter, Alarme oder Ressourcen an, wie sie aus OSEK/VDX-OS bekannt sind. Events würden einen Wartezustand erfordern, wie ihn Extended Tasks in OSEK/VDX-OS bieten. Dies würde aber dem streng zeitgesteuerten und deterministischen Konzept von OSEKtime widersprechen. Daher hat der aus OSEK/VDX-OS bekannte Zustand *waiting* in OSEKtime keine Daseinsberechtigung.

## 2.2 Dispatching in OSEKtime

OSEKtime ist ein streng zeitgesteuertes Echtzeitbetriebssystem. Das bedeutet, dass bereits während des Entwicklungsprozesses in einer Ablauftabelle, der Dispatcher Table, festgelegt wird, wann eine Task ihre Ausführung startet. Wird zu diesem Zeitpunkt bereits eine Task ausgeführt, so wird sie verdrängt. Aus diesem Grund spricht man im Zusammenhang von OSEKtime vom Dispatcher anstelle von Scheduler. Ein Scheduler

trifft selbst die Entscheidung (meist nach Priorität) welche Task (aus dem Zustand *ready*) ausgeführt wird und erstellt somit den Zeitplan selbst. Ein Dispatcher startet nur Prozesse nach einer fest definierten Ablauftabelle und besitzt keine Entscheidungsgewalt. Nur der Dispatcher ist in der Lage Tasks zu aktivieren. Es gibt keine Events und auch keine andere Möglichkeit, durch die sich Tasks gegenseitig starten. Der Dispatcher selbst wird durch einen Timer-Interrupt gestartet, der im Wesentlichen der logischen internen Zeit entspricht, die mit der globalen Zeit synchronisiert sein sollte. Werden Tasks verdrängt, weil der Ausführungszeitpunkt anderer Tasks eingetroffen ist, so werden die Tasks auf einen Stack abgelegt (Stack Based Scheduling). Dieser Stack ist nach dem LIFO-Prinzip organisiert, d. h. es wird immer die zuletzt preempte Task runtergenommen und ausgeführt. Dieses Vorgehen ist in Hinsicht auf die Deadline-Einhaltung etwas verwunderlich. Andrerseits sollte die zum Entwicklungszeitpunkt festgelegte Dispatcher Table eine Deadline-Einhaltung vorsehen. Auf dem Boden des Stacks liegt die Leerlauf-Task ttIdleTask. Sie wird beim Systemstart als erste Task gestartet, später mehr dazu. Tasks können nur durch sich selbst beendet werden, d. h. sie laufen so lange, bis sie komplett abgearbeitet sind. Einen kompletten Ablauf der Dispatcher Table nennt man Dispatcher Round. Während einer solchen Runde kann eine Task mehrmals aktiviert werden. Dadurch wird eine Task künstlich priorisiert. Es ist allerdings zu beachten, dass die erste Ausführung beendet sein muss, bevor die nächste gestartet werden kann.
OSEKtime unterstützt das Konzept der Anwendungsmodi. Für jeden Modus wird eine eigenständige Dispatcher Table erstellt. Im laufenden Betrieb kann mit `SwitchAppMode()` die Ablauftabelle gewechselt werden. Der Wechsel selbst wird jeweils am Ende einer Dispatcher Round durchgeführt.

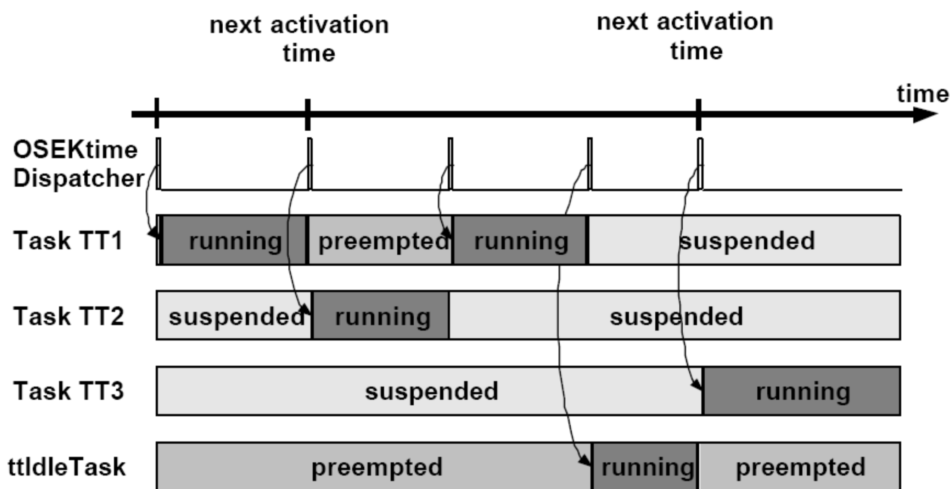



Abbildung 2: Beispiel eines Task-Ablaufs [5]

In Abbildung 2 sehen wir ein Beispiel eines Task-Ablaufs. Zu Beginn läuft die ttIdle-Task die verdrängt wird sobald der Dispatcher die Task TT1 aktiviert. Bereits beim nächsten Dispatching-Zeitpunkt wird die Task TT2 aktiv, die die nicht vollständig abgearbeitete Task TT1 verdrängt. Bei der nächsten Dispatcheraktivierung ist die Task

TT2 vollständig abgearbeitet und die Task TT1 kann ihre Ausführung fortsetzen. Beim darauf folgenden Dispatching ist Task TT1 abgearbeitet und keine andere Task muss laut Dispatcher Table aktiviert werden. Aus diesem Grund wird vom Boden des Stacks die ttIdleTask geholt und zur Ausführung gebracht. Bereits beim nächsten Dispatching wird diese von der Task TT3 zurück auf den Stack verdrängt.

## 2.3 Interrupt-Verarbeitung

In OSEKtime und OSEK/VDX-OS wird zwischen maskierbaren und nicht-maskierbaren Interrupt-Service-Routinen (ISRs) unterschieden. Maskierbare ISRs werden vom Betriebssystem abgefangen und kontrolliert abgearbeitet. Nicht-maskierbare ISRs laufen am Betriebssystem vorbei und lassen sich somit nicht beeinflussen. In OSEK/VDX-OS können Interrupts laufende Tasks grundsätzlich jederzeit unterbrechen. Dies würde in OSEKtime die Einhaltung von Deadlines stark erschweren. Aus diesem Grund wird in der Dispatcher Table ein Zeitfenster definiert, in dem Interrupts genau einmal auftreten dürfen. Dies kann in der Art geschehen, dass bestimmte Tasks bei ihrer Ausführung nicht unterbrochen werden. Interrupts werden vom Betriebssystem und nicht von der Anwendung enabled und disabled. Tritt laut Dispatcher Table der Zeitpunkt ein, an dem das Zeitfenster eines Interrupts beginnt, so wird dieser Interrupt enabled. Wird nun ein entsprechender Interrupt ausgelöst, so beginnt unverzüglich seine Abarbeitung. Was dort geschehen soll, legt der Entwickler mit dem Makro `ttISR` fest. Der Interrupt wird disabled und bleibt es bis zum nächsten Eintritt seines Zeitfensters. Während dieser Zeit kann der gleiche Interrupt nicht noch einmal eintreten. In Abbildung 3 ist dieses Vorgehen

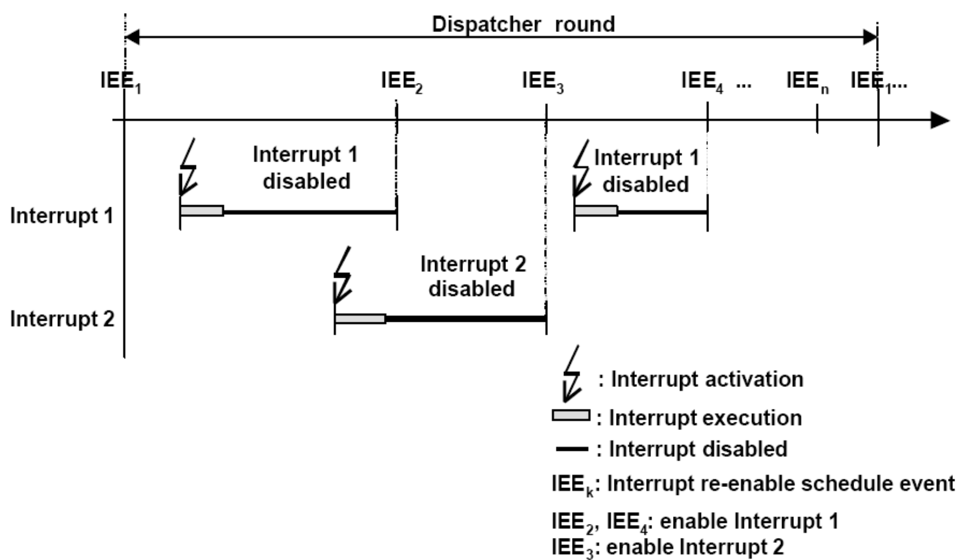



Abbildung 3: Interrupt-Verarbeitung [5]

dargestellt. Hierbei sollte beachtet werden, dass die Interrupt re-enable schedule events $IEE_k$ in der Dispatcher Table zur Entwicklungszeit definiert werden. Nichtsdestotrotz

sind nicht-maskierbare Interrupts möglich. Diese sollten jedoch vermieden werden, da sie ein Verschieben der OSEKtime-OS-dispatching-Interrupts bewirken.

## 2.4 Deadline-Überwachung

Die Dispatcher Table beinhaltet neben den Startzeitpunkte einer jeden Task auch die Deadline, also die Worst Case Execution Time (WCET). Diese besagt, wann eine Task spätestens vollständig abgearbeitet sein muss. Das Auftreten und Abarbeiten von möglichen Interrupt-Service-Routinen sollte in die Berechnung der Deadline einfließen. Die Deadline-Überwachung wird zur Laufzeit vom Dispatcher übernommen. Es gibt zwei Möglichkeiten, wann der Dispatcher die Deadline-Überschreitung kontrolliert. Bei der strikten Task-Deadline-Überwachung prüft der Dispatcher zum exakten Zeitpunkt der Deadline, ob sich die entsprechende Task noch in Ausführung befindet. Bei der nicht-strickten Task-Deadline-Überwachung wird zu einem „passenden“ Zeitpunkt, der zwischen der Deadline und dem Ende der Dispatcher Round liegen muss, geprüft. Welches Vorgehen gewünscht ist, wird in der Dispatcher Table mit dem Attribut Deadline-Monitoring festgelegt. In dem Beispiel aus Abbildung 4 wird die nicht-strikte Task-

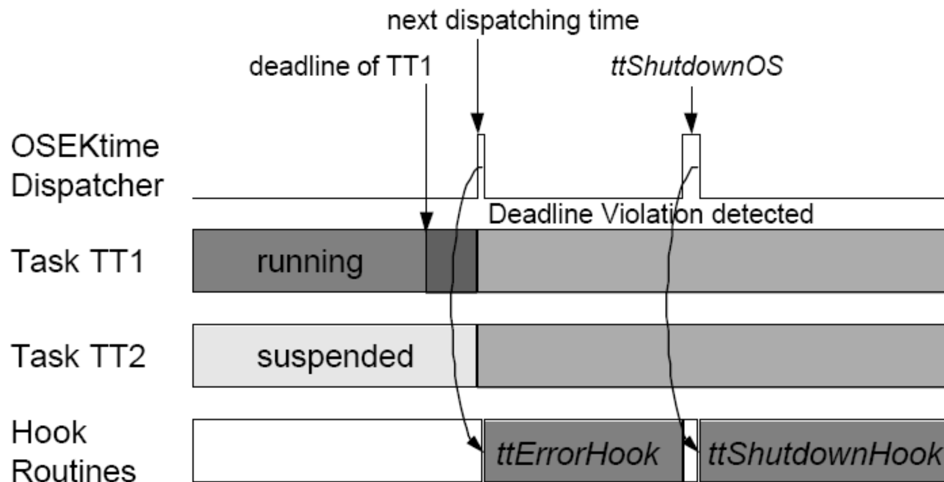



Abbildung 4: Deadline-Überwachung [5]

Deadline-Überwachung verwendet, da die Überschreitung erst beim nächsten Dispatching geprüft wird. Sobald eine Deadline-Verletzung festgestellt wird, ruft der Dispatcher die Funktion `ttErrorHook()` auf. Diese wird vom Entwickler implementiert und ermöglicht eine eigene Fehlerbehandlung. Nach Verlassen dieser Funktion beendet sich OSEKtime selbst und ruft die Funktion `ttShutdownHook()` auf. In dieser Funktion kann nun verblieben werden, um einen Neustart des Systems zu verhindern. Wird jedoch diese Funktion verlassen, so wird automatisch mit `ttStartOS()` ein Neustart initialisiert.

## 2.5 OSEK/VDX-OS als Subsystem

Die oben bereits erwähnte ttIdleTask wird als erste Task beim Systemstart aktiviert. Sie ist nicht in der Dispatcher Table eingetragen, wird also nicht periodisch neu gestartet und besitzt auch keine Deadline. Selbst beim Übergang zu einer neuen Dispatcher Round wird die Leerlauf-Task nicht neu gestartet. Beim aktivieren der ersten Task einer Dispatcher Round wird die ttIdleTask auf den Boden des Stack vom Zustand *preemted* gelegt. Aufgrund des LIFO-Prinzips wird die Leerlauf-Task genau dann aktiv, wenn zu einem Dipatching-Zeitpunkt keine neue Task aktiviert wird und die ttIdleTask sich als einzige Task auf dem Stack befindet.
Es liegt die Idee nahe, diese Leerlaufzeit für Funktionalitäten zu nutzen, die keinen harten zeitlichen Anforderungen unterliegen. Ein Beispiel hierfür wäre die Ansteuerung von Statusanzeigen oder ähnlichen. OSEKtime geht einen Schritt weiter. Während der ttIdleTask kann nämlich ein komplettes OSEX/VDX-OS ausgeführt werden. Dieses Subsystem kann komplexere nicht-zeitkritische Aufgaben übernehmen. Insbesondere können hier wesentliche Merkmale von OSEK/VDX-OS wie die Ereignisorientierung genutzt werden. Da es sich hierbei nur um ein Subsystem von OSEKtime handelt, ist klar, dass jegliche OSEK/VDX-Aktivität unterbrochen wird, sobald eine zeitgetriggerte Task vom Dispatcher aktiviert wird. OSEK/VDX Interrupts können nur in den Zeitabschnitten verarbeitet werden, in denen das Subsystem aktiv ist. Tritt während der Laufzeit einer time-triggerd Task ein OSEK/VDX IRQ auf, so wird dieser von OSEKtime entsprechend verzögert an das Subsystem weitergereicht. Die meisten Interrupt-Aktivitäten werden in der Praxis ohnehin durch OSEKtime-ISRs behandelt. Abbildung 5 zeigt ein Beispiel

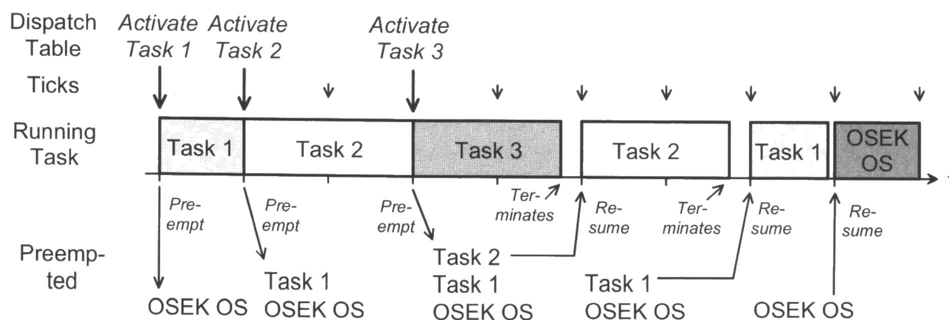



Abbildung 5: Beispiel eines Task-Ablaufes inkl. LIFO-Stack und Subsystem [7]

eines Task-Ablaufs bei OSEKtime. Dabei ist vor allem der LIFO-Stack des Zustands *preemted* schön dargestellt. Auffallend sind jedoch die Lücken, die zwischen Task 3 und Task 2 bzw. Task 2 und Task 1 entstanden sind. Dies passiert genau dann, wenn eine Task die ihr zugesprochene WCET nicht voll ausschöpft und bereits vor dem nächsten Dispatching sich selbst beendet. Diese dynamischen Idle-Zeiten könnten ebenfalls von dem Subsystem genutzt werden. Dieses Vorgehen wurde jedoch nicht spezifiziert.

# 3 OSEKtime FTCom

Mit OSEKtime FTCom bezeichnet man die fehlertolerante Kommunikationsschicht von OSEKtime. FTCom steht für Fault-Tolerant Communication. Die Spezifikation wurde ebenso wie die von OSEKtime im Juli 2001 fertig gestellt und liegt seitdem unverändert vor. Sie abstrahiert von den Eigenheiten eines konkreten Kommunikationsprotokolls und stellt stattdessen eine standardisierte Schnittstelle für den Austausch von Nachrichten zwischen Tasks auf eventuell verschiedenen Steuergeräten zur Verfügung. Die beiden Hauptziele sind die Fehlertoleranz und die Verfügbarkeit einer globalen Zeit.
Ein weiteres Entwurfsziel war die Robustheit gegen Fehler. Das Babbling Idiot Problem nahm eine zentrale Rolle ein. Dabei handelt es sich um ein defektes Steuergerät, das kontinuierlich sinnlose Informationen auf den Bus schreibt und somit einen Zusammenbruch der Kommunikation verursacht. Zur Lösung des Problems kann ein Buswächter vor jeden Netzadapter geschaltet werden, der sicher stellt, dass ein Steuergerät nur zu zuvor festgelegten Zeitpunkten schreibend auf den Bus zugreifen kann. Innerhalb dieser Zeitfenster können zwar immer noch Störungen auftreten, die restliche Kommunikation wird störungsfrei garantiert.

## 3.1 FTCom Schichtenmodell

Die fehlertolerante Kommunikationsschicht FTCom ist selbst in die in Abbildung 6 dargestellten Schichten unterteilt. Der Kommunkationskontroller soll unter anderem feh-

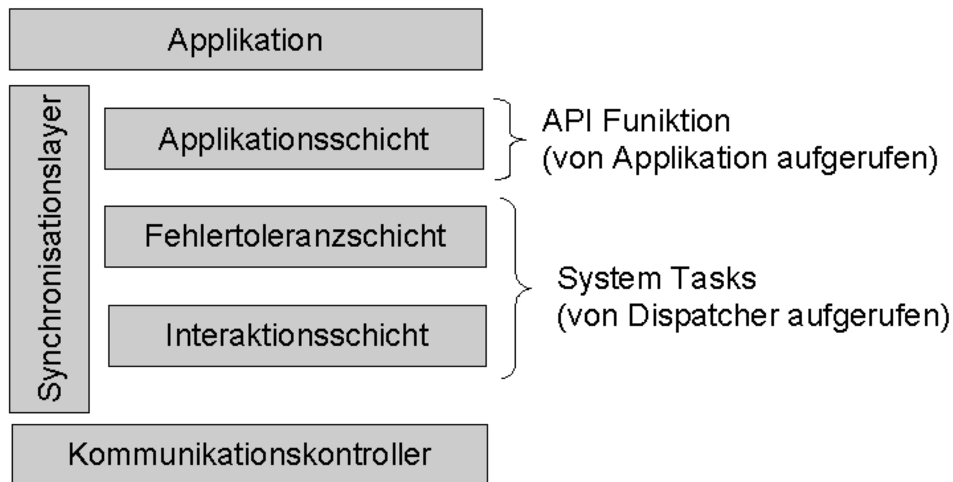


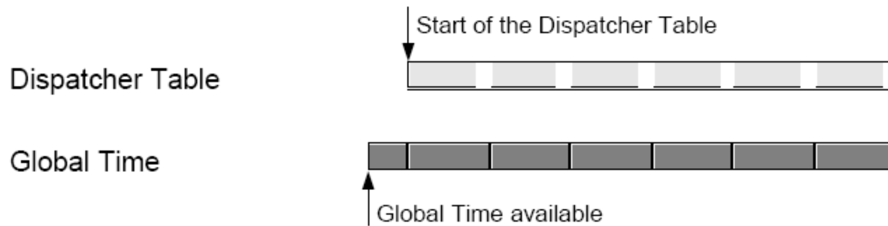

Abbildung 6: FTCom Schichtenmodell

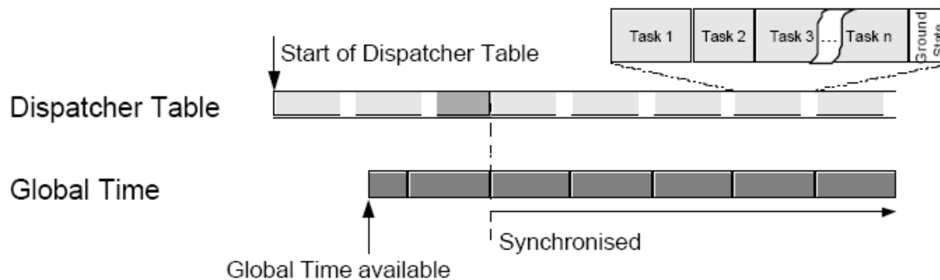
lende, verfrühte sowie verspätete Nachrichten erkennen und entsprechend handeln. Die Erkennung von korrupten Daten teilt er sich mit der Fehlertoleranzschicht.
Die Interaktionsschicht sorgt beim Sender für die Verpackung von mehreren Nachrichten in einzelne Botschaften. Weiterhin wandelt es die Byte Ordnung des Steuergerätes in die Byte Ordnung des Kommunikationsmediums um. Auf der Empfängerseite werden die

Nachrichten aus den empfangenen Botschaften extrahiert und anschließend in die Byte Ordnung des Steuergerätes umgewandelt.
Die Fehlertoleranzschicht baut auf der Interaktionsschicht auf und kümmert sich um die fehlertolerante Kommunikation. Dies wird vor allem durch das redundante Übertragen von Nachrichten realisiert. Auf der Senderseite wird eine Applikationsnachricht repliziert, sodass mehrere Instanzen dieser Nachricht entstehen. Die Instanzen werden dann redundant in verschiedenen Botschaften, über verschiedene Kanäle und zu verschiedenen Zeiten übertragen. Auf der Empfängerseite werden die Instanzen auf der Interaktionsschicht extrahiert und an die Fehlertoleranzschicht weitergereicht. Nun muss aus den vielen Instanzen eine Nachricht entstehen, die der Applikationsschicht bereitgestellt wird. Dazu wird ein vorher festgelegter Reduktionsalgorithmus verwendet. Werden konstante Fehler im Wertebereich angenommen, so müssen mindestens 3 oder eine größere ungerade Anzahl an Instanzen übermittelt werden. Als Reduktionsalgorithmus kann das Majoritätsvotum eingesetzt werden. OSEKtime FTCom bietet einige vordefinierte Reduktionsalgorithmen. Es besteht aber auch die Möglichkeit, eigene Algorithmen zu definieren.
Die Applikationsschicht bietet der Applikation drei Funktionen zur Nachrichtenübertragung, nämlich zum Versenden (`ttReciveMessage`), zum Empfangen (`ttSendMessage`) und zum Invalidieren (`ttInvalidateMessage`) von Nachrichten.
Während die Funktionen der Applikationsschicht von der Applikation aufgerufen werden, wird die Funktionalität der Fehlertoleranzschicht und der Interaktionsschicht vor dem Applikationsentwickler versteckt und durch FTCom Systemtasks bewerkstelligt. Wenn die Applikation FTCom-Funktionalität benötigt, dann ruft das Betriebssystem die FTCom-Funktionen auf, nicht die Applikation selbst. FTCom wird also durch eigenständige Tasks implementiert, die ein Kommunikationsschedule erfordern.
Der Synchronisationslayer stellt die globale Zeit zur Verfügung. Wann dies geschieht, wird im folgenden Abschnitt beschrieben.

## 3.2 Synchronisation der Systemzeit

Die globale Zeit wird über den Synchronisationslayer der fehlertoleranten Kommunikationsschicht zur Verfügung gestellt. Die Synchronisation erfolgt sowohl beim Systemstart als auch im Normalbetrieb zyklisch am Ende der Dispatcher Round. Der Ground State ist der Zeitabschnitt zwischen zwei Dispatcher Runden. Seine Dauer ist variabel und wird für die Synchronisation verlängert bzw. verkürzt. Die Synchronisation wird durch das Setzen der lokalen Zeit auf die globale Zeit durchgeführt und erfolgt im Ground State. Die Zeitsynchronisation über das Bussystem selbst, z.B. die verwendeten FlexRay-Mechanismen, ist nicht spezifiziert. Die Anwendung kann mit `ttGetOSSyncStatus()` den aktuellen Status der Synchronisation abfragen. `ttGetGlobalTime()` liefert hingegen die globale Zeit. Die Funktion `ttSyncTimes()` wird vom Synchronisationslayer aufgerufen, um die Drift zwischen der lokalen und globalen Zeit zu berechnen. Diese Differenz wird, wie bereits erwähnt, dazu verwendet, den Ground State zu erweitern oder zu verringern. Abbildung 7 stellt die drei Szenarien dar, die sich beim Systemstart oder dem Verlust der globalen Zeit zur Laufzeit ergeben.



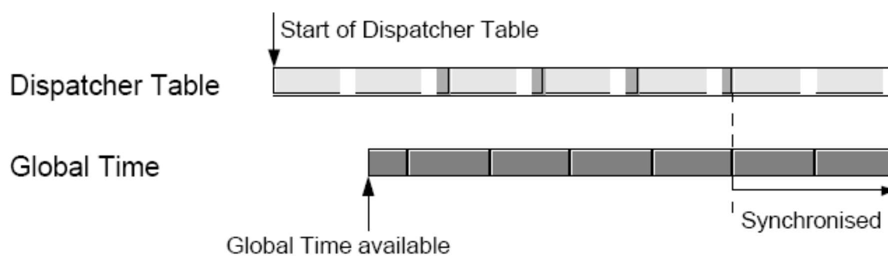




Abbildung 7: Start-up und Synchronisation [5]

**Synchronous Start-up:** Das Steuergerät beginnt die Ausführung der Dispatcher Round erst, wenn die globale Zeit zur Verfügung steht.
**Asynchronous Start-up - hard:** Das Steuergerät beginnt direkt mit der Ausführung der Dispatcher Round. Sobald die globale Zeit zur Verfügung steht und die Drift zwischen der lokalen und globalen Zeit bekannt ist, wird am Ende der Dispatcher Round gewartet. Dabei wird die Dauer des Ground State entsprechend manipuliert, sodass die folgende Dispatcher Round synchron startet.
**Asynchronous Start-up - smooth:** Hier wird wie oben asynchron mit der Ausführung gestartet. Die Anpassung der lokalen an die globale Zeit findet jedoch nicht wie oben schlagartig (hard) statt. Es wird ein weicher (smooth) Übergang verwirklicht, in dem nach jeder Dispatcher Round die Drift etwas verkleinert wird. Es findet also eine schrittweise Anpassung statt.

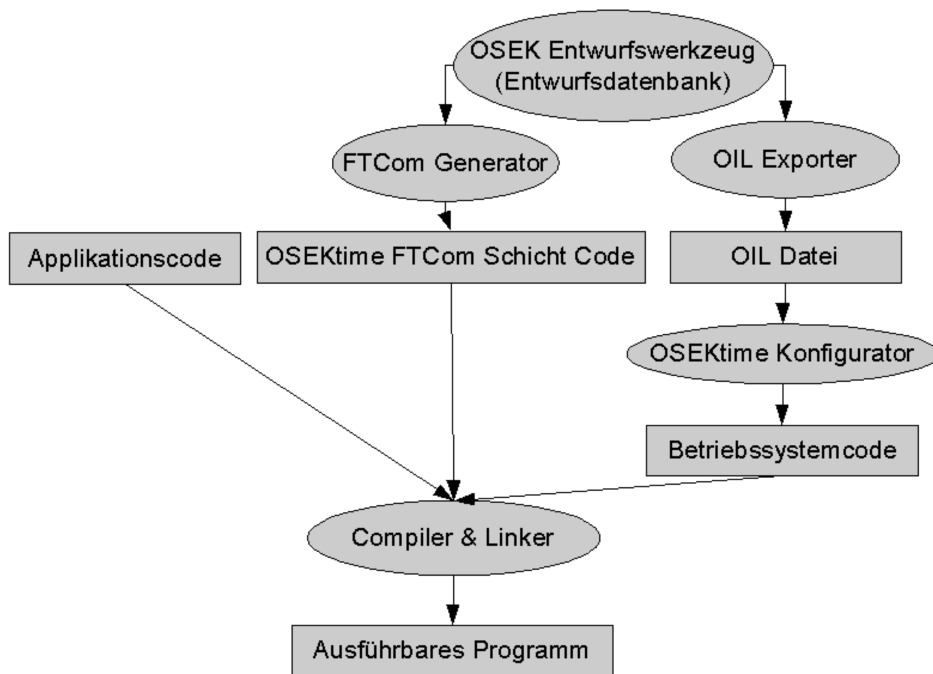



Abbildung 8: Toolkette [9]

# 4 Toolkette

Mit Hilfe der Toolkette wird aus dem ersten Entwurf ein ausführbares Programm erstellt, das die Applikationen, das Betriebssystem und die Kommunikationsschicht enthält. Mit einem OSEK-Entwurfswerkzeug wird die Entwurfsdatenbank verwaltet. Darin sind unter anderem Informationen über die Nachrichten, die zwischen den einzelnen Steuergeräten ausgetauscht werden, sowie Zeitpunkte, wann ein Nachrichtenaustausch passiert, enthalten. Der FTCom Generator erstellt aus diesen Informationen die fehlertolerante Kommunikationsschicht. In der Entwurfsdatenbank sind ebenfalls Informationen über die Startzeitpunkte der Applikations- und Systemtasks abgelegt. Diese werde zunächst durch den OIL Exporter in eine Datei im OSEK Implementation Language Format geschrieben. Die OIL-Datei enthält auch eine Beschreibung des Kernels hinsichtlich der von der Applikation benötigten Betriebsmittel. Der OSEKtime Konfigurator generiert aus diesen Informationen einen auf die beschriebenen Erfordernisse optimierten Kernel sowie die zugehörige Dispatcher Table. Zum Schluss werden der Applikationscode die generierte Kommunikationsschicht und das Betriebssystem zu einem ausführbaren Programm kompiliert und gelinkt. Jetzt muss das Programm nur noch auf das Steuergerät eingespielt werden.

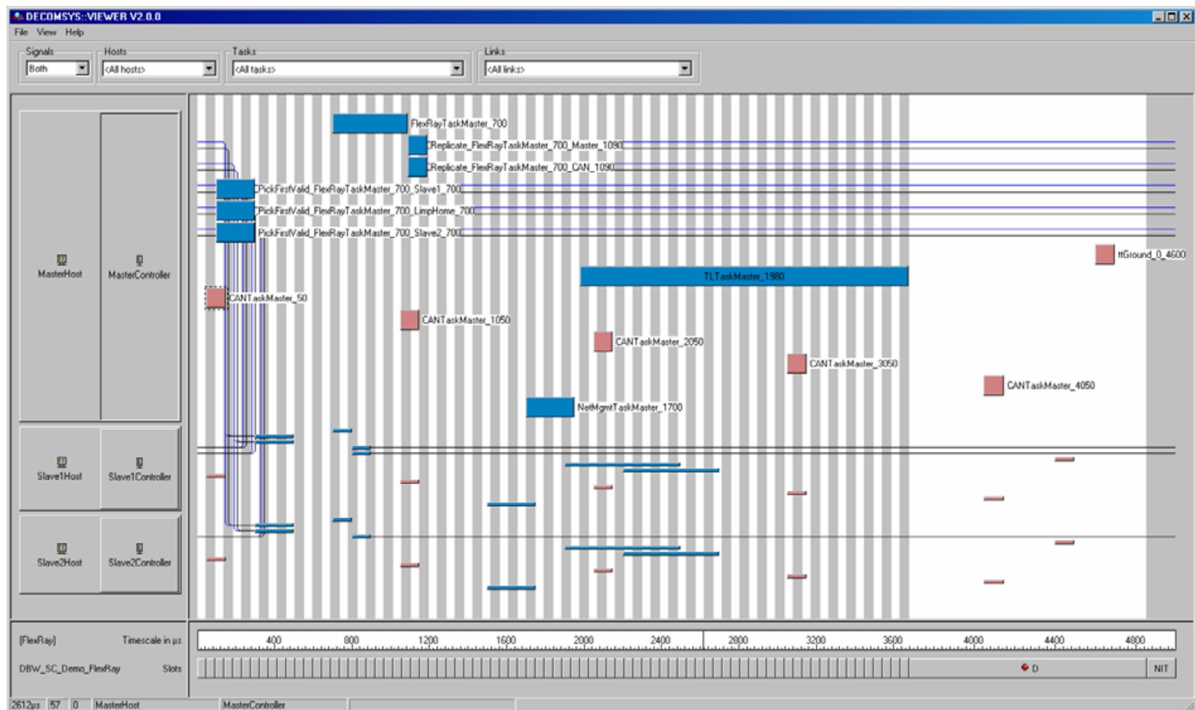



Abbildung 9: Task Schedule und Kommunikationsschedule in TimeCore [10]

# 5 TimeCore

TimeCore von der DECOMSYS GmbH bietet eine für den Systemdesigner unterstützende Werkzeugkette sowie einen integrierten Entwurfsprozess an. Abbildung 9 zeigt ein Werkzeug der TimeCore Werkzeugkette, das eine Task Schedule sowie das zugehörige Kommunikationsschedule darstellt. Die Tasks sind hierbei durch rote und blaue Blöcke abgebildet. Die Linien zwischen den Blöcken repräsentieren Signale, die zwischen den Tasks ausgetauscht werden.

# 6 Zusammenfassung

OSEKtime ist aufgrund der Zeitsteuerung grundsätzlich anders aufgebaut als OSEK/VDX. Das time-triggered OSEK/OS erzielt Determinismus auf Ebene des Betriebssystems. Die fehlertolerante Kommunikationsschicht FTCom erzielt bei der Kommunikation zwischen den einzelnen Steuergeräten ein deterministisches Zeitverhalten. Erst gemeinsam wird Determinismus im gesamten verteilten System erreicht. Die Fehlertoleranz und die Verfügbarkeit der globalen Zeit tragen entscheidend dazu bei, OSEKtime für sicherheitsrelevante Applikationen wie X-by-Wire zu favorisieren. So ist OSEKtime nicht als Ersatz oder als Alternative zum OSEK/VDX-Standard zu sehen, sondern vielmehr als eine Ergänzung für spezielle Einsatzzwecke.

# Literatur


[1] *OSEK OS. Version 2.2.3.* February 2005. – frei erhältlich unter http://portal.osek-vdx.org/files/pdf/specs/os223.pdf

[2] Schoof, Dr. J.: *OSEKtime - Standard für zeitgesteuerte Betriebssysteme.* – frei erhältlich unter http://joscho.de/PUB/ei2001.pdf

[3] Schoof, Dr. J.: *OSEKtime - Betriebssystem-Standard für X-by-Wire.* – frei erhältlich unter http://www.joscho.de/PUB/BSsymp03.pdf

[4] Fastnacht, Felix ; Schoof, Dr. J.: *OSEKtime - die ersten praktischen Erfahrungen mit dem neuen Standard.* – frei erhältlich unter http://joscho.de/PUB/ei2002b.pdf

[5] *OSEK/VDX time triggered operating system. Version 1.0.* July 2001. – frei erhältlich unter http://portal.osek-vdx.org/files/pdf/specs/ttos10.pdf

[6] Homann, Matthias: *OSEK; Betriebssystem-Standard für Automative und Embedded Systems.* 2. überarbeitete Auflage. Bonn : mitp-Verlag, 2005

[7] *Kapitel 7.* In: Zimmermann, Werner ; Schmidgall, Ralf: *Bussysteme in der Fahrzeugtechnik; Protokolle und Standards.* 2. Auflage. Wiesbaden : Vieweg, 2007

[8] *OSEK/VDX fault tolerant communication. Version 1.0.* July 2001. – frei erhältlich unter http://portal.osek-vdx.org/files/pdf/specs/ftcom10.pdf

[9] Galla, Thomas M. ; Olig, Jochen: *OSEKtime - Eine Softwareplattform für sicherheitsrelevante verteilte Applikationen im Automobil.* – frei erhältlich unter http://members.chello.at/thomasmgalla/papers/2003-08-21_Paper.pdf

[10] Galla, Thomas M.: *TimeCore - Software Komponenten und Entwurfswerkzeuge für sicherheitsrelevante verteilte Applikationen im Automobil.* – frei erhältlich unter http://members.chello.at/thomasmgalla/papers/2004-04-24_Paper_Elektronik_Automotive.pdf