AUTOSAR-Softwarearchitektur für Kfz-Systeme 8

Die *Automotive Open System Architecture* (AUTOSAR) Initiative, eine Konsortium, dem mittlerweile alle führenden Automobilhersteller und Zulieferer angehören, übernahm die Vorarbeiten von OSEK und HIS aus den 1990er Jahren und definiert heute eine vollständige Softwarearchitektur für Steuergeräte.

8.1 Einführung

Die AUTOSAR Softwarearchitektur entkoppelt die Anwendungssoftware weitgehend von der Hardware der Steuergeräte. Die Software besteht aus Funktionsmodulen, den Softwarekomponenten, die unabhängig voneinander und durch verschiedene Hersteller entwickelt und dann in einem weitgehend automatisierten Konfigurationsprozess zu einem konkreten Projekt zusammengebunden werden sollen (Abb. 8.1).

Die Entkopplung zwischen Hardware und Software sowie zwischen den verschiedenen Softwarekomponenten erfolgt durch ein Grundsoftwarepaket (*Basic Software*), das neben der Mikrocontroller- und Steuergeräte-Abstraktionsschicht (*ECU and Microcontroller Hardware Abstraction Layer HAL*) im sogenannten *Service Layer* anwendungsunabhängige Dienste wie das Betriebssystem, Kommunikationsprotokolle und eine Speicherverwaltung enthält. Das Zusammenspiel der Softwarekomponenten der Fahrsoftware erfolgt über eine Zwischenschicht, das AUTOSAR *Run Time Environment*, das insbesondere den Datenaustausch regelt und gelegentlich auch als *Virtual Function Bus* bezeichnet wird. Die Grundidee geht dabei so weit, dass die Softwarekomponenten sogar beliebig auf unterschiedliche Geräte verteilt werden sollen, ohne dass sich, abgesehen von anderen Latenzzeiten, funktionale Unterschiede ergeben oder eine Änderung der Implementierung innerhalb der Komponenten notwendig wird.

Bezüglich der Basissoftware setzt AUTOSAR auf die Vorarbeiten von OSEK, HIS, ASAM und ISO sowie der Industriekonsortien für CAN, FlexRay und LIN. Die dort definierten Konzepte und Standards für Betriebssystem, Hardwaretreiber und Protokol-

W. Zimmermann und R. Schmidgall, *Bussysteme in der Fahrzeugtechnik*, ATZ/MTZ-Fachbuch, 367
DOI 10.1007/978-3-658-02419-2_8, © Springer Fachmedien Wiesbaden 2014

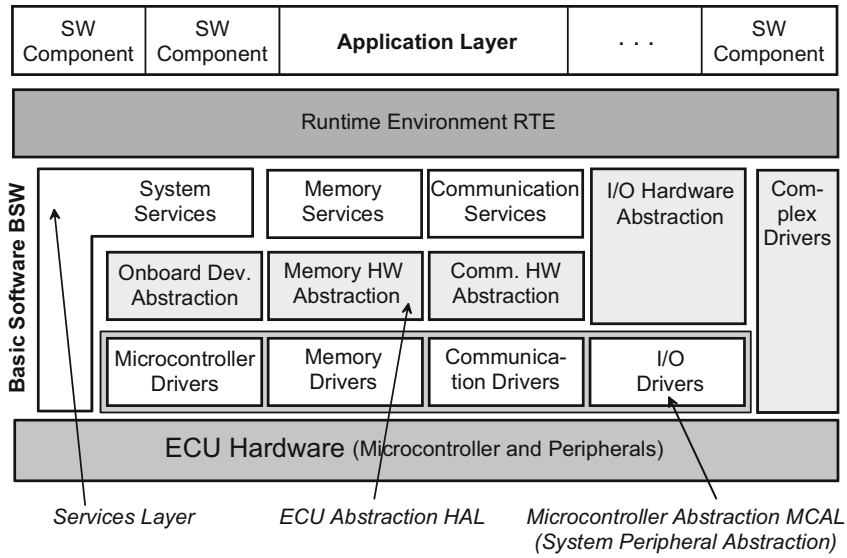

Abb. 8.1 Schichtenmodell der AUTOSAR-Basissoftware

le wurden teilweise übernommen, wobei aber neben funktionalen Erweiterungen die Durchgängigkeit der Schnittstellen und Kommunikationsmechanismen innerhalb von AUTOSAR zu Lasten der völligen Rückwärtskompatibilität angestrebt wird.

Wie frühere Initiativen versteht sich auch AUTOSAR nur als Standardisierungsgremium, das Spezifikationen erarbeitet, aber keine verbindliche Implementierung vorschreibt, sondern dies kommerziellen Anbietern im freien Wettbewerb überlässt („Cooperate on standards, compete on implementation"). Da der AUTOSAR-Ansatz allerdings sehr komplex ist, fördert die Initiative prototypische Referenzimplementierungen, die die Machbarkeit nachweisen. Die Softwarezulieferer und Werkzeuganbieter, die solche Referenzimplementierungen im Rahmen von AUTOSAR entwickeln, erhoffen sich davon natürlich einen späteren Wettbewerbsvorsprung und versuchen verständlicherweise auch, bereits in ihren Häusern existierende Lösungen in den Standardisierungsprozess einfließen zu lassen. Mitte 2006 wurde die Spezifikation der Basissoftware als AUTOSAR 2.0 erstmals allgemein zugänglich gemacht, die Spezifikationen waren teilweise aber noch unvollständig oder vorläufig. Weitere Ergänzungen und Vervollständigungen folgten als AUTOSAR 3.0/3.1 im Jahr 2008 und 4.0 Ende 2009. Mittlerweile leidet auch AUTOSAR unter dem üblichen Nebeneinander unterschiedlicher Standardversionen. Verschiedene Hersteller hatten bereits Serienprojekte auf Basis von Version 3.0/3.1 aufgesetzt, die aus Zeitgründen nicht mehr vollständig auf Version 4.x umgestellt werden konnten. Einige auch für die laufenden Serienprojekte wichtigen Teilkonzepte wurden aber erst mit Version 4.0 spezifiziert. Daher wurde nachträglich noch eine Version 3.2 veröffentlicht, die einige der neuen Konzepte aus Version 4 rückwärts in die alte Version 3.x übernimmt. Parallel dazu wurde 2013 eine

weiterentwickelte Version 4.1 veröffentlicht. Weitere Versionen sind im Zeitabstand von etwa 2 Jahren angekündigt.

Erhebliche Fortschritte bei der Qualität erhoffen sich die AUTOSAR-Träger durch die Automatisierung der Softwareintegration für ein konkretes Steuergerät (Abb. 8.2). Die einzelnen Softwarekomponenten werden konventionell oder modellgestützt mit Hilfe von Werkzeugen wie Matlab/Simulink, Ascet oder TargetLink und den zugehörigen Code-Generatoren entwickelt und sorgfältig getestet [1]. Zusätzlich zum eigentlichen Programmcode stellt der Komponentenlieferant eine Beschreibungsdatei (*SW Component Description*) bereit, die die Eigenschaften der Komponente, insbesondere die Schnittstellen, RAM/ROM-Bedarf, Laufzeitbedarf usw. angibt. In derselben Weise liefern die Steuergerätehersteller Beschreibungsdateien (*ECU Resource Description*), die die Eigenschaften der Steuergeräte wie Rechenleistung, Speichergrößen, Anzahl der Ein- und Ausgänge usw. präzise definieren. Als dritte Quelle dient eine Beschreibung der gewünschten Systemfunktionen und der geforderten Randbedingungen (*System Constraint Description*). Mit Hilfe eines Generierungswerkzeugs werden die Funktionen auf verschiedene Steuergeräte aufgeteilt (*System Configuration*) und die notwendigen Ressourcen zugeordnet (*ECU Configuration*). Daraus erzeugt ein weiteres Werkzeug dann die reale Softwareimplementierung für die einzelnen Steuergeräte. Dabei wird insbesondere die RTE-Softwareschicht, die die Kommunikation zwischen den Softwarekomponenten sicherstellt und überwacht, automatisch generiert.

8.2 Überblick über die AUTOSAR-Basissoftware

Die Basissoftware lässt sich in die vier vertikalen Säulen Systemdienste, Speicherverwaltungsdienste, Kommunikationsdienste und Hardware-Ein-/Ausgabedienste unterteilen (Abb. 8.3), wobei jede dieser Säulen horizontal aus den drei Schichten *Service Layer*, *ECU Abstraction Layer* und *Microcontroller Abstraction Layer* besteht. Als fünfte Säule steht mit den monolithischen *Complex Drivers* eine Möglichkeit zur Verfügung, die Schichtenarchitektur zu umgehen. Dies kann bei Peripheriekomponenten notwendig sein, bei denen die konventionelle Schichtenaufteilung die geforderten Zeitbedingungen nicht einhalten kann, z. B. bei der Ansteuerung von Ventilendstufen mit Mikrosekundenauflösung bei gleichzeitiger Auswertung von Ventilrückmeldesignalen in Echtzeit. Gleichzeitig erlauben *Complex Drivers* auch eine Migrationsstrategie, bei der existierende Module in einer Übergangsphase zunächst nur mit AUTOSAR-konformen Schnittstellen zum *Runtime Layer* versehen und erst im Lauf der Zeit an das geforderte Schichtenmodell angepasst werden. Für jede Säule und jede einzelne Schicht sieht AUTOSAR mindestens ein, meist aber mehrere Softwaremodule vor.

Hardwaretreiber und Hardwareabstraktion Während die Schnittstellen des *Service Layers* und der darüber liegenden Komponenten hardwareunabhängig sein sollen, sind die unteren beiden Schichten der Basissoftware hardwarespezifisch (Abb. 8.1 und 8.3). Die zunächst

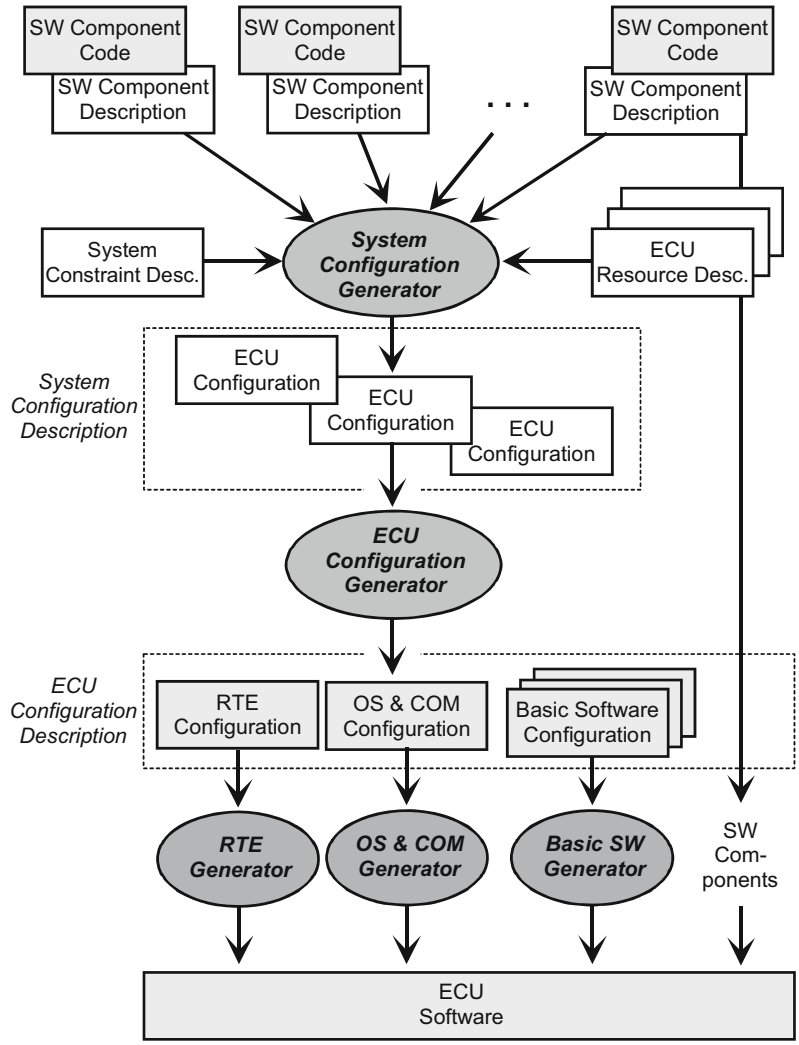

Abb. 8.2 AUTOSAR-Werkzeugkette

willkürlich erscheinende Unterscheidung in zwei hardwarenahe Schichten signalisiert die Vorstellung des AUTOSAR-Konsortiums, wer das jeweilige Modul bereitstellen wird. Module des *Microcontroller Abstraction Layer* (MCAL), der anfangs auch *System Peripheral Abstraction Layer* (SPAL) genannt wurde, steuern die Mikrocontroller-interne Peripherie an und müssen vom Hersteller des Prozessors geliefert werden. Module des *ECU Abstraction Layers HAL* sind für externe Peripherie-ICs zuständig und sollen von deren Lieferanten oder vom Hersteller des Steuergerätes zur Verfügung gestellt werden. Letzterer wird in der Regel auch die restliche Basissoftware bereitstellen. Dabei werden OEMs ihren Gerätelie-

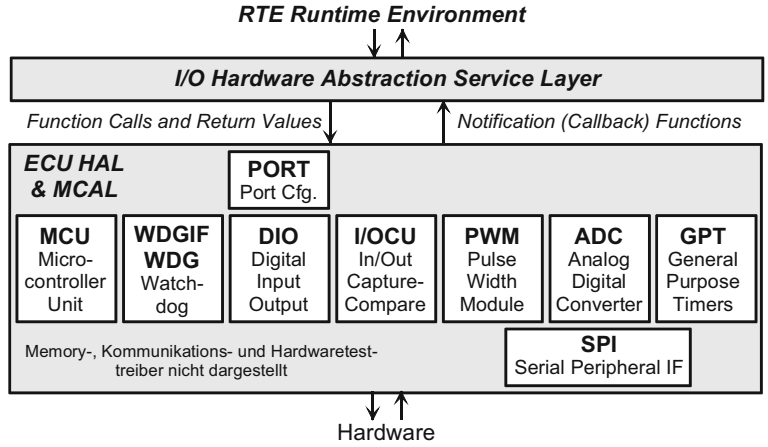

Abb. 8.3 Abstraktionsschichten für die Hardwareperipherie

feranten verstärkt den Einsatz der generischen Basissoftwarepakete vorschreiben, die von einigen Softwarehäusern angeboten werden.

Die von AUTOSAR definierten Hardwaretreiber folgen im Wesentlichen denselben Konzepten und stellen ähnliche Funktionen zur Verfügung, wie sie bereits von HIS definiert wurden (Abschn. 7.3). Die konkreten Programmierschnittstellen (*Application Programming Interface API*) sind aber nicht direkt HIS-kompatibel. Die wesentlichsten Treibermodule sind in Tab. 8.1 aufgeführt.

Zusätzlich zu den in Abb. 8.3 dargestellten Treibern gibt es noch spezielle Treiber für Speicherbausteine und Kommunikationscontroller, die Teil der *Memory Services* bzw. des Kommunikationsstacks sind. Daneben sind noch einige Module mit Selbsttestfunktionen z. B. für den Mikrocontrollerkern, den RAM- oder den Flash-ROM-Speicher vorgesehen, die beim Systemstart, auf Anforderung von Diagnosediensten oder zyklisch im Hintergrund ablaufen können.

Jeder Treiber besitzt eine Initialisierungsfunktion `ModulName_Init()`, mit der die Hardwarebaugruppe konfiguriert wird, sowie eine Deinitialisierungsfunktion `ModulName_DeInit()`. Die Mehrzahl der Treiber verwendet eine *synchrone Schnittstelle*, d. h. die Funktion erledigt beim Aufruf ihre Aufgabe in kurzer Zeit vollständig und endet mit der Rückgabe des Ergebnisses bzw. einer Status- oder Fehlerinformation an das aufrufende Programm (vgl. Abb. 7.18 und 7.19). In der Regel sind diese Funktionen wiedereintrittsfest (*reentrant*), d. h. eine laufende Abfrage darf durch eine andere Abfrage aus einer anderen präemptiven Task unterbrochen werden.

Treiber für Aufgaben, die längere Zeit dauern, wie beispielsweise die Wandlung mehrerer Analog-Digital-Eingangssignale oder das Löschen eines Speicherblocks im Flash-ROM, arbeiten *asynchron*. In diesem Fall startet der Aufruf der entsprechenden API-Funktion die Bearbeitung lediglich und kehrt sofort zum aufrufenden Programm zurück. Die eigentliche

Tab. 8.1 Treiber für Hardwarekomponenten

Mikrocontroller Grundfunktionen	
Mikrocontroller (MCU)	Initialisierung der Taktgeneratoren (*Clock*, *PLL*), des/der CPU-Betriebsmodi und Speicherbereiche, Abfrage des Reset-Grundes bzw. Auslösen eines Resets. Die Grundinitialisierung der CPU erfolgt vor Aufruf des MCU-Treibers durch den *Startup Code* des Mikrocontrollers, der nicht Teil des AUTOSAR-Standards ist.
Digital- und Analog-Ein-/Ausgabe sowie Zeitgeber	
Schaltsignale (Digital-Ein-/Ausgänge) (DIO, PORT Digital Input/Output)	Schreiben und Lesen eines einzelnen Ein/Ausgangs (*Channel*), mehrerer Ein/Ausgänge (*Group*) oder eines ganzen 8 bzw. 16bit Microcontroller-*Ports* (DIO-Treiber). Die Konfiguration der Ein/Ausgänge (auch der ICU-Ein- bzw. PWM-Ausgänge) erfolgt über die Funktionen des PORT-Treibers.
Pulsbreitenmodulierte Ausgangssignale (PWM Pulse Width Modulated Outputs)	Setzen der Frequenz und des Tastverhältnisses eines PWM-Ausgangssignals Setzen und Lesen des Ausgangswertes Rückruffunktion (*Notification Callback*) bei steigender und/oder fallender Signalflanke an einem PWM-Ausgang.
Impulssignal-Ein- und Ausgänge (ICU Input Capture Unit) (OCU Output Compare Unit)	Messen von Zeitpunkten, Zeitabständen und der Anzahl von Signalwechseln bei positiven und/oder negativen Flanken von Eingangssignalen. Erzeugen von Impulssignalen zu definierten Zeitpunkten. Rückruffunktion (*Notification Callback*) bei steigender und/oder fallender Signalflanke an einem Eingang bzw. nach Vorliegen von Messwerten.
Zeitgeber (General Purpose Timer GPT)	Starten und Stoppen von einmaligen (*One Shot*) oder periodischen (*Continuous*) Zeitgebern. Abfrage der abgelaufenen Zeit bzw. der Zeit bis zum Ablauf. Rückruffunktion (*Notification Callback*) beim Ablauf eines Zeitgebers.
Analoge Eingänge (ADC Analog to Digital Converter Inputs)	Messen eines einzelnen (*Channel*) analogen Eingangssignals oder einer Gruppe von Eingangssignalen (*Group*). Die Messung kann als Einzelmessung (*One Shot*) oder kontinuierlich (*Continuous*) durchgeführt werden. Das Starten der Messung kann durch einen Software-Funktionsaufruf (*On Demand*), durch das Ereignis (*Trigger*) eines Timer-Kanals oder eines Hardware-Eingangssignals erfolgen. Rückruffunktion (*Notification Callback*), wenn Messwerte vorliegen bzw. der Messwertpuffer gefüllt ist.
Watchdog (WDG, WDGIF)	Konfigurieren und Triggern des Watchdogs (WDG). Das *Watchdog Interface* (WDGIF) ist eine Zwischenschicht für den Fall, dass es mehr als einen Watchdog im System gibt. Die eigentliche Überwachung der Anwendungssoftware erfolgt durch den Systemdienst *Watchdog Manager*.

Tab. 8.1 (Fortsetzung)

Serial Peripheral Interface (SPI)	Treiber für die Anbindung externer Bausteine, z. B. EEPROMs oder A/D-Umsetzer, die über den SPI-Bus an die CPU angeschlossen sind. Verschiedene Ausbaustufen (*Level*), die nur einen oder aber mehrere Kommunikationsaufträge (*Job, Sequence*) gleichzeitig abwickeln können und wahlweise interne Datenpuffer (*Buffer*) bereitstellen.
Speicherbausteine	
Ansteuerung von internen und externen Flash-ROM und EEPROM-Bausteinen siehe Abschnitt *Memory Services*	
Kommunikationsschnittstellen	
Treiber für interne und externe CAN-, LIN-, FlexRay- oder Ethernet-Controller siehe Abschnitt *Kommunikationsstack*	

Bearbeitung erfolgt dann im Hintergrund entweder automatisch durch die Hardware oder mit Softwareunterstützung. Für diesen Fall verfügt der Treiber über interne Funktionen, die vom Betriebssystem zyklisch aufgerufen werden. Die Anwendung kann den Stand der Bearbeitung entweder selbst mit Hilfe einer entsprechenden API-Funktion abfragen (*Polling*) oder sich durch eine Rückruf-Funktion (*Callback* oder *Notification Function*) über die erfolgreiche Beendigung oder auftretende Fehler informieren lassen. Fehler werden meist zusätzlich auch an den *Diagnostic Event Manager* gemeldet, der eine Art zentralen Fehlerspeicher bildet.

Zwischen den Hardwaretreibern und dem RTE-Laufzeitsystem bzw. den Anwendungen liegt die *I/O Hardware Abstraction* Ebene, die die hardwarenahen Register-, Port- und Pinwerte auf logische *Signale* für die Anwendungsebene abbilden soll. Im programmtechnischen Sinn sind *Signale* Variablen mit einem bestimmten Wertebereich und einer bestimmten, durch den Datentyp definierten Auflösung. Die Spezifikation teilt *Signale* in die Klassen analoge Signale (Spannungen, Ströme, veränderliche Widerstände), diskrete Signale mit einem oder mehreren Bits (*Discrete Signal Group*), pulsbreitenmodulierte Signale mit Periodendauer und Tastverhältnis sowie Fehlerinformationen (*Diagnosis Class*) ein. Unter letzteren werden Statusinformationen über Kurzschlüsse und Unterbrechungen usw. an den Anschlüssen bzw. in den Endstufen verstanden. Bei der Konfiguration der Basissoftware sollen für alle *Signale* Typ (Klasse), Datenrichtung (Ein- oder Ausgang), Wertebereich, physikalische Einheit, Auflösung und Genauigkeit angegeben werden. Ob und wie die *Signale* zu filtern oder zu entprellen sind und welche Art der Fehlerüberwachung (Signal Range Check, Plausibilität, …) notwendig ist, lässt die gegenwärtige Spezifikation jedoch weitgehend offen. Dementsprechend wird auch keine Programmierschnittstelle festgelegt, sondern lediglich allgemein gefordert, dass *I/O Hardware Abstraction* Module eine Initalisierungsroutine `IoHwAb_Init...()` sowie `IoHwAb_Get...()` und `IoHwAb_Set...()` Funktionen für den Zugriff auf die Signale bereitstellen sollen.

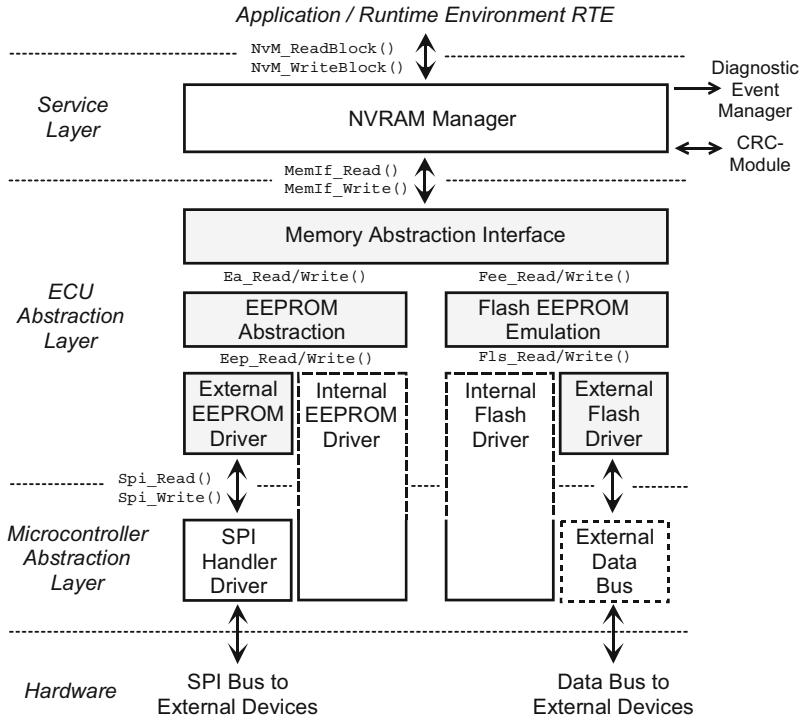

Abb. 8.4 Aufgabenverteilung am Beispiel der *Memory Services*

Memory Services Die Aufgabenteilung zwischen den einzelnen Schichten aus Abb. 8.1 und das Zusammenspiel verschiedener Module soll am Beispiel der *Memory Services* zur Verwaltung des nicht-flüchtigen Datenspeichers (*NVRAM Non Volatile Memory*) erläutert werden (Abb. 8.4). Der nicht-flüchtige Speicher kann aus einem Mikrocontroller-internen oder seriell bzw. parallel angeschlossenen EEPROM bestehen oder durch ein internes oder externes Flash-ROM simuliert werden.

Für die Anwendungsebene bildet der *Service Layer* mit dem *NVRAM Manager* die einzige Schnittstelle zu den nicht-flüchtigen Speichern. Im Gegensatz zu den unteren Schichten muss er Anforderungen mehrerer Anwendungstasks gleichzeitig bedienen können. Dazu speichert er alle Lese-, Schreib- oder Löschanforderungen in einer Warteschlange, priorisiert sie und reicht sie hintereinander (*Serialisierung*) an die darunter liegenden Schichten weiter. Die Anwendungstasks arbeiten währenddessen asynchron weiter. Zu bemerken ist, dass immer wenn im folgenden vom Zugriff von Anwendungskomponenten auf Komponenten der Basissoftware die Rede ist, in der realen Implementierung stets das *Runtime Environment RTE* zwischengeschaltet ist, ohne dass dies ausdrücklich erwähnt wird.

Die Daten im nicht-flüchtigen Speicher werden in Form von Blöcken verwaltet, deren Größe (max. 64 KB), Zuordnung und Adresslage zu den verschiedenen internen

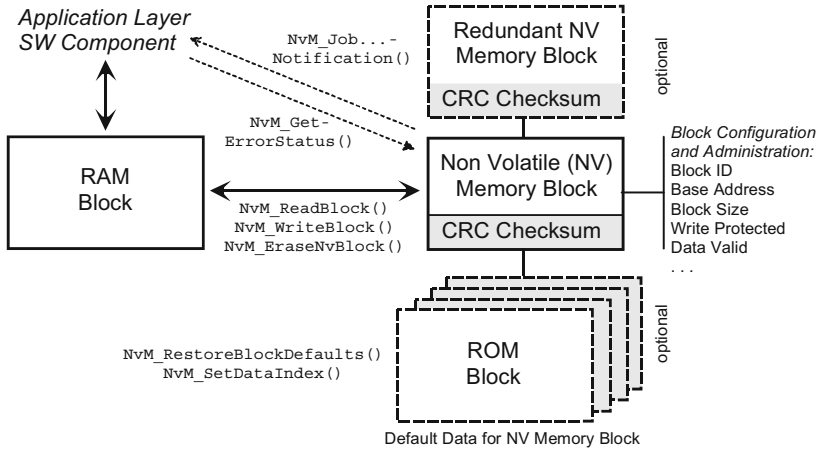

Abb. 8.5 Verwaltung von Daten im nicht-flüchtigen Speicher

und externen Speicherbausteinen statisch konfiguriert und in einer Tabelle festgelegt ist (Abb. 8.5). Die Auswahl der Datenblöcke aus Anwendungssicht erfolgt über sogenannte Blockidentifier, ohne dass die Anwendung den eigentlichen Speicherort kennen muss. Die Anwendung liest und schreibt dabei direkt jeweils nur eine Kopie der Daten im RAM. Diese Kopie wird bei der Initialisierung des *NVRAM Managers* für jeden Speicherblock automatisch erstellt und beim Stoppen wieder zurückgeschrieben. Die Anwendung kann aber auch explizit einen Datenblock von oder zum nicht-flüchtigen Speicher kopieren (`NvM_ReadBlock()`, `NvM_WriteBlock()`). Das Überschreiben der Daten im nicht-flüchtigen Speicher lässt sich durch entsprechende Konfiguration für jeden einzelnen Block explizit verhindern. Wahlweise werden die Blöcke durch eine CRC-Prüfsumme geschützt. Optional kann im nicht-flüchtigen Speicher eine redundante Kopie desselben Datenblocks oder im ROM ein Datensatz mit Ersatzwerten abgelegt werden, auf die im Fehlerfall, z. B. bei fehlerhafter CRC-Prüfsumme, umgeschaltet wird. Weiterhin ist es möglich, statt eines EEPROM-Datenblocks einen von mehreren ROM-Datenblöcken auszuwählen, um Datensatzvarianten für unterschiedliche Ausstattungs- oder Ländervarianten zu realisieren.

Unterhalb des *NVRAM Managers* liegt das *Memory Abstraction Interface*, das für jedes EEPROM bzw. Flash-ROM ein entsprechendes Submodul verwendet (Abb. 8.4). Diese Schnittstelle leitet die Schreib-, Lese- und Löschanfragen nach einer Adressumsetzung direkt an die darunter liegenden Treiber weiter. Dabei wird der *Block Identifier* der oberen Schicht auf die Bausteinauswahl und interne Adressierung abgebildet. Auch das Problem der begrenzten Lösch/Schreibzyklen bei EEPROM- bzw. Flash-ROM-Bausteinen wird auf dieser Ebene teilweise gelöst, indem die Zugriffe für entsprechend konfigurierte Adressbereiche automatisch auf mehrere unabhängige Speicherblöcke verteilt werden. Dies reduziert allerdings lediglich die Wahrscheinlichkeit von Speicherfehlern, eine echte Überprüfung der Daten erfolgt dabei nicht.

Der *EEPROM* bzw. *Flash Driver* stellt Routinen zum Lesen, Schreiben, Löschen und Vergleichen der Datenblöcke zur Verfügung. Die Funktionalität entspricht weitgehend der entsprechenden HIS-Spezifikation (siehe Abschn. 7.3 und 9.4.3). Die eigentliche Programmier-API ist aber nicht kompatibel, so dass vorhandene HIS-Treiber nicht direkt übernommen werden können. Im Fall eines Mikrocontroller-internen Speichers enthält der Treiber die vollständige Programmierlogik und wird als Teil des *Microcontroller Abstraction Layers* betrachtet. Im Fall eines externen Speicherbausteins gilt der Treiber dagegen als Teil des *ECU Abstraction Layers* und enthält nur die Baustein-spezifische Logik, während der eigentliche Hardwarezugriff über einen weiteren, darunterliegenden Treiber im *Microcontroller Abstraction Layer* erfolgt, der z. B. den Zugriff über den SPI-Bus durchführt und dabei auch konkurrierende Zugriffe an weitere, an diesem Bus angeschlossene Bausteine priorisiert und serialisiert.

System Services Zu den allgemeinen Systemdiensten, die AUTOSAR definiert, gehören

- *Betriebssystem und Steuerung des Systemzustands*
  Multitasking-Scheduling mit Mechanismen zur Synchronisation und Intertask-Kommunikation (AUTOSAR OS, siehe Abschn. 8.3) sowie die Steuerung des Betriebszustands des Steuergerätes (*ECU State Manager*) vom Einschalten (*Start up*) über Stromsparzustände (*Sleep*, *Wake up*), den Notlaufbetrieb im Fehlerfall (*Limp Home*) bis zum Abschalten (*Shutdown*). Die Überwachung des zeitlichen Ablaufs der Anwendungssoftware erfolgt durch den *Watchdog Manager*.
- *Bibliotheken mit Hilfsfunktionen*
  Funktionen, die Algorithmen für Regler, Interpolationen und sonstige Fest- und Gleitkommaberechnungen (IFL, MFL), Authentifizierungs-, Kompressions- und Verschlüsselungsverfahren (CAL, CSM), Prüfsummenberechnung (CRC) sowie Bitmanipulation (BFX) implementieren.
- *Fehlerspeicher, Funktionssteuerung und Diagnoseschnittstelle*
  Fehler, die in den Anwendungen und Modulen der Basissoftware auftreten, werden an ein zentrales Modul, den *Diagnostic Event Manager* (DEM) gemeldet, das einen Fehlerspeicher verwaltet und Schnittstellen zum Diagnosemodul (DCM) sowie zur Funktionssteuerung (FIM) aufweist.

Das AUTOSAR-Steuergerät kann sich in einem der im Abb. 8.6 dargestellten Zustände befinden. Die Zustände werden vom *ECU State Manager* (EcuM) verwaltet, der für die Initialisierung des Steuergerätes, der Basissoftware und den Start des Betriebssystems sowie dessen geordnetes Abschalten zuständig ist. Dauerhaft ist das Gerät entweder abgeschaltet (*Off*), eingeschaltet (*Run*) oder in einem Stromsparmodus (*Sleep*), die übrigen Zustände werden nur kurzzeitig beim Übergang zwischen diesen drei Grundzuständen durchlaufen. Unmittelbar nach dem Reset wird zunächst der Flash-Lader (*Bootloader*) aktiviert, dessen Aufbau und Funktionsweise von AUTOSAR derzeit nicht spezifiziert wird. In der Regel wird dort geprüft, ob sich ein gültiges Programm im Flash-ROM befindet und dieses dann

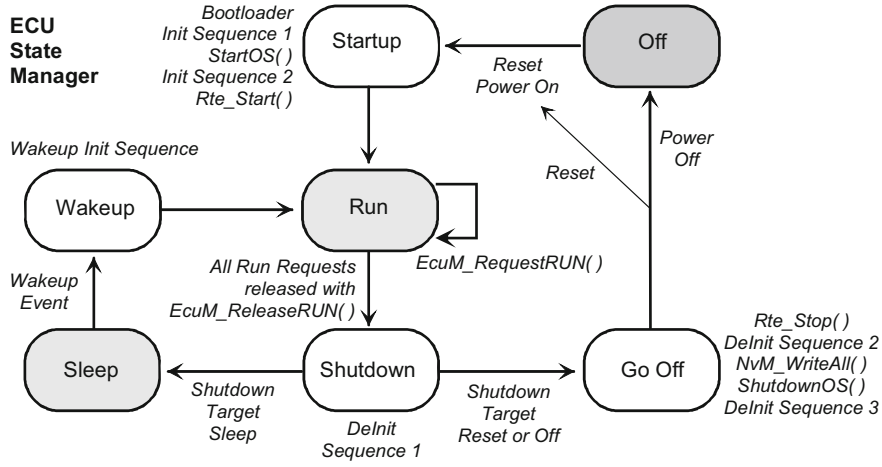

Abb. 8.6 Betriebszustände des Steuergerätes (vereinfacht)

gestartet. Danach beginnt im Zustand *Startup* die Grundinitialisierung des Mikrocontrollers mit der Funktion `EcuM_Init()` und das Betriebssystem (AUTOSAR OS) wird über den bereits von OSEK OS bekannten Aufruf `StartOS()` gestartet. Bereits unter Kontrolle des Betriebssystems erfolgt dann in dessen erster Task `EcuM_MainFunction()` die weitere Initialisierung des Steuergerätes. Am Ende dieser mehrstufigen Startphase sind die Basissoftware mit dem Betriebssystem, den Kommunikationsschnittstellen, den Hardwaretreibern sowie das *Runtime Environment RTE* vollständig funktionsfähig und das System geht in den Zustand *Run* über. Für die Initialisierung stellen die Treiber in der Regel eine Funktion `ModuleName_Init()` zur Verfügung, die vom EcuM aufgerufen wird. In welcher Reihenfolge dies für die einzelnen Module geschieht, muss bei der Konfiguration des Systems festgelegt werden.

Das Steuergerät verbleibt im Zustand *Run*, wenn mindestens eine Anwendungskomponente den Zustand mit `EcuM_RequestRUN()` dauerhaft anfordert. Der Übergang in den Zwischenzustand *Shutdown* erfolgt, wenn alle Komponenten, die den Zustand *Run* angefordert haben, ihn durch `EcuM_ReleaseRUN()` wieder freigegeben. Ob das Steuergerät von dort aus vollständig abgeschaltet wird (Zustand *Off*), ein Software-Reset ausgelöst oder in den Stromsparmodus (Zustand *Sleep*) umgeschaltet wird, hängt davon ab, welcher Zielzustand von der Anwendung mit Hilfe der Funktion `EcuM_SelectShutdownTarget()` eingestellt wurde. Welche Softwarekomponenten die genannten Funktionen überhaupt verwenden und den Zustand des Steuergerätes beeinflussen dürfen, wird bei der Konfiguration des Systems festgelegt und zur Laufzeit überprüft.

Beim Abschalten werden in mehreren Stufen Softwaremodule und Hardwaretreiber deinitialisiert, das *Runtime Environment RTE* angehalten, Fehlerspeicher- und andere persistente Daten in den nicht-flüchtigen Speicher geschrieben, das Betriebssystem gestoppt und schließlich das Steuergerät abgeschaltet bzw. über einen Reset ein Neustart ausgelöst. Beim

Übergang in den Zustand *Sleep* werden diejenigen Hardwarekomponenten, die das Gerät wieder aufwecken sollen, entsprechend initialisiert. In der Regel lösen diese Komponenten, z. B. Bedienschalter, Zeitgeber oder Kommunikationscontroller, die eine Datenbotschaft empfangen, einen Hardwareinterrupt aus, in diesem Zusammenhang als *Wakeup Event* bezeichnet. Welche Baugruppen des Gerätes bzw. Mikrocontrollers durch Abschalten der Takt- oder Spannungsversorgung in den Stromsparmodus geschickt werden, ist gerätespezifisch.

Die vorliegende Darstellung ist stark vereinfacht. Alle beschriebenen Zustände enthalten mehrere Unterzustände. Beim Übergang zwischen den Zuständen werden praktisch stets sogenannte Callout- und Callback-Funktionen aufgerufen. Callout-Funktionen sind Funktionen, in denen die Entwickler steuergerätespezifische Funktionen implementieren können. Über Callback-Funktionen (*Notifications*) werden die anderen Softwaremodule über die Zustandsänderungen informiert.

Mit AUTOSAR 4.0 wurde alternativ zum oben beschriebenen *ECU State Manager* mit seinen fest vordefinierten Zuständen und Zustandsübergängen (*Fixed State Machine*) die Möglichkeit eingeführt, die Zustände und Übergänge frei zu wählen. Dieser neue *ECU State Manager* behandelt nur noch die frühe *Startup* und die späte *Shutdown* Phase nach dem von AUTOSAR vordefinierten Konzept und übergibt die weitere Behandlung, insbesondere den Ablauf der *Run* bzw. *Sleep*-Zustände an andere *Manager*, die vom Hersteller frei definiert werden dürfen.

Der *Diagnostic Event Manager* (DEM) verwaltet den zentralen Fehlerspeicher des Systems, dessen Daten über den *NVRAM Manager* im nicht-flüchtigen Speicher gespeichert werden (Abb. 8.7). Die eigentlichen Überwachungsfunktionen, mit denen die Fehler erkannt werden, befinden sich aber nicht im DEM, sondern in den Komponenten der Anwendungssoftware bzw. in den verschiedenen Modulen der Basissoftware. Sobald diese Module einen Fehler feststellen, melden sie ein sogenanntes Fehlerereignis (*Error Event)* an den DEM, wobei in der Regel zu jedem Fehler auch eine Reihe von Statusinformationen sowie Umgebungsbedingungen (*Freeze Frame*) mitgeteilt werden und der übliche Mechanismus mit Fehlerentprellung und Fehlerheilung verwendet wird, wie er im Abschn. 5.3.4 beschrieben wurde. Über den DEM greift auch der *Diagnostic Communication Manager* (DCM), der die Diagnosedienste der Diagnoseprotokolle UDS und OBD ausführt, auf die Daten im Fehlerspeicher zu. Bei jedem Fehlerereignis kann der DEM zusätzlich auch noch den *Function Inhibition Manager* (FIM) informieren, der an zentraler Stelle Informationen für das Freigeben oder Sperren von Funktionsgruppen verwaltet. Solche Funktionsgruppen könnten z. B. alle an der Abgasrückführung oder an der Steuerung der Zündung beteiligten Softwarekomponenten sein. Bei der Konfiguration des Systems kann festgelegt werden, bei welchen Fehlern oder bei welcher Kombination von Fehlern eine solche Funktionsgruppe nicht mehr ausgeführt werden soll. Der FIM selbst verwaltet dabei nur die Bedingungen und verknüpft diese zu einer Gesamtinformation. Die eigentliche Entscheidung, ob die Funktion weiterhin ausgeführt wird oder nicht, treffen die Anwendungssoftwarekomponenten selbst, indem sie den Status beim FIM regelmäßig abfragen. Der FIM ist auch als Schnittstelle für die Fahrzeugapplikation vorgesehen, mit der zen-

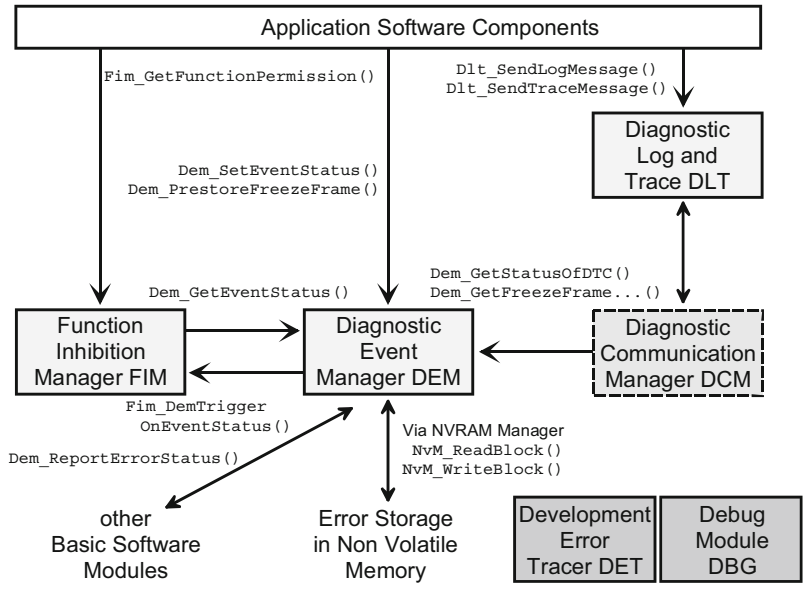

Abb. 8.7 Zentraler Fehlerspeicher und Funktionsauswahl

tral Funktionsgruppen, die z. B. in einer bestimmten Ausstattungsvariante eines Fahrzeugs nicht benötigt werden, abgeschaltet werden können.

Zusätzlich zur Speicherung von Fehlern im normalen Fahrzeugbetrieb können alle Softwaremodule so konfiguriert werden, dass sie in der Entwicklungsphase eine wesentlich intensivere Fehlererkennung durchführen, z. B. die generelle Überprüfung der Aufrufparameter von API-Funktionen. Die dabei erkannten Fehler werden an den *Development Error Tracer* (DET) gemeldet. Die Information enthält eine Angabe über das Modul, in dem der Fehler auftrat, die betroffene API-Funktion sowie einen Fehlercode. Wie diese Information im DET gespeichert oder weiterverarbeitet werden soll, ist dagegen nicht spezifiziert. Gedacht ist dies als Schnittstelle zu externen Entwicklungs- und Testwerkzeugen wie Fehlerloggern oder Debuggern.

Mit AUTOSAR 4.0 wurden die Fehlererkennungsmöglichkeiten nochmals erweitert. Das *Diagnostic Log and Trace* Modul (DLT) erlaubt es, Anwendungssoftwarekomponenten neben den standardisierten Diagnosefehlern weitere Fehlermeldungen abzusetzen. Daneben gibt es ein *Debug* Modul (DBG), das einem externen Debugger Zugriff auf alle Steuergeräte-internen Variablen und Funktionsaufrufe geben soll. Beide Module kommunizieren nach außen entweder über eine der standardisierten Kommunikationsschnittstellen oder über eine herstellerspezifische Debug-Schnittstelle. Während der Debugger ausschließlich für die Entwicklungsphase vorgesehen ist und daher keine Zugriffsschutzmechanismen enthält, kann das DLT-Modul auch in Seriengeräten verwendet werden. Dazu wird es über einen UDS-Diagnosedienst aktiviert und das Gerät in eine spezielle Diagnose-

sitzung umgeschaltet. Die Verteilung der Fehlerfunktionen auf DET, DLT und DBG sowie das Applikationsprotokoll XCP, das ebenfalls zu diesem Umfeld gehört, macht die Behandlung leider etwas unübersichtlich.

Die Überwachung des zeitlichen Ablaufs der einzelnen Anwendungssoftwarekomponenten erfolgt durch den *Watchdog Manager*. Dieser wird von jeder Softwarekomponente über einen *RTE Port* (siehe Abschn. 8.6) periodisch getriggert. Wenn dies von allen Softwarekomponenten regelmäßig ausgeführt wird, triggert der *Watchdog Manager* den Hardware-Watchdog über den *Watchdog Interface* Treiber. Bleibt der Trigger dagegen längere Zeit und/oder wiederholt aus, wechselt der *Watchdog Manager* in einen *Recovery*-Zustand, in dem die Komponente versuchen kann, das Problem zu beseitigen. Wenn dies innerhalb einer bestimmten Zeit nicht funktioniert, teilt der *Watchdog Manager* den noch funktionierenden Anwendungen mit, dass demnächst ein Steuergeräte-Reset notwendig sein wird und löst diesen nach einer weiteren Wartezeit aus, indem er den Hardware-Watchdog nicht mehr triggert. Die Überwachungs- bzw. Wartezeiten sind für jede einzelne zu überwachende Softwarekomponente individuell und unabhängig von der Periode des Hardware-Watchdogs konfigurierbar.

8.2.1 Funktionale Sicherheit

Die Ablaufüberwachung (*Program Flow Monitoring*) durch den *Watchdog Manager* ist Teil des Konzepts zur Gewährleistung der funktionalen Sicherheit nach ISO 26262. Als weitere Elemente kann das Betriebssystem die Ausführungsdauer und -häufigkeit von Programmen, die Stackauslastung sowie Speicherzugriffe überwachen (siehe Abschnitt 8.3). Hardwarekomponenten werden durch geeignete Routinen wie Speichertests geprüft. Auf Anwendungsebene greifen die üblichen Überprüfungsmechanismen für Sensor- und Aktorsignale wie Signalbereichsüberwachung (*Signal Range Check*) oder Plausibilitätsprüfungen. Die Datenübertragung kann seit AUTOSAR 4.0 durch die *End-to-End-Communication Protection* (E2E) zusätzlich abgesichert werden. Diese E2E-Bibliothek kann sowohl auf der PDU-Ebene des Kommunikationsstacks (siehe Abschn. 8.4) als auch auf der RTE-Ebene (siehe Abschn. 8.6) eingesetzt werden, um die Buskommunikation zwischen verschiedenen Steuergeräten oder die Kommunikation zwischen mehreren CPU-Kernen innerhalb eines Geräts zu sichern. Zusätzlich zu den eigentlichen Nutzdaten werden dabei ein Sequenzzählerwert (4 bit), eine Prüfsumme (8 bit), die aus den Datenwerten und einer Daten- bzw. Botschaftskennung ermittelt wird, und/oder die invertierten Nutzdaten übertragen. Die Prüfwerte werden auf der Senderseite durch die API-Funktion `E2E_P0xProtect()` erzeugt und auf der Empfängerseite durch `E2E_P0xCheck()` überprüft. Optional kann dabei auch eine Zeitüberwachung der Übertragung stattfinden.

8.3 Betriebssystem AUTOSAR OS

AUTOSAR OS ist aufwärts kompatibel zu OSEK OS, wie es in Abschn. 7.2.1 beschrieben wurde. Es verwendet dieselben API-Aufrufe und verweist zur Semantik der Betriebssystemaufrufe häufig auf die Spezifikation des OSEK-Konsortiums bzw. die zugehörige ISO 17356-3 Norm.

In vier verschiedenen Ausbaustufen wird die Grundfunktionalität von OSEK OS um Konzepte aus OSEK Time und Protected OSEK (siehe Abschn. 7.2.4) erweitert. Die Erweiterungen sind allerdings funktional nur ähnlich und verwenden andere API-Funktionen. Dies dürfte aber kaum nachteilig sein, da OSEK Time und Protected OSEK bisher ohnehin selten oder gar nicht eingesetzt wurden. Die verschiedenen Ausbaustufen, bei AUTOSAR *Scalability Classes* genannt, unterscheiden sich im Wesentlichen im Umfang der gegenüber OSEK OS neu hinzugekommenen Speicherschutzfunktionen bzw. Zeitüberwachungen (Tab. 8.2). Diese Ausbaustufen sollten nicht mit den OSEK OS Konformitätsklassen (siehe Tab. 7.1) verwechselt werden, die es theoretisch auch weiterhin gibt, die bei der Konfiguration des Betriebssystems bei AUTOSAR aber praktisch keine Rolle mehr spielen.

OSEK OS verwendet ein rein ereignisgesteuertes, auf Prioritäten basierendes Multitasking-Konzept. Zeitgesteuerte Abläufe müssen mit Hilfe von Alarmen nachgebildet werden, was bei komplexeren Abläufen schnell unübersichtlich werden kann. AUTOSAR OS führt daher zusätzlich vordefinierte zeitliche Taskabläufe mit Hilfe sogenannter *Schedule Tabellen* ein, die mit Hilfe von Zählern abgearbeitet werden. Der Zähler (*OSEK Counter*) ist in der Regel mit einem Hardwarezeitgeber gekoppelt, kann aber auch durch einen Softwarezähler gebildet werden, der mit `IncrementCounter()` von einer Task oder Interrupt Service Routine inkrementiert wird. Bei der Konfiguration des Betriebssystems wird festgelegt, bei welchem Zählerstand (*Expiry Point*) welche Task aktiviert oder welches *OS Event* gesetzt werden soll. Der Ablauf beginnt entweder automatisch oder wird durch Aufruf der API-Funktion `StartScheduleTable...()` aktiviert bzw. durch `StopScheduleTable()` gestoppt (Abb. 8.8). Bei der Aktivierung kann der eigentliche Startzeitpunkt durch einen Offset verzögert und eine Wiederholperiode angegeben werden. Statt der periodischen Wiederholung des Ablaufs ist auch ein einmaliges Durchlaufen

Tab. 8.2 AUTOSAR OS Ausbaustufen (*Scalability Classes*)

Ausbaustufe	Class 1	Class 2	Class 3	Class 4
OSEK OS kompatible API mit den Erweiterungen …	… zeitgesteuerte Tasks (Schedule Table, Counter Interface) und Stack-Überwachung			
OSEK OS Alarm Callback	Ja	Nein, da nicht kompatibel mit Speicherzugriffsschutz und Zeitüberwachung		
Zugriffsschutzmechanismen	Nein		Ja	
Zeitüberwachung und Zeitsynchronisation	Nein	Ja	Nein	Ja

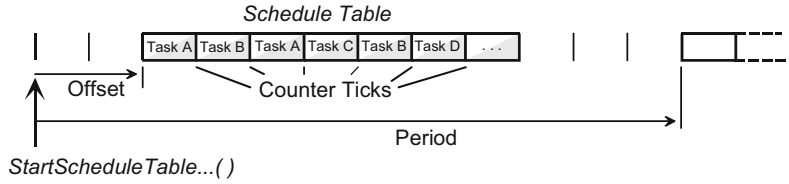

Abb. 8.8 Schedule Tabellen

einer *Schedule Tabelle* möglich. Durch `NextScheduleTable()` kann auf eine andere Tabelle umgeschaltet werden. Wenn mehrere Zähler verwendet werden, können auch mehrere *Schedule Tabellen* gleichzeitig aktiv sein. Eine mögliche Anwendung der *Schedule Tabellen* ist die Abarbeitung von Funktionen synchron zum Kommunikationsablauf eines FlexRay-Bussystems. Dazu koppelt man den Zähler für die *Schedule Tabelle* mit dem *Cycle Counter* oder *Macrotick Counter* des Bussystems (siehe Abschn. 3.3).

Die Grundidee der *Schedule Tabellen* entspricht dem zeitgesteuerten Scheduling von OSEK Time und dessen *Dispatch Tabelle* (siehe Abschn. 7.2.4). Im Gegensatz dazu konkurrieren bei AUTOSAR OS diejenigen Tasks, die durch die *Schedule Tabellen* aktiviert werden, allerdings mit gewöhnlichen OSEK OS Tasks um die Ausführung. Das heißt, die Tasks werden zwar zum vordefinierten Zeitpunkt (Zählerstand) in den Zustand *Ready* (siehe Abb. 7.3) versetzt, wann sie dann aber tatsächlich ablaufen dürfen, hängt immer noch von ihrer Priorität relativ zu den anderen Tasks ab. Um einen streng deterministischen Ablauf zu gewährleisten, ist also zusätzlich noch die Festlegung geeignet hoher Prioritäten und eine sorgfältige statische Ablaufanalyse in der Entwicklungsphase notwendig. Im Gegensatz zu OSEK OS kann die Einhaltung der Zeitbedingungen aber durch die ebenfalls neue Zeitüberwachung zur Laufzeit überprüft und der zeitliche Ablauf mit einer globalen Zeitbasis synchronisiert werden. Mit AUTOSAR 4.0 steht dazu ergänzend der *Synchronized Time Base Manager* zur Verfügung. Dieser generiert auf Basis von FlexRay- oder TTCAN-Buszyklen eine relative und/oder absolute Zeitbasis für alle Steuergeräte, die an diesem Bussystem angeschlossen sind.

Abhängig von der Ausbaustufe verfügt AUTOSAR OS über verschiedene Überwachungsmechanismen, um die Zuverlässigkeit zu erhöhen. Dabei wurden die bereits bei *Protected OS* vorgeschlagenen Konzepte (siehe Abschn. 7.2.4) weitgehend übernommen. Im einfachsten Fall erfolgt bei jeder Taskumschaltung lediglich eine Überprüfung, ob eine Task ihren jeweils maximal zulässigen Stackbereich nicht überschritten hat (*Stack Monitoring*). Bei einem Stackfehler wird das System heruntergefahren.

Optional kann für jede Task bzw. Interrupt Service Routine einzeln konfigurierbar eine Zeitüberwachung stattfinden. Damit wird die Ausführungszeit (*Excecution Time Budget*) vom Start bis zum Ende einer Task vom Betriebssystem überwacht. Mit einer weiteren Überwachung kann eine zu hohe Aufrufhäufigkeit bzw. Wiederholrate (*Inter-Arrival Rate*) z. B. bei Interrupts erkannt werden. Schließlich kann auch noch die maximale Zeit überwacht werden, während der eine Task eine Ressource blockiert oder einen Interrupt sperrt

(*Locking Time*). Im Fehlerfall wird eine als *Protection Hook* bezeichnete Funktion aufgerufen, die bei der Konfiguration des Betriebssystems bereitzustellen ist. In dieser Funktion kann entschieden werden, ob die fehlerhafte Task gestoppt, neu gestartet oder das ganze System heruntergefahren werden soll.

Ebenso von Protected OS übernommen wurde das Konzept der Anwendungsgruppe (*Application*). In einer Anwendungsgruppe werden Betriebssystemobjekte, d. h. Tasks und Ressourcen wie Events oder Alarme, zusammengefasst, die zur Erfüllung einer bestimmten Systemfunktion eng zusammenwirken müssen. Innerhalb einer Anwendungsgruppe können diese Objekte sich gegenseitig beliebig aufrufen oder verwenden, dürfen aber selbst keine Objekte aus einer anderen Anwendungsgruppe benutzen oder von fremden Gruppen aus verwendet werden. Ein großer Teil der dazu notwendigen Überprüfungen kann bereits bei der Systemgenerierung, d. h. in der Konfigurations- und Übersetzungsphase, stattfinden. Zusätzlich kann aber auch beim Aufruf von Betriebssystemfunktionen oder von einer Anwendertask zur Laufzeit eine entsprechende Überwachung durchgeführt werden (*Service Protection*). Dazu stehen die API-Funktionen `CheckObject...()` und `Check...MemoryAccess()` zur Verfügung. Im Fehlerfall wird wiederum die *Protection Hook* Funktion aufgerufen. Da die Überwachungsmechanismen den Mikrocontroller zur Laufzeit belasten, können sie für einzelne Anwendungsgruppen, die sogenannten *Trusted Applications*, bei der Systemkonfiguration abgeschaltet werden. Außerdem gibt es mit der API-Funktion `CallTrustedFunction()` die Möglichkeit, aus einer *Non-Trusted Application* heraus eine Funktion in einer *Trusted Application* aufzurufen. Neben den zugehörigen Tasks wird auch der gesamten Anwendungsgruppe ein Zustand (*Accessible*, *Restarting*, *Terminated*) zugeordnet. Wenn eine Anwendungsgruppe mit `TerminateApplication()` gestoppt und gegebenenfalls neu gestartet wird, beendet das Betriebssystem alle zugehörigen Tasks und gibt alle Ressourcen frei.

Zukünftig werden verstärkt Mikrocontroller eingesetzt werden, die eingebaute Hardwareschutzmechanismen mitbringen. Diese Prozessoren verfügen über einen privilegierten und einen nicht-privilegierten Betriebsmodus (*Kernel Mode* und *User Mode*). Während das Betriebssystem den privilegierten Modus verwendet, arbeiten Anwendungen gewöhnlich im nicht-privilegierten Modus und können damit auf bestimmte Steuerregister des Prozessors, z. B. auf dessen Zeitgeber- und Interruptsteuerung, nicht zugreifen. In ähnlicher Weise überwachen derartige Prozessoren auch Speicherzugriffe (*Memory Access Protection*), so dass eine Anwendungsgruppe nur auf den ihr zugeordneten Speicherbereich zugreifen kann und vor dem Zugriff durch andere Anwendungsgruppen geschützt ist. Solche in Hardware implementierten Mechanismen sind erheblich leistungsfähiger als rein softwarebasierte Überwachungen. Bei der Softwarekonfiguration und im *Run Time Environment RTE* aber wird die *Memory Access Protection* auch in AUTOSAR 4.0 nur rudimentär unterstützt.

Die Konfiguration von AUTOSAR OS erfolgte bei AUTOSAR 2.0 noch mit Hilfe der bekannten OIL-Konfigurationsdateien (siehe Tab. 7.3), wobei für die neu hinzugekommen Funktionen eine als OIL 3.x bezeichnete Syntaxerweiterung vorgeschlagen wurde. Seit AUTOSAR 3.0 ist nun auch beim Betriebssystem die XML-basierte Konfiguration

verbindlich. Damit existierende OSEK OS Systeme einfach nach AUTOSAR migriert werden können, erwartet man, dass die Hersteller der Konfigurationswerkzeuge Import-/Exportschnittstellen für OIL vorsehen.

Die AUTOSAR OS Spezifikation weist darauf hin, dass es für bestimmte Aufgaben, beispielsweise bei Navigationssystemen oder anderen Infotainment-Geräten mit grafischen Benutzeroberflächen, auch weiterhin notwendig sein kann, andere Betriebssysteme, z. B. Embedded Linux oder Windows CE, einzusetzen. Für solche Systeme wird vorgeschlagen, einen *AUTOSAR OS Abstraction Layer OSAL* einzuführen, damit AUTOSAR Funktionen, wie z. B. AUTOSAR-kompatible Kommunikationsstacks oder Diagnosefunktionen, einfacher auf diese Systeme portierbar werden.

Mit AUTOSAR 4.0 wird eine Unterstützung für mehrere CPU-Kerne in einem Steuergerät eingeführt (*MultiCore OS*). Das Konzept sieht vor, dass auf jedem CPU-Kern eine eigene Instanz des AUTOSAR OS Betriebssystems läuft. Tasks und Ressourcen werden den einzelnen CPU-Gruppen in Form von Anwendungsgruppen während der Entwicklungs- und Konfigurationsphase fest zugeordnet, d. h. das System ist statisch konfiguriert, eine dynamische Lastverteilung zwischen den CPU-Kernen ist nicht vorgesehen. Die CPU-Kerne müssen sich beim Systemhochlauf synchronisieren und das Task-Scheduling gleichzeitig starten, anschließend erfolgt das Scheduling aber für jeden CPU-Kern unabhängig. Die OSEK bzw. AUTOSAR OS Programmierschnittstelle kann unverändert bleiben, da Tasks und Ressourcen über Kennziffern identifiziert werden, die für das Gesamtsystem über alle Kerne hinweg eindeutig sein müssen. Funktionen wie `ActivateTask()` oder `SetEvent()` können damit auch Tasks, Alarme und Events in anderen Kernen steuern. Ressourcen dagegen sind nur lokal verwendbar und können nicht zur Synchronisation zwischen den verschiedenen Kernen eingesetzt werden. Als Synchronisationsmechanismus zwischen den verschiedenen Kernen wurden sogenannte *Spin Locks* eingeführt. *Spin Locks* werden mit `GetSpinLock()` angefordert und mit `ReleaseSpinLock()` wieder freigegeben. Falls der *Spin Lock* bei der Anforderung durch einen anderen CPU-Kern belegt ist, fragt die anfordernde Task den *Spin Lock* solange ab, bis er wieder frei ist. Falls dies nicht zulässig ist, kann mit `TryToGetSpinLock()` auch ein nicht-blockierender Reservierungsversuch veranlasst werden.

Der Datenaustausch zwischen zwei CPU-Kernen erfolgt über einen gemeinsamen Speicherbereich (*Shared Memory*) mit Hilfe der *Inter-OS-Application Communication IOC* Funktionen. Diese Funktionen implementieren eine gepufferte oder ungepufferte *Sender-Receiver*-Kommunikation, wie sie in Abschn. 8.6 beschrieben wird. Aus Sicht der Softwarekomponenten der Anwendungsebene ist der Datenaustausch damit unabhängig davon, ob die Tasks auf demselben oder auf verschiedenen CPU-Kernen erfolgt.

Basic Software (BSW) Scheduler Naturgemäß verwenden nicht nur die Softwarekomponenten der Anwendungsebene sondern auch die Module der Basissoftware die Mechanismen des Betriebssystems. In beiden Fällen versucht die AUTOSAR Architektur jedoch, die direkte Benutzung von Betriebssystem-API-Funktionen zu vermeiden, sondern eine Zwischenschicht einzuschalten. Für die Softwarekomponenten der Anwendung ist dies das

Runtime Environment RTE, das in Abschn. 8.6 beschrieben wird. Innerhalb der Basissoftware dient der *BSW Scheduler* als virtuelle Zwischenebene. Seit AUTOSAR 4.0 wird er nicht mehr separat beschrieben, sondern konsequenterweise als Teil der RTE-Spezifikation betrachtet.

Alle Basissoftwaremodule bestehen aus Funktionen, die einmalig oder periodisch aufgerufen werden müssen, aber nach jedem Aufruf unmittelbar wieder enden, ohne interne Wartezustände zu kennen. In der Regel hat jedes Modul eine Initalisierungs-, eine Deinitialisierungs- und eine Hauptfunktion, in der die normale Funktionalität realisiert wird, sowie diverse Callback- und Notification-Funktionen.

Programmtechnisch gesehen implementiert der *BSW-Scheduler* lediglich einige C-Funktionen, in denen die Aufrufe dieser Basissoftwarefunktionen in der richtigen Reihenfolge enthalten und entsprechend ihrer Aufrufhäufigkeit gruppiert sind. Diese C-Funktionen ihrerseits werden vom eigentlichen Betriebssystem AUTOSAR OS als Tasks aufgerufen. Trotz seines Namens überlässt der *BSW Scheduler* das Scheduling dem eigentlichen Betriebssystem. Die C-Funktionen sollen weitgehend automatisch erzeugt werden, wenn der Steuergeräte-Integrator die Basissoftware konfiguriert. Der Sinn dieser Zwischenebene wird deutlich, wenn man bedenkt, dass auch die Basissoftware nach Vorstellung der AUTOSAR-Protagonisten nicht als vollständiges Softwarepaket bereitgestellt, sondern wie die Fahrsoftware aus Komponenten verschiedener Lieferanten zusammengesetzt wird. Mit der Konfiguration des BSW-Schedulers und der Zusammenfassung von Aufrufen in einigen C-Funktionen legt der für diese Integration Verantwortliche die Ablaufhäufigkeit und Ablaufreihenfolge dieser einzelnen Module fest, ohne dass für jedes Modul eine eigene Task und komplexe Synchronisations-Ressourcen verwendet werden müssen.

8.4 Kommunikationsstack AUTOSAR COM, Diagnose DCM

Abbildung 8.9 zeigt den relativ komplexen Aufbau der Kommunikationsdienste. AUTOSAR unterstützt die Bussysteme CAN, TTCAN, FlexRay, LIN und Ethernet. Die Ankopplung an MOST ist noch nicht ausgearbeitet. Soweit abzusehen, wird der MOST-Protokollstapel aber einen anderen Aufbau aufweisen, da sich die komplexe dienstorientierte Anwendungsschnittstelle der *MOST Network Services* (siehe Abschn. 3.4) nicht direkt auf die relativ einfachen, botschaftsorientierten Schnittstellen abbilden lässt, die von CAN, LIN oder FlexRay verwendet werden.

Gegenüber den Anwendungssoftwarekomponenten wird eine einheitliche, vom spezifischen Bussystem unabhängige Schnittstelle angeboten, die im Wesentlichen aus den Modulen *Diagnostic Communication Manager DCM* und *AUTOSAR COM* besteht. Das DCM Modul wickelt die Off-Board-Kommunikation für die UDS- und OBD-Diagnosedienste nach ISO 14229 und ISO 15031 (Abschn. 5.2 und 5.3) ab. Das ältere KWP 2000-Protokoll wird nicht direkt unterstützt. Seit Version 4 erlaubt AUTOSAR auch den Einsatz des im Nutzfahrzeugbereich weitverbreiteten SAE J1939 Protokolls (Abschn. 4.5) für CAN. Zu dessen Unterstützung sind parallel zum Standard DCM ein spezieller *J1939 DCM*, ein

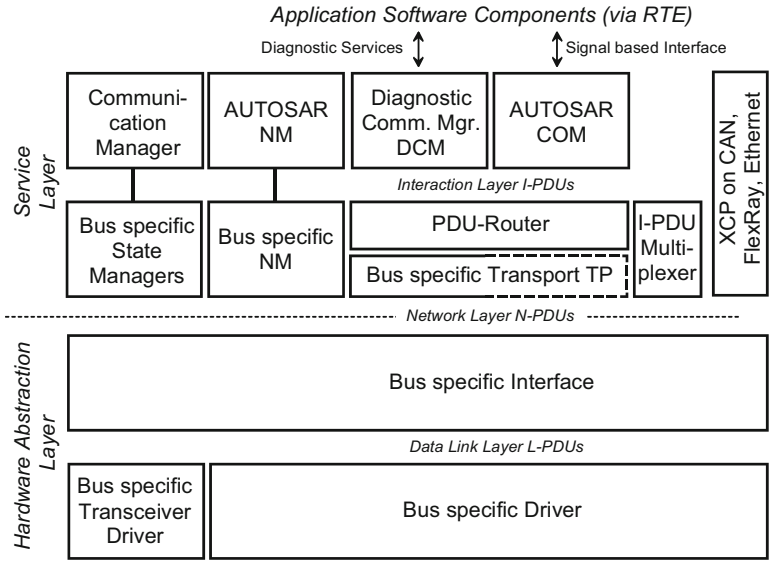

Abb. 8.9 Kommunikationsstack mit jeweils busspezifischen Komponenten für CAN, FlexRay, LIN und Ethernet

J1939 Request Manager sowie ein spezielles J1939 *Network Management* vorgesehen. Das Netzmanagement beschränkt sich dabei auf die bei SAE J1939 mögliche dynamische Zuweisung von Adressen.

Das Modul AUTOSAR COM ist für die On-Board-Kommunikation zuständig und stellt die AUTOSAR-Variante von OSEK COM (Abschn. 7.2.2) dar. Die darunterliegende *Protocol Data Unit (PDU) Router* und *IPDU-Multiplexer* Schicht verteilt die Botschaften auf die jeweiligen Bussysteme und kann als Gateway auch ankommende Botschaften von einem Bussystem direkt auf ein anderes weiterleiten, solange keine Zwischenspeicherung der Botschaften notwendig ist. Unter der Routing-Schicht liegen die busspezifischen Transportprotokolle TP und die Hardwaretreiber für die jeweiligen Kommunikationscontroller.

Langfristig soll für alle Bussysteme sowohl für die Off-Board- als auch für die On-Board-Kommunikation ein Transportprotokoll verwendet werden. Vorläufig wird das Transportprotokoll aber nur für die Diagnosekommunikation eingesetzt. Auf der CAN-Seite basiert es auf ISO 15765-2 (Abschn. 4.1) oder alternativ auf SAE J1939/21 (Abschn. 4.5). Für FlexRay wird das in ISO 10681-2 (Abschn. 4.2) beschriebene Transportprotokoll eingesetzt. In AUTOSAR-Version 3 wurde noch eine abweichende Vorgängervariante des aktuellen FlexRay-Transportprotokolls eingesetzt (siehe Abschn. 4.2 in der 3. Auflage dieses Buches), die bis auf weiteres alternativ unterstützt wird. Für LIN ist das Transportprotokoll kein separates Modul, sondern als Teil des LIN-Protokolls (Abschn. 3.2) in das *LIN Interface* integriert. Ethernet verwendet DoIP nach ISO 13400-2 (Abschn. 4.6).

Für die Applikation werden *XCP on CAN*, *FlexRay* oder *Ethernet* (Abschn. 6.2) als Protokolle genannt. AUTOSAR bezieht sich auf die entsprechenden ASAM-Normen und definiert die notwendigen Schnittstellen zum RTE und zum busspezifischen Interface. FIBEX, das offene ASAM-Beschreibungsformat für die On-Board-Kommunikation, und ODX, das ASAM-Format für Diagnosedaten, sind noch nicht nahtlos in AUTOSAR integriert, da die AUTOSAR-Konfiguration mehr Informationen benötigt als in diesen Formaten vorhanden sind. Die AUTOSAR-Werkzeuge müssen daher entsprechende Export- und Importschnittstellen bereitstellen.

Der *Communication Manager* (ComM) initialisiert den gesamten Kommunikationsstack und verwaltet im Zusammenspiel mit den busspezifischen *State Managern* und dem Netzmanagement, das im folgenden Kapitel detaillierter beschrieben wird, den Zustand der einzelnen Kommunikationskanäle (Abb. 8.10). Er kann funktional als die für die Kommunikation zuständige Erweiterung des *ECU State Managers* verstanden werden, wobei damit nicht nur auf der Ebene des gesamten Gerätes sondern auch für jedes einzelne Bussystem *Sleep-Wake Up*-Szenarien möglich werden. Die Zustände werden für jedes Bussystem separat verwaltet. Die im Abb. 8.10 dargestellten Zustände haben mehrere Unterzustände. Bei FlexRay beispielsweise werden praktisch alle *Protocol Operation Control* Zustände aus Abb. 3.29 abgebildet. Bei CAN gibt es Unterzustände, mit denen das Auftreten von *Error active/passive* und *Bus Off* Fehlern (vgl. Abschn. 3.1.4) erfasst wird. Die Hauptzustände sind *No Communication (Offline)*, in dem die Busschnittstelle abgeschaltet ist, und *Full Communication (Online)*, in dem sowohl gesendet als auch empfangen wird. Zu Testzwecken ist es außerdem möglich, den Zustand *Silent Communication* einzustellen, in dem zwar empfangen, aber nicht gesendet werden kann. Die Umschaltung zwischen den Zuständen erfolgt auf Anforderung der Anwendungssoftware bzw. des *Diagnostic Communication Managers* und berücksichtigt die Rückmeldungen des Netzmanagements und der Hardwaretreiber über den Bus und aufgetretene Fehler. Umgekehrt informiert der *Communication Manager* andere Komponenten mit Hilfe von Callback-Funktionen über Zustandsänderungen. Wie beim *ECU State Manager* wird bei der Konfiguration festgelegt, welche Softwarekomponenten Zustandsänderungen anfordern dürfen. Liegen mehrere unterschiedliche Anforderungen vor, so wird der jeweils höherwertige Zustand eingestellt, z. B. Senden und Empfangen statt nur Empfangen.

Off-Board-Kommunikation: Diagnostic Communication Manager: Der *Diagnostic Communication Manager* (Abb. 8.11) behandelt ankommende OBD- und UDS-Diagnosebotschaften nach ISO 15031 bzw. ISO 14229. In der gegenwärtigen Spezifikation muss dabei jede Anfrage (*Request*) beantwortet werden (*Response*), bevor die nächste Anfrage verarbeitet werden kann. Außerdem gibt es noch eine Reihe von Einschränkungen von Diagnosediensten, die in AUTOSAR konformen Implementierungen bisher noch nicht unterstützt werden müssen.

Die Behandlung von Diagnosesitzungen (*Diagnostic Session Control*), Zugriffsschutz (*Security Access*) sowie die Sicherstellung des geforderten Zeitverhaltens einschließlich der *Tester Present*-Botschaften sowie die Formatierung der Antworten mit den erforderlichen

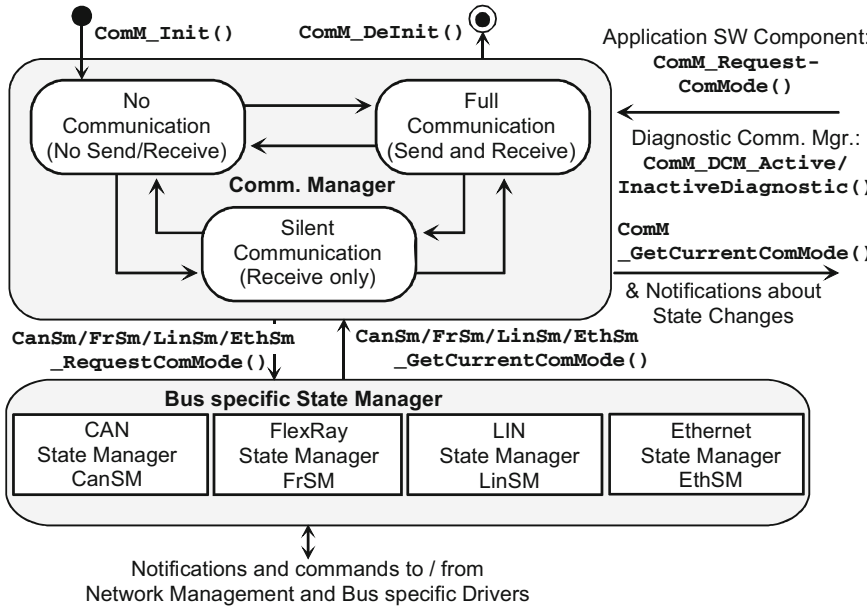

Abb. 8.10 Buszustände des *Communication Manager* (vereinfacht)

Headern (siehe Abb. 5.6 und 5.9) erfolgt selbstständig in den DCM Unterkomponenten DSL und DSD. Diagnosedienste, die sich auf den Fehlerspeicher beziehen (siehe Tab. 5.17 und 5.22), werden in der Unterkomponente DSP verarbeitet, die über den schon beschriebenen *Diagnostic Event Manager* (Abb. 8.7) auf den Fehlerspeicher zugreift. Kern der Ver-

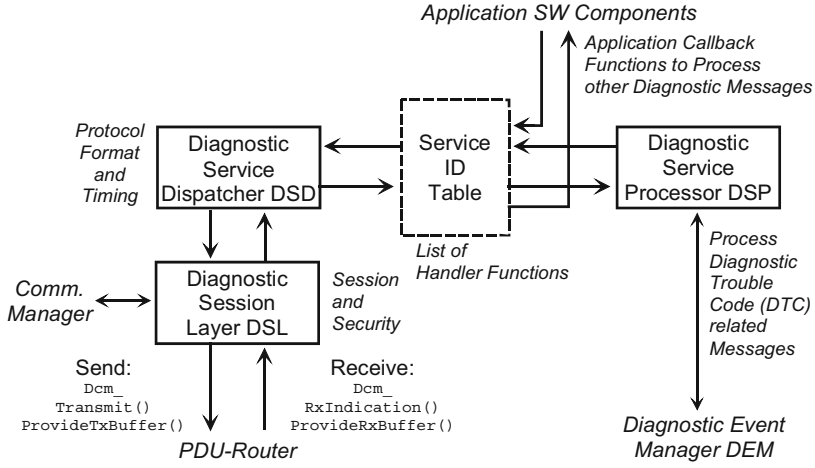

Abb. 8.11 Struktur des Diagnostic Communication Managers DCM

arbeitung der Diagnosedienste ist die *Service Identifier Tabelle*, in die bei der Konfiguration des Systems für jeden Dienst eine entsprechende Bearbeitungsfunktion eingetragen wird. Eine große Anzahl von UDS und OBD-Diagnosediensten, z. B. die OBD-Funktionen zum Lesen von Messwerten (Tab. 5.23) werden direkt im DSP implementiert, wobei die notwendigen Messdaten über RTE-Zugriffe geholt werden. Diagnosedienste, die nicht intern bearbeitet werden, können mittels Rückruffunktionen (*Callback*) an Softwarekomponenten der Anwendungsschicht delegiert werden. Für alle unbekannten Dienste sendet das Modul selbstständig eine *Service not Supported* Meldung (siehe Tab. 5.2). Der DCM kann mehrere *Service Identifier Tabellen* enthalten, zwischen denen zur Laufzeit umgeschaltet werden kann. Auf diese Weise können bei Bedarf mehrere unterschiedliche Diagnoseprotokolle unterstützt werden.

Mittlerweile bietet der DCM auch eine Schnittstelle, um einen HIS-kompatiblen Flash-Lader über UDS-Diagnosedienste zu aktivieren, wenn das Steuergeräteprogramm ausgetauscht werden soll (siehe Abschn. 9.4).

Wünschenswert wäre, bei der Konfiguration des *Diagnostic Communication Managers* eine Import-/Exportschnittstelle zu den im Diagnosebereich mittlerweile üblichen ODX Diagnosedatensätzen (Abschn. 6.6) zur Verfügung zu stellen. Dies bleibt allerdings dem Hersteller des Konfigurationswerkzeugs vorbehalten, die AUTOSAR-Spezifikationen machen dazu kaum Vorgaben.

On-Board-Kommunikation: AUTOSAR COM: Die Schnittstelle für die signalbasierte On-Board-Kommunikation (Abb. 8.9) beruht auf OSEK COM Version 3.0, wobei allerdings einige Mechanismen nicht berücksichtigt werden, weil sie durch andere AUTOSAR-Konzepte bereits abgedeckt sind. Zu den nicht vorhandenen Funktionen gehören die Botschaftswarteschlangen (*Queued Messages*), da die Zwischenspeicherung nötigenfalls bereits im *Runtime Environment RTE* erfolgt. Nicht unterstützt werden außerdem Botschaften veränderlicher Länge (*Dynamic and Zero Size Messages*) sowie die senderseitige Botschaftsfilterung (siehe Abschn. 7.2.2). Periodisch versendete Botschaften werden unterstützt, nicht aber die OSEK-Funktionen `StartPeriodic()` und `StopPeriodic()`.

Außerdem wird nicht OIL für die Konfiguration verwendet, da der gesamte interne und externe Datenaustausch bei AUTOSAR für alle Anwendungskomponenten transparent sein soll und einheitlich über das *Runtime Environment RTE* abgewickelt wird. Allerdings werden dieselben Konfigurationsparameter verwendet wie bei OIL, sie werden lediglich in das bei AUTOSAR übliche XML-Format abgebildet. Das Starten und Stoppen der Kommunikation erfolgt durch den *Communication Manager* und nicht über die bekannten OSEK COM-Funktionen `StartCOM()` und `StopCOM()`.

Gegenüber der Anwendungsschicht, d. h. dem RTE, wird eine signalorientierte Schnittstelle verwendet (Abb. 8.12). Welche Signale zu einer Botschaft (*Protocol Data Unit PDU*) zusammengefasst werden und unter welchen Bedingungen diese zu versenden sind, wird wie bei OSEK COM bei der Konfiguration festgelegt. Neben einfachen Signalen können Signale zu Gruppen zusammengefasst werden, so dass beim Senden bzw. Empfangen stets ein konsistenter Satz von Werten befördert wird. Die zugehörigen API-Funktionen sind

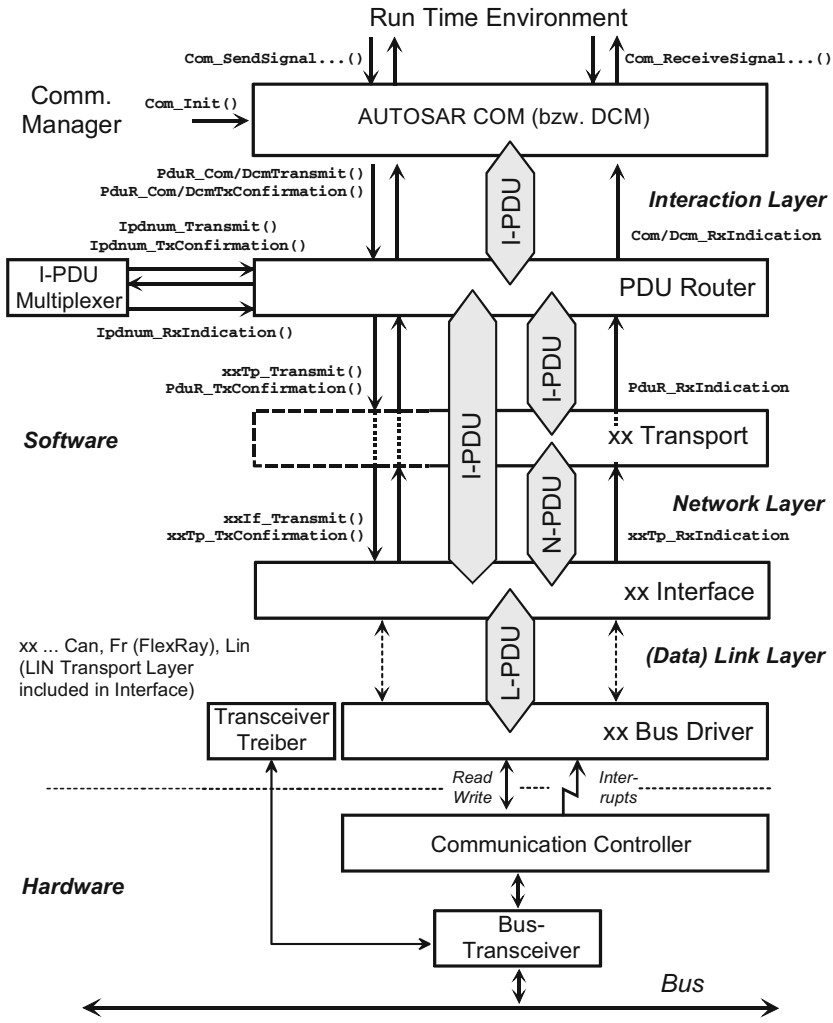

Abb. 8.12 Prinzip des AUTOSAR-Protokollstapels

u. a. `Com_SendSignal...()` und `Com_ReceiveSignal...()`. Daneben kann AUTOSAR COM intern ein Signal auf mehrere Botschaften verteilen.

Mechanismen der Protokollstapel, Transportprotokolle und PDU Router: Die signalorientierte COM-Schicht verwendet kein Transportprotokoll, so dass Botschaften, zumindest bei CAN und LIN, nicht mehr als 8 Datenbytes enthalten können. Bei FlexRay sind PDUs bis 254 Byte möglich. Für die Diagnosekommunikation des DCM dagegen wird bei CAN das bewährte, als ISO TP in ISO 15765-2 standardisierte Transportprotokoll unverändert weiterverwendet (Abschn. 4.1), alternativ vor allem für Nutzfahrzeuge kann das

Transportprotokoll nach SAE J1939/21 (Abschn. 4.5) eingesetzt werden. Für FlexRay wurde mit ISO 10681-2 ein Transportprotokoll definiert, das diese Konzepte übernimmt, nach mehreren Modifikationen aber nicht mehr direkt aufwärts kompatibel ist (Abschn. 4.2).

Der gesamte Protokollstapel (Abb. 8.12) verwendet unabhängig vom tatsächlichen Bussystem dieselbe Grundstruktur. Beim Senden wird die Botschaft (PDU) von der COM bzw. DCM Schicht über den *PDU Router* und gegebenenfalls die Transportschicht an das Interface des Bussystems durchgereicht. Dieses beauftragt über den entsprechenden Treiber den Kommunikationscontroller mit dem eigentlichen Versenden. Die Sendefunktion kehrt nach Erteilen des Sendeauftrags sofort zurück. Später, wenn die Botschaft dann tatsächlich erfolgreich versendet wurde, erhält der Auftraggeber durch eine Rückruffunktion eine Bestätigung (*TxConfirmation, Tx = Transmit*). Wird eine Botschaft empfangen, werden die oberen Schichten durch Rückruffunktionen (*RxIndication*, *Rx = Receive*) informiert und können die Botschaft lesen. Wann eine Botschaft versendet werden soll (*Direct Triggered*, *Periodic* usw., siehe Abschn. 7.2.2), ob Timeout-Überwachungen bzw. Empfangsfilter verwendet werden sollen und über welches der verschiedenen Bussysteme die Botschaft versendet bzw. empfangen werden soll, kann für jede PDU in der Entwicklungsphase festgelegt werden und wird in einer Tabellenstruktur gespeichert (*PDU Routing Table* u. a.). Weiterhin ist es möglich, eine PDU aus anderen PDUs zusammenzusetzen (*IPDU Multiplexing*). Innerhalb der Protokollsoftware werden die PDUs über Kennziffern (*PDU ID*) identifiziert. Die Weiterleitung der Botschaftsinhalte durch die einzelnen Schichten erfolgt in der Regel so, dass zeit- und speicherplatzraubende Kopiervorgänge möglichst vermieden werden. In der Art, wann und von wem die Daten kopiert werden und wer die notwendigen Puffer bereitstellen muss, unterscheiden sich die Protokollstapel für die einzelnen Bussysteme dann im Detail doch.

CAN-Protokollstapel: Das *CAN Interface* ist für alle CAN-Kanäle eines Steuergerätes verantwortlich (Abb. 8.13). Sind mehrere CAN-Controller eingebaut, können diese vom selben CAN-Treiber verwaltet werden, sofern sie untereinander kompatibel sind. Für CAN-Controller unterschiedlichen Typs sind dagegen verschiedene parallel arbeitende CAN-Treiber zuständig. Der CAN-Treiber spricht den CAN-Controller durch Schreib- und Lesebefehle über den Adress-/Datenbus oder den SPI-Bus des Mikrocontrollers an. Die Rückmeldungen des CAN-Controllers nach dem Absenden oder Empfangen von Daten erfolgen entweder durch zyklisches Abfragen durch den CAN-Treiber (Polling) oder optional im Interruptbetrieb. Um das Polling zu gewährleisten, müssen verschiedene interne Funktionen des CAN-Treibers vom Betriebssystem regelmäßig aufgerufen werden. Das über dem Treiber liegende *CAN Interface* verwendet grundsätzlich eine synchrone Aufrufschnittstelle. Beim Senden mit `CanIf_Transmit()` wird die Botschaft mit Hilfe der Funktion `Can_Write()` direkt in den Sendespeicher des CAN-Controllers kopiert und eine Sendeanforderung ausgelöst. Ist kein Sendespeicher frei, speichert das *CAN Interface* die Botschaft in einen internen Puffer, der dann zu einem späteren Zeitpunkt selbstständig zum CAN-Controller kopiert wird. In beiden Fällen kehrt `CanIf_Transmit()` sofort nach dem Kopieren der Botschaft zurück, bevor die Botschaft tatsächlich versendet ist.

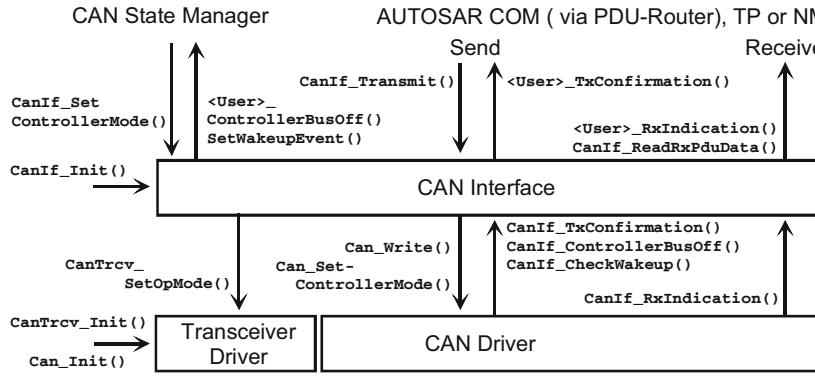

Abb. 8.13 CAN-Protokollstapel mit einer Auswahl von API-Funktionen

Rückmeldungen an die darüberliegende Schicht, also AUTOSAR COM, NM, DCM oder TP bzw. den dazwischengeschalteten PDU-Router sowie den *CAN State Manager*, erfolgen über Callback-Funktionen. Nachdem der CAN-Controller eine Botschaft erfolgreich abgesendet hat, teilt der CAN-Treiber dies dem *CAN Interface* durch Aufruf der Funktion `CanIf_TxConfirmation()` und dieses der Anwendung durch die Funktion `User_TxConfirmation()` mit. Empfängt der CAN-Controller eine Botschaft, meldet der Treiber dies dem *CAN Interface* durch die Callback-Funktion `CanIf_RxIndication()` und dieses der Anwendung durch `User_RxIndication()`. Bevor die Anwendung informiert wird, können der *CAN Message Identifier* und die Länge (*Data Length Code*) der empfangenen Botschaft optional aber zuerst in einem statisch konfigurierten Botschaftsfilter überprüft und gegebenenfalls verworfen werden. Auf diese Weise wird die Akzeptanzfilterung von Botschaften aus Sicht der Anwendung unabhängig davon, ob ein Basic CAN- oder ein Full CAN-Controller verwendet wird (siehe Abschn. 3.1.6). Die darüberliegende Schicht liest die Daten dann mit `CanIf_ReadRxPduData()` und kopiert sie in einen eigenen Botschaftspuffer.

Der *CAN State Manager* verwaltet im Zusammenspiel mit dem *CAN Interface* für jeden CAN-Controller die Zustände *Uninitialized*, *Stopped*, *Started*, *Sleep* und *Bus Off*. Die Zustandsumschaltung erfolgt durch `CanIf_SetControllerMode()` bzw. automatisch bei den Zuständen *Bus Off* und *Sleep* durch Rückmeldungen des CAN-Controllers bzw. CAN-Transceivers über die Treiber, falls der CAN-Controller mehrfach Übertragungsfehler erkannt und sich daher abgeschaltet hat (*Bus Off*) bzw. wenn er im Stromsparmodus *Sleep* eine neue Botschaft empfängt (*Wake Up*).

Unterstützung für TTCAN und CAN FD: Der Protokollstapel für TTCAN nach ISO 11898-4 (Abschn. 3.1.8) hat dieselbe Struktur wie bei CAN. AUTOSAR setzt einen Kommunikationscontroller voraus, der die zeitliche Synchronisation mit dem TTCAN-Buszyklus weitgehend in Hardware erledigt. Der TTCAN-Treiber verfügt im Vergleich zum CAN-Treiber daher nur über einige wenige zusätzliche API-Funktionen, mit denen

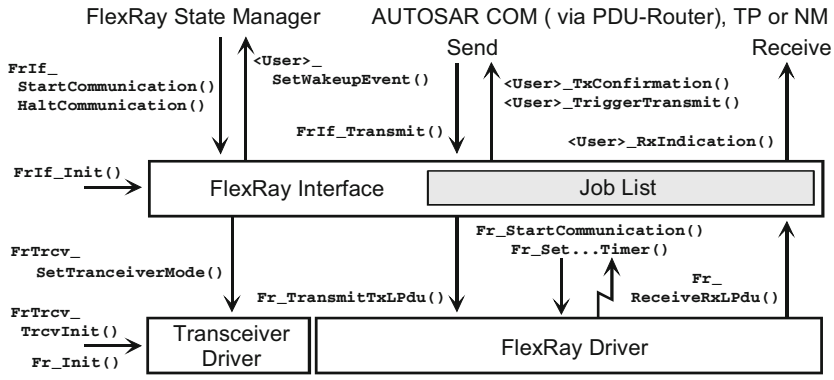

Abb. 8.14 FlexRay-Protokollstapel mit einer Auswahl von API-Funktionen

der Baustein als Zeit-Master oder Slave konfiguriert und der Synchronisationszustand abgefragt werden kann. Das *TTCAN Interface* verwendet intern mit der *Job List* dasselbe Konzept wie der nachfolgend beschriebene FlexRay-Protokollstapel, um die Bearbeitung der Botschaften synchron mit dem TTCAN-Buszyklus zu organisieren.

Für CAN FD sind nur kleinere Anpassungen in der Konfiguration bezüglich der Bitratenumschaltung und wegen größeren Botschaftslänge notwendig.

FlexRay-Protokollstapel: Die Struktur des FlexRay-Protokollstapels (Abb. 8.14) entspricht jener bei CAN, die interne Funktionsweise dagegen weicht deutlich ab. Während beim ereignisgesteuerten CAN-Bus die Botschaften praktisch direkt mit dem Aufruf von `CanIf_Transmit()` versendet werden, ist dies bei FlexRay nur möglich, wenn die Sendefunktion `FrIf_Transmit()` synchron zum Kommunikationsablauf auf dem Bussystem aufgerufen wird (*Immediate Transmit*). Da sich diese Synchronität in der Praxis nicht durchhalten lässt, wird jedoch in der Regel ein entkoppeltes Versenden konfiguriert. Dabei merkt die Sendefunktion `FrIf_Transmit()` die Botschaft zunächst lediglich zum späteren Senden vor (*Decoupled Transmit*). Zudem können Botschaften auch so konfiguriert werden, dass sie auch ohne expliziten Aufruf von `FrIf_Transmit()` regelmäßig versendet werden. Das eigentliche Versenden auf dem Bus erfolgt im entsprechenden FlexRay-Zeitschlitz automatisch durch die interne Ablaufsteuerung im *FlexRay Interface*. Dazu wird bei der Konfiguration des Systems eine sogenannte *FlexRay Job List* erstellt, d. h. eine Liste von Operationen (*Job*), die synchron mit dem Zeitraster des FlexRay-Bussystems abgearbeitet wird. Die wichtigsten Operationen sind:

- *Transmit With Decoupled Buffer Access*: Kopiert die Botschaft mit Hilfe einer von der Anwendung bereitgestellten Callback-Funktion `User_TriggerTransmit()` in einen internen Puffer und von dort später in den Sendepuffer des Kommunikationscontrollers. Durch diesen Mechanismus mit Zwischenspeicherung kann die Anwendung Daten

versenden, ohne sich selbst zwingend mit dem Zeitraster auf dem Bussystem synchronisieren zu müssen (*Decoupled Transmit*).
- *Provide Tx Confirmation*: Bestätigung über das erfolgreiche Versenden einer Botschaft mit Hilfe einer Callback-Funktion `User_TxConfirmation()`.
- *Receive And Indicate*: Kopiert eine empfangene Botschaft aus dem Empfangspuffer des Kommunikationscontrollers in einen internen Zwischenspeicher und ruft danach eine Callback-Funktion `User_RxIndication()` der Anwendung auf, die die Botschaft dann in einen eigenen Puffer kopieren kann (*Immediate Reception*).
- *Receive And Store* und *Provide Rx Indication*: Aufteilung des Kopiervorgangs in den internen Zwischenspeicher und der Mitteilung an die Anwendung zu einem späteren Zeitpunkt in zwei Teiloperationen (*Decoupled Reception*).
- *Prepare PDU*: Dynamisches Umkonfigurieren des FlexRay-Kommunikationscontrollers (siehe Abschn. 3.3.6), falls dieser nicht genügend interne Puffer für alle in einem Kommunikationszyklus zu übertragenden FlexRay-Frames hat.

Die Abarbeitung der Kommunikationsoperationen erfolgt in einer Interrupt-Service-Routine (*FlexRay Job List Execution Function*), die über den FlexRay-Treiber vom Kommunikationscontroller synchron zu den Makroticks des FlexRay-Bussystems aufgerufen wird. Um die Aufrufzeitpunkte zu beeinflussen bzw. weitere Aktivitäten mit dem Bussystem zu synchronisieren, stellen FlexRay-Interface und -Treiber eine Reihe von Funktionen wie `FrIf_Set...Timer()` oder `FrIf_Enable...Timer()` zur Verfügung, wobei die *Timer* sich in diesem Zusammenhang auf das Zyklus- und Makrotick-Zeitraster des FlexRay-Bussystems beziehen.

Der AUTOSAR FlexRay-Spezifikation merkt man an, dass FlexRay parallel zu AUTOSAR entwickelt wurde und weite Teile der Kommunikation auf altbewährten CAN- und LIN-Konzepten beruhen. Die neuen Möglichkeiten von FlexRay sind nicht immer vollständig nutzbar, allerdings werden die Einschränkungen mit neueren AUTOSAR-Versionen zunehmend gelockert. So wurden Datenbotschaften innerhalb des AUTOSAR-Kommunikationsstapels beispielsweise stets auf Basis von sogenannten *Protocol Data Units* (PDU) behandelt, die unabhängig vom Bussystem sein sollen. Mit Rücksicht auf CAN und LIN darf die Länge einer AUTOSAR COM PDU daher zunächst nicht mehr als 8 Byte betragen. Weil FlexRay im Gegensatz zu CAN und LIN aber Botschaften mit bis zu 254 Datenbytes effizienter übertragen könnte, müssen größere Busbotschaften (*FlexRay Frames*) bei der Konfiguration eventuell mit Hilfe eines *Frame Construction Plans* aus mehreren PDUs zusammengesetzt werden. Damit der Empfänger weiß, welche PDUs innerhalb eines Frames sich tatsächlich geändert haben und welche nicht, kann optional ein sogenanntes Update-Bitfeld mitübertragen werden. Um die Problematik zu entschärfen, lässt AUTOSAR für FlexRay auch PDUs mit mehr als 8 Byte zu. Diese können dann aber nicht mehr beliebig auf die anderen Bussysteme umgesetzt werden. Bei Botschaften von AUTOSAR DCM, die in jedem Fall ein Transportprotokoll verwenden, besteht diese Problematik nicht. Ähnlich umständlich wird die bei FlexRay mögliche parallele Übertragung sicherheitskritischer Botschaften über zwei Kanäle unterstützt. Diese muss über die *Job*

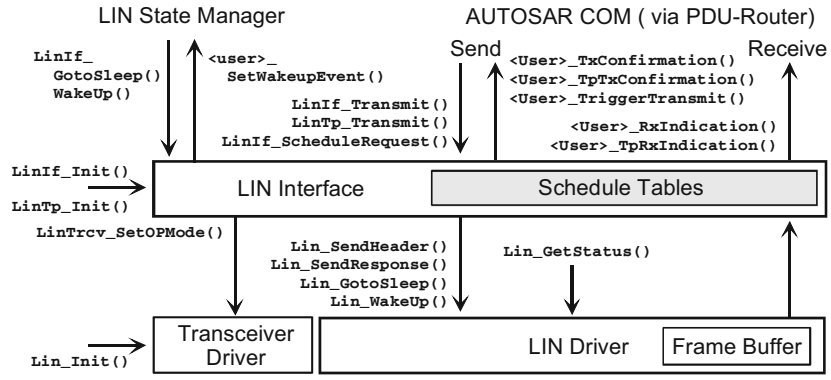

Abb. 8.15 LIN-Protokollstapel mit einer Auswahl von API-Funktionen

Listen für jeden Kanal manuell konfiguriert werden. Eine Überprüfung, ob das Versenden oder Empfangen solcher Botschaften tatsächlich konsistent auf beiden Kanälen erfolgt, ist innerhalb des Protokollstapels zunächst nicht vorgesehen und muss von der Anwendung übernommen werden.

Der *FlexRay State Manager* verwaltet im Zusammenspiel mit dem *FlexRay Interface* für jeden Controller die Zustände *Online* und *Offline* mit den verschiedenen Unterzuständen des FlexRay Protokolls. Die Umschaltung zwischen den Zuständen erfolgt mit Funktionen wie `FrIf_StartControllerCommunication()` oder `FrIf_HaltControllerCommunication()`. Analog zum FlexRay-Controller wird auch der Bustransceiver in den entsprechenden Zustand geschaltet. Der Zugriff auf die Protokoll-Zustände des Kommunikationscontrollers (Abschn. 3.3.6), den Synchronisationszustand oder das Senden von *Wakeup Pattern* und anderen Symbolen erfolgt mit Funktionen wie `FrIf_GetPOCStatus()`, `FrIf_GetSyncState()`, `FrIf_SendWUP()`, `FrIf_AllowColdstart()` usw.

LIN-Protokollstapel: Der LIN-Protokollstapel (Abb. 8.15) implementiert die Protokollversion LIN V2.x für LIN-Master-Steuergeräte (siehe Abschn. 3.2). Geräte mit reiner Slave-Funktionalität werden nicht unterstützt. Im Gegensatz zu CAN und FlexRay ist das Transportprotokoll dabei kein separates Modul, sondern in das *LIN Interface* integriert.

Das Betriebssystem ruft periodisch eine interne Funktion des *LIN Interface* auf, welche die Botschaftstabellen (*Schedule Tables*) abarbeitet und entsprechende LIN Header bzw. Response-Botschaften sendet und empfängt. Die in der LIN-Spezifikation vorgeschlagene Programmierschnittstelle *LIN API* (siehe Abschn. 3.2.8) wird bei AUTOSAR nicht verwendet. Stattdessen kann mit Hilfe der Funktion `LinIf_ScheduleRequest()` zwischen verschiedenen vorkonfigurierten Botschaftstabellen umgeschaltet werden, die einmalig oder periodisch abgearbeitet werden. Mit Hilfe der Funktion `LinIf_Transmit()` können *Sporadic Frames* (siehe Abschn. 3.2.4) versendet werden. Die Funktion `LinTp_Transmit()` erlaubt *Diagnostic Frames*. Dies ist eine etwas unglücklich gewählte Be-

zeichnung, hinter der sich eine Variante des Transportprotokolls ISO 15765-2 verbirgt, mit der auch UDS- oder KWP 2000-Diagnosebotschaften übertragen werden können, die sich aber allgemein für beliebige segmentierte Daten eignet (siehe Kapitel 3.2.5). Dabei gibt es mit `user_ProvideTx/RxBuffer()` ein ähnliches Buffer-Handling wie bei FlexRay. Wie bei CAN und FlexRay wird mit `LinIf_GotoSleep()` und `LinIf_WakeUp()` der Stromsparmodus unterstützt.

Für die Konfiguration des Bussystems wird, wie bei AUTOSAR üblich, ein XML-Format verwendet. Die herkömmliche LIN-Konfiguration über LDF- und NCF-Dateien in der *LIN Configuration Language* (siehe Abschn. 3.2.6) ist nicht vorgesehen. Gegebenenfalls werden die AUTOSAR-Werkzeughersteller daher auch hier Konverterprogramme bereitstellen müssen, um mit anderen LIN-spezifischen Werkzeugen kompatibel zu bleiben. Die dynamische Konfiguration von Slave-Steuergeräten (siehe Abschn. 3.2.7) wird indirekt unterstützt, indem die entsprechenden Botschaften in speziellen Botschaftstabellen vordefiniert werden, die dann bei Bedarf nach Aufruf von `LinIf_ScheduleRequest()` einmalig abgearbeitet werden.

TCP/IP-Ethernet-Protokollstapel: Erstmals wurde Ethernet (Abschn. 3.5) als weiteres Bussystem und TCP/IP als Netzwerkprotokoll in AUTOSAR 4.0 aufgenommen. Der Protokollstapel ist zweigeteilt. Der obere Teil (Abb. 8.16) besteht aus dem TCP/IP-Protokollstack, der die bekannten *Internet* Protokolle TCP/IP und UDP/IP sowie die diverse Hilfsprotokolle implementiert [2, 3], sowie einer *Socket Adaptor* Schicht, die die Verbindung zum PDU-Router bzw. zum *Diagnostic Over IP*-Protokolls nach ISO 13400 (siehe Abschn. 4.6) herstellt. Über den *Socket Adaptor* greift auch *UDP NM*, das Teil des im folgenden Abschnitt beschriebenen Netzmanagements ist, auf das Bussystem zu.

Zwischen dem *Socket Adapter* und dem TCP/IP-Stack wird die aus der PC-Welt bekannte *Socket API* verwendet werden. *Sockets* sind eine Abstraktion einer Kommunikationsverbindung, die aus der Kombination der lokalen IP und Portadressen und der entsprechenden Werte der Gegenseite besteht. UDP sendet einzelne Botschaften, die auch von verschiedenen Gegenstellen empfangen werden können und nicht bestätigt werden. TCP-Verbindungen dagegen sind stets 1 : 1-Verbindungen, die einen unbeschränkten Datenstrom austauschen und Empfangsbestätigungen und automatische Übertragungswiederholung im Fehlerfall sicherstellen.

Alle *Sockets* werden mit den Funktionen `TcpIp_GetSocket()` und `TcpIp_Bind()` konfiguriert Der *Socket*, der über die Funktion `TcpIp_TcpConnect()` die TCP-Verbindung zu einer Gegenstelle aktiv aufbaut, wird als *Client Socket* bezeichnet. Die Gegenstelle, die mittels `TcpIp_TcpListen()` auf eingehende Verbindungen wartet, ist ein *Server Socket*. Wenn eine Verbindung zustande kommt, wird dies auf der Client-Seite durch die Callback-Funktionen `SoAd_TcpConnected()` bzw. beim Server durch `SoAd_TcpAccepted()` angezeigt. Das Senden erfolgt mit den Funktionen `TcpIp_Tcp/UdpTransmit()`. Beim Empfang von Daten wird die übergeordnete Schicht wieder durch eine Callback-Funktion `SoAd_RxIndication()` informiert und kann die Daten dann über den mitgelieferten Pointer auslesen.

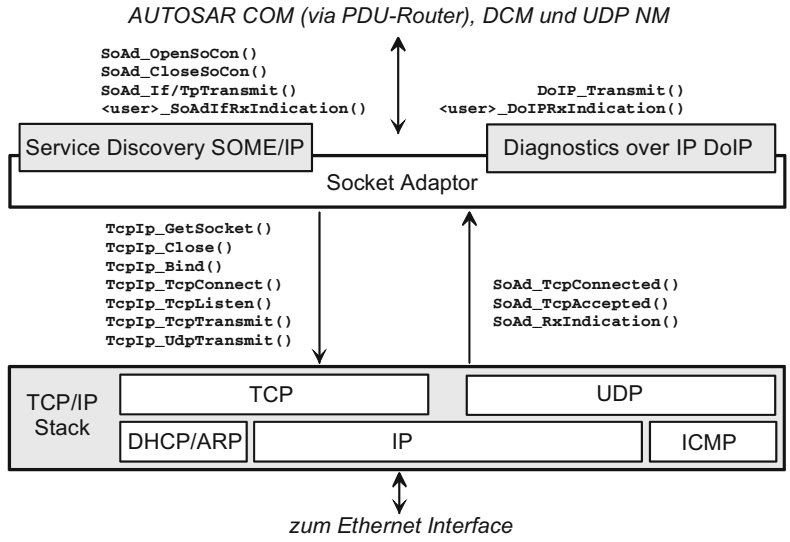

Abb. 8.16 Oberer Teil des TCP/IP-Ethernet-Protokollstapels

Die Aufgabe der *Socket Adapter* Schicht besteht darin, die verbindungsorientierte Socket-TCP/IP-Welt an die botschaftsorientierte PDU-Welt der Steuergeräte anzubinden. Bei der Konfiguration wird festgelegt, welche PDUs durch TCP bzw. durch UDP übertragen werden und welches die zugehörigen IP- und Portadressen sind. Im einfachsten Fall erzeugt der *Socket Adapter* für jede PDU einen eigenen *Socket*. Da deren Verwaltung zur Laufzeit jedoch sehr speicher- und zeitaufwendig sein kann, kann die *Socket Adapter* Schicht optional auch mehrere PDUs in einer einzigen TCP- bzw. UDP-Botschaft zusammenfassen. Um solche zusammengefassten Botschaften zu verwalten, kann ein zusätzlicher PDU Header eingefügt werden. Der Header besteht aus einer Kennziffer sowie einer Längenangabe, die jeweils 4 Byte groß sind.

Der untere Teil des Protokollstapels (Abb. 8.17) ist wie bei den anderen Bussystemen in eine allgemeine Interfaceschicht sowie eine Treiberschicht für den Ethernet-Controller und den Ethernet-Transceiver unterteilt. Der Zustand des Ethernet-Busses wird durch den *Ethernet State Manager* verwaltet.

8.5 Netzmanagement AUTOSAR NM

Das *AUTOSAR Netzmanagement* (NM) übernimmt viele Prinzipien aus OSEK NM (Abschn. 7.2.3), ist aber nicht aufwärts kompatibel dazu. Auch AUTOSAR NM sammelt Informationen über die Buskommunikation aller am Bus angeschlossenen Steuergeräte. Zusammen mit dem *ECU State Manager,* dem *Communication State Manager* und dessen untergeordneten, busspezifischen *State Managern* wird der Betrieb des eigenen Steuerge-

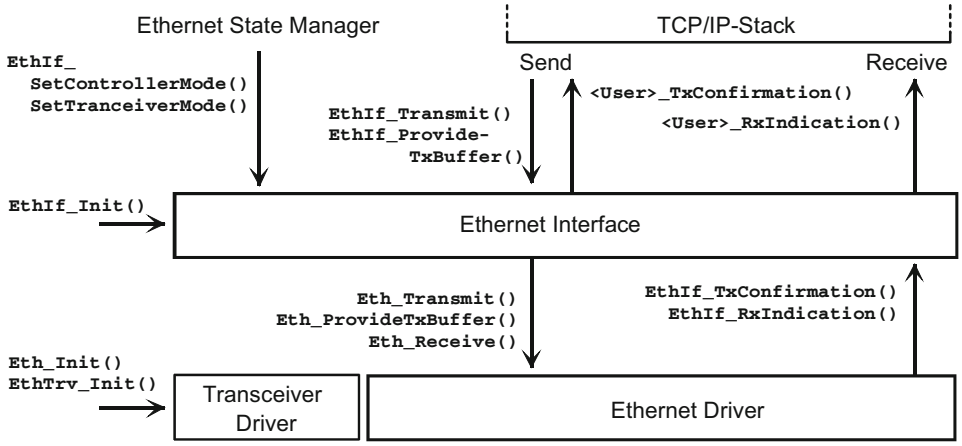

Abb. 8.17 Unterer Teil des TCP/IP-Ethernet-Protokollstapels

rätes beeinflusst. Hauptaufgabe ist dabei der kontrollierte Übergang des Steuergerätes und seiner Buskommunikation zwischen dem stromsparenden Betrieb (*Sleep*) und dem normalen Betrieb (*Run bzw. Normal*) mit den entsprechenden *Wakeup* bzw. *Shutdown* Sequenzen (Abb. 8.6).

Wie der *Communication Manager* besteht auch das Netzmanagement aus einem busunabhängigen Interface Modul (*Generic Network Management Interface*) und busspezifischen Submodulen für CAN, FlexRay, LIN und Ethernet (Abb. 8.18).

AUTOSAR verwendet ein einfaches dezentrales und direktes Netzmanagement (vgl. Abschn. 7.2.3):

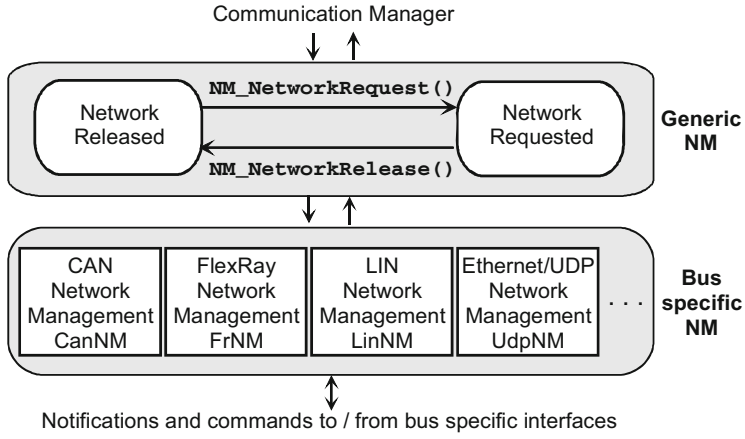

Abb. 8.18 Struktur der Netzmanagement-Komponenten

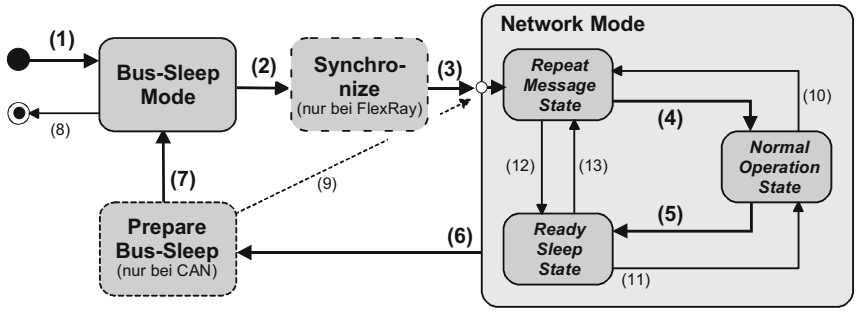

Abb. 8.19 Busspezifische Netzwerkzustände

- Benötigt ein Netzteilnehmer (*Knoten*) das Bussystem (*Network Requested*), so sendet er zyklisch eine spezielle NM-Botschaft (*NM PDU*).
- Benötigt der Netzteilnehmer das Bussystem nicht mehr (*Network Released*), so stellt er das Senden der zyklischen NM-Botschaft ein.

Dabei gibt es zwei Arten von Steuergeräten: *Aktive NM-Knoten* senden selbst NM-Botschaften und empfangen solche Botschaften von anderen Geräten. *Passive NM-Knoten* senden selbst keine NM-Botschaften, werten empfangene NM-Botschaften aber ebenfalls aus. Sendet kein anderer Knoten mehr NM-Botschaften, so wird diese Information vom Netzmanagement an den *Communication Manager* und von dort an den *ECU State Manager* weitergegeben. Dieser entscheidet, ob das Steuergerät ebenfalls die Kommunikation einstellen darf und versetzt gegebenenfalls den Kommunikationscontroller und den Bustreiber bzw. das gesamte Steuergerät in den Ruhezustand. Benötigt ein Netzteilnehmer wieder das Bussystem, so beginnt er damit, erneut NM PDUs zu versenden. Die Kommunikationscontroller bzw. Bustreiber der Steuergeräte müssen so aufgebaut sein, dass sie derartige Botschaften auch im Ruhezustand erkennen und sich selbst und den zugeordneten Mikrocontroller z. B. über einen Interrupt wieder aktivieren. Das Netzmanagement informiert daraufhin den *Communication Manager*, dieser reaktiviert über die untergeordneten *Bus State Manager* die Bussysteme und über den *ECU State Manager* das restliche Steuergerät (*Wakeup Event*). Jedes der am Netzmanagement beteiligten Module besitzt ein eigenes komplexes Zustandsmodel, das durch diverse Timer und Ereignisse gesteuert wird.

Die internen Zustände des busspezifischen Netzmanagements (Abb. 8.19) und das Format der NM-Botschaften (*NM PDU*) sind bei CAN und bei FlexRay annähernd gleich. Die NM-Botschaft (Abb. 8.20) enthält den *NM Control Bit Vector,* die Identifikation des Senders (*Source Note ID*) und in den verbleibenden Bytes der max. 8 Byte langen Botschaft beliebige Daten (*User Data*).

In der obersten Ebene, dem *Generic NM* (Abb. 8.18), kennt der lokale Knoten für jedes Bussystem die Zustände *Network Released* und *Network Requested*, die anzeigen, ob der Knoten selbst mit anderen Knoten kommunizieren will. Die Umschaltung zwi-

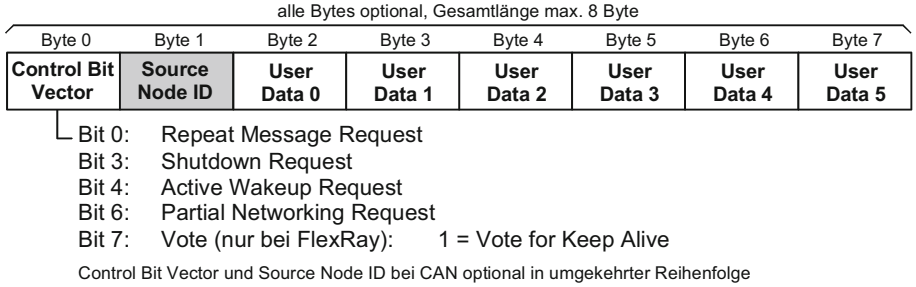

Abb. 8.20 Botschaftsaufbau für das Netzmanagement (*NM PDUs*)

schen diesen Zuständen erfolgt über die API-Funktionen `Nm_NetworkRequest()` bzw. `Nm_NetworkRelease()`. Darunter liegen die Zustände nach Abb. 8.19:

- Der ***Bus Sleep Mode*** ist der Anfangszustand nach der Initialisierung durch die API-Funktion `Nm_Init()` (Zustandsübergang 1 in Abb. 8.19) und der finale Zustand, wenn erkannt wird, dass kein Netzteilnehmer mehr das Netzwerk benötigt (7) bzw. vor dem Abschalten (8). In diesem Zustand werden keine NM-Botschaften gesendet.
- Bei FlexRay existiert der Zwischenzustand ***Synchronize***, in dem das Netzmanagement vor dem Übergang zum Netzbetrieb (2, 3) wartet, bis die Synchronisation mit dem Bussystem abgeschlossen ist.
- Der Netzbetrieb ***Network Mode*** zerfällt in drei Unterzustände:
  Sobald der lokale Knoten signalisiert, dass er das Bussystem benötigt (*Network Requested*), wechselt das Netzmanagement in den ***Repeat Message State*** (3). Sofern das Steuergerät als *aktiver Knoten* konfiguriert ist, beginnt es mit dem zyklischen Senden von NM-Botschaften. Durch eine NM-Botschaft mit gesetztem *Repeat Message Request Bit* (Abb. 8.20) kann der Sender die anderen Busteilnehmer jederzeit in den *Repeat Message State* versetzen, d. h. eine Synchronisation des Netzmanagements für das gesamte Bussystem erzwingen (10, 13). Nach Ablauf einer definierbaren Mindestzeit, die sicherstellen soll, dass *aktive Knoten* in jedem Fall für andere Netzwerk-Teilnehmer sichtbar sind, wechselt das Netzmanagement aus dem *Repeat Message* in den *Normal Operation State* (4).
  Im ***Normal Operation State*** senden *aktive Knoten* weiter zyklisch NM-Botschaften und halten so den Bus aktiv. Wird das Netzwerk vom lokalen Knoten nicht mehr benötigt (*Network Released*), so wechselt er in den *Ready Sleep State* (5).
  Im ***Ready Sleep State*** sendet der lokale Knoten keine weiteren NM-Botschaften mehr, der Bus bleibt aber noch aktiv. Der Zustand wird solange beibehalten, wie von mindestens einem anderen Netzteilnehmer NM-Botschaften empfangen werden. Sendet kein anderer Netzteilnehmer mehr NM-Botschaften, so wechselt das NM (bei CAN über den *Prepare Bus Sleep Mode*) in den *Bus Sleep Mode* (6, 7).

- Der ***Prepare Bus Sleep Mode***, den es nur bei CAN gibt, ist dem *Bus Sleep Mode* beim Übergang (6) aus dem Netzbetrieb *Network Mode* vorgeschaltet. Dabei werden noch alle anstehenden Botschaften versendet, aber keine neuen Botschaften mehr für das Senden angenommen. Nach Ablauf einer Wartezeit folgt dann der *Bus Sleep Mode* (7) bzw. erneut der Netzbetrieb (9), falls das Bussystem während der Wartezeit wieder benötigt oder die NM-Botschaft eines anderen Knotens empfangen wird. Bei FlexRay ist der *Prepare Bus Sleep Mode* nicht notwendig, da dort alle Zustandsübergänge ohnehin im gesamten Netz synchron erfolgen.

Besonderheiten beim CAN-Netzmanagement CanNm: Für die zuverlässige Funktion des Netzmanagements müssen verschiedene Warte- und Überwachungszeiten im gesamten Netz konsistent konfiguriert werden. Um dabei zu vermeiden, dass alle CAN-Knoten gleichzeitig mit dem Senden beginnen und eine hohe Spitzenbuslast erzeugen, kann für jeden Knoten die Verzögerungszeit zwischen dem Eintritt in den *Network Mode* und dem Sendebeginn individuell festgelegt werden.

Um die mittlere Buslast zu reduzieren, kann optional der sogenannte *Bus Load Reduction* Mechanismus verwendet werden. Dabei werden als Periodendauer für das Versenden der NM-Botschaften alternativ zwei verschiedene Werte verwendet. Normalerweise wird die Sendeperiode bei allen Knoten auf einen festen, im gesamten Netz gleichen Defaultwert gesetzt. Empfängt ein Knoten bei aktivierter Lastreduktion dagegen vor Ablauf seiner eigenen Sendeperiode die NM-Botschaft eines anderen Busteilnehmers, so startet er seinen Sendezeitgeber erneut mit einem zweiten, aber kleineren und für alle Knoten unterschiedlich konfigurierten Wert. Dadurch verringert sich die Menge der tatsächlich versendeten NM-Botschaften allmählich. Am Ende bleiben nur noch die zwei Knoten mit den beiden kleinsten Sendeperioden übrig. Die Buslastreduktion ist nur im *Normal Operation Mode* wirksam, so dass im *Repeat Message State* weiterhin alle aktiven Knoten für das Netzmanagement erkennbar bleiben.

Besonderheiten beim FlexRay-Netzmanagement FrNM: Im Gegensatz zu CAN, bei dem die Periodendauern der NM-Botschaften und die verschiedenen Warte- und Überwachungszeiten als echte Zeiten definiert sind, sind die entsprechenden Werte bei FlexRay Vielfache des FlexRay-Kommunikationszyklus.

Die NM-Botschaften können sowohl im statischen als auch im dynamischen Segment des FlexRay-Kommunikationszyklus gesendet werden (vgl. Abschn. 3.3). Im statischen Teil kann die NM-Botschaft auf Wunsch das *Network Management* Feld eines FlexRay-Frames nutzen. Im dynamischen Teil ist FlexRay *Cycle Multiplexing* möglich. Außerdem ist es im selben Netz zulässig, sowohl NM-Botschaften zu senden, die ausschließlich das *Control Bit* Feld mit gültigem *Vote Bit* enthalten (sogenannte *NM Vote PDU*), als auch vollständige Botschaften mit Datenfeld, bei denen das *Vote Bit* im *Control Bit Feld* wahlweise gültig ist oder ignoriert werden darf. Die dadurch entstehenden Kombinationsmöglichkeiten sind recht unübersichtlich.

Der sogenannte *Repetition Cycle* beschreibt die Anzahl der Kommunikationszyklen, die notwendig ist, damit jeder Busknoten mindestens eine NM-Botschaft gesendet hat. Der in Abb. 8.19 dargestellte *Synchronize* Zustand ist vorgesehen, um den lokalen Knoten mit dem *Repetition Cycle* der anderen Teilnehmer zu synchronisieren. Alle Zustandsübergänge erfolgen im Netz stets synchron am Ende dieses *Repetition Cycles*.

LIN-Netzmanagement LinNM und UDP-Netzmanagement UdpNM: Seit AUTOSAR 4.0 ist auch ein Netzmanagement für LIN-Master-Steuergeräte sowie für TCP/IP-Ethernet-Systeme definiert. Das Ethernet-Netzmanagement verwendet UDP-Botschaften. Jeder Bus wird wiederum über eine Zustandsmaschine mit den Zuständen *Bus Sleep* und *Network Mode* verwaltet.

Buskoordination in Gateways und OSEK NM: Die *Generic NM*-Spezifikation beschreibt, wie das Netzmanagement für mehrere Bussysteme zusammenwirken soll, wenn die Busse koordiniert in den *Bus Sleep* Zustand übergehen müssen. Solche Koordinationsfunktionen werden in der Regel in Gateway-Steuergeräten realisiert. Im Wesentlichen besteht die Koordination darin, dass das Gateway die Bussysteme mit NM-Botschaften aktiv hält, bis alle anderen Geräte auf den beteiligten Bussen das Senden der NM-Botschaften eingestellt haben. Dabei darf das Gateway für einzelne Bussysteme statt AUTOSAR NM-Botschaften auch OSEK NM-Botschaften verwenden. Zur Integration von OSEK NM in AUTOSAR NM verweist AUTOSAR allerdings nur auf herstellerspezifische Erweiterungen.

Partial und Pretended Networking: Während bisher zur Energieeinsparung komplette Bussysteme oder ganze Steuergeräte abgeschaltet wurden, soll zukünftig auch das Abschalten von Teilnetzen oder einzelnen Funktionen in Steuergeräten möglich sein.

Für das *Partial Networking* wird jedes an einem Bussystem angeschlossenen Steuergeräte einer oder mehreren Gruppen (*Partial Network Cluster*) zugeordnet. Wenn nun in einem bestimmten Fahrzustand des Fahrzeugs die Funktionen einer dieser Gruppen nicht notwendig sind, stellt diese Gruppe das Senden und Empfangen von Busbotschaften vollständig ein, während die übrigen Steuergeräte am Bus weiter miteinander kommunizieren. Das entsprechende Kommando wird über eine NM Botschaft verteilt (Abb. 8.20). Durch eine spezielle Busbotschaft, die von den Geräten auch im (teil)-abgeschalteten Zustand erkannt werden muss, können die *schlafenden* Geräte reaktiviert werden. Dazu sind geeignete Bustransceiver und Kommunikationscontroller notwendig, die es derzeit nur für CAN gibt (siehe Abschn. 3.1.9). Zukünftig soll es auch für FlexRay und Ethernet Lösungen geben.

Beim *Pretended Networking* stellen Steuergeräte nicht die Buskommunikation ein, sondern schalten intern Hardware- und Softwareteilfunktionen ab (*ECU Degradation*), simulieren nach außen hin aber ein unverändertes Kommunikationsverhalten. Das Umschalten zwischen Normalbetrieb und eingeschränktem Betrieb kann jedes Steuergerät unabhängig von anderen ausführen. Daher kann es bei der Neuentwicklung von Geräten schrittweise eingeführt werden, während beim *Partial Networking* praktisch alle Geräte an einem Bussystem angepasst werden müssen.

8.6 Virtual Function Bus VFB, Runtime Environment RTE und Softwarekomponenten

Aus Sicht der Anwendungsentwickler besteht ein AUTOSAR Steuergerät aus Softwarekomponenten, die über so genannte Ports miteinander kommunizieren (Abb. 8.21). Auch die oberste Schicht der Basissoftware erscheint dem Anwendungsentwickler als eine Reihe von Softwarekomponenten. Die interne Funktionsweise des Betriebssystems oder des Kommunikationsprotokollstacks bleiben ihm weitgehend verborgen. Nach den AUTOSAR-Regeln für Softwarekomponenten verwendet der Anwendungsentwickler API-Funktionen anderer Module niemals direkt oder greift direkt auf Funktionen oder Variablen anderer Komponenten zu, sondern benutzt stets die jeweiligen Ports mit ihren definierten Schnittstellen.

Aus programmtechnischer Sicht sind die Port-Schnittstellen API-Funktionen bzw. Makros des *Runtime Environments* wie `Rte_Read...()` oder `Rte_Write...()`, die bei der Entwicklung des Systems mit Hilfe eines Generatorwerkzeugs aus den Schnittstellenbeschreibungen der Komponenten automatisch erzeugt werden (Tab. 8.3). Der gesamte Datenaustausch inklusive gegebenenfalls nötiger Typkonversionen wird durch diese RTE-Funktionen gekapselt. Aus Sicht der Softwarekomponente spielt es dabei keine Rolle, ob der Datenaustausch innerhalb desselben Steuergerätes erfolgt und damit einfach auf Variablen oder Funktionen im Speicher des Steuergerätes zugegriffen wird, oder ob die Komponente sich in einem anderen Steuergerät befindet. In diesem Fall kommuniziert das RTE über AUTOSAR COM und eines der Bussysteme mit dem anderen Steuergerät und überträgt die Daten mit Hilfe von Netzwerkbotschaften bzw. ruft die gewünschten Prozeduren im anderen Steuergerät auf (*Remote Procedure Call RPC* bzw. *Remote Method Invocation RMI*).

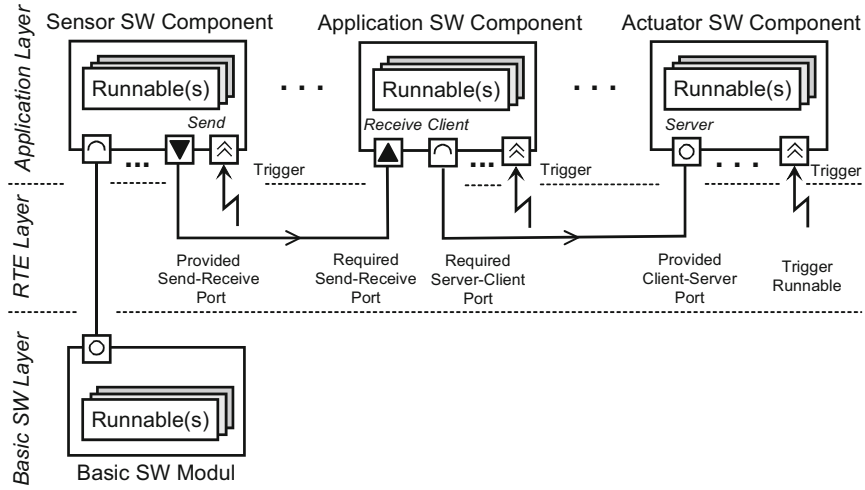

Abb. 8.21 Kommunikation von Softwarekomponenten über das RTE

Diese Grundidee, dass alle Softwarekomponenten über alle Steuergeräte in einem Fahrzeug hinweg so miteinander kommunizieren sollen, als ob sie in einem einzigen großen Steuergerät ablaufen würden, wird von AUTOSAR als *Virtual Functional Bus VFB* bezeichnet. Das *Runtime Environment* (in Verbindung mit dem Kommunikationsstack) ist die Steuergeräte-interne Implementierung dieses *Virtual Functional Bus* Konzeptes. Die theoretisch mögliche, beliebige Verteilung von Funktionen auf verschiedene Steuergeräte beeinflusst natürlich durch unterschiedliche Übertragungsverzögerungen deren Laufzeitverhalten. Daher wird es notwendig, auch das zeitliche Verhalten von Softwarekomponenten eindeutig zu beschreiben. Mit AUTOSAR 4.0 wurde daher mit den *Timing Extensions* eine Möglichkeit geschaffen, zeitliche Anforderungen auf der System- bzw. *Virtual Function Bus*-Ebene zu definieren und das Zeitverhalten von Anwendungs- und Basissoftwarekomponenten anzugeben.

Bei den Ports unterscheidet AUTOSAR zwischen *Required Ports* („Eingänge"), d. h. einer Schnittstelle einer anderen Komponente, die verwendet werden soll, sowie *Provided Ports* („Ausgänge"), d. h. einer Schnittstelle, die den anderen Komponenten zur Verfügung gestellt wird. Außerdem wird danach unterschieden, ob über die Schnittstelle vorzugsweise Daten ausgetauscht (*Sender-Receiver Port*) oder ob Funktionen aufgerufen werden (*Server-Client Port*). Diese vier möglichen Kombinationen werden durch unterschiedliche grafische Notationen kenntlich gemacht (Abb. 8.21). Die Daten oder Funktionen, die sich hinter dem Port verbergen, werden als *Interface* bezeichnet. Die Verbindung zwischen den Ports ist ein *Connector*, wobei ein *Required Port* nur mit einem *Provided Port* verbunden werden kann, wenn das *Interface* der beiden kompatibel ist, d. h. der *Provided* Port mindestens die Daten und Funktionen bereitstellt, die der *Required Port* verwenden will.

Das *Client-Server*-Interface kann synchron oder asynchron arbeiten, wie dies bereits bei den Hardwaretreibern beschrieben wurde. Die Funktionen des Interface können beliebig vordefinierte Eingabe- und Rückgabeparameter verwenden. Ein Server kann von einem oder mehreren Clients aus aufgerufen werden (1 : 1 oder 1 : n Zuordnung *Provided Port* (Server) zu *Required Port* (Client)).

Das *Sender-Receiver*-Interface kann sowohl einfache Datentypen wie boolesche Variablen, Integer-, Real-Werte, Charaktergrößen oder Bitfelder (*Opaque Data Type*) als auch komplexe Datentypen wie Strings, Arrays und Strukturen (*Records*) austauschen. Von den Daten wird im RTE der jeweils letzte geschriebene Wert zwischengespeichert (als *Data Distribution* Semantik bezeichnet), optional kann aber auch eine FIFO-Warteschlange auf der Empfängerseite (*Queue*) angelegt werden (auch als *Event Distribution* Semantik bezeichnet). Neben dem *expliziten* Lesen und Schreiben über API-Funktionen (Tab. 8.3), bei denen der Zugriff zu beliebigen Zeitpunkten direkt auf die in der RTE zwischengespeicherten Werte erfolgt, ist (bei sogenannten *Category 1 Runnables*, die intern nicht auf äußere Triggerereignisse waren, siehe unten) auch ein sogenannter *impliziter* Zugriff möglich. Dabei werden für eine Komponente beim Start einer *Runnable Entity* (siehe unten) lokale Kopien der Daten angelegt, mit denen die Komponente dann arbeitet. Erst wenn diese Komponente wieder endet, werden die veränderten Werte wieder in die RTE-Schicht zurückkopiert. Beim *Sender-Receiver*-Interface ist eine 1 : 1, eine 1:n oder eine n : 1 Zuordnung zwischen

Tab. 8.3 Auswahl von RTE API-Funktionen. Alle API-Namen werden bei der Generierung um den Namen des Moduls bzw. Ports ergänzt

Funktionen für ein Sender/Receiver Port-Interface	
Explizites Lesen und Schreiben	

`Rte_Read()`
`Rte_Write()`
`Rte_Invalidate()`
	Lesen bzw. Schreiben eines Datenwertes (ohne FIFO-Queue, d. h. der aktuelle Wert wird gelesen bzw. überschrieben).
	Datum als ungültig kennzeichnen bzw. auf Initialisierungswert zurücksetzen

`Rte_Receive()` `Rte_Send()`	Empfangen bzw. Senden eines Datenwertes (mit FIFO, d. h. Daten werden in derselben Reihenfolge empfangen, in der sie gesendet wurden). Beim Lesen wird (mit Timeout) gewartet, bis ein Datenwert empfangen wird (*Data Received Event oder Data Receive Error Event*). Beim Senden wird nur der Sendeauftrag an AUTOSAR COM erteilt, aber nicht gewartet, bis das Senden tatsächlich erfolgt ist
`Rte_Feedback()`	Warten, bis das Senden eines Datums nach Aufruf von `Rte_Send()` tatsächlich erfolgt ist (*Data Send Completed Event*)
Implizites Lesen und Schreiben	
`Rte_IRead()` `Rte_IWrite()`	Lesen eines Datenwerts Schreiben eines Datenwertes
Funktionen für ein Client-Server Port-Interface	
`Rte_Call()`	Synchroner oder asynchroner Aufruf einer Server-Funktion
`Rte_Result()`	Abfrage der Rückgabewerte einer asynchron aufgerufenen Server-Funktion. Wartet, bis die Funktion ausgeführt wurde (*Asynchronous Server Call Returns Event*)
Kommunikation und Synchronisation innerhalb einer Softwarekomponente	
`Rte_Irv(I)Read()` `Rte_Irv(I)Write()`	Explizites oder implizites Lesen und Schreiben von *Interrunnable Variablen*
`Rte_Enter()` `Rte_Exit()`	Synchronisation durch kritische Abschnitte (*Exclusive Areas*, *Critical Sections*)
Betriebsmodi des Steuergerätes	
`Rte_Mode()` `Rte_Switch()`	Lesen oder Umschalten eines der Betriebsmodi über den *ECU State Manager*, *Communication Manager* oder eine andere Komponente
Allgemeiner Ablauf	
`Rte_Start()` `Rte_Stop()`	Initialisieren und Beenden des RTE durch den *ECU State Manager*

Provided Port (Sender) und *Required Port* (Receiver) möglich. Eine automatische Synchronisation zwischen mehreren Sendern oder Empfängern erfolgt aber nicht.

Bei den Softwarekomponenten unterscheidet AUTOSAR die *normalen* Komponenten der Anwendungssoftware sowie diejenigen Teile der Basissoftware, die mit der Anwendungssoftware zusammenarbeiten. Auch dabei ruft die Anwendungssoftware keine Funktionen der Basissoftware direkt auf, sondern interagiert ebenfalls über das RTE. Diese Komponenten der Basissoftware, z. B. AUTOSAR COM, werden im AUTOSAR-Jargon als *Ser-*

Tab. 8.4 Triggerereignisse (*RTE Events*) und Aktionen

Bezeichnung	Ereignis	Aktivieren (Starten) einer Task	Fortsetzen ab einem Wartepunkt
Timing Event	Auslösen eine Zeitgebers (OSEK OS Alarm) für das periodische Triggern einer *Runnable Entity*	Ja	Nein
Mode Switch Event	Nach Umschalten eines der Steuergerätebetriebsmodi	Ja	Nein
Ereignisse bei einem Sender/Receiver Port-Interface			
Data Received Data Receive Error	Empfang einer Datenbotschaft Empfangsfehler Beim Empfang von Daten kann optional ein Datenfilter konfiguriert werden, so dass der Wertebereich von Daten o. Ä. überprüft werden kann.	Ja Ja	Ja Nein
Data Send Completed	Sendebestätigung einer Botschaft	Ja	Ja
Ereignisse bei einem Client-Server Port-Interface			
Operation Invoked	Aufruf einer Server-Funktion	Ja	Nein
Asynchronous Server Call Returns	Abschluss des asynchronen Aufrufs einer Server-Funktion	Ja	Ja

vice Komponenten und deren Ports als *Service Ports* bezeichnet. Schnittstellen, die über Betriebszustände des Steuergeräts bzw. eines Teils des Steuergerätes, z. B. eines Bussystems, informieren, werden *Mode Ports* genannt. Im Gegensatz zu zwei Anwendungskomponenten, die auch über Steuergerätegrenzen hinweg kommunizieren können, kann eine Anwendungskomponente nur die Dienste der Basissoftware im eigenen Steuergerät in Anspruch nehmen.

Neben der Kommunikation zwischen den Softwarekomponenten ist das *Runtime Environment* auch dafür verantwortlich, die Softwarekomponenten zu triggern. Jede Softwarekomponente enthält eine oder mehrere Prozeduren (*Runnable Entities*), die durch das RTE gestartet werden können. Der Start einer Prozedur kann einmalig oder periodisch erfolgen. Eine Prozedur kann entweder nach dem Durchlaufen sofort wieder enden (*Category 1 Runnable*, entspricht einer OSEK/OS *Basic Task*) oder intern an einem sogenannten *Wartepunkt* auf ein weiteres Triggerereignis warten (*Category 2 Runnable*, entspricht einer OSEK/OS *Extended Task*). Ein solches Triggerereignis, als *RTE Event* bezeichnet, kann beispielsweise der Empfang eines neuen Datenwertes oder ein Signalisierungsereignis sein, mit dem das Versenden eines Datenwertes oder der Aufruf einer Funktion bestätigt wird (Tab. 8.4).

Für den Datenaustausch zwischen den *Runnable Entities* innerhalb einer Softwarekomponente stehen (Komponenten)-globale Variable, die sogenannten *Interrunnable Variables* zur Verfügung, auf die ebenfalls mit expliziten oder impliziten Lese- und Schreiboperatio-

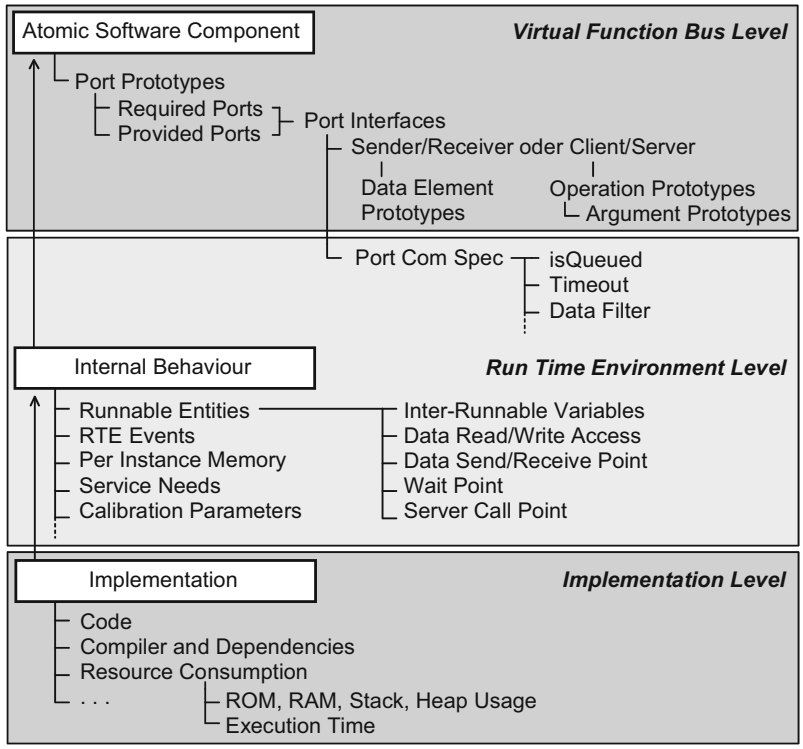

Abb. 8.22 Konfiguration einer AUTOSAR-Softwarekomponente (vereinfacht)

nen zugegriffen werden kann. Zur Synchronisation dienen die aus anderen Betriebssystemen bekannten kritischen Abschnitte, die bei AUTOSAR als *Exclusive Areas* bezeichnet werden.

Aus dem Vorgenannten sollte klar geworden sein, weshalb AUTOSAR Softwarekomponenten die API-Funktionen des Betriebssystems oder des COM-Moduls im Idealfall gar nicht direkt verwenden müssen. Das *Runtime Environment* bietet mit den Ports und den Triggermöglichkeiten abstrakte Mechanismen für den Datenaustausch und die Ablaufsteuerung. Diese abstrakten Mechanismen werden, so die AUTOSAR-Vorstellung, bei der jeweiligen Komponenten-Konfiguration in einem Satz von XML-Dateien für jede Softwarekomponente (Abb. 8.22) beschrieben und dann durch Generierungswerkzeuge mehr oder weniger automatisch in entsprechenden Mechanismen und API-Aufrufe von AUTOSAR OS und AUTOSAR COM abgebildet.

Diese Generierungswerkzeuge, die nicht vom AUTOSAR-Konsortium sondern von kommerziellen Anbietern bereitgestellt werden sollen, benötigen, soweit man das an ersten vorliegenden Prototypen beurteilen kann, aber sehr viel Unterstützung durch menschliche Entwickler. Die Aufteilung der *Runnable Entities* auf AUTOSAR OS Tasks und der *RTE*

Events auf OS Events, Alarme und Callback/Notification Funktionen, die Einrichtung der Scheduling Tabellen und die Verteilung der Datenwerte auf Busbotschaften ist aufwendig.

Die Belange der Applikation der Steuergeräte berücksichtigt AUTOSAR durch die Bereitstellung von *Calibration Ports*, die den Zugriff auf die Einstellparameter der Softwarekomponenten erlauben. Messgrößen werden wie andere Signale über gewöhnliche Port-Schnittstellen nach außen geführt. Die Beschreibung der Mess- und Kalibrierschnittstelle und die Umrechnung zwischen physikalischen und steuergeräteinternen Werten lehnt sich stark an das von MDX und CDF bekannte Modell an und soll zukünftig mit den ASAM-Standards (siehe Abschn. 6.5.2) harmonisiert werden.

8.7 Beispiel einer einfachen Anwendung

Als Beispiel soll die Überwachung des Ölkreislaufs eines Motors betrachtet werden (Abb. 8.23). Die Öltemperatur und der Ölstand werden durch analoge Sensoren gemessen. Ein Öldruckschalter erfasst den Öldruck binär. Die Motordrehzahl wird von einem hier nicht näher betrachteten weiteren System zur Verfügung gestellt. Die Überwachung vergleicht diese Werte mit intern vorgegebenen Grenzwerten. Wenn ein Fehler festgestellt wird, wird der Fahrer über eine Warnlampe informiert. Außerdem kann der Fahrer mit Hilfe einer Check-Taste die aktuelle Öltemperatur und den Ölstand abfragen, die dann auf einem LCD-Display angezeigt werden. Das Problem wird in drei Teilaufgaben aufgetrennt. In der Signalvorverarbeitung werden die Sensorsignale linearisiert und gefiltert. Anschließend erfolgt die eigentliche Überwachung, bei der überprüft wird, ob die Signale im zulässigen Bereich liegen und untereinander plausibel sind. In der dritten Stufe wird die Warnlampe angesteuert beziehungsweise bei Druck auf die Check-Taste die Öltemperatur und der Ölstand angezeigt.

Im AUTOSAR-Entwurf (Abb. 8.24) sind die Sensoren und die Anzeigeinstrumente, d. h. die Hardwarekomponenten, gestrichelt dargestellt. Die drei Teilaufgaben werden durch je eine Softwarekomponente (SWC) abgebildet. Die Softwarekomponenten haben mehrere Ports, die über die AUTOSAR RTE-Schicht miteinander verknüpft werden. Mit

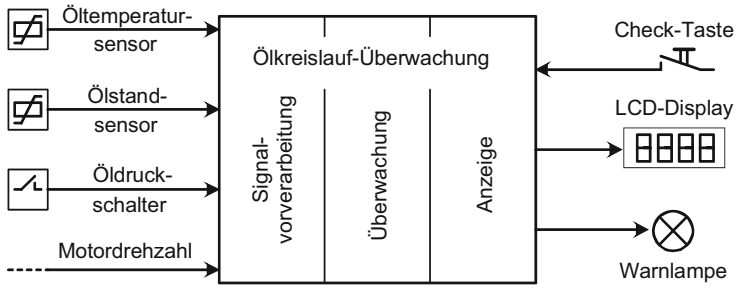

Abb. 8.23 Ölkreislauf-Überwachung

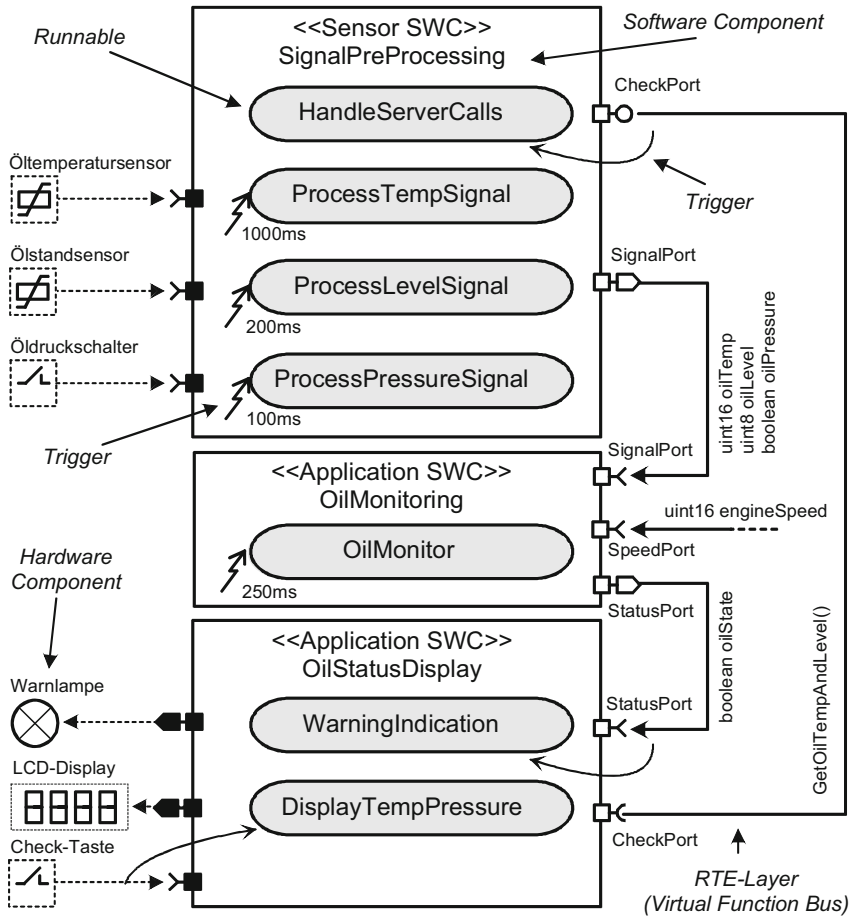

Abb. 8.24 AUTOSAR-Modell der Ölkreislauf-Überwachung

einer Ausnahme verwenden die Ports das *Sender-Receiver*-Paradigma. Dabei kann die *Data-Distribution-Semantik* eingesetzt werden, d. h. es wird jeweils der letzte verfügbare Wert benutzt, Warteschlangen (*Queues*) sind nicht notwendig. Die Abfrage des Ölstands und der Öltemperatur für die Anzeige dagegen verwendet das *Client-Server*-Konzept, weil sie nur bei Druck auf die Check-Taste aktiv werden soll. Da die Sensorwerte jederzeit verfügbar sind, arbeitet die *Client-Server*-Schnittstelle synchron, d. h. die Abfrage `GetOilTempAndLevel()` liefert sofort das Ergebnis.

Die Implementierung in den Softwarekomponenten erfolgt über *Runnables*. Durch die eigenständig ablaufenden Softwarefunktionen können die periodisch erfolgende Verarbeitung der verschiedenen Sensorsignale und die Überwachung mit unterschiedlichen Abtastzeiten erfolgen. Intern werden hierzu *RTE Timing Events* verwendet. Die Ansteue-

Tab. 8.5 Ausschnitt aus dem Programmcode für die Funktion OilMonitor

```
void OilMonitor(void)        // Wird periodisch alle 250ms aktiviert
{  uint16 temperature;
   uint16 speed;
   boolean status = false;
   . . .
   RTE_READ_SignalPort_oilTemp(&temperature);   // Signale einlesen
   RTE_READ_SpeedPort_engineSpeed(&speed);
   . . .
   if (temperature > ... && temperature < ... && ... )
   {  status = true;                            // Überwachungslogik
   }
   . . .
   RTE_WRITE_StatusPort_oilState(status);       // Ergebnis ausgeben
}
```

rung der Warnlampe dagegen erfolgt, sobald ein neuer `oilState`-Wert bereit steht. Intern dient dazu ein *RTE Data Received Event* als Triggerereignis für das *Runnable* `WarningIndication`.

Tabelle 8.5 zeigt einen Ausschnitt aus dem C-Programm für das Runnable `OilMonitor`. Die `RTE_READ`- und `RTE_WRITE`-Funktionen, mit denen auf die Eingangs- und Ausgangssignale zugegriffen wird, können von einem AUTOSAR-Werkzeug direkt aus der Beschreibung der Ports des Modells in Abb. 8.24 generiert werden. Ähnliche Funktionen würden auch für die auf der linken Seite des Bildes dargestellten Hardwareschnittstellen erzeugt. Letztlich wird auf die elektrischen Ein- und Ausgangssignale durch die Hardwaretreiber der AUTOSAR-Basissoftware zugegriffen, z. B. über das DIO- oder das ADC-Modul. Die AUTOSAR-Methodik erlaubt aber keinen direkten Aufruf der Basissoftware durch die Anwendungssoftware. Daher muss die Basissoftware so konfiguriert werden, dass der *I/O Hardware Abstraction Layer* die notwendigen Signale der *Hardware-Ports* für den RTE-Layer anbietet (Abb. 8.3).

8.8 Ausblick

Damit Anwendungskomponenten wieder verwendbar oder einfach austauschbar werden, müssen auch die Schnittstellen der Komponenten inhaltlich standardisiert werden. Vergleicht man die derzeitige Spezifikation mit derjenigen von Bussystemen, so ist allenfalls der „*Data Link und Network Layer*“ spezifiziert, während der „*Application (Interface) Layer*“ noch weitgehend offen ist. Entsprechende Arbeitspakete wurden für die Bereiche

Antriebsstrang (*Powertrain*), Fahrwerk (*Chassis*), Karosserie- und Komfortelektronik (*Body and Comfort*) sowie Insassensicherheit (*Occupant and Pedestrian Safety*) definiert. Dort werden die Schnittstellen für die wichtigsten Komponenten zumindest grob festgelegt und mit jeder AUTOSAR Version verfeinert.

Zukünftige Steuergeräte werden sicher nicht schlagartig auf die AUTOSAR-Architektur umgestellt. Der schrittweise Umstellungsprozess von einer rein proprietären Implementierung auf eine vollständig AUTOSAR-konforme Lösung soll durch die drei sogenannten *Implementation Conformance Classes* ICC unterstützt werden:

- ICC 1-Steuergeräte führen lediglich das zentrale *Run Time Environment RTE* ein. Die Schichten oberhalb und unterhalb des RTE verwenden zunächst weiterhin bewährte Module. ICC1 konzentriert sich auf die schnelle Einführung von AUTOSAR bei neuen Funktionen auf der Anwendungsebene. Bei den proprietären Komponenten werden Schnittstellen zum RTE so nachgerüstet, dass neue Anwendungskomponenten nach AUTOSAR-Standard integriert werden können.
- ICC 2-Geräte erlauben zusätzlich eine schrittweise Migration der Basissoftware. Dazu wird die Basissoftware in die vertikalen Säulen Betriebssystem, Kommunikationsstack und Hardwaretreiber untergliedert. Die Schnittstellen zwischen diesen mehr oder weniger monolithischen Blöcken und zum RTE entsprechen bereits dem AUTOSAR-Standard, deren interne Feinstruktur dagegen noch nicht.
- ICC 3 ist vollständig AUTOSAR-konform und implementiert die feingliedrigere AUTOSAR-Struktur mit allen geforderten Schnittstellen.

Zukünftige Projekte werden zeigen, ob das Konfigurieren von Komponenten, deren innere Semantik wegen der enorm vielfältigen Funktionalität nur schwer in kompakter Form vollständig beschrieben werden kann, und die Pflege der Konfigurations- und Generierungswerkzeuge tatsächlich weniger Aufwand verursacht und dabei wirklich weniger Fehler entstehen als bei der klassischen Softwareentwicklung. Zusätzliche Schwierigkeiten werden sich dadurch ergeben, dass AUTOSAR Anwendungsbereiche wie das Infotainment-Segment aber auch die Fahrerschnittstelle mit den Anzeigesystemen im Armaturenbrett derzeit noch rudimentär abdeckt und die Integration „älterer" Datenformate wie ASAM ODX, FIBEX, CanDB oder LDF schwierig bleibt. AUTOSAR verweist darauf, dass die Werkzeuge eben entsprechende Import- bzw. Export-Schnittstellen bereitstellen müssten. Ob die Konfigurationsdatenformate auf beiden Seiten der Schnittstellen die notwendigen semantischen Inhalte haben, um eine konsistente Konversion in beiden Richtungen zu gewährleisten, wie sie beim *Round Trip Engineering* in der praktischen Entwicklungsarbeit notwendig ist, wird sich zeigen.

Aufgrund der vielen Optionen und umfangreichen Konfigurationsmöglichkeiten können AUTOSAR-Konzepte bei unbedachtem Einsatz zu höherem Speicherverbrauch und Rechenleistungsbedarf führen als konventionelle Lösungen. Daher gibt es, unter anderem im HIS-Arbeitskreis, Versuche, eine weniger komplexe Teilmenge von AUTOSAR zu definieren und die Konfigurationsoptionen einzuschränken.

Die vollständige Umstellung auf durchgängig AUTOSAR-konforme Lösungen auf Anwendungsebene wird noch einige Jahre dauern. Schnell erfolgreich ist AUTOSAR vor allem in dem Bereich, den OSEK/VDX nur bruchstückhaft abgedeckt hat. Nämlich die Bereitstellung einer Hardware-Abstraktions-Schicht, eines einheitlichen, verschiedene Bussysteme integrierenden Kommunikationsstapels sowie allgemeiner Betriebssystemdienste. Insoweit verspricht die AUTOSAR Basissoftware, wenn man den Infotainment-Bereich einmal ausklammert, endlich *das* einheitliche Betriebssystem für komplexe Kfz-Steuergeräte zu sein. Vielleicht kann sich die Entwicklung von Fahrzeugsystemen nun wirklich auf die kundenerlebbaren Funktionen konzentrieren.

8.9 Normen und Standards zu Kapitel 8

AUTOSAR	AUTOSAR Press Information Pack – Information Pack, VDI Konferenz Electronic Systems for Vehicles, Baden-Baden, 2005, www.autosar.org AUTOSAR Technical Overview, Version 2.2.1 – Part of Release 3.0, 2008 www.autosar.org AUTOSAR Layered Software Architecture, Version 2.2.1 – Part of Release 3.0, 2008, www.autosar.org AUTOSAR Layered Software Architecture, Version 3.0.0 – Part of Release 4.0, 2009, www.autosar.org AUTOSAR Layered Software Architecture, Version 3.3.0 – Part of Release 4.1, 2013, www.autosar.org AUTOSAR Specifications, www.autosar.org – Sammlung von mehr als 180 Dokumenten (Release 4.1, 2013) mit jeweils eigener Versionierung, unterteilt in die Hauptgruppen Main (Overview) Software Architecture (General, Communication Stack, Diagnostic Services, System Services, Memory Stack, Peripherals, Implementation and Integration, Runtime Environment) Methodology and Tools Conformance Testing Application Interfaces 2011 wurde das Update 3.2 für Release 3.x und 2013 das Update 4.1 für Release 4.x veröffentlicht. HIS Recommendation for a scalable AUTOSAR stack. Version 1.4, 2009, www.automotive-his.de
Safety	ISO 26262-1 bis -10 Road vehicles – Functional safety, 2009, www.iso.org IEC 61508-0 bis -7 Functional safety of electrical/electronic/programmable electronic safety-related systems, 1998 bis 2000, www.iec.ch

Internet Protokolle	IPv4 Internet Protocol Version 4, RFC 791 IPv6 Internet Protocol Version 6, RFC 2460 Requirements for Internet Hosts – Communication Layers RFC 1122 Transmission of IP Datagrams over Ethernet Networks RFC 894 UDP User Datagram Protocol, RFC 768 TCP Transmission Control Protocol, RFC 793 ARP Address Resolution Protocol, RFC 826 Neighbor Discovery Protocol, RFC 4861 DHCP Dynamic Host Configuration Protocol, RFC 2131 AutoIP Dynamic Configuration of Link-Local Addresses, RFC 3927 ICMP Internet Control Message Protocol, RFC 792 DNS Domain Name System RFC 1034 und RFC 1035 Internet Engineering Task Force IETF, www.ietf.org

Literatur

[1] O. Kindel, M. Friedrich: Softwareentwicklung mit AUTOSAR. dpunkt Verlag, 1. Auflage, 2009

[2] R. Stevens: TCP/IP Illustrated. Addison-Wesley, 3 Bände, 2002

[3] E. Hall: Internet Core Protocols. O'Reilly, 2000