2 Echtzeitbetriebssysteme

In den letzten Jahren hat sich die Automobilindustrie zu einem der wesentlichen Anwender von Echtzeitbetriebssystemen für eingebettete Systeme entwickelt. Relativ zeitig erkannten sowohl die Automobilhersteller als auch deren Zulieferer die Notwendigkeit, neben Hardwarestandards auch Standards für die verwendete Software festzulegen. Dies umfasst auch die Standardisierung der verwendeten Betriebssysteme. Dabei ist dem Zeitverhalten der Betriebssysteme die gebührende Aufmerksamkeit zu schenken.

2.1 Allgemeines zu Echtzeitbetriebssystemen

Betriebssysteme stellen die Basis eines jeden Rechnersystems dar. Dabei besteht ein Betriebssystem aus einer Sammlung von Programmen, welche die Betriebsmittel verwalten und die Ausführung von anderen Programmen überwachen und steuern. Speziell im Bereich der eingebetteten Systeme sind vielfach Echtzeitbetriebssysteme zu finden, die die zentrale Grundlage eines Rechnersystems innerhalb eines Echtzeitsystems darstellen. Echtzeitbetriebssysteme ermöglichen es, dem Gesamtsystem ein *deterministisches Laufzeitverhalten* zu geben. Das bedeutet, das Ergebnis muss nicht nur korrekt sein, sondern auch zu einem vorgegebenen Zeitpunkt bereitgestellt werden. Echtzeitsysteme müssen nicht besonders schnell sein, sondern nur „schnell genug“ und deterministisch.

2.1.1 Grundlegende Begriffe

Zunächst gilt es, einige verwendete Begriffe zu erklären:

Unter einem *Betriebssystem* versteht man diejenigen Programme eines digitalen Rechnersystems, die zusammen mit den Eigenschaften der Rechenanlage die Grundlage der möglichen Betriebsarten des digitalen Rechnersystems bilden und insbesondere die Abwicklung von anderen Programmen steuern und überwachen [Di5]. Im Umfeld eingebetteter Systeme stellt sich ein Betriebssystem als einfacher Verwalter von Systemressourcen (im einfachsten Fall der Resource Prozessor) dar.

Der *Echtzeitbetrieb* ist der Betrieb eines Rechnersystems, bei dem Programme zur Verarbeitung anfallender Daten ständig derart betriebsbereit sind, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu vorbestimmten Zeitpunkten anfallen [Di5].

Ein *Echtzeitsystem* realisiert den Echtzeitbetrieb unter den oben angegebenen Bedingungen. Die Entwicklung eines Echtzeitsystems erfolgt entweder ereignisgesteuert oder zeitgesteuert.

Unter einem *Prozess* verstehen wir den kleinsten, nicht mehr teilbaren Aufgabeninhalt, den ein Mikroprozessor ausführen kann. Er ist ein Kompositum aus einer sequenziell auszuführenden Berechnungsvorschrift (sequenzieller Programmcode) und dem zugehörigen Datenraum sowie sämtlichen Informationen zum Prozesszustand (Programmzähler und bestimmte Register des Prozessors).

Ein *Prozessor* ist eine „Ausführungsstation“, die zu einem gegebenen Zeitpunkt maximal einen Prozess auszuführen vermag.

© Springer Fachmedien Wiesbaden 2014
K. Reif, *Automobilelektronik*, ATZ/MTZ-Fachbuch, DOI 10.1007/978-3-658-05048-1_2

Der Übergang von der Ausführung eines Prozesses zur Ausführung eines anderen Prozesses wird als *Kontextwechsel* bezeichnet.

Weiterhin unterscheidet man zwischen Prozessen und *Threads*. Hier dienen als Unterscheidungsmerkmale die Separierung des Adressraumes und die damit höhere Komplexität des Verwaltungsaufwandes für Prozesse. Threads teilen sich einen gemeinsamen Adressraum. Der im Folgenden verwendete Begriff *Task* ist ein Oberbegriff über die Begriffe Prozess und Thread.

Wenn man die in eingebetteten Systemen anzutreffenden Betriebssysteme mit Standardbetriebssystemen – beispielsweise UNIX, Linux, Windows – vergleicht, stellt man fest, dass erstere meist deutlich kleiner sind – sowohl bezüglich des Codevolumens als auch bezüglich des Funktionsumfanges. Dies ermöglicht eine kosteneffiziente Integration in eingebettete Systeme. Weiterhin enthalten sie spezielle Mechanismen zur Herbeiführung eines verlässlichen Zeitverhaltens der einzelnen Prozesse. Betriebssysteme, die die letztgenannte Eigenschaft erfüllen, bezeichnet man auch als *Echtzeitbetriebssysteme* (Real-Time Operating System RTOS). Daneben existieren auch Betriebssysteme für eingebettete Systeme, die keine Echtzeiteigenschaften besitzen.

2.1.2 Echtzeitbegriffe

In Bild 2-1 werden einige wichtige Zeitbegriffe im Zusammenhang mit Echtzeitsystemen dargestellt. Unter *Echtzeitanforderungen* werden die zeitlichen Festlegungen verstanden, die das Zeitverhalten des Tasks bestimmen, das für eine Steuerung oder Regelung in Echtzeit notwendig ist. Dabei wird die Zeitspanne von der Aktivierung des Tasks bis zu dem Zeitpunkt, zu dem der Task spätestens abgeschlossen sein muss (*absolute Deadline*, häufig auch nur Deadline genannt), als *relative Deadline* bezeichnet. Die *Response-Zeit* beschreibt den Zeitabschnitt von der Aktivierung bis zum tatsächlichen Ende der Abarbeitung und schließt damit die *Ausführungszeit*, die mit dem tatsächlichen Start des Tasks beginnt, mit ein.

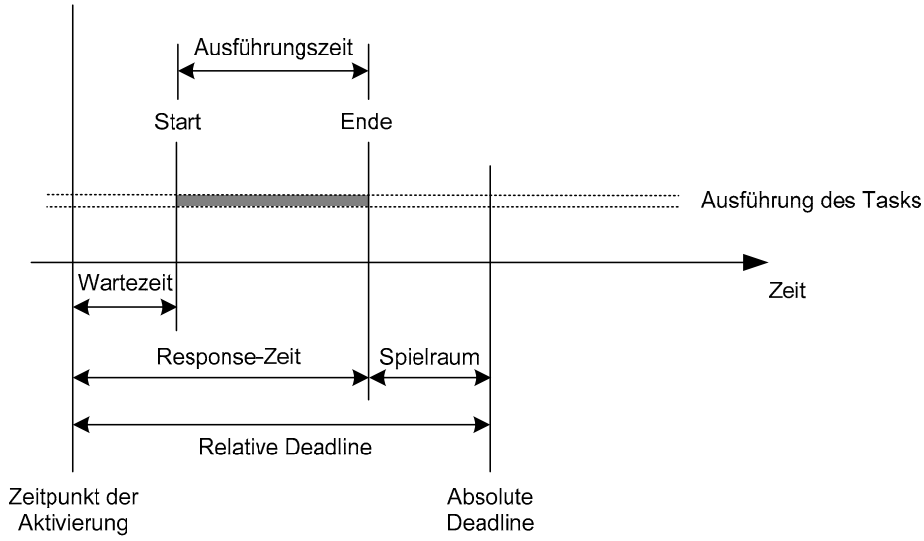

Bild 2-1 Parameter zur Festlegung von Echtzeitanforderungen (nach [Do1], [Li2])

Zeitschranken

Für Anforderungen an Echtzeitsysteme hinsichtlich ihrer Fähigkeit, eine Berechnung bis zu einem festgelegten Zeitpunkt (Deadline) zu erledigen, existieren unterschiedliche Definitionen. Die folgende Definition lehnt sich an die Begriffe aus [Do1], [Pr2] und [Wö1] an.

Unter einer *harten Deadline* versteht man dabei eine zeitliche Vorgabe, welche unbedingt eingehalten werden muss. Ein verspätetes Liefern eines erwarteten Ergebnisses führt zu einem Fehlverhalten des Systems. Im Sinne einer Kostenfunktion bedeutet dies, dass das nicht rechtzeitige Liefern des erwarteten Ereignisses zu einem negativen Nutzen, also zu einem Schaden führt (Bild 2-2a). Harte Echtzeitanforderungen stellen beispielsweise Bremssysteme sowie die Ansteuerung von Verbrennungsmotoren.

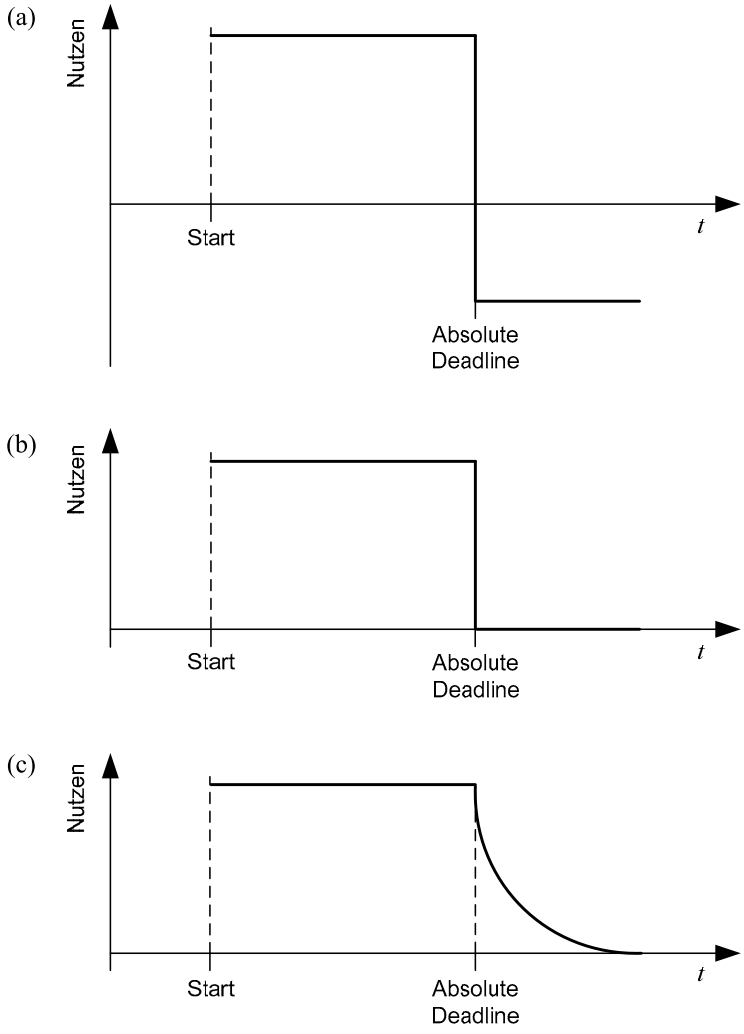

Bild 2-2 Nutzenfunktionen von Echtzeitbetriebssystemen als Funktion der Bereitstellungszeit *t*:
(a) Harte Deadline.
(b) Feste Deadline.
(c) Weiche Deadline

Im Gegensatz dazu wird bei einer *festen Deadline* und bei einer *weichen Deadline* die Nutzenfunktion nie negativ. Das Verhalten der Nutzenfunktion unterscheidet sich jedoch (siehe Bild 2-2b, c). Ein Beispiel für ein System mit einer festen Deadline, d. h. festen Echtzeitanforderungen, ist die Geschwindigkeitsregelung. Ein Beispielsystem mit einer weichen Deadline, d. h. weichen Echtzeitanforderungen, ist das Multimediasystem im Fahrzeug.

Echtzeitbetrieb

Echtzeitsysteme ändern ihren Zustand als Funktion der Zeit. Dabei wird der Zustand des Gesamtsystems durch eine Anzahl von Zustandsvariablen beschrieben. Ein solcher Zustand kann Größen wie z. B. die Geschwindigkeit eines Fahrzeuges, den aktuellen Gang oder die aktuelle Einspritzmenge in einem Verbrennungsmotor umfassen. Das Rechnersystem muss nun ein internes Abbild des Zustandes ermitteln oder erstellen. Dabei ist zu beachten, dass ein solches Abbild nur für eine begrenzte Zeit gültig ist. Weiterhin muss berücksichtigt werden, dass zu jeder Zustandsänderung rechtzeitig ein neues Abbild erstellt werden muss, um ebenfalls rechtzeitig auf diese Änderung reagieren zu können.

Echtzeitarchitektur

Zur Ermittlung des Systemzustandes und zur Berechnung der Reaktion bestehen zwei grundlegende Vorgehensweisen, die zu unterschiedlichen Architekturen von Echtzeitsystemen führen. Man unterscheidet nach [Ko2] *ereignisgesteuerte* (event-triggered) und *zeitgesteuerte* (time-triggered) Systeme.

Bei einem zeitgesteuerten System erfolgt die Aktualisierung des Zustandsabbildes periodisch in festgelegter Abfolge. Als Zeitgeber kann eine interne Uhr dienen, in verteilten Systemen kann als Zeitgeber – über eine Uhrensynchronisation – die globale Zeitbasis genutzt werden. Im Gegensatz dazu wird bei einem ereignisgesteuerten System das Abbild mit jeder Zustandsänderung aktualisiert.

Die Festlegung, welche Architektur zu benutzen ist, ist oft nicht einfach zu treffen. Vereinfacht kann man sagen, dass eine Abwägung zwischen Vorhersagbarkeit und Flexibilität zu treffen ist. Weiterhin kann das verwendete Bussystem einen Einfluss auf die Entscheidung zwischen einem ereignisgesteuerten oder einem zeitgesteuerten System haben.

Rechtzeitigkeit und Determinismus von Echtzeitsystemen

Echtzeitrechnersysteme sind nicht grundsätzlich schnelle Systeme. Zunächst geht es in diesem Zusammenhang um die rechtzeitige Reaktion auf Umgebungsereignisse. Im Kontext der Betriebssysteme versteht man unter der *Latenz* die Zeitspanne zwischen dem Auftreten eines Ereignisses und der entsprechenden Reaktion, z. B. der entsprechenden Ausgabe an den Aktor [Ko2].

Für manche Anwendungen genügt es nicht, dass das Ergebnis einer Berechnung rechtzeitig vorliegt. Vielmehr muss sichergestellt werden, dass der Zeitpunkt der Bereitstellung zusätzlich einen geringen *Jitter* (Differenz zwischen maximaler und minimaler Latenz) aufweist. Dieses Problem tritt besonders bei regelungstechnischen Problemstellungen auf.

Zur Sicherstellung des Determinismus von Echtzeitsystemen benötigt man die Kenntnis der *maximalen Ausführungszeit* (Worst Case Execution Time WCET). Dies ist die maximale Ausführungszeit eines bestimmten Programmteils auf einer bestimmten Ausführungsplattform. Dabei sind alle möglichen Verzögerungen (Interrupts, Cacheverhalten etc.) zu berücksichtigen, die auftreten können.

2.1.3 Prozess und Prozesszustände

Eines der grundlegenden Konzepte in der Theorie der Betriebssysteme ist der Prozessbegriff. Prozesse stellen das grundlegende Beschreibungsmittel von Nebenläufigkeiten dar. Ein Prozess ist ein laufendes Programm zusammen mit den zugehörigen Betriebsmitteln sowie den dazugehörenden aktuellen Werten des Programmzählers, der Register und der Variablen. Dabei kann ein Prozess mehrere grundlegende Zustände annehmen. Im Folgenden wird von vier grundlegenden Prozesszuständen ausgegangen (siehe Bild 2-3): Beim Systemstart ist der Zustand jedes Prozesses „suspended“. Durch die Aktivierung eines Prozesses wechselt dieser in den Zustand „ready“. Damit ist dem Scheduler dieser Prozess als ablauffähig bekannt gemacht. Bei tatsächlicher Ausführung geht der Prozess in den Zustand „running“ über. Dieser Zustand kann entweder durch Beendigung des Tasks (terminate) oder Verdrängung (preempt) verlassen werden. Letzteres geschieht zum Beispiel, wenn ein höher priorisierter Task den gerade aktuell ausgeführten Task zwangsweise unterbricht. Nun nutzt dieser Prozess den Prozessor und kann auf Betriebsmittel zugreifen. Beim Warten auf ein Ereignis wechselt der Prozess nach „waiting“, und ihm wird vom Scheduler der Prozessor entzogen. Dieser Zustand wird in manchen Systemen als „blocked“ bezeichnet. Nach der Abarbeitung aller Anweisungen beendet sich der Prozess selbständig und gibt die ihm zugeteilten Betriebsmittel zurück.

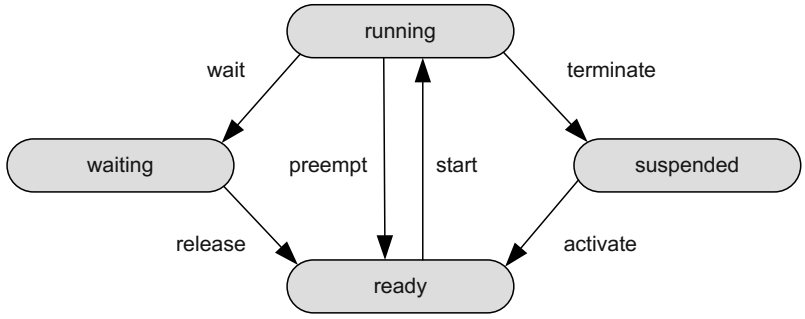

Bild 2-3 Grundlegende Prozesszustände und Übergänge (angelehnt an [Os1], daher wurden englische Begriffe gewählt)

2.1.4 Kontextwechsel

Um einen Prozess in den Zustand „running“ zu versetzen, muss ein anderer Prozess verdrängt werden (Kontextwechsel). Damit dieser später – durch einen weiteren Kontextwechsel – fortgesetzt werden kann, müssen bestimmte Informationen durch das Betriebssystem gesichert werden. Dies umfasst mindestens den Programmcode, den Datenbereich und einen so genannten Task-Control-Block (TCB).

In den Task-Control-Block werden zum Kontextwechsel alle für die Prozessausführung relevanten Registerinhalte des Prozessors gesichert. Wenn der Prozessor einem Prozess wieder zugeteilt wird, werden die Registerinhalte durch Auslesen des dem Prozess zugehörigen Task-Control-Blocks wiederhergestellt. Viele Prozessoren enthalten für diesen Vorgang optimierte Hardware, damit nicht mehrere Register gesichert und geladen werden müssen, sondern nur zwischen verschiedenen Registerbänken umgeschaltet werden muss.

2.1.5 Scheduling

Der Dispatcher führt die im vorangegangenen Abschnitt dargelegten Zustandsübergänge durch. Bis auf den Zustand „running“ besitzt jeder Prozesszustand mindestens eine Warteschlange im Dispatcher. Die Zuordnung einer Ausführungsreihenfolge zu einem gegebenen Prozesssystem und einer gegebenen Aktivierungssequenz wird als Scheduling bezeichnet. Diese Festlegung wird von einem Scheduler berechnet. Als Resultat erhält man einen Schedule, welcher eine derartige Ausführungsreihenfolge darstellt. Umgangssprachlich fasst man den Dispatcher und den Scheduler zusammen und spricht nur vom Scheduler. Der vereinfachte Ablauf, um die Zustandswechsel der Tasks zu steuern, ist in Bild 2-4 beispielhaft dargestellt. Es zeigt die in diesem Zusammenhang relevanten Funktionalitäten eines Echtzeitbetriebssystems.

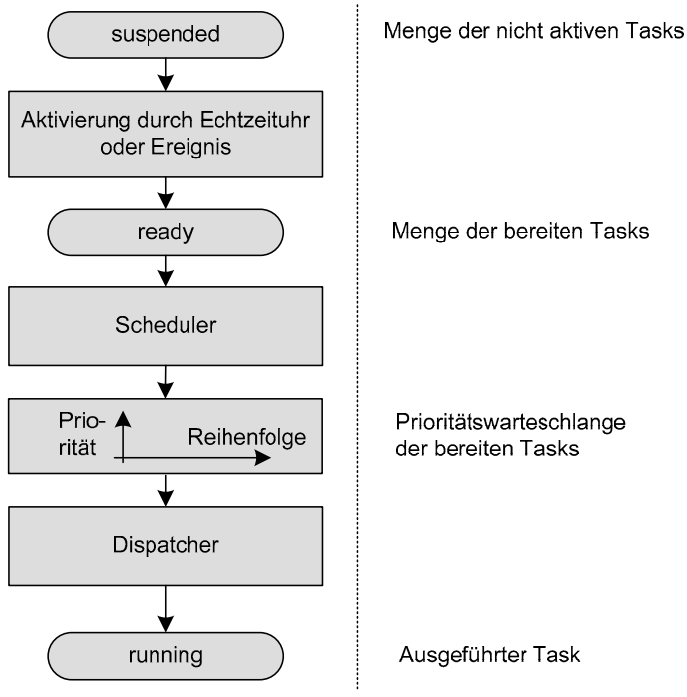

Bild 2-4 Beispiel eines vereinfachten Ablaufs innerhalb eines Echtzeitbetriebssystems (nach [Sc1])

Das Scheduling für Echtzeitsysteme kann sowohl statisch als auch dynamisch erfolgen. Bild 2-5 systematisiert die prinzipiellen Schedulingansätze für Echtzeitbetriebssysteme. Beim *statischen Scheduling* erfolgt die Berechnung der Schedule vorab, d. h. nicht während der Laufzeit. Damit erhält man ein absolut deterministisches Laufzeitverhalten, welches aber nicht

mehr änderbar ist. Im Gegensatz dazu wird bei einem *dynamischen Scheduling* die Berechnung während der Laufzeit durchgeführt, was einen erhöhten Aufwand erfordert. Schedules, die teilausgeführte Prozesse jederzeit zu Gunsten anderer, höher priorisierter Prozesse (durch einen Kontextwechsel) unterbrechen, heißen *präemptiv*.

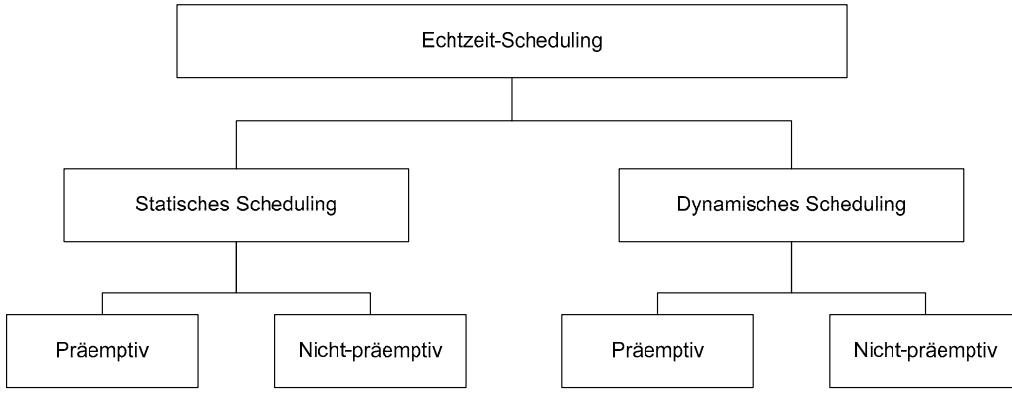

Bild 2-5 Klassifikation von Scheduling-Algorithmen (statische Prioritäten, nach [Ko2])

Wenn alle Zeitbedingungen des Prozesssystems erfüllt sind, also keine Deadline überschritten wird, bezeichnet man diesen Schedule als *zulässig* (feasible). Als *schedulebar* bezeichnet man Prozesssysteme, die für jede bezüglich der Prozessparameter mögliche Aktivierungssequenz einen zulässigen Schedule besitzen.

Es existieren verschiedene Strategien, nach denen das Task-Scheduling erfolgen kann. Beispiele für die verwendeten Strategien sind z. B. das Round-Robin-Verfahren, bei dem allen Prozessen nacheinander für jeweils einen kurzen Zeitraum Zugang zu den benötigten Ressourcen gewährt wird. Beim Least-Laxity-First-Ansatz wählt der Scheduler diejenigen Prozesse aus, die den geringsten Spielraum (Laxity, vgl. Bild 2-1) haben. Ein Überblick über verschiedene Scheduling-Algorithmen ist in [Ho3] zu finden.

2.1.6 Vertreter von Echtzeitbetriebssystemen

Die Anzahl der Standard-Echtzeitbetriebssysteme ist sehr groß. Ursache dafür ist vor allem die Notwendigkeit einer zugeschnittenen Speziallösung für den jeweiligen Einsatzbereich. Zu den Vertretern von Echtzeitbetriebsystemen gehören VxWorks, QNX, OSEK/VDX, LynxOS und RTLinux [Pr2]. Die Auswahl des einzusetzenden Betriebssystems richtet sich nach unterschiedlichen Gesichtspunkten. Dazu zählen z. B. Kostenaspekte, die Toolkettenintegration und die verwendete Hardware. OSEK/VDX ist im Automobilbereich das Standardbetriebssystem und wird im Folgenden näher erläutert.

2.2 OSEK/VDX

Jede Innovation im Fahrzeugbereich ist direkt oder indirekt mit elektronischen Steuerungen verbunden. Auf Grund der steigenden Anzahl von elektronischen Systemen erhöht sich auch die Anzahl der beteiligten Entwicklungspartner. Das Projekt OSEK/VDX (Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug) wurde 1993 ins Leben gerufen, um im Bereich der eingebetteten Betriebssysteme für Kraftfahrzeuge eine Standardisierung zu erreichen.

2.2.1 Historie

Als Gründungsmitglieder der OSEK-Organisation waren BMW, Bosch, DaimlerChrysler, Opel, Siemens, VW und die Universität Karlsruhe aktiv tätig. Die französischen Hersteller PSA und Renault schlossen sich im Jahre 1994 an und brachten VDX (Vehicle Distributed Executive) mit ein. Die zusammengeführten Spezifikationen wurden 1995 erstmals veröffentlicht. Die OSEK/VDX-Organisation steht unter der Führung der oben genannten Firmen. Momentan sind mehr als 50 Partner aus dem Umfeld der Automobilelektronik an der Weiterentwicklung der Standards beteiligt.

Die OSEK/VDX-Organisation hat mehrere Standards veröffentlicht. Tabelle 2.1 listet diese auf. Teile der OSEK/VDX-Spezifikationen wurden in die ISO 17356 [Is9] überführt. Diese umfasst OSEK-OS 2.2.1, OSEK-COM 3.0.2, OSEK-NM 2.5.2 und OSEK-OIL 2.4.1. Die Arbeiten der OSEK/VDX-Gremien finden ihre Fortsetzung im 2003 gestarteten AUTOSAR-Projekt (siehe Abschnitt 2.3).

Tabelle 2.1 Übersicht über die OSEK/VDX-Standards

Standard	Version
Betriebssystem (OSEK-OS)	2.2.3
Kommunikation (OSEK-COM)	3.0.3
Netzmanagement (OSEK-NM)	2.5.3
OSEK Implementation Language (OIL)	2.5
Time Triggered OS (OSEKtime)	1.0
Fault Tolerant Communication (FTCOM)	1.0
OSEK Runtime Interface (ORTI)	2.2

2.2.2 Grundlegende Eigenschaften von OSEK-Betriebssystemen

OSEK-OS umfasst die herstellerübergreifende Spezifikation für ein Echtzeitbetriebssystem sowie deren Softwareschnittstellen und Funktionen für Kommunikations- und Netzwerkmanagement (siehe Abschnitt 2.2.8 und 2.2.9). Damit ergibt sich der in Bild 2-6 dargestellte Grundaufbau.

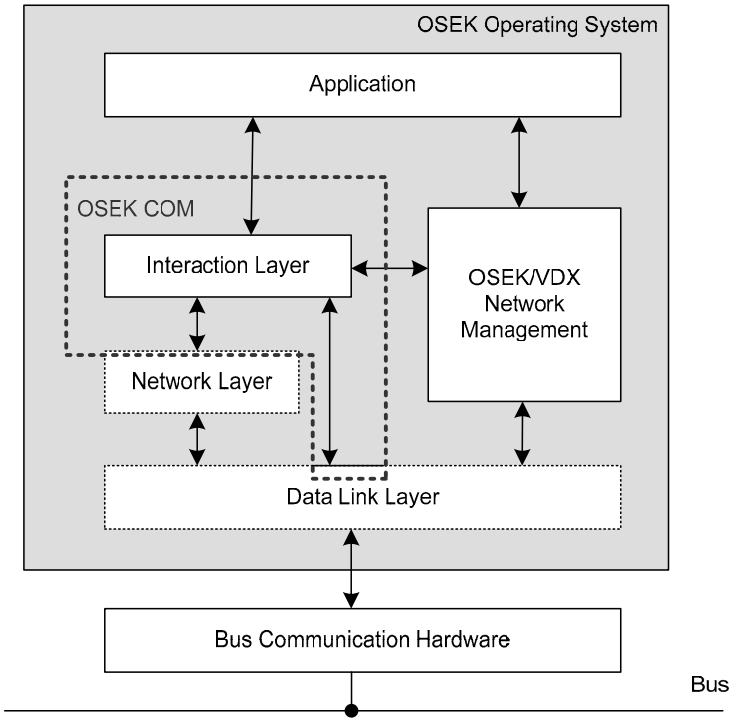

Bild 2-6 Grundaufbau eines OSEK/VDX-Systems (es wurden die englischen Begriffe verwendet, um konform mit der Spezifikation zu sein)

Den speziellen Bedingungen in der Automobilbranche wie Verlässlichkeit, Echtzeitfähigkeit und Kostensensitivität wird durch folgende Eigenschaften Rechnung getragen: Das OSEK/VDX-Betriebssystem ist statisch konfigurier- und skalierbar. Die Anzahl der Tasks, die Ressourcen und die benötigten Funktionen sind statisch. Das heißt, die Konfiguration wird vor der Laufzeit festgelegt und ist dynamisch nicht änderbar. Nur auf diese Weise kann ein geringer Speicherbedarf realisiert werden. Die OSEK/VDX-Spezifikation erlaubt es weiterhin, dass Programme direkt im ROM ausgeführt werden können. Anwendungssoftware kann zwischen verschiedenen Betriebssystemen, welche gemäß des OSEK/VDX-Standards implementiert wurden, ausgetauscht werden. Schließlich ermöglicht die Spezifikation ein vorhersagbares und dokumentiertes Verhalten der Betriebssystemimplementierung, das für die harten Echtzeitbedingungen in der Automobilindustrie ausreichend ist.

Das OSEK/VDX-konforme Betriebssystem ist ein Ein-Prozessor-Betriebssystem für verteilte eingebettete Systeme. Es stellt eine einheitliche Systemumgebung zur Ausführung von automobilspezifischen Anwendungen zur Verfügung. Dabei ist zu beachten, dass streng gesprochen OSEK-OS kein eigenes Betriebssystem darstellt, sondern lediglich eine Spezifikation, nach der der Softwarehersteller OSEK-konforme Betriebssysteme erstellen kann.

2.2.3 Betriebsmittel

Ein OSEK-OS-konformes Betriebssystem stellt die grundlegenden Betriebsmittel zur Verfügung. Dazu gehören als zentrale Elemente die Tasks als Prozessäquivalent. Es sind Mechanismen zur Synchronisation und Signalisierung zwischen den Tasks und ein Konzept zur Realisierung zeitabhängiger Dienste vorhanden. Weiterhin existiert eine Grundfunktionalität für den Austausch von Nachrichten und die Unterstützung bei der Fehlerbehandlung.

Task

Das grundlegende Zustandsverhalten von OSEK-Tasks ist bereits in Abschnitt 2.1.3 behandelt worden. Hierbei ist der Begriff des Prozesses durch den Begriff des Tasks zu ersetzen. Jeder Task besitzt eine bestimmte *Priorität*, die statisch vergeben wird. Das bedeutet, sie kann während der Ausführung nicht dynamisch verändert werden. Die Priorität legt fest, wie wichtig die Ausführung der Tasks bzw. die in dem Task enthaltenen Programmschritte sind.

OSEK/VDX schreibt zur Definition von Tasks eine C-Erweiterung vor. Diese wird über Makros realisiert. Ein Task verhält sich wie eine Funktion, wobei eine Übergabe von Parametern nicht möglich ist, jedoch gelten die gleichen Sichtbarkeitsregeln (nach C-Standard). Das folgende Codebeispiel veranschaulicht die Definition von Tasks und stellt gleichzeitig die kürzest mögliche OSEK/VDX-Anwendung dar:

```
DeclareTask(RteTaskSensor); /* Task declaration */
....
TASK(RteTaskSensor)
{
   /* do something */
   TerminateTask(); /* finish the task */
}
```

Interruptverwaltung

Externe und interne Ereignisse innerhalb einer Recheneinheit erzeugen eine Unterbrechungsanforderung (Interrupt Requests IRQ), durch welche Interrupt-Service-Routinen (ISR) angestoßen werden. Damit besteht die Möglichkeit, einen festen Ablauf zu unterbrechen. Unter OSEK/VDX wird durch Interrupt-Service-Routinen jeder Task unterbrochen. Grundsätzlich sollten Interrupt-Service-Routinen klein und schnell abzuarbeiten sein, da sie meist bis zu ihrem Ende durchlaufen werden. Anhand der Nutzung von Betriebssystemfunktionen können Interrupt-Service-Routinen innerhalb von OSEK/VDX in zwei Kategorien eingeteilt werden.

Die Interrupt-Service-Routinen der Kategorie 1 besitzen den geringsten Overhead, da diese Interrupt-Service-Routinen keinen Einfluss auf die Verwaltung von Tasks haben und keine Nutzung von Betriebssystemfunktionen erfolgt.

Innerhalb von Interrupt-Service-Routinen der Kategorie 2 ist die Nutzung von Betriebssystemfunktionen möglich. Das Betriebssystem muss einen Funktionsrahmen zur Verfügung stellen. Damit bekommt das Betriebssystem die Möglichkeit, vor der Ausführung der Routine einige Aktionen durchzuführen. Weiterhin findet ein eventuell erforderliches Scheduling niemals in der Interrupt-Service-Routine statt, sondern erst bei deren Beendigung. Im OSEK/VDX-Be-

triebssystem existieren Möglichkeiten, einzelne Unterbrechungsanforderungen oder Gruppen von Unterbrechungsanforderungen ab- und anzuschalten.

Ähnlich wie für Tasks erfordert die Definition von Interrupt-Service-Routinen eine C-Sprach-erweiterung:

```
ISR(I_CAN_CTRL0_TX)
{
   /* place user code here */
   Can_IsrSrn4TransmitHandler0();
}
```

Eventsteuerung

Events dienen zur Steuerung des Programmflusses und werden im OSEK-OS zur Signalisierung zwischen Tasks und Interrupt-Service-Routinen benutzt. Sie können von beliebigen Tasks und Interrupt-Service-Routinen der Kategorie 2 ausgelöst werden. Events dienen weiterhin zum Aufbau von ereignisgesteuerten Diensten und Client-Server-Beziehungen zwischen Tasks. Manche Tasks, nämlich die so genannten *Extended Tasks,* können ohne CPU-Belastung warten (vgl. Zustand „waiting“ in Bild 2-3), wobei ihnen bestimmte Ereignisse durch Events signalisiert werden, die zu einem Wechsel aus dem Zustand „waiting“ führen. Die Nutzung von Events als Interface zwischen den beteiligten Kommunikationspartnern resultiert in kleinen, genau beschriebenen Schnittstellen. Technisch werden Events durch Bitfelder realisiert, wobei jedes Bit genau einen Event repräsentiert.

Counter und Alarme

Echtzeitfähige Systeme benötigen die Fähigkeit zur Behandlung von zeitabhängigen Diensten. OSEK/VDX stellt dazu die Kombination von Countern und Alarmen zur Verfügung. Ein Counter kann beliebige Ereignisse zählen. Dazu werden sehr oft die „Ticks“ einer internen Uhr (clock) verwendet. Möglich ist auch das Zählen von externen Ereignissen oder das Auftreten von Fehlern innerhalb der Anwendung. Erreicht der Counter einen voreingestellten Wert, so wird eine festgelegte Aktion ausgeführt. Mögliche Aktionen sind das Aktivieren eines Tasks, das Auslösen eines Events oder das Aufrufen einer Rückruffunktion (callback function). Eine *Rückruffunktion* ist dabei eine Funktion, die einer anderen Funktion als Parameter übergeben wird und von dieser unter gewissen Bedingungen aufgerufen wird.

Mit Hilfe dieses Konzeptes lässt sich die zyklische Ausführung von Tasks realisieren, wobei modernere OSEK/VDX-Varianten für die zyklische Taskaktivierung die Möglichkeit der so genannten *Scheduletables* anbieten. Weiterhin kann ein Task auch nach jeder Kurbelwellenumdrehung gestartet werden. Synchronität zwischen verschiedenen Tasks kann durch das Binden von mehreren Alarmen an einen Counter realisiert werden.

Ressourcenverwaltung

In Multitaskingsystemen besteht grundsätzlich das Problem der Konsistenzsicherung von gemeinsam benutzten Systemressourcen, wie z. B. Speicher oder Hardwarekomponenten. In der Praxis werden dazu Semaphoren, das Unterbinden von Kontextwechseln und das Unterbinden von Interrupts benutzt. Semaphoren signalisieren den Zustand der gemeinsam benutzten Systemressourcen und werden zur Steuerung des Taskzustandes („running“ oder „waiting“) benutzt.

Ein Problem, das bei der Verwendung von Semaphoren auftreten kann, ist das Entstehen von Verklemmungen (Deadlocks). Darunter versteht man Situationen, in denen Tasks im Zustand

„waiting“ verharren, weil diese jeweils auf eine durch einen anderen Task gesperrte Ressource warten. Dies ist in Echtzeitsystemen nicht tragbar.

Eine Möglichkeit, Verklemmungen zu vermeiden, ist die Methode der *Prioritätsgrenze* (Priority Ceiling Protocol). Die Prioritätsgrenze einer Ressource ist die höchste Priorität aller Tasks, die diese Ressource verwenden. Beim OSEK/VDX wird für den Task, der gerade die Ressource belegt, die aktuelle Priorität während der Belegung auf die Prioritätsgrenze hochgesetzt. So wird vermieden, dass er während der Ausführung verdrängt wird. Bei Freigabe der Ressource wird die Priorität des Tasks auf den ursprünglichen Wert zurückgesetzt.

Messages

Der aktuelle OSEK-OS-Standard verlangt auch eine Message-Basisfunktionalität. Basis ist ein asynchrones Kommunikationsprotokoll, wobei sich das dazugehörige Application Programming Interface (API) auf das Ablegen eines Wertes und die Abfrage bestimmter Werte beschränkt. Weiterhin kann beim Senden oder Empfangen von Nachrichten eine Aktion getriggert werden. Auf diese Weise besteht die Möglichkeit, beim Empfangen einer Nachricht den zugehörigen Task zu wecken oder auch zu starten.

Man unterscheidet zunächst die Kommunikation innerhalb des Steuergerätes und die Kommunikation über ein Bussystem. Die Kommunikation wird durch den OSEK-COM-Layer (siehe Abschnitt 2.2.8) ausgeführt.

Hooks und Fehlerbehandlung

Mit Hilfe von *Hooks* werden Mechanismen sowohl für eine zentrale als auch für eine dezentrale Fehlerbehandlung zur Verfügung gestellt. Dabei handelt es sich um vom Entwickler bereit zu stellende Funktionen, die das Betriebssystem in bestimmten Situationen aufruft (vgl. Bild 2-7).

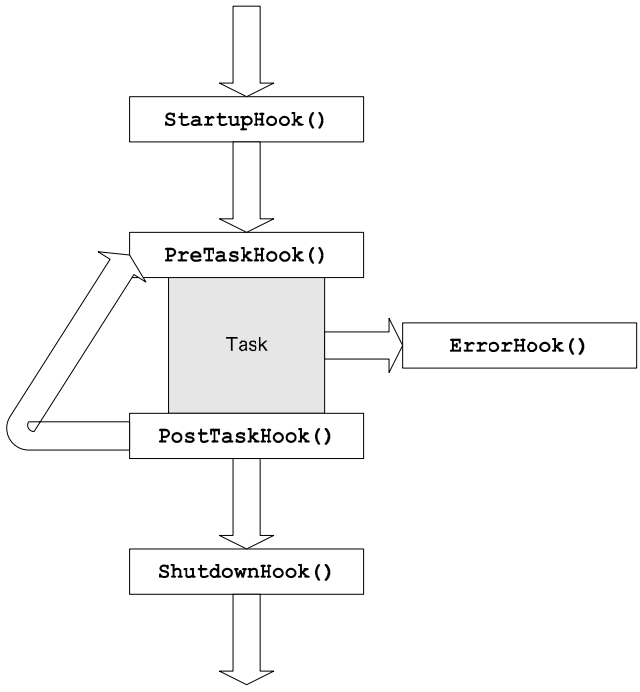

Bild 2-7 Wichtige Hooks im OSEK/VDX

Der StartupHook() wird beim Hochfahren des Systems (noch vor dem Start des Schedulers) ausgeführt und kann für systemweite Initialisierungen benutzt werden. Beim Herunterfahren wird der ShutdownHook() ausgeführt. Der ErrorHook() wird jedes Mal aufgerufen, wenn beim Aufruf einer API-Routine ein Fehler auftritt. Dabei wird der Fehlercode dem Hook übergeben. Der PreTaskHook() und der PostTaskHook() werden bei jedem Taskwechsel in oder aus dem Zustand „running“ aufgerufen.

2.2.4 Skalierbarkeit

OSEK/VDX wird auf einer breiten Anzahl von Mikroprozessoren benutzt, sowohl auf 8-Bit-Prozessoren als auch auf leistungsfähiger 32-Bit-Hardware. Um eine betriebsmitteleffiziente Anpassung des Betriebssystems an die Anwendung zu ermöglichen, stellt OSEK/VDX vier verschiedene *Kompatibilitätsklassen* (Conformance Classes) zur Verfügung (siehe Tabelle 2.2). Sie unterscheiden sich in der Anzahl der möglichen Tasks, deren Prioritäten und deren Art (Basic und Extended Tasks) sowie in der Ausführung des Schedulers.

Tabelle 2.2 Kompatibilitätsklassen

Klasse	Eigenschaften
BCC1 (Basic Conformance Class 1)	Nur einfache Tasks Nur eine Aktivierung pro Task erlaubt Nur ein Task pro Prioritätslevel
BCC2 (Basic Conformance Class 2)	Nur einfache Tasks Mehrfache Aktivierung pro Task erlaubt Mehr als ein Task pro Prioritätslevel möglich
ECC1 (Extended Conformance Class 1)	Extended Tasks sind erlaubt Nur eine Aktivierung pro Task erlaubt Nur ein Task pro Prioritätslevel
ECC2 (Extended Conformance Class 2)	Extended Tasks sind erlaubt Mehrfache Aktivierung pro Task erlaubt Mehr als ein Task pro Prioritätslevel möglich

2.2.5 Prioritätssteuerung

Wie bereits erwähnt, hat ein Task eine ihm zugewiesene Priorität. Über diese wird gesteuert, in wie weit sich die Tasks untereinander unterbrechen können. Grundsätzlich darf ein Task nur von einem Task mit einer höheren Priorität unterbrochen werden. Der Scheduler stützt sich bei der Auswahl des Tasks, der als nächstes ausgeführt werden soll, auf die Taskpriorität. Es wird der ablauffähige Task mit der höchsten Priorität ausgeführt.

Der Zeitpunkt, zu dem der Scheduler in Aktion tritt, hängt von der Schedulingstrategie ab. OSEK/VDX kennt drei Modi. Im *nicht-präemptiven Modus* sind Tasks nicht durch andere Tasks unterbrechbar. Ein Taskwechsel kann erst nach Beendigung des Tasks oder durch das Warten des Tasks auf einen Event erreicht werden (vgl. Bild 2-8a). Im Gegensatz dazu ist im *präemptiven Modus* jeder Task durch einen anderen Task mit höherer Priorität unterbrechbar.

Damit läuft zu jedem Zeitpunkt der ausführbare Task mit der höchsten Priorität (vgl. Bild 2-8b). Im *gemischten Modus* ist es möglich, das Verhalten jedes einzelnen Tasks individuell festzulegen.

Das OSEK/VDX-Betriebssystem unterscheidet mit der Interruptebene, der Betriebssystemebene und der Taskebene drei Prioritätsebenen. Es unterstützt eine implementierungsspezifische Anzahl von Taskprioritäten. Für die Interruptebene müssen prozessorspezifische Eigenheiten berücksichtigt werden. Weiterhin muss eine Zuordnung von Betriebssystemprioritäten zu Hardwareprioritäten erfolgen, was besonders sorgfältig zu erfolgen hat, da dies eine häufige Fehlerquelle darstellt.

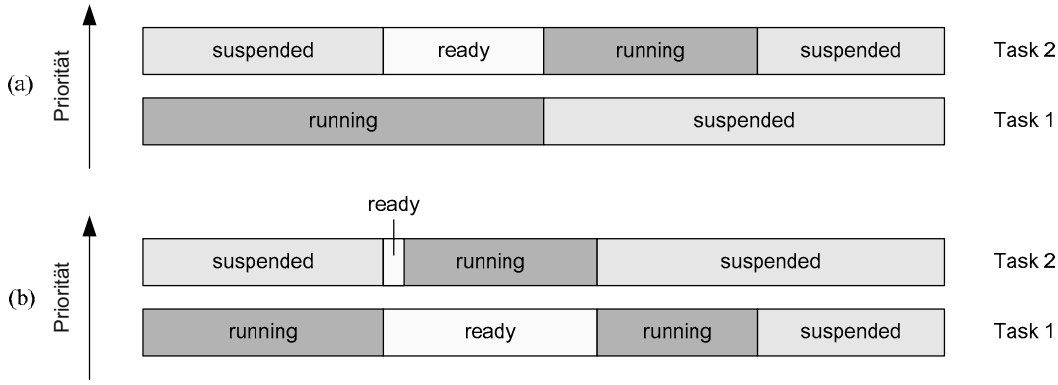

Bild 2-8 Unterschiedliche Schedulingstrategien: (a) Nicht präemptiv. (b) Präemptiv

2.2.6 Konfiguration

Das OSEK/VDX-Entwicklungsmodell setzt voraus, dass das vollständige Betriebssystem statisch konfiguriert wird und sich somit zur Laufzeit nicht ändern kann. Diese Konfiguration des OSEK-OS-Kernels und der weiteren Komponenten erfolgt über eine Datei, die in der so genannten OSEK Implementation Language (OIL) geschrieben ist, wobei häufig grafische Konfigurationswerkzeuge benutzt werden. Diese ermöglichen eine Beschreibung der Betriebsmittel und deren mögliche Verknüpfung. Dazu zählen z. B.:

- die verwendete Scheduling-Strategie,
- die Priorität von Tasks,
- die Sichtbarkeit von Events in Tasks,
- die Zuordnung von Interrupt Requests und Interrupt-Service-Routinen,
- Verknüpfungen zwischen Countern und Alarmen sowie die von Alarmen ausgelösten Aktionen,
- Messages,
- Hook-Routinen.

Eine derartige Datei zur Konfiguration kann z. B. folgendermaßen aussehen:

```
CPU OSEK_Demo
{
  OS Example_OS
  {
    MICROCONTROLLER = Intel80x86;
    STATUS = EXTENDED;
    STARTUPHOOK = FALSE;
    ERRORHOOK = FALSE;
    SHUTDOWNHOOK = FALSE;
  };
  TASK Sample_TASK
  {
    TYPE = EXTENDED;
    PRIORITY = 12;
    SCHEDULE = FULL;
    AUTOSTART = TRUE;
    ACTIVATION = 1;
  };
  COUNTER System_COUNTER
  {
    MINCYCLE = 1;
    MAXALLOWEDVALUE = 1000;
    TICKSPERBASE = 5;
  };
  ALARM Sample_ALARM
  {
    ACTION = ACTIVATETASK
    {
      TASK = Sample_TASK;
    };
    AUTOSTART = FALSE;
    COUNTER = System_COUNTER;
  };
  ISR Sample_ISR
  {
      CATEGORY = 2;
  };
};
```

Mit Hilfe der Konfiguration kann man das benötigte OSEK-OS an die Anforderungen anpassen. Bild 2-9 zeigt den prinzipiellen Ablauf beim Erzeugen von Steuergerätesoftware. Durch einen entsprechenden Ablauf wird aus dem Programmcode und den entsprechenden Konfigurationsfiles ein ausführbares und downloadbares Kompilat erzeugt.

Prinzipiell besteht die Möglichkeit, die Konfiguration für verschiedene Plattformen zu benutzen. Dabei bietet es sich an, diese Konfigurationen in unterschiedliche Teile zu strukturieren, die per include-Anweisung eingebunden werden können.

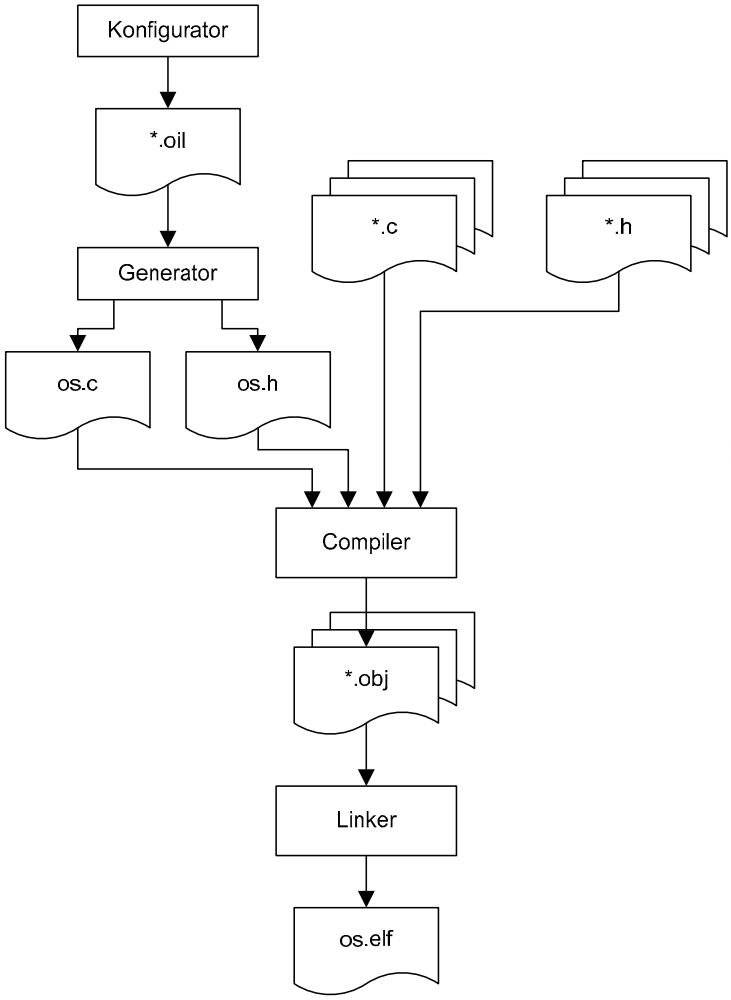

Bild 2-9 Ablauf der Softwareerstellung in einem OSEK/VDX-System

2.2.7 Hochlauf

Der Hochlauf (Start-up) eines elektronischen Steuergerätes ist ein wichtiger Gesichtspunkt der Entwicklung, da die Systemfunktionen möglichst schnell zur Verfügung stehen müssen. Ein Steuergerät muss in unterschiedlichen Betriebsmodi starten können. Dazu zählen z. B. der normale Betriebsmodus, ein Modus für die Neuprogrammierung der Steuergeräte oder ein Testmodus. OSEK/VDX unterstützt dazu die so genannten *Application Modes*, die es ermöglichen, die Steuergerätesoftware in klare funktionale Teile zu zerlegen, die zu unterschiedlichen Zeitpunkten ausgeführt werden. Die Festlegung, welcher Application Mode benutzt wird, wird während des Hochlaufs getroffen und kann im laufenden Betrieb nicht mehr geändert werden.

Der eigentliche Start-up gliedert sich in mehrere Teile. Zunächst müssen die Hardware, der Mikrocontrollerkern sowie das Speichersystem initialisiert werden. Daran schließt sich die Initialisierung des verwendeten Laufzeitsystems an, und es wird die Funktion main() aufgeru-

fen. In dieser erfolgen weitere Einstellungen der Peripherie sowie der Aufruf der Funktion StartOS() mit dem verwendeten Application Modes. Die Funktion StartOS() wird erst beim Aufruf der Funktion ShutdownOS() beendet.

2.2.8 Kommunikation

Die Spezifikation OSEK-COM stellt die Kommunikationsschicht innerhalb eines OSEK/VDX-Systems dar. Dabei werden sowohl innerhalb als auch zwischen den verschiedenen Steuergeräten Schnittstellen und Protokolle zum Datenaustausch spezifiziert (vgl. Bild 2-6). OSEK-COM ist primär auf die Zusammenarbeit mit einem Betriebssystem ausgelegt, das den OSEK-OS-Spezifikationen genügt. Die Spezifikation umfasst den Interaction Layer, den Network Layer sowie den Datalink Layer. Zur Kommunikation existieren unterschiedliche API-Befehle, wobei SendMessage() und ReceiveMessage() die wichtigsten sind.

2.2.9 Netzwerk-Management

Die Spezifikation OSEK-NM ist verantwortlich für das Netzwerk-Management zwischen den unterschiedlichen Steuergeräten. Hauptaufgabe des Netzwerk-Managements ist es, die Netzwerkknoten zu überwachen, um die Sicherheit und die Verfügbarkeit des Netzwerkes sicherzustellen. Weiterhin dient es dazu, den Anwendungsprogrammen Informationen über die Erreichbarkeit von Busteilnehmern bereitzustellen. OSEK-NM bietet im Zusammenspiel mit OSEK-OS und OSEK-COM (siehe auch Bild 2-6) den Mechanismus der direkten sowie der indirekten Überwachung an [Ho3].

Bei der direkten Überwachung werden bestimmte Nachrichten nach dem Token-Prinzip in einem logischen Ring versandt (vgl. Abschnitt 1.1.7). Dabei wird jeder Knoten von allen Knoten überwacht. Ein Nachteil der direkten Überwachung ist die erhöhte Buslast. Für Systeme, bei denen die direkte Überwachung nicht möglich oder nicht notwendig ist, gibt es auch die Möglichkeit der indirekten Überwachung. Hierbei werden die Nachrichten der Anwendungen überwacht. Dies funktioniert nur bei Knoten, die periodisch Nachrichten verschicken.

2.2.10 OSEK/VDX-Erweiterungen

OSEK/VDX hat sich de facto als Standard in der Automobilbranche durchgesetzt, was durch die ISO-Norm 17356 [Is9] weiter gefördert wird. Im Laufe der Entwicklungen sind weitere Anforderungen wie z. B. die Notwendigkeit von Schutzmechanismen zur Nutzung mehrerer Anwendungen auf einem Steuergerät sowie die Nutzung zeitgesteuerter Bussysteme entstanden. So stellt z. B. OSEK-FTCom Schnittstellen und Protokolle für eine fehlertolerante Kommunikation zur Verfügung. Innerhalb der AUTOSAR-Initiative (siehe Abschnitt 2.3) wird OSEK/VDX in Form des darunter liegenden Betriebssystems weiterentwickelt.

Für bestimmte Anwendungen bietet OSEKtime Dienste für verteilte, fehlertolerante Echtzeitanwendungen an, die z. B. die Task-Synchronisation über eine globale Zeitbasis erfordern. Sollten die Funktionen von OSEK-OS und OSEKtime-OS gleichzeitig benötigt werden, ist es möglich, beide parallel auf einem Steuergerät laufen zu lassen. Dabei läuft das klassische OSEK-OS als Background-Task unter dem OSEKtime-OS.

Tasks innerhalb von OSEKtime besitzen ein anderes Task-Modell, d. h., Tasks sind grundsätzlich Funktionen ohne Endlosschleifen oder Wartevorgänge. Für jeden Task ist die maximale Ausführungszeit (WCET) bekannt. Die Zeitpunkte des Aufrufes eines Tasks werden im Voraus berechnet und in einer Ablauftabelle (Dispatch-Tabelle) gespeichert.

Wie bereits beschrieben, bilden am Ende des Entwicklungsprozesses die so genannte Fahrsoftware und der OSEK-Kern eine Softwareeinheit. Dabei wird weder eine Adressraumisolation noch Speicherschutz umgesetzt. Anforderungen seitens der Automobilhersteller wie auch der Hersteller von Steuergeräten (speziell im Zusammenhang mit der Integration unterschiedlicher Software auf einem Steuergerät) führten zu Überlegungen, OSEK-OS um bestimmte Schutzmechanismen zu erweitern.

Neuere OSEK/VDX-Versionen unterstützen inzwischen einen Speicherschutz auf Anwendungsebene sowie einen Laufzeitschutz für Tasks. Für den Speicherschutz werden leistungsfähige Mikrocontroller benötigt, welche eine Memory-Management-Unit (MMU) oder eine Memory-Protection-Unit (MPU) implementiert haben.

2.3 AUTOSAR

Moderne Fahrzeuge besitzen eine Vielzahl von Funktionen. Die Erstellung dieser Funktionen erfolgt meist unter der Nutzung fahrzeugspezifischer Lösungen. Damit entstehen erhöhte Schwierigkeiten, bereits entwickelte Funktionalitäten zwischen verschiedenen Fahrzeugbaureihen wieder zu verwenden. Dies führt zu einer weiter steigenden Komplexität im Entwicklungsprozess. Daher wurde im Juli 2003 die Automotive Open System Architecture (AUTOSAR) als eine internationale Organisation gegründet, die das Ziel verfolgt, einen offenen Standard für Elektrik/Elektronik-Architekturen in Kraftfahrzeugen zu etablieren. Mitglieder sind verschiedene Automobilhersteller und Zulieferer von Elektronikkomponenten. Als so genannte Core-Partner arbeiten Bosch, BMW, Continental, Daimler, Ford, General Motors, PSA Peugeot Citroën, Toyota und Volkswagen aktiv an der Weiterentwicklung des Standards mit. Die Core-Partner werden durch weitere Premium-Mitglieder sowie assoziierte Mitglieder unterstützt. Eine der Grundideen des Konsortiums lautet „Zusammenarbeit bei Standards – Wettbewerb bei der Umsetzung“ (Cooperate on Standards – Compete on Implementation). Folgende Fragestellungen werden behandelt:

- Standardisierung wichtiger Systemfunktionen,
- Skalierbarkeit,
- Verschiebbarkeit von Funktionen im Fahrzeugnetzwerk,
- Integration und Austauschbarkeit von Software verschiedener Hersteller,
- Unterstützung so genannter Commercial Off-The-Shelf-Software (COTS), d. h. von „Seriensoftware“ ohne individuelle Anpassung,
- Wartbarkeit über den gesamten Produktlebenszyklus.

AUTOSAR bietet neben einer standardisierten und werkzeuggestützten Konfiguration der hardwarenahen Software auch die Möglichkeit, Anwendungssoftware in Softwarekomponenten zu strukturieren und diese mit Hilfe einer statischen Middleware transparent (z. B. auch über Steuergerätegrenzen hinaus) miteinander kommunizieren zu lassen. Dieses Prinzip trägt zu einer erhöhten Wiederverwendbarkeit und Portierbarkeit der Anwendungssoftware und damit zu einer beschleunigten Entwicklung zukünftiger Anwendungen bei.

2.3.1 Entwicklungshistorie und Roadmap

Innerhalb der AUTOSAR-Initiative werden unter Berücksichtigung dieser Fragestellungen Standards definiert, auf deren Grundlage zukünftige Anwendungen im Fahrzeug entwickelt werden können. Man erhofft sich durch die Nutzung der definierten Standards, die wachsende

Komplexität bei der Entwicklung von elektrischen und elektronischen Fahrzeugkomponenten beherrschbar zu halten. Bislang wurden ein Basis-Softwarekern, funktionale Schnittstellen sowie Methoden zur Integration von Software definiert. Die Erarbeitung und Verabschiedung der Standards erfolgt in verschiedenen Arbeitsgruppen. Eine gemeinsam erarbeitete Roadmap sichert sowohl die Inhalte als auch den Zeithorizont ab. Erste Spezifikationen stehen seit Mitte 2005 zur Verfügung, Steuergeräte mit Teilfunktionen gemäß dem AUTOSAR-Standard werden seit 2009 in Serienfahrzeugen verbaut. Die weitere AUTOSAR-Entwicklung fokussiert sich auf die Stabilisierung des Standards und der Integration notwendiger Erweiterungen.

2.3.2 Softwarekomponenten

Bild 2-10 zeigt die grundlegende AUTOSAR-Softwarearchitektur. Die Anwendungssoftware ist in unabhängigen Einheiten, den so genannten Softwarekomponenten (SWC), organisiert. Eine solche Komponente kapselt die Implementierungsdetails und stellt einfache und klar definierte Schnittstellen zur Verfügung. Auf diese Weise ist die Softwarekomponente ein wichtiges Strukturierungs- und Architekturelement der gesamten Steuergerätesoftware.

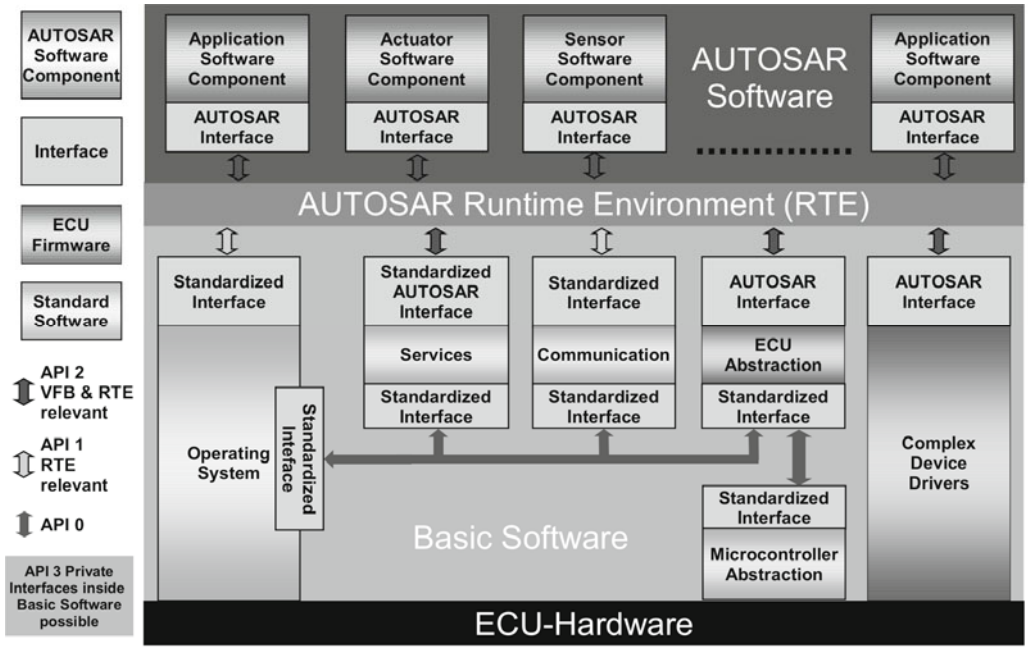

Bild 2-10 AUTOSAR-Softwarearchitektur (aus [Au2]; es wurden die englischen Begriffe verwendet, um konform mit der Spezifikation zu sein): ECU Steuergerät, API Programmierschnittstelle (Application Programming Interface)

Neben den hardwareunabhängigen Softwarekomponenten existieren noch Sensor-Softwarekomponenten sowie Aktor-Softwarekomponenten. Diese Komponenten dienen zur Ankopplung von Sensoren oder Aktoren, was bedeutet, dass sie hardwarespezifisch sind. Ziel dieser Komponenten ist es, physikalische Eigenschaften eines Sensors oder eines Aktors in speziellen Softwarekomponenten zu kapseln. Auf diese Weise existiert ein einheitlicher Zugriff auf die Hardware der Steuergeräte.

Nicht weiter teilbare Softwarekomponenten werden als Atomic Software Components bezeichnet. Zusätzlich können Softwarekomponenten hierarchisch strukturiert werden. Diese so genannten Compositions können neben Softwarekomponenten wiederum Kompositionen von Softwarekomponenten enthalten. Die Nutzung dieser Strukturierungskonzepte ist Grundlage zur Bildung einer Komponentenbibliothek, die logisch zusammenhängende Funktionalitäten als Softwarekomponenten oder – im Falle von komplexeren Konstrukten – als Kompositionen zur Verfügung stellen kann.

Ein wichtiger Baustein der Beschreibung einer Softwarekomponente ist das interne Verhalten (Internal Behavior), welches den Ablauf der einzelnen ausführbaren Elemente (ein Algorithmus oder eine Funktion) innerhalb einer jeden Softwarekomponente beschreibt. Zusätzlich wird zur Beschreibung ein Triggerereignis benötigt. Dieses Triggerereignis beschreibt den Start eines solchen Elements. Dazu zählen:

- zeitliche Ereignisse (Timing Events),
- Ereignisse beim Senden oder Empfangen von Daten (Data Received Event, Data Receive Error Event, Data Send Completed Event),
- Ereignisse bei Zustandsänderungen (Mode-Switch Event),
- Ereignisse im Zusammenhang mit Client-Server-Operationen (Operation Invoked Event, Asynchronous Server Call Returns Event).

Weitere Informationen über Triggerereignisse sind in [Au2] zu finden.

In einer Softwarekomponente befinden sich verschiedene Codeabschnitte, die so genannten Runnables. Diese Runnables, aus Softwarearchitektursicht die kleinsten Teile, sind Funktionen innerhalb einer Softwarekomponente, welche in verschiedenen Zeitrastern oder eventbasiert von der Basissoftware aufgerufen werden können.

In Bild 2-10 sind weiterhin Complex Device Drivers (CDD) dargestellt. Diese heben die Schichtenstruktur wieder auf, indem sie alle Schichten der Basissoftware durchschneiden. Auch wenn sie auf den ersten Blick die Schichtenstruktur verletzen, erfüllen Complex Device Drivers wichtige Aufgaben. Sie öffnen zunächst einen Weg, um bestehende Software stufenweise in eine AUTOSAR-Softwarearchitektur zu integrieren, indem bestehende Software um die entsprechenden Schnittstellen erweitert wird. Weiterhin existieren in Steuergeräten immer Softwareteile, typischerweise zur Ansteuerung spezieller Sensoren und Aktoren, die spezielle Anforderungen an die benötigte Software stellen und für die keine AUTOSAR-Treiber existieren oder spezielle Optimierungen notwendig sind.

2.3.3 Kommunikationsarten

Der Datenaustausch zwischen Softwarekomponenten kann sowohl innerhalb eines Steuergerätes als auch zwischen unterschiedlichen Steuergeräten stattfinden. Die Kommunikation zwischen AUTOSAR-Softwarekomponenten findet über Ports statt, wie in Bild 2-11 skizziert. Dabei wird zwischen der Client-Server- und der Sender-Receiver-Kommunikation unterschieden. Beide besitzen durch ihre unterschiedlichen Eigenschaften verschiedene Anwendungsgebiete.

Das Modell der Client-Server-Kommunikation geht davon aus, dass eine Rückmeldung über den erfolgreichen Datenaustausch zwischen Client und Server stattfindet. Dabei startet der Client den Kommunikationsaufbau und auf dem Server eine bestimmte Aktion. Vereinfacht ausgedrückt stellt die Client-Server-Kommunikation einen Funktionsaufruf in einer anderen Komponente dar.

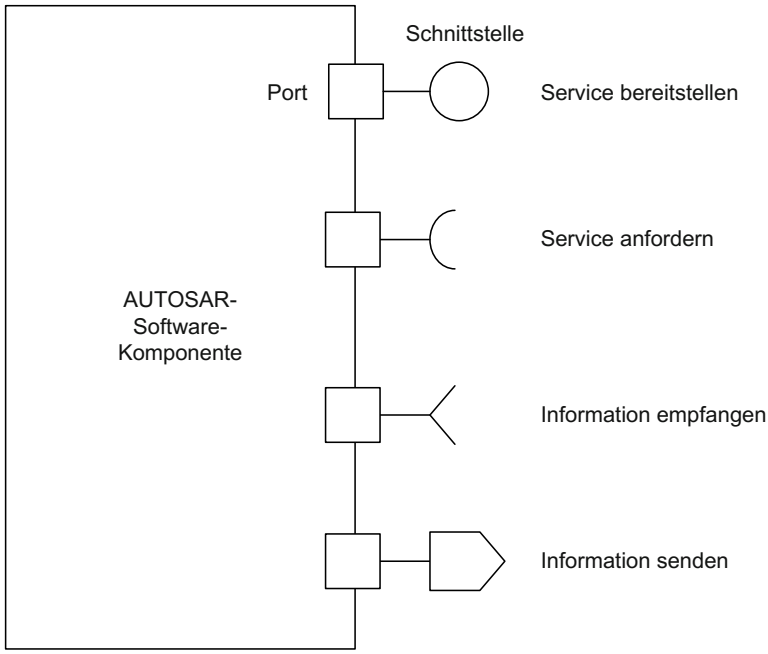

Bild 2-11 Mögliche Kommunikationsarten einer AUTOSAR-Softwarekomponente

Die Sender-Receiver-Kommunikation ist ein Kommunikationsmuster, welches zum asynchronen Informationsaustausch zwischen Sender und Receiver benutzt werden kann. Dabei ist es möglich, 1:1-, 1:*m*- und *n*:1-Beziehungen aufzubauen. Es erfolgt keine Rückmeldung an den Sender und der Empfänger entscheidet selbst, ob und wann die empfangene Information genutzt wird.

2.3.4 Basissoftware

Alle Softwaremodule, die unterhalb des Runtime Environment angesiedelt sind und die spezifische Dienste erbringen, werden als Basissoftware bezeichnet. Dabei wird versucht, die Arbeiten und Standards von OSEK, HIS [Hi2], ASAM [As3] und ISO sowie der Industriekonsortien für CAN, Flexray, LIN (vgl. Kapitel 1) zu nutzen. Ein wesentliches Ziel von AUTOSAR ist es, die Funktionalität der Basissoftware zu standardisieren. Daher existieren für alle relevanten Module entsprechende Beschreibungsmodelle. Dazu gehören z. B. die Softwaremodule für die Kommunikationssysteme, die Softwareteile für die Nutzung von Speicher und die Module zur Nutzung typischer Mikrocontrollerperipherie. In Bild 2-12 realisieren alle Softwareteile, die unterhalb des Runtime Environment abgebildet sind, Funktionalitäten der Basis-Software.

Typischerweise erfolgt die Einführung von AUTOSAR-Konzepten im Bereich der Basissoftware über den Weg der Kommunikationssoftware [Hi1], da hier die Standardisierung am weitesten fortgeschritten ist. Durch den hohen Vernetzungsgrad moderner Fahrzeuge ergab sich hier der größte Nutzen bei der AUTOSAR-Einführung.

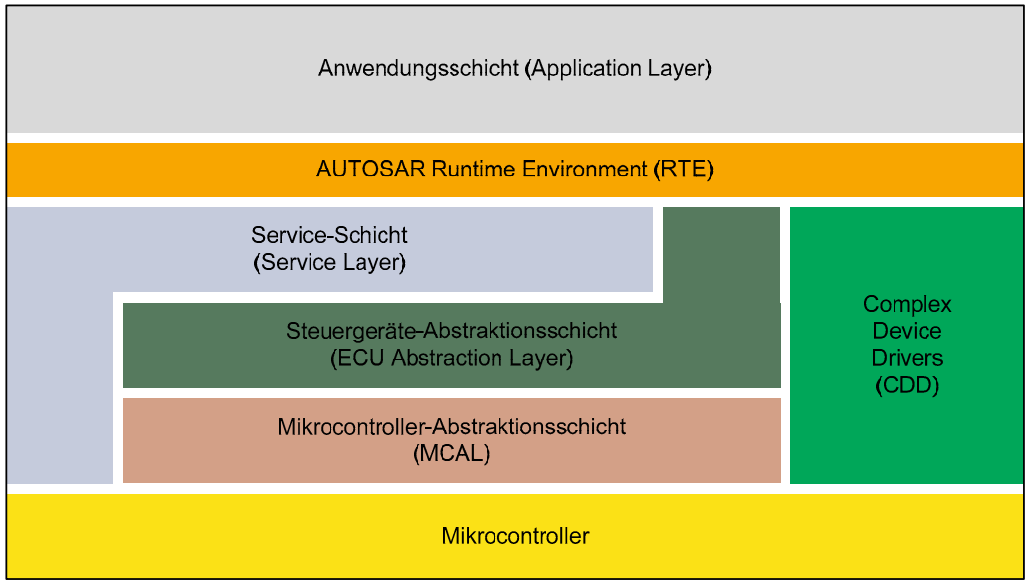

Bild 2-12 Übersicht über die Schichtenstruktur [Au2]

Allgemein besitzen die Kommunikationsstacks eine hohe Komplexität. Am Beispiel des Aufbaus des CAN-Kommunikationsstacks (siehe Bild 2-13) soll die typische Struktur erläutert werden. Weitere Informationen über den Aufbau unterschiedlicher Kommunikationsstacks sind in [Ki2] zu finden. Oberhalb des Runtime Environment wird eine signalbasierte Schnittstelle benutzt. Das Runtime Environment stellt dazu das entsprechende Signalmapping zwischen dem RTE-Signal und dem zugehörigen Identifier im Kommunikationsstack her. Die Zusammenfassung der Signale auf bustypische Strukturen bzw. Botschaften erfolgt im COM. Der PDU-Router vermittelt bustypische Strukturen zwischen den unterschiedlichen busspezifischen Schnittstellen und den verschiedenen Nutzern. In Bild 2-13 sind beispielhaft nur der PDU-Router als Nutzer und das CAN-Interface als busspezifische Schnittstelle dargestellt. Das CAN-Interface abstrahiert die Ansteuerung unterschiedlicher CAN-Treiber. Aufgaben des CAN-Interface sind:

- Initialisierung der CAN-Treiber,
- Handling und Versand der bustypischen Botschaften,
- zugehöriges Fehlerhandling.

Zwischen dem CAN-Interface und der eigentlichen Hardware (CAN-Controller und CAN-Transceiver) befinden sich der CAN-Driver und der CAN-Transceiver-Driver. Der CAN-Transceiver-Driver nutzt Dienste des DIO-Drivers (Digital Input/Output), um die notwendigen Mikrocontroller-Pins zu setzen und den CAN-Transceiver anzusteuern. Die Information über den Empfang neuer Nachrichten kann entweder interruptgesteuert oder durch einen Pollingmechanismus realisiert werden.

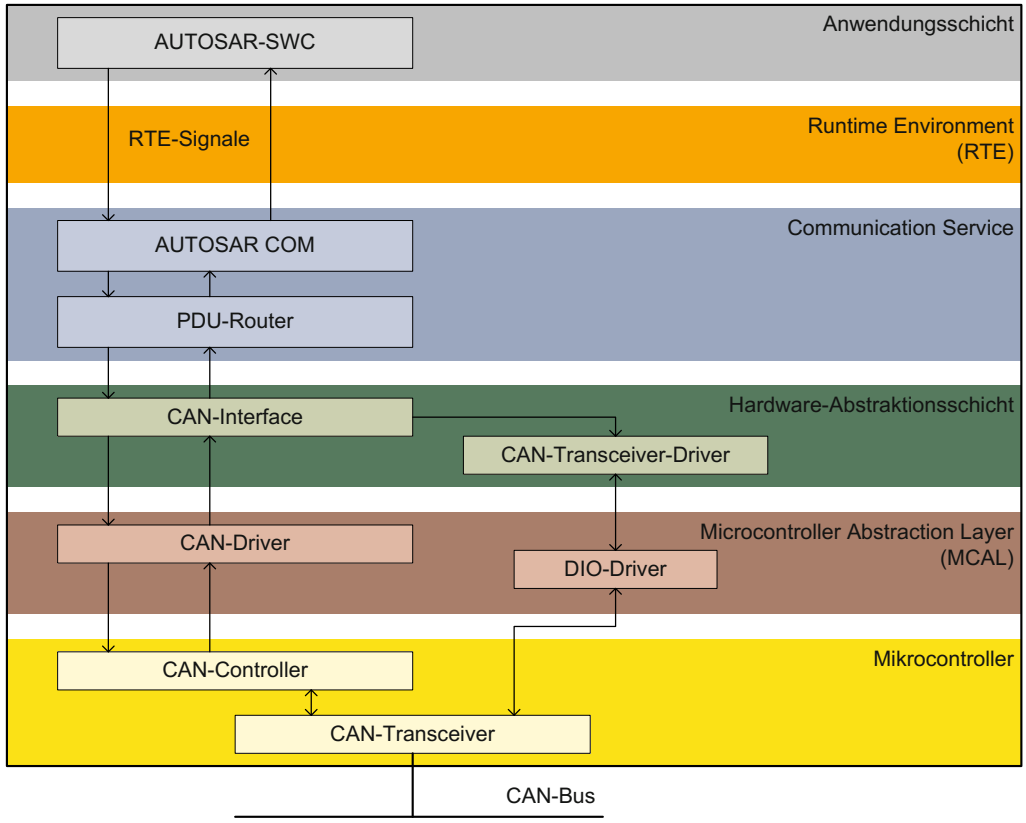

Bild 2-13 Aufbau, Zusammenhänge und Ablauf im CAN-Kommunikationsstack. SWC Software-Komponente, RTE Runtime Environment, PDU Protocol Data Unit, DIO Digital Input/Output

2.3.5 Virtueller Funktionsbus

Die Entwicklung komplexer Funktionen, die häufig auch im Steuergeräteverbund realisiert werden, erzwingt den konsequenten Einsatz von Softwarearchitekturmethoden, wobei hier besonders die Idee der *Separation of Concerns* [Di6] erwähnt werden soll. Diese Idee besagt, dass Aufgaben in unabhängige Teilbereiche zerlegt werden, die auch unabhängig voneinander entwickelt werden können.

Des Weiteren ist es notwendig, von einer auf einzelne Steuergeräte bezogenen Sicht den Weg zu einem steuergeräteübergreifenden und funktionsorientierten Entwicklungsansatz zu gehen. Dies erfordert, dass verschiedene Aspekte des zu realisierenden Systems in sauber definierten Teilen steuergeräteunabhängig umgesetzt werden. AUTOSAR kombiniert dazu die Schichten- und die Komponentenarchitektur und versucht auf diese Weise fachliche und technische Softwareteile zu trennen.

Für einen Entwickler einer neuen Fahrzeugfunktion besteht diese Funktion aus einer Menge von Software-Komponenten (siehe Abschnitt 2.3.2), die über Ports miteinander kommunizieren (siehe Abschnitt 2.3.3). Auch die notwendigen Systemservices der Basissoftware (siehe Abschnitt 2.3.4) sind über entsprechende Ports ansprechbar.

Diese grundsätzliche Idee eines virtuellen Funktionsbusses ist in Bild 2-14 dargestellt. Dabei stellt der virtuelle Funktionsbus (Virtual Functional Bus, VFB) das übergreifende Kommunikationskonzept der AUTOSAR-Architektur in einem Steuergeräteverbund dar. Diese steuergeräteübergreifende Schicht setzt sich aus allen durch AUTOSAR spezifizierten Kommunikationskonzepten zusammen. Der virtuelle Funktionsbus ist ein Modellierungselement, das während des Generierungsprozesses in Steuergerätecode überführt wird, d.h. er existiert auf der Implementierungsebene der einzelnen Steuergeräte in Form des Runtime Environment. Das Ziel des Runtime Environment ist es, für die eigentliche Anwendungssoftware die tiefer gelegenen Softwareschichten zu verbergen. Damit wird klar, dass Anwendungs-Software-Komponenten grundsätzlich hardwareunabhängig implementiert werden müssen, wie am Beispiel des CAN-Kommunikationsstacks bereits gezeigt wurde.

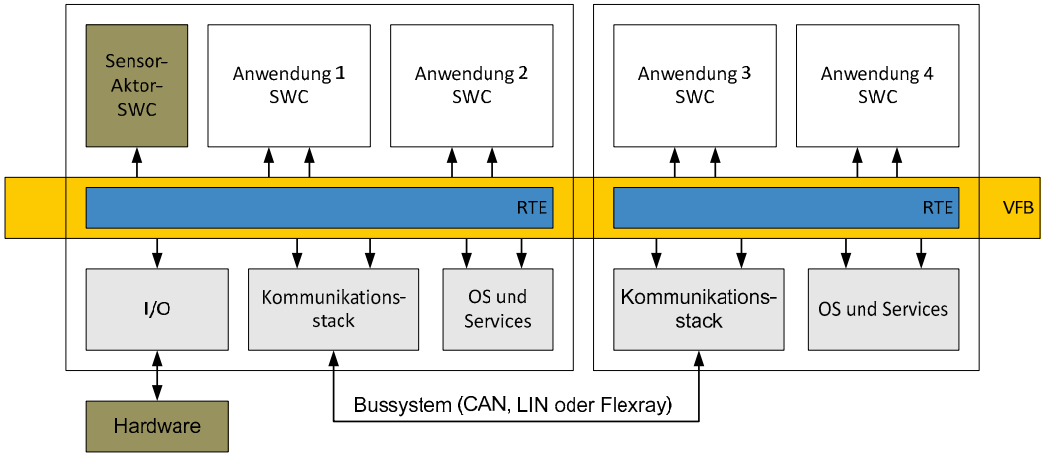

Bild 2-14 Darstellung des virtuellen Funktionsbusses zwischen zwei Steuergeräten und einer möglichen Verteilung von Softwarekomponenten zwischen den Steuergeräten. I/O Ein- und Ausgabe, OS Betriebssystem, RTE Runtime Environment, SWC Software-Komponente, VFB virtueller Funktionsbus

Zwei Anwendungen können unter Nutzung so genannter Kommunikationsports ohne Kenntnis des realen Signalpfades Informationen austauschen. Dabei werden die Softwareteile für die Ansteuerung der Sensorik und Aktorik (Sensor-SWC, Aktor-SWC und zugehörige I/O-Treiber) in Abhängigkeit verschiedener Randbedingungen auf geeigneten Steuergeräten integriert (Deployment). Die eigentlichen Anwendungen werden entsprechend der funktionalen Anforderungen verteilt. Fahrzeugspezifische Funktionen können unter Ausnutzung dieser Eigenschaft zunächst ohne Kenntnis der im Fahrzeug verwendeten Vernetzungstopologie entwickelt werden. Die tatsächlich verwendeten Signalpfade werden im späteren Phasen der Entwicklung durch eine Konfiguration festgelegt. Bei der Verteilung der Software auf die Steuergeräte ist das unterschiedliche Zeitverhalten der Kommunikation innerhalb eines Steuergerätes und zwischen den Steuergräten zu berücksichtigen.

2.3.6 Laufzeitumgebung

Das Kernstück des AUTOSAR-Architekturkonzepts ist die Laufzeitumgebung (Runtime Environment, RTE). Sie implementiert den virtuellen Funktionsbus auf der Steuergeräte-Ebene und ist damit eine einfache und statisch konfigurierbare Middleware. Sie stellt für jede Software-

komponente eines Steuergerätes spezifische Funktionsaufrufe zur Verfügung, die nur dieser Anwendung zugeordnet und bekannt sind. Für die Konfiguration des Runtime Environment muss somit bekannt sein, welche Softwarekomponente auf welchem Steuergerät ausgeführt wird. Entsprechend dieser Kenntnis wird das Runtime Environment anschließend von einem Softwaretool generiert. Aufgabe dieses Werkzeuges ist es, ein leistungsfähiges und effizientes Runtime Environment zu generieren.

2.3.7 AUTOSAR-OS

Der AUTOSAR-Standard nutzt, wie in Abschnitt 2.2.10 beschrieben, zur Festlegung eines Betriebssystems die grundlegenden Konzepte des OSEK-OS. Das AUTOSAR-OS stellt in der Regel nur grundlegende Funktionalitäten wie eine statische Ablaufplanung (z. B. Scheduletables), das Ressourcenmanagement, die Zeitverwaltung und ein Interrupt-Handling zur Verfügung.

Um eine höhere Absicherung zu erreichen, sind die grundlegenden Konzepte von OSEKtime und Protected-OS (siehe Abschnitt 2.2.10) in das AUTOSAR-OS eingeflossen, die die Speicherschutzfunktionen und die Zeitüberwachungen realisieren. In Bezug auf die umgesetzten Schutzmechanismen existieren vier so genannte Scalability Classes. Über den Umfang der verfügbaren Mechanismen gibt Tabelle 2.3 Auskunft. Um diese Schutzmechanismen anwenden zu können, werden verstärkt Mikrocontroller eingesetzt, die bereits in Hardware realisierte Schutzmechanismen enthalten. Diese in Hardware realisierten Schutzmechanismen sind erheblich leistungsfähiger als rein softwarebasierte Lösungen. Derartige Mikrocontroller verfügen typischerweise über einen so genannten Kernel Mode und einen User Mode, um den privilegierten vom nicht-privilegierten Betriebsmodus zu separieren [Ma4].

Die Mechanismen der Zeitüberwachung ermöglichen eine Überwachung der definierten Zeitbudgets zur Laufzeit und im Fehlerfall eine entsprechende Fehlerreaktion. Weiterhin besteht die Möglichkeit, den zeitlichen Ablauf im Steuergerät mit einer externen Zeitbasis zu synchronisieren. Speziell bei der Nutzung von Flexray (siehe Abschnitt 1.2.4) ergeben sich damit vielfältige Möglichkeiten [Re9].

Tabelle 2.3 Übersicht über den Funktionsumfang der unterschiedlichen Scalability Classes

Scalability Class	Schedule Tables	Zeitüberwachung	Globale Zeit/ Synchronisierung	Speicherschutz	Bemerkung
SC4	×	×	×	×	
SC3	×	–	–	×	Basiert auf Konzepten von OSEKtime
SC2	×	×	×	–	Basiert auf Konzepten des Protected OS
SC1	×	–	–	–	

Das ursprüngliche OSEK-Konzept erlaubt nur ereignisgesteuertes Multitasking. Zeitgesteuerte Abläufe müssen aufwendig über Alarme nachgebildet werden. Dieses Vorgehen führt schnell zu komplexen und unübersichtlichen Systemen. Um eine übersichtliche Darstellung komplexer zeitlicher Vorgänge zu ermöglichen, wurde im AUTOSAR-OS das Konzept der Scheduletabellen (Schedule Tables) umgesetzt. Die prinzipielle Idee ist in Bild 2-15 dargestellt. Dabei werden vordefinierte zeitliche Abläufe mittels entsprechender Einträge in einer Scheduletabelle organisiert. Bei Erreichen der entsprechenden Zählerstände wird die zugehörige Aktion ausgeführt. Exemplarisch sind die Aktionen der Taskaktivierung (ActivateTask) und das Setzen eines Events (SetEvent) in der entsprechenden C-Syntax dargestellt. Ein streng deterministischer Ablauf ergibt sich jedoch nur, wenn zusätzlich zur Ablaufplanung eine passende Festlegung der Taskprioritäten erfolgt und entsprechende Schedulinganalysen [Je1], [Mü1] während aller relevanten Entwicklungsphasen durchgeführt werden.

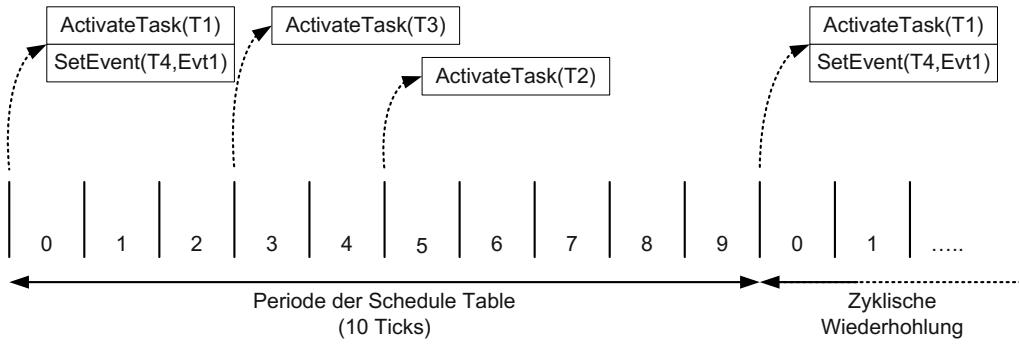

Bild 2-15 Beispiel einer Scheduletabelle (vordefinierter zeitlicher Ablauf mit zugehörigen Aktionen und Zählerständen); die dargestellten Zahlen (0, 1, …, 9) repräsentieren Zählerstände; T1, T2, T3, T4 repräsentieren Tasks und Evt1 einen Event entsprechend Abschnitt 2.2.3

2.3.8 Ausblick

Ein wichtiger Aspekt bei der Einführung und Nutzung von AUTOSAR stellt das sich verändernde Verhältnis zwischen Fahrzeughersteller und Zulieferer dar. Der gemeinsame Standard ermöglicht es den Automobilherstellern, sich auf die Kernkompetenz zu fokussieren und markenprägende Eigenschaften durch eine Funktions- und Softwareeigenentwicklung schnell umzusetzen. Des Weiteren bietet sich die Möglichkeit, durch einen gemeinsamen Standard neue Geschäftsmodelle zu entwickeln [Sc7]. Die AUTOSAR-Versionen ab Release 3.1 und folgend haben einen stabilen Stand erreicht und werden in Serienprojekten bereits eingesetzt. In den Versionen ab Release 4.0 werden unter anderem folgende Punkte bearbeitet und ergänzt:

- Verbesserung der Methodik sowie der Templates,
- die Erweiterung des Standards um Aspekte der funktionalen Sicherheit (vgl. Kapitel 9),
- die Berücksichtigung von zeitlichen Eigenschaften beim Entwurf von Steuergeräten durch ein Timing-Modell [Ri2],
- erste Ansätze zur Beschreibung Multicore-spezifischer Funktionalitäten,
- die Unterstützung weiterer Kommunikationsstandards wie z. B. von LIN 2.1 und Flexray 3.0,
- Standardisierung der Anwendungsschnittstellen.

http://www.springer.com/978-3-658-05047-4

Automobilelektronik
Eine Einführung für Ingenieure
Reif, K.
2014, XVI, 499 S. 350 Abb., 250 Abb. in Farbe.,
Softcover
ISBN: 978-3-658-05047-4