Software-Standards: OSEK und HIS

7

7.1 Einführung

Die immer schwerer zu beherrschende Komplexität der Fahrzeugelektronik mit den stark vernetzten Teilsystemen einerseits, sowie der Wunsch nach Kostensenkung andererseits fördern das Bestreben nach Standardisierung. Der Grundgedanke ist, eine Komponentenarchitektur einzuführen, bei der durch eine saubere Definition von Schnittstellen und Kommunikationsmechanismen zwischen den Komponenten eine einfache Austauschbarkeit und Wiederverwendbarkeit sichergestellt werden kann. Während dabei in der Vergangenheit vorzugsweise Hardwarekomponenten verstanden und daher beispielsweise Gehäusebauformen, Sensorschnittstellen u. ä. betrachtet wurden, konzentriert man sich heute verstärkt auf Softwarekomponenten [1].

Begonnen haben diese Aktivitäten bei den Bussystemen und Kommunikationsprotokollen für die On- und Off-Board-Kommunikation bereits vor 1990, wie in den vorigen Kapiteln dargestellt wurde. Um 1995 wurde dann unter der Bezeichnung *Offene Systeme für die Elektronik im Kraftfahrzeug/Vehicle Distributed Executive* (OSEK/VDX) ein Vorschlag für einen Betriebssystemstandard vorgelegt. Seit etwa 2000 wird unter den Stichworten *Hersteller-Initiative Software* (HIS), *Automotive Open Systems Architecture* (AUTOSAR) und *Japanese Automotive Software Platform Architecture* (JASPAR) in verschiedenen Arbeitskreisen an der Standardisierung der Steuergeräte-Softwarearchitektur und der Entwicklungs- und Testmethoden gearbeitet.

Die Aktivitäten aller derartigen Initiativen beginnen in der Regel innerhalb einer kleinen Gruppe von Fahrzeugherstellern unter Mitwirkung ihrer wichtigsten Zulieferer. Danach schließen sich andere Fahrzeughersteller und Zulieferer und schließlich auch Hersteller von Entwicklungswerkzeugen an. Die Arbeitsgebiete der verschiedenen Initiativen überlappen sich oft stark und die meisten Firmen finden sich nach kurzer Zeit in allen diesen Gremien in unterschiedlichen Rollen wieder. Während manche an einer echten Standardisierung und schnellen Fortschritten interessiert sind, arbeiten andere mit, um Standardisierungsbestrebungen, die dem eigenen Produktportfolio gefährlich werden könn-

W. Zimmermann und R. Schmidgall, *Bussysteme in der Fahrzeugtechnik*, ATZ/MTZ-Fachbuch, 331
DOI 10.1007/978-3-658-02419-2_7, © Springer Fachmedien Wiesbaden 2014

ten, zu verzögern oder um zu sehen, woran der Wettbewerb arbeitet. Zielsetzungen, Stand und Qualität der erzielten Resultate lassen sich für Außenstehende und oft auch für die Beteiligten nur schwer beurteilen, doch können die Erfahrungen mit OSEK/VDX wohl auf die neueren Bestrebungen übertragen werden. Als problematisch erweist sich generell, dass die Beteiligten bei den Standards kooperieren sollen, bei den tatsächlich entwickelten Produkten aber konkurrieren. Im Rahmen der Standardisierung entsteht daher meist nur ein Spezifikationspapier, allenfalls noch eine Referenzimplementierung für einige Teilaspekte, aber kein einheitliches, sofort einsetzbares Produkt. Dazuhin haben die Firmen unterschiedliche Zeitvorstellungen. Während die einen in aller Ruhe den Standard diskutieren und mit der Produktentwicklung erst einmal abwarten, erfolgt bei anderen aufgrund von feststehenden Serienterminen die Produktentwicklung parallel zur Standardisierung, oft schneller als der Fortschritt im Standardisierungsgremium und muss Rücksicht auf proprietäre, bereits existierende Lösungen nehmen. So gibt es heute mindestens ein Dutzend von kommerziell verfügbaren Betriebssystemen, die alle OSEK/VDX *kompatibel* sind. Doch praktisch jedes dieser Betriebssysteme verfügt über herstellerspezifische Erweiterungen, die Spezifikationslücken schließen, Unklarheiten oder alternative Möglichkeiten unterschiedlich interpretieren und immer nur für einen Teil der eingesetzten Hardwareplattformen verfügbar sind. Alle bieten eine proprietäre Entwicklungsumgebung, ohne die komfortables Entwickeln nicht sinnvoll möglich ist, und sind – allen gegenteiligen Bemühungen zum Trotz – von untereinander nicht völlig kompatiblen Übersetzungswerkzeugen (Compiler, Linker) abhängig, so dass der Umstieg zwischen auf dem Papier OSEK/VDX-kompatiblen Betriebssystemen immer noch eine aufwendige Angelegenheit bleibt, die gut überlegt sein will.

Das vorliegende Kapitel versucht nicht, dieses im Fluss befindliche Gebiet in allen Einzelheiten darzustellen, sondern lediglich einen Überblick über die sich abzeichnende Grundstruktur zu geben. Die grobe Softwarearchitektur ist im Abb. 7.1 dargestellt:

- Die für den Betrieb des Fahrzeugs notwendige Software soll funktionsorientiert gegliedert werden, z. B. in Module für die Leerlaufregelung oder die Fahrgeschwindigkeitsregelung, aber auch in Module, die einen komplexen Sensor, z. B. den Drehratensensor eines ESP-Systems, oder ein komplexes Stellglied bedienen. Diese Module sollen als eigenständige Softwarekomponenten realisiert werden, so dass sie unabhängig von den anderen Komponenten funktional sind. Durch eine saubere Definition ihrer Schnittstellen und der Kommunikationsmechanismen, mit denen die Komponenten Daten austauschen, soll die Austauschbarkeit der Komponenten erreicht werden. Die Komponenten der so genannten *Fahr-* oder *Anwendungssoftware* werden als Baukasten betrachtet, mit dem ein Kfz-Hersteller Ausstattungsvarianten implementiert und sich so vom Wettbewerb differenziert. Durch die Komponentenarchitektur wollen die Kfz-Hersteller erreichen, dass sie diese Komponenten teilweise selbst entwickeln können und teilweise von Zulieferern unabhängig von der Hardware des Steuergerätes zukaufen und kombinieren können.

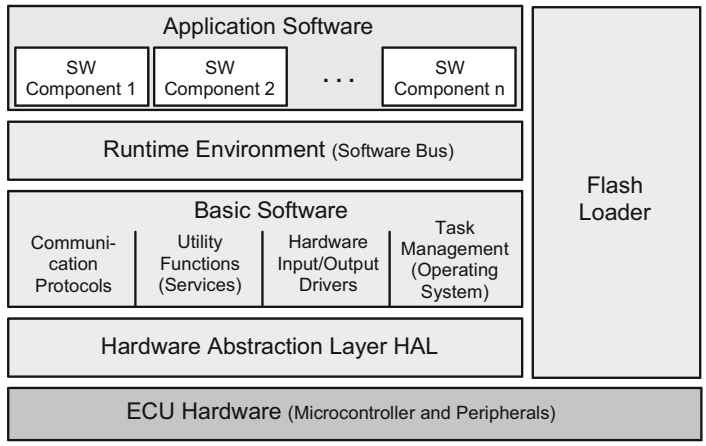

Abb. 7.1 Software-Grundarchitektur zukünftiger Steuergeräte nach AUTOSAR/HIS

- Die für den Datenaustausch zwischen den einzelnen Softwarekomponenten zuständige Schnittstelle ist die Laufzeitumgebung (*Run Time Environment*). Der Entwurf eines derartigen *Software-Bussystems* ist einer der Schwerpunkte der AUTOSAR-Aktivitäten.
- Die *Basis-Software* umfasst alles, was man bei üblichen Computern gemeinhin als *Betriebssystem* bezeichnet. Der etwas ungewöhnliche Begriff *Basis-Software* ist hier notwendig, weil die OSEK/VDX-Spezifikation lediglich das *Task Management*, d. h. die Ablaufsteuerung und Zuteilung von CPU-Rechenzeit definiert, diese Teilfunktionalität aber als *Betriebssystem* bezeichnet. Andere klassische Betriebssystemaufgaben wie die Schnittstelle zur Hardware, Kommunikationsprotokolle oder die Bereitstellung von Dienstprogrammen beispielsweise zur Verwaltung des Fehlerspeichers werden von OSEK/VDX nicht oder nur unvollständig abgedeckt, sollen aber zukünftig ebenfalls strukturiert und vereinheitlicht werden.
- Die reale Hardware des Steuergerätes wird durch die Hardwareabstraktionsschicht (*Hardware Abstraction Layer HAL*) von der darüber liegenden Software entkoppelt. Ändert sich die Anschlussbelegung des Gerätes oder die logische Zuordnung der Mikrocontrollerperipherie, so muss im Idealfall nur diese Schicht angepasst werden. Eng damit verbunden sind die in die darüber liegende Basis-Software integrierten Ein-/Ausgabe-Funktionen (*Input/Output Driver*), welche den Zugriff auf die Hardware weiter abstrahieren. Dabei ist leider nicht eindeutig, wo genau die Trennlinie zwischen den beiden Ebenen verläuft. Der Entwurf einheitlicher Ein-/Ausgabe-Funktionen ist einer der Schwerpunkte der HIS- bzw. AUTOSAR-Aktivitäten. Diese Softwareteile sollen in der Regel vom Hersteller der Steuergeräte-Hardware bzw. des eingesetzten Mikrocontrollers oder Peripheriebausteins geliefert werden.
- Um den Inhalt des Programm- und Datenspeichers nicht nur beim Gerätehersteller, sondern auch beim Fahrzeughersteller und in der Werkstatt ändern zu können und

so eine individuelle Anpassung an das Fahrzeug bzw. nachträgliche Änderungen und Erweiterungen der Software durchführen zu können, muss der Steuergerätespeicher (Flash-ROM) im eingebauten Zustand von außen neu programmiert werden können. Die in HAL und Basis-Software, z. B. im UDS-Protokollstapel, integrierten Funktionen würden dies zwar prinzipiell erlauben, können aber nicht verwendet werden, wenn diese gegebenenfalls selbst ersetzt werden sollen. Daher wird eine eigenständige, kleine Softwarekomponente, der *Flash-Lader* vorgesehen, die unabhängig vom Rest der Steuergerätesoftware funktionsfähig ist und nur für diese Aufgabe eingesetzt wird. Entsprechende Standardisierungsaktivitäten erfolgen vor allem im Rahmen von HIS. Das Innenleben eines derartigen Flash-Laders wird in Kap. 8 beschrieben.

Überträgt man diese Architektur auf die Welt von Personalcomputern, so entsprächen die Komponenten der Anwendungssoftware Programmen wie Word, Excel, Powerpoint, Internet Explorer usw. Die Laufzeitumgebung findet ihre Entsprechung in *Middleware*-Mechanismen wie COM/DCOM, ActiveX oder CORBA, mit denen etwa eine Excel-Tabelle in eine Powerpoint-Präsentation eingebettet wird, sowie dem *Application Programming Interface* des Betriebssystems, z. B. der Win32-API oder den Linux bzw. Posix System Calls. Die Basis-Software entspricht dem, was man von Personalcomputern als Kern-Betriebssystem kennt, also Windows, Linux oder Solaris, und dort für das Starten von Programmen, die Zuteilung von Rechenzeit und Speicherplatz, die Dateiverwaltung und die Netzanbindung zuständig ist. Die Hardware-Ein-/Ausgabefunktionen und die Hardware-Abstraktionsschicht ähneln den Hardwaretreibern (*Device Driver*) und dem BIOS (*Basic Input Output System*) eines PCs. Auch den Flash-Lader findet man bei PCs wieder, wenn man ein BIOS-Update für die fest eingebauten Komponenten auf der Hauptleiterplatte des Rechners durchführt, die bereits unmittelbar beim Einschalten funktionsfähig sein müssen, damit der Rechner das eigentliche Betriebssystem von der Festplatte laden und starten kann.

7.2 OSEK/VDX

Den Kern des OSEK/VDX-Systems (Abb. 7.2) bildet *OSEK OS* (*Operating System*), das ein ereignisgesteuertes Echtzeit-Multitasking-Betriebssystem mit Möglichkeiten zur Task-Synchronisation sowie Ressourcenverwaltung festlegt [2, 3]. OSEK geht von einem verteilten System aus und definiert eine Interaktionsschicht *OSEK COM* (*Communication*), die sowohl den Datenaustausch zwischen den Tasks innerhalb eines Steuergerätes (interne Kommunikation) zulässt als auch die Kommunikation zu Tasks in anderen Steuergeräten über ein Bussystem (externe Kommunikation). Für die Überwachung und Verwaltung eines solchen Bussystems wurde *OSEK NM* (*Network Management*) definiert. Im Hinblick auf eine effiziente Implementierung mit geringen Anforderungen an Rechenleistung und Speicherplatzbedarf wird das gesamte System weitgehend statisch in der Entwicklungsphase konfiguriert. Die OSEK-Spezifikationen umfassen sowohl die Definition des Konzepts

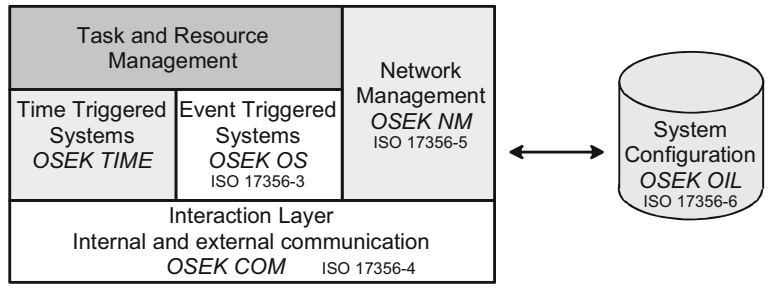

Abb. 7.2 Grundkomponenten des OSEK/VDX-Systems

und der zugehörigen Mechanismen als auch ein *Application Programming Interface* (API), d. h. eine Programmierschnittstelle in der Programmiersprache C. Für die Beschreibung der Konfiguration eines Systems wurde *OSEK OIL* (*Implementation Language*) definiert. Aus einer derartigen Konfigurationsdatei kann ein geeignetes Entwicklungswerkzeug dann weitgehend automatisch die Task-Verwaltungsstrukturen der Steuergeräte-Software erzeugen.

Um die Skalierbarkeit von einfachen Anwendungen beispielsweise in der Karosserieelektronik mit 8 bit Mikrocontrollern bis hin zu komplexen Motorsteuergeräten mit 32 bit Mikrocontrollern sicherzustellen, sind die drei Hauptbestandteile OS, COM und NM jeweils eigenständig definiert, d. h. sie könnten theoretisch auch unabhängig voneinander eingesetzt werden, und sind in sich in mehrere optionale Ausbaustufen untergliedert, bei OSEK als *Conformance Classes* bezeichnet.

Die zunächst als Industriestandard entstandenen Spezifikationen befinden sich mittlerweile als ISO 17356 in der offiziellen Standardisierung. Die Spezifikationen für die OSEK/VDX-Teilkomponenten sind historisch gewachsen und existieren in verschiedenen, nicht notwendigerweise kompatiblen Versionen. Einen Überblick über den jeweils aktuellen Stand gibt das so genannte *OSEK Binding Dokument* (ISO 17356-1 und -2), die Weiterentwicklung und Anpassung des Systems erfolgt mittlerweile im Rahmen von AUTOSAR.

Viele Details des Systems sind eng an die Mechanismen des ebenfalls ereignisgesteuerten CAN-Busses angelehnt, da CAN zum Zeitpunkt der OSEK-Definition das einzige verfügbare Kfz-Bussystem mit ausreichender Leistungsfähigkeit für verteilte Systeme war. Die Diskussion über zeitgesteuerte Bussysteme wie FlexRay für X-by-Wire-Anwendungen hat auch vor dem Betriebssystem nicht Halt gemacht und dort zur Definition der zeitgesteuerten Betriebssystemvariante *OSEK Time* und einer Kommunikationsschicht mit besserer Fehlertoleranz *OSEK FTCOM* (*Fault Tolerant Communication*) geführt. Beide haben allerdings als reale Produkte kaum Verbreitung gefunden und sind mittlerweile in den weiterführenden AUTOSAR-Konzepten aufgegangen.

Bei den heute in Steuergeräten noch üblichen Mikrocontrollern existieren nur wenige in die CPU-Architektur direkt integrierte Mechanismen, um einen zuverlässigen Betrieb auch

bei fehlerhafter Software sicherzustellen. Solche Konzepte sind bei den leistungsfähigeren und wesentlich teureren Mikroprozessoren, wie sie in Arbeitsplatzrechnern eingesetzt werden, seit Längerem bekannt. Dort wird hardwaremäßig eine Trennung von Betriebssystem und Anwendungen mit unterschiedlichen Speicheradressräumen (*Memory Protection*) erzwungen, so dass eine Fehlfunktion eines einzelnen Anwendungsprogramms andere Anwendungsprogramme und das Betriebssystem in der Regel nicht beeinträchtigt. Zukünftig werden Mikrocontroller für Kfz-Anwendungen solche Konzepte unterstützen und könnten von der *OSEK Protected OS* Erweiterung genutzt werden, die im Rahmen des HIS-Arbeitskreises definiert wurde.

Um die Entwicklung von Testwerkzeugen zu erleichtern, liegt mit OSEK ORTI (*OSEK Run Time Interface*) eine Spezifikation für die Schnittstelle zwischen der OSEK/VDX-Laufzeitsoftware im Steuergerät und externen Debuggern, Emulatoren und anderen Software-Testwerkzeugen vor.

7.2.1 Ereignisgesteuerter Betriebssystemkern OSEK/VDX OS

Wie die meisten Betriebssysteme setzt OSEK/VDX voraus, dass die von der Steuergeräte-Software zu bearbeitende Aufgabenstellung in verschiedene Teilaufgaben unterteilt wird, die mehr oder weniger unabhängig voneinander gleichzeitig bearbeitet werden müssen, z. B. die Regelung der Turboladerdrehzahl eines Motors bei gleichzeitiger Steuerung der Einspritzmenge und Ansteuerung der Zündung. Die zu einer solchen Teilaufgabe gehörenden Programmteile werden unter dem Begriff *Task* zusammengefasst und als Einheit vom Betriebssystem verwaltet. Innerhalb einer Task erfolgt der Programmablauf dabei sequenziell. Die einzelnen Tasks dagegen laufen scheinbar parallel zueinander ab. Da das Steuergerät aber üblicherweise nur über einen einzigen Mikrocontroller verfügt und dieser nicht mehrere Teilaufgaben gleichzeitig bearbeiten kann, wird die Illusion der parallelen Bearbeitung dadurch erreicht, dass das Betriebssystem in rascher Abfolge zwischen den einzelnen Teilaufgaben umschaltet (*Multitasking*). Die dafür im Betriebssystem verantwortliche Komponente wird als *Scheduler* bezeichnet. Das OSEK/VDX-Modell sieht vor, dass sich alle im Steuergerät definierten Tasks in einem der drei bzw. vier im Abb. 7.3 dargestellten Zustände befinden. Zu einem betrachteten Zeitpunkt kann genau eine einzige Task tatsächlich vom Mikrocontroller ausgeführt werden. Diese und nur diese Task befindet sich im Zustand *Running*. Alle Tasks, die auch gerne laufen würden, befinden sich im Zustand *Ready*. Diejenigen Tasks, die gerade überhaupt nicht benötigt werden, z. B. die Funktionen des UDS-Protokolls, solange das Steuergerät nicht mit einem Diagnosetester verbunden ist, sind im Zustand *Suspended*. Andere Teilaufgaben, die zwar prinzipiell gerade laufen könnten, aber auf das Eintreffen eines äußeren Ereignisses warten müssen, z. B. die Funktionen des UDS-Protokolls während einer Diagnosesitzung, wenn sie auf das Eintreffen der nächsten UDS-Botschaft warten, befinden sich im Zustand *Waiting*.

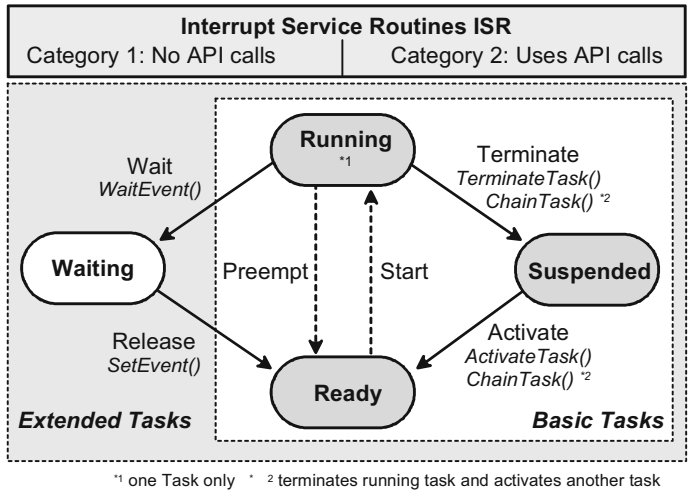

Abb. 7.3 Zustandsmodell für OSEK/VDX Tasks und zugehörige API-Funktionen

OSEK/VDX unterscheidet dabei Tasks, die nie auf äußere Ereignisse warten wollen, d. h. den Zustand *Waiting* überhaupt nicht benötigen (*Basic Tasks*), und solche Tasks, die auch den Zustand Waiting verwenden (*Extended Tasks*).

Die in Abb. 7.3 mit durchgezogenen Linien gezeichneten Zustandsübergänge werden von der gerade laufenden Task für sich selbst (*Terminate*, *Wait*) oder für andere Tasks (*Activate*, *Release*) durch den Aufruf der entsprechenden Betriebssystemfunktionen bewirkt. Die Entscheidung dagegen, welche Task als nächste laufen darf (*Start*) bzw. ob eine gerade laufende Task verdrängt wird (*Preempt*), d. h. erzwungenermaßen pausieren muss, trifft allein der Scheduler des Betriebssystems auf Basis von Task-Prioritäten. Dabei wird jeder Task in der Entwicklungsphase eine zur Laufzeit nicht veränderliche Prioritätsstufe fest zugeordnet. Die niedrigste Prioritätsstufe ist 0. Die Obergrenze ist implementierungsabhängig, vorgeschrieben sind bei BCC (siehe unten) mindestens 8 bzw. bei ECC mindestens 16 Prioritätsstufen. Wenn der Scheduler aufgerufen wird, wählt er aus der Menge aller gerade im Zustand *Ready* befindlichen Tasks diejenige Task aus, die die höchste Priorität besitzt. Wenn die Priorität dieser Task höher ist als die Priorität der gerade laufenden Task, wird die laufende Task verdrängt (*Preempt*) und die neue Task gestartet. Die verdrängte Task erhält den Zustand *Ready* und ihr aktueller Status (*Task Context*), d. h. CPU-Registerinhalte, lokale Variable usw., wird so gespeichert, dass die Task zu einem späteren Zeitpunkt an der unterbrochenen Stelle fortgesetzt werden kann. Für den Fall, dass mehrere der bereiten Tasks dieselbe Prioritätsstufe haben, die Auswahl nach der Prioritätsstufe allein also nicht eindeutig ist, verwendet das Betriebssystem für jede Prioritätsstufe zusätzlich eine Warteschlange, wobei dann die an der vordersten Stelle der Warteschlange stehende Task zum Zug kommt. Wenn eine Task aktiviert wird oder aus dem Zustand *Wait* kommt,

Tab. 7.1 Konformitätsklassen (*Conformance Classes*) für die Implementierung

		Zustand Wait zulässig?	
		Nein: Nur Basic Task	Ja: Basic und Extended Tasks
Mehrere Tasks je Prioritätsstufe und/oder mehrere gleichzeitige Aktivierungen derselben Task?	Nein	**BCC1** Basic Conformance Class 1	**ECC1** Extended Conformance Class 1
	Ja	**BCC2** Basic Konformen Class 2	**ECC2** Extended Konformen Class 1

wird sie ganz hinten in der Warteschlange, beim Verdrängen dagegen an vorderster Stelle eingereiht.

Da die Realisierung der Warteschlangen für Tasks mit gleicher Prioritätsstufe sowie des Wartezustands selbst relativ aufwendig ist, schlägt OSEK/VDX verschiedene Implementierungsstufen, die so genannten Konformitätsklassen (Tab. 7.1) vor. Der Speicherbedarf einer Implementierung, die nur die einfachste Klasse BCC1 unterstützt (keine Task verwendet den Zustand *Wait*, alle Tasks haben unterschiedliche Prioritätsstufen) ist deutlich geringer als derjenige für einen Scheduler, der sämtliche Möglichkeiten (ECC2) bereitstellt.

Neben der Priorität einer Task wird für jede Task festgelegt, unter welchen Umständen der Scheduler aktiv wird (*Scheduling Policy*). Im Fall einer Task, die nur nicht-präemptives Scheduling (*Non Preemptive Scheduling*) zulässt, wird der Scheduler nur dann aktiv, d. h. es kann ein Wechsel zu einer anderen Task nur erfolgen, wenn die gerade laufende Task sich mit `TerminateTask()` bzw. `ChainTask()` selbst beendet, mit `WaitEvent()` in den Wartezustand übergeht oder mit `Schedule()` selbst den Scheduler aufruft (*kooperatives Multitasking*). Im allgemeineren Fall einer Task mit präemptivem Scheduling (*Full Preemptive Scheduling*) dagegen kann der Scheduler die Task an jeder beliebigen Stelle unterbrechen, wenn eine Task mit höherer Priorität in den Zustand *Ready* versetzt wird (Präemptives Multitasking). Dies kann durch `ActivateTask()`, `SetEvent()` oder das Freigeben einer Ressource von der gerade laufenden Task selbst veranlasst worden sein, oder durch ein äußeres Ereignis wie einen aktivierten Alarm, eine eintreffende Nachricht oder einen Interrupt bewirkt werden, die nachfolgend noch beschrieben werden. Tasks, die präemptives Scheduling verwenden, müssen damit rechnen, jederzeit unterbrochen zu werden. Wenn es innerhalb einer solchen Task Passagen gibt, bei denen eine Unterbrechung nicht zulässig ist, z. B. um die Datenkonsistenz beim Lesen oder Schreiben eines Speicherblocks sicherzustellen, kann die selbstständige Aktivierung des Schedulers mit Hilfe von `GetResource(RES_SCHEDULER)` und `ReleaseResource(RES_SCHEDULER)` zeitweise blockiert werden. Wenn alle Tasks nur das nicht-präemptive Scheduling verwenden, wird der Speicherplatzbedarf des Schedulers wiederum verringert. Meist allerdings werden beide Formen gemischt (*Mixed Preemptive*), wobei länger laufende Tasks in der Regel das präemptive Scheduling

Tab. 7.2 Programmstruktur einer OSEK/VDX Task und einer ISR in C

```
DeclareTask(myTaskName);

TASK(myTaskName)
{  do
   {  . . .                          // Programmcode der Task
   } while (!abortCondition);
   TerminateTask();                  // Alternativ: ChainTask(. . .)
}

ISR(myIsrName)
{  . . .
}
```

verwenden, damit Task höherer Priorität nicht unnötig lange bis zur Ausführung warten müssen. Bei kurzen Tasks dagegen kann es sinnvoll sein, das nicht-präemptive Scheduling zu verwenden, weil eine präemptive Taskumschaltung zu einer höherprioren Task unter Umständen ohnehin nicht viel schneller erfolgt als wenn die niedrigerpriore Task zu Ende ausgeführt wird, oder wenn die Task eine Aufgabe erledigt, die keinesfalls unterbrochen werden darf.

Neben dem Unterbrechen der laufenden Task durch den Scheduler gibt es den Mechanismus der Interrupts, die in der Regel durch Hardwaresignale im Mikrocontroller oder seinen Peripheriebausteinen ausgelöst werden und unabhängig vom Scheduler zum Ausführen vordefinierter Anwenderfunktionen, der so genannten *Interrupt Service Routinen ISR*, führen (Abb. 7.3). Interrupts unterbrechen daher auch Tasks mit nicht-präemptivem Scheduling. Innerhalb der *Interrupt Service Routine* (ISR Category 2) darf eine eingeschränkte Auswahl von OSEK-API-Funktionen, z. B. `SetEvent()` oder `ActivateTask()`, verwendet werden. Der Scheduler wird aber erst dann wieder aktiviert, wenn die *Interrupt Service Routine* endet und die normale Taskausführung wieder aufgenommen wird. Einfache *Interrupt Service Routinen* (ISR Category 1), welche selbst keine API-Funktionen aufrufen wollen, aktivieren bei Beendigung den Scheduler nicht, so dass die unterbrochene Task einfach direkt fortgesetzt wird. Falls es innerhalb normaler Tasks notwendig ist, Unterbrechungen durch Interrupts zu verhindern, können die API-Funktionen `DisableXInterrupts()`, `EnableXInterrupts()` bzw. `SuspendXInterrupts()`, `ResumeXInterrupts()` verwendet werden, mit denen alle (X = All) bzw. nur die Interrupts der Kategorie 2 (X = OS) zeitweilig gesperrt werden.

Programmtechnisch ist eine Task in der Implementierung eine einfache, parameterlose C-Funktion, deren Prototyp mit dem Makro `DeclareTask()` und deren Funktionskopf mit `TASK()` definiert wird (Tab. 7.2). Die Funktion enthält meist eine Endlosschleife mit Abbruchbedingung, in der die eigentlichen Aufgaben der Task ausprogrammiert werden, und endet mit der Funktion `TerminateTask()`. Innerhalb der Funktion können beliebige andere Funktionen aufgerufen werden. Die Festlegung des Scheduling-Verfahrens (*Full-* bzw. *Non-Preemptive*), die Angabe, ob es sich um eine Basic oder Extended Task handelt, sowie weiterer Attribute erfolgt in einer später (Tab. 7.3) noch zu beschreibenden OIL-

Konfigurationsdatei. Dabei wird auch angegeben, ob eine Task nach dem Starten des Multitaskings sofort automatisch in den Zustand *Ready* versetzt werden soll (*AUTOSTART*) oder ob die Task zunächst im Zustand *Suspended* verbleibt und erst durch eine andere Task zur Laufzeit aktiviert wird. *Interrupt Service Routinen* sind einfache C-Funktionen, die mit dem Makro `ISR()` definiert werden.

Die Aufrufstruktur aller OSEK OS API-Funktionen ist einheitlich gehalten. Betriebssystemobjekte wie Tasks, Events usw. werden über Kennziffern identifiziert, wobei der Programmierer symbolische Konstanten (in C: #define) verwendet, die über die OIL-Konfigurationsdatei festgelegt werden. Ausgangsparameter werden über Call by Reference übergeben. Der eigentliche Rückgabewert der Funktion ist ein Statuscode, der mitteilt, ob die Funktion erfolgreich ausgeführt wurde. Die Fehlerüberprüfungen zur Laufzeit sind im Hinblick auf Zeit- und Speicherbedarf allerdings einfach gehalten.

Die Synchronisation zwischen verschiedenen Tasks oder zwischen einer *Interrupt-Service-Routine* und einer Task erfolgt beispielsweise über *Events*. Ein *Event* ist ein binäres Signal, das bei der Konfiguration des Systems einer *Extended Task* zugeordnet wird. Diese und nur diese Task kann mit `WaitEvent()` auf eines oder mehrere der ihr zugeordneten *Events* warten, bis eine beliebige andere *Extended* oder *Basic Task* oder eine *Category 2 Interrupt Service Routine* eines dieser *Events* mit `SetEvent()` setzt. Daraufhin wird die wartende Task in den Zustand *Ready* versetzt und kann, abhängig von ihrer Priorität, weiterlaufen. Die Task kann das *Event* dann mit `ClearEvent()` zurücksetzen und später erneut auf das Ereignis warten. Ist ein *Event* bereits gesetzt, wenn eine Task auf das *Event* warten will, läuft die Task sofort weiter. Mit `GetEvent()` kann jederzeit abgefragt werden, ob ein eigenes oder ein fremdes *Event* bereits gesetzt ist oder nicht.

Mittels *Events* können auch periodisch wiederkehrende Aufgaben erledigt werden. *Events* alleine bieten allerdings keine Möglichkeit, sich selbstständig periodisch zu setzen. Hierfür bietet OSEK/VDX als Erweiterung das Konzept der *Alarme*. *Alarme* werden wie *Events* während der Entwicklungsphase in der Konfigurationsdatei definiert. Dort wird festgelegt, ob das Betriebssystem beim Auftreten eines *Alarms* ein bestimmtes *Event* setzen, eine bestimmte Task aktivieren oder eine Anwenderfunktion (*Callback Routine*) aufrufen soll. Das Auslösen des *Alarms* erfolgt durch einen Zähler (*Counter*), der dem *Alarm* zugeordnet wird. Zähler können die in Mikrocontrollern oft zu findenden Zeitgeber (*Timer*) sein, die Signale zu bestimmten absoluten oder relativen Zeitpunkten und/oder mit definierter Periodendauer liefern, oder *Capture-Compare*-Kanäle, die beispielsweise die Impulse eines inkrementellen Drehzahlsensors erfassen und damit die Bestimmung von Drehzahl und Winkellage der Kurbel- oder Nockenwelle eines Motors erlauben. Aus Sicht von OSEK/VDX ist ein Zähler ein abstraktes Betriebssystemobjekt, das aus einem aktuellen Zählerstand sowie drei vorkonfigurierten Parametern besteht. Der abstrakte Zähler beginnt bei 0 aufwärts zu zählen, bis er seinen maximalen Zählerstand (`MAXALLOWEDVALUE`) erreicht, dann springt er auf 0 zurück und zählt erneut aufwärts. Die minimale Wiederholperiode `(MINCYCLE)` gibt an, um wie viel Schritte der Zähler mindestens weiterzählen muss, bevor erneut ein *Alarm* ausgelöst wird. Der dritte und gegebenenfalls weitere Parameter (`TICKSPERBASE`) sind implementierungsabhängig,

Tab. 7.3 Beispiel einer OIL-Konfigurationsdatei

```
OIL_VERSION = "2.0";

// Include OS implementation specific definitions
#include <osekOsVendor.oil>

CPU myApplication   // *** Configuration for one microcontroller ****
{
    OS exampleOsekOS {  // *** General operating system parameters **
        STATUS = EXTENDED;   // Uses ext. API function return values
        ErrorHook = FALSE;   // Specify which hook routines are used
        StartupHook = FALSE;
        . . .
    };

    TASK myFirstTask {  // *** Define a task ************************
        TYPE = EXTENDED;     // This is an extended task
        PRIORITY = 10;       // Task priority is 10
        SCHEDULE = FULL;     // Task uses full preemptive scheduling
        ACTIVATION = 1;      // No multiple task activation
        AUTOSTART = TRUE;    // Task will be in state Ready when ...
                             // ... multitasking starts
        EVENT = triggerEvent;// This task uses event triggerEvent
        RESOURCE= myResource;// This task uses resource myResource
        MESSAGE = myMsg1;    // This task uses message myMsg1
    };

    TASK mySecondTask { // *** Define another task ******************
        TYPE = EXTENDED;
        PRIORITY = 11;       // Task priority is 11
        SCHEDULE = FULL;
        ACTIVATION = 1;
        AUTOSTART = FALSE;   // Task will be in state Suspended ...
                             // ... when multitasking starts
        EVENT = NONE;        // Task does not use any events
        RESOURCE = myResource;
        MESSAGE = myMsg2, myMsg3;
    };

    RESOURCE myResource;// *** Define a resource *********************

    EVENT triggerEvent{ // *** Defines an event **********************
        MASK = AUTO;
    };

    COUNTER myCounter { // *** Defines a counter *********************
        MAXALLOWEDVALUE = 65535; // Parameters of the assoc. counter
        TICKSPERBASE = 100;
        MINCYCLE = 50;
    };

    ALARM myAlarm {     // *** Defines an alarm **********************
        COUNTER = myCounter;    // Alarm based on counter myCounter
        ACTION = ACTIVATETASK;  // When the alarm is signaled, it ...
        {   TASK = myFirstTask; //... activates task myFirstTask
        }
    };

    ISR myISR {         // *** Defines an interrupt service routine *
        CATEGORY = 2;
    };
};
```

da OSEK/VDX selbst keine API-Funktionen für den Zugriff auf die hinter dem Zähler stehende Hardwarebaugruppe des Mikrocontrollers bereitstellt. Reale Implementierungen verwenden diese Parameter, um die Taktfrequenz von Zeitgebern, Verteilerfaktoren und ähnliches anzugeben, und stellen Bibliotheksfunktionen für den Zugriff auf die entsprechende Mikrocontrollerhardware bereit. OSEK/VDX selbst erlaubt es, mit den API-Funktionen `SetRelAlarm()` und `SetAbsAlarm()` zur Laufzeit den Zählerstand zu ändern, bei dem der *Alarm* ausgelöst werden soll. Der Zählerstand kann sowohl als absoluter Wert als auch relativ zum aktuellen Zählerstand sowie als Wiederholperiode für periodische *Alarme* angegeben werden. Die Parameter und der aktuelle Zählerstand können mit `GetAlarmBase()` und `GetAlarm()` abgefragt, ein gestarteter Alarm mit `CancelAlarm()` vor dem Auslösen abgebrochen werden.

Um den Zugriff auf einen Speicherbereich, eine Hardwarekomponente oder ein anderes Objekt zwischen mehreren Tasks zu synchronisieren und einen konkurrierenden Zugriff zu verhindern, verwendet OSEK/VDX das Konzept der *Ressource*. Dasselbe Problem tritt auch bei Arbeitsplatzrechnern auf, wenn mehrere Programme gleichzeitig Dateien ausdrucken oder über das Netzwerk versenden wollen. Um den gleichzeitigen Zugriff auf den Drucker oder die Netzwerkkarte auszuschließen (*Mutual Exclusion*) setzen derartige Betriebssysteme so genannte *Semaphoren* (andere Bezeichnung: *Mutex*) ein. Die dabei entstehende Problematik soll anhand von Abb. 7.4 erläutert werden. Eine Task 1 niedriger Priorität läuft und belegt eine Ressource, z. B. den Drucker, wobei es dem Betriebssystem diese Belegung durch eine dem Drucker zugeordnete Semaphore mitteilt. Kurz danach wird eine hochpriore Task 3 aktiviert und verdrängt, da auch die Betriebssysteme von Arbeitsplatzrechnern das Multitasking wie OSEK/VDX mit Prioritäten abwickeln, sofort die Task 1. Nun will aber auch die Task 3 die bereits von Task 1 belegte Ressource verwenden und meldet dies dem Betriebssystem über dieselbe Semaphore. Da der Drucker aber bereits belegt und ein Abbruch mitten innerhalb des Druckvorgangs nicht sinnvoll möglich ist, versetzt das Betriebssystem die Task 3 trotz ihrer höheren Priorität in den Wartezustand und lässt die Task 3 erst dann weiterlaufen, wenn die niederpriore Task 1 den Drucker wieder freigibt. Die hochpriore Task 3 muss beim Zugriff auf eine gemeinsame Ressource also gegebenenfalls auch auf eine niederpriore Task warten. Dies ist prinzipiell nicht vermeidbar, weshalb gemeinsam genutzte Ressourcen von jeder Task stets nur so kurz wie irgendmöglich reserviert werden sollten. Leider muss die Task 3 aber nicht nur auf die Task 1 warten, die die gemeinsame Ressource gerade verwendet, sondern im ungünstigsten Fall sogar auf andere Tasks, deren Priorität niedriger ist als die Priorität von Task 3 aber höher als die von Task 1. Im Beispiel wird eine Task 2 mittlerer Priorität lauffähig, während Task 3 wartet, weil Task 1 die gemeinsame Ressource blockiert. Da die Task 2 eine höhere Priorität als Task 1 hat, hindert sie die Task 1 am Laufen. Task 3 muss daher zusätzlich noch auf die aus ihrer Sicht niedrigerpriore Task 2 warten, obwohl diese noch nicht einmal die gewünschte Ressource blockiert. Dieses als Prioritätsinversion bezeichnete Verhalten ist in Echtzeitsystemen, bei denen die zeitrichtige Abarbeitung von Funktionen sicherheitskritisch ist, unzulässig.

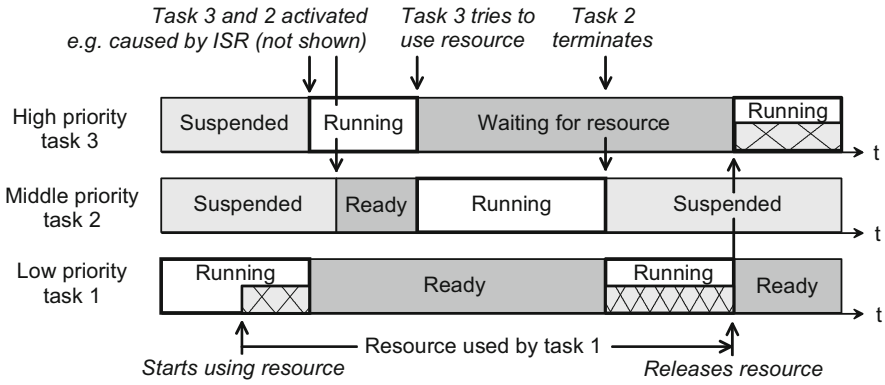

Abb. 7.4 Prioritätsinversion bei Systemen ohne dynamische Prioritätserhebung

OSEK/VDX löst das Problem ähnlich wie andere Echtzeitbetriebssysteme durch eine dynamische Prioritätsanhebung (*Priority Ceiling*). Nicht nur den Tasks sondern auch der gemeinsamen Ressource wird eine Priorität zugeordnet. Deren Wert wird mindestens so groß gewählt wie die Priorität der höchstprioren Task, die diese gemeinsame Ressource verwendet. Wenn nun eine Task niedriger Priorität mit `GetResource()` den Zugriff auf die Ressource reserviert, wird die Priorität dieser Task vom Betriebssystem selbstständig auf die höhere Priorität dieser Ressource angehoben und erst nach dem Freigeben mit `ReleaseResource()` wieder auf den niedrigeren Wert der Task abgesenkt (Abb. 7.5). Task 3 muss natürlich weiterhin warten, bis Task 1 die Ressource freigibt, aber Task 2 kann in dieser Zeit aufgrund der dynamisch erhöhten Priorität die Task 1 nicht mehr verdrängen und verzögert die Task 3 daher nicht noch weiter. Da das Warten der Task 3 auf die Ressource dabei im Zustand *Ready* erfolgen kann und der Zustand *Waiting* dafür gar nicht notwendig ist, kann das Ressourcen-Konzept auch mit den einfacheren *Basic Tasks* und nicht nur mit *Extended Tasks* angewendet werden. Definiert werden Ressourcen und ihre Prioritäten wiederum während der Entwicklungsphase in der Konfigurationsdatei. Während der Laufzeit muss die Anwendung lediglich die genannten beiden Funktionsaufrufe verwenden.

Dasselbe Konzept wird, wie oben schon dargestellt, auch angewendet, um den Aufruf des Schedulers zeitweilig zu verhindern, indem der Scheduler einfach ebenfalls als Ressource betrachtet und entsprechend von einer Task reserviert wird, die sich nicht durch eine andere Task unterbrechen lassen will.

Der Hochlauf eines Systems mit OSEK/VDX besteht in der Regel aus den in Abb. 7.6 dargestellten vier Schritten. Der eigentliche Betriebssystemstart erfolgt durch die API-Funktion `StartOS()`, nachdem der Mikrocontrollerkern, das Laufzeitsystem der verwendeten Programmiersprache (bei Kfz-Steuergeräten in der Regel C oder seltener C++) und die Hardware initialisiert wurden. Damit wird das Multitasking aktiviert, der Scheduler des Betriebssystems zum ersten Mal aufgerufen und die erste Task gestartet, die

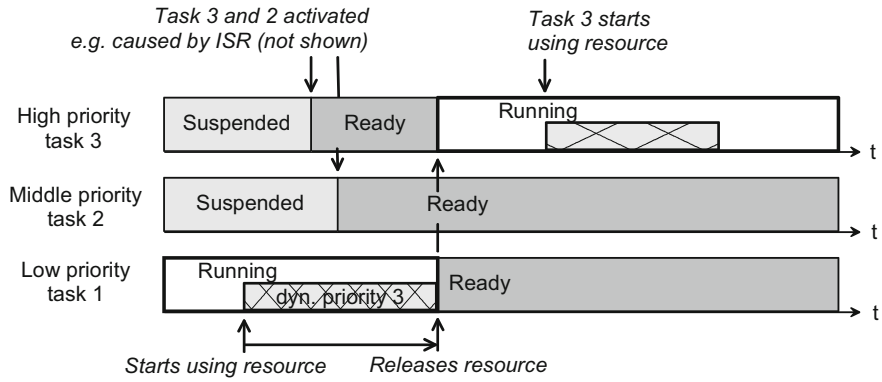

Abb. 7.5 Dynamische Prioritätserhöhung bei der Verwendung von Ressourcen bei OSEK (Annahme: Priorität der gemeinsamen Ressource sei 3)

grundsätzlich als *Autostart*-Task definiert sein muss. Die Funktion `StartOS()` kehrt erst dann zurück, wenn innerhalb einer Task die Funktion `ShutdownOS()` aufgerufen und dadurch das Multitasking beendet wird. Da Kfz-Steuergeräte ihre Funktion in der Regel nur beim Abstellen der Spannungsversorgung tatsächlich einstellen dürfen, wird das Multitasking in einem solchen Fall üblicherweise wieder gestartet, was im Abb. 7.6 durch die `while` Schleife angedeutet wird.

OSEK sieht die Möglichkeit vor, bei der Systemkonfiguration mehrere Anwendungsmodi zu definieren und den zu verwendenden Anwendungsmodus beim Aufruf von `StartOS()` als Parameter anzugeben. In der Konfiguration werden jede Task und alle übrigen Betriebssystemobjekte wie Events, Alarme usw. einem oder mehreren dieser Anwendungsmodi zugeordnet. Anwendungsmodi können beispielsweise der grundsätz-

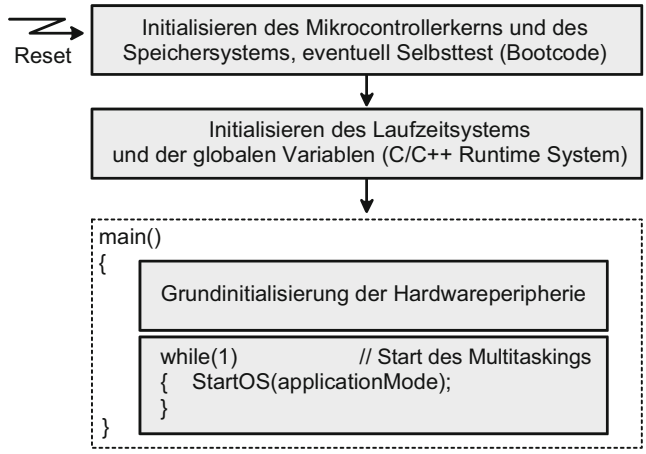

Abb. 7.6 Hochlauf des Systems

lich immer vorhandene Normalbetrieb (`OSDEFAULTAPPMODE`), ein Testmodus für den Gerätehersteller oder ein End-of-Line-Programmiermodus sein. Das Konzept des Anwendungsmodus erlaubt dem Betriebssystem auch, für jeden Anwendungsmodus eigene Tabellen für die internen Verwaltungsstrukturen zu erstellen und so Speicherplatz und Laufzeitbelastung zu optimieren. Außerdem hilft es den Systemgenerierungs- und Debuggingwerkzeug in gewissen Grenzen, um in der Entwicklungsphase zu erkennen, wenn ein Anwendungsprogramm Tasks oder ähnliches aktivieren will, die im aktuellen Anwendungsmodus keinen Sinn machen. Das Umschalten von einem Anwendungsmodus zu einem anderen bei laufendem Multitasking ist allerdings nicht möglich. Das Multitasking muss mit `ShutdownOS()` beendet und dann mit `StartOS()` im neuen Anwendungsmodus wieder gestartet werden. Für die Umschaltung zwischen internen Betriebsmodi der Anwendung wie normalem Fahrbetrieb und Notlaufbetrieb, bei denen eine vollständige Unterbrechung des Multitaskings in der Regel nicht zulässig ist, ist der OSEK-Mechanismus daher weniger geeignet.

Das OSEK-Entwicklungsmodell (Abb. 7.7) setzt voraus, dass das gesamte Betriebssystem, d. h. alle Tasks, Events, Alarme usw., mit allen Parametern in der Entwicklungsphase statisch konfiguriert wird und sich zur Laufzeit nicht ändert. Die Konfiguration wird in einer Textdatei mit Hilfe der so genannten *OSEK Implementation Language* (OIL) beschrieben (Tab. 7.3), wobei die Anbieter von OSEK/VDX-Betriebssystemen in der Regel ein *Konfigurationswerkzeug* mit grafischer Oberfläche mitliefern, so dass diese Datei nicht manuell mit einem Texteditor erstellt werden muss. (Abb. 7.7). Aus dem Quellcode der Anwendung, der Konfigurationsdatei und dem generischen OSEK/VDX-Betriebssystemcode, der vom Anbieter oft als Bibliothek mit vorkompilierten Modulen geliefert wird, erstellt dann ein Generierungswerkzeug den Quellcode für das auf die konkrete Anwendung zugeschnittene Steuergeräteprogramm. Optional werden außerdem Dateien im *OSEK Run Time Interface* (ORTI) Format generiert, mit deren Hilfe das Testen des Systems mit einem symbolischen Debugger erleichtert wird. Bei der Implementierung der generischen OSEK OS Bibliothek und beim *Generierungswerkzeug* hat der Betriebssystemanbieter jede Freiheit, solange er die in den OSEK-Spezifikationen spezifizierten Schnittstellen und die OIL-Syntax unterstützt. Die Anbieter nehmen in der Regel proprietäre Erweiterungen vor, z. B. herstellerspezifische Attribute in der Konfigurationsdatei, und stellen zusätzliche API-Funktionen zur Verfügung, was den Einsatz im Einzelfall komfortabler machen, die Portabilität aber erheblich einschränken kann.

Das OSEK-Entwicklungsmodell setzt außerdem voraus, dass der zeitrichtige Ablauf im System, die Speicherauslastung in Worst Case Situationen usw. während der Entwicklungsphase mit Methoden und Werkzeugen, zu denen die OSEK-Spezifikationen keinerlei Vorgaben machen, ausreichend simuliert und getestet wurden. Das OSEK-Betriebssystem selbst enthält mit Ausnahme einiger so genannter *Hook*-Funktionsaufrufe keine Unterstützung für die Überwachung und Behandlung von Überlast- oder Fehlersituationen. Die *Hook*-Funktionen können vom Anwendungsprogramm bereitgestellt werden und werden vom Betriebssystem zu definierten Zeitpunkten aufgerufen, wenn das Multitasking gestartet bzw. beendet wird (*Startup* bzw. *Shutdown Hook*), wenn der Scheduler eine Task

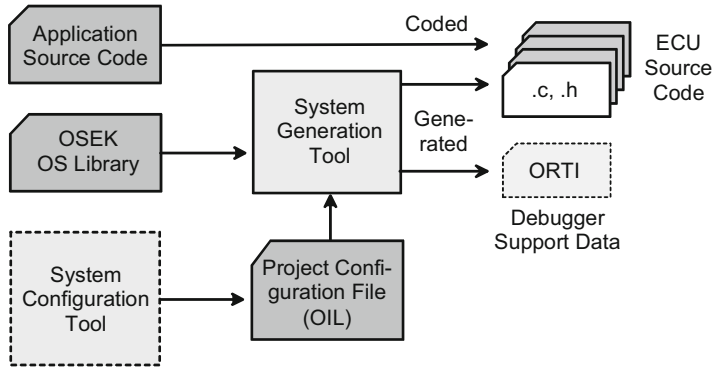

Abb. 7.7 OSEK/VDX Entwicklungsmodell

verdrängt (*Post Task Hook*), bevor er eine neue Task startet (*Pre Task Hook*), oder wenn der Aufruf einer API-Funktion mit einer Fehlermeldung endet (*Error Hook*). Glücklicherweise ergänzen viele Betriebssystemhersteller diese nur rudimentären Möglichkeiten durch ausgefeiltere Debugging- und Simulationsmöglichkeiten wenigstens für die Entwicklungsphase.

7.2.2 Kommunikation in OSEK/VDX COM

OSEK COM spezifiziert eine botschaftsbezogene Kommunikation zwischen Tasks innerhalb eines Steuergerätes (interne Kommunikation) oder über ein Bussystem zwischen mehreren Steuergeräten (externe Kommunikation). Dabei wird versucht, die Anwendungs-Task von der Kommunikation zu entkoppeln, so dass es aus Sicht der Tasks, von der unterschiedlichen Übertragungsdauer abgesehen, keine Rolle spielt, ob sie innerhalb eines Geräts oder über ein externes Bussystem miteinander kommunizieren (Abb. 7.8). Die OSEK COM Spezifikation wurde mehrfach stark überarbeitet. Im Laufe der Zeit verschwanden die Definition eines eigenen Transportlayers und anderes, für das mittlerweile eigenständige Normen vorliegen, und die Spezifikation wurde beim Übergang von Version 2.x zu 3.x deutlich übersichtlicher. Parallel dazu entwickelten sich die kommunikationsbezogenen Teile der OIL-Spezifikation, da die Kommunikation ebenfalls mit OIL konfiguriert wird. OSEK COM wird sinnvollerweise in Verbindung mit OSEK OS eingesetzt, obwohl es sich theoretisch um eine eigenständige Spezifikation handelt. Transportschicht und Bussystem, die unterhalb der COM Interaktionsschicht liegen, sind (mittlerweile) beliebig, die Mehrzahl der Anbieter von OSEK-Komponenten unterstützt CAN (ISO 15765-2 und 11898), teilweise LIN und zukünftig wohl FlexRay.

Eine Task, die eine Botschaft versenden will (*Sender-Task*), verwendet die API-Funktion `SendMessage()`, der als Parameter ein Zeiger auf die zu sendenden Daten übergeben wird. Die Funktion kopiert die Botschaft in einen internen Puffer, stößt bei externer Kom-

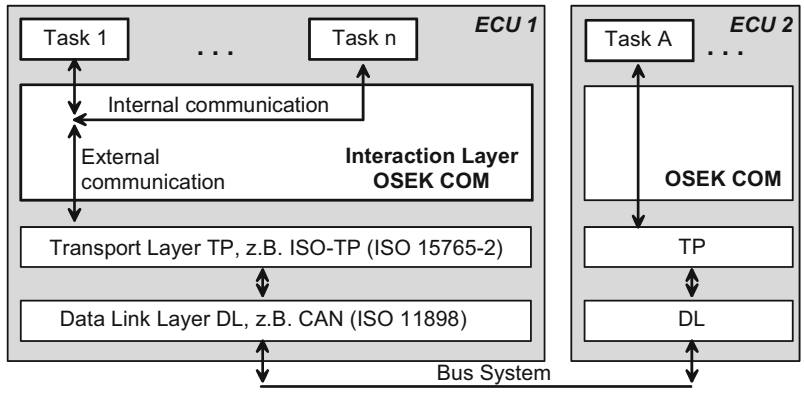

Abb. 7.8 Struktur von OSEK COM

munikation die Übertragung über das Bussystem an, ohne die Übertragung selbst abzuwarten, und kehrt zur aufrufenden Task zurück. Diese kann die Daten sofort löschen oder verändern, ohne dass die versendete Botschaft beeinflusst wird. Die Tasks, die in der Konfigurationsdatei als Empfänger der Botschaft konfiguriert wurden, werden über die neue Botschaft informiert. Die Information (*Notification*) kann durch das Starten einer eigenen Task, das Setzen eines OSEK OS Events oder eines OSEK COM Flags bzw. den Aufruf einer vom Anwendungsprogramm bereitzustellenden Callback-Funktion erfolgen. Die Empfänger-Task liest die Botschaft mit der API-Funktion `ReceiveMessage()`. Dabei wird die Botschaft in einen Speicherbereich der Empfänger-Task kopiert. OSEK-COM unterscheidet zwischen einfachen Botschaften (*Unqueued Messages*) und Botschaftswarteschlangen (*Queued Messages*). Bei den einfachen Botschaften puffert der Interaktionslayer stets nur die Daten der letzten Botschaft. Diese Daten können von einer oder mehreren Empfänger-Tasks beliebig oft ausgelesen werden. Sobald eine Sender-Task die Botschaft erneut sendet, werden die alten durch die neuen Daten überschrieben. Im Gegensatz dazu speichert eine Botschaftswartelange die Daten mehrerer aufeinander folgender Übertragungen der Botschaft bis zu einer fest konfigurierten Maximalzahl nach dem First In First Out-Prinzip. Beim Lesen erhält der Empfänger die jeweils ältesten in der Warteschlange enthaltenen Daten, die daraufhin aus der Warteschlange gelöscht werden. Wenn die Warteschlange einer Botschaft voll ist, werden neu ankommende Daten verworfen und die Empfangs-Task wird beim nächsten Lesen durch eine Statusmeldung über den Verlust informiert. Falls für die Botschaft mehrere Empfänger konfiguriert sind, richtet OSEK für jeden Empfänger eine eigene Warteschlange ein.

Alle Botschaften müssen in der Entwicklungsphase in der OIL-Konfigurationsdatei des Steuergerätes (etwas umständlich) definiert werden (Abb. 7.9). In den TASK-Abschnitten der Konfigurationsdatei wird festgelegt, welche Tasks welche Botschaften verwenden. Im MESSAGE Abschnitt wird für jede Botschaft festgelegt, ob es sich um eine interne oder externe Sende- bzw. Empfangsbotschaft handelt, ob eine Warteschlange verwendet wird und

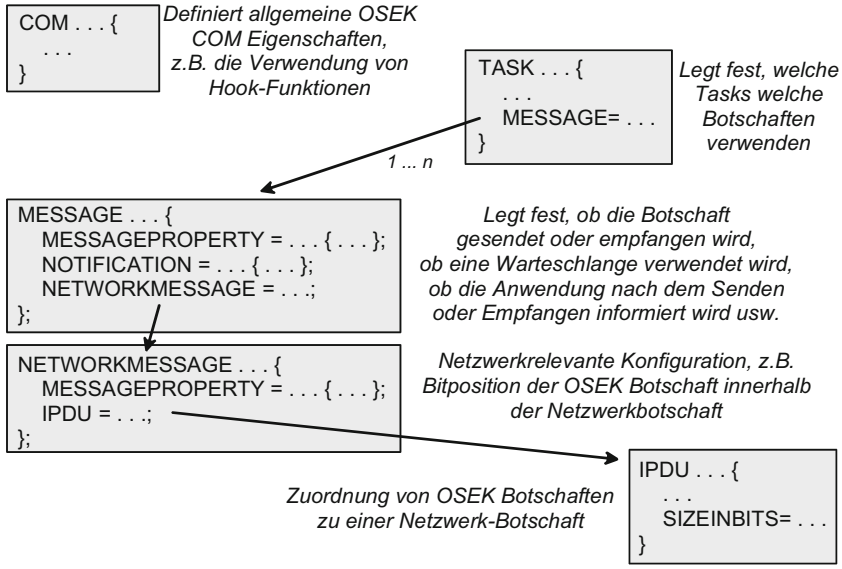

Abb. 7.9 Struktur der OIL-Konfigurationsdatei für OSEK-COM

wie lange diese gegebenenfalls ist usw. Als OSEK COM Botschaft werden häufig einzelne steuergeräteinterne Variable definiert, die von der Transportschicht nicht einzeln, sondern mit anderen Variablen zu einer Botschaft zusammengefasst gesendet werden sollen. Unter NETWORKMESSAGE erfolgt die Zuordnung einer oder mehrerer OSEK-Botschaften zu einer Netzwerk-Botschaft, bei OSEK als *IPDU … Interaction Layer Protocol Data Unit* bezeichnet. Diese wird dann der Transportschicht des Bussystems übergeben, deren wichtigste Eigenschaften unter IPDU konfiguriert werden, soweit sie für OSEK COM relevant sind. Wie die Transportschicht die Botschaften weiter zusammenfasst oder segmentiert, spielt für OSEK COM keine Rolle. Bei rein internen Botschaften sind die Abschnitte NETWORKMESSAGE und IPDU nicht notwendig.

OSEK COM wird beim Hochlauf des Systems in der Regel von einer OSEK OS Task aus mit `StartCOM()` gestartet werden. Ein Neustart nach einem vorherigen `StopCOM()` ist jederzeit möglich. Beim Starten werden für die Daten der Botschaften auch die bei der Konfiguration festgelegten Initialisierungswerte hergestellt, die gegebenenfalls mit `InitMessage()` zur Laufzeit geändert werden können.

Ähnlich wie OSEK OS erlaubt OSEK COM ebenfalls verschiedene Ausbaustufen (*COM Conformance Class*), deren Implementierung zu einem unterschiedlich großen Speicheraufwand führt. Alle Klassen unterstützen die interne Kommunikation mit einfachen Botschaften ohne Botschaftswarteschlangen. Dies ist auch die Mindestanforderung für OSEK OS, da dessen Intertaskkommunikation darauf basiert. Die wesentlichen Unterschiede zwischen den Ausbaustufen sind in Tab. 7.4 dargestellt.

Tab. 7.4 OSEK COM Ausbaustufen (COM Conformance Classes)

	CCCA	CCCB	CCC0	CCC1
Externe Kommunikation	Nein		Ja	
Botschaftswarteschlangen	Nein	Ja	Nein	Ja
Botschaften mit variabler Länge inkl. Länge 0 Automatisches periodisches Senden Botschaftsfilter Überwachung von Botschafts-Deadlines/Timeouts	Nein			Ja

`SendMessage()` und `ReceiveMessage()` sind für Botschaften mit fester Datenlänge vorgesehen. In Ausbaustufe CCC1 gibt es zusätzlich die API-Funktionen `SendDynamicMessage()` und `ReceiveDynamicMessage()` für Botschaften mit variabler Datenlänge bis zu einem vorkonfigurierten Maximalwert. Diese beiden Funktionen sind nur für die externe Kommunikation und ohne Botschaftswarteschlange einsetzbar. Mit der Funktion `SendZeroMessage()` kann sogar eine Botschaft ganz ohne Daten versendet werden. Damit kann eine Task in einem Steuergerät beispielsweise eine Task in einem anderen Steuergerät triggern, was mit einem OSEK OS Event nur bei Tasks auf demselben Mikrocontroller möglich ist.

Beim Senden und Empfangen lassen sich zusätzlich weitere automatische Verarbeitungsschritte integrieren:

- Bei der externen Kommunikation ist eine Konvertierung der Byte-Reihenfolge möglich, da Mikroprozessoren bei der Speicherung von Mehrbyte-Daten teilweise unterschiedlich arbeiten (Little Endian und Big Endian). Dazu ruft die Interaktionsschicht eine vom Anwender bereitgestellte Funktion auf, die die Umwandlung vornimmt. Der Mechanismus ist für jede einzelne Botschaft konfigurierbar.
- Beim Empfangen und bei externen Botschaften auch beim Senden kann in Implementierungen der Konformitätsklasse CCC1 eine Botschaftsfilterung erfolgen. Für jede Botschaft kann konfiguriert werden, ob die Botschaft immer, nie oder nur dann weiter verarbeitet, d. h. gesendet oder empfangen wird, wenn die empfangenen Daten sich gegenüber der letzten Übertragung geändert haben, einen bestimmten Wert oder Wertebereich aufweisen oder nicht usw.
- Ebenfalls nur in Konformitätsklasse CCC1 gibt es die Möglichkeit, dass eine Botschaft ohne Zutun der Anwendung periodisch gesendet wird. Während eine Botschaft nach dem Aufruf von `SendMessage()` üblicherweise so schnell wie möglich gesendet wird (*Triggered Direct Transmission*), wird die neue Botschaft bei der periodischen Übertragung (*Periodic*) lediglich in den lokalen Sendedatenspeicher eingetragen, dessen Inhalt mit konfigurierbarer Wiederholfrequenz und Phasenlage dann periodisch gesendet wird, ohne dass die Anwendung dies nochmals explizit veranlassen muss. Botschaftsabhängig ist auch eine Mischform beider Übertragungsarten möglich (*Mixed*

Transmission). Das periodische Senden kann für alle bei der Konfiguration als periodisch definierten Botschaften gemeinsam jederzeit mit `StartPeriodic()` und `StopPeriodic()` gestartet und beendet werden.

- Sowohl beim Senden als auch beim Empfangen lässt sich (bei CCC1) optional der Zeitabstand (*Deadline, Timeout Monitoring*) überwachen. Beim Senden wird die Anwendung informiert, wenn eine Botschaft nicht innerhalb einer Maximalzeit nach der Sendeaufforderung durch `SendMessage()` tatsächlich über das Bussystem gesendet wird. Auch der Mindestabstand zwischen zwei Sendeversuchen lässt sich konfigurieren. Beim Empfangen wird der maximale Zeitabstand vom Empfang einer Botschaft bis zum erneuten Empfang derselben Botschaft überwacht.

7.2.3 Netzmanagement mit OSEK/VDX NM

Bei einem echten Netzmanagement werden Busadressen und Routinginformationen für Gateways dynamisch konfiguriert und detaillierte Statusinformationen ermittelt. OSEK NM dagegen erlaubt nur eine einfache Überwachung, ob die vorab festgelegten Busteilnehmer Botschaften senden und empfangen können und stellt den Anwendungsprogrammen diese Informationen bereit. Trotzdem soll der von OSEK vorgegebene Name hier beibehalten werden. Theoretisch ist OSEK NM als eigenständige Komponente spezifiziert, in der Praxis wird es jedoch zusammen mit OSEK OS und OSEK COM implementiert. Das überwachte Bussystem ist theoretisch ebenfalls beliebig, sofern es sich um ein System handelt, bei dem jeder Teilnehmer die gesamte Kommunikation aller anderen Teilnehmer mithören kann. Entstanden ist OSEK NM aber parallel zu OSEK COM im Einsatz mit CAN.

Man unterscheidet

- **Direktes Netzmanagement**: Dabei werden spezielle Botschaften für das Netzmanagement über das Bussystem versendet. Dadurch entsteht eine (geringe) zusätzliche Busbelastung und es können nur solche Busteilnehmer überwacht werden, die intern ebenfalls eine Netzmanagement-Komponente enthalten.
- **Indirektes Netzmanagement**: Dabei werden nur die im normalen Betrieb ohnehin versendeten Botschaften beobachtet. Damit können nur diejenigen Steuergeräte erfasst werden, die mindestens eine Botschaft periodisch versenden, wobei die Herkunft der Botschaft diesem Steuergerät eindeutig zuordenbar sein muss. In der indirekten Ausführung können auch Steuergeräte überwacht werden, die selbst keine Netzmanagement-Komponente enthalten. Falls die Wiederholfrequenz der überwachten Botschaft gering ist, kommt es zu großen Verzögerungen bei der Überwachung. Steuergeräte, die nur ereignisgesteuert und nicht periodisch Botschaften aussenden, oder Knoten, die nur Botschaften empfangen aber nicht senden, können nicht überwacht werden.

OSEK NM erlaubt beides, geht aber davon aus, dass innerhalb eines Bussystems durchgängig nur eines der beiden Verfahren verwendet wird. Dabei gibt es keine Zentralinstanz (*zentrales Netzmanagement*), sondern jedes Steuergerät führt seine eigene Überwachung durch (*dezentrales Netzmanagement*).

Als Status der Busteilnehmer liefert OSEK NM die Information, ob ein Teilnehmer an der Buskommunikation teilnimmt (*Node Present*) oder nicht (*Node Absent*). Detailliertere Werte sind nicht spezifiziert, doch darf ein Hersteller Erweiterungen implementieren. Die Information wird in einer Konfigurationsliste (*Configuration*) verwaltet, wobei die Liste statisch aufgebaut ist und alle vom Netzmanagement erfassten Busteilnehmer bereits in der Entwicklungsphase festgelegt werden müssen. Die Konfiguration enthält auch die Information über das eigene Steuergerät, wobei dessen Zustände in der Spezifikation statt als *Present* und *Absent* in einigen Passagen als *Not Mute* und *Mute* bezeichnet werden. Gemeint ist damit, dass der eigene Knoten keine Botschaften senden kann bzw. glaubt, dass diese von keinem anderen Busteilnehmer empfangen werden können. Das Netzmanagement hält lediglich die Konfigurationsliste auf dem Laufenden, die weitere Auswertung der Liste und die sich daraus ergebenden Eingriffe in den Betrieb des Steuergerätes, wenn bestimmte Kommunikationspartner nicht verfügbar sind, bleibt dem Anwendungsprogramm überlassen. Dieses muss die Konfigurationsliste mit `GetConfig()` abfragen und kann sie mit `CmpConfig()` mit einem Sollzustand oder einem vorigen Stand vergleichen. Optional kann sich die Anwendung auch durch Aktivieren einer Task oder Setzen eines Events vom Netzmanagement informieren lassen, wenn sich ein Eintrag in der Konfigurationstabelle geändert hat.

Abbildung 7.10 zeigt das Zustandsmodell des Netzmanagements. Durch den Funktionsaufruf `StartNM()` wird das Netzmanagement gestartet und geht in den Zustand *NMNormal* über. In der Konfigurationsliste werden alle Netzknoten zunächst mit *Node Absent*, d. h. als nicht über das Bussystem erreichbar eingetragen und die Überwachung beginnt wie weiter unten im Detail beschrieben. Falls der Kommunikationscontroller den Totalausfall des Bussystems meldet, bei CAN z. B. einen Bus-Off-Fehler, geht das Netzmanagement in den Zustand *NMLimpHome* über, in dem die Überwachung eingestellt wird. Dafür ist keine API-Funktion vorgesehen, sondern es wird davon ausgegangen, dass es eine interne Schnittstelle gibt, über die OSEK COM bzw. die für den Kommunikationscontroller zuständige Treibersoftware diese Information direkt an OSEK NM liefert. Weiterhin ist es möglich, dass die Anwendung die Buskommunikation vorübergehend einstellen und den Kommunikationscontroller und gegebenenfalls weitere Teile des Steuergerätes in einen Strom sparenden Zustand schalten will. Damit es nicht zu Überwachungsfehlern kommt, versetzt die Anwendung dazu auch das Netzmanagement mit Hilfe der API-Funktion `GotoMode(BusSleep)` nach einer konfigurierbaren Wartezeit in den Zustand *NMBusSleep*, in dem die Überwachung ebenfalls eingestellt wird. Bei Wiederaufnahme der Buskommunikation wird die Überwachung durch `GotoMode(Awake)` wieder aufgenommen. Eine Anwendung kann den aktuellen Zustand der OSEK NM Komponente über die Funktion `GetStatus()` jederzeit abfragen. Mit der Funktion `CmpStatus()` kann der aktuelle leicht mit einem früheren Zustand verglichen werden.

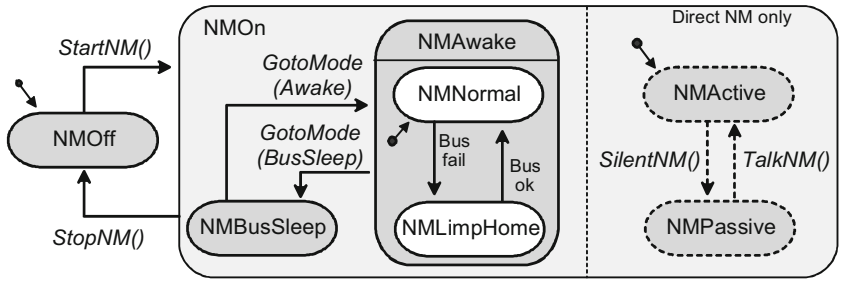

Abb. 7.10 Vereinfachtes Zustandsmodell von OSEK NM ohne transiente Zwischenzustände (*gestrichelte Zustände* nur beim direkten Netzmanagement)

Die eigentliche Überwachung erfolgt bei *indirektem Netzmanagement* wie folgt:

- Für jede überwachte Botschaft wird überprüft, ob sie innerhalb eines vorkonfigurierten Zeitfensters mindestens einmal empfangen wurde. Falls ja, wird der Sender als *Node Present* eingestuft, falls nein als *Node Absent*. In beiden Fällen wird das Zeitfenster nach Ende des vorigen Zeitfensters erneut gestartet.
- Im einfachsten Fall wird für alle Botschaften dasselbe Zeitfenster (*Global Timeout*) verwendet. Wenn die Wiederholperioden der einzelnen Botschaften dagegen stark unterschiedlich sind, ist es sinnvoller, für jede Botschaft ein eigenes Überwachungszeitfenster (*Timeout per Message*) zu konfigurieren.
- Optional kann für jede Botschaft auch noch ein Fehlerzähler konfiguriert werden, der bei ausbleibenden Botschaften bis zu einem Maximalwert heraufgezählt und bei wieder eintreffenden Botschaften wieder heruntergezählt wird. Ein Knoten wird in diesem Fall erst dann als *Node Absent* eingestuft, wenn der Fehlerzähler den Maximalwert erreicht hat. Durch diese *Fehlerfilterung* werden kurzzeitig auftretende Fehler in der Erkennung unterdrückt.

OSEK NM verwendet für die Überwachung Mechanismen von OSEK OS, z. B. Alarme, bzw. von OSEK COM, z. B. *das Message Deadline Monitoring*. Über eine interne Schnittstelle informiert OSEK COM das Netzmanagement OSEK NM über ankommende überwachte Botschaften sowie das Überschreiten von Zeitschranken.

Bei *direktem Netzmanagement* werden alle Knoten eines Netzes mit einer festen, eindeutigen Kennung (*Node Identifier*) in aufsteigender Reihenfolge konfiguriert und tauschen periodisch Botschaften des in Abb. 7.11 dargestellten Formates aus. Die Botschaften enthalten die Kennung des Senders und des Empfängers (*Source* und *Destination ID*) sowie ein Opcode-Feld, indem der Typ der Botschaft kodiert ist. Vorgesehene Botschaftstypen sind *Ring*, *Alive* und *Limp Home*. Außerdem kann die Ring-Botschaft ein optionales Datenfeld enthalten, zu dem die Spezifikation aber keine Vorgaben macht, außer dass die Daten bei stabilem Netzbetrieb (siehe unten) über `ReadRingData()` gelesen und über `TransmitRingData()` gesetzt werden können. Die Codierung der Knoten-

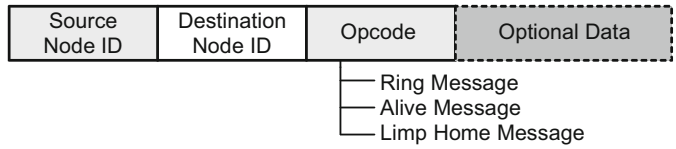

Abb. 7.11 Format der Botschaften für das direkte Netzmanagement

Kennungen und des Opcode-Feldes sowie deren Abbildung auf ein konkretes Bussystem, z. B. auf CAN Message Identifier, sind herstellerabhängig. Die Spezifikation enthält jedoch einen Implementierungsvorschlag für CAN.

Im *stabilen Betrieb* des Netzes bilden die Knoten einen logischen Ring und tauschen Ring-Botschaften aus. Die Reihenfolge ist dabei durch die Kennungen der Knoten vorgegeben. Ein Knoten wartet, bis er von seinem Vorgänger, d. h. dem Knoten mit der nächstniedrigeren Kennung, eine Ring-Botschaft mit seiner eigenen Kennung als Zieladresse enthält (Abb. 7.12). Der Zielknoten wartet die Zeit T_{typ} ab und sendet eine weitere Ring-Botschaft an seinen Nachfolger, d. h. den Knoten mit der nächsthöheren Kennung weiter. Der Nachfolger des Knotens mit der höchsten Kennung ist der Knoten mit der niedrigsten Kennung, so dass der Ring geschlossen wird. Jeder Knoten hört die Ring-Botschaften aller anderen Knoten mit und frischt jedes Mal den Inhalt seiner Konfigurationstabelle auf, d. h. er trägt jeden Knoten, dessen Ring-Botschaft er im korrekten Zeitfenster mitgehört hat, als aktiv (*Node Present*) ein. Knoten, von denen bekannt ist, dass sie gerade nicht aktiv sind (*Node Absent*) werden beim Senden der Ring-Botschaften übersprungen.

Erkennt ein Knoten, dass ein adressierter Knoten nicht innerhalb der Zeit $T_{\text{Max}} > T_{\text{Typ}}$ auf die Ring-Botschaft antwortet, z. B. aufgrund eines Fehlers oder weil dieser Knoten sich in der Zwischenzeit abgeschaltet hat, oder stellt ein Knoten fest, dass er selbst bei den Ring-Botschaften übergangen wurde, z. B. weil er gerade neu eingeschaltet wurde, so sendet er eine *Alive*-Botschaft aus. Die *Alive*-Botschaft enthält die Kennung des Senders, die Empfänger-Kennung ist beliebig. Als Reaktion auf die erste *Alive*-Botschaft versetzen alle Knoten ihre Konfigurationstabellen mit `InitConfig()` in den Ausgangszustand,

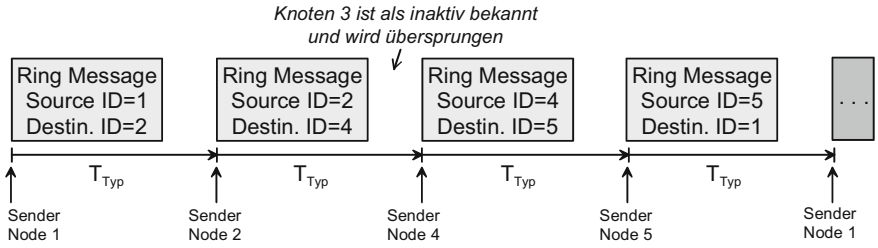

Abb. 7.12 Ring-Botschaften bei *stabilem Betrieb* eines Netzes mit 5 Knoten, wobei der Knoten 3 gerade nicht aktiv ist

bei dem zunächst alle Knoten außer dem eigenen als inaktiv (*Node Absent*) markiert sind. Dieser Zustand des Netzes, in dem *Alive*-Botschaften versendet werden, wird in der Spezifikation als *dynamischer Netzzustand* bezeichnet. Bei jeder weiteren *Alive*-Botschaft wird der jeweilige Sender dann in der Konfigurationsliste als aktiv (*Node Present*) markiert. Jeder Knoten, der selbst eine *Alive*-Botschaft versendet hat und mindestens eine *Alive*-Botschaften empfangen hat, kennt einen potenziellen Nachfolger und beginnt nach der Zeit T_{Typ} mit dem Versenden einer Ring-Botschaft an seinen potenziellen Nachfolger, falls er nicht vorher die Ring-Botschaft eines anderen Steuergerätes erhält. Dabei kann es kurzzeitig vorkommen, dass mehrere Ring-Botschaften im Netz im Umlauf sind. Da aber alle Knoten ihre Konfiguration bei jeder empfangenen Botschaft auffrischen, verschwinden die zusätzlichen Botschaften sehr schnell und das Netz erreicht wieder einen stabilen Zustand, in dem nur noch eine einzige Ring-Botschaft reihum weitergereicht wird. Wenn es einem Knoten nicht gelingt, nach einer bestimmten Anzahl von Versuchen selbst *Alive*-Botschaften zu senden oder wenn er während der Zeit T_{max} wiederholt keine gültigen Ring-Botschaften erhält, schaltet sich das Netzmanagement des Knotens in den Zustand *LimpHome*. In diesem Zustand muss der Knoten davon ausgehen, dass keine Kommunikation über das Bussystem möglich ist. Trotzdem sendet er noch in größeren Zeitabständen T_{Error} eine *LimpHome*-Botschaft, falls ein anderer Busteilnehmer ihn doch noch empfangen kann. Sobald er selbst wieder eine gültige Botschaft erhält oder wenn die Anwendung das Netzmanagement neu startet, wird der Normalbetrieb wieder aufgenommen. In der Konfigurationsliste oder in einer separaten Liste vermerkt jeder Knoten auch diejenigen Steuergeräte, von denen er *LimpHome*-Botschaften empfangen hat.

Eine Anwendung kann sich optional sowohl beim Empfang einer Ring-Botschaft als auch bei einer Änderung der Netz-Konfiguration durch Aktivierung einer Task oder Setzen eines Events informieren lassen. Falls erforderlich, kann sie das Aussenden von Ring- oder *Alive*-Botschaften über `SilentNM()` zeitweilig sperren und über `TalkVM()` wieder freigeben. Um einen kontrollierten Übergang in den *BusSleep*-Zustand im gesamten Bussystem zu ermöglichen, sind im Opcode-Feld der NM-Botschaften Bits vorgesehen, mit denen ein Steuergerät die anderen Geräte auffordern kann, in den *BusSleep*-Zustand überzugehen und diese Geräte ihre Bereitschaft dazu bestätigen können, bevor der Übergang nach einer Wartezeit dann tatsächlich erfolgt.

Im Vergleich zu OSEK OS und auch zu OSEK COM lässt die OSEK NM Spezifikation erhebliche Freiheitsgrade für die Implementierung, die die Portierung einer Anwendung erschweren. Praktisch alle Funktionen werden als optional und das Netzmanagement als skalierbar bezeichnet. Es gibt aber nur wenige Vorgaben für eine sinnvolle Minimalfunktionalität oder in sich konsistente Ausbaustufen. Die Codierung der Botschaften oder der Aufbau der Konfigurationsliste werden vage oder gar nicht spezifiziert und die Festlegung von Elementen für OSEK NM in der OIL-Konfigurationsdatei fehlt vollständig, so dass jeder Hersteller beliebig vorgehen kann. In der Spezifikation finden sich lediglich einige unvollständig definierte Makros, die alle mit der Bezeichnung `Init...` beginnen, mit denen die Konfiguration beschrieben und aus denen das Systemgenerierungswerkzeug dann wiederum den entsprechenden Programmcode erzeugen soll.

7.2.4 Zeitgesteuerter Betriebssystemkern OSEK Time, Fehlertoleranz OSEK FTCOM und Schutzmechanismen Protected OSEK

Mit *OSEK Time* wurde eine zeitgesteuerte Betriebssystem-Erweiterung des ereignisgesteuerten OSEK OS veröffentlicht und durch *OSEK FTCOM* (*Fault Tolerant COM*) ergänzt, die zu OSEK Time passende Variante von OSEK COM. Die Version 1.0 ist allerdings bis heute unverändert die einzig öffentlich zugängliche Spezifikation und es mangelt an realen Produkten. Die Weiterentwicklung erfolgt inzwischen im Rahmen von AUTOSAR. OSEK Time ist ein eigenständiges Betriebssystem mit einem eigenen Taskmodell, das aber mit einem OSEK OS Betriebssystem und dessen Taskmodell auf demselben Mikrocomputer koexistieren kann. OSEK Time hat dabei absoluten Vorrang, so dass OSEK OS und dessen Tasks nur dann laufen, wenn OSEK Time im *Leerlauf* ist, d. h. wenn keine OSEK Time Task zur Bearbeitung ansteht. Die gilt auch für Interrupts, die dem OSEK OS System zugeordnet sind. Diese bleiben gesperrt, solange OSEK Time Tasks laufen. Dadurch soll ein streng deterministischer Ablauf aller OSEK Time Tasks erreicht werden.

Im Gegensatz zu OSEK OS sind Tasks bei OSEK Time grundsätzlich Funktionen ohne Endlosschleifen oder Wartevorgänge, die nach ihrem Start ablaufen und wieder enden. Die maximale Ausführungszeit (*Worst Case Execution Time*) jeder Task muss genau bekannt sein. Welche Task zu welchem Zeitpunkt gestartet werden soll, wird während der Entwicklungsphase festgelegt und in einer Ablauftabelle (*Dispatch Tabelle*) gespeichert. Vor dem Start befinden sich alle Tasks im Zustand *Suspended* (Abb. 7.13). Der Scheduler wird bei OSEK Time *Dispatcher* genannt wird, weil er keine echten Entscheidungsaufgaben (*Scheduling*) zu erledigen hat, sondern lediglich nach der vordefinierten Ablauftabelle Tasks startet (*Dispatching*). Er wird regelmäßig mit einer festen Periodendauer, dem OSEK Time Zeittakt (*Tick*) aktiviert, und startet dann diejenige Task (Zustand *Running*), deren Startzeitpunkt (*Activation Time*) erreicht ist (Abb. 7.14). Die neue Task verdrängt die gerade laufende Task in jedem Fall. Diese wechselt in den Zustand *Preempted*. Wenn die neue Task beim nächsten Zeittakt bereits beendet und kein Startzeitpunkt einer weiteren Task erreicht ist, darf die verdrängte Task weiterlaufen. Falls die Task dagegen noch nicht beendet ist, läuft sie weiter oder wird, falls der Startzeitpunkt einer weiteren Task erreicht ist, von dieser ebenfalls verdrängt. Die Wiederaufnahme der verdrängten Tasks erfolgt in der umgekehrten Reihenfolge, in der sie verdrängt wurden (*Stack Based Scheduling*).

Der gesamte Vorgang wiederholt sich nach einer vordefinierten Zeit (*Dispatcher Round*) und die Ablauftabelle wird erneut durchlaufen. Dabei können einige der Tasks während eines Durchlaufs auch mehrfach aktiviert werden. Die Tasks selbst haben auf ihre eigene oder die Aktivierung anderer Tasks keinerlei Einfluss. Task-Prioritäten und Scheduler/Dispatcheraufrufe sind weder erforderlich noch vorhanden. Allerdings kennt auch OSEK Time das Konzept der Anwendungsmodi und kann im Gegensatz zu OSEK OS mit `SwitchAppMode()` im laufenden Betrieb den Anwendungsmodus und damit die Ablauftabelle umschalten.

In den Zeitabschnitten, in denen keine OSEK Time Task laufen will, dürfen OSEK OS Task laufen. OSEK OS bildet aus Sicht von OSEK Time dessen Leerlauf-Task `ttIdleTask`.

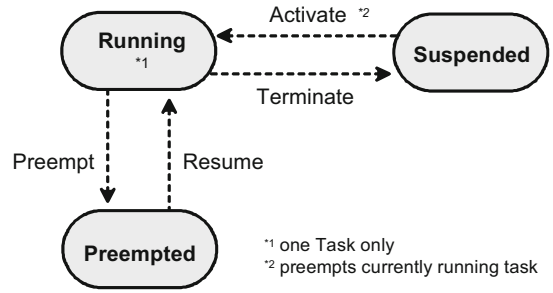

Abb. 7.13 Zustandsmodell der OSEK Time Tasks

Die von OSEK OS bekannten Events, Ressourcen, Alarme usw. können innerhalb von OSEK Time Tasks nicht verwendet werden.

Bei der Aufstellung der Ablauftabelle in der Entwicklungsphase muss mit theoretischen Methoden und gegebenenfalls Simulationswerkzeugen sichergestellt werden, dass der vorgesehene Ablauf auch realisierbar ist, d. h. dass die Tasks nicht nur periodisch gestartet werden, sondern dass sie auch rechtzeitig beendet werden. In keinem Fall darf dieselbe Task wieder aktiviert werden, bevor sie beendet ist. In die Ablauftabelle werden nicht nur die Startzeitpunkte (*Activation Time*) für jede Task eingetragen, sondern auch die Zeitpunkte, zu denen die Task spätestens beendet sein muss (*Deadline*), so dass der Dispatcher die Laufzeit einer Task überwachen und das Anwendungsprogramm informieren kann, indem er eine von der Anwendung bereitgestellte Funktion `ttErrorHook()` aufruft. Danach beendet sich OSEK Time in jedem Fall selbst und ruft eine weitere von der Anwendung bereit zu stellende Funktion `ttShutdownHook()` auf, in der OSEK Time dann gegebenenfalls mit `ttStartOS()` erneut gestartet werden kann.

Der Start des Betriebssystems erfolgt mit der API-Funktion `ttStartOS()`. Danach startet OSEK Time als erstes die Leerlauf-Task `ttIdleTask`, d. h. das gegebenenfalls vorhandene OSEK OS Sub-Betriebssystem. Mit `ttShutdownOS()` wird das gesamte System gestoppt, während `OSStop()` innerhalb einer OSEK OS Task lediglich das OSEK OS Sub-Betriebssystem beendet. OSEK Time kann keine Events, Alarme und Ressourcen des OSEK

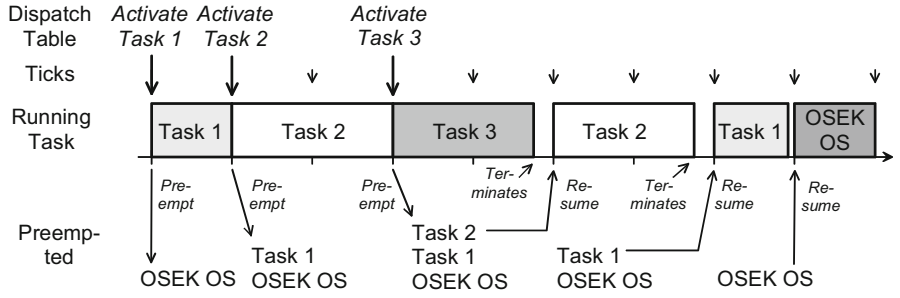

Abb. 7.14 Beispiel eines Task-Ablaufs bei OSEK Time

OS Betriebssystems benutzen, aber mit diesem Daten über Botschaften oder notfalls auch über globale Variable austauschen.

Das Zeitraster für die Ablaufsteuerung wird in der Regel von einem Zeitgeber des Mikrocontrollers abgeleitet. Bei verteilten Systemen ist es möglich, das Zeitraster aller Steuergeräte über das Bussystem zu synchronisieren. Die Synchronisation erfolgt beim Starten und wiederholt sich automatisch im laufenden Betrieb. Dabei kann konfiguriert werden, ob der Dispatcher erst nach erfolgter Synchronisation mit der Aktivierung von Tasks beginnen soll, oder ob das System sofort asynchron startet und nach erfolgter Synchronisation das Zeitraster schlagartig oder schrittweise an das globale Zeitraster anpasst. Die Anwendung kann mit `ttGetOSSyncStatus()` den Stand der Synchronisation und mit der FTCOM-Funktion `ttGetGlobalTime()` das über das Bussystem synchronisierte globale Zeitraster abfragen. Das Verfahren zur Zeitsynchronisation über das Bussystem selbst dagegen, z. B. die Verwendung der entsprechenden FlexRay-Mechanismen, ist leider nicht spezifiziert.

Während Interrupts bei OSEK OS laufende Task jederzeit unterbrechen können, sofern sie nicht mit Hilfe einer der API-Funktionen gesperrt werden, unterliegen Interrupts bei OSEK Time ebenfalls der Kontrolle der Ablauftabelle. Dort werden für jeden Interrupt Zeitfenster definiert, in denen der Interrupt auftreten darf. Wenn er tatsächlich auftritt und die *Interrupt Service Routine* bearbeitet wurde, bleibt er bis zum Beginn seines nächsten Zeitfensters gesperrt. Definiert werden OSEK Time Interrupts mit dem Makro `ttISR`. Interrupts des OSEK OS Subsystems werden grundsätzlich nur in den Zeitbereichen freigegeben, in denen keine OSEK Time Task läuft. Das Sperren und Freigeben von Interrupts durch die API-Funktionen von OSEK OS wirkt sich lediglich auf dessen Interrupts aus, bei den OSEK Time Interrupts hat es keine Wirkung.

Die Kommunikation von OSEK Time Tasks untereinander, mit OSEK OS Tasks oder über das Bussystem mit anderen Steuergeräten kann über die OSEK FTCOM Funktionen `ttSendMessage`() und `ttReceiveMessage()` in ähnlicher Weise erfolgen wie bei OSEK COM für OSEK OS Tasks. Die Information an die Anwendung, wenn eine Botschaft gesendet oder empfangen wurde, findet, da OSEK Time keine Events oder ereignisgesteuerte Taskaktivierung kennt, nur über Flags statt, die von den Tasks mit `ttReadFlag()` abgefragt werden müssen. Dass FTCOM unvollständig spezifiziert ist, wird beim Aspekt Fehlertoleranz besonders deutlich. Botschaften sollen über mehrere Kanäle redundant gesendet und empfangen werden, wie dies beispielsweise bei FlexRay vorgesehen ist (siehe Abschn. 3.3). Die Übereinstimmung der redundant empfangenen Daten soll selbstständig überprüft werden. Wie dies geschieht und welche Reaktion bei auftretenden Fehlern erfolgt, bleibt aber offen.

Im Gegensatz zu „richtigen“ Rechnern besitzt die Mehrzahl der Mikrocontroller, die heute in Kfz-Steuergeräten eingesetzt werden, aus Kostengründen keine eingebauten Hardwareschutzmechanismen, mit denen gewährleistet werden könnte, dass die Fehlfunktion in einem Teil der Steuergerätesoftware keine oder nur begrenzte Auswirkungen auf andere Teile des Gerätes hat. Mit der zunehmenden Komplexität der Software einerseits, vor allem aber vor dem Hintergrund, dass sich das Entwicklungsmodell ändert und die Software

zukünftig nicht mehr in einer Verantwortung entwickelt wird, sondern aus Komponenten verschiedenster Hersteller „zusammengesteckt“ werden soll, findet hier ein Umdenken statt. Solche Hardwareschutzmechanismen sorgen dafür, dass eine Softwarekomponente nur Zugriff auf ihren eigenen Programmcode und ihre eigenen Daten hat und dass eine gewöhnliche Softwarekomponente keinen direkten Zugriff auf den Mikrocontroller und dessen Peripheriebaugruppen hat und so durch Sperren der Interrupts und Verändern von Steuerregistern nicht bereits bei banalsten Softwarefehlern die Gesamtfunktion des Systems gefährden kann. Man spricht in diesem Zusammenhang von privilegierter und nicht privilegierter Software, Zugriffsschutz und geschützten Adressräumen. Der Einsatz eines Mikrocontrollers mit Hardwareschutzmechanismen setzt auf der Softwareseite die Verwendung eines entsprechenden Betriebssystems voraus. Im Rahmen der HIS wurde daher ein Vorschlag erarbeitet, um OSEK OS entsprechend zu erweitern. Reale Produkte mit den *OSEK OS Extensions for Protected Applications* (*Protected OS*) konnten sich jedoch ähnlich nicht durchsetzen. Die dabei entwickelten Konzepte wurden jedoch in die AUTOSAR-Aktivitäten übernommen und werden in Kap. 8 beschrieben.

7.2.5 Scheduling, Taskprioritäten und Zeitverhalten bei OSEK OS und AUTOSAR OS

Zu den kritischen Punkten beim Entwurf verteilter Echtzeitsysteme gehören die Übertragungsverzögerungen (Latenz und Jitter) von Mess-, Steuer- und Reglerdaten zwischen den Steuergeräten. Nur ein Teil der Verzögerungen entsteht durch die eigentliche Busübertragung. Noch größere Verzögerungen ergeben sich oft bei der Bereitstellung der Daten durch die Anwendungssoftware und deren Weiterverarbeitung in den jeweiligen Protokollstapeln auf der Sender- und Empfängerseite (vgl. Abb. 2.17). Das Zeitverhalten der wichtigsten Kfz-Bussysteme wurde bereits in Kap. 3 analysiert. Hier soll nun das Verhalten der Software untersucht werden, soweit es durch das Betriebssystem beeinflusst wird. Da AUTOSAR OS (Abschn. 8.2) dieselben Scheduling-Konzepte verwendet wie OSEK OS, gilt die folgende Analyse für beide Betriebssysteme. Die Darstellung orientiert sich an [4, 5] und geht von folgenden Randbedingungen aus (Abb. 7.15):

- Die maximalen Laufzeiten (*Worst Case Execution Time WCET*) $T_{\mathrm{E,k}}$ der $k = 1 \ldots \mathrm{N}$ Tasks seien bekannt. *Interrupt Service Routinen* werden als Tasks hoher Priorität modelliert. Die Dauer von Taskwechseln und Betriebssystemaufrufen wird als Teil der Tasklaufzeit betrachtet. Für *Extended Tasks*, die Endlosschleifen und Wartepunkte enthalten, wird $T_{\mathrm{E,k}}$ als maximale Laufzeit bis zum Erreichen des nächsten Wartepunktes interpretiert.
- Der Mindestzeitabstand zwischen aufeinanderfolgenden Aktivierungen sei für jede Task bekannt. Diese Zeit $T_{\mathrm{P,k}}$ entspricht der *Interarrival Time* bei ereignisgesteuerten und der Periodendauer bei zyklischen Tasks.

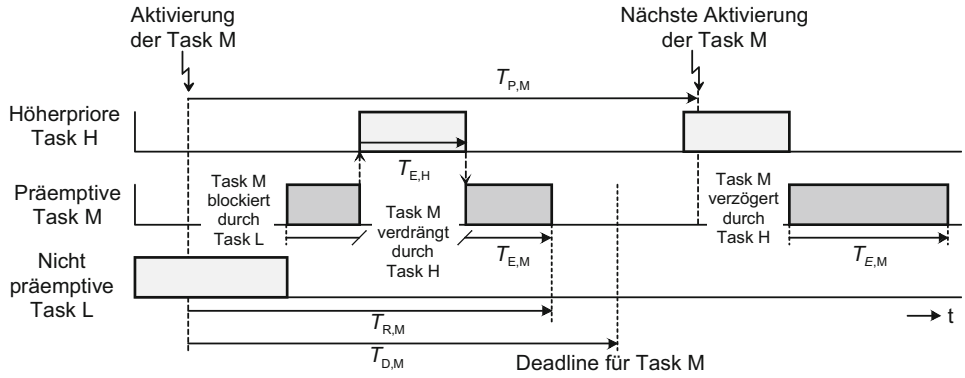

Abb. 7.15 Szenario zur Berechnung der Task-Antwortzeit

- Alle Tasks sollen auf einer einzigen CPU laufen. Damit die Tasks sicher ausgeführt werden, muss für die CPU Auslastung (*CPU Load*) gelten:

$$\mathrm{CL} = \sum_{k=1}^{\mathrm{N}} T_{\mathrm{E,k}} / T_{\mathrm{P,k}} < 100\,\% \tag{7.1}$$

- Die Verzögerung zwischen der Aktivierung einer Task k bis zu deren Ende, d. h. bis die Task vollständig ausgeführt ist, wird als Antwortzeit (*Response Time*) $T_{\mathrm{R,k}}$ bezeichnet. In Echtzeitsystemen gibt es in der Regel obere Grenzwerte (*Deadline*) $T_{\mathrm{D,k}}$ für die Antwortzeiten. Sinnvollerweise ist $T_{\mathrm{D,k}} \leq T_{\mathrm{P,k}}$.
- Task M sei die Task, deren Antwortzeit abgeschätzt werden soll. EP(M) beschreibt die Menge der Tasks, die dieselbe Priorität (*Equal Priority*) haben, einschließlich Task M selbst. HP(M) ist die Menge der Tasks, die eine höhere Priorität haben. LPNP(M) ist die Menge der Tasks, die eine niedrigere Priorität haben *und* als *Non Preemptive* konfiguriert sind. LPRS(M) ist die Menge aller Tasks mit niedrigerer Priorität, die eine Ressource verwenden, die auch von Task M verwendet wird.

Im günstigsten Fall beginnt die Ausführung der Task M unmittelbar nach ihrer Aktivierung und wird nicht durch Interrupts oder höher priore Tasks unterbrochen. Ihre minimale Antwortzeit ist daher

$$T_{\mathrm{R,M,min}} = T_{\mathrm{E,M}} \tag{7.2}$$

Die maximale Antwortzeit kann wie folgt abgeschätzt werden:

$$T_{\mathrm{R,M,max}} \leq \sum_{k \in \mathrm{EP(M)}} T_{\mathrm{E,k}} + \max_{k \in \mathrm{LPNP(M)},\, \mathrm{j} \in \mathrm{LPRS(M)}} \left(T_{\mathrm{E},k},\, T_{\mathrm{RS,j}} \right) + \sum_{k \in \mathrm{HP(M)}} \left\lceil \frac{T_{\mathrm{R,M,max}}}{T_{\mathrm{P},k}} \right\rceil T_{\mathrm{E},k} \tag{7.3}$$

Der erste Term beschreibt die Ausführungszeit der betrachteten Task M selbst sowie die Verzögerung, die von Tasks verursacht wird, die dieselbe Priorität haben. Diese werden

Tab. 7.5 Einfaches OSEK OS System

Task	Laufzeit T_E	Periode T_P	Priorität	$T_{R,min}$	$T_{R,max}$		
					Startwert	1. Iteration	2. Iteration
A	1 ms	3 ms	Hoch	1 ms	1 ms	1 ms	
B	1 ms	6 ms	Mittel	1 ms	1 ms	2 ms	2 ms
C	1 ms	15 ms	Nieder	1 ms	1 ms	3 ms	3 ms

zuerst ausgeführt, falls sie bereits aktiviert sind, wenn Task M aktiviert wird. Tasks gleicher Priorität werden nämlich vom OSEK Scheduler in einer Warteschlange in der Reihenfolge der Aktivierung (FIFO-Prinzip) verwaltet.

Der letzte Term beschreibt die Ausführungszeit höher priorer Tasks, die den Start von Task M verzögern oder ihre Ausführung unterbrechen. Dabei wird berücksichtigt, dass die höher prioren Tasks bei größerer Antwortzeit eventuell auch mehrfach aktiv werden können. Der Ausdruck in … ist auf den nächsten ganzzahligen Wert aufzurunden. Falls Task M als *Non Preemptive* konfiguriert wurde, können höher priore Tasks den Start von Task M zwar verzögern. Sobald Task M aber läuft, kann sie in diesem Fall nur noch von Interrupt Service Routinen unterbrochen werden.

Der mittlere Ausdruck steht für die größte Dauer, für die der Start der betrachteten Task M durch Tasks mit niedrigerer Priorität blockiert werden kann. Eine derartige Blockade kann durch eine Task mit niedrigerer Priorität erfolgen, wenn diese als *Non Preemptive* konfiguriert ist und bei Aktivierung von Task M gerade ausgeführt wird. Andererseits kann Task M aufgrund des *Priority Ceiling* Konzepts auch durch eine niedriger priorisierte Task, die eigentlich als *Preemptive* konfiguriert ist, blockiert sein, solange diese eine Ressource reserviert hält, die Task M ebenfalls verwenden will. T_{RS} sei die maximale Reservierungsdauer dieser Ressource.

Gleichung 7.3 für die Berechnung der Antwortzeit von OSEK und AUTOSAR Tasks hat eine ähnliche Struktur wie Gl. 3.6 für die Berechnung der Latenzzeiten von CAN-Botschaften. Wie dort kann auch diese Gleichung nur iterativ gelöst werden.

Ein einfaches Beispiel mit drei Tasks, die alle als präemptiv konfiguriert sind und keine gemeinsamen Ressourcen verwenden, zeigt Tab. 7.5. Die Priorität der Tasks ist dabei nach dem Verfahren des *Deadline Monotonic* bzw. *Rate Monotonic Scheduling* festgelegt, d. h. je kürzer die Deadline bzw. die Periodendauer einer Task ist, desto höher wird deren Priorität gewählt. Für rein präemptive Betriebssysteme ergibt dies die kürzest möglichen Antwortzeiten.

Der in Gl. 7.3 betrachtete *Worst Case* Fall, der zu den längsten Antwortzeiten und zum größten Jitter führt, ergibt sich, wenn praktisch alle Tasks gleichzeitig aktiviert werden. Dies kann verhindert oder zumindest reduziert werden, wenn die Periodendauern der Tasks als ganzzahlige Vielfache gewählt und geeignete Phasenverschiebungen zwischen den Aktivierungszeitpunkten festgelegt werden wie in Abb. 7.16. Wie bei CAN ist deren

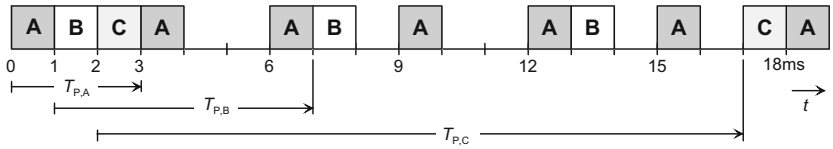

Abb. 7.16 Optimierter Task-Ablauf

Berücksichtigung bei der Worst Case Analyse recht aufwendig [6, 7], so dass eine Werkzeugunterstützung sinnvoll ist (siehe Abschn. 8.8).

7.3 Hardware-Ein- und Ausgabe (HIS IO Library, IO Driver)

Die von der Hersteller-Initiative Software HIS vorgeschlagene Architektur für Hardwarezugriffe ähnelt der Struktur gängiger Betriebssysteme für „richtige" Computer (Abb. 7.17).

Gegenüber der Anwendungssoftware wird eine logische Zwischenschicht (*IO Library*) mit einer generischen Schnittstelle definiert. Die Zwischenschicht stellt den Anwendungen im Wesentlichen eine Funktion zum Lesen (`Read` bzw. `Get`) und eine Funktion zum Schreiben (`Write` bzw. `Set`) von Datenbytes zur Verfügung. Bei Bedarf kann die Anwendung die Peripheriekomponente, z. B. die PWM-Ausgabeeinheit eines Mikrocontrollers oder einen CAN-Kommunikationscontroller, initialisieren (`Init`) und parametrieren (`Ioctl` sowie treiberspezifische Funktionen) bzw. wieder deinitialisieren (`DeInit`), falls die Komponente nur zeitweise verwendet wird. Für jede Peripheriekomponente wird ein Hardwaretreiber (*IO Driver*) verwendet, der den Zugriff auf diese Komponente kapselt. Die Schnittstelle zwischen der Zwischenschicht und dem Hardwaretreiber entspricht exakt der Schnittstelle zwischen Anwendung und *IO Library*, d. h. die Zwischenschicht reicht die Aufrufe im Wesentlichen (mit Ausnahme bei gepufferten asynchronen Aufrufen, sie-

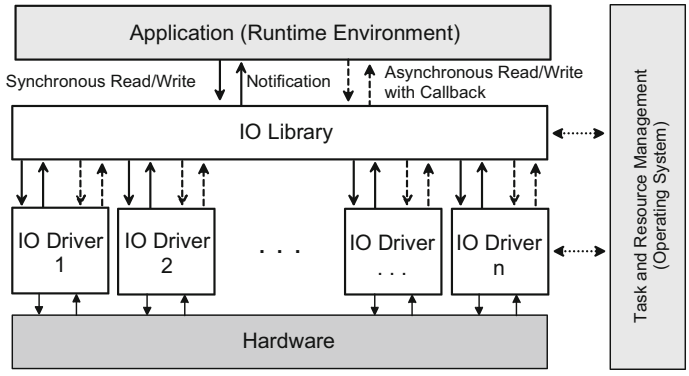

Abb. 7.17 Architektur für Hardwarezugriffe nach HIS

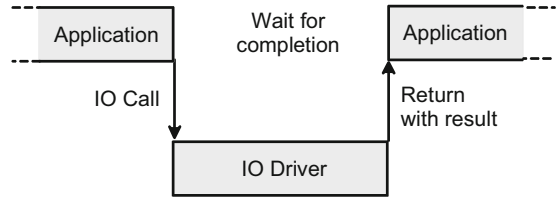

Abb. 7.18 Synchrone Ein-Ausgabe-Schnittstelle

he unten) nur durch. In der praktischen Implementierung kann die *IO Library* z. B. durch C-Makros realisiert werden, so dass durch die logische Zwischenschicht kein Laufzeitoverhead entsteht.

Die Schnittstellenfunktionen existieren in einer synchronen und in einer asynchronen Variante. Dabei muss ein Hardwaretreiber nur eine der beiden Varianten unterstützen. Bei der *synchronen* Variante wartet das aufrufende Programm, bis die gewünschte Lese- oder Schreiboperation ausgeführt ist (Abb. 7.18). Für die Mehrzahl der Hardwarebaugruppen, bei denen in der Regel nur einfache Register oder Speicherzellen gelesen oder geschrieben werden müssen, ist dies die bevorzugte und am einfachsten zu realisierende Variante. Bei Peripheriekomponenten, bei denen der Zugriff länger dauern kann, z. B. beim Schreiben von EEPROMs, wird man die *asynchrone* Variante bevorzugen (Abb. 7.19). Dort stößt die Anwendung den Lese- oder Schreibvorgang nur an und arbeitet weiter. Der Hardwaretreiber führt den Hardwarezugriff selbstständig durch und meldet sich auf Wunsch, wenn der Vorgang abgeschlossen ist, durch Aufruf einer von der Anwendung bereitgestellten Funktion (*Callback*). Da hierbei Anwendung und Hardwaretreiber (scheinbar) parallel ablaufen, setzen die asynchronen Funktionen eine Interrupt-Verarbeitung oder ein Multitasking-Betriebssystem voraus, in der Regel OSEK/VDX, das die Rechenzeit der CPU so verteilt, dass sowohl die Anwendung als auch der Hardwaretreiber ausgeführt werden. Sowohl in der synchronen als auch in der asynchronen Variante können die Hardwaretreiber so konfiguriert werden, dass sie beim Auftreten eines Fehlers (*Error Hook*) oder bestimmter Ereignisse in der Hardware, z. B. wenn sich der Zustand eines Digitaleingangs verändert, eine von der Anwendung bereitgestellte Funktion aufrufen (*Notification Callback*). Für die Implementierung der ereignisgetriggerten Rückruffunktionen ist auf Mikrocontrollerebene eine interruptfähige Peripheriehardware notwendig oder der Hardwaretreiber muss wieder Dienste des Multitasking-Betriebssystems verwenden.

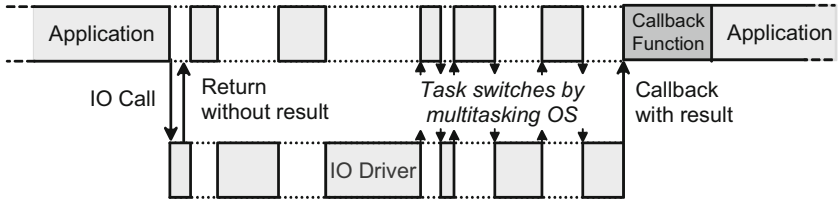

Abb. 7.19 Asynchrone Ein-Ausgabe-Schnittstelle

Um eine schlanke Implementierung auch auf kleineren Mikrocontrollern zu ermöglichen, muss ein einzelner Hardwaretreiber gleichzeitig nur einen einzigen Aufruf bedienen können (*Non reentrant*). Falls er nochmals aufgerufen wird, bevor der vorherige Aufruf abgeschlossen ist, darf er den erneuten Aufruf mit einer Fehlermeldung ablehnen. Es ist Aufgabe der Anwendung, solche Konflikte zu vermeiden. Dazu kann die Anwendung abfragen (*Check*), ob der Hardwaretreiber noch mit einem vorhergehenden Aufruf beschäftigt ist. Optional kann die *IO Library*-Zwischenschicht die asynchronen Aufrufe in Form eines First In-First Out-Zwischenspeichers puffern. Die Zwischenschicht wird in diesem Fall aufwendiger, während der Hardwaretreiber weiterhin nur einen einzelnen Aufruf bedient.

7.4 HIS Hardwaretreiber für CAN-Kommunikationscontroller (HIS CAN Driver)

Neben den Treibern für Mikrocontroller-Standardkomponenten wie digitalen Ein- und Ausgängen, PWM- und ADC-Einheiten usw. definierte HIS einen Treiber für CAN. Die Grundkonzepte des HIS CAN Treibers haben Eingang in die entsprechenden Spezifikationen des AUTOSAR-Kommunikationsstacks gefunden, der in Abschn. 8.3 beschrieben wird. Der HIS CAN Treiber sowie die HIS Hardwaretreiber, die in den ersten beiden Auflagen dieses Buches noch ausführlicher beschrieben wurden, spielen in der Praxis heute keine Rolle mehr sondern wurden durch die entsprechenden AUTOSAR Treiber abgelöst.

7.5 HIS Flash-Lader

Der Flash-Lader oder Bootlader ist eine eigenständige Softwarekomponente, mit der das Erst- und Nachprogrammieren des Steuergerätespeichers (*Flashen*) durchgeführt werden kann. Sie muss bereits dann funktionsfähig sein, wenn das Steuergerät noch kein Betriebssystem und keine Anwendungssoftware enthält bzw. wenn diese durch eine neuere Version ersetzt werden sollen. Der Flash-Lader muss daher autark sein und die notwendigen Funktionen für eine einfache Ablaufsteuerung, Hardwaretreiber für das Flash-ROM und die Kommunikationsschnittstelle sowie einen rudimentären Protokollstapel für das Diagnoseprotokoll mitbringen.

Mitte 2006 veröffentlichte HIS die Spezifikationen für einen vollständigen Flash-Lader, nachdem bereits 2002 ein reiner HIS Flash-Treiber spezifiziert worden war. Der HIS Flash-Lader definiert die komplette Infrastruktur für den Flash-Prozess, verwendet für die hardwarespezifischen Teile nun aber die entsprechenden AUTOSAR-Komponenten (siehe Kap. 8). Ansonsten ist er unabhängig vom AUTOSAR-Laufzeitsystem. Außerdem gibt der HIS-Vorschlag vor, welche Sequenz von UDS-Diagnosediensten für die Programmierung verwendet werden soll. Eine ausführliche Darstellung des Flash-Prozesses auf Basis des HIS Flash-Laders findet sich in Abschn. 9.4.

7.6 Normen und Standards zu Kapitel 7

OSEK	OSEK/VDX Operating system specification, Version 2.2.3, 2005 OSEK/VDX Communication Version 3.0.3, 2004, www.osek-vdx.org OSEK/VDX Network management Version 2.5.3, 2004, www.osek-vdx.org OSEK/VDX System generation – OIL: OSEK Implementation Language Version 2.5, 2004, www.osek-vdx.org OSEK/VDX Time-Triggered Operating System Version 1.0, 2001 OSEK/VDX Fault-Tolerant Communication Version 1.0, 2001 OSEK/VDX Run-Time Interface Version 2.2, 2005, www.osek-vdx.org OSEK/VDX Binding specification Version 1.4.2, 2004, www.osek-vdx.org ISO 17356-1 Road vehicles – Open interface for embedded automotive applications – Part 1: General structure, 2005, www.iso.org ISO 17356-2 Road vehicles – Open interface for embedded automotive applications – Part 2: OSEK/VDX Binding specification, 2005, www.iso.org ISO 17356-3 Road vehicles – Open interface for embedded automotive applications – Part 3: OSEK/VDX Operating system (OS), 2005, www.iso.org ISO 17356-4 Road vehicles – Open interface for embedded automotive applications – Part 4: OSEK/VDX Communication (COM), 2005, www.iso.org ISO 17356-5 Road vehicles – Open interface for embedded automotive applications – Part 5: OSEK/VDX Network management (NM), 2006, www.iso.org ISO 17356-6 Road vehicles – Open interface for embedded automotive applications – Part 6: OSEK/VDX Implementation language (OIL), 2006 OSEK OS Requirements for protected applications under OSEK Version 1, 2002, www.automotive-his.de OSEK OS Extensions for protected applications Version 1.0, 2003, www.automotive-his.de
HIS	HIS API IO Library Version 2.0.3, 2004, www.automotive-his.de HIS API IO Driver Version 2.1.3, April 2004, www.automotive-his.de HIS CAN driver functional overview and configuration Version 1.01, 2004, www.automotive-his.de HIS CAN driver specification Version 1.0, 2003, www.automotive-his.de HIS Functional specification of a flash driver Version 1.3, 2002, www.automotive-his.de HIS flash-driver calling conventions Version 1.0, Juni 2003, www.automotive-his.de HIS Flash Loader Specification Version 1.1, Juni 2006, www.automotive-his.de HIS–konforme Programmierung von Steuergeräten auf Basis von UDS Version 1.0, September 2006, www.automotive-his.de HIS Security Module Specification Version 1.1, Juli 2006, www.automotive-his.de HIS CAL – Cryptographic Abstraction Layer Version 1.0, 2007, www.automotive-his.de HIS Funktions-Freischaltung „light“ – Format FSC Version 1.7, 2007, www.automotive-his.de

Literatur

[1] J. Schäuffele, T. Zurawka: Automotive Software Engineering. Springer-Vieweg Verlag, 5. Auflage, 2013

[2] M. Homann: OSEK. Betriebssystem-Standard für Automotive und Embedded Systems. Mitp-Verlag, 1. Auflage, 2005

[3] J. Lemieux: Programming in the OSEK/VDX environment. CMP Books-Verlag, 2001

[4] N. Audsley, A. Burns, M. Richardson. K. Tindell, A. Wellings: Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling. Software Engineering Journal, Heft 5, 1993, S. 284–292

[5] W. Lei, W. Zhaohui, Z. Mingde: Worst-Case Response Time Analysis for OSEK/VDX Compliant Real-time Distributed Control Systems. Proceedings der IEEE International Computer Software and Applications Conference, 2004, S. 148–153

[6] J.C. Palencia, M.G. Harbour: Schedulability Analysis for Tasks with Static and Dynamic Offsets. Proceedings des IEEE Real-Time Systems Symposiums, 1998, S. 26–37

[7] R. Racu, R. Ernst, K. Richter, M. Jersak; A Virtual Platform for Architectural Integration and Optimization in Automotive Networks. SAE Transactions Vol 116: Journal of Passenger Car Electronic and Electrical Systems, 2007, S. 372–380