

ATZ/MTZ-Fachbuch

Werner Zimmermann
Ralf Schmidgall

Bussysteme in der Fahrzeugtechnik

Protokolle, Standards und Softwarearchitektur

5. Auflage



Springer Vieweg

ATZ/MTZ-Fachbuch

Die komplexe Technik heutiger Kraftfahrzeuge und Motoren macht einen immer größer werdenden Fundus an Informationen notwendig, um die Funktion und die Arbeitsweise von Komponenten oder Systemen zu verstehen. Den raschen und sicheren Zugriff auf diese Informationen bietet die regelmäßig aktualisierte Reihe ATZ/MTZ-Fachbuch, welche die zum Verständnis erforderlichen Grundlagen, Daten und Erklärungen anschaulich, systematisch und anwendungsorientiert zusammenstellt.

Die Reihe wendet sich an Fahrzeug- und Motoreningenieure sowie Studierende, die Nachschlagebedarf haben und im Zusammenhang Fragestellungen ihres Arbeitsfeldes verstehen müssen und an Professoren und Dozenten an Universitäten und Hochschulen mit Schwerpunkt Kraftfahrzeug- und Motorentechnik. Sie liefert gleichzeitig das theoretische Rüstzeug für das Verständnis wie auch die Anwendungen, wie sie für Gutachter, Forscher und Entwicklungingenieure in der Automobil- und Zuliefererindustrie sowie bei Dienstleistern benötigt werden.

Werner Zimmermann · Ralf Schmidgall

Bussysteme in der Fahrzeugtechnik

Protokolle, Standards und
Softwarearchitektur

5., aktualisierte und erweiterte Auflage

Mit 278 Abbildungen und 103 Tabellen



Springer Vieweg

Prof. Dr.-Ing. Werner Zimmermann
Fakultät Informationstechnik
Hochschule Esslingen
Esslingen, Deutschland

Dr. Ralf Schmidgall
Plochingen, Deutschland

ISBN 978-3-658-02418-5
DOI 10.1007/978-3-658-02419-2

ISBN 978-3-658-02419-2 (eBook)

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

Springer Vieweg
© Springer Fachmedien Wiesbaden 2007, 2008, 2011, 2014
Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung, die nicht ausdrücklich vom Urheberrechtsgesetz zugelassen ist, bedarf der vorherigen Zustimmung des Verlags. Das gilt insbesondere für Vervielfältigungen, Bearbeitungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürfen.

Gedruckt auf säurefreiem und chlorfrei gebleichtem Papier.

Springer Vieweg ist eine Marke von Springer DE. Springer DE ist Teil der Fachverlagsgruppe Springer Science+Business Media
www.springer-vieweg.de

Vorwort zur 5. Auflage

„Das Schöne an Standards ist, dass es so viele davon gibt.“ Dieses Zitat stammt aus Andrew Tanenbaums Buch über Computernetze, das erstmals 1981 veröffentlicht wurde. Es gilt uneingeschränkt auch für die heutige Kraftfahrzeugelektronik.

Als die erste Auflage des vorliegenden Buches 2006 erschien, hatte die Welt der automobilen Kommunikation eine stürmische Entwicklung hinter sich. Innerhalb weniger Jahre waren die seit den 1990er Jahren erfolgreich eingesetzten CAN- und K-Line Schnittstellen durch die neuen Konzepte LIN, FlexRay und MOST ergänzt worden. Bei den Übertragungsprotokollen hatten sich KWP 2000 und OBD für die Diagnose im PKW und SAE J1939 bei Nutzfahrzeugen etabliert. Im Softwarebereich war OSEK/VDX als Betriebssystem in den Steuergeräten eingeführt und die Flash-Programmierung, mit der die Steuergerätesoftware wie ein Ersatzteil im Laufe der Lebensdauer eines Fahrzeugs ausgetauscht werden kann, wurde Stand der Technik. Weil die Entwicklung sehr rasch verlaufen war, schien es damals an der Zeit, die neuen Konzepte in einem Buch im Zusammenhang darzustellen.

Wer damals allerdings geglaubt hatte, dass die Welt der automobilen Kommunikation konsolidieren würde, sah sich getäuscht. Auch wenn sich die Bussysteme seither eher evolutionär weiterentwickelt haben, so hat sich der Softwarebereich umso stürmischer bewegt. Nicht nur die Kommunikation, sondern die gesamte Architektur der Steuergeräte zu ordnen, ist Ziel von AUTOSAR, das die gesamte Industrie massiv umtreibt. Und parallel dazu wird in der ASAM-Initiative versucht, den Aufwand für die immer umfangreichere Fahrzeugdiagnose nicht explodieren zu lassen. Entsprechend nahm in den folgenden Auflagen dieses Buches der Softwareanteil immer weiter zu. Gekürzt konnte dagegen kaum werden, da die Automobilindustrie bei aller Innovation die schöne Angewohnheit hat, neue Lösungen zügig einzusetzen, bewährte Konzepte aber keineswegs aufs Altenteil zu schicken, sondern weiter zu pflegen.

Deutet sich heute eine Konsolidierung an oder geht die stürmische Entwicklung weiter? Letzteres. Bei den Bussystemen sind mit *CAN Flexible Data-Rate* (CAN FD) und *Automotive Ethernet/IP* zwei neue Spieler aufgetaucht, die für Furore sorgen könnten. CAN FD schickt sich an, FlexRay bei höheren Datenraten nicht kampflos das Feld zu überlassen und Ethernet könnte, vom High-End her kommend, zunächst MOST verdrängen und später vielleicht sogar die Echtzeitdomäne von FlexRay erobern.

Beide Neuentwicklungen werden in dieser Auflage genauso ausführlich dargestellt wie die Neuerungen bei *AUTOSAR 4.x*, das jetzt ein ganzes Kapitel einnimmt, und *Open Test Sequence Exchange OTX*, das die Test-Entwicklung weiter vereinfachen soll. Wie in den vergangenen Auflagen wurden wieder die Änderungen bei den bekannten Systemen MOST 150, FlexRay 3.0 und den Sensor-Aktor-Bussen wie PSI5 sowie die seit der letzten Auflage erschienen ISO-Versionen der ASAM-Standards, World-Wide Harmonized OBD und die Fortschritte bei der Car-to-Car-Kommunikation eingearbeitet. Die Zeitanalyse wurde auf die Transportprotokolle ausgeweitet.

Unverändert geblieben ist unser Grundansatz: Das Buch soll einen Überblick aus Sicht des Anwenders geben, der als Entwickler von Fahrzeugen und Steuergeräten solche Bus-systeme einsetzen oder Automobilsoftware entwickeln will. Dabei stehen die Konzepte und Zusammenhänge im Vordergrund, die sich aus den Standard-Dokumenten allein nur selten direkt erschließen. Wer Protokolle oder Software selbst implementieren muss, kommt am Ende nicht umhin, die eigentlichen Standard-Dokumente im Detail zu studieren, sollte mit diesem Buch aber einen deutlich leichteren Einstieg finden. Das Buch konzentriert sich auf Systemaspekte, Hardwarethemen und EMV wurden weitgehend ausgeklammert.

Im Text werden die Normen und Standardschriften naturgemäß häufig zitiert. Aufgrund der Vielzahl wurde darauf verzichtet, an all diesen Stellen explizit Quellen anzugeben. Vielmehr finden sich am Ende jedes Kapitels Tabellen mit den jeweils einschlägigen Literaturstellen. Hersteller- und Produktnamen werden ohne ausdrückliche Erwähnung von eingetragenen Markennamen und Markenrechten verwendet.

Ein steter Diskussionspunkt bei deutschsprachiger Technikliteratur ist die Verwendung englischsprachiger Begriffe. Da die Originaldokumente praktisch ausschließlich in Englisch verfügbar sind, haben wir bewusst darauf verzichtet, die normspezifischen Fachbegriffe ins Deutsche zu übersetzen. In der Regel wird bei der ersten Erwähnung die deutsche Übersetzung angegeben. Anschließend wird dann der Originalbegriff verwendet, um den *Wiedererkennungswert* beim Lesen der englischsprachigen Originale im Anschluss an dieses Buch zu erhöhen. Aufmerksamen Lesern wird auffallen, dass die Begriffe oft von den in der IT-Welt etablierten Bezeichnungen abweichen. Die Standards der Automobilelektronik sind über viele Jahre gewachsen, wurden häufig von Ingenieuren aus unterschiedlichsten Fachgebieten ohne Rücksicht auf andere Normen erstellt. Während ein IT-Ingenieur kein Verständnisproblem hat, wundert sich der Fahrzeugingenieur möglicherweise, wenn Steuergeräte als Server und Diagnosetester als Client bezeichnet werden und die darauf laufende Software Instanzen von Objekten generiert, die Diagnoseservices aufrufen. Um Zusammenhänge darzustellen, haben wir uns um Durchgängigkeit bemüht, kommen aber letztlich nicht umhin, die Originalbegriffe zu verwenden, auch wenn diese den akademischen Ansprüchen eindeutiger, einheitlicher Bezeichnungen nicht immer genügen.

Dieses Buch geht auf eine Anregung von *Wolfgang Schmid* zurück, ohne dessen beharrliches Drängen wir diese herausfordernde Aufgabe kaum angegangen wären.

Unser besonderer Dank gilt allen Mitautoren an dieser und an früheren Auflagen, *Reinhard Dapper* und *Ewald Schmitt* vom Springer-Vieweg Verlag und ihren Mitarbeitern sowie allen ungenannten Helfern, die zu diesem Buch beigetragen haben.

Den Geschäftsführern *Joachim Tauscher* (Smart In Ovation), *Wolfgang Neu* (Smart Test-solutions), *Ewald Hartmann* (Samtec Automotive Software & Electronics), *Dieter Schaller*, *Hans-Dieter Kübler* (vormals ebenfalls Samtec Automotive), *Thomas Riegraf* (Vector Informatik), *Jörg Supke* (emotive) und *Michael Kirschner* (Bereichsleiter bei Bosch Engineering) sind wir für viele Jahre freundschaftlicher Unterstützung verbunden, in denen wir auf ihr Know-How und das ihrer Mitarbeiter zurückgreifen durften. Dank unseren Kollegen bei der Daimler AG, der Robert Bosch GmbH, dem STZ Rechnereinsatz und der Hochschule Esslingen. Und nicht zuletzt unseren Familien.

Stuttgart, im Frühjahr 2014

Werner Zimmermann, Ralf Schmidgall

Autoren der 5. Auflage

Prof. Dr.-Ing. Werner Zimmermann lehrt Regelungstechnik, Systementwurf, Digital- und Rechnersysteme in der Fakultät Informationstechnik an der Hochschule Esslingen. Zuvor leitete er bei der Robert Bosch GmbH eine Entwicklungsabteilung für Motorsteuergeräte von Dieselfahrzeugen.

Dipl.-Ing. (FH) MSc. Ralf Schmidgall arbeitet in der Entwicklung Fahrzeugdiagnose bei der Daimler AG. Zuvor war er bei einem Hersteller von Testsystemen in der Softwareentwicklung für Kommunikationsprotokolle tätig.

Unter Mitwirkung von:

Oliver Falkner, Produktmanager CANoe/CANalyzer, *Matthias Wernicke*, Produktmanager DaVinci, und *Andreas Patzer*, Produktmanagement Measurement and Calibration bei der Vector Informatik GmbH, verfassten die Abschn. 9.1, 9.2 und 9.3.

Ewald Hartmann und *René Brzezinski*, Geschäftsführer der Samtec Automotive Software & Electronics GmbH, sowie *Sascha Rieuxinger*, Teamleiter Softwareentwicklung, und ihre Mitarbeiter haben den Abschn. 9.5 gestaltet.

Steffen Lang, vormals Entwicklungsingenieur bei der Bose Automotive GmbH, hat wesentlichen Anteil an Abschn. 3.4.

Dr. Kai Richter und *Dr. Marek Jersak*, Geschäftsführer der Symtavision GmbH, sind wir für den Abschn. 9.8 dankbar.

Dr. Jörg Supke, Geschäftsführer der emotive GmbH, zeichnete für Abschn. 9.7 verantwortlich und lieferte einen Beitrag zum Abschn. 6.9.

Joachim Tauscher, Geschäftsführer der Smart In Ovation GmbH, und *Nicole Peters* trugen Abschn. 9.4 zu diesem Buch bei.

Hinweis Sicher enthält auch dieses Buch kleinere und größere Fehler und Ungenauigkeiten. Angesichts der Vielzahl an Protokollen, Spezifikationen und Normen, von denen in der Regel mehrere Varianten, Vorabversionen und Revisionen existieren, ist dies auch bei größter Sorgfalt leider nicht auszuschließen. Für Hinweise und Korrekturvorschläge an <http://www.hs-esslingen.de/~zimmerma/automotive> sind wir dankbar.

Inhaltsverzeichnis

1	Anwendung von Bussystemen und Protokollen	1
1.1	Überblick	2
1.2	Kfz-Bussysteme, Protokolle und Standards	5
1.3	Standardisierung bei Kfz-Bussystemen und Software	7
1.4	Neuere Entwicklungen	9
	Literatur	12
2	Grundkonzepte und einfache Kfz-Bussysteme	13
2.1	Grundlagen	13
2.1.1	Elektrotechnische Grundlagen	13
2.1.2	Topologie und Kopplung von Bussystemen	17
2.1.3	Botschaften, Protokollstapel, Dienste (Services)	18
2.1.4	Kommunikationsmodelle, Adressierung	20
2.1.5	Zeichen- und Bitstrom-basierte Übertragung, Nutzdatenrate	25
2.1.6	Buszugriffsverfahren, Fehlererkennung und Fehlerkorrektur	27
2.1.7	Jitter und Latenz bei der Datenübertragung	29
2.1.8	Elektrik/Elektronik-(E/E)-Architekturen	30
2.2	K-Line nach ISO 9141 und ISO 14230	32
2.2.1	Entwicklung von K-Line und KWP 2000	33
2.2.2	K-Line Bus-Topologie und Physical Layer	34
2.2.3	Data Link Layer	36
2.2.4	Einschränkungen für emissionsrelevante Komponenten (OBD)	40
2.2.5	Schnittstelle zwischen Software und Kommunikations-Controller	40
2.2.6	Ältere K-Line-Varianten	41
2.2.7	Zusammenfassung K-Line – Layer 1 und 2	41
2.3	SAE J1850	42
2.4	Sensor-Aktor-Bussysteme	45
2.4.1	SENT – Single Edge Nibble Transmission nach SAE J2716	45
2.4.2	PSI 5 – Peripheral Sensor Interface 5	46
2.4.3	ASRB 2.0 – Automotive Safety Restraint Bus (ISO 22898)	49
2.4.4	DSI – Distributed Systems Interface	51

2.5	Normen und Standards zu Kap. 2	54
	Literatur	55
3	Kfz-Bussysteme – Physical und Data Link Layer	57
3.1	Controller Area Network CAN nach ISO 11898	57
3.1.1	Entwicklung von CAN	57
3.1.2	Bus-Topologie und Physical Layer	58
3.1.3	CAN Data Link Layer	61
3.1.4	Fehlerbehandlung	63
3.1.5	Einsatz von CAN – Höhere Protokolle	63
3.1.6	Schnittstelle zwischen Protokoll-Software und CAN-Controller	64
3.1.7	Zeitverhalten von CAN-Systemen, Wahl der Botschaftspriorität	67
3.1.8	Time-Triggered-CAN (TTCAN) – Deterministischer Buszugriff	72
3.1.9	Energiesparmaßnahmen: Wakeup und Partial Networking	75
3.1.10	Höhere Datenraten: CAN Flexible Data-Rate CAN FD	76
3.1.11	Zusammenfassung CAN – Layer 1 und 2	78
3.2	Local Interconnect Network LIN	79
3.2.1	Überblick	79
3.2.2	Data Link Layer	81
3.2.3	Neue Botschaftstypen bei LIN V2.0	84
3.2.4	LIN Transportschicht und ISO Diagnose über LIN	85
3.2.5	LIN Configuration Language	86
3.2.6	Dynamische Konfiguration von LIN-Slave-Steuergeräten	90
3.2.7	LIN Application Programming Interface (API)	92
3.2.8	Zeitverhalten von LIN-Systemen	94
3.2.9	Zusammenfassung LIN – Layer 1 und 2	95
3.3	FlexRay	96
3.3.1	Bus-Topologie und Physical Layer	97
3.3.2	Data Link Layer	99
3.3.3	Netzwerk-Start und Takt-Synchronisation	102
3.3.4	Fehlerbehandlung, Bus Guardian	105
3.3.5	Konfiguration und übergeordnete Protokolle	106
3.3.6	Zeitverhalten von FlexRay-Systemen, Beispiel-Konfiguration	107
3.3.7	Schnittstelle zum FlexRay-Controller	112
3.3.8	Weiterentwicklung FlexRay 3.x	116
3.3.9	Zusammenfassung FlexRay – Layer 1 und 2	118
3.4	Media Oriented Systems Transport MOST	119
3.4.1	Bus-Topologie und Physical Layer	121
3.4.2	Data Link Layer	121
3.4.3	Kommunikationscontroller	128
3.4.4	Network Services und Funktionsblöcke	129
3.4.5	Netzmanagement	133

3.4.6	Höhere Protokollsichten	135
3.4.7	Beispiel für Systemstart und Audioverbindung	135
3.4.8	Zusammenfassung MOST	138
3.5	Automotive Ethernet	138
3.5.1	Ethernet nach IEEE 802.3	138
3.5.2	Autotauglicher Physical Layer BroadR-Reach	141
3.5.3	Echtzeitfähigkeit mit IEEE 802.1 Audio-Video-Bridging AVB	142
3.5.4	Höhere Protokollsichten IP, TCP und UDP	145
3.6	Normen und Standards zu Kapitel 3	149
	Literatur	150
4	Transportprotokolle	153
4.1	Transportprotokoll ISO TP für CAN nach ISO 15765-2	154
4.1.1	Botschaftsaufbau	154
4.1.2	Flussteuerung, Zeitüberwachung und Fehlerbehandlung	156
4.1.3	Dienste für die Anwendungsschicht (Application Layer Services)	157
4.1.4	Protokoll-Erweiterungen	159
4.1.5	Adressierung bei KWP 2000/UDS – Zuordnung von CAN Identifiern	159
4.1.6	Bandbreite des ISO TP für CAN	159
4.2	Transportprotokoll für FlexRay nach ISO 10681-2	162
4.2.1	Botschaftsaufbau und Adressierung	162
4.2.2	Verbindungsarten und Übertragungsablauf	163
4.2.3	Bandbreitensteuerung	165
4.2.4	Fehlerbehandlung und Implementierungshinweise	166
4.2.5	Bandbreite des FlexRay Transportprotokolls	166
4.3	Transportprotokoll TP 2.0 für CAN	169
4.3.1	Adressierung und CAN Message Identifier	170
4.3.2	Broadcast-Botschaften	170
4.3.3	Dynamischer Kanalaufbau und Verbindungsmanagement	171
4.3.4	Datenübertragung	174
4.4	Transportprotokoll TP 1.6 für CAN	175
4.4.1	Botschaftsaufbau	176
4.4.2	Dynamischer Kanalaufbau	176
4.4.3	Datenübertragung und Datenrichtungswechsel	177
4.5	Transportprotokoll SAE J1939/21 für CAN	178
4.5.1	Übertragungsarten, Adressierung und CAN Message Identifier	179
4.5.2	Segmentierte Datenübertragung (Multi Packet)	182
4.6	Transportprotokoll DoIP nach ISO 13400	183
4.7	Transportprotokoll für CAN FD	187
4.8	Normen und Standards zu Kapitel 4	187
	Literatur	188

5	Diagnoseprotokolle – Application Layer	189
5.1	Diagnoseprotokoll KWP 2000 (ISO 14230-3)	192
5.1.1	Überblick	192
5.1.2	Diagnosesitzungen (Diagnostic Management)	194
5.1.3	Adressierung der Steuergeräte nach KWP 2000 und UDS	197
5.1.4	Bussystem-abhängige Dienste (Network Layer Protocol Control)	199
5.1.5	Fehlerspeicher lesen und löschen (Stored Data Transmission)	200
5.1.6	Daten lesen und schreiben (Data Transmission), Ansteuern von Steuergeräte-Ein- und Ausgängen (Input/Output Control)	200
5.1.7	Speicherblöcke auslesen und speichern (Upload, Download)	202
5.1.8	Start von Programmen im Steuergerät (Remote Routine Activation)	202
5.1.9	Erweiterte Dienste (Extended Services)	203
5.2	Unified Diagnostic Services UDS nach ISO 14229/15765-3	203
5.2.1	Unterschiede zum KWP 2000 Diagnoseprotokoll	204
5.2.2	Überblick über die UDS-Diagnosedienste	204
5.2.3	Response on Event Dienst	208
5.3	On-Board-Diagnose OBD nach ISO 15031/SAE J1979	212
5.3.1	Überblick OBD-Diagnosedienste	213
5.3.2	Auslesen des Fehlerspeichers und von Steuergerätewerten	214
5.3.3	Abfrage der Testergebnisse für abgasrelevante Komponenten	217
5.3.4	OBD-Fehlercodes	218
5.3.5	Data Link Security	221
5.3.6	Pass-Through-Programmierung	221
5.3.7	Beispiel	222
5.4	Weiterentwicklung der Diagnose	224
5.4.1	World-Wide Harmonized On-Board Diagnose nach ISO 27145	225
5.5	Normen und Standards zu Kapitel 5	227
	Literatur	228
6	Anwendungen für Messen, Kalibrieren und Diagnose (ASAM AE MCD)	229
6.1	Einführung	229
6.2	Busprotokolle für Aufgaben in der Applikation (ASAM AE MCD 1MC)	232
6.2.1	CAN Calibration Protocol CCP	234
6.2.2	Extended Calibration Protocol XCP	241
6.2.3	AML-Konfigurationsdateien für XCP und CCP	254
6.2.4	Interface zwischen Bus und Applikationssystem ASAM MCD 1b	256
6.3	Field Bus Exchange Format FIBEX	261
6.4	Überblick über ASAM AE MCD 2 und MCD 3	270
6.5	Applikationsdatensätze nach ASAM MCD 2 MC	272
6.5.1	ASAP2/A2 L-Applikationsdatensätze	272
6.5.2	Calibration Data Format CDF und Meta Data Exchange MDX	275
6.6	ODX-Diagnosedatensätze nach ASAM AE MCD 2D	277

6.6.1	Aufbau des ODX-Datenmodells	278
6.6.2	DIAG-LAYER: Hierarchische Diagnosebeschreibung	280
6.6.3	VEHICLE-INFO-SPEC: Fahrzeugzugang und Bustopologie	283
6.6.4	COMPARAM-SPEC und COMPARAM-SUBSET: Busprotokoll	286
6.6.5	DIAG-COMM und DIAG-SERVICE: Diagnosiedienste	288
6.6.6	Einfache und komplexe Datenobjekte	292
6.6.7	SINGLE-ECU-JOB und MULTIPLE-ECU-JOB: Diagnoseabläufe	302
6.6.8	STATE-CHART: Diagnosesitzungen	304
6.6.9	ECU-CONFIG: Beschreibung der Steuergeräte-Konfiguration	305
6.6.10	ECU-MEM: Beschreibung der Flash-Programmierung	306
6.6.11	FUNCTION-DICTIONARY: Funktionsorientierte Diagnose	308
6.6.12	Packaged ODX und ODX-Autorenwerkzeuge	309
6.6.13	ODX Version 2.2 und ISO 22901	309
6.7	ASAM AE MCD 3-Server	310
6.7.1	Funktionsgruppe M – Messen	312
6.7.2	Funktionsgruppe C – Kalibrieren	313
6.7.3	Funktionsgruppe D – Diagnose	314
6.8	MVCI-Schnittstelle für Diagnosetester nach ISO 22900	316
6.9	OTX-Beschreibung von Testabläufen nach ISO 13209	319
6.9.1	Grundkonzepte und Aufbau von OTX	319
6.9.2	OTX-Core-Datenmodell	321
6.9.3	OTX-Core-Programmelemente	322
6.9.4	OTX-Erweiterungen	323
6.10	Normen und Standards zu Kapitel 6	328
	Literatur	329
7	Software-Standards: OSEK und HIS	331
7.1	Einführung	331
7.2	OSEK/VDX	334
7.2.1	Ereignisgesteuerter Betriebssystemkern OSEK/VDX OS	336
7.2.2	Kommunikation in OSEK/VDX COM	346
7.2.3	Netzmanagement mit OSEK/VDX NM	350
7.2.4	Zeitgesteuerter Betriebssystemkern OSEK Time, Fehlertoleranz OSEK FTCOM und Schutzmechanismen Protected OSEK	355
7.2.5	Scheduling, Taskprioritäten und Zeitverhalten bei OSEK OS und AUTOSAR OS	358
7.3	Hardware-Ein- und Ausgabe (HIS IO Library, IO Driver)	361
7.4	HIS Hardwaretreiber für CAN-Kommunikationscontroller (HIS CAN Driver)	363
7.5	HIS Flash-Lader	363
7.6	Normen und Standards zu Kapitel 7	364
	Literatur	365

8	AUTOSAR-Softwarearchitektur für Kfz-Systeme	367
8.1	Einführung	367
8.2	Überblick über die AUTOSAR-Basissoftware	369
8.2.1	Funktionale Sicherheit	380
8.3	Betriebssystem AUTOSAR OS	381
8.4	Kommunikationsstack AUTOSAR COM, Diagnose DCM	385
8.5	Netzmanagement AUTOSAR NM	397
8.6	Virtual Function Bus VFB, Runtime Environment RTE und Softwarekomponenten	403
8.7	Beispiel einer einfachen Anwendung	408
8.8	Ausblick	410
8.9	Normen und Standards zu Kapitel 8	412
	Literatur	413
9	Werkzeuge, Anwendungen und Einsatzgebiete	415
9.1	Entwurf und Test der On-Board-Kommunikation	415
9.1.1	Entwicklungsprozess mit <i>CANoe</i> von Vector Informatik	415
9.1.2	Netzwerkdesign mit dem Network Designer	416
9.1.3	Simulation des Gesamtsystems in <i>CANoe</i>	420
9.1.4	Restbussimulation als Entwicklungsumgebung für Steuergeräte	422
9.1.5	Integration des Gesamtsystems	424
9.2	System- und Softwareentwurf für Steuergeräte	424
9.2.1	Systementwurf mit <i>PREEvision</i> von Vector Informatik	426
9.2.2	Entwicklung der Anwendungssoftware im AUTOSAR-Prozess	427
9.2.3	Systemtest und Applikation	428
9.3	Werkzeuge zur Applikation von Steuergeräten	429
9.3.1	Steuergeräte-Applikation mit <i>CANape</i> von Vector Informatik	430
9.4	Flash-Programmierung von Steuergeräten	433
9.4.1	Rahmenbedingungen	434
9.4.2	Flash-Speicher	437
9.4.3	Flash-Programmierprozess	440
9.4.4	Beispiel eines Flash-Laders: <i>ADLATUS</i> von SMART IN OVATION	448
9.4.5	Softwaretest von Flash-Ladern und Busprotokollen	453
9.5	Diagnosewerkzeuge in Entwicklung und Fertigung	457
9.5.1	Beispiel für Diagnosewerkzeuge: <i>samDia</i> von Samtec Automotive	459
9.6	Autorenwerkzeuge für Diagnosedaten	469
9.7	Diagnose-Laufzeitsysteme und OTX Diagnose-Sequenzen	471
9.7.1	Open Test Framework von <i>emotive</i> als OTX Werkzeug	473
9.8	Echtzeitverhalten der Steuergeräte-Kommunikation	478
9.8.1	Kennwerte für das Echtzeitverhalten	478
9.8.2	Echtzeitanalyse mit <i>SymTA/S</i> von Syntavision	480
	Literatur	482

10 Kommunikation zwischen Fahrzeugen	483
10.1 Mautsysteme	483
10.2 Car2Car-Konsortium und Vehicle2X-Kommunikation	484
10.3 Normen und Standards zu Kapitel 10	488
Literatur	489
Web-Adressen	491
Abkürzungen	495
Sachverzeichnis	501

Bereits bei Einführung der ersten mikroprozessorgesteuerten Systeme wie Motronic und ABS im Kraftfahrzeug ab 1980 wurden Daten zwischen diesen Systemen und nach außen ausgetauscht. Während zwischen den Systemen (On-Board) zunächst vor allem Punkt-zu-Punkt-Verbindungen mit Analogsignalen oder einfachen Schaltsignalen verwendet wurden, diente die erste echte Datenkommunikation zum Anschluss des Diagnosetesters in der Kfz-Werkstatt (Off-Board). Sehr schnell war eine herstellerübergreifende Lösung nötig, wobei Bosch als in Europa führender Hersteller elektronischer Steuergeräte eine von vielen Fahrzeugherstellern übernommene Spezifikation vorlegte, die später als ISO 9141 standardisiert wurde. Diese Spezifikation legte zunächst wenig mehr als die Zahl der Verbindungsleitungen, die elektrischen Signalpegel und das Bitformat der Zeichenübertragung fest. Die Bedeutung der übertragenen Daten sowie die in der Werkstatt angewendeten Diagnoseverfahren selbst blieben offen und wurden weiterhin Hersteller-spezifisch implementiert.

Mit der Einführung des von Bosch vorgestellten, später als ISO 11898 und SAE J1939 standardisierten CAN Busses ab 1990 hielt auch bei der On-Board-Kommunikation zwischen den Steuergeräten innerhalb des Fahrzeugs ein Datennetz (*Bussystem*) Einzug. Auch hier war zunächst im Wesentlichen die Bitbene spezifiziert, während die Bedeutung der ausgetauschten Daten (*Protokoll*) nicht festgelegt war und immer noch je nach Gerät, Fahrzeug oder Hersteller unterschiedlich implementiert wurde. Mit dem Siegeszug der Mikroelektronik in modernen Fahrzeugen wurde das System derart komplex und das Datenaufkommen so hoch, dass Neufahrzeuge heute über mehrere, miteinander vernetzte Bussysteme (*Netzwerke*) verfügen (Abb. 1.1). Die Beherrschung dieser Komplexität, der Kostendruck, der Wunsch, Fahrzeuge weltweit anzubieten, sowie Vorschriften der Gesetzgeber zwangen die Fahrzeughersteller und ihre Zulieferer schließlich, nach standardisierten Lösungen für die Bussysteme und die zum Datenaustausch verwendeten Protokolle zu suchen [1–6].

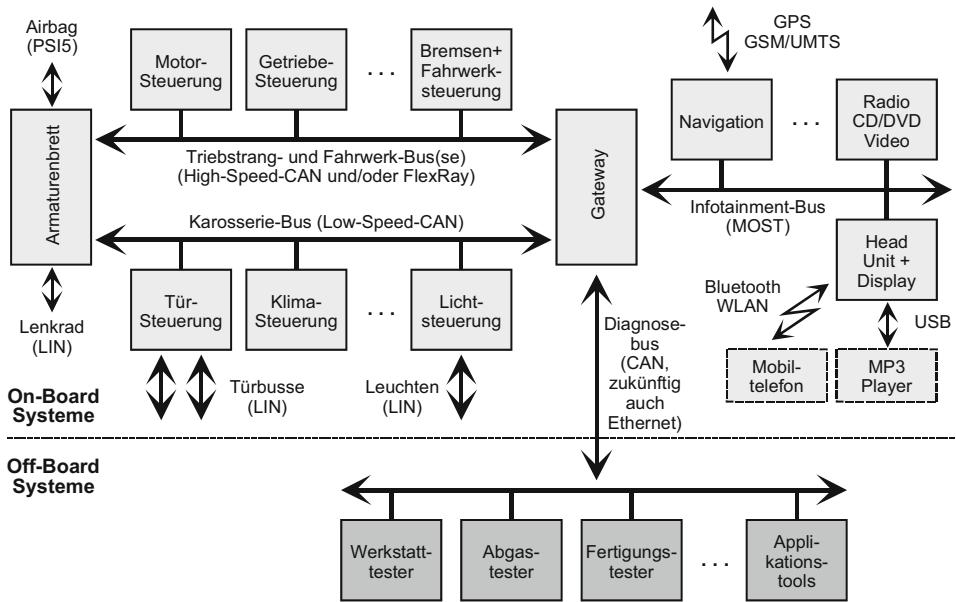


Abb. 1.1 Bussysteme eines modernen Fahrzeugs

1.1 Überblick

Grob lassen sich dabei folgende Anwendungsgebiete unterscheiden (Tab. 1.1):

- **On-Board-Kommunikation** zwischen den Kfz-Steuergeräten im Fahrzeug
Dieses Aufgabenfeld lässt sich heute in Teilgebiete unterteilen, für die in der Regel unterschiedliche Buskonzepte verwendet werden:
 - **High-Speed-Systeme für Echtzeit-Steuerungsaufgaben**
Die Steuerung und Regelung von Motor, Getriebe, Bremsen und Fahrwerk ist nur möglich, wenn die Steuergeräte Sensorinformationen austauschen und Stelleingriffe über mehrere Systeme hinweg koordinieren. Die notwendigen Informationen sind nur wenige Bytes lang, müssen aber periodisch mit hoher Frequenz, kurzer Verzögerungszeit (Latenz) und großer Zuverlässigkeit übertragen werden. Für diese Anwendungen wurde CAN entworfen. Durch moderne Fahrwerkssteuerungen steigen die Anforderungen. Daher wurde CAN zu CAN FD weiterentwickelt und neue Bus-systeme wie FlexRay konzipiert.
 - **Low-Speed-Systeme zur Kabelbaum-Vereinfachung**
Auch für simplere Aufgaben wie das Ansteuern von Lampen und Fenstermotoren werden Bussysteme eingeführt, um den Kabelbaum zu vereinfachen. Da vor allem Schaltsignale übertragen werden, sind die erforderlichen Datenraten niedrig. Haupt-gesichtspunkt sind die Kosten je Busknoten. Für diese Aufgaben wird eine verein-

Tab.1.1 Anwendungsbereiche und Anforderungen an Bussysteme im Kfz

Anwendung	Botschafts-länge	Botschafts-rate	Resultierende Datenrate	Latenz-zeiten	Fehler-sicherheit	Kosten
On-Board-Kommunikation						
High-Speed-Steuering	Kurz	Hoch	Hoch	Sehr kurz	Extrem hoch	Mittel
Low-Speed-Steuering	Kurz	Niedrig	Niedrig	Mäßig	Sehr hoch	Sehr niedrig
Infotainment	Lang	Hoch	Sehr hoch	Mäßig	Mäßig	Hoch
Assistenz	Mischung von Steuerungs- und Infotainment-Anforderungen					
Off-Board-Kommunikation						
Werkstattdiagnose	Kurz	Niedrig	Niedrig	Unwichtig	Gering	Niedrig
Fertigungstest beim Hersteller inkl. End-of-Line-Programming (<i>Flashen</i>)	Lang bis sehr lang	Niedrig	Hoch	Unwichtig	Mäßig	Unwichtig
Applikation am Fahrzeug und an Prüfständen in der Entwicklung (<i>Umprogrammieren, Kalibrieren</i>)	Kurz	Mittel bis hoch bei Messaufgaben	Mittel bis hoch	Kurz	Gering	Unwichtig
Car to X	Mittel	Mittel	Hoch	Mittel	Hoch	Hoch

fachte CAN-Variante ergänzt durch das noch kostengünstigere Bussystem LIN eingesetzt. Daneben entstanden spezielle Busse wie PSI5 oder Safe-by-Wire für die Vernetzung von Rückhaltesystemen.

- **Infotainment-Systeme**

Mit der Einführung von Infotainment-Systemen im Fahrzeug für Navigation, Video, Audio und Telefon müssen sehr große Datenmengen beispielsweise zwischen dem DVD-Wechsler, dem verteilten Audio-System und dem Display im Armaturenbrett ausgetauscht werden. Dabei ist der Hauptgesichtspunkt ein hoher Datendurchsatz, während die Anforderungen an die Latenzzeiten und die Übertragungszuverlässigkeit wesentlich geringer sind als bei den High-Speed-Systemen für Echtzeit-Steuerungsaufgaben. Die Anbindung an die Außenwelt und die Einbindung von Geräten der Unterhaltungselektronik erfolgt über Funkschnittstellen wie GSM/UMTS/LTE und GPS oder Consumer-Schnittstellen wie USB bzw. Bluetooth.

- **Fahrerassistenz-Systeme**

Diese Systeme verbinden Merkmale aus dem Infotainment-Bereich wie Videoübertragung mit Merkmalen von Echtzeit-Steuerungsaufgaben.

- **Off-Board-Kommunikation** zwischen Fahrzeug und externen Geräten:

- **Diagnose-Kommunikation in der Werkstatt und beim Abgastest**

Für die Fehlersuche und für Abgasprüfungen in Werkstätten ist eine Kommunikationsschnittstelle zu einem externen Tester notwendig. Die Anforderungen an Datenrate und Fehlertoleranz sind relativ gering. Die Schnittstelle ist teilweise vom Gesetzgeber standardisiert (US OBD, European OBD), darüber hinaus hat jeder Kfz-Hersteller seinen Hausstandard. Veränderungen an dieser Schnittstelle verursachen einen hohen Folgeaufwand in den Werkstätten, da die Werkstatttester weltweit angepasst werden müssen. Steuergeräte unterstützen daher in der Regel verschiedene herstellerspezifische Varianten des Diagnoseprotokolls, wobei sich Steuergerät und Tester automatisch auf eine zu beiden kompatible Protokollvariante verständigen müssen. Zunehmend wird erwartet, dass die Steuergeräte eines Fahrzeugs in der Werkstatt umprogrammiert werden können. Dabei sind höhere Datenraten sowie Zugriffsschutzmechanismen wichtig, um sicherzustellen, dass nur die zum Fahrzeug passende, vom Fahrzeughersteller freigegebene Softwarevariante verwendet und die Umprogrammierung nur durch autorisierte Werkstätten durchgeführt wird.

- **Fertigungstest beim Kfz- und beim Steuergeräte-Hersteller**

Kfz-Hersteller versuchen, mit wenigen Steuergeräte-Hardwarevarianten viele Fahrzeugmodelle mit Ausstattungsvarianten und Länderausführungen durch Softwarevarianten im Steuergerät abzudecken. Um die Logistik zu vereinfachen, wird häufig die gesamte oder ein großer Teil der Software erst beim Kfz-Hersteller ins Steuergerät geladen (*Flashen*). Für den Programmiervorgang sowie für die Fertigungsprüfung des Fahrzeugs wird ebenfalls die Diagnose-Schnittstelle eingesetzt. Gelegentlich wird diese dabei in einem Modus betrieben, der einen höheren Datendurchsatz zulässt, um kürzere Zykluszeiten zu erreichen. Da die Werkstatt im Prinzip dieselbe Diagnoseschnittstelle benutzt, sind Zugriffsschutzmechanismen notwendig, da-

Tab. 1.2 Klassifikation der Bussysteme nach der Bitrate

Class	Bitrate	Typische Vertreter	Anwendung
Diagnose	< 10 Kbit/s	ISO 9141-K-Line	Werkstatt- und Abgastester
A	< 25 Kbit/s	LIN, SAE J1587/1707	
B	25 ... 125 Kbit/s	CAN (Low Speed)	Karosserieelektronik
C	125 ... 1000 Kbit/s	CAN (High Speed)	Antriebsstrang, Fahrwerk, Diagnose
D	> 1 Mbit/s	FlexRay, TTP, Ethernet	X by Wire, Backbone-Netz
Infotainment	> 10 Mbit/s	MOST, Ethernet	Multimedia (Audio, Video)

mit sicherheitskritische Daten des Steuergeräts nicht von Unbefugten manipuliert werden.

– **Applikation am Prüfstand und im Fahrzeug in der Entwicklungsphase**

Während der Anpassung eines Steuergerätes an ein bestimmtes Fahrzeug ist die Durchführung umfangreicher Messungen an Komponentenprüfständen und im Fahrzeug notwendig. Dabei müssen teilweise mehrere tausend Parameter im Steuergerät appliziert werden. Da die Steuergeräte-Hard- und Software bereits in einem seriennahen Zustand sein sollen, erfolgt die Datenerfassung und die Adaption der Parameter in der Regel über die Standardschnittstellen für die On- und Off-Board-Kommunikation, wobei die Schnittstellenprotokolle für diesen Zweck erweitert werden. Da der Applikationsaufwand bei praktisch jeder Fahrzeugvariante neu anfällt, wird inzwischen herstellerübergreifend versucht, die Applikation mit Werkzeugen zu unterstützen und zu automatisieren. Dazu wurden Standard-Schnittstellen wie ASAM MCD für den Messdatenaustausch, die Parameterverstellung und die Datenthaltung geschaffen.

– **Car to Car und Car to Infrastructure (Car to X)**

Schon heute kommunizieren Fahrzeuge etwa in Mautsystemen automatisch mit Verkehrszähl- und Leiteinrichtungen (*Car to Infrastructure*). Diese Systeme werden erweitert, um den Verkehrsfluss besser steuern und durch Kommunikation zwischen den Autos (*Car to Car*) Unfälle vermeiden zu helfen.

Aus dem dargestellten Anforderungsprofil insbesondere bezüglich der Übertragungsgeschwindigkeit ergibt sich eine Einteilung der Bussysteme in verschiedene Klassen (Tab. 1.2), wobei die Grenzen zwischen den Klassen natürlich fließend sind.

1.2 Kfz-Bussysteme, Protokolle und Standards

Wenn von Kfz-Bussystemen gesprochen wird, fallen in der Regel Schlagworte wie CAN, LIN, FlexRay usw. Neulinge identifizieren mit diesen Begriffen hauptsächlich den physikalisch direkt sichtbaren Teil der Kommunikationsschnittstelle mit Steckern, Kabeln und den

Tab. 1.3 OSI-Schichtenmodell für Bussysteme und Protokolle (Datennetze), Hinweis: Schicht 0 ist keine offizielle Schicht des OSI-Modells

Layer (Schicht)		Aufgabe
7 Application	Anwendung	Allgemein verwendbare Dienste für den Anwender, z. B. Lesen des Fehlerspeichers usw.
6 Präsentation	Darstellung	
5 Session	Sitzungssteuerung	
4 Transport	Datentransport	Aufteilung und Zusammensetzen der Daten mehrerer Botschaften (Segmentierung)
3 Network	Vermittlung	Routing, Adressvergabe, Teilnehmererkennung und -überwachung
2 Data Link	Sicherung	Botschaftsaufbau, Buszugriff, Fehlersicherung, Flusskontrolle
1 Physical	Bitübertragung	Elektrische Signalpegel, Bitcodierung
0 Mechanical	Mechanik	Steckverbinder und Kabel

für die Kommunikation zuständigen Elektronikschaltungen. In Veröffentlichungen werden in diesem Zusammenhang vor allem übertragungstechnische Details auf Bitebene wie Signalpegel, Zugriffsverfahren und die Reihenfolge und Bedeutung der einzelnen Bits auf den Busleitungen ausführlich beschrieben. Die Softwareentwickler, die die Übertragungssoftware implementieren müssen, interessieren sich vor allem für die Programmierschnittstelle der eingesetzten *Buscontroller*, Fragen der Formatierung und Zwischenspeicherung der Daten sowie der Behandlung von Übertragungsfehlern. Für den Anwender dagegen sind vor allem die Bedeutung der übertragenen, eigentlichen Nutzdaten und deren Formate von Interesse, die durch die typischen Spezifikationen von Bussystemen wie CAN, LIN oder FlexRay überhaupt nicht definiert sind.

Um die verschiedenen Aufgabenstellungen bei der Datenkommunikation auseinander zu halten, kann das von der ISO standardisierte Open-System-Interconnect-(OSI)-Schichtenmodell (Tab. 1.3) dienen, das die Kommunikationshierarchie beschreibt. Die grau markierten Schichten haben für Kfz-Anwendungen (noch) keine Bedeutung.

Vor dem Spiegel realer Standards betrachtet ist das OSI-Modell allerdings eher akademischer Natur. Es hilft zwar beim Verständnis, aber die realen Standards definieren häufig nur Teile einer Schicht, fassen Aufgaben mehrerer Schichten in einer Ebene zusammen oder trennen eine Schicht auf mehrere Standards auf. Dazu kommt, dass für dieselbe Aufgabenstellung oft mehrere, voneinander abweichende Standards existieren oder umgekehrt dieselbe technische Lösung in verschiedenen Standards beschrieben wird. Noch unübersichtlicher wird die Situation dadurch, dass die unterschiedlichen Normen, selbst wenn sie

vom selben Normungsgremium stammen, Begriffe unterschiedlich, gar nicht oder sogar unterschiedliche Dinge gleich benennen.

Die weithin bekannten Kfz-Bussysteme wie CAN, LIN oder FlexRay (Tab. 1.4) legen in der Regel lediglich die Schichten 0 bis 2 fest, wobei meist sogar nur ein Ausschnitt exakt spezifiziert wird. So lässt etwa die von Bosch veröffentlichte Grundspezifikation CAN 2.0 A/B die mechanischen Aspekte (Stecker, Kabel) völlig außen vor und beschreibt die Schaltung zur Busankopplung und die Signalpegel nur beispielhaft ohne exakte Vorgaben. Die darauf aufsetzende ISO 11898 übernimmt die Bosch-Spezifikation für Schicht 2 weitgehend und liefert für Schicht 1 in nachträglich hinzugekommenen Anhängen Spezifikationsvorschläge nach. Der für die Kfz-Diagnoseschnittstelle grundlegende Standard ISO 9141 dagegen spezifiziert zunächst nur die Schichten 0 und 1 und lässt selbst dabei noch eine Reihe von nur bedingt kompatiblen Varianten zu. Erst spätere Ergänzungen beschreiben Teile der Schicht 2.

Für die höheren Schichten existieren erst seit jüngster Zeit Standards, z. B. wird die Schicht 7 für die abgasrelevanten Teile der Kfz-Diagnose durch den Gesetzgeber in ISO 15031 definiert. Dabei bleibt zunächst offen, ob auf den unteren Schichten die klassische ISO 9141-Schnittstelle oder ein CAN-Bus zum Einsatz kommt. In zusätzlichen Ergänzungen des Standards mit ISO 14230 bzw. ISO 15765 dagegen werden dann Spezifikationen nachgeliefert, die auf K-Line- oder CAN-Spezifika abheben, obwohl es sich eigentlich um eine Schicht 7 Spezifikation handelt.

Da die unteren Schichten immer von den höheren, die höheren Schichten oft, aber praktisch nie vollständig von den unteren Schichten unabhängig sind, wird in den folgenden Kapiteln die Darstellung aufgespalten in die großen Bereiche:

- **Kfz-Busse** (Schicht 0 bis 2), werden in Kap. 2 und 3 beschrieben.
- **Transportprotokolle** (Schicht 3 und 4), werden in Kap. 4 dargestellt.
- **Anwendungsprotokolle** (Schicht 5 bis 7), werden in Kap. 5 und 6 erläutert.
- **Softwarestandards** und **Anwendungen** werden in Kap. 7 bis 10 vorgestellt.

1.3 Standardisierung bei Kfz-Bussystemen und Software

Die Tab. 1.5, 1.6 und 1.7 geben einen Überblick über die im Kfz-Bereich relevanten Standards. Bussysteme mit eingeschränkter Marktbedeutung (z. B. VAN, CCD, D2B, TTP), Busse, deren Hersteller erklärt haben, das Bussystem nicht weiterzuentwickeln oder ablösen zu wollen (z. B. ABUS, Byteflight), und Bussysteme, die nur für sehr spezielle Anwendungen, z. B. die Vernetzung von Airbag-Steuergeräten, eingesetzt werden, wurden nicht aufgeführt.

Tab.1.4 Kfz-Busse (Schicht 0 bis 2), Web-Links zu den Herstellern im Anhang

Bustyp	Anwendung	Europ. Standards	US-Standards
<i>Zeichen-basiert (UART)</i>			
K-Line	Diagnose	ISO 9141	
SAE J1708	Diagnose, Class A On-Board		SAE J1708 (Truck and Bus) 9,6 kbit/s
LIN	Class A On-Board		SAE J2602
<i>PWM-basiert</i>			
SAE J1850	Diagnose, Class A/B On-Board		SAE J1850 (PWM Ford, VPWM GM, Chrysler) 10,4 und 41,6 kbit/s
<i>Bitstrom-basiert</i>			
CAN	Class B/C On-Board, Diagnose	ISO 11898 Bosch CAN 2.0 A, B 47,6 ... 500 kbit/s	SAE J2284 (Passenger Cars) 500 kbit/s
CAN FD		ISO 11992	SAE J1939 (Truck and Bus)
TTICAN		CAN für Zugfahrzeuge und Anhänger	250 kbit/s
		ISO 11783 ISOBUS	500 kbit/s
		CAN für Landmaschinen (Basis J1939)	
FlexRay	Class D On-Board		Herstellerkonsortium, ISO 17458 , 10 Mbit/s
TTP	Class D On-Board		Herstellerkonsortium
MOST	Infotainment		Herstellerkonsortium 25, 50, 150 Mbit/s
Ethernet	Diagnose Flash Programmieren		IEEE 802.3, ISO 13400 10/100 Mbit/s
<i>Diverse</i>			
PSI5, SENT ASRB, DSI	Class A/B On-Board Sensor-Aktor-Bus		Herstellerkonsortien

Tab. 1.5 Transportprotokolle (Schicht 4)

Transportprotokoll	Anwendung	Europ. Standards	US-Standards
ISO TP	CAN	ISO 15765-2	
FlexRay TP	FlexRay	ISO 10681-2	
SAE J1939	CAN		SAE J1939/21
TP 1.6, TP 2.0	CAN	Hausstandard VW/Audi/Seat/Skoda	
DoIP	Ethernet	ISO 13400-2	

Tab. 1.6 Anwendungsprotokolle (Schicht 7)

Protokoll	Anwendung	Europ. Standards	US-Standards
ISO 9141-CARB	Diagnose US OBD	ISO 9141-2 Veraltete US-Diagnoseschnittstelle	SAE J1979, J2190
KWP 2000 Keyword Protocol	Diagnose (allgemein und OBD)	ISO 14230 KWP 2000 Diagnostics on K-Line	
UDS Unified Diagnostic Services	Diagnose (allgemein und OBD)	ISO 14229 UDS Unified Diagnostic Services ISO 15765 UDS on CAN	
OBD	Diagnose US OBD European OBD	ISO 15031 (identisch mit den US-Standards)	SAE J1930, J1962, J1978, J1979, J2012, J2186
CCP Can Calibration Protocol	Applikation	ASAM AE MCD ASAM-Konsortium Automotive Electronics	
XCP Extended Calibration Protocol		Measurement, Calibration and Diagnostics	

1.4 Neuere Entwicklungen

Nachdem der Beginn des Jahrhunderts zunächst mit LIN, FlexRay und MOST eine Vielfalt neuer Bussysteme im Fahrzeug gebracht hat, setzte sich allmählich die Erkenntnis durch, dass die heterogene Vielfalt letztlich aufwendig und nur schwer zu beherrschen ist. Die Wirtschaftskrise im Jahr 2009 hat diese Überlegungen sicher noch beschleunigt. Für diese Vielfalt gab es wohl drei Hauptursachen, die überwunden werden müssen, bevor eine echte Konsolidierung möglich ist:

- Im Automobilbereich gibt es ein ausgeprägtes Kostenbewusstsein, das sich allerdings stark auf die Stückkosten konzentriert. Der Engineeringaufwand dagegen wird im Hinblick auf die hohen Stückzahlen oft wenig beachtet, es sei denn er verursacht zu lange

Tab. 1.7 Standards für die Implementierung von Anwendungen mit Bussystemen

Standard	Anwendung	Europ. Standards
OSEK/VDX	Betriebssystem	ISO 17356-3 OSEK OS
	Kommunikation	ISO 17356-4 OSEK COM
	Netzmanagement	ISO 17356-5 OSEK NM
ASAM AE MCD	Applikation	Standardisierte Mess-, Kalibrier- und Diagnosewerkzeuge mit den wichtigen Teilstandards ODX/OTX für Diagnosedaten und -tests FIBEX Datenformat für die Beschreibung der Buskommunikation
HIS	Flashen Hardwaretreiber	Hersteller Initiative Software HIS Softwaremodule für Steuergeräte
AUTOSAR	Softwarearchitektur	Softwarestruktur zukünftiger Steuergeräte inklusive AUTOSAR OS, COM und NM

Entwicklungsdauren (*Time to Market*). LIN ist ein klassisches Beispiel für eine Lösung, die technisch nicht mehr kann als der schon lange vorher etablierte CAN, aber geringfügig niedrigere Stückkosten hat. Welcher Aufwand in die Grundsatzentwicklung, Einführung und Pflege von LIN, den Upgrade existierender CAN-Werkzeuge für LIN, die Schulung von Mitarbeitern und die Entwicklung unzähliger CAN/LIN-Gateways geflossen ist, ist schwer zu beziffern und wird von den Protagonisten neuer Lösungen oft sogar bewusst klein gerechnet. Die Kosten von Feldausfällen, die direkt oder indirekt aus Spezifikations- oder Testlücken beim Zusammenspiel von LIN und CAN entstanden, sind wohl selbst den Verantwortlichen nicht wirklich bekannt. Ehrlicherweise muss man diese verborgenen Kosten einer neuen, oft komplexeren Lösung der möglichen Stückkostenreduktion im Vergleich zu einer vorhandenen, einfacheren Lösung gegenüberstellen.

- Viele Lösungen spiegeln die Organisationsstrukturen der Automobilhersteller und ihrer Zulieferer wieder. Historisch wurden Motor, Getriebe, Fahrwerk und Karosserie von verschiedenen Abteilungen, oft sogar Entwicklungs- oder Geschäftsbereichen verantwortet. Auch wenn die Beteiligten an den Schnittstellen ihrer Komponenten zur Zusammenarbeit gezwungen waren, wird doch oft versucht, eine für den eigenen Zuständigkeitsbereich optimale Lösung zu finden. Nicht überzeugend auch, wenn neue Konzepte in Forschung und Vorentwicklung ohne Rücksicht auf Erfahrungen aus der Serien- und Fertigungspraxis entwickelt werden. Auf diese Weise entstehen typische domänenspezifische Lösungen. MOST orientiert sich fast ausschließlich an den Anforderungen des Infotainment. FlexRay merkt man an, dass die Entwickler den Fahrwerksbereich mit seinen zeitkritischen verteilten Regelsystemen und hohen Sicherheitsanforderungen im Blick hatten, während die Flexibilität, mit der Karosserielektroniker noch kurz vor Serienanlauf einige weitere Signale in eine vorhandene oder neue Busbotschaft packen wollen, beim Entwurf sicher nicht im Vordergrund stand. Ärger bereiten solche Insellösungen aber erst dann, wenn Steuergeräte im Fahrzeug dort eingebaut werden müssen,

wo noch ein freier Bauraum zu finden ist. Und wenn ausgerechnet dort das für die Aufgabenstellung eigentlich zuständige Bussystem nicht zur Verfügung steht. Dann wird die Ansteuerung des Motorlüfters schon einmal von der Lichtsteuerung übernommen und das Infotainment-Steuergerät bekommt noch eine CAN- oder LIN-Schnittstelle, weil die Radiobedientasten im Lenkrad nun einmal kein MOST-Interface besitzen. Richtig hinderlich werden diese Domänengrenzen, wenn Fahrerassistenzsysteme integriert werden sollen, die nur durch das Zusammenwirken von Funktionsgruppen mehrerer Domänen realisierbar sind.

- Haupttreiber der Komplexität ist nicht zuletzt das Beharrungsvermögen der Automobilindustrie, das durch die breiteren Modellpaletten und paradoxerweise gerade durch kürzere Entwicklungszyklen erzwungen wird. Beides ist nur möglich, wenn viele Teilsysteme über mehrere Modellbaureihen und Fahrzeuggenerationen wiederverwendet werden. Dies erzwingt Aufwärtskompatibilität und verhindert, dass Neuentwicklungen ältere Lösungen vollständig ablösen. Hatten die ersten Motorsteuergeräte zusätzlich zur ihren proprietären PWM-Schnittstellen ein K-Line-Diagnoseinterface, so kamen später ein, dann mehrere CAN-Busse dazu, ohne dass die K-Line wirklich verschwand. Mittlerweile haben Motorsteuergeräte auch ein FlexRay- und ein LIN-Interface und vielleicht demnächst auch noch eine Ethernet-Schnittstelle. Aber ob die PWM-Schnittstellen wirklich verschwunden sind, ist so sicher nicht. Das alte CAN-Interface, das die Einführung von FlexRay überlebt hat, kann sich freuen, zu CAN FD erweitert zu werden, und muss nicht befürchten, durch Ethernet vollständig ersetzt zu werden. Mit jedem neuen Konzept nimmt die Komplexität zu statt ab. Und so, wie bei CAN, bei LIN und schließlich bei FlexRay wird man sich bei der Einführung von Ethernet vorrangig auf das Übertragungsmedium konzentrieren, statt endlich auch im PKW-Bereich die Anwendungsschnittstelle, d. h. die zu übertragenden Echtzeitdaten, zu standardisieren, wie man das im Nutzfahrzeubereich mit SAE J1939/71 schon vor 20 Jahren gemacht hat.

Natürlich kennt die Automobilindustrie diese Problembereiche und geht dagegen an:

- Elektrik/Elektronik-(E/E)-Architekturen werden gezielter konzipiert und analysiert.
- AUTOSAR ist ein richtiger, wenn auch selbst sehr komplexer Schritt, die Komplexität durch hierarchische Strukturen, Kapselung und eine stärkere Automatisierung der Softwareerstellung beherrschbar zu machen.
- CAN FD ist der Versuch, durch evolutionäre Änderungen den eher revolutionären Umstieg auf FlexRay in vielen Systemen unnötig zu machen.
- Ethernet könnte langfristig vielleicht wirklich zu einer Bereinigung der Bussystem-Landschaft im Automobil führen.

Bis dahin ist allerdings ein langer Weg, so dass der Entwickler und damit dieses Buch von Ausgabe zu Ausgabe immer mehr statt weniger Themengebiete abdecken muss.

Literatur

- [1] Robert Bosch GmbH, K. Reif, K.-H. Dietsche (Hrsg.): Kraftfahrtechnisches Taschenbuch. Springer Vieweg Verlag, 27. Auflage, 2011
- [2] R. K. Jurgen (Hrsg.): Automotive Electronics Handbook. McGraw Hill Verlag, 2. Auflage, 1999
- [3] G. Conzelmann, U. Kiencke: Mikroelektronik im Kraftfahrzeug. Springer Verlag, 1. Auflage, 1995
- [4] J. Schäuffele, T. Zurawka: Automotive Software Engineering. Springer-Vieweg Verlag, 5. Auflage, 2013
- [5] K. Reif: Automobilelektronik. Springer-Vieweg Verlag, 4. Auflage, 2012
- [6] H. Wallentowitz, K. Reif: Handbuch Kraftfahrzeugelektronik. Springer-Vieweg Verlag, 2. Auflage, 2011

In diesem Kapitel werden die elektrotechnischen Grundlagen dargestellt und einfache Kfz-Kommunikationsschnittstellen wie K-Line, SAE J1850 sowie einige Sensor-Aktor-Busse betrachtet.

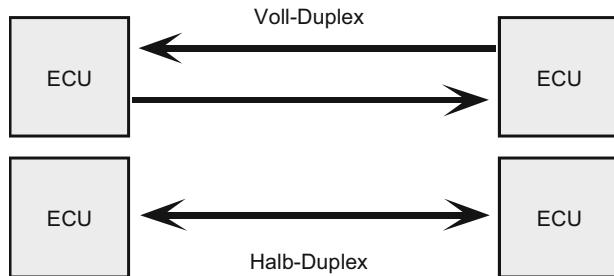
2.1 Grundlagen

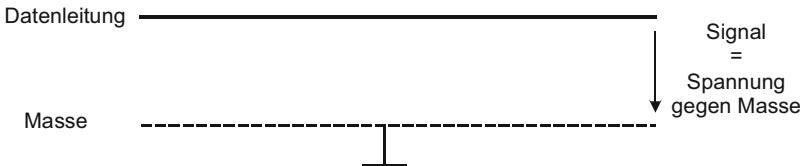
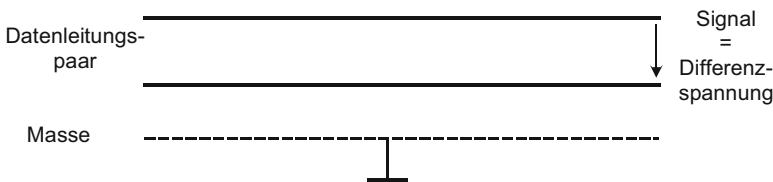
Zunächst werden die für die Anwendung im Kfz wichtigen Grundeigenschaften von Bussystemen vorgestellt. Dabei wird davon ausgegangen, dass der Leser mit den allgemeinen Grundlagen und Grundbegriffen von Datennetzen vertraut ist [1–3].

2.1.1 Elektrotechnische Grundlagen

Die Datenübertragung im Kfz erfolgt in der Regel bitweise seriell. Im einfachsten Fall sind zwei Geräte direkt miteinander verbunden (Abb. 2.1).

Abb. 2.1 Punkt-zu-Punkt-Verbindung



Ein-Draht-Leitung:**Zwei-Draht-Leitung:****Abb. 2.2** Ein-Draht- und Zwei-Draht-Leitung

Abhängig davon, ob eine gemeinsame bidirektionale Leitungsverbindung oder ein Paar von jeweils unidirektionalen Leitungsverbindungen vorgesehen wird, ist eine Halb- oder Voll-Duplex-Verbindung möglich:

Voll-Duplex: Gleichzeitiges Senden und Empfangen ist möglich.

Halb-Duplex: Jedes Steuergerät kann nur abwechselnd senden oder empfangen. Standardverfahren bei Kfz-Bussystemen.

Die elektrischen Verbindungen können als Ein-Draht- oder als Zwei-Draht-Leitung ausgeführt werden (Abb. 2.2). Bei der kostengünstigen Ein-Draht-Leitung erfolgt die Signalrückführung über die Fahrzeugkarosserie (Masse). Derartige Verbindungen sind anfällig für die Ein- und Abstrahlung elektromagnetischer Störungen. Daher muss mit relativ großen Signalpegeln, im Kfz häufig mit Batteriespannungspegel, und niedrigen Bitraten gearbeitet werden. Üblich ist dies z. B. bei LIN oder ISO 9141 mit einer bidirektionalen Ein-Draht-Leitung (K-Line bei ISO 9141) als Halb-Duplex-Verbindung, seltener mit zwei unidirektionalen Ein-Draht-Leitungen als Voll-Duplex-Verbindung.

Zwei-Draht-Leitungen, deren Adern oft noch zusätzlich verdrillt werden (*Twisted Pair*) sind weniger anfällig für EMV-Probleme (Elektromagnetische Verträglichkeit). Dabei wird die Spannung zwischen den beiden Leitungen zur Signalübertragung verwendet (Differenzsignal). Daher kann mit kleineren Signalpegeln und höheren Bitraten gearbeitet werden. Im Kfz werden für Class B und C Netze, z. B. CAN oder FlexRay, üblicherweise Halb-Duplex-Zwei-Draht-Verbindungen eingesetzt. Abgeschirmte Zwei-Draht-Leitungen oder Paare von Zwei-Draht-Leitungen, also insgesamt vieradrige Verbindungen, wie sie heute bei Voll-Duplex-Ethernet-LANs üblich sind, werden im Kfz in der Regel aus Kostengründen nicht verwendet. Dasselbe gilt auch für optische Netze mit Lichtwellenleitern (LWL), die mit Ausnahme des Infotainment-Bussystems MOST im Kfz bisher nicht eingesetzt wer-

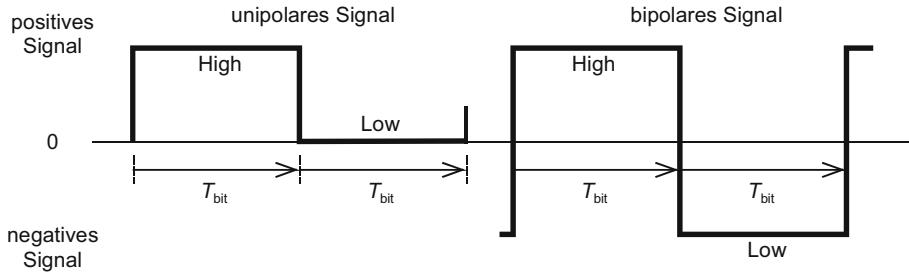


Abb. 2.3 Unipolare und bipolare Signale

den. Neben den teuren, in jedem Steuergerät notwendigen elektrooptischen Wendlern sind auch die Steckverbindungen für LWL fehleranfällig und die LWL selbst knickempfindlich, so dass die Verlegung zusammen mit den üblichen Kabelbäumen problematisch sein kann.

Bei den Ein-Draht-Verbindungen sind die Signale in der Regel *unipolar*, das Differenzsignal bei den Zwei-Draht-Verbindungen, z. B. CAN, meist *bipolar* (Abb. 2.3). Die digitale Übertragung erfolgt üblicherweise mit positiver Logik, d. h. die höhere Spannung (*High*) entspricht der *logischen 1*, die kleinere Spannung (*Low*) der *logischen 0* (*binäre Signale*). Verwendet man wie bei FlexRay einen dritten, zwischen *High* und *Low* liegenden Spannungspegel um z. B. den Ruhezustand des Bussystems abzubilden, spricht man von *ternären Signale*n.

Da die EMV-Störstrahlung umso größer ist, je häufiger Signalflanken auftreten, arbeitet man in der Regel mit Non-Return-to-Zero-(NRZ)-Codierung, d. h. das Signal bleibt für die gesamte Dauer eines Bits T_{bit} konstant auf Low oder High. Lediglich das veraltete SAE J1850 arbeitet mit pulsweitenmodulierten Signalen.

Der Wert

$$f_{\text{bit}} = \frac{1}{T_{\text{bit}}}$$

wird als *Bitrate* bezeichnet. Heute übliche Bitraten liegen im Bereich von 10 kbit/s (Class A-Netz, z. B. K-Line), d. h. *Bitdauer* $T_{\text{bit}} = 100 \mu\text{s}$, bis zu 500 kbit/s (Class C-Netz, z. B. CAN), d. h. $T_{\text{bit}} = 2 \mu\text{s}$. Bei Class D-Netzen (z. B. FlexRay) werden heute bis zu 10 Mbit/s, d. h. $T_{\text{bit}} = 100 \text{ ns}$, verwendet und im Infotainmentbereich (z. B. MOST) sind schon heute 25 Mbit/s, d. h. $T_{\text{bit}} = 40 \text{ ns}$, im Einsatz, zukünftig 150 Mbit/s.

Ebenfalls aus EMV-Gründen versucht man, die Steilheit (*Slew Rate*) der Signalflanken durch entsprechende Treiberschaltungen so langsam zu machen wie für eine gegebene Bitrate gerade noch zulässig.

Die Ausbreitung der Signale auf den Leitungen erfolgt mit Lichtgeschwindigkeit. Aufgrund der dielektrischen Eigenschaften des Kabelmaterials reduziert sich diese gegenüber der Lichtgeschwindigkeit im Vakuum etwa um den Faktor 2 ... 3 auf ca.

$$c = 10 \dots 15 \frac{\text{cm}}{\text{ns}} .$$

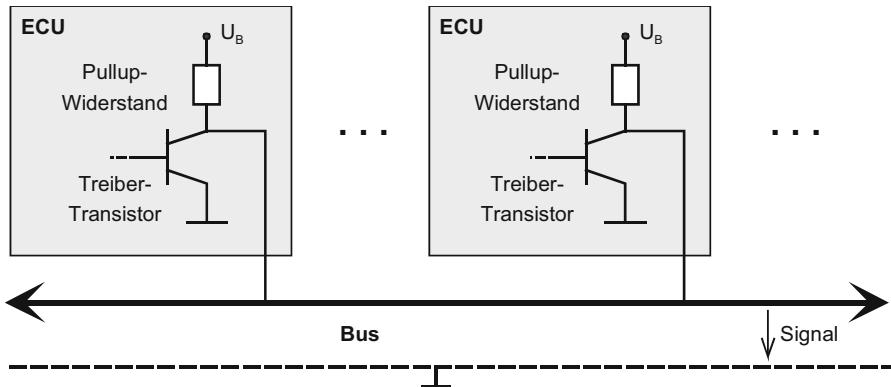


Abb. 2.4 Linien-Bus

Bei einer $l = 20 \text{ m}$ langen Verbindungsleitung, wie sie in einer LKW-Zugmaschine mit Auflieger oder einem Omnibus durchaus denkbar ist, beträgt die Signallaufzeit auf der Leitung in einfacher Richtung

$$T_d = \frac{I}{c} = 125 \dots 200 \text{ ns} .$$

Zusätzliche Verzögerungen entstehen in den Leitungstreibern und -empfängern in den Steuergeräten. Für $T_d > 0,1 \cdot T_{\text{bit}}$ treten in der Praxis bereits deutliche Welleneffekte auf (Reflexionen an den Leitungsenden und an Stoßstellen wie Abzweigungen und Steckverbinder), welche die Übertragungssicherheit beeinträchtigen. Für Bussysteme mit höheren Bitraten müssen die Kabelführung, die Steckverbinder und die Kabelausführung sorgfältig gewählt werden. Zusätzlich müssen die Kabel elektrisch *abgeschlossen* werden, d. h. an den beiden Kabelenden sollten an das Kabel angepasste Abschlusswiderstände vorgesehen werden, um die genannten Welleneffekte zu vermeiden. Meist werden diese Abschlusswiderstände in diejenigen Steuergeräte eingebaut, die an den beiden Kabelenden angeschlossen werden, oder sie werden in das Kabel bzw. die Steckverbinder integriert. Steuergeräte, die bei Bussystemen nicht an den Leitungsenden sitzen sondern über Stichleitungen dazwischen angeschlossen sind (Abb. 2.4), dürfen keine Abschlusswiderstände enthalten.

Im Gegensatz zu einer Punkt-zu-Punkt-Verbindung sind bei einem echten Bus mehrere Steuergeräteausgänge an dieselbe Leitung angeschlossen. Die Signalverknüpfung erfolgt dabei nach dem Wired-OR-Grundprinzip, wie es im Abb. 2.5 für eine Ein-Draht-Busleitung dargestellt ist.

Wenn beide Steuergeräte ein *High*-Signal oder gar kein Signal senden wollen, sperren sie den jeweiligen Treibertransistor. Durch die so genannten *Pull-Up*-Widerstände wird die Busleitung dann auf den Pegel der Versorgungsspannung U_B gezogen. Sobald ein Steuergerät ein *Low*-Signal senden will, schaltet es den jeweiligen Treibertransistor durch. Da der leitende Transistor viel niederohmiger ist als die *Pull-Up*-Widerstände, wird die Busleitung damit praktisch kurzgeschlossen und das Signal *Low* gesendet. Wenn dagegen zwei Steuer-

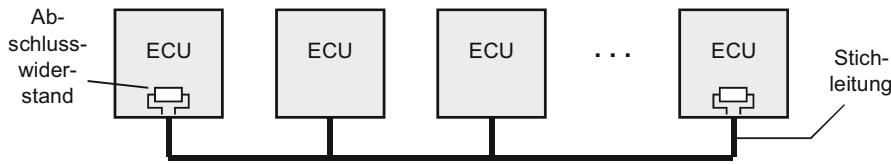


Abb. 2.5 Wired-OR-Prinzip

geräte gleichzeitig zu senden versuchen und ein Gerät *Low*, das andere Gerät *High* sendet, „gewinnt“ dasjenige Steuergerät, das das *Low*-Signal auf den Bus legt. Man bezeichnet das *Low*-Signal in diesem Fall als dominant, das bei einer solchen Kollision „unterlegene“ *High*-Signal als rezessiv. Um solche Kollisionen zu erkennen, muss jeder Sender die tatsächlich auf dem Bus liegenden Signalpegel überwachen und mit seinem Sendewunsch vergleichen.

2.1.2 Topologie und Kopplung von Bussystemen

Im Gegensatz zu einer Punkt-zu-Punkt-Verbindung können bei einem Datennetz mehrere Steuergeräte (*ECU Electronic Control Unit*) untereinander Daten austauschen. Für solche Datennetze existiert eine Reihe von unterschiedlichen Aufbauformen (Topologien), von denen die in der Praxis im Kfz am häufigsten zu findende Topologie der so genannte *Bus* ist. Aus diesem Grund wird der Begriff *Bus* häufig, aber nicht ganz präzise, als Synonym für alle Datennetze im Kfz verwendet.

Ein Bus, noch präziser ein *Linien-Bus*, entsteht, wenn mehrere Steuergeräte über kurze Stichleitungen an dieselben Verbindungsleitungen angeschlossen werden (Abb. 2.4). Durch ein Buszugriffsverfahren muss geregelt werden, welches Gerät zu welchem Zeitpunkt senden darf, um Kollisionen zu verhindern bzw. wie Kollisionen erkannt und aufgelöst werden.

Die gesendeten Daten können von allen anderen Steuergeräten empfangen werden. Wenn die gesendeten Daten nicht für alle angeschlossenen Geräte (*Broadcast*) bestimmt sind, muss bei Beginn der Datenübertragung oder mit jedem Datenpaket eine Adressierungsinformation übertragen werden, die einen einzelnen Empfänger (*Unicast*) oder mehrere Empfänger (*Multicast*) auswählt.

FlexRay unterstützt zwar auch Linien-Busse, bevorzugt aber Stern-Strukturen (Abb. 2.6). Ringe sind beim Infotainment-Bus MOST üblich.

Baum-ähnliche Netze findet man heute als Zusammenschaltung mehrerer Linien-Busysteme. So kann man aus Sicht des Diagnosetesters das Fahrzeugnetz in Abb. 1.1 als Baum sehen, von dessen zentralem Knoten, dem *Gateway*, die Bussysteme für den Triebstrang, die Karosserie und die Infotainment-Kommunikation als Zweige abgehen, wobei der Karosserie-Bus weiter in die Türbusse verzweigt. An denjenigen Stellen, an denen Steuergerät und Bus bzw. mehrere Bussysteme zusammengeschaltet werden, muss die Koppelstelle bestimmte Umsetzungsaufgaben erledigen (Tab. 2.1).

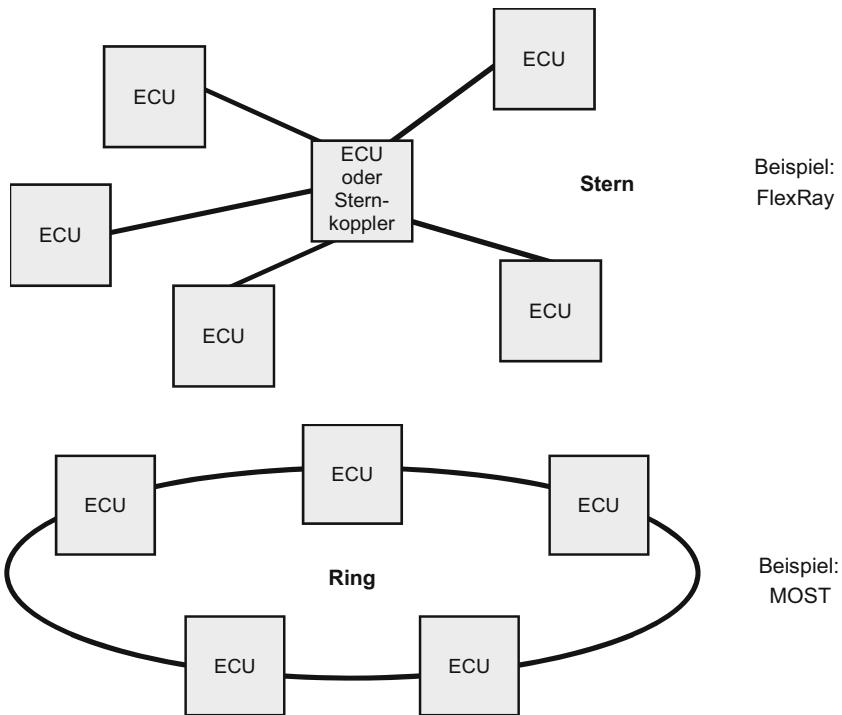


Abb. 2.6 Weitere Netz-Topologien

2.1.3 Botschaften, Protokollstapel, Dienste (Services)

Bei der Übertragung werden die Nutzdaten mit zusätzlichen Informationen versehen (Abb. 2.7). Vor den eigentlichen Nutzdaten (*Payload*) wird üblicherweise ein Vorspann (*Header*) übertragen, der Adressinformationen enthält (Sender und Empfänger der Daten), Angaben über die Anzahl der zu übertragenden Daten und gegebenenfalls zu deren Art. An die zu übertragenden Nutzdaten wird häufig ein Nachspann (*Trailer*) angehängt, der Informationen zur Fehlerprüfung und -korrektur enthält.

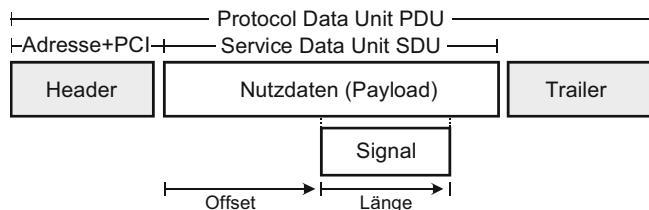


Abb. 2.7 Aufbau einer Botschaft

Tab. 2.1 Kopplung von Bussystemen

Bus-Koppler	Aufgabe
Transceiver (Abb. 2.11)	Umsetzung der Bussignale auf die Steuergeräte-internen Spannungsspegel für die Sendesignale (Transmitter) und die Empfangssignale (Receiver), andere Bezeichnung <i>Bustreiber</i> oder <i>Leitungstreiber</i> , oft auch <i>physisches Businterface</i> (PHY). Findet sich in jedem Steuergerät.
Repeater	Signalverstärker zwischen Leitungsabschnitten ohne <i>Eigenintelligenz</i> . Erlaubt aus elektrotechnischer Sicht längere Leitungen, führt aber zu zusätzlichen Signallaufzeiten. Logisch bilden die beiden über einen Repeater gekoppelten Leitungsstücke einen einzigen Bus. Üblich im aktiven Sternpunkt von FlexRay; möglich bei CAN zur Kopplung von Zugmaschine und Anhänger, wird aber in der Regel als <i>Gateway</i> realisiert.
Gateway (Abb. 1.1)	Kopplung mehrerer Netze mit unterschiedlichen Protokollen, z. B. Umsetzung von CAN-Botschaften auf einen LIN-Bus, oder Bussen mit unterschiedlichen Adressbereichen und Bitraten, z. B. zwischen einem Triebstrang-CAN-Bus mit 500 kbit/s und 11 bit-Adressen (Identifier) und einem Karosserie-CAN-Bus mit 125 kbit/s und 29 bit-Adressen.
Switch, Hub, Router	Bei Kfz-Netzen nicht üblich. Router-Funktion oft im Gateway.

Bei einigen Protokollen werden auch zwischen die Nutzdaten nach bestimmten Vorschriften noch weitere Daten zur Übertragungssteuerung oder Trennung von Datenblöcken eingefügt, z. B. *Stuffing-Bits* oder ähnliches.

Im Bereich der Nutzdaten werden häufig mehrere, inhaltlich nicht notwendigerweise direkt zusammengehörige Datenwerte übertragen, z. B. die Motordrehzahl und die Kühlwassertemperatur. Wenn es sich bei diesen Daten um Messwerte handelt, spricht man von *Signalen*. Ein derartiges *Signal* wird durch seine Länge sowie seine Lage bezogen auf den Anfang des Nutzdatenbereichs (Offset in Bit oder Byte) und gegebenenfalls eine Umrechnungsvorschrift definiert, die den Zusammenhang zwischen dem eigentlichen physikalischen Wert, z. B. Drehzahl in 1/min, und dessen hexadezimaler Codierung beschreibt.

Gemäß dem OSI-Schichtenmodell durchlaufen die Daten von der Anwendung bis zur eigentlichen Übertragung über die Busleitungen mehrere Schichten (vgl. Tab. 1.3), die jeweils eigene Header und Trailer verwenden. Die Botschaft der nächsthöheren Ebene wird beim Senden auf der nächstniedrigeren Ebene als Payload verstanden und um die Header und Trailer der aktuellen Schicht ergänzt (Abb. 2.8). Beim Empfangen dagegen werden Header und Trailer der aktuellen Schicht entfernt, bevor die Payload dieser Schicht an die nächsthöhere Ebene weitergegeben wird. Innerhalb einer Schicht werden im Idealfall nur die zugehörigen Header und Trailer verarbeitet, der Inhalt der jeweiligen Payload dagegen ist für die Schicht selbst bedeutungslos. Auf diese Weise entsteht ein *Protokollstapel*. Der Header einer Schicht wird häufig als *Protocol Control Information PCI* bezeichnet, weil er das für diese Schicht spezifische Protokoll beschreibt, die Nutzdaten als *Service Data Unit SDU* und der gesamte Datenblock als *Protocol Data Unit PDU*.

In der Regel ist die Länge einer Botschaft auf dem Bussystem begrenzt. So erlaubt z. B. CAN nur maximal 8 Byte, d. h. 64 bit Nutzdaten in einer Botschaft. In diesem Fall wird eine

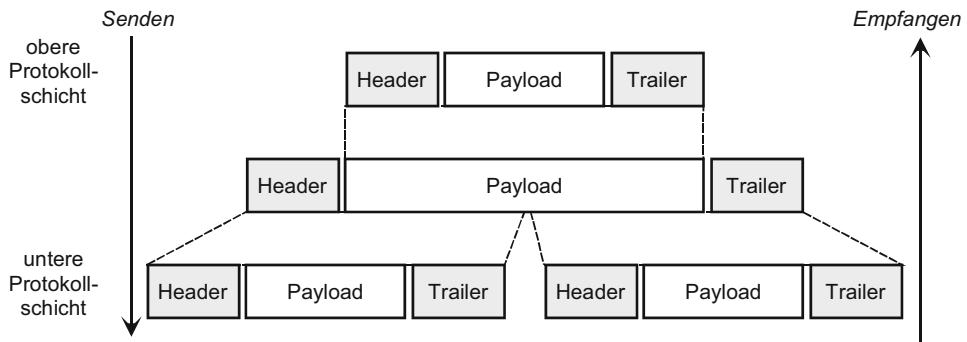


Abb. 2.8 Protokollstapel (Protocol Stack)

längere Botschaft der oberen Schicht beim Senden in mehrere Botschaften aufgeteilt (*Segmentierung*) und beim Empfangen wieder aus mehreren Botschaften der unteren Schicht zusammengesetzt (*Desegmentierung*). Der umgekehrte Fall, dass eine untere Ebene längere Botschaften erlaubt als eine höhere Ebene und daher mehrere Botschaften beim Senden zu einer Botschaft zusammengefasst werden, ist bei Kfz-Bussen derzeit noch selten, kommt aber z. B. bei FlexRay in Betracht.

Bei sauber spezifizierten Protokollen besitzt jede Schicht des Protokollstapels eine exakt definierte Reihe von Funktionen, so genannte *Dienste* (*Services*). Die übergeordnete Protokollsicht verwendet diese Dienste, um Botschaften an die darunter liegende Schicht zu senden, von dieser zu empfangen oder Konfigurations- und Statusinformationen mit dieser Schicht auszutauschen.

Der Begriff *Botschaft* (engl.: *Message*) ist dabei nicht eindeutig. Je nach Ebene in einem Protokoll und in unterschiedlichen Standards mit unterschiedlicher Bedeutung gebraucht, finden sich auch die Begriffe *Datenblock*, *Paket* und *Rahmen* (*Frame*).

2.1.4 Kommunikationsmodelle, Adressierung

Der Datenaustausch zwischen zwei Steuergeräten über einen Protokollstapel vermittelt den Anwendungen im Idealfall die Illusion, ohne zwischengeschaltetes Bussystem direkt miteinander zu kommunizieren (Abb. 2.9). Der Protokollstapel kapselt die Details der Kommunikation, wobei auch innerhalb des Protokollstapels jede Ebene den Eindruck haben sollte, direkt mit der entsprechenden Ebene des anderen Kommunikationspartners zu interagieren.

In der Regel wird die Kommunikation zwischen den Steuergeräten im Fahrzeug während der Entwicklungsphase festgelegt, da die verbauten Geräte und die Struktur der austauschenden Informationen vorab bekannt sind und damit keine Laufzeitkonfiguration mehr nötig ist. Bei manchen Protokollen ist trotzdem am Beginn der Kommunikation eine gegenseitige Verständigung vorgesehen, ähnlich der Wahl der Telefonnummer und

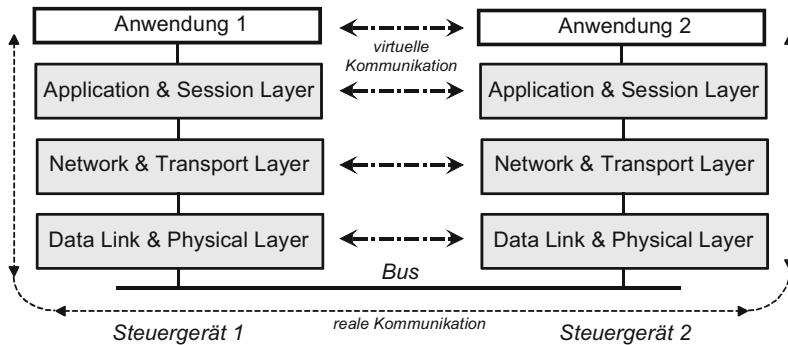


Abb. 2.9 Kommunikation zwischen Anwendungen über einen Protokollstapel

dem folgenden Verbindungsauflauf bei Telefongesprächen. Derartige Protokolle werden als *verbindungsorientiert* bezeichnet und bestehen üblicherweise aus den drei Phasen *Verbindungsauflauf*, *Datenaustausch* (Kommunikationsphase) und *Verbindungsabbau*. Können Geräte dagegen spontan ohne vorherige Absprache kommunizieren, bezeichnet man dies als *verbindungslose* Kommunikation.

Bei Kfz-Bussystemen werden Steuer- und Regeldaten praktisch immer im *Broadcast*-Verfahren versendet (Abb. 2.10). Das Steuergerät sendet seine Botschaften unaufgefordert. Jeder Empfänger entscheidet, ob er die Botschaften ignoriert oder den Dateninhalt auswertet (*Empfangs- oder Akzeptanzfilterung*). In der Regel besitzen die Protokolle zusätzlich *Adressierungsmechanismen*, mit denen ein einzelner (*Unicast*) oder mehrere Empfänger (*Multicast*) gezielt angesprochen werden können.

Für die Adressierung von Botschaften gibt es zwei gängige Verfahren:

- Gerätebasierte Adressierung (Stations- oder Teilnehmeradressierung)
- Inhaltsbasierte Adressierung (Botschafts-Kennung, Identifier).

Bei der *gerätebasierten Adressierung* (Stations- oder Teilnehmeradressierung) wird jedes Steuergerät durch eine eindeutige Adresse (Nummer) identifiziert, wobei jede Botschaft neben der *Zieladresse*, d. h. der Adresse des Empfängers, in der Regel auch die *Quelladresse*, d. h. die Adresse des Senders enthält. Für die Aussendung von Botschaften an mehrere

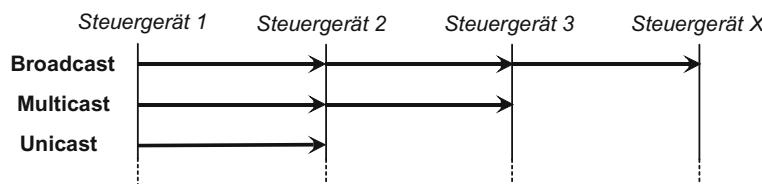


Abb. 2.10 Botschaftsversendung an alle, eine Gruppe oder einen einzelnen Empfänger

(*Multicast*) oder alle Empfänger (*Broadcast*) existieren spezielle Adressen, die gelegentlich als funktionale Adressen bezeichnet werden. Da die Anzahl der Steuergeräte in einem Fahrzeug relativ klein ist, genügen für Adressen einige Bit, typ. 5...8. Werden mehrere Busse über ein Gateway verknüpft, wird in den höheren Protokollsichten meist eine für das Gesamtsystem eindeutige Geräteadresse verwendet, die vom Gateway in einer unteren Protokollsicht in eine nur für das adressierte Bussystem eindeutige Geräteadresse umgesetzt wird.

Anstelle der gerätebasierten Adressierung kann auch eine *inhaltsbasierte Adressierung* verwendet werden, wie z. B. bei CAN. Dabei wird eine Botschaft mit einer bestimmten, adressähnlichen Kennung (Identifier) versehen, die den Inhalt der Botschaft kennzeichnet, also anzeigt, dass eine Botschaft beispielsweise Informationen über die Motordrehzahl enthält. Da in heutigen Fahrzeugen sehr viele solcher Informationen über die Bussysteme gesendet werden, benötigt man hierfür längere *Adressen*, bei CAN z. B. sind alternativ 11 bit oder 29 bit Kennungen üblich. Die inhaltsbasierte Adressierung führt zu einer geringeren Busbelastung und zu einer besseren Datenkonsistenz im gesamten Netzwerk, wenn Daten häufig an mehrere Teilnehmer versendet werden müssen.

Sowohl bei gerätebasiert als auch bei inhaltsbasiert Adressierung sehen bei einem Bussystem alle angeschlossenen Steuergeräte jede Botschaft und müssen auf Basis der Adresse entscheiden, ob sie die Botschaft tatsächlich empfangen und weiterverarbeiten oder einfach ignorieren wollen (*Akzeptanzfilterung*). Bei Bussystemen mit niedriger Bitrate findet die Akzeptanzprüfung in der Regel in der Protokollsoftware statt. Bei Protokollen mit höheren Bitraten findet die Protokollverarbeitung für die Schichten 1 und 2 und damit nach Möglichkeit auch die Akzeptanzfilterung dagegen im Kommunikationscontroller statt, einem speziellen IC oder einem auf dem Mikrocontroller des Steuergerätes integrierten Schaltungsteil, häufig auch als *Media Access Controller* (MAC) bezeichnet (Abb. 2.11). Bei der gerätebasierten Adressierung ist die Akzeptanzfilterung verhältnismäßig einfach, da nur die eigene Zieladresse sowie ggf. die Multicast- und Broadcast-Adressen erkannt werden müssen, während bei der inhaltsbasierten Adressierung oft mehrere Dutzend oder gar Hunderte *Adressen* (Botschaftskennungen) erkannt werden müssen. Gängige Kommunikationscontroller können typischerweise so programmiert werden, dass sie die Kennung von 8 bis 16 Botschaften hardwaremäßig erkennen, wobei mit Hilfe von Bitmasken auch ganze Bereiche von Adressen festgelegt werden können. Für die übrigen Botschaftskennungen, die ebenfalls empfangen werden sollen, muss eine Multicast-Kennung eingerichtet werden, für die die Akzeptanzfilterung dann softwaremäßig erfolgt.

Während der Sender bei *Broadcast*-Kommunikation in der Regel keine Bestätigung der Empfänger erhält (*Unacknowledged Communication*), erwartet der Sender bei *Unicast*-Sendungen oft eine positive Bestätigung (*Acknowledged Communication*) des Empfängers, dass die Botschaft korrekt empfangen wurde (Abb. 2.12). Verzichtet man, um das Bussystem zu entlasten, auf die positive Bestätigung, so wird zumindest eine Fehlermeldung (*Not Acknowledged*) erwartet, wenn einer der Empfänger einen Übertragungsfehler erkannt hat. In der Regel versendet der Sender die Botschaft darauf erneut. Ein Kommunikationsprotokoll, bei dem sich die Anwendung darauf verlassen kann, dass der *Network and*

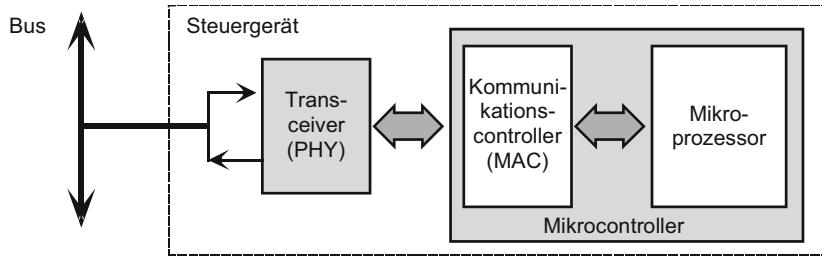


Abb. 2.11 Typisches Bus-Interface eines Steuergerätes (PHY Physical Interface, MAC Media Access Control)

Transport bzw. Data Link Layer selbstständig gewährleisten, dass die Information beim Empfänger korrekt ankommt, bezeichnet man als *gesicherte* Übertragung, das Gegenteil als *ungesichert*.

Das *Broadcast*-Verfahren, bei dem der Sender „ungefragt“ Informationen versendet, ähnelt der Ausstrahlung von Rundfunk- und Fernsehprogrammen. Im Gegensatz dazu fordert ein Gerät beim *Request-Response*-Verfahren (Abb. 2.13) ein anderes Gerät durch eine Anfrage-Botschaft (*Request*) gezielt auf, Informationen (*Response*) zu liefern. Dieses Kommunikationsmodell wird unter dem Namen *Client-Server*-Verfahren auch im Internet verwendet, bei dem ein Web-Browser (*Client*) eine Anfrage an einen Web-Server stellt und von dort eine Antwort, z. B. eine HTML-Seite, erhält. Wird die Anfrage mit *Multi-cast*-Addressierung gesendet, muss der Sender mit Antworten von mehreren Empfängern rechnen. Benötigt man dieselbe Information wiederholt, entlastet das *Publisher-Subscriber*-Verfahren das Bussystem. Dabei meldet der an der Information interessierte Partner sein Interesse an der Information einmalig an (*Subscribe*), worauf der andere Partner die Information dann ohne weitere Anfragen regelmäßig (*Periodic*) oder bei Änderungen der Daten (*On Event*) liefert.

Ist die Informationsmenge größer als die Nutzdatenlänge einer einzelnen Botschaft, muss die Information vor der Übertragung auf der Senderseite zerstückelt (*Segmentierung*) und auf der Empfängerseite wieder zusammengesetzt (*Desegmentierung*) werden. Dies erfolgt in der Regel im *Network and Transport Layer* des Protokollstapels, dem sogenannten *Transportprotokoll* (siehe Kap. 4). Um den Empfänger vor zu großen Datenmengen oder zu schnell aufeinanderfolgenden Botschaften zu schützen, verfügen diese Protokolle in der

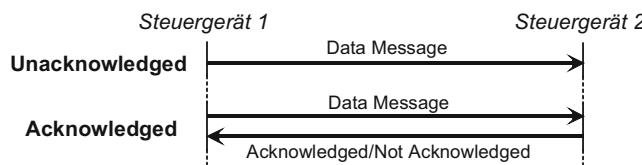


Abb. 2.12 Unbestätigte und bestätigte Kommunikation

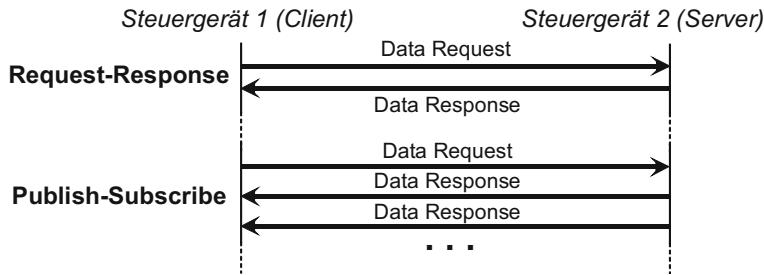


Abb. 2.13 Abfrage von Informationen

Regel auch über eine *Flusssteuerung*, bei der der Empfänger dem Sender mitteilt, wie viele Botschaften in welchem Abstand er empfangen bzw. den Sender zu einer Sendepause veranlassen kann.

Die Schnittstelle zwischen Anwendung und Protokollstapel bzw. den verschiedenen Schichten eines Protokollstapels verwendet üblicherweise die in Abb. 2.14 dargestellte Ablaufreihenfolge. Die Anwendung beauftragt den Protokollstapel mit dem Versenden einer Botschaft (*Transmit Request*) und lässt sich optional die durchgeführte Übertragung bestätigen (*Transmit Confirmation*). Danach wartet die Anwendung im einfachsten Fall, bis die Antwort eintrifft (*synchrone* oder *blockierender Betrieb*). Falls die Anwendung dagegen weiterarbeitet (*asynchrone* oder *nicht-blockierender Betrieb*), muss der Protokollstapel die Anwendung informieren, wenn eine Antwort eingetroffen ist. Diese Rückmeldung kann durch Setzen eines Flags erfolgen, das die Anwendung von Zeit zu Zeit abfragt (*Polling*), oder indem der Protokollstack eine Interrupt- oder Rückruffunktion der Anwendung aufruft (*Receive Indication*, *Notification* oder *Callback*).

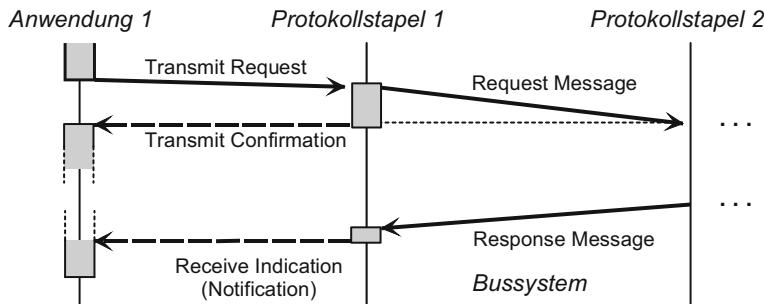


Abb. 2.14 Ablaufreihenfolge beim Senden und Empfangen von Botschaften

2.1.5 Zeichen- und Bitstrom-basierte Übertragung, Nutzdatenrate

Da bei den Bussystemen die Bitrate f_{bit} , d. h. die Geschwindigkeit, mit der die einzelnen Bits über die Busleitungen übertragen werden, durch die elektrotechnischen Randbedingungen vorgegeben ist, und der Header und der Trailer ebenfalls Übertragungszeit benötigen (Protokoll-Overhead), reduziert sich die Nutzdatenrate f_{Daten} mindestens im Verhältnis

$$f_{\text{Daten}} = f_{\text{bit}} \cdot \frac{\text{Anzahl der Nutzdatenbits}}{\text{Anzahl der Nutzdatenbits} + \text{Anzahl der Steuerbits}}$$

mit *Anzahl der Steuerbits* = *Anzahl der Bits im Header + Trailer + Sonstige*.

Für die eigentliche Datenübertragung über die Busleitungen werden zwei verschiedene Verfahren eingesetzt:

- Zeichen-basierte Übertragung, üblich bei niedrigen Bitraten, z. B. K-Line, LIN,
- Bitstrom-basierte Übertragung, üblich bei hohen Bitraten, z. B. CAN, FlexRay.

Bei der *Zeichen-basierten Übertragung* (andere Bezeichnungen: *asynchrone Übertragung*, Start-Stopp-Verfahren, *UART*-basierte Übertragung) werden die zu übertragenden Daten einer Botschaft einschließlich Header und Trailer in Gruppen von je 8 bit (ein Zeichen) unterteilt und jedes Zeichen einzeln übertragen (Abb. 2.15). Zwischen den Zeichen befindet sich die Übertragungsleitung im Ruhezustand (*logisch 1*). Als Kennzeichnung für den Beginn eines Zeichens wird ein Startbit (*logisch 0*) übertragen, anschließend die 8 bit des Zeichens beginnend mit dem niederwertigsten Bit (LSB). Danach wird eventuell noch ein Paritätsbit gesendet, mit dem einfache Übertragungsfehler erkannt werden können. Anschließend wird ein Stoppbit (*logisch 1*) gesendet. Bis zum Beginn des nächsten Zeichens bleibt die Leitung wieder im Ruhezustand. Vorteil dieses Verfahrens ist, dass der Kommunikationscontroller sehr einfach ist und als so genannter *UART* (Universal Asynchronous Receiver and Transmitter) auf praktisch jedem Mikrocontroller bereits integriert ist. Der *UART* wickelt das Senden und Empfangen eines Zeichens inklusive der Erzeugung von Start- und Stoppbit sowie Paritätsbit und dessen Überprüfung selbstständig ab. Die Bereitstellung und Weiterverarbeitung jedes Zeichens dagegen muss durch die Protokollsoftware erfolgen, so dass das Verfahren zu einer relativ hohen Rechnerbelastung führt bzw. in der Praxis auf niedrige Datenraten beschränkt ist. Die Bitrate ist konstant und wird durch den *UART* erzeugt. Der zeitliche Abstand zwischen den einzelnen Zeichen T_{ICB} (*Inter Character Break*) sowie der Abstand zwischen dem letzten Zeichen einer Botschaft und dem ersten Zeichen der nächsten Botschaft T_{IFB} (*Inter Frame Break*) wird in den Protokollen typischerweise durch Minimal- und Maximalwerte vorgegeben, um der Protokollsoftware ausreichend Zeit für das Bereitstellen und Verarbeiten der Daten zu geben. Der Overhead der Übertragung ist hoch. Selbst wenn alle Zeichen ohne Pause übermittelt werden könnten, lässt sich wegen der für jeweils 8 Datenbits zusätzlich nötigen Start- und Stoppbits im günstigsten Fall (ohne Paritätsbit und Pausen) eine Nutzdatenrate

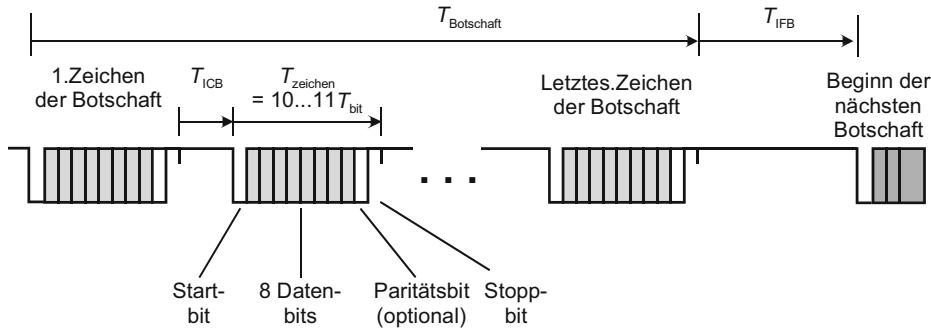


Abb. 2.15 Zeichen-basierte Übertragung (ICB ... Inter Character Break, IFB ... Inter Frame Break)

von maximal

$$f_{\text{Daten}} = f_{\text{bit}} \cdot \frac{8}{10}$$

erreichen. Durch die Pausenzeiten zwischen den Zeichen und Botschaften wird die Nutzdatenrate nochmals verringert.

Bei der *Bitstrom-basierten Übertragung* (Abb. 2.16, andere Bezeichnung: *synchrone Übertragung*) werden alle zu einer Botschaft gehörenden Daten inklusive Header und Trailer ohne Pause als ein zusammengehörender Block (Frame) im Bittakt übertragen (so genannter Bitstrom). Die Pause zwischen den Frames T_{IFB} (Inter Frame Break) wird im Protokoll wieder mit Minimal- und Maximalwerten vorgegeben. Gegenüber der zeichenbasierten Übertragung ergibt sich durch die innerhalb der Botschaft pausenfreie Übertragung eine deutlich höhere Nutzdatenrate. Allerdings ist auch ein wesentlich aufwendigerer Kommunikationscontroller notwendig, der die gesamte Übertragung in Hardware abwickelt, so dass die Protokollsoftware lediglich die Botschaft zur Verfügung stellen bzw. weiterverarbeiten muss.

Sender und Empfänger arbeiten in der Regel mit jeweils einem eigenen lokalen Taktgenerator für den Bittakt. Obwohl Sender und Empfänger stets mit derselben nominalen Bitrate arbeiten, sind kleinere Toleranzen in der Praxis nicht vermeidbar.

Bei der Zeichen-basierten Datenübertragung wird der Bittaktgenerator des Empfängers durch die Signalflanke zu Beginn des empfangenen Startbits synchronisiert. Die bis zum

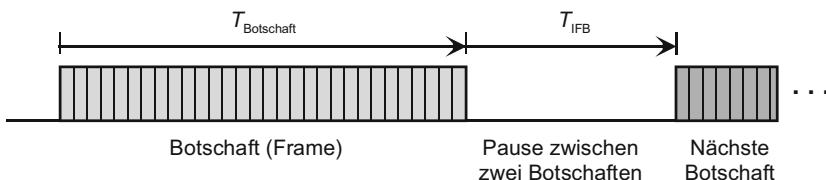


Abb. 2.16 Bitstrom-basierte Übertragung (IFB ... Inter Frame Break)

Ende eines Zeichens aufsummierten Toleranzen der Bitzeiten müssen deutlich kleiner sein als eine Bitdauer, sonst werden die Bits am Zeichenende nicht mehr korrekt empfangen. Da ein Zeichen aus relativ wenig Bits besteht, typ. 10 bit (1 Start-, 8 Daten-, 1 Stopp-Bit), sind verhältnismäßig große Toleranzen zulässig, z. B. bei einem aufsummierten, zulässigen Fehler von $0.5 T_{\text{bit}}$ bis zu $\pm 5\%$.

Bei der Bitstrom-basierten Zeichenübertragung mit Botschaften von zum Teil weit über 100 bit würden sich die erforderlichen Toleranzen von weniger als ein Zehntel der Toleranzen der Zeichen-basierten Übertragung praktisch nicht mehr realisieren lassen. Aus diesem Grund muss der Bitaktgenerator während des Empfangs einer Botschaft nach-synchronisiert werden. Während bei Büro-LANs dazu spezielle Bit-Codierverfahren verwendet werden (z. B. Manchester-Codierung), die eine höhere Übertragungsbandbreite erfordern und größere EMV-Probleme verursachen, müssen bei der in Kfz-Bussystemen üblichen NRZ-Codierung zusätzliche Synchronisationsbits in den Datenstrom eingefügt werden. Ein Beispiel dafür ist das so genannte *Bit-Stuffing*, bei dem nach einer bestimmten Anzahl von Bits einer Botschaft (bei CAN z. B. 5 bit) immer dann ein zusätzliches Stuff-Bit eingefügt wird, wenn zu viele aufeinanderfolgende Datenbits alle *logisch 0* oder *logisch 1* sind und so im übertragenen Signal keine Signalflanke zur Nachsynchronisation des Bit-taktgenerators vorhanden wäre. Bit-Stuffing vergrößert die Länge einer Botschaft abhängig von den Datenbits, bei CAN z. B. im ungünstigsten Fall um bis zu 25 %.

2.1.6 Buszugriffsverfahren, Fehlererkennung und Fehlerkorrektur

Wie in Abschn. 2.1.1 bereits dargestellt, kommt es zu einer Kollision, wenn mehrere Steuergeräte gleichzeitig versuchen, über den Bus Daten zu senden. Alle verbreiteten Bussysteme müssen daher eine Strategie für den Buszugriff einsetzen, mit der Kollisionen erkannt bzw. vermieden werden. Gemeinsamer Grundsatz bei allen Kfz-Bussystemen ist dabei die Nicht-Unterbrechbarkeit laufender Übertragungen. Die gerade auf dem Bus übertragene Botschaft wird in jedem Fall vollständig übertragen. Jeder Sender muss überprüfen, ob der Bus frei ist, bevor er mit einem Sendeversuch beginnt. Welches Steuergerät senden darf, wird mit einem der in Tab. 2.2 dargestellten Zugriffsverfahren bestimmt.

Alle Protokolle verwenden Verfahren zur Fehlererkennung und teilweise auch Fehlerkorrektur auf verschiedenen Ebenen des Protokollstapels. Bei der Übertragung werden die gesendeten Datenbits praktisch immer mit den tatsächlichen Datenbits auf den Busleitungen verglichen, um Kollisionen und Übertragungsfehler zu erkennen. Bei der Zeichenübertragung können die einzelnen Zeichen durch Paritätsprüfung, ganze Botschaften durch Prüfsummen, in der Regel *Cyclic Redundancy Check CRC*, gesichert werden. Neben der Korrektheit der Daten werden auch die Zeitbedingungen des Übertragungsprotokolls überwacht (*Timeout*). In der Regel erhält der Sender eine Quittung für die fehlerfreie Übertragung vom Empfänger. Dies kann, wie bei CAN, durch ein unmittelbar am Ende der Botschaft gesendetes Quittungsbit, eine spezielle Bestätigungsbotschaft (*Acknowledge ACK*) oder im Rahmen der regulären Antwortbotschaft (*Response*) des Empfängers gesche-

Tab. 2.2 Buszugriffsverfahren

Master-Slave	Es gibt genau ein Steuergerät (Master), das eine Datenübertragung beginnen darf. Dieses Steuergerät fragt die anderen Steuergeräte (Slaves) regelmäßig oder bei Bedarf ab. Slaves dürfen Daten nur als Antwort auf eine an sie gerichtete Abfrage senden. Sie müssen selbst dann auf die Anfrage durch den Master warten, wenn sie sehr dringende Informationen zu senden haben. Untereinander können die Slaves nicht direkt kommunizieren. Wenn der Master ausfällt, ist entweder gar keine Kommunikation mehr möglich oder es muss ein Verfahren vorhanden sein, um aus den Slaves einen neuen Master zu bestimmen. Das Verfahren wird z. B. beim LIN-Bus verwendet.
Asynchron (Event Based)	Jedes Steuergerät kann zu beliebigen Zeiten auf den freien Bus zugreifen. Wenn mehrere Steuergeräte zeitgleich den Bus belegen, werden die elektrischen Signale auf dem Bus verfälscht. Da jedes Steuergerät zur Fehlererkennung ohnehin die Signale auf dem Bus überwacht, wird die Kollision erkannt und die Steuergeräte stellen die Übertragung sofort ein und versuchen es zu einem späteren Zeitpunkt nochmals.
CSMA/CD Carrier Sense Multiple Access Collision Detect	Mit diesem relativ einfach zu implementierenden Verfahren lässt sich bei zeitlich statistisch verteilten Botschaften im Mittel der beste Datendurchsatz erzielen. Aus diesem Grund wurde das CSMA/CD -Verfahren ursprünglich bei dem im Bürobereich dominierenden Ethernet-LAN verwendet. Das Verfahren ist allerdings nicht deterministisch. Da alle Botschaften gleichberechtigt sind und mit jeder anderen Botschaft kollidieren können, kann für keine Botschaft garantiert werden, wie lange es höchstens dauert, bis sie übertragen wird. Bei sehr hoher Busbelastung kann es infolge ständiger Kollisionen sogar dazu kommen, dass keine Daten mehr übertragen werden können.
CSMA/CR Carrier Sense Multiple Access Collision Resolution	Für die Anwendung im Kfz wird dieses Verfahren daher z. B. bei CAN so modifiziert, dass bei Kollisionen derjenige Sender „gewinnt“ (<i>Arbitrierung</i>), dessen Botschaft die höhere Priorität hat. Ein Sender mit einer Botschaft, die eine niedrigere Priorität hat, stellt das Senden sofort ein. Die Botschaft mit der höheren Priorität wird ohne Kollision weiter übertragen (CSMA/CR). Die Priorität einer Botschaft wird im Header der Botschaft codiert. Bei diesem Verfahren ist die Übertragung zumindest für Botschaften mit hoher Priorität deterministisch.
Token-Passing	Die Sendeberechtigung (Token) läuft reihum. Nach einem bestimmten Verfahren gibt ein Steuergerät, wenn seine Sendezzeit abgelaufen ist oder wenn es keine weiteren Daten zu senden hat, die Sendeberechtigung an ein anderes Gerät ab. Da das Weitergeben des Tokens überwacht werden sollte, ist das Verfahren relativ aufwendig und wird daher im Kfz nur bei den asynchronen Kanälen von MOST verwendet.

Tab. 2.2 (Fortsetzung)

Zeitsynchron (Time Based)	Jedes Steuergerät erhält ein bestimmtes periodisches Zeitfenster, in dem dieses und nur dieses Steuergerät senden darf. Wenn bestimmte Geräte mehr oder wichtigere Daten zu senden haben als andere, erhalten sie entweder längere Zeitfenster oder mehrere Fenster in kürzeren Zeitabständen. In Kfz-Anwendungen wird die Zuordnung der Zeitfenster in der Regel in der Entwicklungsphase statisch konfiguriert. Damit das Verfahren funktioniert, müssen alle Steuergeräte mit einer gemeinsamen Zeitbasis arbeiten.
TDMA Time Division Multiple Access	Im Gegensatz zu CSMA ist das Zeitverhalten der Übertragung streng deterministisch, d. h. es kann für jede Botschaft exakt vorhergesagt werden, wie lange es im ungünstigsten Fall dauert, bis sie übertragen wird. Wenn ein Steuergerät gerade keine Daten zu übertragen hat, bleibt sein Zeitfenster frei. Das Verfahren eignet sich besonders gut für periodische Übertragungen wie z. B. bei Messwerten und Regelkreisen, aber nur schlecht für seltene, spontane, aber eventuell sehr dringende Botschaften wie z. B. Fehlermeldungen, oder seltene, aber sehr lange Datenübertragungen, wie sie z. B. beim <i>Flashen</i> notwendig sind. Nach diesem Verfahren arbeiten z. B. FlexRay oder TTCAN, sehen aber aus den erwähnten Gründen ein spezielles Zeitfenster vor, in dem die Steuergeräte auch asynchron auf den Bus zugreifen können.

hen. Im Fehlerfall wiederholt der Sender meist die Übertragung. Der Empfänger verwirft die fehlerhaft empfangenen Daten.

Steuergeräte, die auch bei ausgeschalteter Zündung noch funktionsfähig sein müssen, z. B. die Steuerung für die Innenraumbeleuchtung, werden in der Regel in Ruheperioden in einen Stromsparmodus (*Stand By, Sleep*) versetzt, um die Batteriestandzeit zu verlängern. Durch ein externes Signal, z. B. den Türkontakt, werden sie wieder *aufgeweckt* und können dann andere Steuergeräte über spezielle oder beliebige Busbotschaften (*Wake Up*) ebenfalls wieder in den Normalbetriebszustand versetzen. Dies ist Aufgabe des sogenannten Netzmanagements, siehe Kap. 7.

2.1.7 Jitter und Latenz bei der Datenübertragung

Bei den meisten Anwendungen im Kfz ist eine zuverlässige Übertragung relativ kleiner Datenmengen unter Echtzeitaspekten (*Real Time*) notwendig. Im Gegensatz zu Büro-LANs steht nicht so sehr der absolute Datendurchsatz im Vordergrund, sondern die Frage, wie zuverlässig die Zeitbedingungen der Übertragung eingehalten werden:

- Bei periodisch übertragenen Daten, wie sie bei Regelungen üblich sind, muss die Periodendauer der Übertragung nicht nur für die gegebene Aufgabenstellung genügend klein sein, die tatsächliche Periodendauer der einzelnen Übertragungen darf auch nur bis zu einem vorgegebenen Maß schwanken (*Übertragungs-Jitter*).

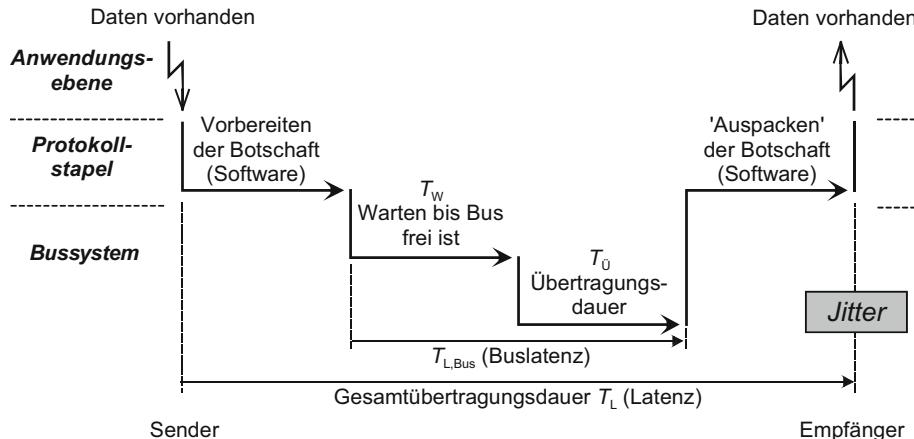


Abb. 2.17 Verzögerungen bei der Datenübertragung

- Bei nicht-periodisch zu übertragenden Daten, wie sie für Steuerungsaufgaben typisch sind, ist die maximale Verzögerung (*Übertragungs-Latency*) wichtig.

Neben der eigentlichen Übertragungsdauer auf dem Bus, die durch die Bitrate, die Datenmenge und den Protokolloverhead vorgegeben ist, ist vor allem das Bus-Zugriffsverfahren für die zeitliche Unsicherheit verantwortlich. Daher greifen viele Optimierungen insbesondere bei neueren Kfz-Bussystemen an dieser Stelle an und führen oft zu heftigen Glaubenskriegen um das *am besten geeignete Bussystem*.

Es darf jedoch nicht vergessen werden, dass wegen des Protokollüberbaus sowie der vielen anderen Aufgaben, die die relativ leistungsschwachen Mikroprozessoren in Kfz-Steuergeräten noch zu erfüllen haben, ein erheblicher Teil der Verzögerungszeiten auf die softwareseitige Verarbeitung der Daten zurückgeht. Dies gilt insbesondere, je komplexer das Übertragungsprotokoll ist und je mehr Teile der Protokollverarbeitung nicht im Kommunikationscontroller in schneller Hardware abgearbeitet werden können. Aus diesem Grund ist immer eine Gesamtbetrachtung von der Bereitstellung der Daten auf der Anwendungsebene beim Sender bis zu deren Verfügbarkeit auf der Anwendungsebene beim Empfänger notwendig (Abb. 2.17). Insbesondere bei Class C-Bussen dominieren in der Praxis meist die Zeit für die softwareseitige Bearbeitung und der dort ebenfalls entstehende Jitter im Vergleich zur reinen Busübertragung.

2.1.8 Elektrik/Elektronik-(E/E)-Architekturen

Die ersten vernetzten Fahrzeuge in den 1990er Jahren hatten eine simple Elektronikarchitektur, bei der einfach alle Steuergeräte an einen gemeinsamen CAN-Bus angeschlossen wurden (Abb. 2.18). Für die Diagnose gab es zusätzlich eine K-Line-Schnittstelle, die teil-

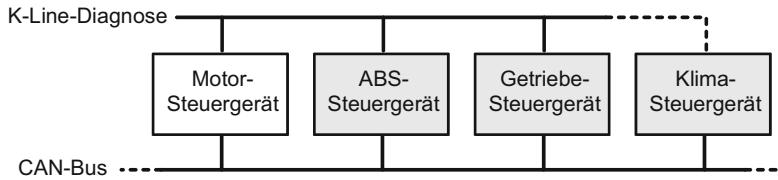


Abb. 2.18 Klassische Elektronik-Architektur

weise als Bus, teilweise aber auch für jedes einzelne Gerät nach außen geführt wurde. Zentrale Fahrzeugfunktionen wurden meist zusätzlich in das Motor-Steuergerät integriert, da dies das einzige Steuergerät war, das sicher in jedem Fahrzeug vorhanden war, während es sich bei den anderen Geräten oft um in unterschiedlichen Kombinationen anzutreffenden Sonderausstattungen handelte.

Nachdem immer mehr Steuergeräte eingebaut wurden, nahm die Busauslastung so weit zu, dass zuerst ein zweiter und später weitere Busse eingesetzt werden mussten. Die Busse wurden durch einzelne Steuergeräte verbunden, die an mehrere Bussysteme angeschlossen und zusätzlich zu ihren normalen Funktionen für den Datenaustausch zwischen den Bussystemen verantwortlich waren. Die Struktur der Systeme ergab sich primär aus dem Wunsch nach einer möglichst günstigen Verkabelung und aus den mehr oder weniger zufälligen Einbauorten der Geräte. In manchen Fällen war sie auch einfach ein Abbild der Organisationsstruktur des Herstellers. Weil die so gewachsenen Systeme schwer zu beherrschen waren und der Wildwuchs in der Steuergeräte-Kommunikation oft zu dubiosen Fehlerbildern führte, setzte sich in den 2000er Jahren allmählich eine klarer strukturiertere Architektur durch, bei der die Steuergeräte nach Anwendungsbereichen gruppiert und die Bussysteme über ein zentrales *Gateway* gekoppelt werden (Abb. 2.19). Die klassischen Anwendungsbereiche (*Domänen*) sind Triebstrang (*Powertrain*), Fahrwerk (*Chassis*), Karosserie (*Body*) sowie Unterhaltungselektronik und Navigation (*Infotainment*).

Mit dem Aufkommen von Fahrerassistenzsystemen wird die Trennung in die unterschiedlichen Domänen allerdings immer weniger möglich, weil die Assistenzsysteme häufig domänenübergreifende Funktionen benötigen und damit der Kommunikationsbedarf zwischen den einzelnen Domänen massiv ansteigt. Das zentrale *Gateway*-Steuergerät (Abb. 2.19) erweist sich dabei als Flaschenhals. Möglicherweise werden zukünftige Fahrzeuge daher die *Gateway*-Funktion wieder dezentralisieren und die einzelnen Gateways über einen Hochgeschwindigkeitsbus (*Backbone*) miteinander verbinden (Abb. 2.20). Gleichzeitig ist denkbar, dass die dezentralen *Gateways* nicht mehr als normale Steuergeräte, sondern als Hochleistungsrechner (*Domain Controller*) ausgeführt werden. Diese *Domain Controller* sollten dann sämtliche rechenleistungsintensiven Aufgaben übernehmen, während die übrigen Steuergeräte zu intelligenten Sensor-Aktor-Ansteuereinheiten vereinfacht werden könnten.

Das Konzept für die Aufteilung der Fahrzeugfunktionen auf verschiedene Steuergeräte, deren Vernetzung, die Optimierung der Einbauorte und Verkabelung sowie die Vertei-

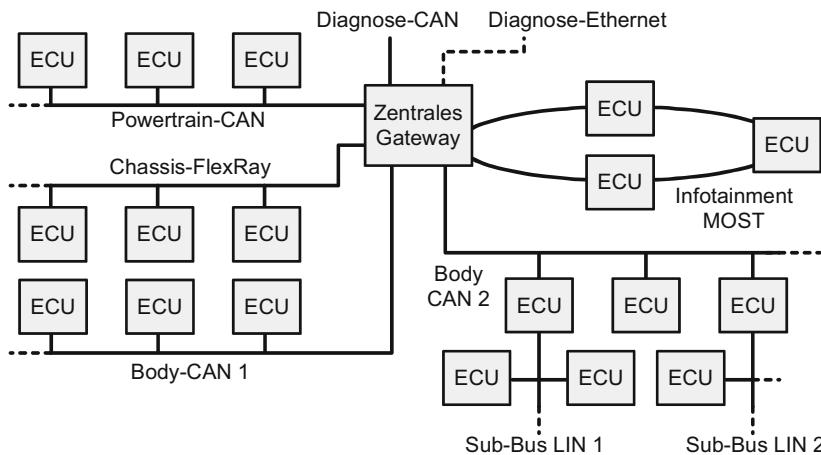


Abb. 2.19 Aktuelle heterogene Elektronik-Architektur

lung und Steuerung der notwendigen elektrischen Energie im Fahrzeug, die sogenannte Elektrik/Elektronik-(E/E)-Architektur, ist heute eine der komplexesten Aufgaben bei der Entwicklung einer Fahrzeugfamilie.

2.2 K-Line nach ISO 9141 und ISO 14230

K-Line ist das älteste in europäischen Kfz eingesetzte Busprotokoll für Diagnoseaufgaben. Es handelt sich um ein zeichen-orientiertes Protokoll (vgl. Abschn. 2.1.5), das sich mit den

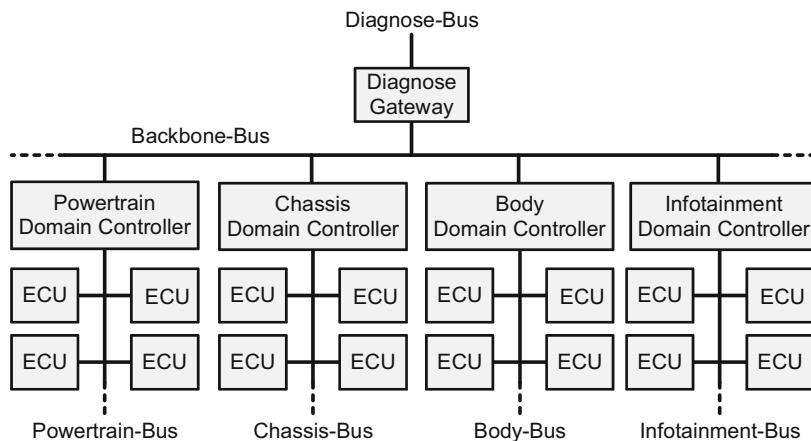


Abb. 2.20 Mögliche zukünftige Elektronik-Architektur

in nahezu jedem Mikrocontroller und jedem Rechner vorhandenen seriellen Schnittstellen auf UART-Basis einfach implementieren lässt.

2.2.1 Entwicklung von K-Line und KWP 2000

Die Schnittstelle ist in den 80er Jahren als Firmenstandard entstanden und wurde 1989 als ISO 9141 standardisiert. Der Standard definierte im Wesentlichen die elektrischen Eigenschaften, die Art der Bitübertragung und die Art der Kommunikationsaufnahme (Initialisierung, „Reizung“) zwischen Steuergerät und Diagnosetester. Während der Initialisierungsphase tauschen die Geräte ein so genanntes *Keyword* aus, durch das sich die beiden Geräte auf ein gemeinsames Datenprotokoll für den sich anschließenden Austausch von Diagnosedaten verstständigen. Die zulässigen Werte für das *Keyword* und die darauf aufbauenden Diagnoseprotokolle selbst sind nicht Bestandteil der ISO 9141-Norm und waren lange Zeit genauso wie die verwendeten Leitungsverbindungen und Steckverbinder extrem herstellerspezifisch und untereinander inkompatibel.

Anfang der 90er Jahre wurden in USA auf Vorschlag zunächst der kalifornischen und dann der Bundesumweltbehörden CARB und EPA unter der Bezeichnung OBD (On Board Diagnose) gesetzliche Vorschriften für die Überwachung emissionsrelevanter Komponenten im Kfz erlassen. Neben Vorgaben, welche Arten von Fehlern im Fahrbetrieb erkannt und wie diese Fehler dem Fahrer angezeigt und im Gerät gespeichert werden müssen, wird eine definierte Diagnoseschnittstelle gefordert, mit der Behörden, Polizei und Werkstätten Testgeräte an das Fahrzeug anschließen können, um die emissionsrelevanten Komponenten zu prüfen. Neben der von den amerikanischen Herstellern favorisierten SAE J1850 Schnittstelle akzeptierten die Behörden auf Drängen der europäischen Hersteller auch eine auf ISO 9141 basierende Schnittstelle. Dabei wurden die im ursprünglichen Standard enthaltenen Freiheitsgrade und Wahlmöglichkeiten mit dem Standardzusatz ISO 9141-2 (ISO 9141-CARB) eingeschränkt und präzisiert, wobei aber immer noch über herstellerspezifische Keywords erhebliche Protokollvarianten insbesondere für den nicht emissionsrelevanten Teil der Diagnose zulässig waren. Parallel mit der Weiterentwicklung der OBD-Vorschriften in USA übernahm Mitte der 90er Jahre die EU die amerikanischen Vorschriften in modifizierter Form (European OBD, OBD-II, OBD-III). In diesem Rahmen wurde unter dem Sammelbegriff *Keyword Protocol 2000* (KWP 2000) auch die Spezifikation der Diagnoseschnittstelle als ISO 14230 weiter präzisiert und auf die höheren Protokollebenen ausgedehnt. Der Standard umfasst die in Tab. 2.3 aufgeführten Teile.

Der Physical Layer und der Data Link Layer einschließlich der für emissionsrelevante Komponenten wesentlichen Einschränkungen werden in den folgenden beiden Abschnitten beschrieben, der Application Layer in Kap. 5.

Tab. 2.3 Normen für die KWP 2000 K-Line-Diagnose

ISO 14230-1	Physical Layer der allgemeinen Diagnoseschnittstelle für KWP 2000 (Kompatibel zu ISO 9141-2)
ISO 14230-2	Data Link Layer
ISO 14230-3	Implementierungshinweise, die dem Application Layer zuzuordnen sind.
ISO 14230-4	Einschränkungen für Physical und Data Link Layer bei der Diagnose emissionsrelevanter Komponenten (OBD)

2.2.2 K-Line Bus-Topologie und Physical Layer

K-Line ist ein bidirektonaler Ein-Draht-Bus, über den der gesamte Datenverkehr abgewickelt wird (Abb. 2.21). Optional ist zusätzlich eine weitere, aber nur unidirektionale Ein-Draht-Leitung, die L-Line, möglich, die aber nur für die Initialisierungsphase verwendet wird. Eine Mischung von Geräten mit und ohne L-Line innerhalb desselben Fahrzeugs ist zulässig. Der Diagnosetester kann entweder direkt mit dem fahrzeuginternen Bus verbunden oder über ein Gateway-Steuergerät geführt werden. Die Konfiguration mit Gateway-Steuergerät findet sich in der Praxis meist dann, wenn der fahrzeuginterne Bus nicht auf K-Line sondern auf CAN basiert. Der KWP 2000 Application Layer für CAN wurde in ISO/DIS 15765-3 so definiert, dass er mit dem KWP 2000 Application Layer nach ISO 14230 weitgehend kompatibel war.

Die Logikpegel auf K- und L-Line sind relativ zur Bordnetzspannung (Batteriespannung U_B) mit $> 0,8 U_B$ für *High* und $< 0,2 U_B$ für *Low* definiert (auf der Senderseite im Fahrzeug, beim Tester werden kleinere Toleranzen gefordert).

Diagnosetester werden heute üblicherweise auf Basis von PCs aufgebaut. Da deren serielle RS232 C-Schnittstelle mit anderen Logikpegeln und getrennten Sende- und Empfangsleitungen arbeitet, ist für den Tester ein interner oder externer Schnittstellenkonverter notwendig.

Die Bitübertragung selbst erfolgt Standard-UART kompatibel zeichenweise mit 1 Start-Bit, 8 Daten-Bits und 1 Stopp-Bit je Zeichen.

Obwohl die Steuergeräte über die K-Line theoretisch auch untereinander kommunizieren können, findet die Kommunikation in der Praxis nur zwischen dem Tester und einem Steuergerät statt, gegebenenfalls mit dem Gateway als Zwischenstation.

Die Bitrate (Baudrate) wird vom jeweiligen Steuergerät im Bereich 1,2 kbit/s bis 10,4 kbit/s frei vorgegeben, der Tester muss sich darauf einstellen. Die einzelnen Steuergeräte können mit unterschiedlicher Bitrate arbeiten. Für emissionsrelevante Steuergeräte wird eine feste Bitrate von 10,4 kbit/s gefordert. Für Sonderanwendungen in der Applikations- und Fertigungsphase (*Messen*, *Verstellen*, *Flashen*) werden, außerhalb der elektrischen Spezifikationen des Standards auch höhere Bitraten verwendet, was zu Übertragungs- und Kompatibilitätsproblemen führen kann.

Passend zur Bitrate werden im Standard auch zulässige Leitungskapazitäten, Innenwiderstände sowie Pullup- und Pulldown-Widerstände der Leitungstreiber, geforderte Über-

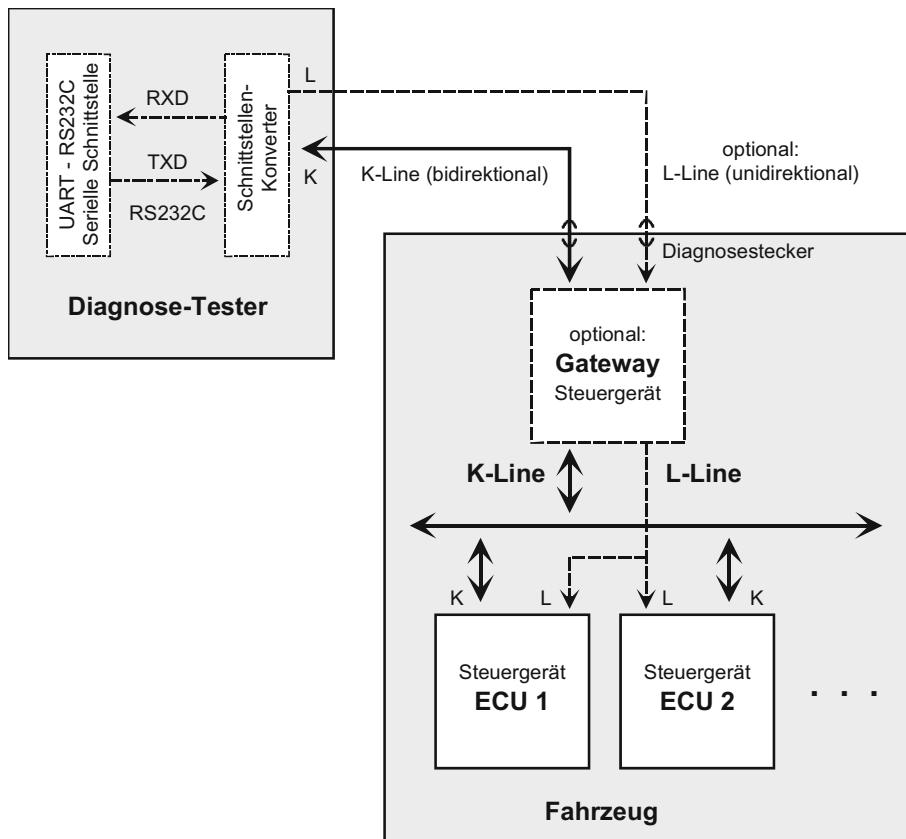


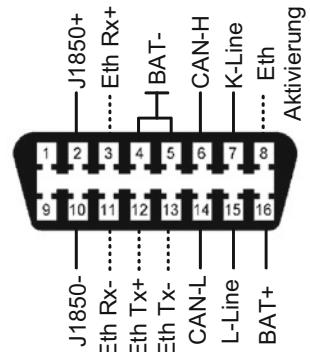
Abb. 2.21 K-Line Bus-Topologie (ohne Masse und Batterieverbindungen)

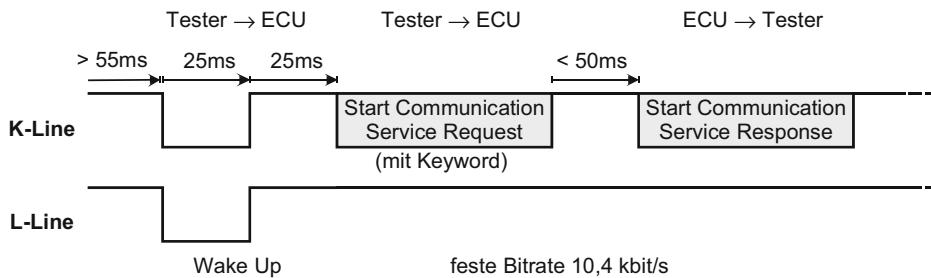
spannungsfestigkeit, Toleranzen der Bitrate usw. für eine maximale Bitrate von 10,4 kbit/s definiert. Die geforderten Innenwiderstände der Treiber für 12 V-Bordnetze (PKW, amerikanische LKW) und 24 V-Bordnetze (europäische LKW) sind unterschiedlich, doch können Treiberschaltungen für Steuergeräte so ausgelegt werden, dass sie für beide Bordnetzspannungen verwendbar sind. Dies gilt auf der Testerseite wegen der dort generell engeren Spezifikationen nur bedingt.

Die Form der Diagnosesteckdose ist durch ISO 15031-3 und SAE J1962 einheitlich vorgeschrieben (Abb. 2.22). Diese 16-polige Steckverbindung enthält Anschlüsse für Bus-Systeme nach ISO 9141, SAE J1850 und CAN sowie Batteriespannungsanschlüsse für den Tester. Die im Bild nicht bezeichneten Kontakte dürfen herstellerspezifisch belegt werden. Es ist zu erwarten, dass einige dieser Anschlüsse zukünftig durch weitere Bussysteme belegt werden, z. B. LIN, FlexRay oder Ethernet.

In Fahrzeugen amerikanischer Hersteller (z. B. Chrysler, Ford, GM) wurde lange SAE J1850 verwendet, in europäischen Fahrzeugen ISO 9141/ISO 14230 K-/L-Line, in neueren

Abb. 2.22 OBD-Diagnosesteckdose (Buchse) nach ISO 15031-3/SAE J1962



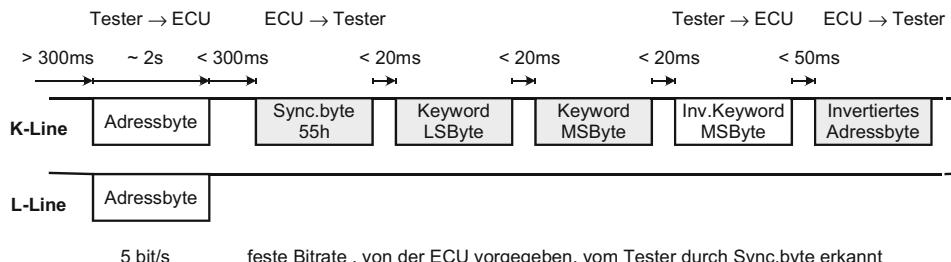
**Abb. 2.23** Schnelle Initialisierung

- Die K- und L-Line waren nach dem Einschalten für min. 300 ms bzw. nach dem Abbau der vorigen Verbindung für min. 55 ms im Ruhezustand *High*.
- Der Tester legt K- und L-Line gleichzeitig für 25 ms auf *Low* und anschließend für 25 ms wieder auf *High* (Wake Up Pattern).
- Die weitere Kommunikation findet nur noch auf der K-Line statt, die L-Line bleibt dauernd auf *High*.
- Der Tester sendet eine *Start Communication Service Request*-Botschaft (im normalen Botschaftsformat nach Abb. 2.25) mit der Adresse des Steuergerätes, zu dem die Verbindung aufgebaut werden soll.
- Das adressierte Steuergerät antwortet innerhalb von 50 ms mit einer *Start Communication Service Response*-Botschaft, die das so genannte Keyword (siehe unten) enthält.

Diese Initialisierungssequenz dauert ca. 100 ms.

Soll mit einer anderen Baudrate als 10,4 kbit/s gearbeitet werden, muss die so genannte *5 Baud Initialisierung* eingesetzt werden (Abb. 2.24):

- Die K- und L-Line waren für min. 300 ms im Ruhezustand *High*.
- Der Tester sendet gleichzeitig auf der K- und der L-Line mit der Bitrate 5 bit/s ein 8 bit langes Adresswort mit der Adresse des Steuergerätes, zu dem die Verbindung aufgebaut werden soll.

**Abb. 2.24** 5 Baud Initialisierung

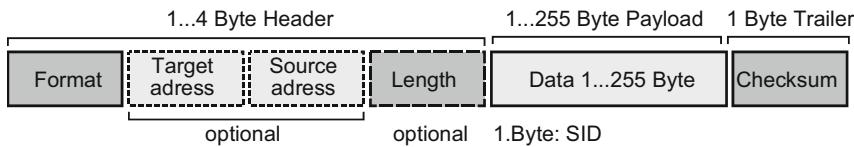


Abb. 2.25 KWP 2000 Botschaftsformat

- Die weitere Kommunikation findet nur noch auf der K-Line statt, die L-Line bleibt dauernd auf High.
- Das adressierte Steuergerät antwortet innerhalb von 300 ms, indem es ein Zeichen mit dem Synchronisationsmuster 55 h und anschließend im Abstand von jeweils max. 20 ms die beiden Bytes des Keywords (siehe unten) sendet. Das Steuergerät sendet dabei mit seiner eigenen, festen Bitrate im Bereich 1,2 ... 10,4 kbit/s.
- Der Tester misst die Bitdauer im Synchronisationsmuster und erkennt damit, mit welcher Bitrate das Steuergerät arbeitet. Er schaltet selbst auf diese Bitrate um und sendet das zweite Keybyte innerhalb von 20 ms invertiert als Echo zurück.
- Das Steuergerät quittiert den Verbindungsaufbau, indem es innerhalb von 50 ms seine invertierte Adresse als Echo zurücksendet.

Diese Initialisierungssequenz dauert ca. 2,5 s.

Der Verbindungsabbau erfolgt mit Hilfe einer *Stop Communication Service Request*-Botschaft (siehe unten) ausgehend vom Tester.

Datenaustausch, Botschaftsformat Die gesamte Kommunikation erfolgt nach dem Request-Response-Verfahren. Der Diagnosetester sendet eine Botschaft als Anfrage (*Request*) an das Steuergerät, das Steuergerät sendet eine Botschaft mit der gewünschten Antwort (*Response*) oder einer Fehlermeldung zurück. Die übertragenen Botschaften haben das in Abb. 2.25 dargestellte Format.

Mit dem Formatbyte wird angezeigt, ob die beiden optionalen Adressbytes und das Längenbyte folgen oder nicht. Das Längenbyte gibt an, wie viele Datenbytes folgen. Für Botschaften mit ≤ 63 Datenbytes kann die Datenlänge innerhalb des Formatbytes codiert und das Längenbyte weggelassen werden. Die Prüfsumme im Trailer ist die mod 256 Summe aller Bytes (außer dem Prüfsummenbyte selbst).

Über das Keyword zeigt das Steuergerät dem Tester während der Initialisierung an, welche Header-Optionen es erwartet und selbst verwenden wird. Die Bezeichnung KWP 2000 für das Protokoll resultiert aus dem Wertebereich 2000 bis 2031, den das Keyword haben darf. Zusätzlich zur Festlegung, ob die optionalen Adressbytes und/oder das optionale Längenbyte verwendet werden, signalisiert das Steuergerät, ob für die Timeout-Überwachung bei der Übertragung der normale oder ein erweiterter Timeout-Parametersatz (Tab. 2.4) verwendet werden soll. Mit Hilfe der *Access Timing Parameter Request*-Botschaft kann der Tester dann später auch andere Timeout-Parameter einstellen, sofern das Steuergerät dies unterstützt.

Tab. 2.4 Default-Timeout-Werte, Bezeichnungen der Norm in ()

Abstand zwischen ...	Min. ... Max.
... zwei Bytes innerhalb einer Botschaft (ECU: P1, Tester: P4)	0 ... 20 ms
... Tester-Anfrage und Steuergeräte-Antwort (P2)	25 ... 50 ms
... Steuergeräte-Antwort und nächster Tester-Anfrage (P3)	55 ms ... 5 s

Mit Ausnahme von Punkt-zu-Punkt-Verbindungen müssen Tester und Steuergeräte adressiert werden. Dabei wird über das Formatbyte zwischen so genannter funktionaler und physikalischer Adressierung unterschieden. Physikalische Adressen sind beliebige, innerhalb eines Fahrzeugs eindeutige 8 bit-Werte, die vom Kfz-Hersteller festgelegt werden. Funktionale Adressen sind in SAE J2178 definierte 8 bit-Wertebereiche für einzelne Klassen von Steuergeräten, z. B. 10 ... 17 h für Motorsteuergeräte, 18 ... 1Fh für Getriebe-steuergeräte, F0...FDh für Diagnosetester usw. Ein Steuergerät darf bei der Initialisierung und beim Empfang von Daten nur reagieren, wenn es seine eigene Adresse erkennt.

Dienste für die Datenübertragung (Communication Services) Das erste Byte der übertragenen Nutzdaten in einer Botschaft enthält den so genannten *Service Identifier* (SID), mit dem der Inhalt der Botschaft gekennzeichnet wird. Die folgenden Nutzdatenbytes enthalten dann die zugehörigen Parameter bzw. Antworten. Für jede Tester-Anfrage (Request) und für die zugehörige Steuergeräte-Antwort (Response) ist jeweils eine eigene SID definiert. Bei den Antworten gibt es in der Regel unterschiedliche SIDs für eine positive und für eine negative Antwort (Fehlermeldung).

Folgende Dienste müssen in der Protokollsoftware implementiert werden:

- **Start Communication Service Request** (SID 81 h): Verbindungsaufbau mit Übermittlung des Keywords
- **Stop Communication Service Request** (SID 82 h): Verbindungsabbau
- **Access Timing Parameter Request** (SID 83 h): Auslesen und Ändern der Timeout-Parameter der Datenübertragung. Das Ändern ist optional und muss von einem Steuergerät nicht unterstützt werden.
- **Send Data Service**: Senden eines beliebigen Diagnosedienstes. Der Datenblock mit bis zu 255 Datenbyte einschließlich SID wird vom Application Layer bereitgestellt bzw. beim Empfang an den Application Layer weitergereicht (s. h. Kap. 5).

Fehlerbehandlung Diagnosetester und Steuergerät müssen die empfangenen Botschaften auf Fehler (falsche Botschaftslänge, falsche Prüfsumme, Timeout Fehler) überprüfen:

- Steuergeräte ignorieren falsche Botschaften und senden keine Antwort (wird vom Tester über Timeout erkannt).
- Tester wiederholen eine Anfrage bei fehlerhaften oder ausbleibenden Antworten bis zu insgesamt 3 Mal.

Bei anderen Fehlern (falsches Header-Format, Fehler in den empfangenen Daten) sendet das Steuergerät eine negative Antwort.

Der Tester kann die Signale auf den Verbindungsleitungen überwachen und bei Fehlern die Übertragung wiederholen. Steuergeräte müssen derartige Fehler weder erkennen noch darauf reagieren.

2.2.4 Einschränkungen für emissionsrelevante Komponenten (OBD)

Abgasrelevante Systeme wie Motorsteuergeräte unterliegen einigen Einschränkungen:

- Target und Source Address Byte im Header müssen verwendet werden, das Längenbyte im Header entfällt, d. h. der Header besteht immer aus 3 Byte.
- Botschaften dürfen höchstens 7 Byte Nutzdaten haben.
- Tester-Anfragen sollen für die Zieladresse funktionale Adressen, Steuergeräteantworten müssen für die Zieladresse stets physikalische Adressen verwenden. Die Quelladresse ist immer eine physikalische Adresse.
- Nur die Default-Timeout-Werte dürfen verwendet werden. Abfragen oder Ändern der Timing Parameter ist nicht zulässig.
- Beide Initialisierungsarten sind zulässig, in jedem Fall aber erfolgt die Kommunikation anschließend mit der festen Bitrate von 10,4 kbit/s.

2.2.5 Schnittstelle zwischen Software und Kommunikations-Controller

Die K-Line-Schnittstelle wird praktisch immer über einen gewöhnlichen UART und nicht über einen speziellen Kommunikationscontroller realisiert. Übliche UARTs wickeln lediglich das Senden und Empfangen eines einzelnen Zeichens selbstständig ab, so dass praktisch das gesamte Protokoll inklusive der Timeout-Erzeugung und Überwachung in Software realisiert werden muss. Da die meisten Mikrocontroller-UARTs nicht über größere Sende- und Empfangspuffer verfügen, wird das Senden und Empfangen in der Regel zeichenweise interruptgesteuert abgewickelt. Kritisch dabei ist bei hoher CPU-Auslastung im Allgemeinen das Empfangen. Wenn der Diagnosetester mit den kürzest möglichen Abständen sendet, muss bei 10.4 kbit/s jede Millisekunde ein Zeichen aus dem UART ausgelesen und in einen im ungünstigsten Fall 260 Byte großen RAM-Puffer kopiert werden. Da die Checksumme erst am Ende der Botschaft überprüft werden kann, darf die Weiterverarbeitung der Botschaft eigentlich erst erfolgen, wenn alle Zeichen der Botschaft empfangen worden sind. Die Initialisierungssequenz macht spezielle Maßnahmen notwendig. Da sich UARTs in der Regel nicht auf die niedrige Bitrate von 5 bit/s für die 5 Baud-Initialisierung umprogrammieren lassen, muss auf der Steuergeräteseite die Empfangsleitung beim Warten auf eine 5 Baud-Initialisierungssequenz entweder im *Polling*-Verfahren oder über einen

Capture-Eingang überwacht werden. Für die Erkennung der schnellen Initialisierung lassen sich UARTs in der Regel umprogrammieren, da die Low-Phase von $25\text{ ms} \pm 1\text{ ms}$ der Übertragung eines 0 h-Zeichens bei ca. 360 bit/s mit einer Toleranz von $\pm 4\%$ entspricht, so dass der UART auf diese Baudrate eingestellt werden und einen regulären Empfangsinterrupt auslösen kann. Die anschließende High-Phase kann dadurch überwacht werden, dass anschließend für 25 ms kein weiteres Zeichen mehr empfangen wird, weil jeder Low-Pegel in dieser Zeit vom UART als Startbit eines weiteren Zeichens erkannt werden würde. Ohne vorangegangene Initialisierung dürfen empfangene Zeichen ignoriert werden.

Dieselbe Problematik ergibt sich auf der Testerseite für das Senden der Initialisierungssequenz, insbesondere wenn dieser auf PC-Basis realisiert wird. Da dort in der Regel keine Capture-Compare-Ein-/Ausgänge vorhanden sind, die üblichen PC-Betriebssysteme zu einem relativ großen Zeit-Jitter führen und weder 5 bit/s noch 10.4 kbit/s Standard-Baudaten sind, sind bei enger Interpretation der Zeitvorgaben in der ISO-Spezifikation spezielle Geräte-Treiber und/oder zusätzliche Hardware nötig.

2.2.6 Ältere K-Line-Varianten

Vor Veröffentlichung des KWP 2000-Protokolls wurden andere K-Line-Protokolle verwendet, die alle bezüglich der Bitübertragung und Verbindungsaufnahme auf ISO 9141 und der 5 Baud-Initialisierung beruhen, danach aber über ein spezifisches Keyword auf ein proprietäres Protokoll umschalteten. Weit verbreitet war das Keyword KW 71-Protokoll, das z. B. bei BMW eingesetzt wurde. Dieses Protokoll wurde für Bitraten von 1200 bit/s oder 9600 bit/s implementiert. Es definierte abhängig vom jeweiligen Fahrzeughersteller unterschiedliche Botschaften und Services. Während der Kommunikation fand ein laufendes Handshaking zwischen Diagnosetester und Steuergerät statt, indem das jeweils empfangene Zeichen in invertierter Form als Echo zurückgesendet wurde. Das Protokoll ist inzwischen ebenso veraltet wie die Keyword-Varianten KW 81 und 82 (Opel) oder KW 1281 (VW/Audi).

2.2.7 Zusammenfassung K-Line – Layer 1 und 2

- Bus- oder Punkt-zu-Punkt-Verbindung zwischen Diagnosetester und Steuergerät(en).
- Zeichenorientiertes, UART-basiertes Übertragungsprotokoll mit bidirektonaler Ein-Draht-Leitung (K-Line), Bitrate bis zu 10,4 kbit/s, Signalpegel U_B (Batteriespannung).
- Kommunikation zwischen Diagnosetester und genau einem Steuergerät. Tester und Steuergerät werden über eindeutige, feste Adressen identifiziert.
- Kommunikation wird vom Diagnosetester aufgebaut, Steuergerät gibt Bitrate und Protokolloptionen vor, Diagnosetester muss sich entsprechend einstellen.
- Kommunikation nach dem Request – Response – Verfahren, d. h. Tester stellt Anfrage (Request), Steuergerät antwortet (Response).

Tab. 2.5 Charakteristische Eigenschaften von SAE J1850 Bussystemen

	SAE J1850 PWM	SAE J1850 VPWM
Hauptvertreter	Ford	General Motors, Chrysler
Bit-Codierung	Pulsbreitenmodulation (PWM)	Variable Pulsbreitenmodulation (VPWM)
Bitrate	41,6 kbit/s	10,4 kbit/s (Mittelwert)
Datenleitung	Zwei-Draht (Twisted Pair)	Ein-Draht (Single Wire)
Signalpegel	5 V – Differenzsignal Low < 2,2 V, High > 2,8 V max. 6,25 V	U_{batt} unipolar Low < 3,5 V, High > 4,5 V Max. 20 V
Nutzdaten	0...8 Byte je Botschaft	
Botschaftslänge	max. 101 bit (inkl. Header u. Trailer)	
Buszugriff	CSMA/CA	

- Botschaften mit 1 ... 255 Datenbytes und 2 ... 5 Byte Header/Trailer.
- Abstand zwischen den Bytes einer Botschaft (Inter Character Break) max. 20 ms, Abstand zwischen Tester-Anfrage (Request) und Steuergeräte-Antwort (Response) 25 ... 50 ms. Abstand bis zur nächsten Tester-Anfrage min. 55 ms.
- Übertragungsdauer einer Botschaft mit 255 Datenbytes bei 10,4 kbit/s:
- Best Case: 250 ms, Worst Case: 5,5 s, d. h. Nutzdatenrate < 1 KB/s
- Überwachung durch Prüfsumme, Formatüberwachung und Timeout. Bei vom Tester erkannten Übertragungsfehlern max. 3 Übertragungsversuche. Bei vom Steuergerät erkannten Übertragungsfehlern Fehlermeldung an den Tester.
- Realisierung des Protokolls nahezu vollständig in Software. UART für Zeichenübertragung.

2.3 SAE J1850

SAE J1850 ist ein veraltetes, bitstrom-orientiertes Class A/B-Protokoll für die On- und Off-Board-Kommunikation, das vor allem von amerikanischen PKW-Herstellern eingesetzt wird [4, 5]. Bei genauer Betrachtung handelt es sich dabei eigentlich sogar um zwei auf der physikalischen und der Bitübertragungsschicht zueinander inkompatible Bussysteme, die aber einen gemeinsamen Data Link Layer verwenden (Tab. 2.5).

Im Gegensatz zu K-Line, CAN, LIN und FlexRay, die alle mit *Non Return to Zero* (NRZ) Bitcodierung arbeiten, bei der das Bussignal während einer Bitdauer konstant auf *Low* oder *High* bleibt, verwendet J1850 Impulssignale zur Bitcodierung.

In der PWM-Variante beginnt jedes Bit mit einem *Low-High*-Übergang (Abb. 2.26). Abhängig davon, ob als Datenbit eine 1 oder eine 0 übertragen werden soll, bleibt das Signal dann für 1/3 bzw. 2/3 der Bitdauer auf *High*, bevor es für den Rest der Bitdauer wieder auf *Low* wechselt. Die Bitdauer ist konstant. Bei 41,6 kbit/s ist $T_{bit} = 24 \mu s$.

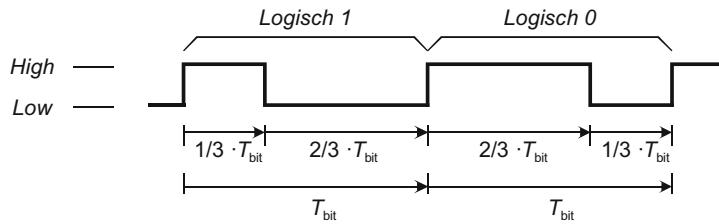


Abb. 2.26 Bitcodierung bei J1850 PWM

Tab. 2.6 Bitcodierung bei J1850 VPWM (Kombination von Dauer und Pegel)

Signalpegel	Signaldauer	Entspricht Datenbit
Low	$T_{\text{bit,short}}$	0
High	$T_{\text{bit,long}}$	0
High	$T_{\text{bit,short}}$	1
Low	$T_{\text{bit,long}}$	1

In der VPWM-Variante beginnt jedes Bit mit einem Übergang und endet mit dem nächsten Übergang. Der Abstand zwischen den Übergängen wird zwischen den beiden festen Werten $T_{\text{bit,short}} = 64 \mu\text{s}$ und $T_{\text{bit,long}} = 2 \cdot T_{\text{bit,short}} = 128 \mu\text{s}$ umgeschaltet. Für jedes der zu übertragenden Datenbits 0 bzw. 1 stehen dabei je zwei unterschiedliche Impulssignale zur Verfügung (Tab. 2.6).

Dabei wird immer diejenige Kombination zur Übertragung des nächsten Datenbits gewählt, die zu einem Wechsel des Signalpegels, d. h. einem Übergang auf der Leitung führt (Abb. 2.27). Die Bitrate ist damit nicht konstant, sondern hängt von der Reihenfolge der Datenbits ab. Im Mittel beträgt sie

$$\frac{1}{(T_{\text{bit,long}} + T_{\text{bit,short}})/2} = \frac{1}{96 \mu\text{s}} = 10,4 \text{ kbit/s}.$$

In beiden Fällen werden spezielle Bits (Start of Frame SOF, End of Data EOD, End of Frame EOF) zur Unterscheidung mit abweichender Impulsdauer codiert.

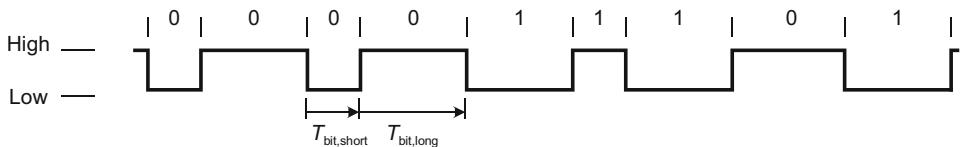


Abb. 2.27 Beispiel für die Übertragung der Datenbitfolge 000011101... bei VPWM

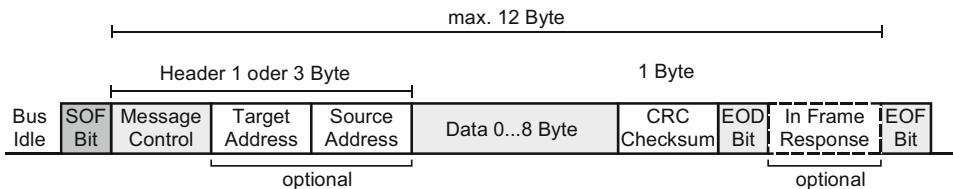


Abb. 2.28 SAE J1850 Botschaftsformat

Abbildung 2.28 zeigt das gemeinsame Botschaftsformat. Die Bits im *Message Control Byte* zeigen an, welche Priorität die Botschaft hat (0...7 mit 0 als höchster Priorität), ob ein 1 Byte (Chrysler) oder ein 3 Byte Header (GM, Ford) verwendet wird und ob eine *In Frame Response* (siehe unten) erwartet wird. Falls mehrere Sender gleichzeitig auf den Bus zugreifen, „gewinnt“ wie bei CAN die Botschaft mit der höchsten Priorität. *Target* und *Source Address Byte* sind bei reiner Punkt-zu-Punkt-Verbindung optional (angezeigt durch ein Bit im *Message Control Byte*). Wie bei KWP 2000 kann der Empfänger (*Target*) funktional oder physikalisch adressiert werden (ebenfalls angezeigt durch ein Bit im *Message Control Byte*). Die Senderadresse (*Source*) ist immer eine physikalische Adresse (siehe Abschn. 5.1.3). Zur Fehlererkennung dient eine 8 bit CRC-Prüfsumme.

Alternativ zum beschriebenen Header mit *Message Control Byte* und den (optionalen) *Target* und *Source Address Bytes* existiert eine Header-Form mit einem reinen 1 Byte-Header, bei dem der Header wie bei CAN als Message Identifier dient. Diese Form, bei der die Empfänger eine inhaltsbezogene Akzeptanzfilterung durchführen müssen, wird vorzugsweise für die On-Board-Kommunikation zwischen Steuergeräten im Fahrzeug eingesetzt.

Eine Besonderheit stellt die optionale *In Frame Response* (*IFR*) dar. Die *IFR* ermöglicht es dem Empfänger, ohne selbst eine eigene Botschaft mit Header usw. senden zu müssen, unmittelbar auf die empfangene Botschaft zu antworten. Die Antwort kann innerhalb der *IFR* optional ebenfalls durch eine CRC-Prüfsumme gesichert werden. Die *IFR* darf ein oder mehrere Bytes umfassen. Durch eine weitere Arbitrierung innerhalb der *IFR* ist es auch möglich, dass mehrere Empfänger mit jeweils 1 Byte auf die Botschaft antworten.

Die Gesamtlänge einer Botschaft für den Header, die Nutzdaten, die CRC-Prüfsumme und die *In Frame Response* (ohne die SOF, EOD und EOF Bits) ist beschränkt auf max. 12 Byte, d. h. bei dem in der Praxis üblichen 3 Byte Header und 8 Datenbytes kann die *In Frame Response* max. 1 Byte lang sein.

Das Format der Dienste für die On-Board-Kommunikation ist in SAE J2178 festgelegt, die hier nicht besprochen werden soll. Für die Off-Board-Kommunikation (OBD-Diagnose) werden die in SAE J1979 definierten Dienste verwendet, die inhaltlich identisch mit ISO 15031 sind und in Abschn. 5.3 erläutert werden.

Aufgrund der ungewöhnlichen Bitcodierung und der bitweisen Arbitrierung ist das SAE J1850-Protokoll trotz der relativ niedrigen Bitrate zu komplex, um es rein software-technisch mit Hilfe der Standardperipherie eines üblichen Mikrocontrollers sinnvoll zu

implementieren. Daher müssen ähnlich wie bei CAN zusätzlich zu den Bustransceivern spezielle J1850-Kommunikationscontroller als separate Bausteine oder auf dem Mikrocontroller integriert verwendet werden.

2.4 Sensor-Aktor-Bussysteme

Neben dem Drang zu immer höheren Übertragungsraten für Steuer- und Regelanwendungen, die zur Entwicklung von FlexRay geführt haben, gibt es auch im Bereich der Low-Cost-Bussysteme offensichtlich noch Einsatzgebiete, für die LIN (siehe Abschn. 3.2) nicht schnell genug, Low-Speed-CAN dagegen zu teuer zu sein scheint [6]. Jedenfalls haben sich in den letzten Jahren mehrere Industriekonsortien gebildet, die versuchen, neue Bussysteme zur Anbindung von Sensoren und Aktoren an Steuergeräte zu definieren und im Rahmen von SAE oder ISO zu standardisieren. Dabei sind praktisch alle großen Automobilelektronikhersteller aktiv und wirken in diesen Konsortien in wechselnder Rollenverteilung mit, obwohl oder weil die Standards teilweise miteinander konkurrieren. Da der Markt in diesem Bereich noch zu unübersichtlich ist, um beurteilen zu können, welche dieser Standards langfristige Bedeutung haben werden, soll hier nur ein kurzer Überblick gegeben werden.

2.4.1 SENT – Single Edge Nibble Transmission nach SAE J2716

Beim SENT-Bus, der seit 2007 als SAE J2716 standardisiert ist, handelt es sich um eine unidirektionale Verbindung, um Messwerte von einem intelligenten Sensor zu einem Steuergerät zu übertragen. SENT ist als Ersatz für Analog- oder PWM-Schnittstellen gedacht und benötigt wie diese mit Versorgungsspannung, Signalleitung und Masse mindestens drei Leitungsanschlüsse am Sensor. Ziel des SENT-Protokolls ist es, bei der Übertragung eine Auflösung von 12 bit zu gewährleisten, was bei Analog- oder einfachen PWM-Signalen schwierig ist.

Eine SENT-Botschaft besteht aus einem Synchronisierimpuls fester Länge (168 µs) sowie einer 4 bit Status-Information, sechs 4 bit-Datenwerten, d. h. zwei 12 bit Messwerten, und einer 4 bit CRC-Prüfsumme. Jeder 4 bit Wert $n = 0, 1, \dots, 15$ (*Nibble*) wird als ein Spannungsimpuls übertragen, bei dem der Abstand der negativen Signallanken gemäß $T_{\text{Impuls}} = 36 \mu\text{s} + n \cdot 3 \mu\text{s}$ variiert wird (Abb. 2.29). Für die Übertragung von zwei 12 bit Messwerten werden damit maximal $T_{\text{Botschaft}} = 816 \mu\text{s}$ benötigt. Die Nutzdatenrate liegt unter 3,7 KB/s, etwa das Dreifache des bei LIN möglichen Wertes. Da der Empfänger die Dauer des Synchronisierimpulses dynamisch ausmessen kann, dürfen auf der Sensorseite einfache RC-Taktgeneratoren verwendet werden.

Das erste Bit der Status-Information wird üblicherweise zur Fehlersignalisierung eingesetzt. Die restlichen Bits werden bei einfachen Sensoren herstellerspezifisch verwendet werden, um z. B. den Messbereich des Sensors zu codieren. Alternativ besteht mit dem sogenannten *Slow Serial Channel* die Möglichkeit, längere Zusatzinformationen zu über-

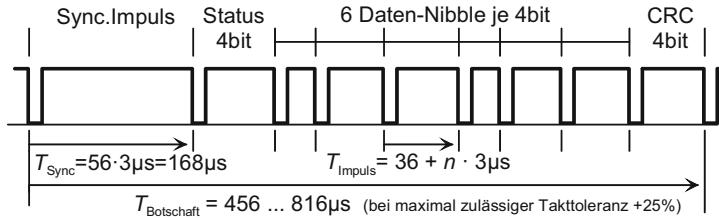


Abb. 2.29 SENT-Botschaft

tragen. Die Zusatzinformation wird über insgesamt 18 SENT-Botschaften verteilt, wobei jede Botschaft innerhalb des Status-Felds nur 2 bit dieser Zusatzinformation enthält, die auf der Empfangsseite dann wieder zur vollständigen Information zusammengefasst werden müssen. Innerhalb der Zusatzinformationen kennzeichnet ein 4 bzw. 8 bit langer *Identifier* den Inhalt der folgenden Daten. Vorgesehen sind *Identifier* um die SENT-Protokollversion, SensorTyp, Sensorhersteller, Seriennummern, Abgleichdaten des Sensors oder genauere Informationen zu aufgetretenen Fehlern (*Diagnostic Error Code*) zu übertragen.

2.4.2 PSI 5 – Peripheral Sensor Interface 5

In direkter Konkurrenz zu SENT steht PSI 5, das seine historischen Wurzeln im Airbag-Bereich hat und bei Beschleunigungssensoren verschiedene Vorgängerschnittstellen ablöst. Mittelfristig soll PSI 5 aber auch in anderen Bereichen der Sensorik im Fahrzeug verwendet werden. Seit Version 2.0 der Spezifikation gibt es daher neben dem Bereich *Airbag* auch Profile für die Anwendungsbereiche Fahrwerk (*Chassis and Safety Control*) und Antriebsstrang (*Powertrain*).

Im Gegensatz zu SENT benötigt der Sensor bei PSI 5 nur zwei Leitungsanschlüsse, weil das Signal durch Spannungs- bzw. Strommodulation über den Versorgungsspannungsanschluss übertragen wird (Abb. 2.30). Die PSI 5-Kommunikation ist in der Grundform wie bei SENT unidirektional. Vom Sensor zum Steuergerät besteht eine Punkt-zu-Punkt-Verbindung, wobei der Sensor seine Daten periodisch selbstständig sendet (asynchroner Betrieb). Alternativ ist aber auch eine Hintereinanderschaltung von Sensoren (*Daisy Chain*) oder ein Linienbus von einem Steuergerät zu mehreren Sensoren möglich, bei dem die Sensoren auf Anforderung des Steuergerätes senden (synchroner Betrieb). Im synchronen Betrieb ist auch eine eingeschränkte bidirektionale Kommunikation möglich.

Das Steuergerät versorgt den Sensor mit einer konstanten Spannung. Die Datenübertragung vom Sensor zum Steuergerät erfolgt, indem der Sensor seine Stromaufnahme um etwa $\Delta I = 26 \text{ mA}$ (13 mA in einer *Low-Power*-Variante) verändert (Abb. 2.30). Die Bitdarstellung erfolgt im Manchester-Code. Eine positive Flanke in Bitmitte überträgt eine „0“, eine negative Flanke eine „1“. An der Bitgrenze wird immer dann eine zusätzliche Signalflanke erzeugt, wenn das folgende Bit denselben Wert hat wie das vorherige Bit. Die Bitrate

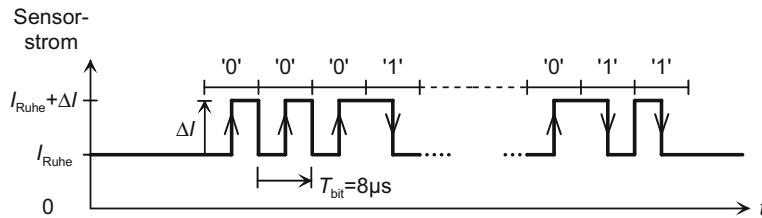


Abb. 2.30 PSI 5-Botschaft vom Sensor zum Steuergerät bei $f_{bit} = 125$ kbit/s

ist üblicherweise 125 kbit/s, darf aus Kompatibilitätsgründen zu einem älteren Protokoll aber auch auf 189 kbit/s eingestellt werden.

Eine vollständige Sensorbotschaft besteht aus 2 Startbits, einem Datenfeld mit n Bit und alternativ 1 Paritätsbit oder einer 3 bit CRC-Prüfsumme (Abb. 2.31a). Seit V2.x enthält das Datenfeld mindestens einen Sensorwert (10 ... 24 bit), optional einen zweiten Sensorwert (0 ... 12 bit), eine Statusinformation (0 ... 2 bit) sowie diverse Steuerbits (0 ... 6 bit), wobei die maximal zulässige Gesamtlänge von 28 bit nicht überschritten werden darf. In der Vorgängerversion V1.3 war das Datenfeld noch auf einen Sensorwert mit $n = 8$, 10 oder 16 bit oder zwei Sensorwerte mit je 10 bzw. 12 bit beschränkt, d. h. maximal 24 bit. Bei der Messwertübertragung wird nur ein Teil des Wertebereichs genutzt. Bei $n = 10$ beispielsweise werden Messwerte nur im Bereich $-480 \dots +480$ übertragen, der restliche Bereich ist für Statusinformationen reserviert. Der Wert +500 etwa dient zur Anzeige eines allgemeinen Sensordefekts.

Bei vergleichbaren Bedingungen wie bei SENT, d. h. $n = 24$ bit und CRC-Prüfsumme, dauert eine PSI 5-Botschaft bei 125 kbit/s nur 232 μs , während SENT im ungünstigsten Fall bis zu 816 μs benötigt.

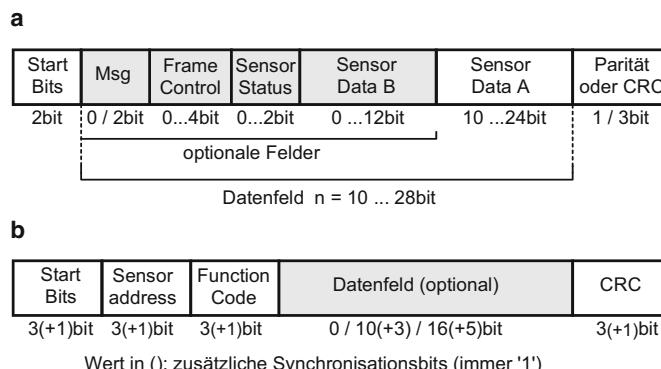


Abb. 2.31 PSI 5-Botschaftsformate. **a** Sensor → Steuergerät, **b** Steuergerät → Sensor

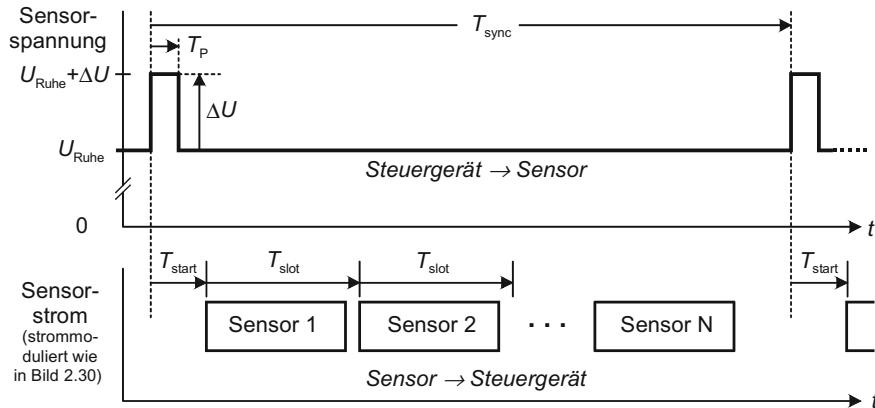


Abb. 2.32 PSI 5-Botschaften im synchronen Betrieb

Wie der *Serial Channel* bei SENT bietet auch PSI5 die Möglichkeit, außer den eigentlichen Messdaten weitere Informationen vom Sensor zum Steuergerät über insgesamt 18 PSI5-Botschaften verteilt zu übertragen. Je Botschaft werden dann im sonst optionalen Message Feld 2 bit der Zusatzinformation gesendet. Die gesamte Zusatzinformation besteht wiederum aus einem 4 bzw. 8 bit *Identifier*, einer 6 bit CRC-Prüfsumme, einigen Steuerbits sowie den eigentlichen Daten (12 bzw. 16 bit).

Im asynchronen Betrieb bestimmt der Sensor selbst, wann und wie oft er Daten sendet. Im alternativen synchronen Betrieb sendet der Sensor immer nur, wenn das Steuergerät die Versorgungsspannung für den Sensor für ca. 30 μ s um mindestens 2,5 V (bei V1.3 waren es 3,5 V) anhebt (Abb. 2.32). Auf diese Weise ist auch ein Busbetrieb möglich, bei dem mehrere Sensoren in vordefinierten Zeitschlitzten (*Slots*) hintereinander senden. Die Reihenfolge der Sensoren, die Anzahl und Dauer der Zeitschlitzte und der Abstand T_{sync} der Synchronisationsimpulse können für jede Anwendung individuell festgelegt werden.

Über die Synchronisationsimpulse ist auch eine langsame Datenübertragung vom Steuergerät zu den Sensoren möglich, d.h. PSI 5 kann damit bidirektional arbeiten. Dabei werden die Synchronisationsimpulse moduliert. Beim sogenannten *Tooth Gap* Verfahren überträgt der eigentliche Synchronisationsimpuls eine logische „1“, während ein fehlender Synchronisationsimpuls als logisch „0“ interpretiert wird. Dieses Verfahren ist allerdings nur bei konstanten Werten von T_{Sync} möglich. Um auch variable Abstände der Synchronisationsimpulse zu erlauben, wenn eine Sensorabfrage beispielsweise drehzahlabhängig erfolgen soll, ist seit Protokollversion V2.0 alternativ auch eine Pulsdauermodulation (*Pulse-Width-Modulation PWM*) der Synchronisationsimpulse möglich. Dabei überträgt ein kurzer Impuls von ca. $T_P = 30 \mu$ s eine logische „0“, ein langer Impuls von ca. 60 μ s eine logische „1“.

Das Botschaftsformat für die Steuergeräte-Sensor-Kommunikation enthält 3 Startbits, ein Adress- und Befehlsfeld sowie ein Datenfeld variabler Länge, eine 3bit CRC Prüfsumme

sowie ein Antwortfeld für den Sensor (Abb. 2.31b). Über das Adressfeld kann ein einzelner Sensor ausgewählt werden, dem über das *Function Code* Feld ein Befehl übermittelt werden soll. Die Antwort des Sensors erfolgt dann in den Synchronisationsimpuls-Zyklen, die auf das letzte CRC-Bit der Steuergerätebotschaft folgen. Damit die Sensoren den Beginn einer Steuergerätebotschaft eindeutig erkennen können, müssen zunächst vor Beginn der eigentlichen Botschaft mindestens 5 Synchronisationsimpulse mit „0“ oder 31 Impulse mit „1“ vorliegen. Außerdem wird nach jedem dritten Bit in die Steuergerätebotschaft ein „1“ Bit eingefügt.

Da unterschiedliche Sensortypen existieren und das Protokoll eine Reihe von Parametern und Optionen bietet, ist ein Initialisierungsmechanismus vorgesehen. Dabei sendet ein Sensor nach dem Einschalten der Versorgungsspannung zunächst wiederholt eine Initialisierungsbotschaft, die Informationen über den Sensor und das verwendete Protokoll enthält, bevor die normale Messdatenübertragung beginnt.

Auf der physikalischen Ebene und bei der Bustopologie dagegen sind einige Kombinationen grundsätzlich unverträglich. Seit Protokollversion 2.x werden daher Profile für verschiedene Anwendungsgebiete definiert. So erlaubt das Profil für Airbag-Anwendungen beispielsweise alternativ Linienbus- oder Daisy-Chain-Topologien, während im Powertrain-Bereich nur Linienbusse empfohlen werden. Für die Übertragung vom Steuergerät zum Sensor ist bei Airbags nur die Tooth-Gap-Methode erlaubt, während in anderen Anwendungen auch die PWM-Methode zugelassen wird. Airbag-Systeme sollen im synchronen Modus möglichst mit einer festen Periodendauer von $500\text{ }\mu\text{s}$ arbeiten, während bei Powertrain auch eine variable Periodendauer in Abhängigkeit der Motordrehzahl eingesetzt werden darf. Da reale Sensoren und Steuergeräte kaum alle in der Protokollspezifikation vorgesehenen Möglichkeiten unterstützen werden, muss im Einzelfall sehr genau geprüft werden, ob die vorgesehenen PSI5-Komponenten tatsächlich vollständig untereinander kompatibel sind.

2.4.3 ASRB 2.0 – Automotive Safety Restraint Bus (ISO 22898)

Im Gegensatz zu SENT und PSI 5 erlaubt das vom Safe-by-Wire Plus Konsortium definierte ASRB 2.0 Protokoll eine vollwertige bidirektionale Kommunikation und eignet sich daher nicht nur für Sensoren sondern auch für Aktoren, die von einem Steuergerät angesteuert werden müssen. Wie PSI 5 stammt es aus dem Airbag-Bereich, kann aber prinzipiell für beliebige Sensoren und Aktoren eingesetzt werden. Die Bustopologie ist sehr flexibel. Neben dem klassischen Linienbus sind bei ASRB auch Daisy-Chain, Ring- und Baumstrukturen möglich.

ASRB ist ein echtes Master-Slave-Bussystem, bei dem über eine Zwei-Draht-Verbindung gleichzeitig Versorgungsspannung und Signal übertragen werden. Im Gegensatz zu PSI 5 wird dabei der Spannungspegel moduliert (Abb. 2.33). Für die erste Hälfte des Bittaktes legt das *Master*-Steuergerät den Bus niederohmig auf eine Spannung von ca. 11 V. Diese *Power* Phase dient der Spannungsversorgung der Sensoren und Aktoren (*Slaves*). In der zweiten

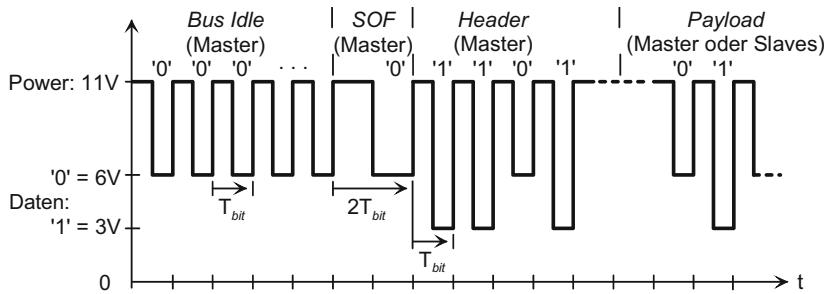


Abb. 2.33 Bitübertragung bei ASRB 2.0

Hälften des Bittaktes, der *Data Phase*, erfolgt die Bitübertragung. Dabei treibt der *Master* den Bus hochohmig mit 6 V. In dieser Phase kann ein Sender den Bus entweder auf 6 V belassen und damit ein „0“ Datenbit senden, oder ihn niederohmig auf 3 V herunterziehen und damit ein „1“ Datenbit senden. Sender darf dabei prinzipiell jeder Busteilnehmer sein. Mögliche, stets vom *Master*-Steuergerät vorgegebene Bitraten sind 20, 40, 80 und 160 kbit/s.

Bereits im Ruhezustand sendet das *Master*-Steuergerät ständig „0“ Datenbits. Der Beginn einer Botschaft wird durch das *Start of Frame (SOF)* Muster angezeigt, bei dem die Bitdauer vom Master verdoppelt wird (Abb. 2.33). Danach folgt der eigentliche Botschaftsheader, der ebenfalls ausschließlich vom *Master* gesendet wird. In der zweiten Hälfte der Botschaft senden je nach Zweck der Botschaft entweder wiederum der *Master* oder einer oder mehrere der *Slaves*. Das Botschaftsschema ähnelt also dem Verfahren bei LIN mit einer *Header-Phase*, die immer vom *Master* gesendet wird, und einer *Payload-Phase*, in der beliebige Busteilnehmer senden können.

Es gibt zwei Botschaftarten, die vom *Master* über den Wert des *SOF*-Bits signalisiert werden (Abb. 2.34). Bei Botschaften zur Abfrage von Sensordaten, sogenannten *S-Frames*, beginnt das *Master*-Steuergerät die Botschaft mit *SOF* = 0 und zwei weiteren *Header*-Bits *MSA* und *SEL*. Daraufhin sendet der erste Sensor seine Daten und eine *CRC*-Prüfsumme, anschließend der zweite Sensor usw. Wie viele Sensoren abgefragt werden, welcher Sensor in welchem Zeitschlitz (*Slot*) antwortet und wie viele Bits die Sensordaten bzw. die *CRC*-Prüfsumme umfassen, wird in der Entwicklungsphase festgelegt und muss dem *Master* und den *Slaves* bekannt sein. *S-Frames* werden in der Regel vom *Master* periodisch gesendet. Schnelle Sensoren liefern bei jeder Abfrage einen neuen Messwert, langsame Sensoren können sich paarweise einen Zeitschlitz teilen und nur bei jeder zweiten Abfrage antworten (*Slot Multiplexing*). Welcher der beiden Partner eines Paares antworten soll, gibt der *Master* im *Header* über das *Select Bit SEL* vor. Bei einer mit den SENT und PSI 5-Beispielen vergleichbaren Konfiguration (Abfrage von 2 Sensor-Messwerten mit je 12 bit und 3 bit *CRC* sowie einer Bitrate von 80 kbit/s) dauert die Übertragung der Messwerte ca. 425 μ s.

Optional kann der *Master* auch einen beliebigen Sensor gezielt abfragen. Dazu setzt er im *Header* das *Multiple Sharing Bit MSA* = 1 und sendet statt eines Slaves im ersten Zeitschlitz selbst ein 6 bit langes Sensor-Adresswort. In den restlichen Zeitschlitzten antworten

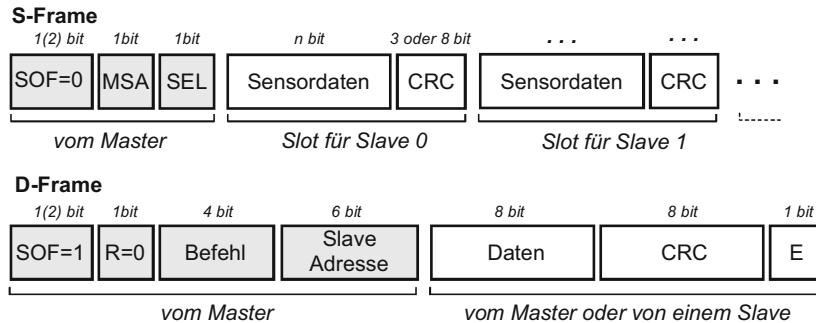


Abb. 2.34 Botschaftsaufbau (SOF-Bit wird mit doppelter Bitdauer gesendet)

die Sensoren wie üblich mit ihren Messdaten. Bei der nächsten Abfrage setzt der *Master* im *Header* wieder *MSA* = 0, worauf der vorher adressierte Sensor im ersten Zeitschlitz seine Messdaten einfügt. Alle übrigen Sensoren antworten bei beiden Botschaften in ihren jeweiligen Zeitschlitzten wie gewöhnlich. *D-Frame* Botschaften dienen zum Lesen oder Schreiben von Daten. Der *Slave* wird über eine eindeutige 6 bit Adresse im *Header* der Botschaft adressiert. Dabei kann lediglich ein einzelnes Datenbyte übertragen werden, das aufwendig über eine 8 bit CRC-Prüfsumme gesichert wird. Ob gelesen oder geschrieben werden soll, wird über das 4 bit lange Befehlsfeld signalisiert. Um den Datenwert lesen oder schreiben zu können, muss der *Master* mit einem *Write Pointer* Befehl zunächst die Speicheradresse im *Slave* auswählen und anschließend in einer zweiten Botschaft mit einem *Read Memory* oder *Write Memory* Befehl den Datenwert lesen bzw. schreiben. Falls mehr als 256 Byte adressiert werden sollen, muss zuvor über eine weitere Botschaft mit dem *Write Page* Befehl der Adressbereich (*Page*) ausgewählt werden. Die Sensor- bzw. Aktor-Konfiguration ist in einem definierten Adressbereich abgelegt und kann vom *Master*-Steuergerät ausgelesen bzw. neu programmiert werden. Am Ende der Botschaft wird das *Error Bit* E = 0 gesendet. Übertragungsfehler können von einem Busteilnehmer signalisiert werden, indem er es mit E = 1 überschreibt. Für das Zünden von Airbags sowie die sofortige Alarmierung des Masters durch einen Crash-Sensor existieren eine Reihe von speziellen Mechanismen.

Mit dem Aufkommen von PSI5 und SENT ist es um ASRB ruhiger geworden. Der zu gehörige ISO 22896 Standard ist seit 2006 unverändert.

2.4.4 DSI – Distributed Systems Interface

DSI ist der Vorschlag eines weiteren Firmenkonsortiums, das mit Rückhaltesystemen auf dieselben Anwendungen zielt wie ASRB und PSI5. DSI verbindet ein Master-Steuergerät mit bis zu 15 Slave-Geräten. In der Protokollversion DSI 2.5 sendet der Master ein spannungsmoduliertes Signal, bei dem die Bitinformation im Tastverhältnis enthalten ist. Das

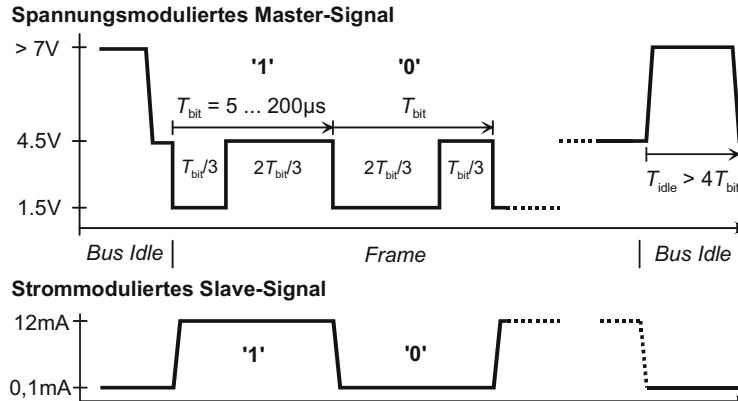


Abb. 2.35 Bitübertragung bei DSI 2.5

Spannungssignal dient gleichzeitig zur Versorgung der Slaves (Abb. 2.35). Die Kommunikation ist bidirektional. Ein Slave antwortet, indem er im vom Master vorgegebenen Bittakt seine Stromaufnahme zwischen 0,1 und 12 mA moduliert.

Das Standard-Botschaftsformat des Masters enthält ein Datenbyte, die 4 bit lange Slave-Adresse und einen 4 bit langen Befehlscode sowie eine 4 bit CRC-Prüfsumme (Abb. 2.36).

Der Slave antwortet zeitversetzt, d. h. wenn der Master bereits die nächste Befehlsbotschaft sendet. Die Antwort enthält zwei Datenbyte und wieder eine 4 bit CRC-Prüfsumme. Durch einen speziellen Befehl kann der Master das Format der Botschaften umschalten. Das Standard-Format wird als *Standard Long Word* bezeichnet. Beim *Short Word Format* entfällt jeweils das erste Byte, beim *Enhanced Format* ist die Daten- bzw. CRC-Länge konfigurierbar.

Mit DSI 3 wurde 2011 ein massiv modifiziertes Nachfolgeprotokoll vorgestellt, das sich als Alternative zu PSI 5 positioniert. Das Grundkonzept der Spannungs- und Strommodulation wird beibehalten, doch ändern sich die Signalpegel und das Zeitverhalten. Es werden zwei Geräteklassen und zwei Übertragungsmodi unterschieden. Im einfachsten Fall (*Sensor Class*) sind nur Sensoren an das Bussystem angeschlossen, die periodisch Messdaten zum Steuergerät (*Master*) übertragen. Dieser *Periodic Data Collection Mode* entspricht annähernd dem synchronen Betrieb von PSI 5. Im Ruhezustand versorgt das DSI 3-

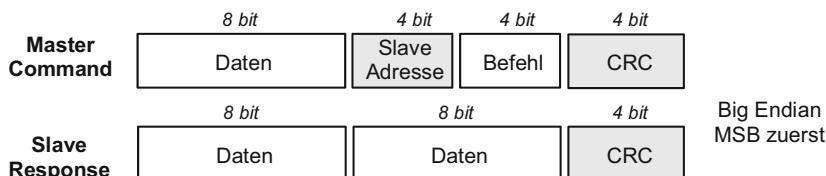


Abb. 2.36 Botschaftsformat beim Distributed Systems Interface DSI 2.5

Spannungsmodulierter Forward Channel: Master zu Slave

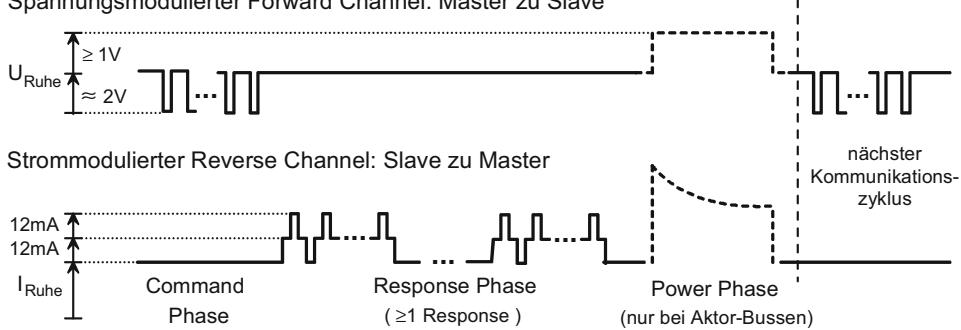


Abb. 2.37 Bitübertragung bei DSI 3

Steuergerät die Sensoren mit einer konstanten Spannung und die Sensoren halten ihre Ruhestromaufnahme konstant (Abb. 2.37). Zur Synchronisation reduziert das Steuergerät periodisch kurzzeitig die Spannung um etwa 2 V. Daraufhin senden die Sensoren zeitlich hintereinander ihre Messdaten. Die Bitcodierung erfolgt wiederum durch Strommodulation, wobei der jeweilige Sensor seine Stromaufnahme um 12 bzw. 24 mA erhöht. Um Übertragungsfehler zu erkennen, dürfen nicht alle möglichen Kombinationen der drei Strompegel verwendet werden, so dass im Schnitt etwa 4 bit je 3 Takte übertragen werden. Der Bittakt beträgt typ. 125 kbit/s, d. h. $T_{bit} = 8 \mu s$. Das Format der Messdatenbotschaft (Periodic Data) ist in weiten Grenzen konfigurierbar (Abb. 2.38). Die Zyklusdauer ist 100 μs bis 5 ms.

Statt der einfachen Synchronisationsimpulse kann das Steuergerät auch langsam selbst Daten an die Sensoren schicken (*Background Diagnostic*), dies ist vergleichbar mit dem *Serial Channel* von PSI 5 und SENT. Für die Bitübertragung des Steuergeräts wird eine Manchester-Codierung (vergl. Abb. 2.30) verwendet, wobei DSI 3 wie bei den einfachen Synchronisationsimpulsen die Sensorspannung um mindestens 2 V moduliert (Abb. 2.37). Die vom Steuergerät gesendete *Command Botschaft* wird dabei über mehrere Perioden verteilt (konfigurierbar). Damit die Sensoren neben den Messdaten auch explizit auf die *Command Botschaften* mit *Response Botschaften* antworten können, müssen dafür Zeitschlüsse der *Response Phase* reserviert werden.

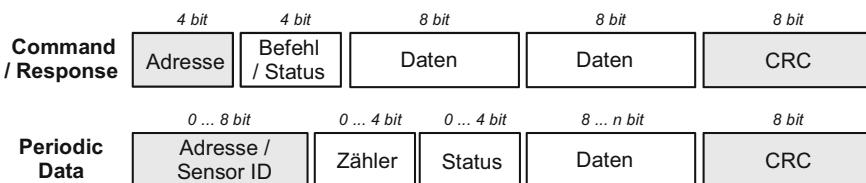


Abb. 2.38 Botschaftsformat bei DSI 3

Sind am Bus nicht nur Sensoren sondern auch Aktoren angeschlossen, ist die Latenz bei der Übertragung von *Command Botschaften* oft kritisch und die Stromaufnahme von Aktoren ist erheblich höher als bei einfachen Sensoren. Für solche *Power Class* Anwendungen ist der reine *Command Response* Übertragungsmodus vorgesehen. Das Steuergerät sendet durch Spannungsmodulation eine vollständige *Command Botschaft*, worauf ein oder mehrere Sensoren mit *Response Botschaften* im Format nach Abb. 2.38 antworten. Das frei konfigurierbare *Periodic Data Format* kann in diesem Modus nicht verwendet werden. Zusätzlich wird je Zyklus eine *Power Phase* eingefügt, in der das Steuergerät die Versorgungsspannung um mindestens 1 V anhebt. Die Aktoren können in dieser Phase ihre internen Pufferkondensatoren aufladen, dabei muss das Steuergerät bis zu 400 mA liefern. Aufgrund der hohen, nicht konstanten Stromaufnahme ist in dieser Phase keine Datenübertragung möglich.

2.5 Normen und Standards zu Kap. 2

K-Line	ISO 9141 Road Vehicles – Diagnostic Systems – Requirements for interchange of digital information, 1989, www.iso.org , ISO 9141-2 Road Vehicles – Diagnostic Systems – Part 2: CARB Requirements for interchange of digital information, 1994 und 1996, www.iso.org ISO 9141-3 Road Vehicles – Diagnostic Systems – Part 3: Verification of the communication between vehicle and OBD II scan tool, 1998, www.iso.org ISO 14230 siehe Kap. 5
SAE J1850	SAE J1850 Class B Data Communications Network Interface, 2006, www.sae.org
Sensor-Aktor-Busse	SAE J2716 SENT – Single-Edge Nibble Transmission for Automotive Applications, 2010, www.sae.org PSI 5 Peripheral Sensor Interface for Automotive Applications, Technical Specification V1.3, 2008, www.psi5.org PSI 5 Peripheral Sensor Interface for Automotive Applications, Technical Specification V2.0, 2011, www.psi5.org , Grundstandard und domänen spezifische Dokumente für Airbag, Vehicle Dynamics und Powertrain PSI 5 Peripheral Sensor Interface for Automotive Applications, Technical Specification V2.1, 2012, www.psi5.org , Grundstandard sowie domänen spezifische Dokumente Safe-by-Wire Plus: Automotive Safety Restraints Bus Specification (ASRB) V2.0, 2004, www.nxp.com/acrobat_download/other/automotive/safe_by_wire_plus.pdf ISO 22896 Road Vehicles – Deployment and sensor bus for occupant safety systems (entspricht ASRB V2.0), 2006, www.iso.org DSI (Distributed Systems Interface Standard) Bus Standard V2.5, 2009, www.dsiconsortium.org DSI3 Bus Standard Rev. 1.0, 2011, www.dsiconsortium.org

Literatur

- [1] A. S. Tanenbaum: Computer Networks. Prentice Hall, 5. Auflage, 2010
- [2] G. Schnell, B. Wiedemann: Bussysteme in der Automatisierungs- und Prozesstechnik. Springer-Vieweg Verlag, 8. Auflage, 2011
- [3] O. Strobel (Hrsg.): Communication in Transportation Systems. IGI Global EBook, <http://www.igi-global.com>, 2013
- [4] R. K. Jurgen (Hrsg.): Automotive Electronics Handbook. McGraw Hill Verlag, 2. Auflage, 1999
- [5] R. K. Jurgen (Hrsg.): Multiplexing and Networking. SAE Verlag, 1999
- [6] D. Paret: Multiplexed Networks for Embedded Systems: CAN, LIN, FlexRay, Safe by Wire. Wiley & Sons Verlag, 1. Auflage, 2007

3.1 Controller Area Network CAN nach ISO 11898

CAN ist das derzeit am häufigsten eingesetzte Kfz-Bussystem sowohl für Low-Speed- als auch für High-Speed-Anwendungen.

3.1.1 Entwicklung von CAN

CAN (Controller Area Network) wurde von Bosch in der zweiten Hälfte der 80er Jahre entwickelt und seit 1991 als erstes Class C-Netz in Fahrzeugen eingesetzt [1, 2]. Kurz danach wurde die Spezifikation mit der Einführung der *29 bit Identifier* nochmals überarbeitet und stellt in der als CAN 2.0A und 2.0B von Bosch veröffentlichten Form bis heute die Basis aller existierenden CAN-Implementierungen dar. Mit dem ISO-Standard ISO 11898 und dem SAE-Standard SAE J2284 wurde das Protokoll für die Anwendung im PKW und mit dem SAE-Standard SAE J1939 für die Anwendung im NKW international standardisiert. Während CAN sich bei den europäischen PKW-Herstellern und bei praktisch allen NKW-Herstellern weltweit sehr schnell durchgesetzt hat, behielten amerikanische PKW-Hersteller auf dem heimischen Markt lange den Class B-Bus SAE J1850 bei. Allmählich stellen aber auch diese Hersteller vollständig auf CAN um. Bei Neufahrzeugen ist CAN ab 2008 die einzige zugelassene Schnittstelle für Diagnosetester bei abgasrelevanten Komponenten (OBD).

Die erste CAN-Implementierung entstand in Zusammenarbeit zwischen Bosch und Intel. Bosch hat aber bereits frühzeitig eine offene Lizenzpolitik betrieben, so dass mittlerweile praktisch jeder Mikrocontroller-Hersteller CAN-Module anbietet und sogar CAN-VHDL-Module für die Integration in ASICs und FPGAs zur Verfügung stehen. Bei der Lizenzierung stellt Bosch ein Referenzmodell und eine Testprozedur zur Verfügung, so dass sichergestellt ist, dass alle CAN-Controller kompatibel miteinander kommunizieren können.

Tab. 3.1 Wichtige CAN-Standards

ISO 11898 – 1	Data Link Layer , entspricht den Bosch Spezifikationen CAN 2.0A und CAN 2.0B
ISO 11898 – 2, 5, 6	Physical Layer für High Speed CAN
ISO 11898 – 3	... und Low Speed CAN
ISO 11898 – 4	Erweiterung des Data Link Layer für zeitgesteuerte Kommunikation (Time Triggered CAN)

Durch die hohen Stückzahlen im Automobilbereich sind CAN-Controller viel preisgünstiger als die meisten ASICs für die in der Automatisierungstechnik verbreiteten Feldbusse wie Profibus. CAN wird daher auch in industriellen Anwendungen als Sensor-Aktor-Bus verwendet (*CAN in Automation*).

Zu den größten Schwächen der ursprünglichen CAN-Spezifikation von Bosch gehört, dass im Wesentlichen nur der *Data Link Layer* spezifiziert ist. Zur Bitrate wie zum *Physical Layer* gab es nur einige Hinweise, um unterschiedlichste Implementierungen zu ermöglichen. Zum *Application Layer* gab es bewusst gar keine Vorgaben. Wie nicht anders zu erwarten, führten diese Freiheitsgrade zu unterschiedlichen Lösungen. Dabei resultieren die Unterschiede beim *Physical Layer* vor allem aus dem notwendigen Kompromiss zwischen gewünschter Buslänge und möglicher Bitrate und sind relativ überschaubar, während die Lösungen beim *Application Layer* weitgehend inkompatibel zueinander sind. Der *Physical Layer* und der *Data Link Layer* einschließlich der für emissionsrelevante Komponenten wesentlichen Einschränkungen werden in den folgenden beiden Abschnitten beschrieben, der *Transport Layer* in Kap. 4 und der *Application Layer* in Kap. 5. Die gesamte Beschreibung soll einen Überblick geben, zu konkreten Details wird auf die Originaldokumente verwiesen (Tab. 3.1).

3.1.2 Bus-Topologie und Physical Layer

CAN ist ein bitstrom-orientierter Linien-Bus, der für eine maximale Bitrate von 1 Mbit/s definiert ist. CAN verwendet einen CSMA/CR-Buszugriff sowie eine Fehlererkennung, die die Reaktion aller Steuergeräte innerhalb einer Bitzeit erfordert. Dadurch muss die Buslänge umso kleiner sein, je höher die Bitrate ist. Die einzuhaltende Bedingung ist

$$\text{Buslänge} \leq 40 \dots 50 \text{ m} \cdot \frac{1 \text{ Mbit/s}}{\text{Bitrate}}. \quad (3.1)$$

Die Formel ist eine Faustformel, da bei hohen Bitraten die Verzögerungszeiten der Bustransceiver oder der bei sehr großen Buslängen in der Automatisierungstechnik gegebenenfalls eingesetzten Repeater die zulässige Buslänge bzw. Bitrate reduzieren. Alle Steuergeräte des Busses müssen mit derselben Bitrate arbeiten.

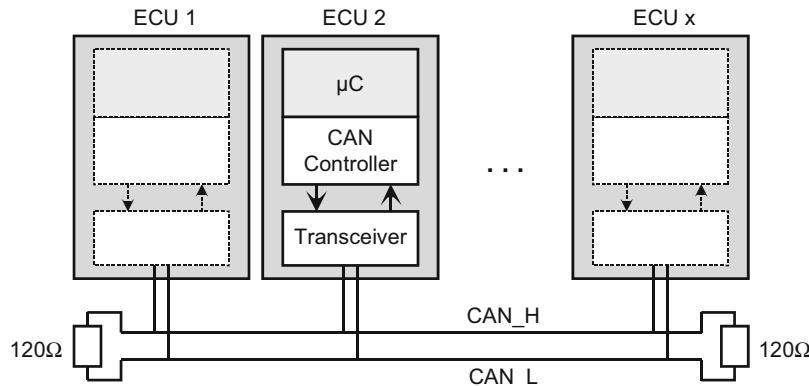


Abb. 3.1 High-Speed-CAN-Bus

Der Anhang ISO 11898-2 zum CAN-ISO-Standard definiert für Applikationen mit Bitraten ≥ 250 kbit/s (*High Speed CAN*, d. h. Class C) die Verwendung einer nach Möglichkeit verdrillten Zwei-Draht-Leitung als echter Linien-Bus mit Stichleitungen von max. 30 cm zu den einzelnen Steuergeräten. Der Bus muss an beiden Enden mit dem Wellenwiderstand der Zwei-Draht-Leitung, typischerweise ca. 120Ω , abgeschlossen werden (Abb. 3.1). Der Signalhub des Differenzspannungssignals liegt bei ca. 2 V (Abb. 3.2). Bei Unterbrechung oder Kurzschluss einer Ader der Zwei-Draht-Leitung fällt der Bus aus. In PKWs werden *High-Speed-CAN*-Busse im Antriebsstrang-Bereich mit Bitraten von typischerweise 500 kbit/s, bei Nutzfahrzeugen auch 250 kbit/s eingesetzt (Class C). Diese Bitraten sind auch in den SAE-Normen für CAN in PKW (SAE J2284) bzw. LKW (SAE J1939) standardisiert. Die neueren Zusatznormen ISO 11898-5 und ISO 11898-6 definieren verschiedene Maßnahmen zur Energieeinsparung.

Im Anhang ISO 11898-3 wird für Bitraten ≤ 125 kbit/s (*Low Speed CAN*, d. h. Class B), z. B. für Anwendungen in der Karosserieelektronik, ebenfalls eine Zwei-Draht-Leitungsverbindung spezifiziert. Aufgrund der niedrigeren Bitrate darf der Bus entsprechend länger sein. Die Busabschlusswiderstände und die Begrenzung auf kurze Stichleitungen entfallen. Der Signalhub des Differenzsignals ist deutlich größer als beim *High Speed CAN* (Abb. 3.2). Der Bus bleibt auch bei Kabelbruch oder Kurzschluss einer Ader noch funktionsfähig. *Low*

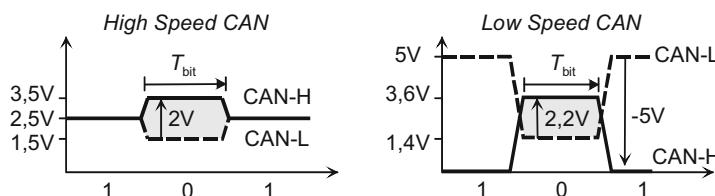


Abb. 3.2 Signalpegel bei High- und bei Low-Speed-CAN nach ISO 11898-2 bzw. 3

Speed CAN wird in europäischen Kfz in der Regel mit 100 bis 125 kbit/s für die Karosserielektronik eingesetzt.

In SAE J2411 ist auch eine Ein-Draht-Ausführung mit Bitraten von 33 kbit/s (Einsatz bei GM) und 83 kbit/s (Einsatz bei Chrysler) und einem Signalhub von 5 V für die Kfz-Komfortelektronik spezifiziert (*Single Wire CAN*).

Für die Verbindung von LKW-Zugmaschinen und Anhängern (*Truck and Trailer*) wird in ISO 11992 eine Punkt-zu-Punkt-Zwei-Draht-Ausführung mit 125 kbit/s und einem Signalpegel in der Größenordnung der Batteriespannung definiert.

Für Anwendungen in Nutzfahrzeugen spezifiziert SAE J1939/11 eine feste Bitrate von 250 kbit/s mit einer Busankopplung, die weitgehend der High Speed CAN Spezifikation von ISO 11898-2 entspricht, allerdings wird eine Abschirmung der Zwei-Draht-Leitung, eine maximale Buslänge von 40 m und eine Höchstzahl von max. 30 Steuergeräten gefordert. In SAE J1939/12 wird eine Variante ohne abgeschirmtes Kabel definiert. Der in SAE J1939/21 spezifizierte Data Link Layer entspricht dem nachfolgend beschriebenen CAN 2.0B.

Der CAN-Anschluss an Kfz-Steuergeräte erfolgt üblicherweise über den normalen Steuergerätestecker. Für die Automatisierungstechnik definiert CAN-in-Automation (CiA) in der Empfehlung CiA DS 102 für Bitraten zwischen 10 kbit/s ... 1 Mbit/s einen Physical Layer ähnlich ISO 11898-2. DeviceNet verwendet 125 ... 500 kbit/s. In der Automatisierungstechnik sind CAN-Anschlüsse über 9 polige Sub-Miniatur-D-Stecker und verschiedene andere Steckertypen im Einsatz.

Für alle CAN-Ausführungsformen gibt es auf dem Markt geeignete Transceiver-Bausteine. Untereinander sind die verschiedenen Varianten des Physical Layer nicht kompatibel, da die Signalpegel unterschiedlich sind (Abb. 3.2). In allen Fällen werden diejenigen Signalpegel hochohmig (*rezessiv*) erzeugt, die eine logische „1“ definieren. Signalpegel, die die logische „0“ übertragen, sind dagegen niederohmig (*dominant*).

Im Gegensatz zu einfachen UARTs kann und muss der Abtastzeitpunkt des Bussignals (*Bit Timing*) bei den meisten CAN-Controllern vom Anwender eingestellt werden (Abb. 3.3). Die Taktperiode, aus der der Bittakt T_{bit} abgeleitet wird, wird als *Quantum* T_Q bezeichnet. Im ersten *Quantum*, dem *Synchronization Segment* T_{SyncSeg} , gibt der Sender die Signallanke aus. Die Signallaufzeiten durch die beiden Transceiver im Sender und im Empfänger und auf der Busleitung werden durch das *Propagation Segment* T_{PropSeg} berücksichtigt. Wegen der *bitweisen Arbitrierung*, die im folgenden Abschnitt beschrieben wird, muss $T_{\text{SyncSeg}} + T_{\text{PropSeg}}$ mindestens doppelt so lang sein wie die maximale Signalverzögerung. Die Abtastung des Bits erfolgt dann ungefähr in der Mitte der verbleibenden Zeit $T_{\text{PhaseSeg1}} + T_{\text{PhaseSeg2}}$ bis zum Bitende. Die Dauer der beiden *Phase Segment* Abschnitte kann vom CAN-Controller dynamisch verlängert bzw. verkürzt werden, um den Bittakt des Empfängers automatisch auf den Bittakt des Senders zu synchronisieren. Die Anpassung erfolgt in Schritten von T_Q , maximal um die sogenannte *Synchronisation Jump Width* $T_{\text{SJW}} = 1 \dots \min(4T_Q, T_{\text{PhaseSeg1}})$. Da das *Bit Timing* insbesondere für Softwareentwickler etwas unübersichtlich ist, enthält z. B. die Norm ISO 15765-4 für die Diagnose abgasrelevanter Systeme über CAN verschiedene Empfehlungen. Bei einer Bitrate von

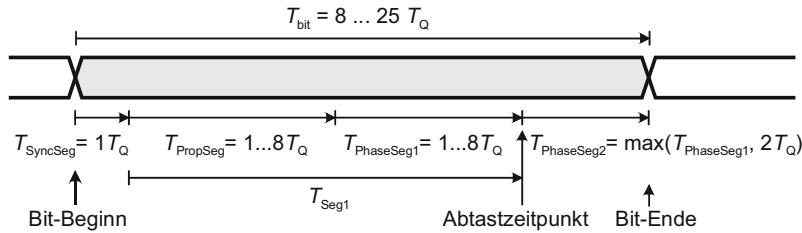


Abb. 3.3 Aufteilung eines CAN-Bits in Zeitabschnitte

500 kbit/s mit einer Toleranz von $\pm 0,15\%$, wie sie für Diagnosetester gefordert wird, und einem *Quantum* $T_Q = 125$ ns ergibt sich eine Bitdauer von $T_{bit} = 16 T_Q$. Dabei wird empfohlen, $T_{Seg1} = T_{PropSeg} + T_{PhaseSeg1} = 12 T_Q$ und $T_{Seg2} = T_{PhaseSeg2} = 3 T_Q$ mit $T_{SJW} = 3 T_Q$ einzustellen.

3.1.3 CAN Data Link Layer

Beim bitstrom-orientierten CAN-Protokoll erfolgt die gesamte Botschaftsübertragung selbstständig durch den Kommunikationsbaustein (CAN-Controller). Die Einzelheiten der Bitübertragung sind daher lediglich für den Entwickler eines CAN-Controllers von Bedeutung, während der CAN-Anwender lediglich eine grobe Vorstellung vom Botschaftsaufbau und Übertragungsablauf haben muss (Abb. 3.4).

CAN ist ein Broadcast-System, bei dem jeder Sender seine Botschaften ohne Ziel- und Quelladresse absendet, sondern jede Botschaft durch eine eindeutige Kennung, den *Message Identifier*, markiert. Ein Verbindungsaufbau ist nicht notwendig. Jedes Steuergerät am Bus empfängt die Botschaft und entscheidet anhand des *Message Identifiers*, ob es die Botschaft weiterverarbeitet oder ignoriert. Die Länge des *Message Identifiers* war ursprünglich 11 bit (CAN 2.0A), in der zweiten CAN-Generation wurden aber zusätzlich auch aufwärts kompatible 29 bit-Identifier (CAN 2.0B) definiert. Daneben werden im Identifier-Feld noch 1 bzw. 3 Steuerbits übertragen.

Beim Buszugriff wird das CSMA/CR-Verfahren verwendet. Jedes Steuergerät kann senden, sobald der Bus für mindestens 3 Bitzeiten frei ist. Der *Message Identifier* kenn-

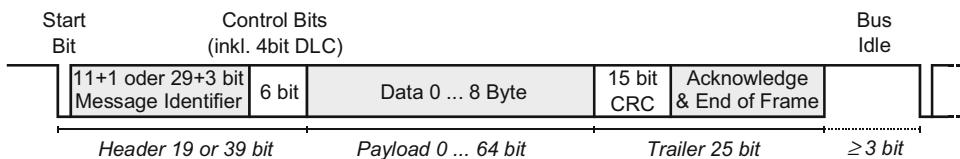
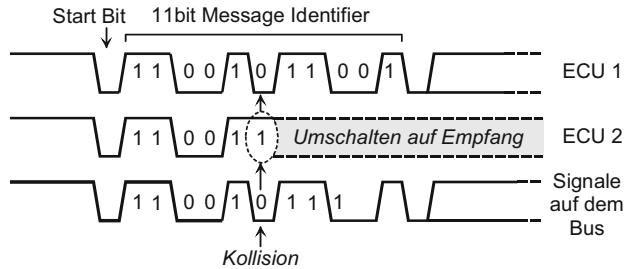


Abb. 3.4 Botschaftsformat (Längenangaben ohne Bit-Stuffing, siehe unten)

Abb. 3.5 Kollision von CAN-Botschaften



zeichnet neben dem Inhalt einer Botschaft auch deren Priorität (niedrigere Zahl bedeutet höhere Priorität). Kommt es dabei zu einer Kollision, so „gewinnt“ die Botschaft mit der höheren Priorität, d. h. der Sender, dessen *Message Identifier* den kleineren Wert hat. Im Beispiel nach Abb. 3.5 beginnen das Steuergerät ECU 1 mit dem *Message Identifier* $110\ 0101\ 1001_B = 659_H$ und das Steuergerät ECU 2 mit dem *Message Identifier* $110\ 0111\ 0000_B = 670_H$ gleichzeitig mit dem Senden. Beim sechsten Bit des *Message Identifiers* kommt es zu einer *Kollision*, weil ECU 2 eine 1 senden will, während ECU 1 eine 0 sendet. Da das 0-Signal im Vergleich zum 1-Signal wesentlich niederohmiger ist, wie bereits im vorigen Abschnitt erläutert wurde, dominiert auf den Busleitungen das 0-Signal. Weil die Steuergeräte beim Senden ständig überprüfen, ob der Signalpegel auf dem Bus dem jeweils gesendeten Bit tatsächlich entspricht, erkennt ECU 2 sofort, dass es zu einer Kollision gekommen ist, stellt das Senden ein und schaltet auf Empfang um. Die Kollisionserkennung und Auflösung wird als *bitweise Arbitrierung* bezeichnet. Die Übertragung der Botschaft des Steuergerätes ECU1 wird unverzögert fortgesetzt, während ECU 2 fruestens dann erneut mit dem Senden beginnt, wenn der Bus nach Ende dieser Botschaft wieder frei ist.

Eine Botschaft kann 0 bis 8 Nutzdatenbytes (*Payload*) übertragen (Abb. 3.4). Die Anzahl steht im *Data Length Code* (DLC) Feld innerhalb der Steuerbits (*Control Bits*). Zur Fehlererkennung wird eine 15 bit lange Prüfsumme (*Cyclic Redundancy Check*) mitgesendet.

Die Empfänger synchronisieren ihren Bittaktgenerator über das Startbit mit dem Sender und werden durch zusätzlich eingefügte, so genannte *Stuff-Bits* nachsynchronisiert. Die Anzahl der *Stuff-Bits* hängt vom den übertragenen Daten ab. Im theoretisch ungünstigsten Fall wird nach jedem 5. Bit ein *Stuff-Bit* so eingefügt, dass niemals mehr als fünf aufeinanderfolgende Bits denselben Wert aufweisen. Da die *Stuff-Bits* einerseits selbst wieder Teil des nächsten *Stuffing*-Blocks sind, andererseits aber das Botschaftsende nach der CRC-Prüfsumme nicht mehr dem *Stuffing* unterliegt, verlängert sich die Botschaft durch das *Stuffing* im ungünstigsten Fall um knapp 25 % [5, 6]. Auf der Empfangsseite werden die *Stuff-Bits* automatisch wieder entfernt.

Die CAN-Controller der am Bus angeschlossenen Steuergeräte überprüfen das Botschaftsformat und die Prüfsumme und senden innerhalb des *Acknowledge* und *End of Frame* Felds eine positive Empfangsbestätigung oder Fehlermeldung (*Error Frame*). Bei einer Fehlermeldung ignorieren alle Empfänger die empfangenen Daten. Dadurch ist sicher-

gestellt, dass die Daten im gesamten Netzwerk konsistent bleiben. Der Sender, der die Fehlermeldung erhält, startet automatisch einen neuen Sendeversuch.

Eine spezielle Botschaftsform ist der *Remote Frame*. Mit diesem Frame, der einen üblichen *Message Identifier*, aber keine Nutzdaten enthält und bei dem ein Bit am Ende des *Identifier* Felds gesetzt wird, fordert ein Steuergerät eine Botschaft mit den zu diesem *Message Identifier* gehörenden Daten von einem anderen Steuergerät an.

3.1.4 Fehlerbehandlung

Die verschiedenen Fehlererkennungsverfahren sorgen für eine hohe Übertragungszuverlässigkeit. Nach verschiedenen Untersuchungen liegt die Restfehlerwahrscheinlichkeit deutlich unter 10^{-11} . Da Fehler unmittelbar nach Erkennung, spätestens am Ende einer Botschaft signalisiert werden, erfolgt sehr schnell eine automatische Übertragungswiederholung.

Jeder CAN-Controller enthält Fehlerzähler, mit deren Hilfe er eigene Sende- und Empfangsfehler protokolliert und Fehlermeldungen sendet (*Error active*). Dabei unterscheidet er auch, ob er die Fehler selbst erkannt hat oder ob sie auch von anderen CAN-Controllern festgestellt wurden. Falls der CAN-Controller dadurch erkennt, dass er selbst fehlerhaft arbeitet, stellt er zunächst eigene Sendeversuche von Fehlermeldungen ein (*Error passive*) und schaltet sich bei anhaltenden Problemen vollständig ab (*Bus off*). Verschwindet das Problem, z. B. weil es durch eine vorübergehende EMV-Störung bewirkt wurde, so aktiviert sich der CAN-Controller wieder.

3.1.5 Einsatz von CAN – Höhere Protokolle

Mit der relativ kurzen Botschaftslänge, dem prioritätsgesteuerten Buszugriff und der hohen Fehlersicherheit wurde CAN bewusst für den Austausch von Mess-, Regel- und Steuerdaten im Echtzeitbetrieb konzipiert und war ursprünglich hauptsächlich als Ersatz für Punkt-zu-Punkt-Verbindungen mit analogen und digitalen Signalen im Kfz vorgesehen. Da für solche Anwendungen höchste Flexibilität und geringstmöglicher Overhead gefordert sind, machten die ursprünglichen CAN-Spezifikationen keinerlei Vorgaben für die Vergabe der *Message Identifier* oder die Bedeutung, Formatierung und Normierung der übertragenen Daten.

Im PKW-Bereich legte und legt praktisch jeder europäische PKW-Hersteller diese Punkte individuell fest. Als Hilfsmittel dafür diente zunächst die so genannte *CAN- oder Kommunikations-Matrix*, in der in einer Tabellenstruktur dargestellt wird,

- welche Steuergeräte welche Botschaften unter welchen Bedingungen und mit welcher Zykluszeit senden,
- welche Steuergeräte diese Botschaften weiterverarbeiten,

- welche Daten (*Signale*) in diesen Botschaften enthalten sind und wie diese Daten normiert sind, d. h. welche Umrechnungsbeziehung zwischen den hexadezimalen Werten und den realen physikalischen oder logischen Größen bestehen,
- mit welcher Priorität, d. h. mit welchen Message Identifiern, die Botschaften gesendet werden.

Die so festgelegten Parameter müssen zumindest innerhalb eines Fahrzeugmodells für alle Steuergeräte einheitlich sein und von den verschiedenen Steuergeräte-Zulieferern gleich implementiert werden. Dafür stellen die Hersteller von CAN-Entwicklungs- und Testumgebungen heute datenbankgestützte Werkzeuge für die Erstellung und Pflege der CAN-*Matrix* (CAN-Datenbank CANdb, FIBEX-Beschreibungen) bereit.

Heute müssen bereits bei einem einzelnen Kfz-Steuergerät während der Entwicklungs- und Applikationsphase mehrere Tausend Datenwerte modellspezifisch festgelegt werden. Daher werden zunehmend automatisierte Applikationswerkzeuge eingesetzt. Die Kfz-Industrie bemüht sich im ASAM-Konsortium, die Daten-Darstellung (Formatierung, Normierung) zumindest auf der Ebene der Datenaustauschformate dieser Applikationswerkzeuge zu standardisieren.

Auf der NKW-Seite wurde die Problematik wesentlich früher erkannt. Basierend auf den Erfahrungen mit dem in USA schon Anfang der 90er Jahre verbreiteten zeichenbasierten Class A-Protokoll J1587/J1708 wurden die oben genannten Punkte für CAN Mitte der 90er Jahre in der Normenfamilie SAE J1939 einheitlich spezifiziert (siehe Abschn. 4.5)

Aufgrund der vorhandenen Infrastruktur in Werkstätten und Fertigungen, die bevorzugt auf der K-Line-Schnittstelle basierte, und des bis Ende der 90er Jahre noch nicht durchgängigen Einsatzes von CAN in allen Kfz-Steuergeräten, galt CAN für die Anwendungsbereiche Diagnoseschnittstelle und End-of-Line-/Flash-Programmierung zunächst als uninteressant. Die CAN-Spezifikationen trafen daher auch keine Vorkehrungen für den Transport von zusammengehörenden Datenblöcken von mehr als 8 Byte. Mittlerweile wird CAN auch auf diesem Gebiet verwendet. Die notwendigen Voraussetzungen dazu schafft die Normenfamilie ISO 15765, die beschreibt, wie das KWP 2000 bzw. UDS-Protokoll über CAN realisiert werden kann und in ISO 15765-2 auch ein Transportprotokoll (TP) definiert, das die Aufteilung von Datenblöcken von bis zu 4 K-Byte in einzelne CAN-Botschaften ermöglicht (siehe Abschn. 4.1).

3.1.6 Schnittstelle zwischen Protokoll-Software und CAN-Controller

Die gesamte Botschaftsübertragung einschließlich Fehlerüberprüfung erfolgt selbstständig durch den Kommunikationsbaustein (CAN-Controller), der heute in der Regel als On-Chip-Modul der eingesetzten Mikrocontroller implementiert ist, während die physikalische Ankopplung an die Busleitungen durch einen separaten CAN-Transceiver-Baustein erfolgt (Abb. 3.1). Praktisch alle neueren CAN-Controller können so konfiguriert werden,

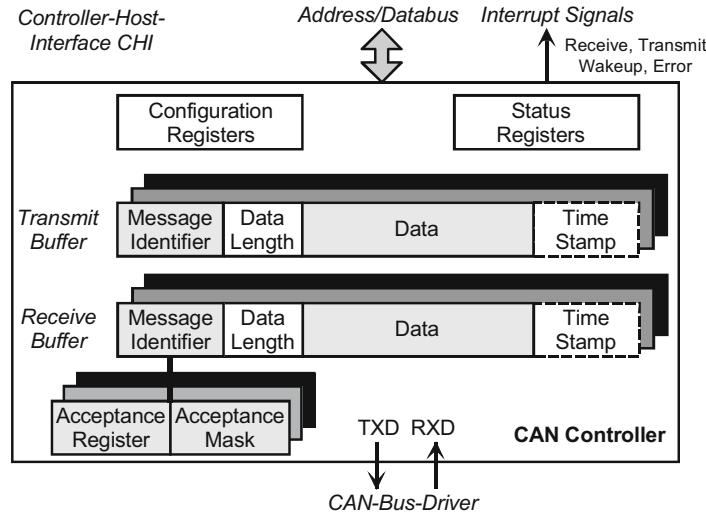


Abb. 3.6 Typischer CAN-Kommunikationscontroller

dass sie sowohl mit 11 bit als auch mit 29 bit Message Identifiern arbeiten können, d. h. CAN 2.0A- und CAN 2.0B-fähig sind.

Aus Softwaresicht stellt der CAN-Controller eine Reihe von Steuerregistern sowie Speicherbereiche für mehrere Botschaften bereit (Abb. 3.6). Für die Übertragung einer Botschaft schreibt die Software den Message Identifier, die Anzahl der Datenbytes und die Datenbytes selbst in den Botschaftsspeicher, das Absenden erfolgt dann selbstständig durch den Controller. Beim Empfang liest die Software die Werte aus dem Botschaftsspeicher aus. Mit Hilfe von Statusbits und Interrupts synchronisieren CAN-Controller und Mikroprozessor den Zugriff auf den Botschaftsspeicher (Handshake) und tauschen Fehlermeldungen aus.

Ursprünglich boten die Halbleiterhersteller zwei verschiedene CAN-Controllertypen an, die auf der Busseite zueinander kompatibel sind und daher im selben CAN-Netz problemlos miteinander kommunizieren können, sich aber im Bereich der Akzeptanzfilterung (siehe unten) und damit im Schaltungs- und Softwareaufwand unterschieden:

- Reine *Basic CAN*-Controller verfügen typischerweise über je einen Speicherbereich für eine zu sendende und eine zu empfangende Botschaft. Die Akzeptanzfilterung, d. h. das Auswerten des Message Identifiers einer empfangenen Botschaft und die Entscheidung, ob die Botschaft überhaupt vom entsprechenden Steuergerät verarbeitet werden muss, erfolgt durch die CAN-Software des Steuergerätes. Um die Softwarebelastung beim Empfang etwas zu reduzieren, verfügen Basic-CAN-Controller aber praktisch immer über eine Möglichkeit, Botschaften mit bestimmten Message Identifiern zu ignorieren. In der Regel wird dies durch eine Bitmaske für den Message Identifier realisiert.

Die Speicherbereiche für die Sende- und die Empfangsbotschaft sind in der Regel doppelt gepuffert (Double Buffer), so dass bereits eine neue Botschaft in den jeweiligen Botschaftsspeicher geschrieben bzw. empfangen werden kann, bevor die vorige Botschaft vollständig gesendet bzw. durch die Software weiterverarbeitet ist. Abhängig von der Implementierungsstrategie des Halbleiterherstellers wird eine vorige Botschaft überschrieben oder die neueste Botschaft ignoriert, wenn der Botschaftsspeicher nicht frei ist. Seltener findet man FIFO-(First In First Out)-Strukturen für den Sende- und Empfangsspeicher, weil das Prioritätskonzept von CAN es eigentlich notwendig macht, dass eine später empfangene oder bereitgestellte Botschaft früher verarbeitet oder versendet wird als eine frühere Botschaft, wenn die spätere Botschaft eine höhere Priorität hat. Dazu wäre ein dynamisches Umorganisieren eines FIFO-artig betriebenen Botschaftsspeichers notwendig. Die Priorisierung der Botschaften ist hinsichtlich der Echtzeiteigenschaften nur dann wirklich konsistent, wenn sie nicht nur beim Buszugriff, sondern auch bei der Botschaftsverarbeitung in der CAN-Software konsequent berücksichtigt wird.

- *Full CAN*-Controller haben typischerweise 8 bis 16 Botschaftsspeicher, wobei jeder Botschaftsspeicher entweder für das Senden oder Empfangen einer bestimmten Botschaft konfiguriert werden kann. Die Akzeptanzfilterung findet automatisch, d. h. in der CAN-Controller-Hardware statt, wobei wiederum für jeden Botschaftsspeicher konfiguriert werden kann, welche Message Identifier akzeptiert werden. Neu ankommende Botschaften überschreiben dabei in der Regel die vorige Botschaft im entsprechenden Botschaftsspeicher. Durch die hardwaremäßige Akzeptanzfilterung ist die Softwarebelastung beim Empfang deutlich geringer als bei *Basic CAN*. Durch die verschiedenen Botschaftsspeicher ist mehr Zeit für die Weiterverarbeitung empfangener Botschaften vorhanden und die Bereitstellung der zu sendenden Botschaften kann gruppenweise und teilweise unabhängig von der Prioritätenreihenfolge erfolgen, da der CAN-Controller unter den zu sendenden Botschaften stets die höchspriore als erste auswählt.

Heutige CAN-Controller (Abb. 3.6) sind meist Mischformen. In der Regel verfügen sie über Speicher für eine Hand voll verschiedene Botschaften mit Hardware-Akzeptanzfilterung (*Full CAN*) sowie Speicher für mindestens eine Sende- und eine Empfangsbotschaft mit Software-Akzeptanzfilterung (*Basic CAN*). Bevor Botschaften versendet und empfangen werden, wird der Kommunikationscontroller durch den steuernden Mikrocontroller (*Host*) initialisiert. Dabei werden Bitrate und Bittiming nach Abb. 3.3 sowie die Interrupts eingestellt, die Akzeptanzfilter gesetzt und der Betriebsmodus ausgewählt. Neben dem Normalbetrieb können CAN-Controller häufig auch so betrieben werden, dass sie Botschaften nur empfangen (*Listen* oder *Silent Mode*), aber nicht am normalen Busbetrieb teilnehmen, d. h. auch keine Empfangsbestätigung (*Acknowledge*) und keine Fehlermeldungen senden. Dieser Modus wird zur Analyse des Datenverkehrs eingesetzt, ohne den Bus zu beeinflussen. Zu Testzwecken in der Entwicklungsphase wird der *Loopback Mode* eingesetzt, bei dem der CAN Controller die Botschaften, die er sendet, nur selbst empfängt, den Bustreiber dagegen komplett abschaltet. Außerdem kann der CAN Controller

Tab. 3.2 Beispiel für die Einstellung eines Akzeptanzfilters für 11 bit Identifier

Bit	10	9	8	7	6	5	4	3	2	1	0
Akzeptanzregister AR	0	1	1	0	1	1	1	0	0	0	0
Akzeptanzmaske AM	1	1	1	1	1	1	1	0	0	0	0
Effektives Akzeptanzfilter AF	0	1	1	0	1	1	1	X	X	X	X

$$37Xh = 370 \dots 37Fh$$

in einen stromsparenden Modus (*Sleep Mode*) geschaltet werden, aus dem er durch den Mikrocontroller oder durch den Empfang einer beliebigen CAN-Botschaft wieder aufgeweckt (*Wakeup*) wird.

Nach dem Verlassen des Konfigurationsmodus kann der Mikrocontroller Botschaften bestehend aus dem Message Identifier, den Datenbytes und deren Anzahl in einen der Sendespeicher schreiben und durch Setzen des zugehörigen Statusbits zum Senden freigeben. Stehen in verschiedenen Sendespeicher mehrere Botschaften zum Senden an, versendet der Kommunikationscontroller je nach Ausführung die Botschaften in der Reihenfolge ihrer Priorität oder der Sendeanforderung. Ob die Botschaft erfolgreich versendet wurde, kann der Mikrocontroller durch Lesen eines Statusbits abfragen (*Polling*) oder sich automatisch (*Transmit Interrupt*) informieren lassen. Optional kann er zusätzlich den Zeitpunkt, zu dem einer der Busteilnehmer den Empfang durch Setzen des *Acknowledge Bit* im Botschafts-Trailer bestätigt hat, aus dem *Time Stamp* Feld des Sendespeichers auslesen. Analog wird der Empfangszeitpunkt einer Botschaft im *Time Stamp* Feld des Empfangsspeichers festgehalten. Welche Botschaften empfangen werden, wird durch die Akzeptanzfilter vorgegeben (Tab. 3.2). Jedes Akzeptanzfilter besteht in der Regel aus einem Register AR, in das der zu empfangene Message Identifier eingetragen wird, und einer Maske AM, die angibt, welche Bits des Message Identifiers überhaupt geprüft werden sollen. Auf diese Weise können Don't Care Bits festgelegt und damit ganze Identifier-Bereiche ausgewählt werden. In dem Beispiel nach Tab. 3.2 wird das Akzeptanzfilter auf 37Xh gesetzt und damit ein Bereich von 370h … 37Fh aktiviert. Der Mikrocontroller erfährt vom Empfang einer Botschaft entweder durch Abfragen des Statusregisters oder durch den *Receive Interrupt*. In ähnlicher Form kann sich der Mikrocontroller informieren oder informieren lassen, wenn eine Botschaft verloren geht, weil alle Empfangsspeicher voll waren, oder wenn es Fehler auf dem CAN-Bus gab.

3.1.7 Zeitverhalten von CAN-Systemen, Wahl der Botschaftspriorität

Die Länge einer CAN Botschaft einschließlich der 3 bit langen *Bus Idle* Phase zwischen zwei aufeinanderfolgenden Botschaften ist

$$T_{Frame} = n_{Frame} \cdot T_{bit} = (n_{Header} + n_{Trailer} + n_{Idle} + n_{Data} + n_{Stuff}) \cdot T_{bit}. \quad (3.2)$$

Tab. 3.3 Botschaftslänge und Datenrate für $f_{\text{bit}} = 1 / T_{\text{bit}} = 500 \text{ kbit/s}$

CAN ID	n_{Data}	Länge ohne Stuffing		Länge mit Stuffing		$f_{\text{Data}} = n_{\text{Data}} / T_{\text{Frame}}$
		$n_{\text{Frame, min}}$	$T_{\text{Frame, min}}$	$n_{\text{Frame, max}}$	$T_{\text{Frame, max}}$	
11 bit	1 byte	55 bit	110 μs	65 bit	130 μs	7,5 KB/s
	8 byte	111 bit	222 μs	135 bit	270 μs	28,9 KB/s
29 bit	1 byte	75 bit	150 μs	90 bit	180 μs	5,4 KB/s
	8 byte	131 bit	262 μs	160 bit	320 μs	24,4 KB/s

Dabei ist $n_{\text{Header}} + n_{\text{Trailer}} + n_{\text{Idle}} = 47 \text{ bit}$ (67 bit), wenn mit 11 bit *Message Identifier* gearbeitet wird. Die Werte für 29 bit Identifier sind in (...) angegeben. Die Länge des Nutzdatenbereichs $n_{\text{Data}} = 0, \dots, 64 \text{ bit}$ ist in Stufen von 8 bit variabel.

Die Anzahl der Stuff-Bits hängt vom Botschaftsinhalt ab und beträgt nach [5, 6]

$$n_{\text{Stuff}} = 0 \dots \left\lfloor \frac{n_{\text{Header}} + n_{\text{Trailer}} + n_{\text{Idle}} + n_{\text{Data}} - 14 \text{ bit}}{4} \right\rfloor, \quad (3.3)$$

wobei [...] bedeutet, dass der Bruch auf den nächsten ganzzahligen Wert abgerundet werden muss. Als Faustformel ergibt sich

$$T_{\text{Frame}} < 1,25 \cdot (47 \text{ bit}(67 \text{ bit}) + n_{\text{Data}}) \cdot T_{\text{bit}}. \quad (3.4)$$

Tabelle 3.3 gibt einen Überblick über die minimale und maximale Länge und die zugehörige Datenrate $f_{\text{Data}} = n_{\text{Data}} / T_{\text{Frame}}$.

CAN arbeitet ereignisgesteuert. Im Idealfall wird die Botschaft sofort gesendet, sobald sie in den Botschaftspuffer des Kommunikationscontrollers kopiert und das entsprechende Sendezeit gesetzt wurde. Vernachlässigt man die Signallaufzeiten der Transceiver und der Leitung gegenüber der Botschaftsdauer, so ist die Zeit zwischen Bereitstellen der Botschaft auf der Sendeseite und deren Verfügbarkeit auf der Empfangsseite, d. h. die Buslatenz, im günstigsten Fall (vergleiche Abb. 2.17):

$$T_{\text{Latenz, min}} = T_{\text{Frame}}. \quad (3.5)$$

Wenn der Bus jedoch gerade belegt ist oder wenn eine andere Botschaft mit höherer Priorität zum Senden ansteht, muss die Botschaft warten, bis der Bus frei wird. Da das CAN Protokoll keine inhärente Obergrenze für diese Wartezeit garantiert, gilt der CAN-Bus als nicht-deterministisch (nicht *hart echtzeitfähig*) und daher im strengen Sinn nicht geeignet für Anwendungen wie Fahrwerksregelungen mit hohen Anforderungen an die zeitliche Präzision. Theoretische Arbeiten von Tindell, Burns und anderen [4–7] und langjährige praktische Erfahrungen haben gezeigt, dass sich unter bestimmten Voraussetzungen auch für CAN maximale Latenzeiten garantieren lassen. Allerdings ergeben sich dabei relativ

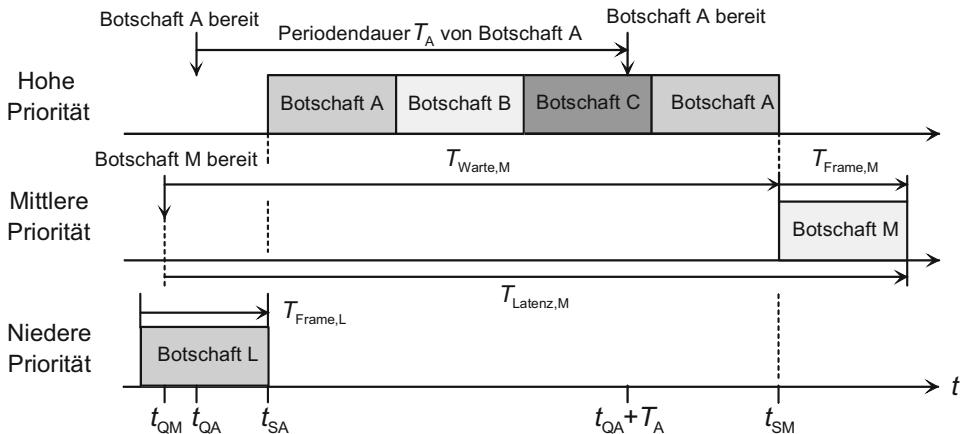


Abb. 3.7 Szenario zur Bestimmung der Worst Case Latenzzeit

große Werte bzw. nur kleine zulässige Busauslastungen, so dass sich CAN für schnelle Echtzeitregelungen nicht so sehr wegen seiner theoretischen Defizite sondern vor allem wegen seiner relativ niedrigen Bitraten nur eingeschränkt eignet.

3.1.7.1 Berechnung der maximalen Latenzzeiten

Abbildung 3.7 zeigt das Szenario, für das die maximalen Latenzzeiten bestimmt werden sollen. Eine Botschaft M mit mittlerer Priorität wird zum Zeitpunkt $t = t_{QM}$ in den Sendepuffer (Sendewarteschlange, engl. *Queue*) gestellt. Zu diesem Zeitpunkt sei der Bus aber gerade durch eine andere Botschaft L belegt. Da eine laufende Übertragung bei CAN nicht abgebrochen wird, wird diese Botschaft unabhängig von ihrer Priorität in jedem Fall vollständig übertragen. Botschaft M muss also mindestens bis zum Zeitpunkt $t = t_{SA}$ warten. In der Zwischenzeit seien nun aber auch weitere Botschaften A , B und C zum Senden bereitgestellt worden, die eine höhere Priorität als Botschaft M aufweisen sollen. Auch auf diese Botschaften muss Botschaft M warten. Im Beispiel ist die Wartezeit so groß, dass die Botschaft A , die mit kurzer Periodendauer T_A zyklisch gesendet wird, während der Wartezeit von Botschaft M sogar zweimal übertragen wird. Erst zum Zeitpunkt $t = t_{SM}$ kann Botschaft M dann endlich gesendet werden.

Die Voraussetzungen für eine Berechnung der Worst Case Latenzzeit sind [4, 5]:

- Die *Message Identifier* und damit die Prioritäten aller Botschaften des CAN Bussystems sowie deren Länge $T_{Frame,k}$ müssen bekannt sein. $LP(M)$ sei die Menge aller Botschaften, die eine niedrigere Priorität haben als die betrachtete Botschaft M . $HP(M)$ sei die Menge aller Botschaften, die eine höhere Priorität haben.
- Für alle Botschaften, die zyklisch gesendet werden, muss deren jeweilige Sendeperiode T_k bekannt sein. Für Botschaften, die nicht zyklisch gesendet werden, wird T_k als der

Mindestabstand zwischen zwei Sendeversuchen interpretiert (*Interarrival Time*) und muss ebenfalls bekannt sein.

- Alle Übertragungen erfolgen streng gemäß der Priorität der Botschaften. Dies muss auch für die Verarbeitung der Botschaften in den Sende- und Empfangspuffern der CAN-Kommunikationscontroller gelten. Bei FIFO-ähnlich organisierten Puffern in einfacheren Kommunikationscontrollern galt dies in der Vergangenheit leider nicht immer.
- Keine Übertragungsfehler und daher auch keine automatische Wiederholung.

Nicht berücksichtigt wird in den folgenden Betrachtungen, dass die Botschaftsdaten durch die Anwendungssoftware des Steuergerätes bereitgestellt werden und der Bereitstellungszeitpunkt damit ebenfalls mehr oder weniger schwankt. Auf das Zeitverhalten typischer Steuergeräte-Betriebssysteme wird in Kap. 7 eingegangen.

Unter den genannten Voraussetzungen ergibt sich die Wartezeit der Botschaft M zu:

$$T_{\text{Warte},M} = \max_{k \in \text{LP}(M)} (T_{\text{Frame},k}) + \sum_{k \in \text{HP}(M)} \left\lceil \frac{T_{\text{Warte},M}}{T_k} \right\rceil \cdot T_{\text{Frame},k}. \quad (3.6)$$

Dabei beschreibt der erste Term die Wartezeit auf die längste Botschaft mit niedriger Priorität. Der Summenterm gibt die Wartezeit auf die Botschaften mit einer höheren Priorität an. Der Faktor $T_{\text{Warte},M} / T_k$, der auf die nächste ganze Zahl aufgerundet werden muss, berücksichtigt, dass die Wartezeit der Botschaft M länger werden kann als die Periodendauer einer höheren Prioritäts Botschaft, so dass diese während der Wartezeit mehrfach gesendet wird. Da $T_{\text{Warte},M}$ auf beiden Seiten der Gleichung auftaucht, kann das Ergebnis leider nicht direkt, sondern nur durch Iteration bestimmt werden. Als Anfangswert der Iteration setzt man typischerweise den Summenterm zu 0. Die Iteration konvergiert, falls die mittlere Buslast

$$\text{BL} = \sum_{\text{alle } k} \frac{T_{\text{Frame},k}}{T_k} < 100 \% \quad (3.7)$$

ist. Die maximale Latenz der Botschaft M ergibt sich dann zu

$$T_{\text{Latenz},M,\text{max}} = T_{\text{Warte},M} + T_{\text{Frame},M} < T_D < T_M. \quad (3.8)$$

Bei einer Echtzeitanwendung wird in der Regel eine Obergrenze T_D (*Deadline*) für die Latenzzeit jeder Botschaft vorgegeben. Sinnvollerweise ist die Obergrenze kleiner als die Periodendauer T_M der jeweiligen Botschaft. Wird diese Bedingung nicht eingehalten, geht die Botschaft eventuell verloren. Bei den meisten Kommunikationscontrollern wird nämlich der Sendebuffer überschrieben, wenn eine weitere Botschaft mit demselben *Message Identifier* in den Puffer geschrieben wird, bevor die vorige Botschaft gesendet wurde. Die Schwankungsbreite der Latenzzeit ist der Jitter

$$T_{\text{Jitter},M} = T_{\text{Latenz},M,\text{max}} - T_{\text{Latenz},M,\text{min}}. \quad (3.9)$$

Tab. 3.4 Satz von CAN Botschaften ($f_{\text{Bit}} = 125 \text{ kbit/s}$, Worst Case Stuffing)

CAN ID	n_{Data}	Period T	$T_{\text{Latency,min}} = T_{\text{Frame}}$	$T_{\text{Latency,max}} = T_{\text{Warte,max}} + T_{\text{Frame}}$
1 (highest priority)	1 byte	50 ms	0,5 ms	1,4 ms
2	2 byte	5 ms	0,6 ms	2,0 ms
3	1 byte	5 ms	0,5 ms	2,6 ms
4	2 byte	5 ms	0,6 ms	3,2 ms
5	1 byte	5 ms	0,5 ms	3,7 ms
6	2 byte	5 ms	0,6 ms	4,3 ms
7	6 byte	10 ms	0,9 ms	5,0 ms
8	1 byte	10 ms	0,5 ms	8,6 ms
9	2 byte	10 ms	0,6 ms	9,2 ms
10 (lowest priority)	3 byte	10 ms	0,7 ms	9,9 ms

Tabelle 3.4 zeigt beispielhaft einen Satz von CAN Botschaften und die nach den obigen Formeln ermittelten Latenzzeiten. Alle Botschaften können innerhalb der geforderten Periodendauer gesendet werden. Allerdings haben die Botschaften mit den CAN IDs 6, 9 und 10 wegen der relativ hohen Busauslastung BL von etwa 85 % nur einen sehr geringen Sicherheitsabstand. Die Busauslastung und damit die maximalen Latenzzeiten könnten verringert werden, wenn mehrere Botschaften mit wenigen Nutzdatenbytes zu einer größeren Botschaft zusammengefasst würden. Naturgemäß ist dies nur bei Botschaften desselben Senders möglich.

3.1.7.2 Festlegung der Botschaftsprioritäten

Das Beispiel zeigt erwartungsgemäß, dass die maximale Latenzzeit sowie der Jitter massiv von der Priorität der Botschaft abhängen. Als Empfehlung für die Botschaftsprioritäten, d. h. die *Message Identifier*, gilt daher, dass die Priorität umso größer, d. h. der *Message Identifier* umso kleiner gewählt werden sollte, je kleiner die Periodendauer bzw. je kürzer die *Deadline* der Botschaft ist (*Rate Monotonic Priority* bzw. *Deadline Monotonic Priority*) [4]. Diese Faustregel ist einfach anzuwenden, in Grenzfällen aber möglicherweise nicht optimal. Sollten einzelne Botschaften die geforderten Latenzzeiten nicht erreichen, kann mit dem in [5] beschriebenen iterativen Verfahren nach Audsley überprüft werden, ob tatsächlich keine Prioritätsverteilung existiert, mit der die Anforderungen eingehalten werden können.

Durch Übertragungsfehler vergrößern sich die Latenzzeiten. Wenn ein Fehler festgestellt wird, sendet der Kommunikationscontroller einen Error Frame mit 31 bit und der Sender wiederholt die fehlerhafte Botschaft. Dies kann modelliert werden, indem man auf der rechten Seite von Gl. 3.6 bei der Iteration den folgenden Term addiert:

$$E(T_{\text{Warte},M} + T_{\text{Frame},M}) = [31T_{\text{bit}} + \max_{k \in \text{HP}(M) \cup M} (T_{\text{Frame},k})] \cdot \left[\frac{T_{\text{Warte},M} + T_{\text{Frame},M}}{T_{\text{Error}}} \right]. \quad (3.10)$$

Dabei wird angenommen, dass die Fehler mit der Fehlerrate $1 / T_{\text{Error}}$ auftreten. Komplexere Fehlermodelle wie bündelartige auftretende Fehler (*Burst Errors*) werden z. B. in [7] beschrieben.

3.1.7.3 Festlegung von Botschaftsperiode und Phasenoffset

Die maximale Latenzzeit tritt dann auf, wenn alle Botschaften gleichzeitig sendebereit werden. Diese Situation kann vermieden werden, wenn für die Botschaftsperioden ganzzahlige Vielfache gewählt und feste Phasenbeziehung zwischen den einzelnen Botschaften festgelegt werden. Da der *Data Link Layer* bei CAN keine netzweite Synchronisation der Kommunikationscontroller bereitstellt, kann dies allerdings nur lokal für den Botschaftssatz eines einzelnen Busknotens erfolgen oder muss mit Hilfe von Synchronisationsbotschaften zwischen den Netzketten durch die Anwendungssoftware sichergestellt werden [8]. Abbildung 3.8a zeigt das Problem, wenn zwei Botschaften nicht synchronisiert werden. Botschaft A werde alle $T_A = 5$ ms gesendet, Botschaft B alle $T_B = 8$ ms. Die Sequenz beginnt mit Botschaft A. Zufällig 3 ms später wird Botschaft B zum ersten Mal gesendet. Bei $t = 11$ ms sollte Botschaft B zum zweiten Mal gesendet werden. Da zu diesem Zeitpunkt aber Botschaft A wieder auf dem Bus liegt, verzögert sich Botschaft B. Dadurch verlängert sich diese Periode effektiv auf 9 ms, während sich die folgende Periode auf 7 ms verkürzt, weil Botschaft B zum Zeitpunkt $t = 19$ ms wieder planmäßig gesendet werden kann. Bei $t = 20$ ms liegt Botschaft B immer noch auf dem Bus, so dass die eigentlich anstehende Botschaft A erst bei $t = 21$ ms gesendet werden kann. Deren Periode verlängert sich dadurch auf 6 ms und wird beim nächsten planmäßigen Senden von Botschaft A auf 4 ms verkürzt. Im Beispiel *jittern* beide Periodendauern somit um ± 1 ms. Wesentlich günstiger ist es, wenn man die Periodendauern als ganzzahlige Vielfache, z. B. $T_A = 5$ ms und $T_B = 2 T_A = 10$ ms wählt und die Botschaftssequenz mit einer Phasenverschiebung $T_A / 2 = 2,5$ ms startet. Da die Botschaften nun zu keinem Zeitpunkt kollidieren, sind die Periodendauern ohne Jitter konstant. Das Beispiel zeigt den positiven Einfluss von Phasenoffsets. Berücksichtigt man diese bei der Berechnung [9], so gibt es allerdings nicht nur ein einziges Worst-Case-Szenario, sondern es müssen sämtliche Kombinationen von Phasenlagen analysiert werden. Dies ist manuell sehr aufwendig, so dass in der Praxis Scheduling-Analysewerkzeuge eingesetzt werden, wie sie in Abschn. 9.8 beschrieben werden.

3.1.8 Time-Triggered-CAN (TTCAN) – Deterministischer Buszugriff

Aufgrund des CSMA/CR-Verfahrens kann nur für die Botschaften mit der höchsten Priorität eindeutig, für andere Botschaften aber nur unter bestimmten weiteren Voraussetzungen die maximale Latenzzeit garantiert werden, innerhalb der die Botschaft auch im ungünstigsten Fall noch sicher übertragen werden kann (siehe voriger Abschnitt). In jedem Fall aber kann es zu deutlich schwankenden Latenzzeiten, d. h. einem großen Jitter kommen, wenn die Steuergeräte zeitlich völlig unsynchronisiert auf den Bus zugreifen und es zu einer kurzzeitigen Belastungsspitze auf dem Bus kommt, weil alle Steuergeräte gleichzeitig

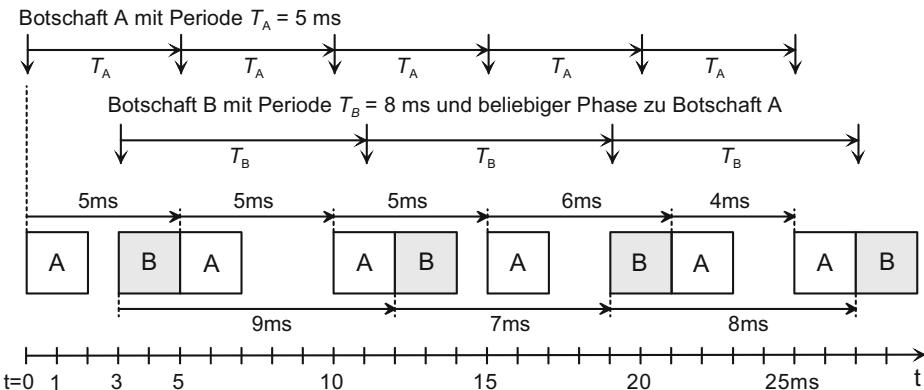
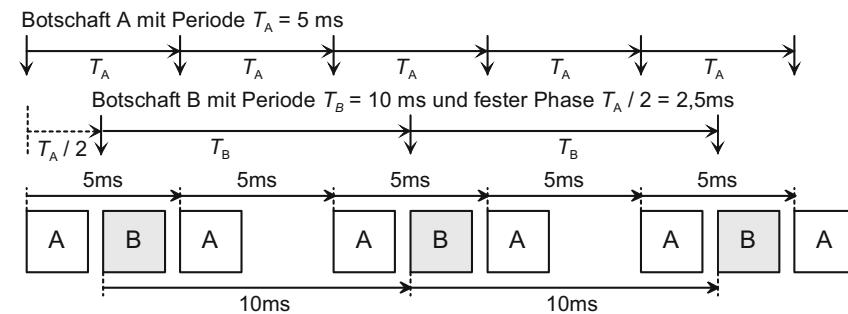
a**b**

Abb. 3.8 Einfluss der Botschaftsperioden und Phasen auf die Latenzzeit **a** beliebige Perioden und Phasen **b** Perioden als ganzzahlige Vielfache und feste Phasenbeziehungen

senden wollen. Mit Hilfe von Berechnungen und Simulationen lassen sich die maximalen Latenzzeiten und der Jitter in der Regel bestimmen. Allerdings sind bei komplexen Netzen mit sehr vielen Steuergeräten und Botschaften in der Praxis nicht alle Randbedingungen für alle Botschaften vollständig bekannt und oft stark von Implementierungsdetails in den Steuergeräten sowie von der aktuellen Konfiguration abhängig. Diese können sich durch Ausstattungsvarianten eines Fahrzeugs während eines Entwicklungsprojekts oder bei einem *Facelift* in der Serienproduktion in schwer vorhersehbarer Form ändern. CAN gilt daher als nicht im strengen Sinn deterministisch arbeitendes Bussystem, auch wenn es in der Praxis in der Regel bei weitem „gut genug“ ist. CAN in seiner *klassischen* Art wird daher für Anwendungen mit extremen Sicherheitsforderungen wie Steer-by-Wire und Brake-by-Wire als nur bedingt geeignet betrachtet.

Um ein streng deterministisches Übertragungsverhalten zu garantieren, muss ein synchronisierter Buszugriff erfolgen, bei dem vorgegeben ist, welches Steuergerät in welchem Zeitfenster auf den Bus zugreifen darf (Time Division Multiple Access TDMA). Hierfür wird in ISO 11898-4 folgendes Konzept für CAN vorgeschlagen:

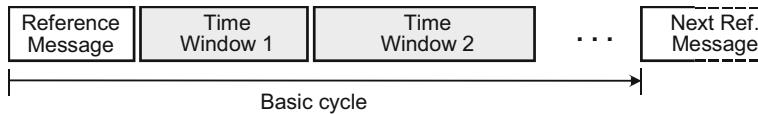


Abb. 3.9 TTCAN Grundzyklus

Von einem Steuergerät, das als Zeit-Master arbeitet, wird periodisch eine Referenz-Botschaft ausgesendet (Abb. 3.9). Damit beginnt ein TTCAN-Grundzyklus (Basic Cycle). Alle anderen Steuergeräte synchronisieren sich durch diese Nachricht mit dem Grundzyklus. Der folgende Abschnitt des Grundzyklus enthält eine frei wählbare Anzahl von Zeitfenstern mit frei definierbarer, auch unterschiedlicher Länge. Es gibt drei Arten derartiger Zeitfenster:

- In den exklusiv reservierten Zeitfenstern darf jeweils genau ein Steuergerät seine Botschaften senden, d. h. diese Zeitfenster sind für die zeitsynchrone (time triggered) Kommunikation vorgesehen. In diesen Fenstern kann es nicht zu einer Kollision auf dem Bus kommen.
- In den arbitrierenden Zeitfenstern dürfen mehrere Steuergeräte ihre Botschaften senden, es wird das gewöhnliche CSMA/CR-Buszugriffsverfahren verwendet, d. h. ereignisgesteuerte (event triggered) Kommunikation. Ist das Zeitfenster lang genug, können mehrere Geräte hintereinander ihre Botschaften übertragen. Jeder Sender muss allerdings vor dem Senden überprüfen, ob seine Botschaft bis zum Ende des Fensters noch vollständig übertragen werden kann.
- Freie Zeitfenster werden für spätere Erweiterungen vorgesehen, so dass weitere Botschaften hinzugefügt werden können, ohne dass das Kommunikationsschema geändert werden muss.

Wenn eine Botschaft sehr oft übertragen werden soll oder wenn ein Steuergerät viele Botschaften zu übertragen hat, können einem Steuergerät auch mehrere Zeitfenster innerhalb eines Grundzyklus zugeordnet werden. Da manche Botschaften nicht in jedem Grundzyklus übertragen werden müssen, können mehrere Grundzyklen zu einem so genannten Systemzyklus (oder Matrixzyklus) zusammengefasst werden, wobei dann die Zeitfenster in den einzelnen Grundzyklen unterschiedlich belegt werden und sich der Gesamtablauf erst mit jedem Systemzyklus wiederholt.

Das gesamte System wird statisch konfiguriert, d. h. die Zeitfenster werden in der Entwicklungsphase und nicht zur Laufzeit festgelegt und den jeweiligen Steuergeräten zugeordnet. Ein einzelnes Steuergerät muss nicht das gesamte Kommunikationsschema kennen, sondern nur wissen, in welchen Zeitfenstern es seine eigenen Botschaften senden darf und weiß, in welchen Zeitfenstern die zu empfangenden Botschaften ankommen müssen.

Um einen Totalausfall bei Defekt des Zeit-Masters zu vermeiden, kann ein System bis zu acht Zeit-Master haben, deren Referenzbotschaft über die Wahl des CAN-Identifiers mit

unterschiedlicher Priorität versehen wird, so dass stets der Master mit der aktuell höchsten Priorität den Grundzyklus vorgibt.

Die Zeitablaufsteuerung kann mit gewöhnlichen CAN-Controllern vollständig in Software realisiert werden, wenn die Zeitfenster ausreichend groß definiert werden und der Zeitjitter akzeptabel ist, der durch die Zeitablaufsteuerung und das Bereitstellen der Sendedaten durch die Software entsteht. Problematisch sind dabei allerdings die automatische Sendewiederholung bei Übertragungsfehlern sowie die automatische Wiederholung von Botschaften, die in den arbitrierenden Zeitfenstern den Buszugriff „verloren“ haben. Falls die Übertragungswiederholung nicht innerhalb des Zeitfensters abgeschlossen werden kann, muss die automatische Wiederholung abgeschaltet werden. Bei neueren CAN-Controllern ist dies in der Regel möglich.

Um die Mikrocontroller-Belastung durch eine rein softwaremäßige Umsetzung der Zeitablaufsteuerung zu vermeiden, stehen mittlerweile entsprechende CAN-Controller zur Verfügung, die das TTCAN-Protokoll weitgehend hardwaremäßig abwickeln und durch die Software lediglich konfiguriert werden müssen.

Die zeitsynchrone Kommunikation eignet sich besonders für Mess- und Regelungsaufgaben, die periodisch durchgeführt werden müssen. Um zusätzliche, insbesondere veränderliche Totzeiten zu verhindern, sollte dabei die Bearbeitung der Mess- und Regelaufgaben im Steuergerät zeitlich mit dem TTCAN-Grundzyklus synchronisiert werden. Für ereignisgesteuerte Botschaften, insbesondere solche mit hoher Priorität, ergibt sich bei TT-CAN in der Regel eine größere Latenzzeit als beim normalen CAN, da nur ein Teil des Grundzyklus für solche Botschaften reserviert werden kann.

Für die übergeordnete Synchronisation im System definiert TTCAN auch ein Verfahren, bei dem eine globale Systemzeit übertragen wird und lokal innerhalb eines Grundzyklus eine Driftkorrektur des lokalen Zeitgebers erfolgt.

TTCAN gewährleistet zwar die deterministische Übertragung von Botschaften, erlaubt aber keine höheren Bitraten als der klassische CAN-Bus, da die bitweise Arbitrierung in den ereignisgesteuerten Zeitfenstern und die positive bzw. negative Empfangsbestätigung innerhalb des *Acknowledge Bits* am Ende jeder Botschaft beibehalten wird. Damit ist das eigentliche Grundproblem, die für anspruchsvolle Anwendungen eigentlich nicht mehr ausreichende Bandbreite, nicht gelöst. TTCAN hat sich daher bisher nicht durchgesetzt. CAN soll aus diesem Grund mittelfristig in den Anwendungsbereichen Fahrwerks- und Triebstrangelektronik durch FlexRay abgelöst werden. Mittlerweile kündigt sich aber mit CAN Flexible Datarate (siehe übernächster Abschnitt) eine interessante Alternative an.

3.1.9 Energiesparmaßnahmen: Wakeup und Partial Networking

Während die meisten Steuergeräte bei abgestelltem Motor ausgeschaltet werden können, müssen einzelne Systeme, z. B. die Zentralverriegelung, weiterhin betriebsfähig sein. Um die Batterieentladung zu verringern, werden aber auch diese Geräte in einen stromsparenden Zustand versetzt (*Sleep Mode*) und von Zeit zu Zeit *aufgeweckt*. Gemäß

ISO 11898-5 können die an einem CAN-Bus angeschlossenen Steuergeräte z. B. durch ein *CAN Wakeup Pattern* aufgeweckt werden. Als *Wakeup Pattern* dient dabei ein dominantes Bit auf dem CAN-Bus mit einer Dauer von mehr als 5 µs nach einem vorherigen rezessiven Ruhezustand. Um Störungen zu unterdrücken, werden Impulse ignoriert, die kürzer sind als 0,75 µs. In der Praxis wird das *Wakeup Pattern* erzeugt, indem eine beliebige CAN-Botschaft versendet wird, die eine ausreichend lange Folge von 0 Bits enthält. Im Schlafzustand des Steuergerätes ist üblicherweise nur der CAN-Transceiver aktiv, der CAN-Kommunikationscontroller und die CPU sind abgeschaltet (Abb. 3.1). Nach dem Empfang des *Wakeup Patterns* dauert es daher einige Zeit, bis das Steuergerät tatsächlich betriebsfähig ist. Wie und wann der Übergang eines Steuergerätes und eines gesamten CAN-Bussystems vom aktiven zum stromsparenden Zustand und umgekehrt erfolgt, muss auf Anwendungsebene festgelegt werden. Zur Koordination der Abläufe dienen z. B. das OSEK bzw. AUTOSAR Netzmanagement (Abschn. 7.2.3 und 8.5) und der AUTOSAR ECU State Manager (Abschn. 8.2).

Mittlerweile hat man erkannt, dass es sinnvoll ist, wenn man nicht alle, sondern gezielt nur einzelne oder eine Gruppe von Geräten aktivieren kann (*Selective Wakeup*). Insbesondere denkt man daran, Steuergeräte nicht mehr nur bei abgestelltem Motor, sondern auch im Fahrbetrieb abzustellen, um auch dort Energie zu sparen. So macht es beispielsweise wenig Sinn, das Steuergerät der Einparkhilfe auch bei Autobahnfahrten aktiv zu halten.

Der Betrieb eines CAN-Systems, bei dem nur ein Teil der Steuergeräte aktiv ist, wird als Teilnetzbetrieb (*Partial Networking*) bezeichnet. Für den Teilnetzbetrieb wurde vorgeschlagen, nicht ein einfaches Bitmuster sondern eine komplette CAN-Botschaft, einen *Wakeup Frame*, zum Aufwecken zu verwenden. Um ein Steuergerät aufzuwecken, müssen der CAN *Message Identifier* und die Datenlänge (*Data Length Code DLC*) des *Wakeup Frames* mit den im Steuergerät vorkonfigurierten Werten übereinstimmen. Außerdem muss ein vordefiniertes Bit innerhalb des Datenfelds gesetzt sein. Dadurch lassen sich mit einer einzigen CAN-Botschaft gezielt bis zu 64 Steuergeräte selektiv aktivieren. Im Gegensatz zu einfachen *Wakeup Pattern*, die von den meisten heute erhältlichen CAN *Transceivers* erkannt werden können, ist die Erkennung einer vollständigen CAN-Botschaft und damit die entsprechenden CAN *Transceiver* erheblich aufwendiger. Da ein wesentlicher Teil der *Data Link Layer* Logik des CAN Kommunikationscontrollers im *Transceiver* dupliziert werden und eine Konfigurationsmöglichkeit des *Transceivers* durch den Mikrocontroller geschaffen werden muss, wird es einige Zeit dauern, bis die zugehörigen CAN-Standardisierungsarbeiten als ISO 11898-6 abgeschlossen und geeignete ICs auf dem Markt verfügbar sein werden, bevor dann die entsprechenden Funktionalitäten im Netzmanagement und auf Anwendungsebene definiert und umgesetzt werden können.

3.1.10 Höhere Datenraten: CAN Flexible Data-Rate CAN FD

Wegen seiner begrenzten Busbandbreite gilt CAN seit einiger Zeit als Flaschenhals für zukünftige Fahrwerks- und Triebstrangsysteme. Der als CAN-Nachfolger konzipierte Flex-

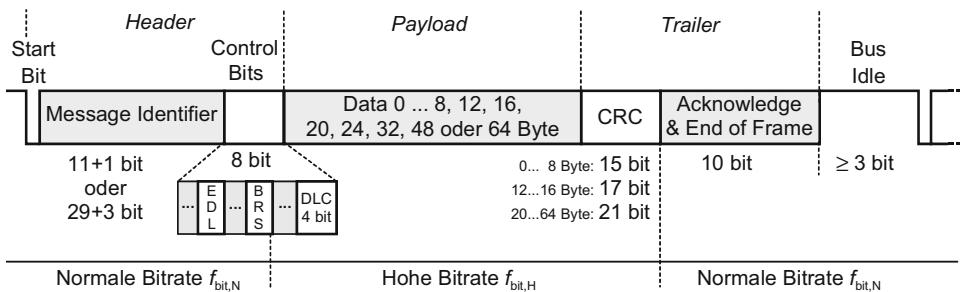


Abb. 3.10 Botschaftsformat bei CAN FD (Längenangaben ohne Bit-Stuffing)

Ray Bus (siehe Abschn. 3.3) konnte bisher jedoch nicht alle Hoffnungen erfüllen. Zum einen steigt die nutzbare Busbandbreite wegen des gerade bei kurzen Botschaften verhältnismäßig ineffizienten FlexRay-Protokolls weniger stark als es die höhere Bitrate erwarten lässt. Zum anderen ist dessen TDMA-Konzept relativ starr, so dass das Kommunikations-schema in der Entwicklungsphase sehr sorgfältig geplant werden muss, weil nachträgliche Änderungen schwierig sind. Auch die Umstellung der Software vom ereignis- zum zeit-gesteuerten Betrieb macht den Übergang vorhandener CAN-Systeme auf FlexRay in der Praxis aufwendig. Mit dem Bosch-Konzept *CAN with Flexible Data-Rate CAN FD*, dessen Spezifikation in Version 1.0 seit 2012 vorliegt, soll die Bandbreite von CAN Systemen in aufwärtskompatibler Weise gesteigert werden. Dies erfolgt durch zwei Maßnahmen:

- Längere Botschaften

Die Nutzdatenlänge einer CAN-Botschaft wird von maximal 8 Byte auf 64 Byte erhöht. CAN FD Botschaften können damit wie bisher 0 bis 8 Byte oder nun auch 12, 16, 20, 24, 32, 48 oder 64 Byte Nutzdaten enthalten. Da die Längenangabe *Data Length Code DLC* innerhalb der *Control Bits* im Header (Abb. 3.4) schon bisher 4 bit groß war, müssen dazu lediglich die bisher ungenutzten Codes vergeben werden. Außerdem muss die CRC-Prüfsumme im Trailer angepasst werden, um bei längeren Botschaften dieselbe Datensicherheit zu gewährleisten. Botschaften mit bis zu 8 Byte Nutzdaten verwenden weiterhin die bisherige 15 bit CRC. Für 12 und 16 Byte verlängert sich die Prüfsumme auf 17 bit, ab 20 Byte wird eine 21 bit CRC verwendet. Um eine CAN FD Botschaft von einer normalen CAN-Botschaft unterscheiden zu können, wird im Header das neue *Extended Data Length EDL* Bit eingeführt (Abb. 3.10).

- Höherer Bittakt

Die begrenzenden Faktoren für die Bitrate bei CAN sind die bitweise Arbitrierung des *Message Identifiers* im Botschaftsheader zur Kollisionserkennung sowie das *Acknowledge Bit* zur Bestätigung des korrekten Empfangs im Botschaftstrailer. Die Auswertung dieser Bits muss in allen Steuergeräten des Busses zuverlässig innerhalb desselben Bittaktes erfolgen. Weil die Signallaufzeit auf den Busleitungen unveränderbar ist, entsteht dadurch die in Gl. 3.1 beschriebene Abhängigkeit zwischen Buslänge und minimaler Bitdauer

Tab. 3.5 Abschätzung der Nutzdatenraten (für 11 bit *Message Identifier*)

n_{Data}	Klassischer CAN 2.0 $f_{\text{bit}} = 500 \text{ kbit/s}$	CAN FD ohne Bitratenumschaltung $f_{\text{bit,N}} = f_{\text{bit,H}} = 500 \text{ kbit/s}$	CAN FD mit Bitratenumschaltung $f_{\text{bit,N}} = 500 \text{ kbit/s}, f_{\text{bit,H}} = 4 \text{ Mbit/s}$
8 Byte		29 KB/s	79 KB/s
16 Byte		35 KB/s	131 KB/s
32 Byte	Nicht möglich	40 KB/s	195 KB/s
64 Byte		44 KB/s	260 KB/s

bzw. maximaler Bitrate. Für das eigentliche Nutzdatenfeld und die CRC-Prüfsumme dagegen gilt diese Forderung nicht unbedingt. CAN FD schlägt daher vor, die Bitrate in diesem Teil der Botschaft deutlich zu erhöhen. Die vorliegende Spezifikation 1.0 nennt allerdings keine konkreten Zahlenwerte. In begleitenden Veröffentlichungen von Bosch [8] werden für den aus ISO 11898-2 bekannten Physical Layer Zielwerte um 4 Mbit/s genannt. Die Bitratenumschaltung ist optional und wird daher durch das neu eingeführte Steuerbit *Bit Rate Switch BRS* im *Control Feld* des Headers signalisiert (Abb. 3.10). Mit dem Ende des CRC-Felds im Trailer wird auf die normale Bitrate zurückgeschaltet.

Der Header einer CAN FD-Botschaft verlängert sich um die neu eingeführten EDL und BRS Bits. Der Trailer vergrößert sich durch die modifizierte CRC-Prüfsumme je nach Nutzdatenlänge um maximal 6 bit. Analog zu Gl. 3.4 lässt sich die Dauer einer Botschaft abschätzen zu

$$T_{\text{Frame}} < 1,25 \cdot \left[29 \text{ bit (49 bit)} \cdot \frac{1}{f_{\text{bit,N}}} + \{26 \text{ bit} + n_{\text{Data}}\} \cdot \frac{1}{f_{\text{bit,H}}} \right] \quad (3.11)$$

wobei n_{Data} die Anzahl der Datenbits darstellt und der Faktor 1,25 wieder das *Bit Stuffing* im ungünstigsten Fall berücksichtigt. Der Wert in (...) gilt für 29 bit *Message Identifier*. Die Nutzdatenrate ist wieder $f_{\text{Data}} = n_{\text{Data}} / T_{\text{Frame}}$. Wie man an den Daten von Tab. 3.5 sieht, verspricht CAN FD tatsächlich eine massive Vergrößerung der Nutzdatenrate, falls die Umschaltung auf hohe Bitraten zuverlässig funktioniert.

3.1.11 Zusammenfassung CAN – Layer 1 und 2

- Kommunikation zwischen mehreren Kfz-Steuergeräten zum Austausch von Mess-, Steuer- und Regelsignalen im Echtzeitbetrieb mit hoher Fehlersicherheit.
- Bitstrom-orientiertes Übertragungsprotokoll mit bidirektonaler Zwei-Draht-Leitung als Linien-Bus. Buslänge und zulässige Länge der Stichleitungen abhängig von der Bitrate, max. 1 Mbit/s bei $< 40 \text{ m}$ Buslänge und Stichleitungen $< 30 \text{ cm}$.

- Übliche Bitraten im Kfz 500 kbit/s (High Speed CAN) und 100 ... 125 kbit/s (Low Speed CAN), andere Bitraten möglich.
- Botschaften mit 0 ... 8 Datenbytes und 6 ... 8 Byte Overhead (Header/Trailer)
- CAN-Controller, Transceiver und Mikroprozessor notwendig.
- Broadcast-System, bei dem die Nachrichten durch Message Identifier gekennzeichnet und beim Empfänger davon abhängig Empfangsentscheidungen (Akzeptanzfilterung) getroffen werden. Die Message Identifier priorisieren gleichzeitig Botschaften beim Sendeversuch (Buszugriff mit CSMA/CR). Message Identifier mit 11 bit und 29 bit Länge (CAN 2.0A und 2.0B), können im selben CAN-Netz gemischt verwendet werden.
- CAN-Controller mit Hardware-Akzeptanz-Filterung (Full CAN) und Software-Akzeptanz-Filterung (Basic CAN) erhältlich.
- Netzweite Datenkonsistenz, da alle CAN-Controller die empfangenen Daten ignorieren, wenn ein oder mehrere Geräte einen Übertragungsfehler erkennen. Automatische Übertragungswiederholung im Fehlerfall. Automatische Abschaltung von defekten CAN-Controllern.
- Übertragungsdauer einer Botschaft mit 8 Datenbytes bei 500 kbit/s unter Berücksichtigung des *Bit-Stuffings* im ungünstigsten Fall:

270 μ s (11 bit Identifier), 320 μ s (29 bit Identifier).

Entspricht der Worst Case Latenz für die Botschaft mit der höchsten Priorität.

- Theoretisch mögliche Nutzdatenrate 29 KB/s (11 bit Identifier), 24 KB/s (29 bit Identifier)
- Verfahren zur Vergrößerung der Nutzdatenrate durch bis zu 64 Byte je Botschaft und dynamische Umschaltung auf höhere Bitraten in Entwicklung.

3.2 Local Interconnect Network LIN

LIN (Local Interconnect Network) ist ein relativ junges Bussystem, das Ende der 90er Jahre entwickelt wurde, um eine kostengünstige Alternative zu Low-Speed-CAN-Bussystemen für einfache Sensor-Aktor-Anwendungen zu bieten, z. B. für die Tür-, Sitz- oder Schiebedach-Elektronik [10]. Triebfeder war der Halbleiterhersteller Motorola (heutiger Name: Freescale) in Zusammenarbeit mit verschiedenen Kfz-Herstellern, die sich zu einem LIN-Konsortium zusammengeschlossen haben. Die Spezifikationen sind offen gelegt und da ein einfaches, zeichenorientiertes Protokoll verwendet wird, das mit jedem UART implementiert werden kann, ist es praktisch frei verfügbar.

3.2.1 Überblick

Der Physical Layer und die Bitübertragungsschicht (8N1 UART-Zeichen mit 8 Datenbits und je einem Start- und Stopbit ohne Parität) entsprechen dem K-Line-Protokoll (siehe

Abschn. 2.2). Die mögliche Bitrate liegt im Bereich 1 ... 20 kbit/s, wobei die drei Bitraten 2,4 kbit/s, 9,6 kbit/s bzw. 19,2 kbit/s empfohlen werden. In der ersten LIN-Generation (LIN Spezifikation V1.3) lag der Schwerpunkt darauf, möglichst kostengünstige Netzknoten zu ermöglichen. Um LIN schnell in den Markt zu bringen, wurde es als Sub-Bus für CAN-Netze positioniert, bei dem ein Master-Knoten (in der Regel gleichzeitig) als Gateway zu einem übergeordneten CAN-Netz fungiert und als einziger Knoten über eine präzise Zeitbasis verfügen und sich um die gesamte Netzkonfiguration kümmern muss. Sämtliche anderen Netzknoten sind Slave-Knoten, deren Bittakt selbstsynchronisierend ausgeführt werden kann und die keinerlei Konfigurationsinformationen über das Gesamtnetzwerk benötigen. Auf der Bussystem-Ebene verfügt LIN nur über wenige Mechanismen, um Übertragungsfehler zu erkennen, und über keinerlei vorgegebene Verfahren, derartige Fehler selbstständig zu korrigieren.

Mit der Version V2.0 der LIN-Spezifikation wurde das Protokoll nicht nur in einigen Teilen präziser spezifiziert, sondern auch um verschiedene, teilweise optionale Mechanismen erweitert, die die Geräte erheblich komplexer machen. Dazu gehört das optionale Tunneln von KWP 2000 oder UDS-Diagnosebotschaften, was dann aber die Implementierung von ISO-Diagnosefunktionen in den LIN-Steuergeräten erfordert, sowie ein zwingend geforderter *Plug-and-Play*-Mechanismus, der die automatische Konfiguration von LIN-Slave-Steuergeräten durch das LIN-Master-Steuergerät erlaubt. Der Mechanismus ist aber nur mit Steuergeräten sinnvoll, die über Flash-ROM- bzw. EEPROM-Speicher verfügen und wurde teilweise kritisiert, weil dies kostentreibend wirkt und damit eigentlich der LIN-Grundidee widerspricht.

Zusätzlich sind die Änderungen nicht vollständig rückwärtskompatibel. So wurde u. a. die Art der Prüfsummenberechnung verändert. Ein V2.0 Master-Steuergerät muss mit V1.3 Slave-Steuergeräten zusammenarbeiten können, während ein V1.3 Master-Steuergerät mit LIN V2.0 Slave-Steuergeräten nicht kommunizieren kann. Mit LIN V2.1 und V2.2 wurden einige Details weitgehend aufwärts kompatibel ergänzt, insbesondere aber einige Unklarheiten aus den Vorgängerversionen beseitigt.

Eine auf Betreiben von GM und Ford leicht modifizierte Variante von LIN V2.0, z. B. Bitrate max. 10,4 kbit/s, wurde als SAE J2602 zur Standardisierung eingereicht. Darüber hinaus gibt es proprietäre LIN-Varianten z. B. bei Toyota oder den sogenannten *Cooling Bus* der Hersteller von Klimaanlagen.

Nach anfänglicher Euphorie zeigte sich, dass das ursprüngliche Ziel, LIN-Knoten zur Hälfte der Kosten eines Low-Speed-CAN-Knotens zu implementieren, nur unter größten Anstrengungen zu erreichen ist. Dies liegt u. a. daran, dass CAN-Bustransceiver aufgrund der riesigen Stückzahlen nur unwesentlich teurer sind als K-Line-Bustransceiver. Auch LIN-Transceiver sind wegen der notwendigen Spannungsfestigkeit lediglich bei wenigen Halbleiterprozessen direkt auf dem Mikrocontroller integrierbar. Wenn die extrem niedrigen Bitraten von LIN tatsächlich ausreichen, können auch Low-Speed-CAN-Netze mit Eindraht-Leitungen realisiert werden. Und schließlich sind die Mehrkosten von CAN-Controllern gegenüber UARTs bei direkter Integration im Mikrocontroller und den heutigen Integrationsdichten fast vernachlässigbar. Umgekehrt macht die notwendige CAN-

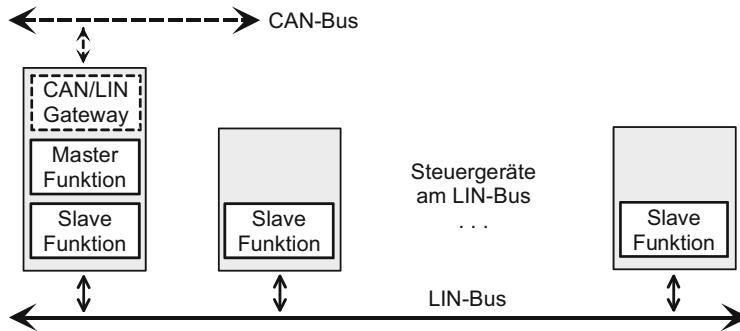


Abb. 3.11 Struktur eines LIN-Bussystems

LIN-Gateway-Funktionalität das Gesamtnetz komplexer und damit auch fehleranfälliger. Trotzdem sind LIN-Busse in der Karosserieelektronik, etwa bei Tür-, Spiegel- und Fensterhebersteuerungen oder Multi-Funktions-Lenkrädern und Innen- und Außenlichtsteuergeräten, mittlerweile weit verbreitet. Die weitere Standardisierung erfolgt inzwischen im Rahmen von ISO 17987.

3.2.2 Data Link Layer

In LIN-Bussystemen, in den LIN-Spezifikationen als *LIN Cluster* bezeichnet, steuert genau ein fest definiertes Steuergerät als *Master* durch Senden von Botschaftsheadern den gesamten Kommunikationsablauf, worauf eines der Steuergeräte als *Slave* mit einer Daten-Botschaft antwortet. Dabei soll auch im Master-Steuergerät das Senden der Daten-Botschaften vom Senden der Header-Botschaften in der Protokollsoftware logisch getrennt werden, so dass das Senden von Daten in allen Steuergeräten grundsätzlich als Slave-Funktionalität realisiert wird (Abb. 3.11). Das Master-Steuergerät übernimmt dabei in der Regel auch die Gateway-Funktion zum übergeordneten CAN-Bus.

Das Master-Steuergerät sendet einen Botschaftsheader aus (Abb. 3.12), der mit mindestens 13 Low-Bits und 1 High-Bit beginnt, von denen der Slave mindestens 11 Low-Bits erkennen muss, nachdem der Bus vorher im Ruhezustand (*Bus Idle*) war. Diese als *Sync Break* bezeichnete Bitfolge ist das einzige nicht Standard-UART gemäße Zeichen und daher von jedem Empfänger zu jedem Zeitpunkt eindeutig als Botschaftsbeginn erkennbar. Danach folgt ein *Sync Byte* mit einer alternierenden Low-High-Bitfolge (55h), die eine Bittakt-Synchronisation der Empfänger erlaubt. Als letztes Zeichen des Headers sendet der Master das *Identifier Byte*. Mit dem LIN-Identifier wird wie bei CAN der Inhalt der folgenden Botschaft gekennzeichnet. Genau ein Steuergerät ist so konfiguriert, dass es auf diesen LIN-Identifier mit einer Daten-Botschaft mit 1 bis 8 Datenbyte und einem Prüfsummenbyte antwortet (*Unconditional Frames*).

Wie CAN arbeitet auch LIN verbindungslos. Der LIN-Identifier kennzeichnet den Inhalt der gewünschten Daten-Botschaft und nicht eine Stationsadresse (inhaltsbezogene

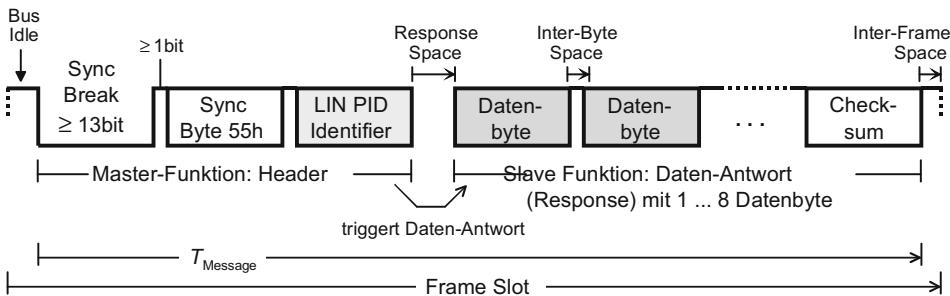


Abb. 3.12 LIN-Botschaftsformat

Tab. 3.6 LIN-Identifier (ohne Paritätsbits)

Identifier (6 bit)	Bedeutung
00h ... 3Bh	Beliebige LIN-Botschaften, Inhalt durch LIN-Konfigurationsdatei festgelegt
3Ch	Master-Request-Frame (MRF) für Transportprotokoll-Botschaften (Diagnose und Konfiguration, siehe Abschn. 3.2.5) 1. Datenbyte 00h: Befehl zum Umschalten in den <i>Sleep Modus</i> 1. Datenbyte ungleich 00h: Diagnosebotschaft nach Abb. 3.13
3Dh	Slave-Response-Frame (SRF) für Transportprotokoll-Botschaften
3Eh, 3Fh	Reserviert für Erweiterungen

Adressierung). Alle am Bus angeschlossenen Steuergeräte müssen die Header empfangen und auswerten und können alle Daten-Botschaften mithören. Der Identifier besteht aus 6 bit, mit denen der Inhalt der folgenden Daten-Botschaft gekennzeichnet wird, sowie 2 Paritätsbits, mit denen der Identifier gegen Übertragungsfehler gesichert wird. Das aus dem eigentlichen Identifier und den beiden Paritätsbits bestehende Byte wird in der LIN Spezifikation als *Protected Identifier PID* bezeichnet. Bei LIN 1.x wurde mit dem Identifier auch die Länge der Botschaft codiert, dabei waren 32 Identifier für Botschaften mit je 2 Datenbyte, 16 mit jeweils 4 Datenbyte und die übrigen für Botschaften mit je 8 Datenbyte vorgesehen. Bei LIN 2.x wird die Datenlänge in der Konfigurationsdatei des Systems (siehe Abschn. 3.2.6) unabhängig vom Wert des Identifiers beliebig zwischen 1 und 8 Datenbyte festgelegt. Für die Dateninhalte selbst gibt es in der LIN-Spezifikation keine Festlegung. Die Identifier 3Ch und 3Dh sind für Kommando- bzw. Diagnose-Botschaften (*Diagnostic Frames*) reserviert (Tab. 3.6), die die nachfolgend beschriebene Transportschicht des LIN-Protokolls verwenden. Die Identifier 3Eh und 3Fh sind für zukünftige LIN-Protokollerweiterungen vorgesehen (*Extended Frames*). (Hinweis: Die für die Identifier hier und im LIN-Standard angegebenen Werte verstehen sich jeweils ohne die beiden Paritätsbits).

Die Daten-Antwort des Slaves wird durch eine Prüfsumme abgesichert. Bei LIN V1.3 umfasst die Prüfsumme nur die Datenbytes (*Classic Checksum*). LIN V2.0 Slaves dagegen müssen in die Berechnung der Prüfsumme zusätzlich noch den Identifier aus dem zugehörigen Header mit einbeziehen (*Enhanced Checksum*). Die einzelnen Bytes werden wie bei

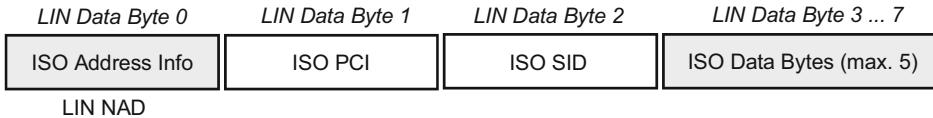


Abb. 3.13 Abbildung einer ISO 15765-2 Single Frame Diagnosebotschaft auf LIN

UARTs üblich von einem Startbit und einem Stoppbit eingerahmt und mit dem LSB zuerst übertragen. Die Übertragung von Mehr-Byte-Daten erfolgt in *Little-Endian*-Reihenfolge, d. h. das niederwertige Byte wird zuerst gesendet.

Die Fehlererkennung und -behandlung ist bei LIN nur rudimentär definiert. Der Sender einer Botschaft erhält weder positive Empfangsbestätigungen noch Fehlermeldungen von den Empfängern. Der jeweilige Sender muss das von ihm gesendete Bussignal mitlesen. Stellt er einen Bitfehler auf der Busleitung fest, bricht er den Sendevorgang ab. Der Master muss erkennen, wenn kein Slave auf eine Header-Botschaft antwortet oder falls bei der Daten-Antwort eine Zeitüberschreitung auftritt. Die Slaves ignorieren Botschaften mit unbekanntem Identifier oder mit fehlerhafter Identifier-Parität sowie Daten-Botschaften mit fehlerhafter Prüfsumme. Festgestellte Fehler werden in der lokalen LIN-Protokoll-Software als Statusinformation gespeichert und können von der Applikationsschicht des jeweiligen Busteilnehmers abgefragt werden (siehe Abschn. 3.2.8). Die weitere Fehlerreaktion der Applikationsschicht ist anwendungsabhängig und in der LIN-Spezifikation nicht definiert. Mit LIN V2.0 neu eingeführt wurde, dass jeder LIN-Slave mindestens ein Status-Bit (*Response Error Bit*) bereitstellen muss, das in einer der periodisch an den Master gesendeten Botschaften enthalten sein muss. Das *Response Error Bit* soll gesetzt werden, wenn der Slave beim Empfang einer Botschaft oder beim Senden einer eigenen Botschaft einen Fehler festgestellt hat. Nach dem Versenden der Botschaft mit dem *Response Error Bit* setzt der Slave das Bit selbstständig zurück. Der Master soll die *Response Error Bits* aller Slave sowie seine eigene Fehlerüberwachung auswerten und in seiner Applikationsschicht geeignet reagieren. Mit welcher Periodendauer die Statusinformation gesendet werden muss und wie der Master reagieren soll, bleibt weiter anwendungsabhängig.

Mit dem Identifier 3Ch und 00h als erstem Datenbyte kann der Master alle Slaves in den so genannten *Sleep Modus* schalten, in dem alle Busaktivitäten eingestellt werden. Ein Slave aktiviert den *Sleep Modus* außerdem selbstständig, wenn der Bus für mindestens 25.000 Bitzeiten (bei LIN V2.0 frühestens nach 4 s und spätestens nach 10 s) inaktiv war. Jeder Busteilnehmer kann den *Sleep Modus* des Busses beenden, indem er eine *Wake Up* Botschaft sendet. Eine *Wake Up* Botschaft besteht aus einem einzelnen 80h Zeichen (ohne Header und Checksumme). Bei LIN V2.0 wurde dies geändert in ein *Low-Signal* für 0,25 ... 5 ms. Alle angeschlossenen Busteilnehmer müssen die *Wake Up* Botschaft erkennen und sich innerhalb von 100 ms neu initialisieren. Der Master beginnt nach mindestens 4 und höchstens 64 Bitzeiten (bei LIN V2.0 nach 100 ms) wieder mit dem Aussenden von Header-Botschaften. Falls dies unterbleibt, wiederholt das Steuergerät die *Wake Up* Bot-

schaft bis zu drei Mal im Abstand von 150 ... 250 ms. Bei Misserfolg kann dieser Ablauf im Abstand von min. 1,5 s wiederholt werden.

Das Timing der LIN-Botschaften ist sehr großzügig spezifiziert. Der Bittakt des Masters soll nicht mehr als $\pm 0,5\%$ vom Nominalwert abweichen. Der Bittakt der Slaves dagegen darf vor der Synchronisation um bis zu $\pm 15\%$ abweichen, wenn er durch das *Sync Byte* auf $\pm 2\%$ genau nachsynchronisiert wird und diese Toleranz bis zum Ende der Botschaft nicht überschreitet. Der Bittakt von Slaves, die sich nicht automatisch nachsynchronisieren, darf um $\pm 1,5\%$ vom Nominalwert abweichen. Für die Dauer des *Sync Break* ist lediglich eine Minimallänge von 14 bit vorgegeben und die Pause zwischen *Sync Byte* und *Identifier Byte* ist beliebig, solange die vorgegebene Maximallänge von 49 Bitzeiten nicht überschritten wird. Ähnlich freizügig wählbar sind der Abstand zwischen dem Ende des Headers und dem Beginn der Daten-Antwort (*Response Space*) sowie die maximal zulässige Pause zwischen den einzelnen Bytes der Daten-Antwort (*Inter Byte Space*). Einzelheiten zu den Zeitbedingungen werden im Abschn. 3.2.8 erläutert.

3.2.3 Neue Botschaftstypen bei LIN V2.0

Neben den ständig periodisch zu sendenden LIN-Botschaften (*Unconditional Frames*) gibt es seit LIN V2.0 auch *dynamische Botschaften*, die als *Sporadic Frames* und *Event Triggered Frames* bezeichnet werden. Wie für normale Botschaften müssen auch für diese Botschaften entsprechende Zeitschlüsse in der Botschaftstabelle reserviert werden.

Sporadic Frames werden aber nur dann versendet, wenn tatsächlich Daten im Master-Steuergerät vorliegen oder vom Master-Steuergerät Slave-Antworten gefordert werden, ansonsten erzeugt der Master keine Header-Botschaft und der Bus bleibt während des entsprechenden Zeitschlusses einfach in Ruhe. Bei den *Sporadic Frames* ist es möglich und üblich, für mehrere Botschaften mit unterschiedlichen Identifiern denselben Zeitschlitz zu reservieren, wobei bei der Konfiguration eine statische Priorität für die Botschaft vergeben wird, so dass der Scheduler im Master jeweils nur die höchspriore Botschaft versendet.

Event Triggered Frames werden verwendet, um mehrere Slave-Steuergeräte mit einer einzigen Botschaft zyklisch abzufragen. *Event Triggered Frames* sind der einzige Botschaftstyp, bei dem mehrere Slave-Funktionen so konfiguriert werden, dass sie auf dieselbe Botschaft antworten können. Tatsächlich antwortet ein Slave allerdings nur dann, wenn sich die zu dieser Botschaft gehörenden Daten innerhalb des Steuergerätes geändert haben. *Event Triggered Frames* sind daher nur für solche Daten geeignet, die zwar von mehreren Geräten geliefert werden, bei denen sich der Wert in der Regel aber nur in einem Gerät tatsächlich ändert. Sinnvoll wäre dies beispielsweise bei der Abfrage von Türkontakteinschaltern, wenn zu erwarten ist, dass nicht mehrere Türen gleichzeitig geöffnet oder geschlossen werden. Falls sich die zugehörigen Daten in keinem der Slave-Steuergeräte geändert haben, antwortet keines der Steuergeräte. Haben sich ausnahmsweise die Daten in mehreren Slave-Steuergeräten gleichzeitig geändert und versuchen daher mehrere Geräte, auf die Header-Botschaft zu antworten, so kommt es bei der Antwort zu einer Kollision.

Die Slaves erkennen die Kollision durch Rücklesen der gesendeten Daten und brechen das Senden vorzeitig ab. Der Master erkennt dies als Empfangs- oder Timeout-Fehler. In den nächsten diesem *Event Triggered Frame* zugeordneten Zeitschlitten fragt der Master die Slave-Steuergeräte der Reihe nach mit normalen Botschaften (d. h. mit separaten Identifizieren) ab, bevor er wieder den *Event Triggered Frame* sendet. D. h. für jedes Steuergerät, das auf den Identifier eines bestimmten *Event Triggered Frame* reagiert, muss es zusätzlich auch noch einen jeweils eindeutigen *normalen* Identifier geben. Aufgrund der Verzögerungen bei Kollisionen sind *Event Triggered Frames* nicht mehr streng deterministisch. Bei LIN V2.1 wird die Kollisionsbehandlung präziser spezifiziert und gefordert, dass der Master nach einer erkannten Kollision auf eine andere Botschaftstabelle umschalten soll, die Kollisionsfreiheit gewährleistet. Diese soll einmalig abgearbeitet werden, bevor wieder auf die ursprüngliche Botschaftstabelle zurückgeschaltet wird.

3.2.4 LIN Transportschicht und ISO Diagnose über LIN

Da Steuergeräte am LIN-Bus wie alle übrigen Kfz-Steuergeräte für die Werkstattdiagnose zugänglich sein müssen, implementiert in der Regel das Master-Steuergerät eine der Standard-Diagnoseschnittstellen (KWP 2000 mit CAN bzw. UDS mit CAN, siehe Kap. 5). Der Datenaustausch mit den LIN-Slave-Steuergeräten dagegen erfolgte in LIN V1.3 ausschließlich über normale LIN-Botschaften mit den üblichen LIN-Botschafts-Identifizieren, so dass die LIN-Slaves selbst keine KWP 2000- oder UDS-Protokollsoftware enthalten müssen. Mit LIN V2.0 dagegen wurde die optionale Möglichkeit eingeführt, das KWP 2000- bzw. UDS-Protokoll nach ISO 15765 und ISO 14229 mit Hilfe eines Transportprotokolls ähnlich ISO TP (siehe Kap. 4) über den LIN-Bus zu „tunneln“ (*LIN Transport Layer*). In diesem Fall muss das LIN-Slave-Steuergerät zusätzlich zum LIN-Protokoll die Anwendungsschicht des UDS bzw. KWP 2000-Protokolls realisieren, was dem ursprünglichen LIN-Ziel, möglichst einfache Slave-Steuergeräte zu ermöglichen, allerdings nicht unbedingt entspricht. Mit LIN V2.1 wurde die pauschale Forderung, das Diagnoseprotokoll zu unterstützen, abschwächend spezifiziert, indem sogenannte Geräteklassen (*Diagnostic Class*) eingeführt wurden. Bei Slave-Geräten der Klasse I, einfachen Sensor-Aktor-Geräten, wird lediglich die Unterstützung von Single Frame-Botschaften für die dynamische Gerätekonfiguration gefordert (siehe Abschn. 3.2.6). Geräte der Klasse II sollen alle Transportprotokollbotschaften (siehe unten) unterstützen, müssen aber nur den UDS/KWP 2000-Diagnosedienst *Read Data By Identifier* implementieren, um Steuergerätedaten, z. B. die Gerätekennung, auslesen zu können. In der höchsten Klasse III wird zusätzlich die Unterstützung von Diagnosesitzungen (*Diagnostic Session Control*), Lesen und Ansteuern von Steuergeräteein- und -ausgängen (*Input Output Control*), des Fehlerspeichers (*Read/Clear Diagnostic Trouble Code Information*) sowie optional der Funktionen für die Flash-Programmierung (siehe Tab. 5.19 und 5.20) gefordert.

Beim Tunneln des UDS/KWP 2000-Protokolls über LIN werden UDS/KWP 2000-Diagnose-Request-Botschaften vom Master-Steuergerät/Gateway (MRF) als LIN-Bot-

Tab. 3.7 Geräteklassen für Transportprotokoll und Diagnose

Gerätekategorie	Transportschicht nach ISO 15765-2 (Kap. 4)	Diagnostic Service Identifier SID nach ISO 15765-3 bzw. 14229 (siehe Kap. 5)
I	Nur Single Frame	Konfigurationsdienste siehe Abschn. 3.2.6 (B2h, B7h, optional B0h, B3h … B6h)
II	Single, First und Consecutive Frame	Konfigurationsdienste wie Klasse I Diagnosedienste 22h, 2Eh
III		Wie Klasse II, zusätzlich Diagnosedienste 10h, 14h, 19h, 2Fh, optional 31h und Flash-Programmierung

schaften mit dem Identifier 3Ch versendet und Diagnose-Response-Botschaften der Slaves (SRF) mit dem Identifier 3Dh abgefragt. Das ISO 15765-2-Botschaftsformat mit Single Frames sowie das Multi-Frame-Format mit First Frames und Consecutive Frames nach Abb. 4.2 wird in die 8 Datenbytes der LIN-Botschaft abgebildet, die bei CAN notwendige Flusssteuerung mit Flow-Control-Botschaften ist bei LIN wegen des zeitsynchronen Botschaftsverkehrs nicht notwendig.

Auf LIN-Ebene sind diese Botschaften als Broadcast-Botschaften zu verstehen, d. h. alle LIN-Slave-Steuengeräte, welche die LIN-ISO-Diagnose unterstützen, müssen die Request-Botschaften mit dem Identifier 3Ch empfangen. Die Auswahl des Steuengerätes, für das die Diagnoseanfrage bestimmt ist, erfolgt über die ISO-Adresse (siehe Abschn. 5.1.3), bei LIN als *Node Diagnostic Address NAD* bezeichnet, die im ersten Datenbyte übertragen wird. Als Geräteadressen sind die Werte 1 bis 7Dh vorgesehen. Mit 7Eh ist eine funktionale Adressierung möglich und 7Fh dient als Broadcast Adresse. Der Slave verwendet in seiner Antwort seine eigene Adresse. Die Master-Request- und Slave-Response-Botschaften mit den Identifizierten 3Ch und 3Dh müssen immer 8 Datenbytes lang sein, leere Datenbytes werden mit FFh aufgefüllt.

Auch für Diagnosebotschaften muss ein Zeitschlitz in der Botschaftstabelle festgelegt werden, der aber nur verwendet wird, wenn das Master-Steuengerät eine Request-Botschaft abzusetzen hat oder eine Slave-Response-Botschaft erwartet.

Während ISO 15765-2, KWP 2000 und UDS relativ detailliert enge Zeitschränke für die Überwachung des Botschaftsverkehrs vorschreiben (siehe z. B. Tab. 3.7), enthält LIN V2.0 keine konkreten Vorgaben. Seit LIN V2.1 wird immerhin gefordert, dass die Zeitspanne zwischen Sendeanforderung und tatsächlichem Versenden einer Botschaft sowie zwischen zwei aufeinanderfolgenden Consecutive Frames nicht mehr als eine Sekunde betragen soll, wobei in einer konkreten Anwendung auch kürzere Zeiten konfiguriert werden dürfen.

3.2.5 LIN Configuration Language

Die LIN-Spezifikation sieht eine Konfigurationssprache vor, mit deren Hilfe die Netzwerkteilnehmer (*Nodes*), die zu übertragenden Daten (*Signals*) und die daraus zusammengesetzten

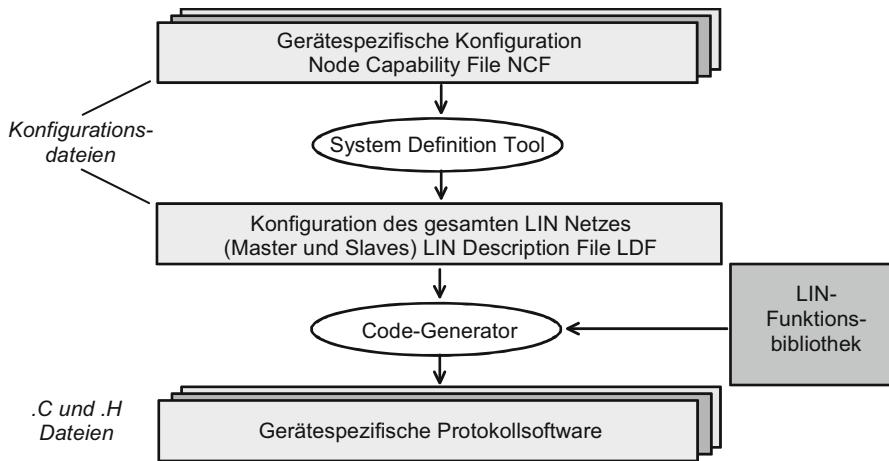


Abb. 3.14 LIN-Konfiguration

stellten Botschaften (*Frames*) spezifiziert werden können. Obwohl es sich bei den Konfigurationsdateien um einfache Textdateien handelt, die theoretisch mit jedem Texteditor erstellt werden können, verwendet man in der Praxis spezielle Konfigurationswerkzeuge. Aus den Konfigurationsdateien für die einzelnen Netzwerkeinnehmer (*Node Capability File NCF*) lässt sich dann die Konfigurationsdatei für das Gesamtnetz (*LIN Description File LDF*) erzeugen (Abb. 3.14). Daraus kann dann ein weiteres Werkzeug automatisch einen Satz von C-Code- und Header-Dateien für die Steuergeräte generieren, die die Master- und Slave-Funktionen implementieren.

Außerdem ist es möglich, auf Basis der LIN-Konfigurationsdatei das Netzwerkverhalten zu simulieren.

Der frei verfügbare LIN-Standard definiert allerdings nur die Konfigurationssprache sowie die Programmierschnittstelle (API) zwischen der LIN-Protokoll-Software und der Anwendungs- bzw. UART-Treibersoftware des Steuergerätes (Abb. 3.15). Bei den Werkzeugen dagegen handelt es sich um kommerzielle Produkte, die von verschiedenen Herstellern angeboten werden.

Die LIN-Konfigurationsdatei enthält folgende Hauptabschnitte (Tab. 3.8):

- Kopf mit Versionsnummer des Busprotokolls und der Konfigurationssprache sowie Bitrate des Netzes.
- Nodes { . . . }: Definiert symbolische Namen für das Master-Steuergerät und für alle Slave-Steuergeräte. Weiterhin werden für das Master-Steuergerät die Scheduling-Periodendauer und deren Zeittoleranz (*Jitter*) angegeben (siehe unten). Im Abschnitt Node_attributes { . . . }, der in LIN V2.0 neu ist, werden für jedes Steuergerät die unterstützte Protokollversion, die Diagnoseadresse NAD, die *LIN Product Identification*

Tab. 3.8 LIN-Konfigurationsdatei

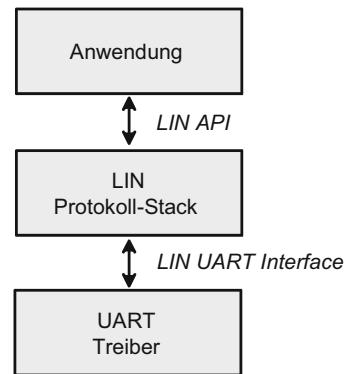
```

LIN_description_file;
LIN_protocol_version = "1.3";
LIN_language_version = "1.3";
LIN_speed = 19.2 kbps;

Nodes {
    Master:CEM,5 ms, 0.1 ms; // Master-Steuergerät, Scheduling-Periode 5ms
    Slaves:LSM,CPM;           // Slave-Steuergeräte
}
Signals {                      // Definition von Signalen
    CPMOutputs:10,0,CPM,CEM;
    HeaterStatus:4,0,CPM,CEM;
    CPMGlowPlug:7,0,CPM,CEM;
    WaterTempLow:8,0,CPM,CEM,LSM;
    WaterTempHigh:8,0,CPM,CEM,LSM;
    CPMFuelPump:7,0,CPM,CEM;
    .
}
Signal_encoding_types {
    Temp {
        physical_value,0,250,0.5,-40,"degree";
        physical_value,251,253,1,0,"undefined";
        logical_value,254,"out of range";
        logical_value,255,"error";
    }
    .
}
Signal_representations {
    Temp:WaterTempLow,WaterTempHigh;
    .
}
Frames {                      // Definition von Botschaften
    VL1_CEM_Frm1:32,CEM,3 { // --- Botschaft 1
        .
    }
    VL1_CPM_Frm1:50,CPM { // --- Botschaft 2
        CPMOutputs,0;
        HeaterStatus,10;
        WaterTempLow,32;
        WaterTempHigh,40;
        CPMFuelPump,56;
    }
    .
}
Schedule_tables {             // Botschaftstabelle
    VL1_ST1 {
        VL1_CEM_Frm1 delay 15 ms; // --- Botschaft 1 alle 15ms
        VL1_CPM_Frm1 delay 20 ms; // --- Botschaft 2 alle 20ms
        .
    }
}

```

Abb. 3.15 Schnittstellen der LIN-Protokoll-Software (LIN Protocol Stack)



(siehe Abschn. 3.2.6) und einige weitere Parameter angegeben, z. B. welche Botschaften dynamisch konfigurierbare LIN-Identifier haben.

- **Signals{...}:** Definiert symbolische Namen für alle zu übertragenen Datenwerte, deren Größe (1...16 bit bzw. 1 bis 8 Byte) sowie einen Initialisierungswert, der verwendet wird, wenn noch kein aktueller Wert von der Applikation vorgegeben bzw. vom Bus empfangen wurde. Außerdem wird für jedes Signal angegeben, welches Steuergerät (*Node*) das Signal senden (*publish*) kann und welche Steuergeräte das Signal empfangen (*subscribe*) wollen. Um kurze Botschaften zu erhalten, können mehrere Signale, die kürzer sind als 16 bit, bei der Übertragung zu einem oder zwei Datenbytes zusammengefasst werden. Zusammengehörige Datenstrukturen, die mehr als 1 Byte benötigen, können zu einem *Byte Array* zusammengefasst werden (in V1.3 noch als *Signal Group* bezeichnet).
- **Signal_encoding_types{...}:** Optional. Definiert den Wertebereich der übertragenen Hexadezimalwerte sowie die Umrechnung in den echten physikalischen Wert nach dem Umrechnungsschema

$$\text{Physikal. Wert} = \text{Skalierungsfaktor} \cdot \text{Hexadezimalwert} + \text{Offset}.$$

- Für jede Umrechnungsformel wird ein symbolischer Name definiert. Im Abschnitt **Signal_representations{...}** kann einem Signal dann eine Umrechnungsformel zugeordnet werden.
- **Frame{...}:** Definiert für jede Botschaft einen symbolischen Namen, den zugehörigen Identifier, das sendende Steuergerät und legt fest, aus welchen Signalen die Botschaft aufgebaut ist. Für jedes Signal wird die Position innerhalb der Botschaft (über einen Bitoffset beginnend ab dem ersten Datenbit der Botschaft) festgelegt. Signale mit weniger als 8 bit können innerhalb eines Bytes zusammengefasst werden.
- **Schedule_tables{...}:** Definiert eine Tabelle mit allen Botschaften (*Frames*), die übertragen werden sollen, samt den zugehörigen Wiederholperioden. Aus dieser Tabelle entnimmt der Master, wann welche Header-Botschaft zu senden ist. Bei der Erzeugung der Konfigurationsdateien muss durch den Anwender bzw. das Konfigurationswerkzeug sichergestellt werden, dass die Wiederholperioden der einzelnen Bot-

schaften nur ganzzahlige Vielfache der Scheduling-Periodendauer sind und dass die Wiederholperioden so groß gewählt werden, dass die Übertragung aller in der Botschaftstabelle definierten Botschaften unter Berücksichtigung der Übertragungsdauer und des Jitters der Scheduling-Periode in jedem Fall tatsächlich möglich ist. Mit LIN V2.0 wurden einige zusätzliche Attribute für diesen Konfigurationsabschnitt eingeführt. Üblich sind mehrere Schedule-Tabellen für unterschiedliche Betriebszustände des Systems.

Die folgenden Abschnitte sind erst seit LIN V2.0 vorhanden:

- Unter `Diagnostic_frames{...}` können vollständige Diagnosebotschaften (LIN-Identifier 3Ch und 3Dh) und mit `Diagnostic_signals{...}` die darin enthaltenen Datenwerte definiert werden.
- Unter `Sporadic_Frames{...}` und `Event_triggered_frames{...}` werden die in Abschn. 3.2.6 beschriebenen *dynamischen Botschaften* konfiguriert.

Node Capability Files, die die Fähigkeiten einzelner Geräte beschreiben, verwenden nahezu dieselbe Syntax, wobei sich aber nur ein Teil der o. g. Angaben findet und bei einigen Daten, z. B. den Wiederholperioden einer Botschaft, der zulässige Wertebereich statt eines einzelnen Wertes angegeben werden kann.

Bei LIN V2.1 wurden einige Details verändert, um die neu hinzugekommenen Botschaften spezifizieren zu können. Alternativ zu LDF/NCF kann auch das neuere FIBEX-Konfigurationsformat (siehe Abschn. 6.3) eingesetzt werden, das sich neben LIN auch für CAN, FlexRay und andere Busse eignet.

3.2.6 Dynamische Konfiguration von LIN-Slave-Steuergeräten

Mit LIN V2.0 wurde ein Mechanismus neu eingeführt, um LIN Slave-Steuergeräte dynamisch zu konfigurieren. Dahinter steckt der Gedanke, dass einfache Steuergeräte vom Hersteller mit einer festen Grundkonfiguration ausgeliefert und dann erst im eingebauten Zustand durch das Master-Steuergerät systemspezifisch konfiguriert werden. Üblich ist dies z. B. bei Sitz- oder Lüftersteuergeräten. Damit entfällt die Lagerhaltung unterschiedlicher Varianten von Slave-Steuergeräten. In Anlehnung an die PC-Welt wird die dynamische Konfiguration auch als *LIN Plug-and-Play* bezeichnet.

In der Grundkonfiguration sind für die Steuergeräte-Diagnoseadresse NAD und alle Botschaften vordefinierte LIN Identifier festgelegt, die dann während des Konfigurationsprozesses durch das Master-Steuergerät umkonfiguriert werden. Im Hinblick auf die schnelle Betriebsfähigkeit des Bussystems nach Einschalten der Spannungsversorgung wird man den Konfigurationsvorgang möglichst nur in der Fahrzeugfertigung oder beim Gerätetausch in der Werkstatt durchführen, so dass LIN V2.0-Steuergeräte sinnvollerweise einen Flash-ROM- oder EEPROM-Speicher für die dynamischen Konfigurationsdaten besitzen sollten.

Das Steuergerät enthält eine feste Gerätekennzeichnung, die *LIN Product Identification*, die in einem 16 bit Wert, der vom LIN-Konsortium vergeben wird, eindeutig den Hersteller des Gerätes und in einem vom Hersteller festgelegten 16 bit-Wert die Funktion und in einem weiteren 8 bit-Wert eine Versionsnummer des Steuergerätes angibt.

Für die Konfiguration wird das bereits für die Diagnosebotschaften definierte Botschaftsformat nach Abb. 3.13 mit den LIN-Identifiern 3Ch für die Botschaften vom Master-Steuergerät und 3Dh für die Antworten des Slave-Steuergerätes verwendet. Als Service Identifier SID werden die bei der ISO-Diagnose als herstellerspezifische deklarierten Werte SID=B0h ... B4h eingesetzt, ab LIN V2.1 zusätzlich B5h ... B7h. Diese Botschaften werden, wie in Abschn. 3.2.4 dargestellt, als Broadcast-Botschaften versendet. Die Auswahl des angesprochenen Steuergerätes erfolgt über das NAD ISO Adressbyte (1. Datenbyte der Botschaft). Zulässige Werte sind 1 ... 126. Mit LIN V2.1 wird NAD = 7Eh = 126 für die funktionale Adressierung von LIN-Steuergeräten bei der Diagnose nach ISO reserviert (siehe Abschn. 5.1.3).

Mit Hilfe einer Diagnosebotschaft mit SID = B1h (*Assign Frame Identifier*) kann das Master-Steuergerät einem im Slave-Steuergerät vordefinierten LIN-Botschafts-Identifier einen neuen Wert zuweisen. Neben dem alten und dem neuen Identifier muss der Master dabei in dieser Botschaft auch noch die Herstellerkennung mitsenden, um die Wahrscheinlichkeit zu erhöhen, dass keine Fehlkonfiguration erfolgt. Ab LIN V2.1 soll diese Botschaft durch eine neue Botschaft mit SID = B7h (*Assign Frame Identifier Range*) ersetzt werden, die inhaltlich dasselbe leistet, aber mehrere Identifier gleichzeitig festlegen kann.

Zuvor oder anschließend zur Überprüfung kann der Master mit Hilfe einer Diagnosebotschaft mit SID = B2h (*Read by Identifier*) die aktuelle Konfiguration auslesen. Neben dem Auswahlbyte, das den zu lesenden Wert auswählt, muss der Master dabei aus Sicherheitsgründen die Herstellerkennung und die Funktionskennung mitsenden. Sendet er als Auswahlbyte den Wert 0, erhält er als Antwort die vollständige *LIN Product Identification*, beim Wert 1 die Seriennummer des Geräteherstellers und mit allen anderen Werten im Bereich 16 ... 63 die LIN-Botschaftskennungen der im Gerät vorkonfigurierten Botschaften. Mit LIN V2.1 wurde der Bereich auf 32 ... 63 eingeschränkt.

Optional ist die Möglichkeit, mit SID = B0h (*Assign NAD*) bzw. SID = B3h (*Conditional Change NAD*) die Steuergeräte-Diagnoseadresse neu festzulegen. Dies ist notwendig, wenn in der Default-Konfiguration mehrere Steuergeräte dieselbe NAD verwenden. Neben der neuen Adresse wird aus Sicherheitsgründen wieder die Hersteller- und Funktionskennung mitgesendet und vom Slave-Steuergerät geprüft, bevor der neue Wert akzeptiert wird. Dieser Mechanismus versagt allerdings, wenn mehrere exakt baugleiche Geräte desselben Herstellers und derselben Default-NAD-Adresse verbaut werden, da diese dann alle auf die Diagnosebotschaft antworten und es zu Kollisionen kommt. Die LIN V2.0-Spezifikation schlägt in diesem Fall vor, dass das Slave-Steuergerät seinen Default-NAD-Wert selbstständig z. B. auf Basis einer eventuell vorhandenen eindeutigen Seriennummer verändert, ohne dies zu konkretisieren. In diesem Fall muss das Master-Steuergerät bei der Konfiguration verschiedene NAD-Adressen „durchprobieren“, bis es von den Slave-Steuergeräten korrekte Antworten ohne Kollisionen erhält. LIN V2.1 verweist hierzu auf ein separates Dokument

für ein noch zu definierendes *Slave Node Position Detection* (SNPD) Verfahren und definiert dafür den neuen Dienst SID = B5h (*Assign NAD by SNPD*).

Ebenfalls neu ist in LIN V2.1 die optionale *Save Configuration* Botschaft (SID = B6h), mit der ein Slave-Steuergerät aufgefordert wird, die dynamisch erzeugte Konfiguration dauerhaft abzuspeichern.

Für den Gerätehersteller ist vorgesehen, mit SID = B4h (*Data Dump*) herstellerspezifische Daten mit dem Steuergerät auszutauschen. Dieser Mechanismus ist für die Fertigung und den Test beim Gerätehersteller gedacht und soll nicht im Systemverbund im Fahrzeug verwendet werden.

3.2.7 LIN Application Programming Interface (API)

Für die Entwickler von LIN-Protokoll-Software enthält die LIN-Spezifikation einen Vorschlag für eine einheitliche Programmierschnittstelle (API), was insbesondere dann sinnvoll ist, wenn die LIN-Protokoll-Stack-Software durch einen automatischen Codegenerator erzeugt oder von einem Softwarelieferanten zugeliefert wird (Abb. 3.15).

Das API enthält folgende wesentlichen Funktionsgruppen:

- **Initialisieren der LIN-Protokoll-Software** (`l_sys_init()`), Konfigurieren (Setzen des Baudatenregisters usw. im UART mit `l_ifc_init()` bzw. `l_ifc_ioctl()`). Bei LIN 1.3 und 2.0 gab es noch Funktionen zum expliziten Aktivieren und Deaktivieren der LIN-Schnittstelle (Starten und Beenden des Empfangens und Sendens von Botschaften mit `l_ifc_connect()` und `l_ifc_disconnect()`). Mit `l_ifc_goto_sleep()` kann die Applikation im Master-Steuergerät das Aussenden der Sleep-Botschaft veranlassen, die die LIN-Aktivitäten aller Steuergeräte abschaltet, bis eine Applikation in einem der Steuergeräte mit `l_ifc_wake_up()` den Bus wieder *aufweckt*.
- **Lesen und Schreiben** (`l..._rd()` bzw. `l..._wr()`) von 1 bit, 8 bit oder 16 bit-Daten bzw. von Byte Arrays (im LIN-Sprachgebrauch: Signale). Beim Lesen wird der jeweils letzte empfangene Wert gelesen, beim Schreiben wird der neue Wert in eine Signal-Tabelle eingetragen. Das eigentliche Senden erfolgt erst, wenn das Steuergerät eine Header-Botschaft mit dem entsprechenden Identifier erhält und das Gerät mit der Daten-Antwortbotschaft reagiert.
- Der **innere Zustand der LIN-Protokoll-Software** wird in verschiedenen implementierungsabhängigen Statusworten, so genannten Flags, gespeichert, die von der Anwendungssoftware gelesen und gelöscht werden können (`l_flg_tst()`, `l_flg_clr()`). Damit kann z. B. abgefragt werden, ob ein bestimmtes Signal empfangen wurde. Mit LIN V2.0 wurde der `l_ifc_read_status()` Aufruf eingeführt, mit dem die Anwendung abfragen kann, welcher Identifier als letztes empfangen wurde und ob Sende- bzw. Empfangsfehler aufgetreten sind.

- Als **Interface zum UART-Treiber** dienen `l_ifc_rx()` und `l_ifc_tx()`. Diese Funktionen werden aufgerufen, wenn ein Zeichen empfangen oder gesendet wurde. Falls der UART bzw. sein Softwaretreiber selbstständig in der Lage ist, den *Sync Break* zu erkennen, wird die Funktion `l_ifc_aux()` aufgerufen.
- Manche LIN-Protokollstapel müssen für interne Verwaltungsoperationen kurzzeitig die **Interrupts des Steuergerätes sperren** können. Dazu muss die Anwendungssoftware die Funktionen `l_sys_irq_disable()` und `l_sys_irq_restore()` zur Verfügung stellen.
- Für die **Master-Funktion** sind `l_sch_tick()` und `l_sch_set()` vorgesehen. Mit `l_sch_set()` wird eine Botschaftstabelle (Schedule Table) aktiviert, d. h. eine Tabelle, die die Liste der periodisch zu übertragenden Botschaften enthält. Der Scheduler arbeitet in einem festen Zeitraster, der Scheduler-Periode. Der Aufruf von `l_sch_tick()` zeigt der Protokollsoftware die nächste Periode an.
- Für die **Transportschicht** und die **ISO-Diagnose über LIN** schlägt die LIN-Spezifikation V2.0 zwei verschiedene APIs vor. Das Master-Steuergerät, das als Gateway zwischen dem CAN-Bus und dem LIN-Bus fungiert, muss im Wesentlichen das Kopieren der Botschaften durchführen, ohne deren Inhalt zu interpretieren. Die dafür vorgesehenen *Raw-API*-Funktionen `ld_put_raw()` und `ld_get_raw()` tragen eine ISO-Diagnosebotschaft in den LIN-Botschaftsspeicher ein bzw. lesen sie aus. Da sich die Übertragungsgeschwindigkeiten von CAN und LIN deutlich unterscheiden, sind im CAN-LIN-Gateway üblicherweise Puffer, z. B. FIFOs, vorhanden. Mit `ld_raw_tx_status()` bzw. `ld_raw_rx_status()` kann der Zustand der Puffer abgefragt werden. Für LIN-Slave-Steuergeräte sind die *Cooked-API*-Funktionen sinnvoller. Hierbei werden mit `ld_receive_message()` und `ld_send_message()` vollständige ISO-Botschaften (ab dem *Diagnostic Service Identifier* SID mit bis zu 4095 Bytes) gelesen und geschrieben. Das Segmentieren und Desegmentieren findet innerhalb der LIN-Protokollsoftware statt. Die Funktionen arbeiten asynchron, d. h. sie kehren sofort zurück, bevor die Daten tatsächlich empfangen bzw. gesendet wurden. Mit `ld_tx_status()` und `ld_rx_status()` kann die Anwendung herausfinden, ob der Vorgang erfolgreich abgeschlossen wurde.
- Für die mit LIN V2.0 neu eingeführte **dynamische Konfiguration** sind für das Master-Steuergerät die Funktionen `ld_assign_frame_id()`, `ld_read_by_id()`, `ld_assign_NAD()` und `ld_conditional_assign_NAD()` vorgesehen, die die in Abschn. 3.2.6 beschriebenen Botschaften erzeugen. Der Erfolg bzw. Misserfolg einer Konfigurationsbotschaft kann mit `ld_check_response()` und `ld_is_ready()` abgefragt werden. Mit `ld_save_configuration()` fordert der Master ein oder mehrere Slave-Steuergeräte auf, die Konfiguration zu übernehmen, die dann dort mit `ld_set_configuration()` gespeichert bzw. mit `ld_read_configuration()` wieder gelesen werden kann.

Tab. 3.9 Botschaftslängen für $f_{\text{Bit}} = 1 / T_{\text{bit}} = 19,2 \text{ kbit/s}$

n_{Data}	$T_{\text{Header, min}}$	$T_{\text{Response, min}}$	$T_{\text{Frame, min}}$	$T_{\text{Frame, max}}$
1 byte	1,8 ms	1,0 ms	2,8 ms	3,9 ms
8 byte		4,7 ms	6,5 ms	9,0 ms

3.2.8 Zeitverhalten von LIN-Systemen

Die Länge einer LIN Botschaft (Tab. 3.9) ist

$$T_{\text{Frame}} = T_{\text{Header}} + T_{\text{Response}} \quad (3.12)$$

mit $T_{\text{Header,min}} = 34 \text{ bit} \cdot T_{\text{bit}}$ und $T_{\text{Response,min}} = \frac{10}{8}(n_{\text{Data}} + 8 \text{ bit}) \cdot T_{\text{bit}}$.

Dabei ist $n_{\text{Data}} = 0, \dots, 64$ bit die Anzahl der Datenbits, die in Stufen von 8 bit variabel ist. Der Faktor 10 / 8 berücksichtigt, dass UART-üblich je Byte zusätzlich ein Start- und ein Stopbit eingefügt werden. Die minimale Länge ergibt sich, wenn das *Sync Break Feld* die kleinste zulässige Länge (13 bit) hat und keine Pausen zwischen den einzelnen Bytes bzw. zwischen Header und Response auftreten. Als maximal zulässige Länge gibt die LIN-Spezifikation vor

$$T_{\text{Frame,max}} = 1,4 \cdot (T_{\text{Header,min}} + T_{\text{Response,min}}). \quad (3.13)$$

Um die Implementierung der LIN-Treibersoftware zu vereinfachen, werden LIN-Botschaften üblicherweise zeitsynchron gesendet, wobei das Master-Steuergerät durch das Senden der Header-Botschaften das Zeitraster (*Frame Slots*) vorgibt und über den Identifier im Header bestimmt, welche Botschaft in diesem Zeitschlitz gesendet wird. Die Reihenfolge und Wiederholperiode aller Botschaften wird statisch während der Entwicklungsphase des Netzes festgelegt und im Master-Steuergerät in einer so genannten Botschaftstabelle (*Scheduling Table*) abgelegt, die von der Master-Funktion abgearbeitet wird (*Message Scheduling*). Dabei können mehrere solche Botschaftstabellen vorhanden sein, zwischen denen die Applikationssoftware im Master-Steuergerät umschalten kann. Bei der Konfiguration der Botschaftstabellen muss der Entwickler, in der Regel mit Hilfe eines Konfigurationswerkzeuges, sicherstellen, dass alle Botschaften mit den festgelegten Wiederholperioden stets übertragen werden können. Dadurch ist das Übertragungsverhalten des LIN-Busses deterministisch.

Die Dauer eines Zeitschlitzes T_{Slot} im Zeitraster (*Scheduling Periode*) wird so gewählt, dass sie unter Berücksichtigung aller Toleranzen (*Scheduling Jitter*) stets größer ist als die Übertragungsdauer T_{Frame} der Botschaft, d. h.

$$T_{\text{Slot}} > T_{\text{Frame,max}}. \quad (3.14)$$

Für ein LIN-Bussystem mit einer Bitrate von 19,2 kbit/s etwa wäre $T_{\text{Slot}} = 10 \text{ ms}$ sinnvoll. Nach jeweils N Zeitschlitzten (*Slots*) wiederholt sich die Botschaftssequenz (Abb. 3.16), so

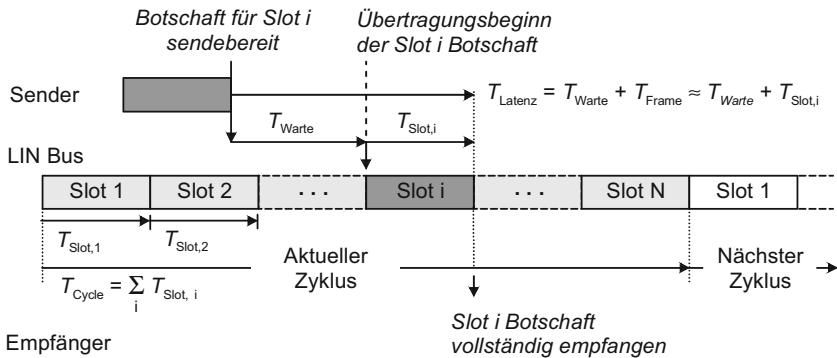


Abb. 3.16 LIN Botschaftszyklus

dass die Zykluszeit

$$T_{\text{Zyklus}} = \sum_{i=1}^N T_{\text{Slot},i} \quad (3.15)$$

ist. Bezogen auf eine einzelne Botschaft sind auch höhere Wiederholraten möglich, indem für eine Botschaft mehrere Zeitschlitzte innerhalb eines Zyklus eingeplant werden. Die Latenzzeit bei der Übertragung setzt sich wie bei CAN aus der Zeit T_{Warte} bis zum Sendebeginn und der eigentlichen Übertragungsdauer T_{Frame} zusammen:

$$T_{\text{Latenz}} = T_{\text{Warte}} + T_{\text{Frame}}. \quad (3.16)$$

Im günstigsten Fall wird die Botschaft auf der Sendeseite direkt vor Beginn des zugeordneten Zeitschlitzes zum Senden bereitgestellt, so dass die Wartezeit entfällt. Im ungünstigsten Fall stehen die Sendedaten erst nach Beginn des Zeitschlitzes zur Verfügung und können daher erst einen vollen Zyklus später gesendet werden:

$$T_{\text{Latenz,min}} = T_{\text{Frame}} < T_{\text{Latenz}} < T_{\text{Latenz,max}} = T_{\text{Zyklus}} + T_{\text{Frame}}. \quad (3.17)$$

Um den Jitter möglichst gering zu halten, sollte die Anwendungssoftware, die die Daten bereitstellt, daher mit dem Zyklus des LIN Bussystems synchronisiert werden.

3.2.9 Zusammenfassung LIN – Layer 1 und 2

- Zeichenorientiertes UART-basiertes Übertragungsprotokoll mit bidirektonaler Ein-Draht-Leitung (wie K-Line), Bitrate zwischen 1 ... 20 kbit/s (üblich 19,2 kbit/s), Signalpegel U_B (Batteriespannung), entwickelt mit der Zielsetzung eines Low-Cost-Bussystems für einfache Sensor-Aktor-Anwendungen, maximale Nutzdatenrate unter 1,2 KB/s (bei 20 kbit/s).

- Anzahl der Busteilnehmer aus elektrischen Gründen und wegen der begrenzten Anzahl von Botschafts-Identifern typ. < 16, relative kurze Buslängen < 40 m.
- Ein Steuergerät als *Master* sendet periodisch Botschafts-Header, eines der Geräte (*Slaves*) antwortet mit max. 8 Datenbytes und einem Prüfsummenbyte.
- Senden aller Botschaften periodisch in festem Zeitraster. Senderaster im Master als Botschaftstabelle (Schedule Table) statisch konfiguriert, unterschiedliche Botschaftstabellen für verschiedene Betriebszustände möglich. Typ. Sendezeitraster 10 ms bei einer max. Botschaftsdauer von ca. 9 ms (Botschaft mit 8 Byte Nutzdaten und Bittakt 20 kbit/s), intern arbeitet der Master dann z. B. mit 5 ms.
- Inhalt der Botschaft wird durch *Identifier* (im vom Master gesendeten Header) ausgewählt (inhaltsbezogene Adressierung wie bei CAN). 60 verschiedene Botschaften möglich. Je zwei Identifier sind für spezielle Botschaften und für zukünftige Protokollerweiterungen vorgesehen.
- Anforderungen an die Bittakt-Genauigkeit der Slaves und gesamtes Protokoll-Timing gering, d. h. Slaves können ohne eigenen Quarz implementiert werden.
- Primitive Fehlerüberwachung (Rücklesen der Bussignale, Prüfsumme, Antwortzeitüberwachung), keine Fehlerkorrektur. Fehlerbehandlung auf Anwendungsebene hersteller- und anwendungsabhängig.
- Master-Steuergerät häufig als Gateway zu anderen Bus-Systemen, z. B. CAN.
- Mit LIN V2.x optionale Botschaften (*Transportprotokoll*), mit denen größere Datenblöcke oder ISO-Diagnosebotschaften über LIN versendet werden können.

3.3 FlexRay

FlexRay wurde vor dem Hintergrund zukünftiger X-by-Wire-Anwendungen (X = Brake, Steer, ...) entwickelt [11–13] und ist heute als ISO 17458 standardisiert. Obwohl dieses Anforderungsprofil in weiten Teilen dem von CAN gleicht, erschien vielen Fachleuten eine Neuentwicklung notwendig und sinnvoll. Haupttriebfeder war zunächst, dass die bei CAN sinnvoll erreichbare Bit- und damit Datenrate aufgrund des CAN-Grundprinzips der bitweisen Arbitrierung auf maximal 1 Mbit/s und die Bus-Topologie auf ein reines Liniennbussystem mit (bei hohen Bitraten) kurzen Stichleitungen beschränkt ist. Dies zwingt teilweise zu einer aus fertigungstechnischer Sicht suboptimalen Kabelführung im Fahrzeug. Dazu kommt, dass CAN zwar eine hohe Übertragungssicherheit gewährleistet, aber Schwächen bei extremen Sicherheitsanforderungen hat. Schwer wiegt dabei, dass CAN zunächst ein rein einkanaliges System ist, das bei Ausfall der Busverbindung versagt. Zweikanalige Aufbauten sind natürlich möglich, aber Mechanismen zur Synchronisation und Plausibilitätsprüfung zwischen den Kanälen fehlen und müssen daher mühsam softwaremäßig nachgebildet werden. Zum anderen ist der asynchrone Buszugriff bei CAN nicht streng deterministisch. Dies führt dazu, dass nur für die höchspriore Botschaft eine maximale Latenzzeit bei der Übertragung garantiert werden kann. Für weniger hoch priorisierte Botschaften lässt sich unter gewissen Voraussetzungen (ähnlich wie beim Task-

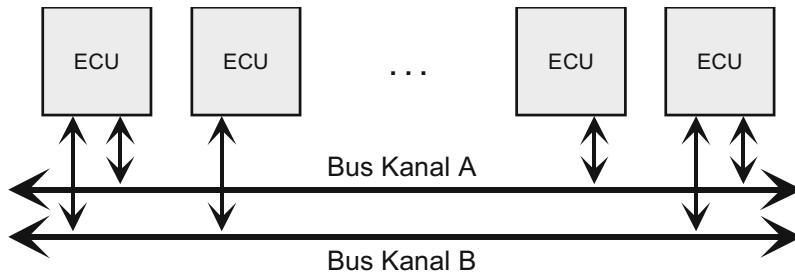


Abb. 3.17 Beispiel eines FlexRay-Busses in Zwei-Kanal-Linien-Struktur

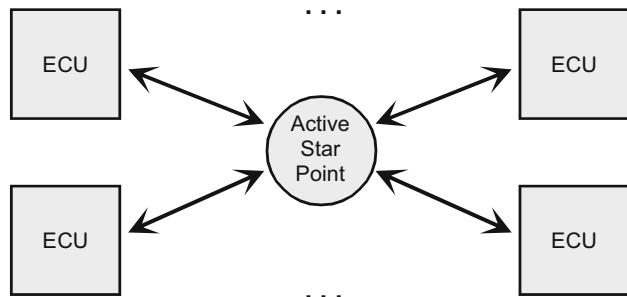
Scheduling in Echtzeitbetriebssystemen) trotzdem eine garantierte Maximallatenz bestimmen, doch ergeben sich dabei unter absoluten Worst Case Bedingungen oft unzulässig große Werte und die Einhaltung der Voraussetzungen ist insbesondere bei hoher Busauslastung nur schwer zu überprüfen und zu gewährleisten. Diese vor allem in der akademischen Welt diskutierte Problematik hat bei CAN zu der aufwärts kompatiblen Erweiterung Time Triggered CAN (TTCAN) geführt (siehe Abschn. 3.1.7). Das Problem der begrenzten Bit- und Datenrate dagegen konnte durch TTCAN nicht gelöst werden. Verschiedene Ansätze aus der akademischen Welt, z. B. der Time Triggered Protocol/Architecture Bus TTP/TTA, und Lösungen für spezielle Anwendungsfelder wie Byteflight, ließen eine Reihe von Konzepten entstehen, die jeweils von verschiedenen Gruppen von Fahrzeugherstellern und Zuliefern verfolgt wurden. Schließlich einigten sich die in Deutschland führenden Hersteller und Zulieferer darauf, die besten Grundideen aus diesen Ansätzen zusammenzubringen und mit FlexRay eine gemeinsame Neuentwicklung durchzuführen, die als offener, gemeinsamer Standard ähnlich wie ehedem CAN zu einer schnellen Marktdurchdringung und damit zu rasch sinkenden Kosten führen sollte. Der Spezifikationsprozess im FlexRay-Konsoritum und dann bei ISO sowie die Entwicklung entsprechender Kommunikationscontroller-ASICs nahm allerdings viel Zeit in Anspruch, so dass die Einführung von FlexRay zunächst verhalten erfolgte. Mittlerweile sind aber eine Reihe von FlexRay-Fahrzeugen im Markt, wobei FlexRay dort CAN eher ergänzt als ersetzt.

3.3.1 Bus-Topologie und Physical Layer

FlexRay erlaubt ein- und zweikanalige Systeme sowohl in Linien- als auch in Stern-Struktur (Abb. 3.17 und 3.18), wobei alle derzeit verfügbaren Kommunikationscontroller zweikanalig ausgeführt sind. Der maximale Abstand zwischen den am weitesten auseinander liegenden Busteilnehmern beim Linien-Bus und in der Stern-Struktur mit passivem Sternpunkt liegt bei 24 m.

Sterne mit passivem Sternpunkt haben möglicherweise schlechte elektrische Eigenschaften und sind daher nur bei kurzen Verbindungen oder niedrigen Bitraten sinnvoll.

Abb. 3.18 Beispiel eines FlexRay-Busses in Ein-Kanal-Stern-Struktur



Üblicherweise sollte der Sternpunkt deshalb mit einem aktiven Sternkoppler ausgeführt werden. Dieser wirkt als bidirektonaler Transceiver und Repeater, der die einzelnen Verbindungen elektrisch entkoppelt, aber empfangene Botschaften an alle angeschlossenen Steuergeräte weiterverteilt. D. h. logisch betrachtet arbeitet auch die Sternstruktur wie ein Bus. Mit aktivem Sternkoppler darf jedes Steuergerät bis zu 24 m vom Sternpunkt entfernt sein. Durch die Zusammenschaltung von zwei Sternpunkten, deren Direktverbindung ebenfalls max. 24 m lang sein darf, lässt sich die Ausdehnung zwischen den am weitesten entfernten Steuergeräten auf maximal 72 m steigern. Bei aktivem Sternpunkt müssen die Leitungen an beiden Enden mit Abschlusswiderständen versehen sein.

Die maximale zulässige Bitrate bei FlexRay ist gegenwärtig 10 Mbit/s. Dies wird sich bei ausgedehnten Netzwerken aber wohl nur in der Struktur mit aktivem Sternpunkt erreichen lassen. In der Linien-Struktur und bei passivem Sternpunkt wird die Leitungslänge und Teilnehmerzahl deutlich kritischer sein. Seit Protokollversion 3 sind auch Bitraten von 2,5 und 5 Mbit/s spezifiziert, die die Implementierung einfacher Linienbusse erleichtern.

Wenn das System zweikanalig, d. h. mit zwei parallelen Bussen, ausgeführt wird, kann jedes Steuergerät entweder an beide Kanäle angeschlossen werden oder auch nur an einen der beiden Kanäle (Abb. 3.17). Eine Kommunikation ist allerdings nur für die Steuergeräte möglich, die am selben Kanal angeschlossen sind, d. h. ein Steuergerät, das nur am Kanal A angeschlossen ist, kann nicht mit einem Steuergerät kommunizieren, das nur am Kanal B angeschlossen ist. Der zweite Kanal kann bei sicherheitskritischen Botschaften sowohl redundant verwendet werden, indem ein Steuergerät dieselbe Botschaft gleichzeitig auf beiden Kanälen sendet, als auch zur Bandbreitenerhöhung, indem auf beiden Kanälen unterschiedliche Botschaften versendet werden.

Theoretisch ist bei zweikanaligen Systemen auch eine gemischte (hybride) Topologie möglich, bei der z. B. Kanal A als Stern und Kanal B als Linienbus aufgebaut wird.

Die Leitungsverbindungen werden als geschirmte und/oder verdrillte Zwei-Draht-Leitungen mit einem Wellenwiderstand zwischen 80 und 110 Ω ausgeführt. An den Leitungsenden eines Linien-Bussystems bzw. an den Enden der Verbindungen zwischen Steuergerät und aktivem Sternpunkt sind wie bei High-Speed-CAN entsprechende Abschlusswiderstände notwendig. Bei passiven Stern-Strukturen sollten die Abschluss-

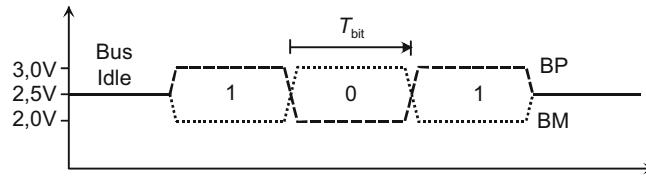


Abb. 3.19 Signalpegel auf den FlexRay-Leitungen BP und BM

widerstände an den am weitesten voneinander entfernten Geräten angeordnet werden. Im Gegensatz zu CAN werden sowohl 0- als auch 1-Bits mit niederohmigen Differenzspannungssignalen (dominant) übertragen (Abb. 3.19). Im Ruhezustand liegen beide Busleitungen hochohmig auf ungefähr 2,5 V.

3.3.2 Data Link Layer

FlexRay verwendet für den Buszugriff ein ähnliches Verfahren wie TTCAN, um Kollisionen zu vermeiden. Der sich periodisch wiederholende Kommunikationszyklus wird in einen statischen und einen optionalen dynamischen Abschnitt eingeteilt, bei FlexRay statisches bzw. dynamisches Segment genannt (Abb. 3.20). Nach dem dynamischen Abschnitt folgt optional ein kurzes Zeitfenster, das so genannte *Symbol Window*, danach bleibt das Netz in Ruhe (*Network Idle Time NIT*), bevor der nächste Kommunikationszyklus beginnt. Jeder Busteilnehmer zählt die Anzahl der Kommunikationszyklen in einem Zykluszähler (*Cycle Counter*) beginnend bei 0 mit (siehe Abschn. 3.3.3). Alle Zeitabschnitte sind ganzzählige Vielfache eines für das gesamte Netzwerk und beide Kanäle identischen virtuellen Zeitrasters, den so genannten *Makroticks*, deren Dauer zwischen 1 und 6 μ s betragen soll. Das statische Segment ist für die Übertragung periodischer Botschaften vorgesehen, das dynamische Segment vorzugsweise für die ereignisgesteuerte Übertragung.

Das *statische Segment* besteht aus einer festen Anzahl von Zeitschlitzten (*Slots*), die so lang sind, dass innerhalb eines solchen Zeitschlitztes eine vollständige FlexRay-Botschaft übertragen werden kann. Die Botschaften im statischen Segment und damit die Zeitschlitzte haben alle dieselbe feste Länge und laufen für beide Kanäle synchron. Das Senderecht innerhalb eines Zeitschlitztes ist für jeden Kanal getrennt genau einem einzigen Steuergerät zugeteilt (*Time Division Multiple Access TDMA*), so dass es (bei fehlerfrei arbeitenden Teilnehmern) niemals zu einer Kollision kommt. Zur redundanten Datenübertragung oder zur Bandbreitenerhöhung kann ein Steuergerät im selben Zeitschlitz auf beiden Kanälen gleichzeitig oder während mehrerer Zeitschlitzte desselben Kommunikationszyklus das Senderecht erhalten. Jedes Steuergerät zählt für jeden Kanal getrennt in einem Zeitschlitzt-Zähler (*Slot Counter*) beginnend bei 1 mit dem ersten Zeitschlitz jedes Kommunikationszyklus (*Cycle*) die Zeitschlitzte mit. Der Stand des *Slot Counters* signalisiert im statischen

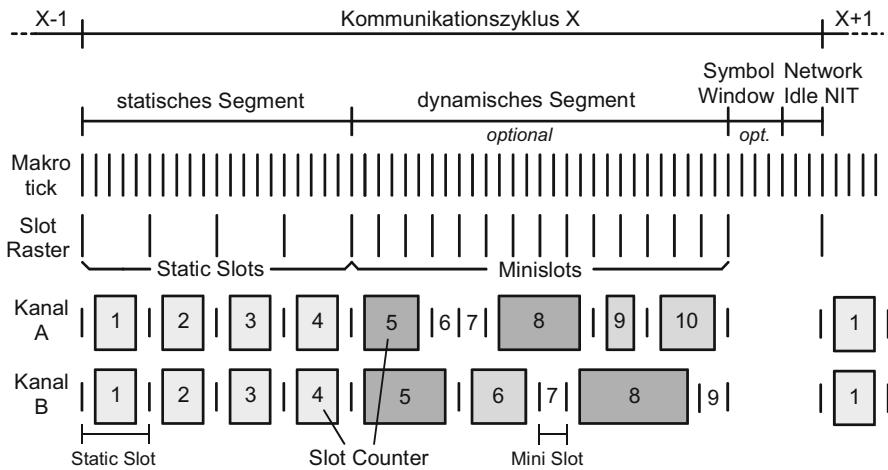


Abb. 3.20 Ablauf der Kommunikation bei FlexRay

Segment daher, welches Gerät aktuell die Sendeberechtigung hat. Das statische Segment muss mindestens 2 und darf maximal 1023 Zeitschlitzte umfassen.

Innerhalb des *dynamischen Segmentes* gibt es ebenfalls Zeitschlitzte, die so genannten *Minislots*, deren Länge jedoch wesentlich kleiner ist als diejenige im statischen Segment. Während eines *Minislots* darf (für beide Kanäle unabhängig voneinander) wiederum genau ein Steuergerät senden. Die gesendete Botschaft darf jetzt aber eine beliebige, auf beiden Kanälen auch unterschiedliche Länge haben, solange ihr Ende die Gesamtlänge des dynamischen Segments nicht überschreitet. Sobald die Botschaft vollständig übertragen ist, wird der *Slot Counter* mit dem nächsten *Minislot* wieder inkrementiert und das Senderecht geht (in der Regel) auf ein anderes Steuergerät über. Hat ein Steuergerät keine Daten zu senden, verzichtet es auf das Senderecht. Der *Slot Counter* wird nach Ende einer gesendeten Botschaft oder sofort, falls keine Botschaft gesendet wurde, mit dem nächsten *Minislot* inkrementiert. Durch die variable Anzahl und Länge von Botschaften verlaufen die Zählerstände der *Slot Counter* während des dynamischen Segments in beiden Kanälen asynchron und können bei jedem Kommunikationszyklus unterschiedliche Werte annehmen (*Flexible Time Division Multiple Access FTDMA*). Innerhalb des dynamischen Segments signalisiert der *Slot Counter* daher nicht nur, welches Steuergerät die Sendeberechtigung hat, sondern indirekt auch die Priorität der zugehörigen Botschaften. Botschaften mit hohen *Slot Counter* Werten, die im dynamischen Teil des aktuellen Kommunikationszyklus nicht übertragen werden können, weil die Botschaften mit niedrigeren *Slot Counter* Werten das Segment bereits ausgeschöpft haben, müssen warten, bis sie in einem der folgenden Kommunikationszyklen übertragen werden können. Die Gesamtzahl der Zeitschlitzte für den statischen und den dynamischen Teil zusammen ist auf maximal 2047 begrenzt. Alle Zeitschlitzte sind ganzzahlige Vielfache der netzweiten, virtuellen *Makroticks*-Taktperiode.

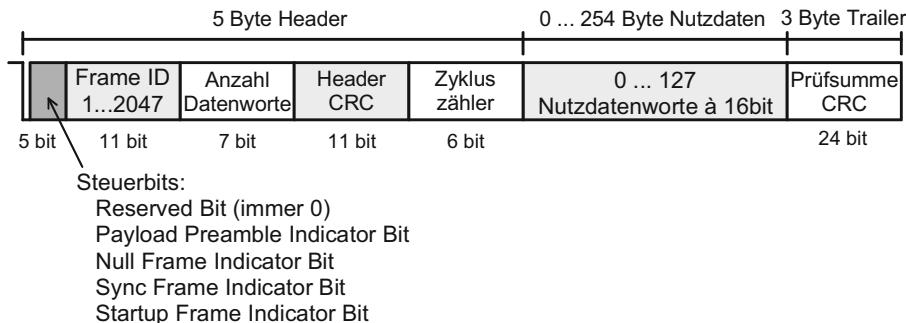


Abb. 3.21 Logisches (Data Link Layer) FlexRay-Botschaftsformat

Das optionale *Symbol Window* kann zur Übertragung des sogenannten *Collision Avoidance CAS* bzw. *Media Access Test MTS* Symbols dienen, einer mindestens 30 bit langen Sequenz von Low Bits, mit denen u. a. der Buswächter (Abschn. 3.3.4) getestet werden soll. Ab Protokollversion 3 sind auch *Wakeup Pattern* im *Symbol Window* vorgesehen. In der *Network Idle Time NIT* erfolgt die Resynchronisation der Taktgeneratoren der Kommunikationscontroller (Abschn. 3.3.3).

Abbildung 3.21 zeigt den Aufbau einer einzelnen FlexRay-Botschaft. Jede Botschaft beginnt mit einem Header, der nach einigen Steuerbits im *Frame ID*-Feld die Nummer des Zeitschlitzes (*Slot*) enthält, in dem die Botschaft gesendet wird, sowie die Nutzdatenlänge. Angegeben wird dabei die Anzahl von 16 bit-Datenworten, obwohl die Daten beliebig in Bytes unterteilt sein können. Es darf aber insgesamt stets nur eine gerade Anzahl von Nutzdatenbytes, also 0, 2, 4, ... usw. gesendet werden. Als letztes Feld enthält der Header die Nummer des aktuellen Kommunikationszyklus, den so genannten Zykluszähler (*Cycle*), der beim Start des Netzes mit 0 initialisiert und mit jedem Kommunikationszyklus inkrementiert wird. Sowohl der Header (ohne den Zykluszähler) als auch die gesamte Botschaft werden jeweils über eine Cyclic-Redundancy-Check-(CRC)-Prüfsumme gegen Übertragungsfehler geschützt.

Über vier der fünf Steuerbits des Headers können Sonderbotschaften angezeigt werden. Das *Payload Preamble Indicator Bit* zeigt im statischen Teil des Kommunikationszyklus an, dass die ersten 0 ... 12 Datenbytes Zusatzinformationen für das Netzwerkmanagement enthalten, im dynamischen Teil, dass die ersten 2 Datenbytes eine Message ID enthalten, d. h. eine Botschaftskennung, die die weiteren Daten der Botschaft kennzeichnet und beim Empfänger ähnlich wie der Message Identifier bei CAN zur Akzeptanzfilterung verwendet werden kann. Mit dem *Null Frame Indicator Bit* kann ein Sender anzeigen, dass die Botschaft keine gültigen Nutzdaten enthält. Dies ist vor allem im statischen Teil des Kommunikationszyklus sinnvoll, wenn ein Sender zum aktuellen Zeitpunkt keine gültigen Nutzdaten hat, wegen einer gegebenenfalls im Empfänger vorhandenen Timeout-Überwachung aber trotzdem in seinem Zeitschlitz eine Botschaft senden will. Im dynamischen Teil des Kommunikationszyklus könnte der Sender in diesem Fall einfach eine Botschaft ohne Nutz-

daten senden, im statischen Teil ist dies nicht möglich, da dort alle Botschaften immer dieselbe Anzahl von Nutzdatenworten haben müssen, unabhängig davon, ob die Daten gültig sind oder nicht. Über das *Startup Frame Indicator Bit* bzw. das *Sync Frame Indicator Bit* kann eine Synchronisation aller Busteilnehmer beim Starten des Netzwerkes bzw. im laufenden Betrieb erfolgen. Details dazu im folgenden Abschn. 3.3.3.

Abbildung 3.21 zeigte den logischen Aufbau einer FlexRay-Botschaft. In der Bitübertragungsschicht werden ähnlich wie bei CAN noch einige zusätzliche Bits in den Datenstrom eingefügt, um die Bitabtastung in den Kommunikationscontrollern zu synchronisieren. Anstelle des bei CAN üblichen, aber bezüglich der Botschaftslänge nicht deterministischen Bit-Stuffings, bei dem abhängig vom Dateninhalt zusätzliche Bits übertragen werden, wird bei FlexRay vor jedem 8 bit-Datenfeld des logischen Botschaftsformats nach Abb. 3.21 zusätzlich eine 1-0-Bitfolge, die so genannte *Byte Start Sequenz BSS*, übertragen, die die effektive Datenrate um 20 % reduziert. Der Beginn der gesamten Botschaft selbst wird durch eine 3 bis 15 bit lange, als *Transmission Start Sequenz TSS* bezeichnete 0-Bitfolge und das *Frame Start Sequenz FSS Bit* eingeleitet. Die Länge der TSS wird so bemessen, dass die Transceiver eines aktiven Sternpunktes ausreichend Zeit für die Umschaltung zwischen Senden und Empfangsrichtung haben. Dabei kann der beim Empfänger ankommende Teil der TSS dynamisch verkürzt werden. Mit einer *Frame End Sequenz FES* 0-1-Bitfolge wird die Botschaft abgeschlossen. Im dynamischen Segment folgt dann noch die *Dynamic Trailing Sequenz DTS*, die aus mindestens einem 0 und einem 1 Bit besteht und den Zeitabschnitt bis zum Beginn des folgenden *Minislots* füllt.

Da die Botschaftslänge im statischen Teil des Kommunikationszyklus für alle Zeitschlitzte gleich sein muss, wird man dort wohl eher kurze Botschaften definieren. Sendet man wie bei CAN Botschaften mit max. 8 Nutzdatenbytes, so ergibt sich bei einer Bitrate von 10 Mbit/s eine maximale Nutzdatenrate für das gesamte Bussystem von max. 500 KB/s. Weil die Zeitschlitzte in der Praxis aber stets etwas größer sein müssen als die Länge einer Botschaft, der Bus im dynamischen Segment wohl kaum vollständig ausgenutzt werden kann und außerdem gegebenenfalls noch Raum für das optionale *Symbol Window* und die *Network Idle Time* bleiben muss, werden die praktisch erreichbaren Werte vermutlich deutlich niedriger liegen. Im Vergleich mit einem 500 kbit/s-CAN-Bus hat ein 10 Mbit/s-FlexRay-Bussystem damit gut die zehnfache Bandbreite.

3.3.3 Netzwerk-Start und Takt-Synchronisation

Ein kritischer Punkt bei jedem zeitsynchronen Bussystem ist die zeitliche Synchronisation der Teilnehmer und der geordnete Start des Netzwerkes. Während bei Systemen mit asynchronem Buszugriff lediglich der Bittakt der einzelnen Teilnehmer synchronisiert werden muss, was bei CAN durch Stuffing-Bits bzw. bei FlexRay durch die *Byte Start Sequenz BSS* erreicht wird, erfordert der TDMA-Buszugriff eine Zeitsynchronisation auf der Ebene der *Makroticks* und der Zeitschlitzte. Aus Zuverlässigkeitssgründen darf dabei nicht nur ein ein-

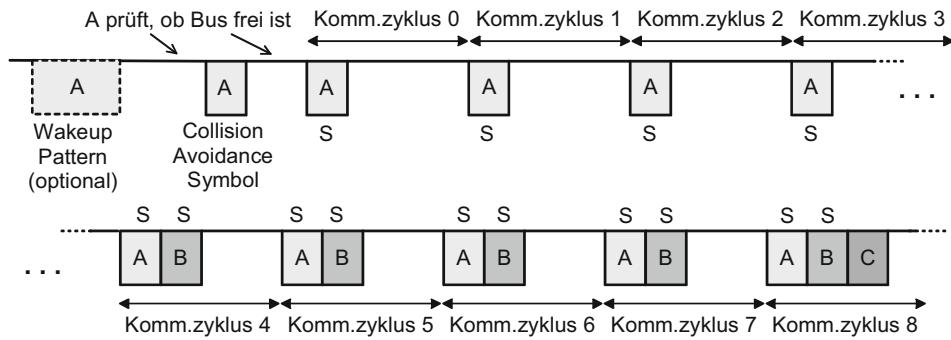


Abb. 3.22 Hochlauf eines FlexRay Netzwerkes (TT-D Prinzip, siehe Abschn. 3.3.8), A: Leading Coldstart Node; B: Following Coldstart Node; C: Normales Steuergerät; S: Startup und Sync Frames

zelles Steuergerät die Funktion eines Zeitmasters übernehmen, sondern es ist eine verteilte Synchronisation notwendig.

Für den Start des Netzes werden bei der Entwicklung des Netzwerks mindestens zwei, idealerweise drei Steuergeräte als Kaltstart-Knoten (*Coldstart Nodes*) festgelegt, die für den Hochlauf verantwortlich sind. Ausgelöst wird der Start entweder durch das Einschalten der Spannungsversorgung oder durch einen beliebigen Busknoten, der auf einem der beiden Bus-Kanäle ein so genanntes *Wakeup Pattern WUP* aus zwei oder mehr *Wakeup Symbolen WUS* sendet, wenn der Bus in Ruhe war.

Ein oder mehrere der Kaltstart-Knoten beginnen daraufhin, auf beiden Bus-Kanälen sogenannte *Collision Avoidance Symbole* CAS zu versenden (Abb. 3.22). Zuvor und danach überprüfen sie, ob der Bus tatsächlich frei ist, so dass im Regelfall nur ein CAS Symbol gesendet wird, weil sich die anderen Kaltstart-Knoten sofort zurückziehen, sobald sie ein fremdes CAS Symbol entdeckt haben. *Wakeup Symbol* WUS und *Collision Avoidance Symbol* CAS sind Bitmuster, die sich von allen übrigen Bitmustern auf dem FlexRay Bus eindeutig unterscheiden und damit allen angeschlossenen Steuergeräten anzeigen, dass der Kaltstart des Netzes mit dem Kommunikationszyklus 0 beginnen wird. Im Gegensatz zum *Wakeup Pattern*, das aus Sicherheitsgründen nur auf einem der beiden Bus-Kanäle versendet werden darf, muss CAS auf beiden Kanälen gleichzeitig ausgesendet werden, da die Kommunikation auf beiden Kanälen synchron initialisiert werden muss. Kaltstart-Knoten müssen also stets an beide Kanäle angeschlossen sein, während das Aufwecken der beiden Bus-Kanäle aus Sicherheitsgründen explizit von zwei verschiedenen Geräten aus erfolgen soll.

Der Hochlauf des Netzes wird fortgesetzt, indem der mit seinem CAS-Symbol erfolgreiche Kaltstart-Knoten, der so genannte *Leading Coldstart Node*, mit dem Kommunikationszyklus 0 beginnt und reguläre Botschaften innerhalb des dafür vorgesehenen Zeitschlitzes aussendet, wobei im Header dieser Botschaften stets das *Startup Frame Indicator Bit* und das *Sync Frame Indicator Bit* (Abb. 3.21) gesetzt sind. Die übrigen Kaltstart-Knoten beginnen, nachdem sie mindestens vier derartige Botschaften empfangen und sich auf das

Zeitraster synchronisiert haben (siehe unten), ebenfalls mit dem Aussenden ihrer Botschaften innerhalb der für sie reservierten Zeitschlitz, wobei auch bei deren Botschaften das *Startup Frame Indicator Bit* und das *Sync Frame Indicator Bit* gesetzt werden. Die übrigen Steuergeräte beginnen, sobald sie mindestens vier aufeinander folgende Botschaften von jeweils zwei unterschiedlichen Kaltstart-Knoten empfangen haben, ebenfalls mit dem Aussenden von Botschaften. Damit ist der Start des Netzes im günstigsten Fall nach den ersten 8 vollständigen Kommunikationszyklen abgeschlossen. Solange mindestens zwei Kaltstart-Knoten Botschaften senden, können sich andere Steuergerät jederzeit neu in die laufende Kommunikation einklinken.

Während der gesamten Kommunikation synchronisieren alle Busteilnehmer ständig ihren lokalen Takt, die so genannten *Mikroticks*, mit dem globalen Takt des Bussystems, den *Makroticks*. Dazu wertet jedes Steuergerät, auch die Kaltstart-Knoten, die Zeitabweichungen von empfangenen Botschaften mit gesetztem *Sync Frame Indicator Bit* aus und korrigiert ständig Frequenz und Phasenlage seiner eigenen Zeitbasis. Die Phasenabweichung (*Offset Error*) wird aus der Lage des Beginns der *Sync Frames* relativ zum Beginn des jeweiligen Slots bestimmt, die Frequenzabweichung (*Rate Error*) aus der zeitlichen Änderung der Phasenabweichung in aufeinanderfolgenden Kommunikationszyklen. Innerhalb des Kommunikationscontrollers werden mehrere Messwerte gespeichert. Bei jedem zweiten Kommunikationszyklus werden aus dem Mittelwert der Messwerte neue Korrekturwerte berechnet, wobei Messwertausreißer ignoriert werden (*Fault Tolerant Midpoint Algorithm*). Die Berechnung findet in jedem zweiten Zyklus im Kommunikationscontroller am Anfang der *Network Idle Time* automatisch statt. Zur Korrektur der Phasenlage wird die *Network Idle Time* dann unmittelbar um die entsprechende Zahl von *Mikroticks* verlängert oder verkürzt, während die Frequenzkorrektur durch geeignete Veränderung der *Makroticks* gleichmäßig über den gesamten Zyklus verteilt wird. Die maximale Taktfrequenzabweichung, die mit diesem Verfahren korrigiert werden kann, liegt bei 1500 ppm, d. h. etwa eine Größenordnung über den Toleranzen üblicher Quarzgeneratoren. Damit sich das System nicht aufschaukelt, wenn mehrere Knoten gleichzeitig ihre Taktperioden anpassen, muss die Empfindlichkeit des Korrekturverfahrens sorgfältig parametert werden (*Cluster Drift Damping*). Die Messdaten und Korrekturwerte des Kommunikationscontrollers können vom steuernden Mikrocontroller gelesen und beeinflusst werden, so dass der Mikrocontroller das Bussystem softwaremäßig auch mit externen Taktquellen synchronisieren kann (*External Offset and Rate Correction*).

Innerhalb eines Netzwerkes sollen mindestens 2 und höchstens 15 Steuergeräte, die so genannten Synchronisationsknoten (*Sync Node*), im statischen Teil des Kommunikationszyklus derartige *Sync Frame* Botschaften aussenden, so dass die Synchronisation auch beim Ausfall einzelner Steuergeräte noch aufrechterhalten werden kann. Da die beiden Kanäle eines zweikanaligen FlexRay-Systems synchron arbeiten müssen, sollten diese Steuergeräte stets Botschaften mit gesetztem *Sync Frame Indicator Bit* auf beiden Kanälen senden.

Der Zeitschlitz im statischen Segment, in dem die *Sync Frame* bzw. *Startup Frame* Botschaft gesendet wird, wird auch als *Keyslot* bezeichnet. Kommunikationscontroller können so konfiguriert werden, dass sie unmittelbar nach dem Netzwerk-Start im sogenannten

Single Slot Mode arbeiten. In diesem Modus senden sie dann je Zyklus nur die Botschaft in ihrem *Keyslot* und verhalten sich in allen anderen Zeitschlitten passiv. Auf diese Weise kann der Netzwerkverkehr beim Hochlauf eines Systems reduziert werden, bis alle Steuergeräte bereit sind, den Normalbetrieb aufzunehmen.

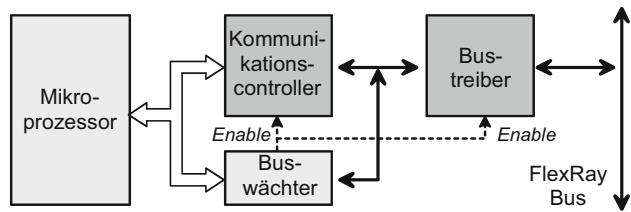
Beim fehlerhaftem Hochlauf des Netzwerks kann es zur sogenannten *Cliquenbildung* kommen. Dabei bilden sich mehrere Gruppen (*Cliquen*) von Steuergeräten, die innerhalb der jeweiligen Gruppe synchronisiert sind, zwischen den Gruppen aber asynchron bleiben. Ein derartiges Verhalten kann z. B. auftreten, wenn zwei FlexRay-Linien-Busse über einen aktiven Stern miteinander verbunden sind und der aktive Stern erst dann eingeschaltet wird, wenn sich die Steuergeräte an den beiden Linien-Bussen jeweils bereits untereinander synchronisiert haben. Cliquenbildung und andere Kommunikationsprobleme können z. B. durch Verwendung des *Network Management Vectors* erkannt werden. Wenn bei einer Botschaft im statischen Segment das *Payload Preamble Indicator Bit* gesetzt ist, werden die ersten maximal 12 Nutzdatenbytes dieser Botschaft als *Network Management Vector* interpretiert. Diese Datenbytes werden vom Kommunikationscontroller in einem speziellen Register gespeichert und dabei mit allen anderen derartigen Botschaften logisch-ODer verknüpft. Wenn jedes Steuergerät innerhalb seines *Network Management Vectors* genau ein eindeutiges Bit setzt und alle übrigen auf 0 belässt, kann jedes Steuergerät im *Network Management Register* erkennen, von welchem Steuergerät es Botschaften empfangen hat und von welchem Steuergerät nicht. Zur Auflösung von *Cliquen* muss die Kommunikation beendet und neu gestartet werden. Grundsätzlich vermeiden lässt sich *Cliquenbildung*, wenn in einem Netz nicht mehr als drei Steuergeräte für die Zeitsynchronisation konfiguriert werden.

3.3.4 Fehlerbehandlung, Bus Guardian

Durch die verschiedenen Zähler (*Cycle Counter*, *Slot Counter*) und Zeitüberwachungen sowie die beiden CRC-Prüfsummen erkennt der Kommunikationscontroller eine große Anzahl von Übertragungsfehlern. Abhängig von der Schwere des Fehlers schaltet er sich dann in einen passiven Modus, in dem er nicht mehr sendet, aber noch Botschaften empfängt und versucht, sich neu zu synchronisieren, oder er schaltet sich ganz ab. In jedem Fall teilt der Kommunikationscontroller der darüber liegenden Software-Protokollschicht den Fehler mit. Eine automatische Botschaftswiederholung durch die Kommunikationscontroller im Fehlerfall, wie sie bei CAN vorhanden ist, gibt es bei FlexRay nicht, um den deterministischen Betrieb auf dem Bus in jedem Fall zu gewährleisten.

In Systemen mit hohen Sicherheitsansprüchen schützt die zweikanalige Struktur gegen Ausfälle eines einzelnen Kanals, doch besteht immer noch die Gefahr, dass ein fehlerhaftes Steuergerät, das an beide Kanäle angeschlossen ist, auf beiden Kanälen unkontrolliert und zu beliebigen Zeitpunkten sendet. Ein derartiges Verhalten wird gelegentlich als *Babbling Idiot* Fehler bezeichnet. Um diesen Fehler zu beherrschen, besteht optional die Möglichkeit, zusätzlich zum Kommunikationscontroller und zu den Bustreibern in jedem Steuergerät

Abb. 3.23 Kommunikationscontroller, Bustreiber und Buswächter (Darstellung bei einem einkanaligen System)



oder zentral im aktiven Sternpunkt einen so genannten Buswächter (*Bus Guardian*) einzusetzen (Abb. 3.23). Der Buswächter ist eine Art einfacher Kommunikationscontroller, der selbst keine Botschaften sendet, aber so konfiguriert ist, dass er den zeitlichen Ablauf eines Kommunikationszyklus kennt und den Sendeteil des Kommunikationscontrollers bzw. Bustreibers nur in denjenigen Zeitschlitten freigibt, in denen das eigene Steuergerät tatsächlich senden darf. Auf diese Weise kann ein fehlerhaft arbeitendes Steuergerät zwar immer noch selbst falsche eigene Botschaften aussenden, aber nicht die Botschaften anderer Steuergeräte stören. Theoretisch sollte ein derartiger Buswächter auch schaltungstechnisch unabhängig vom Kommunikationstreiber realisiert werden, doch wird er aus Kostengründen in der Praxis wohl eher zusammen mit dem Kommunikationscontroller oder gegebenenfalls auch mit den Bustreibern integriert, so dass seine Überwachungsfunktion nur begrenzt redundant ist. Alternativ zu einem *lokalen Bus Guardian* im Steuergerät besteht die Möglichkeit, einen *zentralen Bus Guardian* z. B. im aktiven Sternpunkt anzurufen. Die zugehörigen Spezifikationen sind in beiden Fällen jedoch ausdrücklich als vorläufig gekennzeichnet, serientaugliche Implementierungen existieren noch nicht.

3.3.5 Konfiguration und übergeordnete Protokolle

Ähnlich wie in der Entstehungsphase von CAN sind die unteren Protokollsichten 1 und 2 sehr detailliert, leider teilweise aber auch unübersichtlich spezifiziert. Für die höheren Protokollsichten dagegen fehlen entsprechende Festlegungen noch ganz oder werden gerade erst erarbeitet. Die Kommunikationszyklen mit der Zuordnung der Zeitschlitte zu den einzelnen Steuergeräten werden voraussichtlich überwiegend statisch, d. h. in der Entwicklungsphase konfiguriert werden. Dies gilt insbesondere auch, weil FlexRay-Kommunikationscontroller dynamisch nur umständlich und in der Regel nur mit Unterbrechung der Buskommunikation umkonfiguriert werden dürfen. Von CAN vorhandene Ansätze wie das OSEK-Netzmanagement (siehe Kap. 7) oder ein Transportprotokoll ähnlich ISO 15765-2 (siehe Kap. 4) werden in angepasster Form übernommen. Entsprechende Aktivitäten erfolgen gegenwärtig vor allem im Rahmen der AUTOSAR-Initiative (siehe Kap. 8).

Die Einführung von FlexRay verläuft langsamer als bei CAN, weil der Abstimmungsprozess der Spezifikationen in dem inzwischen viele Mitglieder umfassenden FlexRay-Konsortium naturgemäß zeitaufwendig war und sich dadurch die Bereitstellung von

Tab. 3.10 Botschaftslängen für $f_{\text{Bit}} = 1 / T_{\text{bit}} = 10 \text{ Mbit/s}$

n_{Data}	T_{Frame}	n_{Data}	T_{Frame}
8 byte	17 μs	32 Byte	41 μs
16 byte	25 μs	254 Byte	263 μs

funktionsfähigen Kommunikationscontrollern durch die Halbleiterhersteller verzögert hatte. Zum anderen erfolgt die Einführung der verteilten X-by-Wire-Anwendungen, für die FlexRay als Schlüsseltechnologie konzipiert war, aus technischen und wirtschaftlichen Gründen, die von FlexRay unabhängig sind, schleppender als erwartet.

3.3.6 Zeitverhalten von FlexRay-Systemen, Beispiel-Konfiguration

Die Länge einer FlexRay Botschaft (Abb. 3.21) ist näherungsweise

$$T_{\text{Frame}} \approx \left[\frac{10}{8} (n_{\text{Header}} + n_{\text{Trailer}} + n_{\text{Data}}) + n_{\text{TSS+FSS+FES}} \right] \cdot T_{\text{bit}}. \quad (3.18)$$

Dabei ist $n_{\text{Data}} = 0, \dots, 2032$ bit die Anzahl der Datenbits, die in Stufen von 16 bit variabel ist. Header und Trailer umfassen $n_{\text{Header}} + n_{\text{Trailer}} = 64$ bit. Der Faktor 10 / 8 berücksichtigt, dass der *Physical Layer* je Byte des *Data Link Layers* zusätzlich eine *Byte Start Sequenz BSS* mit 2 bit eingefügt. Zusätzlich ergänzt der Kommunikationscontroller am Frame-Anfang eine 3 bis 15 bit lange *Transmission Start Sequenz TSS*, die Aktivierungsverzögerungen von Transceivern ausgleichen soll, sowie ein *Frame Start FES* und am Frame-Ende zwei *Frame End Sequenz FSS* Bits, insgesamt also $n_{\text{TSS+FSS+FES}} = 6 \dots 18$ bit. In Tab. 3.10 wurde mit 10 bit gerechnet.

Beispielhaft soll ein 10 Mbit/s-FlexRay-System dargestellt werden, das sich an [11] orientiert. Die Botschaften im statischen Segment sollen $n_{\text{Data}} = 16$ Datenbytes enthalten, so dass sich eine Frame-Dauer $T_{\text{Frame}} \approx 25 \mu\text{s}$ ergibt. Das Versenden der Botschaft wird gegenüber dem Beginn des Zeitschlitzes um den sogenannten *Action Point Offset* $T_{\text{AP,Offset}}$ verzögert und muss mindestens 11 Bitzeiten (*Channel Idle Delimiter*) vor Ende des Zeitschlitzes abgeschlossen sein (Abb. 3.24). Durch die Lücken zu Beginn und am Ende des Slots wird sichergestellt, dass der Frame aus Sicht aller Empfänger stets innerhalb desselben Slots beginnt und endet, auch wenn er durch die Bustreiber und die Leitungslaufzeit verzögert wird (*Propagation Delay*, nach Spezifikation max. 2,5 μs) und die lokalen Zeitbasen in Sender und Empfänger trotz der laufenden Taksynchronisation voneinander abweichen (*Assumed Clock Precision* nach [14] typ. 1 ... 3 μs). Im Beispiel werden die Lücken $T_{\text{AP,Offset}}$ und $T_{\text{CH,Idle}}$ zu je 5 μs gewählt, woraus sich die Gesamtdauer eines Zeitschlitzes im statischen Segment zu

$$T_{\text{StaticSlot}} = T_{\text{Frame}} + T_{\text{AP,Offset}} + T_{\text{CH,Idle}} \quad (3.19)$$

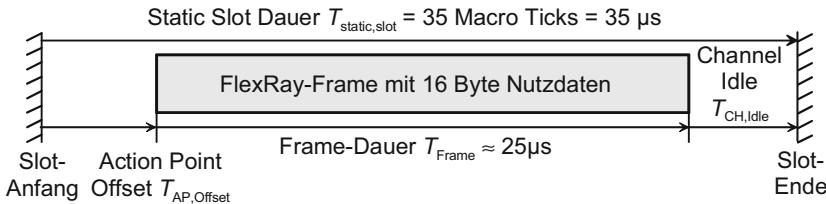


Abb. 3.24 Aufbau eines Zeitschlitzes (Static Slot) im statischen Segment

ergibt, im Beispiel $35 \mu\text{s}$. Die Länge muss ein ganzzahliges Vielfaches der Dauer eines *Macroticks* MT sein, hier wird $T_{MT} = 1 \mu\text{s}$ festgelegt. Laut Norm sind 1 bis $6 \mu\text{s}$ zulässig. In den *Minislots* des dynamischen Segments und im *Symbol Window* sind ähnliche Sicherheitsabstände notwendig. Da ein Frame im dynamischen Segment aber stets mehrere *Minislots* dauert und in unterschiedlichen *Minislots* beginnt und endet, reicht es, die Dauer der *Minislots* mit $T_{DynamicSlot} = 5 T_{MT} = 5 \mu\text{s}$ zu wählen.

Die interne Taktperiode $T_{\mu\text{T}}$ des Kommunikationscontrollers, mit der die Bussignale abgetastet werden, soll nach FlexRay-Spezifikation $T_{Sample} = T_{bit} / 8$ betragen, bei $f_{bit} = 10 \text{ Mbit/s}$ also $T_{Sample} = 12,5 \text{ ns}$. In der Regel wird im Hinblick auf eine möglichst hohe Auflösung $T_{\mu\text{T}} = T_{Sample}$ als *Mikrotick* gewählt, obwohl auch die doppelte Dauer zulässig wäre. Im vorliegenden Beispiel ist damit $T_{MT} = 80 T_{\mu\text{T}}$.

Innerhalb des Kommunikationszyklus mit $T_{Zyklus} = 5 \text{ ms}$ Dauer wird das statische Segment mit $T_{StatSeg} = 3 \text{ ms}$ festgelegt (Abb. 3.25). Insgesamt passen etwa 85 der in Abb. 3.24 dargestellten Botschaften mit je 16 Datenbyte in das statische Segment, so dass sich dafür eine Nutzdatenrate von ca. 270 KB/s ergibt. Für das dynamische Segment, das *Symbol Window* *SYM* und die *Network Idle Time* *NIT* verbleiben zusammen 2 ms . Durch die asymmetrische Aufteilung zwischen statischem und dynamischem Segment lässt sich für schnelle Regelaufgaben effektiv ein $2,5 \text{ ms}$ Zeitraster einrichten, in dem Größen am Anfang und nochmals am Ende des statischen Segments in einem jeweils etwa $500 \mu\text{s}$ langen Fenster übertragen werden ($2,5 \text{ ms Slots}$). Danach folgen Größen, die alle 5 ms übertragen werden und schließlich Signale, die nur jedes 2. Mal, jedes 4. Mal usw. gesendet werden (*Slot* bzw. *Cycle Multiplexing*).

Mit diesem Kommunikationsschema lässt sich das statische Segment in typischen verteilten Regelsystemen einsetzen. Im einfachsten Fall werden die Regeldaten einmal je Zyklus übertragen, d. h. die Abtastzeit des Regelsystems entspricht der Zykluszeit (Fall a) in Abb. 3.26). Durch die asymmetrische Aufteilung der Segmente können die Regeldaten aber auch zweimal je Zykluszeit übertragen werden, so dass sich die Abtastzeit halbiert (Fall b). Sind Regler, Sensoren und Stellglied auf drei Steuergeräte verteilt, kann man die Totzeit im Regelkreis klein halten, indem man die Sensordaten am Anfang des statischen Segments überträgt (Fall c), so dass der Regler ausreichend Zeit hat, die Stellgröße zu berechnen und noch im selben Zyklus weiter an das Stellglied zu senden (*In Cycle Response*).

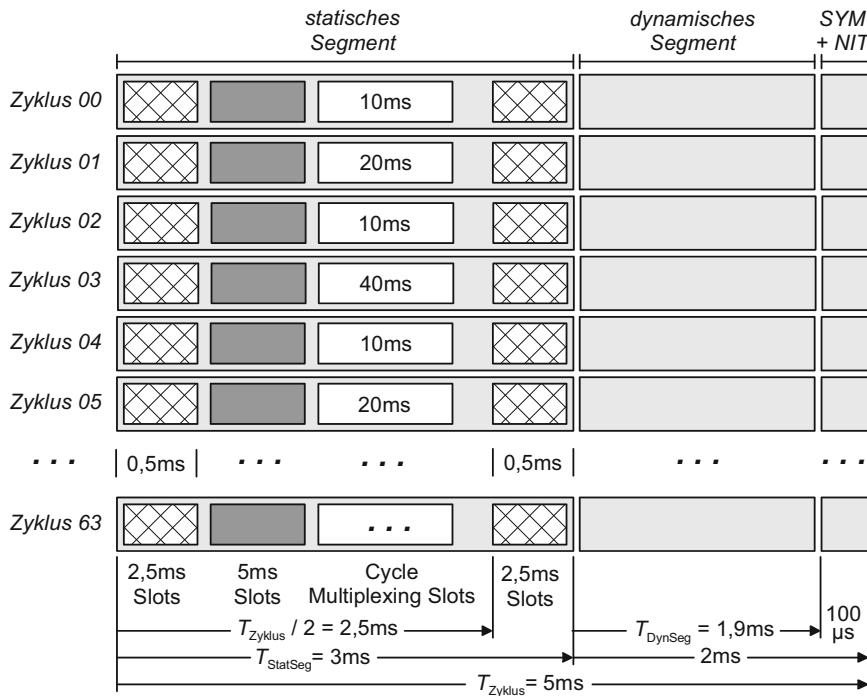


Abb. 3.25 Kommunikationszyklen ähnlich [11]

Im dynamischen Segment können Botschaften variabler Länge versendet werden. Bei maximal 254 Byte Nutzdaten dauert ein Frame bis zu ca. 265 μs , benötigt also zusammen mit dem beschriebenen Sicherheitszuschlag ungefähr 55 Minislots. Das *Symbol Window* *SYM* müsste ausreichend lang für die Übertragung des *Collision Avoidance Symbols* *CAS* (30 bit) plus der *Transmission Start Sequenz* *TSS* (max. 15 bit) sein. Zusammen mit denselben Sicherheitsabständen wie im statischen Segment (Abb. 3.24) ergibt sich dafür eine Dauer von ca. 15 μs . Die *Network Idle Time* *NIT* muss so lang sein, dass der Kommunikationscontroller die Taktfrequenz- und Phasenkorrekturwerte auch im ungünstigsten Fall berechnen und die Phasenkorrektur ausführen kann, wobei die Berechnung bereits während des dynamischen Segments beginnen darf. Im Beispiel werden für das *Symbol Window* und die *Network Idle Time* zusammen ca. $T_{SW} + T_{NIT} = 100\mu s$ reserviert. In das dynamische Segment mit einer Länge von $T_{DynSeg} = T_{Zyklus} - T_{StatSeg} - (T_{SW} + T_{NIT}) = 1,9\text{ ms}$ passen damit gut 7 Botschaften mit 254 Byte Nutzdaten, woraus sich im günstigsten Fall eine Nutzdatenrate von etwa 350 KB/s ergibt. Sendet man auch hier lediglich Botschaften mit im Mittel nur 16 Byte Nutzdaten, kann man zwar über 50 solcher Botschaften übertragen, die effektive Datenrate für das dynamische Segment reduziert sich dann aber auf unter 180 KB/s.

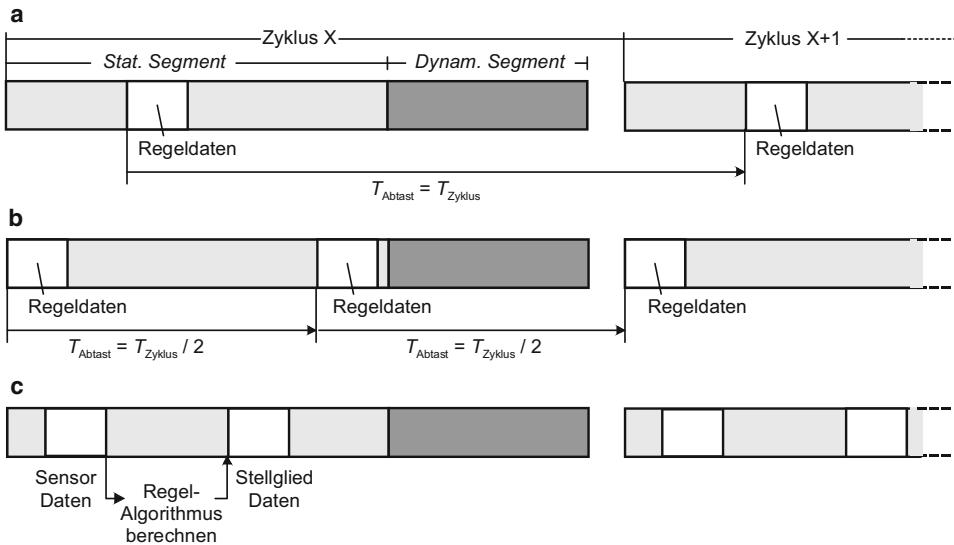


Abb. 3.26 Nutzung des statischen Segments in verteilten Regelsystemen

Tab. 3.11 FlexRay-Parameter bei BMW nach [11]

Zykluszeit	5 ms	Dauer Makrotick	1,375 μ s
Statisches Segment	3 ms	Dynam. Segment inkl. NIT	2 ms
Nutzdaten im stat. Segment	16 Byte	Nutzdaten im dyn. Segment	2 ... 254 Byte
Statische Slots	91	Minislots (Dynamische Slots)	289
Dauer stat. Slot	33 μ s	Dauer Minislot	6,875 μ s

Zum Vergleich mit dem obigen Beispiel zeigt Tab. 3.11 die FlexRay-Parameter, wie sie bei BMW im Serieneinsatz sind.

3.3.6.1 Latenzzeiten im statischen und im dynamischen Segment

Für die Latenzzeit von Botschaften im statischen Segment gelten dieselben Überlegungen wie bei LIN (Abschn. 3.2.8). Es ergibt sich also

$$T_{\text{Frame}} < T_{\text{Latenz,StatSeg}} < T_{\text{Zyklus}} + T_{\text{Frame}}. \quad (3.20)$$

Um den Jitter möglichst gering zu halten, sollte die Anwendungssoftware, welche die Daten bereitstellt, mit dem Zyklus des Bussystems synchronisiert werden.

Im dynamischen Segment kann eine Botschaft in einem späteren Minislot durch Botschaften in früheren Minislots verzögert werden. Ist die Verzögerung so groß, dass die Botschaft nicht mehr sicher bis zum Ende des aktuellen dynamischen Segments übertragen werden kann, so wird die Botschaft für einen oder gar mehrere Zyklen verzögert. Die Zuordnung einer Botschaft zu einem bestimmten Minislot wirkt damit wie die Festlegung

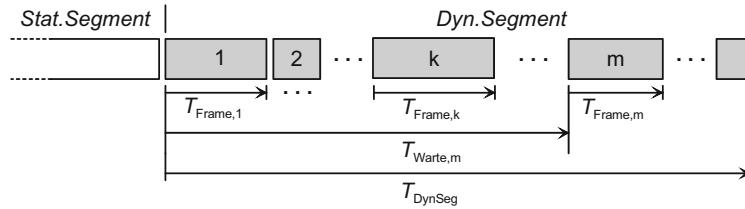


Abb. 3.27 Botschaften im dynamischen Segment

der Priorität einer Botschaft. Für die Berechnung der Latenz werden folgende Annahmen getroffen:

- Die Botschaft wird am Anfang des dynamischen Segments sendebereit. Die Latenzzeit bezieht sich auf diesen Zeitpunkt.
- Die Minislots werden beginnend mit $k = 1$ durchnummeriert, die betrachtete Botschaft soll im Minislot m gesendet werden. Die Dauer eines Minislots sei $T_{DynamicSlot}$.
- Die Botschaften des dynamischen Segments werden periodisch mit der jeweiligen Periode T_k versendet. In der Regel wird $T_k > T_{Zyklus}$ sein, sonst könnte man die Botschaften ja auch im statischen Segment versenden. Für Botschaften, die nicht periodisch versendet werden, wird T_k als Mindestabstand zwischen zwei Übertragungen interpretiert (*Interarrival Time*).

Die Latenzzeit der Botschaft m setzt sich aus der Wartezeit bis zum Beginn der Übertragung und der eigentlichen Übertragungsdauer zusammen (Abb. 3.27):

$$T_{Latenz,DynSeg,m} = T_{Warte,m} + T_{Frame,m}. \quad (3.21)$$

Die kleinstmögliche Wartezeit für die Botschaft m ergibt sich, wenn vor ihr im dynamischen Segment keine anderen Botschaften gesendet werden. Dann muss die Botschaft nur die $(m - 1)$ leeren Minislots abwarten, bevor die Übertragung beginnt:

$$T_{Warte,m,min} = (m - 1) \cdot T_{DynamicSlot}. \quad (3.22)$$

Am größten wird die Latenzzeit für die Botschaft m , wenn in den vorherigen $(m - 1)$ Minislots des dynamischen Segments auch andere Botschaften gesendet werden. Für diesen Fall kann die Wartezeit abgeschätzt werden:

$$T_{Warte,m} \approx \left(\sum_{k=1}^{m-1} T_{frame,k} \right) \bmod T_{DynSeg} + \left\lceil \frac{\sum_{k=1}^{m-1} \left\lceil \frac{T_{Warte,m}}{T_k} \right\rceil T_{frame,k}}{T_{DynSeg}} \right\rceil \cdot T_{Zyklus}. \quad (3.23)$$

Der erste Term beschreibt die Verzögerung durch diejenigen Botschaften, die im selben dynamischen Segment vor Botschaft m gesendet werden. Für nicht belegte Minislots wird $T_{\text{Frame},k} = T_{\text{DynamicSlot}}$. Der zweite Term ist zu beachten, wenn die Botschaften $k = 1 \dots m$ nicht mehr alle in ein einziges dynamisches Segment passen, d. h. wenn

$$\sum_{k=1}^m T_{\text{Frame},k} > T_{\text{DynSeg}} \quad (3.24)$$

ist. In diesem Fall vergrößert sich die Wartezeit um einen oder sogar mehrere Zyklen. Die Anzahl der zusätzlichen Zyklen wird durch den Term in $\lfloor \dots \rfloor$ beschrieben, der auf den nächsten ganzzahligen Wert abgerundet wird. Der Ausdruck in $\lceil \dots \rceil$, der auf den nächsten ganzzahligen Wert aufgerundet werden muss, berücksichtigt, dass die anderen Botschaften während einer längeren Wartezeit mehrfach sendebereit werden können. Ähnlich wie bei CAN (Abschn. 3.1.7) ist auch diese Gleichung nur iterativ lösbar. Gleichung 3.23 liefert nur eine grobe Abschätzung, weil sie nicht berücksichtigt, dass nicht nur Botschaft m sondern auch alle anderen Botschaften stets vollständig innerhalb des laufenden Zyklus gesendet werden müssen. Andernfalls müssen sie ebenfalls für mindestens einen Zyklus warten. Dadurch kann sich sogar die Reihenfolge ändern, wenn die Botschaft in einem früheren Minislot nicht gesendet werden kann, weil sie nicht mehr in das aktuelle Segment passt, während eine eigentlich erst in einem späteren Minislot folgende, aber kürzere Botschaft noch gesendet werden kann. In den einzelnen Zyklen bleiben dann unterschiedliche Minislots frei. Die exakte Berechnung des Worst Case Falles ist aufwendig und wird z. B. in [16] dargestellt.

Gleichung 3.24 unterteilt das dynamische Segment faktisch in eine Klasse von Botschaften, deren Übertragungsverzögerung trotz der ereignisgesteuerten Übertragung im dynamischen Segment deterministisch bleibt, und eine zweite Klasse, deren Übertragungsverzögerung extrem stark schwanken und für die nur schwer eine Obergrenze angegeben werden kann.

3.3.7 Schnittstelle zum FlexRay-Controller

Bevor ein FlexRay-Kommunikationscontroller (Abb. 3.28) Botschaften senden und empfangen kann, muss er zunächst konfiguriert werden. Die FlexRay-Spezifikation definiert dafür einen Zustandsautomaten *Protocol Operation Control POC* (Abb. 3.29) und legt die Befehle fest, mit denen der steuernde Mikrocontroller die Zustandsübergänge über das *Controller Host Interface CHI* auslöst.

Nach dem Einschalten (*Reset*) werden über die Protokollkonfigurationsregister zunächst die Bitrate, die Dauer von Makro- und Mikroticks sowie die Längen von statischem und dynamischem Segment, *Symbol Window* und *NIT* festgelegt. Diese in der FlexRay-Spezifikation beschriebenen mehr als 50 Konfigurationsparameter dürfen nur im Zustand *Config* verändert werden, in dem noch keine Buskommunikation stattfindet. Anschlie-

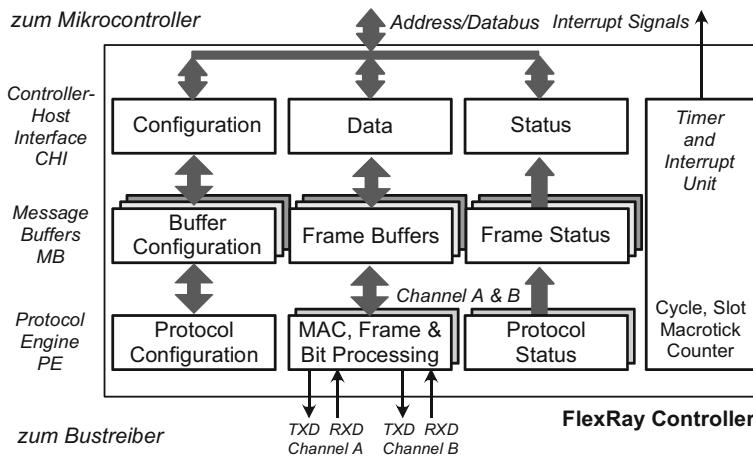


Abb. 3.28 Blockschaltbild eines typischen FlexRay-Kommunikationscontrollers

ßend konfiguriert der Mikrocontroller die Botschaftsspeicher (*Message Buffer*), bevor er die Kommunikation über den Zwischenzustand *Ready* und den Befehl *Run* freigibt. Der Kommunikationscontroller führt dann selbstständig die in Abschn. 3.3.3 beschriebene Integration in eine bestehende Buskommunikation durch bzw. beteiligt sich aktiv am Start des Netzwerkes, falls er zuvor mit dem Befehl *Allow Coldstart* als Kaltstart-Knoten

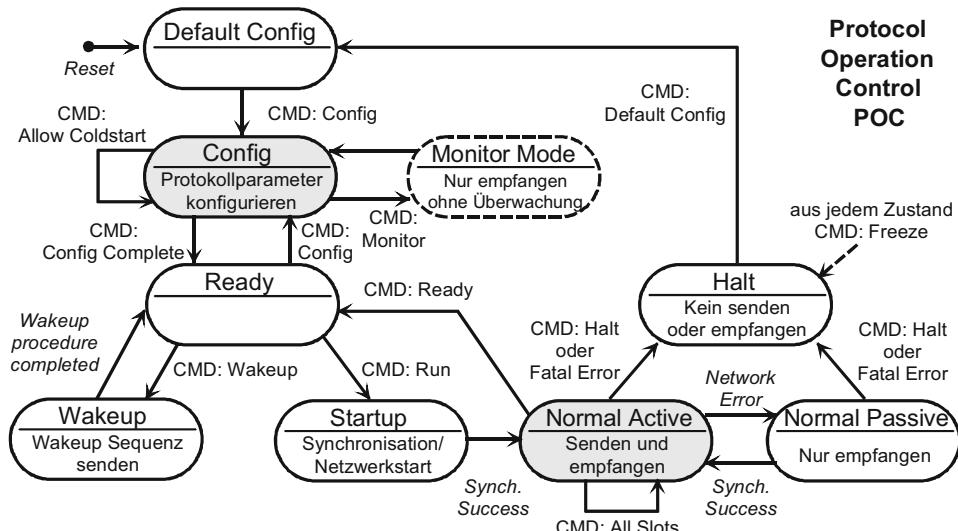


Abb. 3.29 POC Zustände des FlexRay-Kommunikationscontrollers (vereinfacht), *CMD*: Host Command des steuernden Mikrocontrollers; *kursiv*: automatisch erfolgende Zustandsübergänge

konfiguriert wurde. Nach erfolgter Synchronisation wechselt er in den Zustand *Normal Active*, in dem das vorkonfigurierte Senden und Empfangen von Botschaften erfolgt. Falls der Kommunikationscontroller so konfiguriert war, dass er zunächst im *Single Slot Mode* arbeitet, wird er durch den Befehl *Allow All Slots* in den Normalbetrieb umgeschaltet, in dem er in allen konfigurierten Zeitschlitten senden darf. Beim Auftreten von Synchronisationsfehlern wechselt er je nach Schwere des Fehlers in den Zustand *Normal Passive*, in der er nur noch Botschaften empfängt, oder in den Zustand *Halt*, den er nur durch eine Neukonfiguration wieder verlassen kann. Bei Bedarf kann der Mikrocontroller die Kommunikation anhalten und den Kommunikationscontroller in den Zustand *Ready* versetzen, in dem keine Botschaften versendet oder empfangen werden, sondern lediglich der Empfang eines *Wakeup Patterns WUP* erkannt wird. Von dort aus kann der Mikrocontroller entweder selbst das Versenden eines *Wakeup Patterns* verlassen und/oder die Kommunikation über den Befehl *Run* wieder aufnehmen. Zusätzlich lassen sich viele FlexRay-Kommunikationscontroller in einen *Monitor Mode* versetzen, in dem Botschaften und Symbole empfangen werden, ohne dass der Kommunikationscontroller sich auf das Slot- und Zyklusraster aufsynchronisieren muss. Dieser Modus kann ähnlich wie der *Listen* oder *Silent Mode* bei CAN-Kommunikationscontrollern (siehe Abschn. 3.3.6) zu Diagnosezwecken etwa während des Netzwerk-Starts eingesetzt werden.

Ebenso wie die CHI-Befehlsschnittstelle gibt die FlexRay-Spezifikation auch vor, welche Statusinformationen der Controller bereitstellen muss. Zu den über 40 geforderten Werten gehören neben dem aktuellen Makrotick, dem Zykluszähler und den *Slot Counter*n für die beiden FlexRay-Kanäle Informationen über die Zeitsynchronisation und empfangene Symbole, Fehlerflags der verschiedenen Format- und Zeitüberwachungsmechanismen sowie der *Network Management Vector*.

Im Gegensatz zur Konfiguration, Ablaufsteuerung und Statusabfrage des Protokolls definiert die FlexRay-Spezifikation den Aufbau der Botschaftsspeicher (*Message Buffers*) nur relativ grob (Abb. 3.30). Daher sind die Implementierungsunterschiede zwischen verschiedenen Herstellern verhältnismäßig groß. In jedem Fall kann man den oder die FlexRay-Kanäle und die *Frame* bzw. *Slot ID* konfigurieren und einstellen, ob die Botschaft bei jedem Kommunikationszyklus oder nur bei jedem 2., 4. Mal usw. gesendet bzw. empfangen werden soll. Zum Teil wird zur Unterstützung des *Slot Multiplexing* eine Empfangsfilterung auf Basis der 16 bit großen *Message ID* angeboten, die am Anfang einer Botschaft im dynamischen Segment versendet werden kann. Ansonsten findet die Sendeentscheidung bzw. Empfangsfilterung auf Basis der Kanal-, Zyklus- und Slot-Zähler-Konfiguration statt. Wie bei CAN-Controllern lässt sich die Empfangsfilterung meist auch so konfigurieren, dass ein Botschaftsspeicher für eine Gruppe von Botschaften gemeinsam verwendet werden kann.

Zum Senden muss die Anwendung auf dem Mikrocontroller den kompletten *Frame Header* sowie die Nutzdaten bereitstellen. Im statischen Segment reicht in der Regel eine einmalige Konfiguration des *Frame Headers*. Im dynamischen Segment dagegen muss die im *Frame Header* enthaltene CRC-Prüfsumme vom Mikrocontroller jedes Mal neu berechnet werden, wenn sich die Nutzdatenlänge ändert (Abb. 3.21). Der Zykluszähler, der nicht Teil der *Header*-Prüfsumme ist, sowie die CRC-Prüfsumme im *Frame Trailer* werden vom

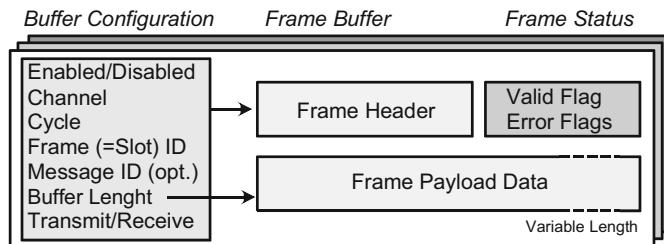


Abb. 3.30 Typischer FlexRay-Botschaftsspeicher (*Message Buffer*)

Kommunikationscontroller automatisch ermittelt. Sobald die Daten komplett in den Puffer kopiert wurden, zeigt der Mikrocontroller dies durch Setzen des *Valid Flag* an. Nach dem tatsächlichen Versenden der Botschaft informiert der Kommunikationscontroller den steuernden Mikroprozessor durch Setzen eines Flags oder durch einen Interrupt. Optional setzt der Kommunikationscontroller das *Valid Flag* nach dem Senden automatisch zurück, so dass stets nur Daten versendet werden, die vom Mikrocontroller aktualisiert wurden (*Event Triggered Transmit*). Befinden sich zum Sendezeitpunkt keine gültigen Daten im *Frame Buffer*, sendet der Kommunikationscontroller im statischen Segment einen *Null Frame*, im dynamischen Teil sendet er dann gar keine Botschaft. Analog setzt der Kommunikationscontroller das *Valid Flag* beim Empfang einer Botschaft und informiert den Mikrocontroller optional per Interrupt über den Empfang. Sowohl beim Versenden als auch beim Empfang prüft der Kommunikationscontroller, ob *Header*-Format sowie Prüfsummen korrekt sind und die Zeitbedingungen bezüglich *Frame*-Beginn und Ende eingehalten wurden und zeigt Fehler durch verschiedene *Error Flags* an.

Weiterhin fordert die Spezifikation die Bereitstellung von mindestens zwei absolut bzw. relativ arbeitenden *Timern*, die den Mikrocontroller beim Erreichen bestimmter Stände des Zyklus-, Makrotick- und/oder *Slot Counters* durch Interrupts alarmieren können, so dass die dort ablaufende Software sich mit dem Bussystem synchronisieren und die Sendebotschaften rechtzeitig bereitstellen kann.

Die Anzahl und Größe der Botschaftsspeicher sowie der Mechanismus, mit dem der Zugriff des Mikrocontrollers und der *Protocol Engine* auf den Botschaftsspeicher synchronisiert werden, sind implementierungsabhängig. Der in [14] beschriebene Kommunikationscontroller beispielsweise kann zwischen 128 Botschaften mit je 48 Byte Nutzdaten und 30 Botschaften mit je 254 Byte Nutzdaten speichern. Ein Teil des Botschaftsspeichers lässt sich alternativ als FIFO konfigurieren, so dass auch Botschaften empfangen werden können, für die kein eigener Botschaftsspeicher bereitgestellt werden kann. Für die Synchronisation beim Zugriff auf den Botschaftsspeicher können alternativ sowohl aufwendigere Doppel-Puffer-Konzepte als auch einfachere Lock/Unlock-Mechanismen implementiert werden [15]. Bei Doppel-Puffer-Konzepten kann der Mikrocontroller einen Botschaftspuffer auslesen, während die *Protocol Engine* bereits eine neue Botschaft in den zugeordneten Hintergrundspeicher schreibt und umgekehrt. Beim Lock/Unlock-Verfahren sperrt

die jeweils zugreifende Einheit den Zugriff für die andere Einheit, so dass zumindest keine inkonsistenten Botschaften gelesen oder gesendet werden, gegebenenfalls aber Botschaften verloren gehen.

Sollen mehr Botschaften gesendet oder empfangen werden als *Message Buffer* zur Verfügung stehen, muss die Treibersoftware die *Buffer* dynamisch umkonfigurieren. Dies ist eine verhältnismäßig anspruchsvolle Operation. Falls ein Empfangs- oder Sendepuffer zu spät aktiviert wird, geht die Botschaft unter Umständen verloren. Wird ein Sendepuffer zu früh aktiviert oder zu spät deaktiviert, kann es sogar zu Kollisionen auf dem Bus kommen, wenn derselbe Zeitschlitz im dynamischen Segment im Multiplex-Betrieb auch von einem anderen Busteilnehmer genutzt wird. Gerade die zeitliche Synchronisation der Treibersoftware im dynamischen Segment ist außerordentlich kritisch, weil die exakte zeitliche Lage eines *Slots* dort davon abhängt, welche Botschaften vorher gesendet oder nicht gesendet wurden. Unter Umständen kann sogar zu einer Verschiebung in einen der folgenden Kommunikationszyklen kommen, falls das Segment bereits durch vorherige Botschaften zu lange belegt war.

3.3.8 Weiterentwicklung FlexRay 3.x

Nachdem das Protokoll mit Version 2.1 A einen stabilen Zustand erreicht hatte, eine Reihe von Kommunikationscontrollern auf dem Markt und erste Fahrzeuge mit FlexRay im Serieneinsatz waren, sollte mit der Ende 2010 veröffentlichten Version 3.0 offene Punkte in der Protokollspezifikation geklärt werden. Die Randbedingungen und gegenseitigen Abhängigkeiten der Parameter eines FlexRay-Systems wurden ausführlicher spezifiziert, doch gibt es leider weiterhin keinen Default-Parametersatz. Die Inbetriebnahme erfordert daher immer noch umfangreiche Konfigurationsarbeiten, die für Entwickler ohne umfangreiche FlexRay-Erfahrung und Werkzeugunterstützung fehlerträchtig und mühsam sein können.

Kleinere Änderungen betreffen das sogenannte *Cycle* bzw. *Slot Multiplexing*, d. h. die Zuteilung der Sendeberechtigung in einem Zeitschlitz auf verschiedene Steuergeräte in unterschiedlichen Kommunikationszyklen. Als Periodendauern für das *Multiplexing* sind nicht mehr nur duale (1, 2, 4, ...) sondern auch dezimale Vielfache (5, 10, ...) von T_{Zyklus} zulässig. Der Zykluszähler, der bisher stets 64 Kommunikationszyklen durchlief, darf bei jedem geraden Wert zwischen 8 und 64 vorzeitig zurückgesetzt werden. Außerdem ist das *Slot Multiplexing* nun nicht nur im dynamischen, sondern auch im statischen Segment erlaubt.

Des weiteren spezifiziert FlexRay 3.0 zusätzlich zu 10 Mbit/s auch die Bitraten 2,5 und 5 Mbit/s vollständig. Dadurch lassen sich EMV-Probleme reduzieren und FlexRay-Systeme in kostengünstigerer Linienbus-Topologie aufbauen, weil eventuell auf den teuren aktiven Sternpunkt verzichtet werden kann. Diese Änderungen wurde u. a. vom JASPAR-Konsortium angeregt, das die Übernahme der FlexRay- und AUTOSAR-Spezifikationen für die Anwendung bei japanischen Herstellern prüft.

Tab. 3.12 Konfiguration bei verschiedenen Netzwerk-Start Konzepten

	TT-D	TT-L	TT-E
Anzahl N der <i>Coldstart Sync Nodes</i>	2 ... 15	1	1 ... 7
Anzahl M der <i>Non-Coldstart Sync Nodes</i>	0 ... 15 - N	0	0
Anzahl der <i>Keyslots</i> je <i>Coldstart Node</i> bzw. <i>Non-Coldstart Sync Node</i>	1	2	2
Anzahl der <i>Keyslots</i> bei den anderen Steuergeräten	0	0	0

Weitere Neuerungen gibt es im Bereich des Netzwerk-Starts und der Takt-Synchronisation. Beim bisherigen, zur Unterscheidung als *Time Trigger – Distributed TT-D* bezeichneten Verfahren sind mindestens zwei Steuergeräte als *Coldstart Nodes* notwendig (Abschn. 3.3.3). Diese Geräte senden in jeweils genau einem Zeitschlitz je Steuergerät, dem *Keyslot*, Botschaften mit gesetzten *Startup Frame* und *Sync Frame Indicator Bit*. Mit Hilfe dieser Botschaften wird das Netz im günstigsten Fall innerhalb von 8 Kommunikationszyklen gestartet und erhält im laufenden Betrieb die zeitliche Synchronität aufrecht (Abb. 3.22). Um die Verfügbarkeit des Netzes bei Ausfall einzelner Geräte zu erhöhen, können insgesamt bis zu 15 Steuergeräte als *Coldstart Nodes* konfiguriert werden (Tab. 3.12). Zusätzlich zu den *Coldstart Nodes* dürfen weitere Steuergeräte als *Non-Coldstart Sync Nodes* betrieben werden. Diese Steuergeräte senden in ihrem jeweiligen *Keyslot* Botschaften mit aktivem *Sync Frame*, aber inaktivem *Startup Frame Indicator Bit*. Dadurch tragen sie zur Takt synchronisation bei, können das Netz aber nicht selbst starten.

Mit Protokollversion V3.0 neu eingeführt wurde das Verfahren *Time Trigger – Local Master TT-L*. Mit *TT-L* sind einfachere Netze möglich, bei denen nur noch ein einziges Steuergerät, der *Local Master*, für den Netzwerk-Start und die Zeitsynchronisation verantwortlich ist. Dieses Gerät sendet in zwei *Keyslots* Botschaften mit gesetzten *Startup Frame* und *Sync Frame Indicator Bits*. Für alle anderen Steuergeräte erscheint der Netzwerk-Start wie bei *TT-D* (Abb. 3.22). Da bei *TT-L* der Hochlauf des bei *TT-D* notwendigen zweiten *Coldstart-Knotens* entfällt, braucht das *TT-L* Netz im günstigsten Fall nur 6 statt 8 Kommunikationszyklen, bis das Netz gestartet ist.

Das neue *Time Trigger – External TT-E* Konzept soll komplexe Systeme ermöglichen, bei denen mehrere FlexRay-Busse synchron zueinander betrieben werden müssen. Eines der Bussysteme, die *Time Source*, gibt den Kommunikationszyklus vor und wird als klassisches *TT-D* Netz betrieben (Abb. 3.21). Das *TT-E* Bussystem, die *Time Sink*, wird über mindestens ein als *Time Gateway* wirkendes Steuergerät mit zwei FlexRay-Schnittstellen angekoppelt. Bezuglich des *Time Source* Netzes stellt das *Time Gateway* einen normalen FlexRay-Knoten mit oder ohne *Coldstart* und *Sync* Eigenschaften dar. Im *Time Sink* Netz dagegen übernimmt das *Time Gateway* die Rolle des *Coldstart* und *Sync* Knotens. Für die anderen Steuergeräte im *Time Sink* Netz wirkt der *TT-E* Netzwerk-Start wie bei einem *TT-L* Netz. Das *TT-E* Netz läuft bezüglich Kommunikationszyklen und *Makrotick* Takt mit einem festen Offset von 40 *Mikroticks* synchron zum *Time Source* Netz. Um die Verfügbarkeit

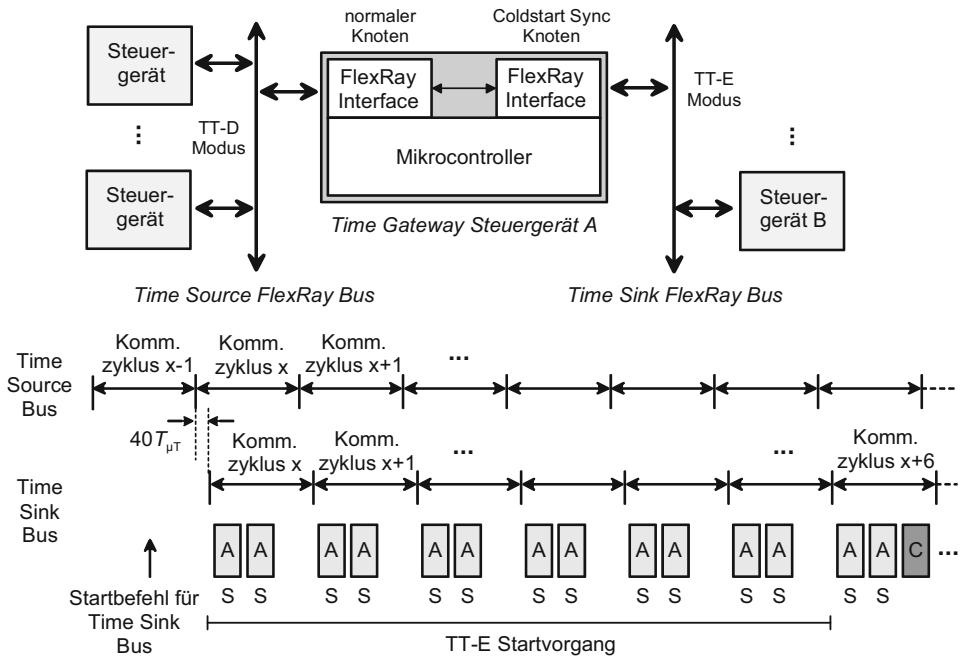


Abb. 3.31 Startvorgang eines TT-E FlexRay Netzes (S ... Startup und Sync Frame)

zu erhöhen, können auch mehrere Steuergeräte als *Time Gateways* zum selben *Time Source* Netz eingesetzt werden.

3.3.9 Zusammenfassung FlexRay – Layer 1 und 2

- Kommunikation zwischen mehreren Kfz-Steuergeräten zum Austausch von Mess-, Steuer- und Reglersignalen im Echtzeitbetrieb mit hoher Fehlersicherheit.
- Aufgrund der hohen Bandbreite auch als Backbone-Netz zu Kopplung von Gateways zwischen verschiedenen Teilnetzen einsatzbar.
- Maximal 64 Steuergeräte je Bussegment.
- Bitstrom-orientiertes Übertragungsprotokoll mit bidirektionaler Zwei-Draht-Leitung als Linien-Bus oder in Sternstruktur. Buslänge bis zum Sternpunkt max. 24 m.
- Ein- und zweikanalige Systeme möglich. Zweiter Kanal für redundante Übertragung in sicherheitskritischen Anwendungen oder zur Erhöhung der Bandbreite.
- Bitraten 2, 5 und 10 Mbit/s, theoretisch zukünftig auch höhere Bitraten möglich.
- FlexRay-Controller, Transceiver und Mikroprozessor notwendig. Buswächter (Bus Guardian) optional.

- Broadcast-System, bei dem die Sendeberechtigung und die Nachrichten festen Zeitschlitten innerhalb eines Kommunikationszyklus zugewiesen sind. Im statischen Teil des Kommunikationszyklus zeitsynchroner (TDMA) Buszugriff mit Zeitschlitten fester Länge, im dynamischen Teil modifiziertes TDMA-Verfahren (FTDMA) mit variablen Zeitschlitten und positionsabhängiger Arbitrierung (Teilnehmer mit früherem Zeitschlitz kann Teilnehmer mit späterem Zeitschlitz blockieren).
- Botschaften mit 0...254 Datenbytes und 8 Byte Header/Trailer. Zusätzliche Steuerbits auf der Bitübertragungsebene mit ca. 20 % Overhead. Nutzbare Datenrate < 500 ... 1000 KB/s je Kanal (bei 10 Mbit/s). Datenübertragung durch CRC-Prüfsummen gesichert, aber keine automatische Sendewiederholung bei Fehlern.

3.4 Media Oriented Systems Transport MOST

Während die bisher diskutierten Bussysteme alle für Steuer- und Regelaufgaben entwickelt wurden (*Control Bus*), ist MOST für Telematik- und Multimedia-Anwendungen (*Infotainment Bus*) konzipiert, d. h. für die Vernetzung von Autoradio, DVD-Wechsler, Autotelefon, Navigationssystem und Bordfernsehgerät [17, 18]. Bei diesen Anwendungen sind die Anforderungen an Echtzeitverhalten und Übertragungssicherheit geringer, die notwendigen Übertragungsbandbreiten aber höher als bei typischen Steuer- und Regelaufgaben. Um Sprache in Telefonqualität übertragen zu können, ist bei unkomprimierter Übertragung eine Nutzdatenrate von min. 8 KB/s notwendig, für nicht komprimierte Musiksignale in Stereo-CD-Qualität (zwei Kanäle mit 16 bit Abtastwerten bei 44,1 kHz Abtastfrequenz) min. 176 KB/s, also deutlich mehr als etwa mit CAN möglich ist. Komprimierte Audiosignale kommen bei Stereo mit 16 KB/s (MP3) bis 24 KB/s (AC3) aus, während für *Dolby Surround Sound* (AC3 5.1) bis zu 56 KB/s notwendig werden. Nach MPEG-1 bzw. -2 komprimierte Videosignale, wie sie z. B. auf DVDs üblich sind, benötigen bei Standard-Fernsehbildqualität um 1,5 MB/s.

Ziel eines Infotainment-Bussystems ist es, Audio- und Videodaten digital und damit störungsfrei zwischen verteilten Geräten zu übertragen. Als direkter Vorgänger von MOST gilt dabei der von der Firma Philips entwickelte Domestic Data Bus D2B. D2B kam jedoch nur in wenigen Fahrzeugen zum Einsatz, seine Weiterentwicklung wurde eingestellt. Neben D2B gab es anfänglich auch Systeme, die sich mit dem SPDIF Standard aus der Unterhaltungselektronik beholfen haben. Aufgrund der fehlenden Möglichkeit, mehrere Komponenten mit SPDIF intelligent zu verbinden, konnte sich dieser Ansatz aber nicht durchsetzen.

MOST war wie D2B ursprünglich die Entwicklung eines einzelnen Herstellers, in diesem Fall der Firma Oasis Silicon Systems (später SMSC, heute Microchip). Als die Weiterentwicklung von MOST auf Betreiben einiger Automobilhersteller 1998 einem Firmenübergreifenden Konsortium, der *MOST Cooperation*, übertragen wurde, setzte es sich rasch in europäischen Oberklassefahrzeugen durch, andere Hersteller reagierten zunächst zurückhaltend, setzen mittlerweile aber in einzelnen Fahrzeugen ebenfalls MOST ein. Dabei

spielt neben den guten Eigenschaften des eigentlichen Bussystems sicher eine wesentliche Rolle, dass MOST eine Spezifikation für die gesamten ISO/OSI-Schichten 1 bis 7 (siehe Kap. 2) bietet und frühzeitig funktionsfähige Buscontroller und Transceiver verfügbar waren. Die MOST-Kommunikationscontroller implementieren zusammen mit den zugehörigen Bus-Transceivern die ISO Protokollsichten 1 und 2, d. h. die Bit- und Datenübertragungsschicht, während die sogenannten *MOST Network Services Layer 1 und 2* die Dienste der ISO-Schichten 3 bis 7 nahezu vollständig abdecken.

Der öffentlich zugängliche Teil der MOST Spezifikationen ist allerdings an einigen Stellen sowohl beim *Physical Layer*, der Datenübertragungsschicht als auch bei den höheren Schichten lückenhaft bzw. nur den Mitgliedern der *MOST Cooperation* zugänglich. Dieses teilweise nicht veröffentlichte und mit Patenten belegte Implementierungs-Know-how wahrt wohl kommerzielle Interessen, behindert aber die schnellere Verbreitung des Standards und sorgt für Verunsicherung, wenn wesentliche Systemkomponenten von einem einzigen Hersteller abhängig sind. Das Konsortium hat angekündigt, dass die Pioniere der MOST-Technologie, die über die entscheidenden Patentrechte verfügen, zukünftig eine offenere Lizenzpolitik betreiben wollen.

Gleichzeitig werden immer öfter Schnittstellen zu *Consumer*-Geräten wie MP3-Playern und Mobiltelefonen gefordert. Neben der Funktechnologie *Bluetooth* müssen USB-Schnittstellen zu Mobiltelefonen und MP3-Playern bereitgestellt und das Internet integriert werden (*Customer Convenience Port*). Daneben werden höhere Bitraten für neue Fahrer-Assistenzsysteme gefordert, wenn etwa Live-Videosignale von einer hochauflösenden Rückfahrkamera übertragen werden sollen. Im Gegensatz zu DVDs, bei denen die Bilder komprimiert gespeichert und übertragen werden, reicht die typische Rechenleistung eingebetteter Systeme in der Regel nicht aus, derartige Signale vor der Übertragung in Echtzeit zu komprimieren.

Nachdem eine Zeit lang überlegt wurde, das für den Automobileinsatz als *IDB 1394* schon standardisierte *Firewire* einzusetzen, gilt mittlerweile *Ethernet* als Favorit, weil damit praktisch alle Internet-tauglichen Geräte ins Fahrzeugnetz eingebunden werden können. Wenn aber *Ethernet* im Fahrzeug und für die Diagnoseschnittstelle ohnehin vorhanden ist, wäre es sinnvoll, wenn es MOST ersetzen und auch gleich die Aufgaben des Infotainment-Netzes übernehmen würde. Vereinzelt wird *Ethernet* im Forschungs- und Vorentwicklungsreich sogar für den Echtzeiteinsatz, d. h. als Alternative zu FlexRay oder CAN, evaluiert (Abschn. 3.5). Die *MOST Cooperation* sieht sich hier in einem möglichen Verdrängungswettbewerb und versucht gegenzuhalten. Helfen sollen höhere Bandbreiten wie MOST150 mit 150 Mbit/s und zukünftig noch mehr, kostengünstigere Busverkabelungen und das Konzept, MOST als *Physical Layer* für *Ethernet*-Datenpakete einzusetzen.

Nicht vergessen werden darf in dieser Diskussion, dass MOST ein etabliertes, im Automobileinsatz ausgereiftes Bussystem ist. *Ethernet* hat sich zwar im Bürobereich seit vielen Jahren bewährt, für den Automobileinsatz mit seinen schwierigen Randbedingungen für den *Physical Layer* und seine Anforderungen an Echtzeitverhalten und Übertragungssicherheit muss es aber erst einmal fahrzeugauglich gemacht werden.

3.4.1 Bus-Topologie und Physical Layer

Die erste und immer noch eingesetzte Generation der MOST Systeme (MOST25) basiert auf einem optischen Übertragungsmedium mit Kunststoff-Lichtwellenleitern (Plastic Optical Fiber POF). Das modulierte Lichtsignal überträgt die Daten Manchester-codiert, wodurch auf der Empfängerseite eine Bittaktsynchronisation möglich wird. Im Steuergerät wird der Lichtstrom von einem optischen Transmitter (*Fiber Optical Transmitter FOT*) empfangen bzw. wieder ausgesendet, zur Weiterverarbeitung im Kommunikationscontroller aber in elektrische Form umgewandelt. Neben der optischen gibt es mittlerweile auch eine elektrische Variante mit einer verdrillten Zwei-Draht-Leitung, die im Zusammenhang mit der zweiten Generation MOST50 eingeführt wurde. Als Weiterentwicklung steht MOST150 mit 150 Mbit/s wieder auf optischer Basis bereit. Zusätzlich gibt es für MOST150 auch eine elektrische Variante, die allerdings mit im Vergleich zu MOST50 teureren Koaxialkabeln arbeitet. Das MOST Übertragungsprotokoll selbst ist unabhängig vom physikalischen Übertragungsmedium.

MOST lässt unterschiedliche Bus-Topologien zu. Am häufigsten eingesetzt wird eine logische Ring-Struktur mit bis zu 64 Steuergeräten (Abb. 3.32). Physikalisch betrachtet sind die Verbindungen zwischen den einzelnen Steuergeräten unidirektionale Punkt-zu-Punkt-Verbindungen, d. h. das optische Signal wird in jedem Steuergerät regeneriert, bevor es zum nächsten Steuergerät weitergesendet wird. Jedes Steuergerät enthält deshalb genau einen Eingang und einen Ausgang. Inaktive Steuergeräte leiten die ankommenden Daten unverändert weiter (Bypass-Betrieb), aktive Geräte entnehmen Daten oder fügen eigene Daten in den Datenstrom ein. Ein vorher festgelegtes Gerät arbeitet als Master-Steuergerät (*Timing Master*) und erzeugt die Botschaften (*Frames*). Die übrigen Steuergeräte (*Slaves*) synchronisieren sich auf den Bit- und Frame-Takt des Masters und entnehmen aus den im Ring umlaufenden Botschaften Empfangsdaten bzw. fügen Sendedaten ein. Das Bussystem arbeitet in der Regel mit einer Bitrate von etwa 25 Mbit/s (MOST25), neuere Buscontroller auch mit 50 Mbit/s (MOST50) und 150 Mbit/s (MOST150). Die Durchlaufverzögerung eines Steuergerätes bei MOST25 beträgt 2 Frames, d. h. etwa 45 μ s (siehe unten), bei MOST50 und MOST150 liegt sie unter 1 μ s.

3.4.2 Data Link Layer

MOST verwendet eine bitstromorientierte Übertragung, bei der 16 Botschaften, hier Frames genannt, zu einem Block zusammengefasst werden (Abb. 3.33). Jede Botschaft durchläuft genau einmal den gesamten Ring. Die Botschaftsraten wird meist auf 44,1 kHz eingestellt und entspricht damit der Abtastrate von Audio-CDs. Daraus ergibt sich eine Botschaftslänge von 22,67 μ s. Alternativ sind 48 kHz möglich. Diese bei DVD-Audio Player oder DAT-Geräte übliche Rate wird seit der Spezifikation 3.0 empfohlen. Die Botschaftsraten muss für das gesamte Netz einheitlich sein. Weicht die interne Abtastrate eines Gerätes von der Botschaftsraten des Bussystems ab, muss das Gerät die Abtastrate konvertieren. MOST

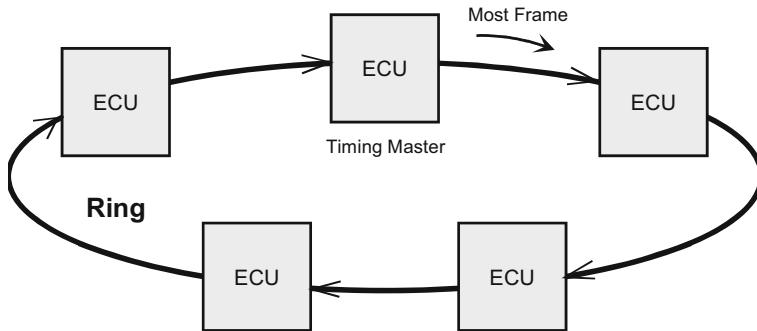


Abb. 3.32 MOST-System in Ring-Struktur

ist grundsätzlich als synchrones System konzipiert, d. h. es wird in kontinuierlichen Taktzyklen immer die gleiche Anzahl an Bytes übertragen. Das Botschaftsformat wurde aber so flexibel gewählt, dass auch asynchrone Übertragungen innerhalb dieser synchronen Struktur möglich sind.

3.4.2.1 MOST25 Frames

Jeder Frame beginnt mit dem Header, der sich aus einer *Präambel* (4 bit) und dem sogenannten *Boundary Descriptor* (4 Bit) zusammensetzt (Abb. 3.33). Die *Präambel* kennzeichnet den Start eines Frames und ermöglicht eine Neusynchronisation auf den an kommenden Bitstrom. Der Blockanfang wird durch eine abweichende *Präambel* im ersten der 16 Frames markiert. Der *Boundary Descriptor* unterteilt das nachfolgende Datenfeld in einen synchronen und einen asynchronen Bereich. Sein Wert bestimmt die Anzahl der synchronen Daten in Vielfachen von 4 Byte (*Quadlets*), wobei die Mindestlänge des synchronen Datenfeldes 24 Byte beträgt. Der Rest der insgesamt 60 Bytes des Datenfelds, d. h. maximal 36 Byte, wird für asynchrone Datenpakete verwendet. Der *Boundary Descriptor* ist für alle Frames gleich und wird im laufenden Betrieb nicht verändert. Auf das Feld mit den synchronen und asynchronen Daten folgt ein 2 Byte großes Feld für Steuerbotschaften. Das abschließende Trailer-Byte enthält weitere Steuer- und Statusinformationen und bietet eine Überprüfung auf Übertragungsfehler. Als Bitfehlerrate wird ein Wert von durchschnittlich 10^{-10} genannt.

3.4.2.2 MOST50 Frames

MOST50 arbeitet mit derselben Frame-Rate und Block-Struktur wie MOST25, aber der doppelten Bitrate. Dadurch passen 128 Byte in einen Frame. Der Header umfasst nun 7 Byte, gefolgt von dem auf 4 Byte verdoppelten Feld für Steuerbotschaften, das in der Dokumentation meist als Teil des dann 11 Byte langen Headers dargestellt wird. Das jetzt 117 Byte lange Datenfeld kann über den *Boundary Descriptor* beliebig zwischen synchronen und asynchronen Daten aufgeteilt werden. Im Gegensatz zu MOST25, wo eine Änderung der Aufteilung nur durch Abbruch und Neuaufbau der synchronen Kommu-

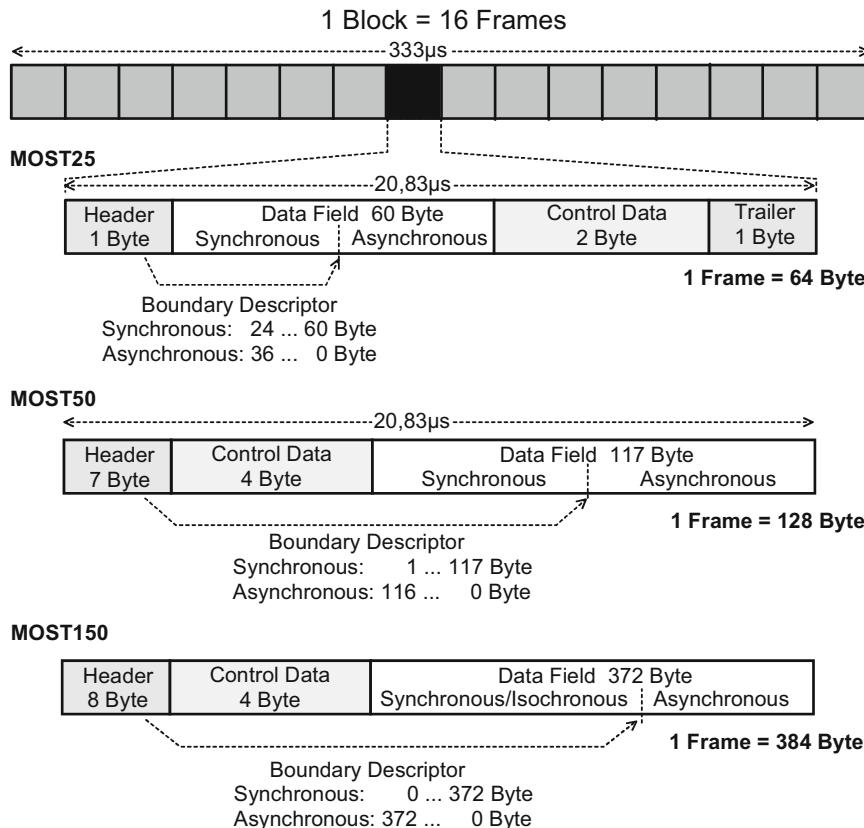


Abb. 3.33 MOST-Botschaftsformat Block und Frame (Frame-Rate 48 kHz)

nikation möglich ist, lässt MOST50 ausdrücklich eine im laufenden Betrieb dynamisch veränderliche Aufteilung zu.

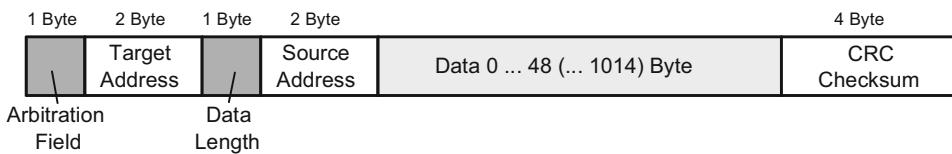
3.4.2.3 MOST150 Frames

MOST150-Frames sind, mit Ausnahme des um ein Byte längeren Headers, aufgebaut wie MOST50-Frames. Wegen des höheren Bittaktes bei gleicher Frame-Dauer passen bis zu 372 Datenbytes in einen Frame.

Entsprechend der Aufteilung der Datenfelder in einem Frame unterscheidet MOST zwischen drei voneinander unabhängigen Gruppen von Datenkanälen für unterschiedliche Anwendungszwecke:

- Synchrone Daten (*Streaming Data* synchron oder isochron), z. B. Audio, Video,
- Asynchrone Daten (*Packet Data*), z. B. TCP/IP Pakete,
- Steuerdaten (*Control Channel*), z. B. Netzmanagement, Gerätekommunikation.

MOST25 und MOST50



MOST150

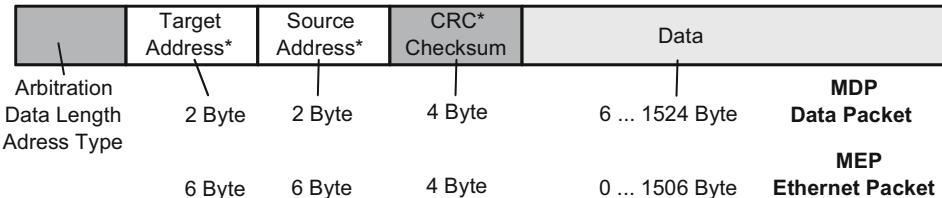


Abb. 3.34 MOST-Datenpakete für asynchrone Daten (Packet Data Channel PDC)

Bevor auf diese drei Gruppen genauer eingegangen wird, ist allerdings eine Vorbemerkung zu dem hier häufig verwendeten deutschen Begriff *Botschaft* notwendig. Auf der Bitübertragungsschicht werden die drei Gruppen von Daten in einer gemeinsamen *physikalischen Botschaft* übertragen, in der MOST-Spezifikation mit dem englischen Begriff *Frame* bezeichnet. Innerhalb eines *Frames* besitzt jede Gruppe Datenfelder mit fester oder variabler Länge. Bei den asynchronen Daten bzw. den Steuerdaten sind diese Datenfelder nur Teile größerer *logischer Botschaften*, die in der Spezifikation als *Asynchronous Data Packet* (manchmal auch *Frame* statt *Packet*, Abb. 3.34) bzw. *Control Data Frame* (Abb. 3.35) bezeichnet werden. Im vorliegenden Text werden diese logischen Botschaften in der Regel als Datenpakete, gelegentlich aber auch einfach als Botschaften bezeichnet.

3.4.2.4 Synchroner Datenbereich (Streaming Data)

Der *synchrone Datenbereich* wird in Zeitschlitz zu je 8 Bit eingeteilt, bei MOST als physikalische Kanäle bezeichnet. Mehrere dieser physikalischen Kanäle können dann zu einem logischen Kanal zusammengefasst werden. Ein einzelner logischer Kanal (*Streaming Channel*) kann dabei zwischen einem Byte und der Gesamtgröße des synchronen Datenbereichs verwenden. Der Buszugriff auf die Zeitschlitz erfolgt zeitsynchron, d. h. nach dem TDMA-Verfahren. Anwendungen fordern exklusive Kanäle an und können über diese Kanäle dann solange Daten übertragen, bis sie die Kanäle wieder freigeben. Das Format der Daten innerhalb dieser Kanäle ist beliebig. Bei Musiksignalen von Audio-CDs beispielsweise werden direkt die auf der CD gespeicherten 16 bit Abtastwerte übertragen. Für ein Stereo-Signal müssen dazu im synchronen Datenbereich vier physikalische Kanäle allokiert werden, jeweils zwei für das rechte und zwei für das linke Audio-Sample. Wird das gesamte Datenfeld ausgenutzt (bei MOST25 60 Byte, bei MOST50 117 Byte, bei MOST150 372 Byte), können 15, 29 bzw. 93 unkomprimierte Audio-Stereo-CD-Kanäle übertragen werden. Synchron

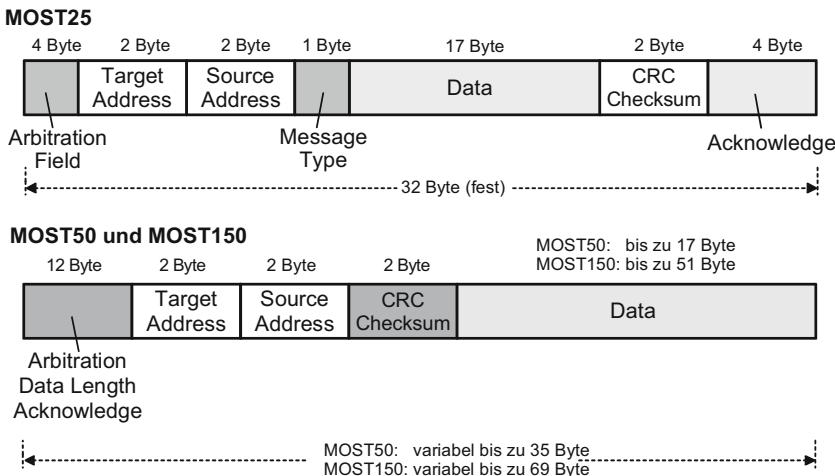


Abb. 3.35 MOST-Datenpakete für Steuerdaten (*Control Data*)

Daten enthalten weder Sender- noch Empfängeradresse, die Administration erfolgt vollständig über die Steuerdaten (*Control Channel*). Genutzt wird der synchrone Datenbereich vor allem für Audio- und Videodaten, deren Übertragung eine hohe und garantierte Bandbreite benötigt.

Während einfache Audiodaten eine feste Abtastrate haben, die jedoch von der MOST Abtastrate abweichen kann, verwenden komprimierte Daten, z. B. MPEG-kodierte Videoströme, oft eine variable Datenrate, benötigen aber dennoch eine garantierte Bandbreite. Dazu müssen die Daten vor der Übertragung entweder auf die MOST- Abtastrate umkodiert werden oder die Frames müssen dynamisch mit Dummy-Daten aufgefüllt werden. Bei MOST25/50 musste dies von der Anwendung erledigt werden, bei MOST150 wurden zusätzlich sogenannte *Isochrone Kanäle* eingeführt, bei denen dies vom MOST Kommunikationscontroller (INIC) übernommen wird.

3.4.2.5 Asynchroner Datenbereich (Packet Data)

Der *asynchrone Datenbereich* ist für formulierte Datenpakete vorgesehen (Abb. 3.34). Da je Frame nur eine begrenzte Zahl von Bytes für asynchrone Daten zur Verfügung steht, wird die Botschaft vom Kommunikationscontroller gegebenenfalls auf mehrere Frames aufgeteilt. Sender und Empfänger der Botschaft werden durch Geräteadressen identifiziert. Der Buszugriff erfolgt bei diesen Botschaften, sobald der asynchrone Datenbereich nicht durch ein anderes asynchrones Datenpaket belegt ist. Zur Erkennung dient das *Arbitration Field*. Details der Arbitrierung sind nicht offengelegt.

Standardmäßig können MOST25-Kommunikationscontroller in einem asynchronen Datenpaket maximal 48 Byte Nutzdaten versenden. Die Datenlänge wird im Feld *Data Length* in Vielfachen von 4 Byte (*Quadlet*) angegeben. Falls die Applikation, die den Kom-

munikationscontroller bedient, die Daten auf der Sendeseite schnell genug liefern bzw. auf der Empfangsseite schnell genug auslesen kann, lässt sich das Datenfeld bei MOST25 auf bis zu 1014 Byte erweitern. Bei MOST50-Systemen ist diese Länge der Standard. Asynchrone Botschaften werden im Kommunikationscontroller durch eine CRC-Prüfsumme gesichert. Die Spezifikation weist auf einen Bestätigungsmechanismus und eine automatische Sendewiederholung im Fehlerfall hin.

Bei MOST150 wurde das Format der Paketdaten verändert und zwei Formatvarianten eingeführt. Beim *MOST Data Packet MDP* Format wird weiterhin die bisherige 16 bit Gerätadressierung verwendet, es können aber jetzt bis zu 1524 Datenbyte übertragen werden. Das *MOST Ethernet Packet MEP* Format ist für das Durchleiten von Ethernet-Botschaften vorgesehen, ohne dass diese wesentlich umformatiert werden müssen. Daher werden für die Adressierung die bekannten 48 bit langen Ethernet MAC-Adressen und der bekannte Ethernet CRC statt der MOST-spezifischen Prüfsumme verwendet. Das Datenfeld kann in diesem Fall bis zu 1506 Byte übertragen.

Der asynchrone Datenbereich wird häufig für die Übertragung von Karteninformationen in Navigationssystemen oder für die Durchleitung von TCP/IP-Verbindungen eingesetzt, da hier kurzzeitig hohe Datenmengen anfallen können.

3.4.2.6 Steuerdaten (Control Data)

Steuerdaten dienen zum einen der Verwaltung des Netzwerkes, zum anderen aber auch der Kommunikation zwischen den Anwendungen in den verschiedenen Steuergeräten im Ring. Die Übertragung findet ereignisorientiert im sogenannten *Control Channel (Kontrollkanal)* statt und bietet nur eine geringe Bandbreite. Die Steuerbotschaft hat bei MOST25 eine feste Länge von 32 Byte (Abb. 3.35). Diese 32 Byte werden beginnend mit dem *Arbitration Field* in jeweils 2 Byte große Einheiten zerstückelt und auf die *Control Data* Felder der 16 aufeinander folgenden Frames eines Blocks verteilt (Abb. 3.33). Die ersten beiden *Arbitration Bytes* der Steuerbotschaft nach Abb. 3.35 werden also im *Control Data* Feld des ersten Frames eines Blocks nach Abb. 3.33, die weiteren zwei *Arbitration Bytes* im *Control Data* Feld des zweiten Frames, die *Target Address* im *Control Data* Feld des dritten Frames übertragen usw. Die Definition eines Blocks ist nur für den Kontrollkanal relevant, da bei MOST25 pro Block genau eine Steuerbotschaft übertragen wird. Die synchronen und asynchronen Datenkanäle dagegen ignorieren Blockgrenzen.

Der Buszugriff auf den Kontrollkanal erfolgt asynchron und prioritätsbasiert, d. h. nach einem CSMA-Verfahren, wobei im Arbitrierungsfeld die Priorität der Botschaft angegeben wird. Steuerbotschaften werden durch eine Prüfsumme und eine Bestätigungsmechanismus geschützt. Im Fehlerfall kommt es zur automatischen Sendewiederholung (*Low Level Retries*). Eine erfolgreiche Übertragung wird vom Empfänger durch eine positive Bestätigung (*Acknowledge*, kurz *ACK*) quittiert. Findet sich kein Abnehmer für eine Botschaft, z. B. durch falsche Adressierung oder wenn der Empfänger keine Puffer zur Zwischenspeicherung frei hat, so interpretiert der Sender dies als ein *Not Acknowledge*, kurz *NAK*.

Im *Target* bzw. *Source Address* Feld wird eine 16 Bit lange Empfänger- bzw. Senderadresse angegeben. Ein einzelnes Steuergerät kann über seine physikalische oder logische

Adresse angesprochen werden. Die physikalische Adresse berechnet sich aus der relativen Position des Steuergerätes im Ring bezogen auf das Master-Steuergerät und wird meist nur beim Systemstart verwendet. Das Master-Steuergerät besitzt stets die physikalische Adresse 400_H , alle nachfolgenden Slave-Steuergeräte erhalten darauf basierend eine um jeweils eins erhöhte Adresse. Die logische Adresse kann individuell vom *Network Master* pro Teilnehmer vergeben werden, muss aber wie die physikalische Adresse ebenfalls im Ring eindeutig sein. Üblicherweise beginnt man hier mit 100_H für den *Network Master* und inkrementiert dann ebenfalls um eins für jedes folgende Gerät. Dieses Verhalten ist jedoch nicht standardisiert. Außerdem besteht die Möglichkeit, eine Steuerbotschaft an eine Gruppe von zusammengehörenden Geräten (*Groupcast*, typisch *Target Address* $300_H + FBlockID$) oder an alle Geräte im Ring (*Broadcast*, typisch *Target Address* $3C8_H$) zu schicken.

Das Feld *Message Type* charakterisiert die Art der Steuerbotschaften. Neben normalen Botschaften, deren Bedeutung auf Anwendungsebene definiert wird, stehen vordefinierte Steuerbotschaften für die Netzwerkverwaltung zur Verfügung. Die *Resource Allocation* und *Resource Deallocation* Botschaften beispielsweise dienen der Reservierung und Freigabe von synchronen Übertragungskanälen. Mit *Remote Read* und *Remote Write* Botschaften kann ein Kommunikationscontroller, z. B. derjenige des Master-Steuergerätes, die Register und damit die Konfiguration eines anderen Kommunikationscontrollers auslesen oder verändern. Über *Remote Get Source* Botschaften kann abgefragt werden, welches Steuergerät Daten in einem bestimmten synchronen Datenkanal sendet.

Die Formatierung der Steuerbotschaften erfolgt selbstständig durch den MOST Kommunikationscontroller. Dieser überprüft auch die Empfängeradresse *Target Address* und die *CRC Checksum* auf Richtigkeit, bevor die Nutzdaten der Steuerbotschaft an die nächst höhere Schicht des Empfängers weitergereicht werden.

Bei MOST50 und MOST150 werden je Frame 4 Byte der Steuerbotschaft übertragen (Abb. 3.35). Das Format der Steuerbotschaft hat sich gegenüber MOST25 geändert. Ihre Länge ist jetzt variabel mit bis zu 35 bzw. 69 Byte, die auf mehrere Frames verteilt werden. Blockgrenzen spielen bei MOST50 und MOST150 keine Rolle mehr.

3.4.2.7 Arbitrierung und Bandbreite

Während im synchronen Datenbereich jeder Sender die notwendige Übertragungsbandbreite fest für sich reservieren kann, konkurrieren die verschiedenen Steuergeräte um die Bandbreite bei den asynchronen und den Steuerdaten. Bei den asynchronen Botschaften wird die Sendeberechtigung mittels des Arbitrierungsfelds von einem Gerät zum anderen weitergereicht (*Token Passing*).

Bei den Steuerdaten erfolgt der Buszugriff über einen, wie die MOST-Spezifikation es ausdrückt, *fairen CSMA-Arbitrationsmechanismus*. Dabei spielt die Priorität der Botschaften eine Rolle, die zwischen 0 (nieder) und 15 (hoch) gewählt werden kann. Aufgrund der geringen Anzahl von Prioritätsstufen sind Konflikte möglich, zumal Steuerbotschaften bei gängigen Kommunikationscontrollern oft mit der Default-Priorität 1 versendet werden. Wie das Token Passing oder die Arbitrierung im Detail erfolgen und welche Worst Case Latenzen sich dabei ergeben, lässt der öffentlich zugängliche Teil der Spezifikation

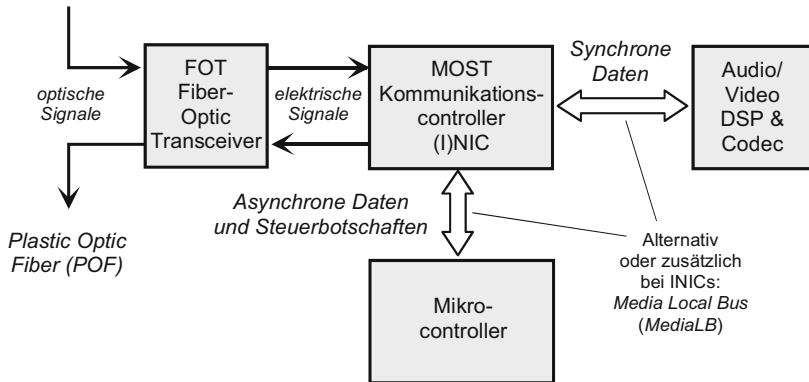


Abb. 3.36 Typischer Aufbau eines MOST-Steuergerätes

allerdings weitgehend im Dunklen. Spezifiziert ist lediglich, dass die Steuerbotschaft mit der höchsten Priorität *gewinnt* und ein einzelner Kommunikationscontroller beim Senden nicht in jedem Frame senden darf.

Die Bestimmung der Bandbreite für die Übertragung ist relativ komplex. Neben der Bitrate, die die Länge der einzelnen Botschaften beeinflusst, sowie der Abtastrate spielt die Aufteilung in synchrone und asynchrone Kanäle eine wichtige Rolle. Ausführliche Beispieldurchrechnungen finden sich in [17].

3.4.3 Kommunikationscontroller

Bei MOST handelt es sich um eine Technologie, die spezifisch für Infotainment-Anwendungen ausgelegt ist und immer noch hauptsächlich in den oberen und mittleren Fahrzeugklassen eingesetzt wird. Dieser geringe Verbreitungsgrad spiegelt sich direkt in einem begrenzten Angebot der Halbleiterhersteller für Kommunikationsbausteine wieder. Während bei CAN der Kommunikationscontroller typischerweise als On-Chip-Modul in vielen Mikrocontrollern bereits integriert ist, ist der Kommunikationscontroller bei MOST meist ein externer Baustein, der über verschiedene Schnittstellen mit anderen Prozessoren im Steuergerät kommuniziert (Abb. 3.36). Aufgrund der unterschiedlichen Frame-Formate sind dabei MOST25, MOST50 und MOST150 Kommunikationscontroller nicht abwärts kompatibel.

Die ersten verfügbaren MOST Kommunikationscontroller, auch als *Network Interface Controller NIC* bezeichnet, implementierten den kompletten Data Link Layer und kanalisierten die Datenströme zu den entsprechenden Datenquellen und Datensenken. In der Regel werden der asynchrone Datenkanal und der Kontrollkanal zusammen über eine Schnittstelle (Parallelbus oder I²C) an den steuernden Mikrocontroller des Gerätes übertragen. Die geforderte Bandbreite hierbei ist eher gering. Die Audio- und Videoinhalte in

den synchronen Datenströmen dagegen benötigen eine hohe Bandbreite und werden über eine separate Schnittstelle, als I²S oder *Streaming Port* bezeichnet, an spezielle Chips zur Signalverarbeitung z. B. MPEG Dekoder, übertragen.

Der gesamte MOST Datenstrom wird vom Kommunikationscontroller empfangen und danach mit einer kurzen Verzögerung auf dem Ring weitergegeben. Während dieser Verzögerung fügt der Kommunikationscontroller zu sendende Daten in den empfangenen Frame ein bzw. extrahiert Daten, die für das lokale Steuergerät bestimmt sind. Für den Kontrollkanal und die asynchronen Daten entscheidet die Zieladresse der jeweiligen Botschaft. Für die synchronen Daten werden in einer *Routing Tabelle* die Kanäle und Zeitschlitzte definiert, in welchen empfangen oder gesendet werden soll.

Der Datenaustausch zwischen der Infotainment-Anwendung auf dem Mikrocontroller und dem Kommunikationscontroller wird von den sogenannten *Funktionsblöcken* und *Network Services* gekapselt. Diese waren ursprünglich als reine Softwareschicht im Mikrocontroller konzipiert, während der Kommunikationscontroller selbst eine klassische Schnittstelle mit Steuer- und Datenregistern hatte. Die *Network Services* stellen eine Schnittstelle zur Verfügung, die direkt auf dem MOST-Nachrichtenformat mit *FBlockID*, *FktIDs* usw. (siehe unten) basiert. Die Einstellparameter des Kommunikationscontrollers werden ebenfalls über spezielle Funktionsblöcke zugänglich gemacht.

Neuere Generationen von Kommunikationscontrollern, sogenannte *Intelligent Network Interface Controller INIC*, enthalten bereits die Funktion des *Network Service Layer* und entlasten damit den Mikrocontroller. Dadurch wird auch der Übergang von MOST25 zu den neueren Generationen erleichtert, weil die INIC API unverändert bleiben soll. Eine weitere Neuerung der INIC-Generation ist die Einführung des *Media Local Bus (MediaLB)*, der die traditionellen Schnittstellen zur Anbindung des Mikrocontrollers, DSPs, Codecs usw. ablösen und die volle MOST50/150-Bandbreite auch steuergeräteintern unterstützt.

3.4.4 Network Services und Funktionsblöcke

Anders als die bisher vorgestellten Bussysteme, bei denen die höheren Protokollsichten in der Regel gar nicht oder erst viel später als der Physical und der Data Link Layer und oft nicht durchgängig festgelegt wurden, spezifiziert MOST zusätzlich Mechanismen auf höherer Ebene. Die Schnittstelle zu diesen höheren Schichten wird von den sogenannten *MOST Network Services* (kurz *NetServices*) gebildet, die von MOST in *Layer 1* und *Layer 2* unterteilt werden. Grob kann man *MOST Layer 1* den Schichten 3 bis 5 des ISO/OSI-Schichtenmodells und *Layer 2* der Schicht 6 zuordnen (Abb. 3.37). Auf den *Network Services* setzen die *Funktionsblöcke* auf, die eine objektorientierte Programmierschnittstelle für die Infotainment-Anwendungen darstellen und der ISO/OSI-Schicht 7 zuzuordnen sind.

Die MOST-Spezifikation definiert die *Network Services* als eine Reihe von Funktionen, mit denen die Zugriffe auf den Kommunikationscontroller und das Netz durchgeführt werden. Einige Hersteller bieten zu ihren Kommunikationscontrollern eine passende Software-Implementierung dieser Dienste an. Neuere MOST Kommunikationscontroller

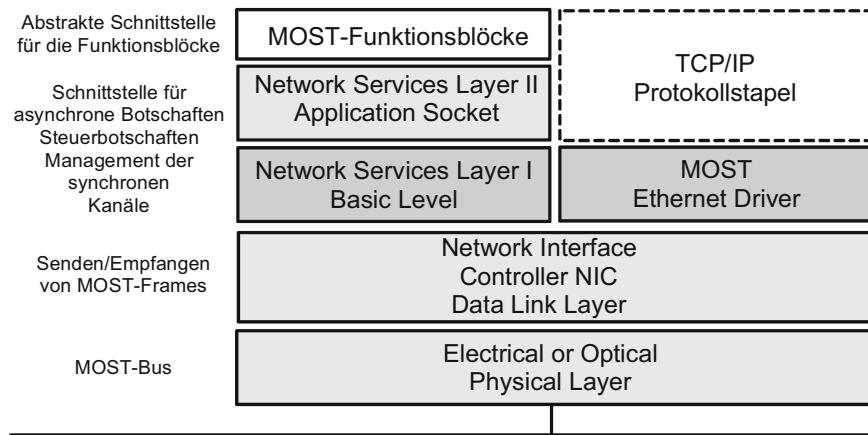


Abb. 3.37 MOST-Protokollstapel mit Erweiterung für Ethernet bei MOST150

(INIC) enthalten bereits eine Hardware-Implementierung der *Layer I*-Dienste. Die *Network Services* verwalteten alle drei Datenkanalgruppen. Für die asynchronen Daten und die Steuerbotschaften stellen die *Network Services* logische Protokollebenen bereit, während synchrone Datenkanäle lediglich verwaltet und die Nutzdaten an die anwendungsspezifische Signalverarbeitung weiterleitet werden.

Die *Layer I*-Dienste umfassen folgende Funktionsgruppen:

- *Synchronous Channel Service SCS* bzw. *Socket Connection Manager SCM*: Reservierung und Freigabe synchroner Kanäle, Konfiguration der Schnittstelle für synchrone Daten (siehe Abb. 3.36). Neuere Netzwerkcontroller (INIC) kapseln das Management synchroner Botschaften durch sogenannte *Sockets*, während bei älteren Kommunikationscontrollern (NIC) die Reservierung und Freigabe von Kanälen und die Verbindungen zur Signalquelle (*Source*) und Signalsenke (*Sink*) explizit durchgeführt werden mussten (siehe Abschn. 3.4.7).
- *Asynchronous Data Transmission Service ADS*: Senden und Empfangen von asynchronen Daten inklusive Zwischenspeicherung und Fehlerbehandlung.
- *Control Message Service CMS*: Senden und Empfangen von Steuerdaten.
- *Application Message Service AMS*: Senden und Empfangen über den Kanal für Steuerdaten mit Segmentierung, falls mehr als 17 Byte Nutzdaten notwendig sind.
- *MOST Transceiver* bzw. *Processor Control Service MCS*: Konfiguration des Kommunikationscontrollers, Setzen des *Boundary Descriptors* und der Frame-Rate.
- *MOST Supervisor MSV*: Konfiguration als *Timing Master* oder *Slave*, Power Management, Netzwerk Start und Stopp (siehe Abschn. 3.4.5).

Zu den *Layer II*-Diensten gehören:

- *Command Interpreter CMD*: Verarbeitung von Botschaften mit Steuerdaten und Aufruf von Anwenderfunktionen für deren Bearbeitung.
- *Address Handler AH* und *Network Master Shadow*: Zugriff auf die Kopie von Datenstrukturen des *Network Masters* wie der *Central Registry* (siehe Abschn. 3.4.5) zur Verwaltung des Netzwerks.
- *Notification Services NTFS*: Automatischer Benachrichtigungsmechanismus, wenn sich Parameter (*Properties*) eines durch einen Funktionsblocks verwalteten Gerätes ändern (siehe unten).

Über den *Layer II*-Diensten liegen die sogenannten Funktionsblöcke, die wie alle übrigen MOST-Objekte mit Hilfe von Kennziffern referenziert werden. Bei den Funktionsblöcken wird diese Kennziffer als *FblockID* bezeichnet. Da einige Funktionsblöcke, z. B. die Diagnose, mehrfach in einem System vorkommen können, muss zur eindeutigen Kennzeichnung zusätzlich zur Funktionsblock-Kennziffer auch eine Instanznummer *InstID* angegeben werden. Die Gruppe der Audio-Funktionsblöcke beispielsweise umfasst unter anderem den Audio-Verstärker (*FBlockID*=22h), die Freisprecheinrichtung (*Hands-free Processor FBlockID*=28h) oder die Audiosignalverarbeitung mit Lautstärkecontroller, Equalizer usw. (*Audio DSP FBlockID*=21h), während das Bedienpanel zur Funktionsgruppe *Human Machine Interface* (*FBlockID*=10h) gehört. Jedes Steuergerät bietet eine bestimmte Anzahl an Funktionsblöcken für den jeweiligen Aufgabenbereich an. Die eigentliche Infotainment-Funktion, beispielsweise das Abspielen einer CD, wird durch das Zusammenspiel mehrerer solcher Funktionsblöcke realisiert, wobei es für die Anwendung gleichgültig ist, ob die Funktionsblöcke im selben Gerät enthalten oder über das Netz verteilt sind, da die Kommunikation zwischen den Funktionsgruppen über die *Network Services* gekapselt wird.

Neben den für die eigentliche Infotainment-Anwendung wichtigen Funktionsblöcken muss jedes Gerät den Funktionsblock *NetBlock* (*FBlockID*=01h), das Master-Steuergerät die Blöcke *NetworkMaster* bzw. *ConnectionMaster* (*FBlockID*=02h und 03h) implementieren, in denen der Zugriff auf den Bus und das Netzmanagement enthalten ist. Weiterhin kann man davon ausgehen, dass der Funktionsblock *Diagnosis* (*FBlockID*=06h) in jedem Steuergerät zu finden ist, über den häufig auch höhere Protokolle wie KWP 2000 oder UDS implementiert werden. Für diejenigen Steuerbotschaften, die zur Koordination des Busbetriebs zwischen den Kommunikationscontrollern versendet werden, existiert zusätzlich der Pseudoblock *GeneralFBlock* mit *FBlockID*=00h. Neben den von MOST definierten Funktionsblöcken gibt es auch einen Bereich, den die Fahrzeughersteller für proprietäre Zwecke nutzen können, z. B. zur Systemüberwachung oder für den Software-Download.

Infotainment-Anwendung sind typischerweise über mehrere Steuergeräte verteilt, z. B. Bediengerät, Radioempfänger, Vorverstärker und aktive Lautsprecher, die über MOST vernetzt sind. Die örtliche Verteilung der Funktionsblöcke und die zwischen ihnen und der Anwendung notwendige Kommunikation über das Bussystem werden durch die *Net-*

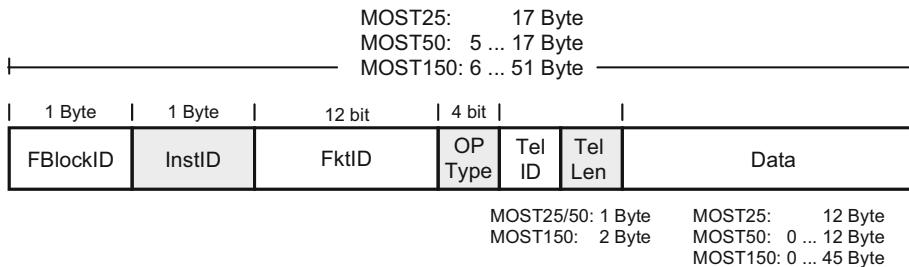


Abb. 3.38 Nutzdatenbereich einer Steuerbotschaft für eine *Network Service* Funktion

work Services gekapselt. Diese Anwendungen rufen die Funktionen einfach mit Hilfe von *FBlockID* und *FktID* auf, unabhängig davon, in welchem Gerät diese Funktionen implementiert sind. Die Auflösung der Netzwerkadressen sowie das Versenden der notwendigen Busbotschaften werden von den *Network Services* übernommen. Dazu werden die Funktionen und ihre Parameter direkt auf den Nutzdatenbereich der Steuerbotschaften abgebildet (Abb. 3.38).

Jeder Funktionsblock stellt eine Reihe von Funktionen (*FktID*) bereit. MOST unterscheidet dabei zwischen *Properties* und *Methods*. *Properties* sind Zustände oder Eigenschaften, deren aktueller Wert durch eine entsprechende *Network Service* Funktion abgefragt oder unmittelbar, d. h. ohne wesentliche zeitliche Verzögerung, verändert werden kann. Ein typisches Beispiel wäre die Lautstärke eines Audioverstärkers. *Methods* sind Aktionen, die durch eine entsprechende *Network Service* Funktion angestoßen werden, dann aber längere Zeit für ihre Ausführung benötigen, bevor ein Ergebnis vorliegt, etwa der Sendersuchlauf des Radioempfängers. Detaillierte Angaben, wie die Funktion ausgeführt werden soll, erfolgen über *OpType*. Typisch für *Properties* sind Operationen wie *Set()* oder *Get()*, bei *Methods* Operationen wie *Start()* oder *Abort()*. Als Parameter für die Funktionen steht eine optionale Parameterliste zur Verfügung, deren variable Länge über das Feld *TelLen* angegeben wird. Falls mehr als 12 Byte benötigt werden, werden die Daten einer logischen Steuerbotschaft auf mehrere physikalische Steuerbotschaften aufgeteilt (segmentiert). Das Feld *TelID* dient dann dazu, die Segmentnummer zu übertragen, bei unsegmentierten Botschaften ist es 0. Die *Network Services* übernehmen sowohl die Segmentierung auf der Senderseite als auch das Zusammensetzen auf der Empfängerseite. Auf Applikationsebene können damit Botschaften mit bis zu 64 KB Daten transparent übertragen werden (*Application Message Service AMS*).

Gewöhnungsbedürftig im Umgang mit den Funktionsblöcken ist, dass die Funktionen bzw. zugehörigen Steuerbotschaften oft als Bündel der verschiedenen Kennziffern, allgemein *FBlockID*.*InstID*.*FktID*.*OpType*(*Parameter*) dokumentiert werden. Beispielsweise steht 22.00.400.0.(20) für *AudioAmplifier.00.Volume.Set(20)*, d. h. der Audioverstärker (*FBlockID*=22h) mit der Instanznummer *InstID*=00h, d. h. der erste vorhandene Verstärker, erhält die Anweisung, die Lautstärke (*FktID*=400h) neu einzustellen (*OpType*= 0), wobei der gewünschte Wert 20 (auf einer Skala von 0 ... 255) ist.

Die hier aufgezeigte Struktur für Steuerbotschaften (*FBlockID*, *FktID* usw.) wird prinzipiell auch für den asynchronen Datenkanal verwendet. Einziger Unterschied ist die Größe der *TellLen*, welche dort auf 12 bit erweitert wurde, um damit eine maximal zulässige Nutzdatenmenge von bis zu 1008 Byte zu spezifizieren.

3.4.5 Netzmanagement

MOST kennt verschiedene Rollen zur Verwaltung des Netzwerkes, die theoretisch auf verschiedene Geräte verteilt werden könnten, praktisch aber meist innerhalb der zentralen Bedieneinheit für das Infotainmentsystem im Armaturenbrett, der *Head Unit*, zusammengefasst werden:

- *Timing Master*: Erzeugung des Bit- und Frame-Taktes.
- *Network Master*: Führt Buch über die Systemkonfiguration, d. h. die im Ring vorhandenen Geräte und deren Eigenschaften (*Central Registry*).
- *Power Master*: Koordiniert das Herunterfahren (*Shutdown*) des Bussystems.
- *Connection Master*: Verwaltet die synchronen Übertragungskanäle.

3.4.5.1 Start und Stopp des Bussystems

Die Verwaltung des Netzes erfolgt durch eine Mischung von Hardware innerhalb des Kommunikationscontrollers und Software im steuernden Mikrocontroller. Der *Network Master* durchsucht beim Start das Netzwerk nach angeschlossenen Steuergeräten (*System Scan*), fragt deren Eigenschaften ab und legt eine Tabelle (*Central Registry*) mit diesen Informationen an. Bei dieser Abfrage wird u. a. ermittelt, welche Geräte welche Funktionsblöcke enthalten. Falls die Steuergeräte nicht neu eingeschaltet, sondern lediglich in einem energiesparenden Zustand (*Sleep*) waren, werden sie durch das Lichtsignal bzw. elektrische Empfangssignal aufgeweckt. Das Abschalten des Bussystems wird vom *Power Master* über Statusbotschaften (*NetBlock.InstID.Shutdown.Query*) angekündigt, danach hat jedes Steuergerät Zeit, die notwendigen Vorbereitungen zu treffen, bevor das Master-Steuergerät das Sendesignal abschaltet. Die Slave-Geräte reagieren auf die ausbleibenden Signale und gehen, zumindest an der Busschnittstelle, ebenfalls in den *Sleep* Zustand über. Das Aufwecken des Bussystems aus dem *Sleep* Zustand kann durch jedes Gerät am Bus erfolgen, in dem es sein Sendesignal wieder einschaltet. Dadurch wachen der Reihe nach alle Geräte im Ring auf und der *Timing Master* beginnt mit dem Aussenden von Frames. Sobald sich die Geräte auf den Takt der Empfangssignale synchronisiert haben (*Lock*, siehe unten), beginnt der *Network Master* mit dem beschriebenen Durchsuchen des Netzes.

3.4.5.2 Systemzustände

Aus Sicht der Anwendung befindet sich der MOST Bus in einem der beiden Zustände *OK* oder *Not OK*. Der momentane Systemzustand wird vom *Network Master* bestimmt und bei

Änderungen über die Nachricht *NetworkMaster.InstID.Configuration.Status* an alle Busteilnehmer verteilt. Bei jedem Systemstart wird zunächst vom Zustand *Not OK* ausgegangen. Sobald der *Network Master* die *Central Registry* erfolgreich aufgebaut hat, wird der Zustand *OK* eingenommen und die Steuergeräte fangen an, miteinander zu kommunizieren. Ändert ein Steuergerät seine Eigenschaften oder stellt der *Network Master* eine Änderung bzw. einen Fehler fest, z. B. ein ab- oder neu eingeschaltetes Gerät, so wird dies allen Steuergeräten über obige Statusbotschaft als *Not OK* mitgeteilt. Dadurch werden alle synchronen Verbindungen unterbrochen und durch den Neuaufbau der *Central Registry* das Netz rekonfiguriert. Bei weniger kritischen Änderungen verbleibt das Netz im Zustand *OK* und es wird lediglich eine Nachricht mit den entsprechenden Informationen über die Änderung verschickt.

Neben diesen beiden Zuständen, die das gesamte System betreffen, verwalten die *Network Services* noch weiteren Zustände individuell für jedes Steuergerät. Die wichtigsten sind *Lock* und *Unlock*. Detektiert werden sie vom Kommunikationscontroller, haben aber direkte Auswirkungen auf die Applikation. Der Zustand *Lock* wird vom Steuergerät dann eingenommen, wenn es sich auf die eingehenden Lichtimpulse synchronisieren kann. Kommt es zu einer Störung dieser Synchronisation, meldet der Kommunikationscontroller dem steuernden Mikrocontroller einen *Unlock*, da dann von fehlerhaft empfangenen Daten auszugehen ist. Die Anwendung wird darauf reagieren und im Fall eines Verstärkers etwa das Audiosignal stumm schalten.

3.4.5.3 Verwaltung der synchronen Kanäle

Wenn ein Gerät einen Übertragungskanal im synchronen Datenbereich anfordert oder wieder freigibt, reserviert dieses Gerät den Kanal über eine *Resource Allocation* bzw. *Resource Deallocation* Botschaft und erhält eine zustimmende oder ablehnende Bestätigung. Der *Connection Master* trägt die Reservierung in eine lokale Tabelle ein (*Resource Allocation Table*), die automatisch (alle 1024 Frames) an die Slave-Steuergeräte verteilt wird. Die Anwendungssoftware muss auf der Netzwerkebene lediglich eingreifen, wenn die geforderte Übertragungsbandbreite nicht mehr frei ist.

3.4.5.4 Ringbruch-Diagnose für die optischen Verbindungen

Bei einer ringförmigen Topologie führt eine einzelne Unterbrechung zum Totalausfall des Gesamtsystems. Mögliche Szenarien, die zu einer Unterbrechung des Rings führen, sind ein defektes Steuergerät oder ein echter Bruch eines Lichtwellenleiters. MOST spezifiziert innerhalb der *Network Services* nur wenige Diagnosemöglichkeiten, wie etwa die beschriebenen *Lock* bzw. *Unlock* Situationen. Die Fahrzeug- bzw. Gerätehersteller verwenden daher proprietäre Diagnosestrategien, die aber praktisch alle auf den gleichen Kernprinzipien beruhen. Die MOST Steuergeräte erhalten neben der optischen Schnittstelle zusätzlich eine elektrische Ein-Draht-, LIN- oder CAN-Schnittstelle. Über diese wird das Gerät geweckt und, wenn auch eventuell sehr primitiv, mit dem Gerät bidirektional kommuniziert. Im Fall der Ringbruch-Diagnose werden alle Geräte aufgefordert, Lichtsignale auszugeben. Danach wird von jedem Steuergerät einzeln abgefragt, ob es selbst ein Lichtsignal empfängt.

Damit lässt sich herausfinden, zwischen welchen Geräten im Ring sich die Fehlerquelle befindet.

Bei elektrischen statt optischen Verbindungen sind alternative Arten der Ringbruchdiagnose einsetzbar, falls erforderlich.

3.4.6 Höhere Protokollsichten

Parallel zu den Funktionsblöcken existiert das in Software zu implementierende Übertragungsprotokoll *MOST High Protocol MHP*. Es ist für die verbindungsorientierte Übertragung zwischen zwei MOST-Geräten vorgesehen und erlaubt die segmentierte Übertragung größerer Datenmengen mit Verbindungsaufbau und -abbau sowie Flusssteuerung. MHP setzt auf den *Network Services Level I* auf und verwendet wahlweise den asynchronen Datenbereich oder Datenpakete für Steuerdaten. Alternativ zu *MHP* können mit Hilfe des *MOST Asynchronous Media Access Control MAMAC* Protokolls unmittelbar Ethernet-Botschaften über MOST25 übertragen werden. Mit *MHP* und *MAMAC* können MOST25-Systeme mit TCP/IP basierten Notebooks oder dem Internet gekoppelt bzw. zur Weiterleitung derartiger Botschaften verwendet werden. Bei MOST150 ist *MAMAC* nicht mehr nötig, da die Unterstützung von *MOST Ethernet Packets MEP* Botschaften direkt in den asynchronen Datenkanal integriert ist.

Audio- und Videodaten sind aus Kopierschutzgründen oft verschlüsselt abgelegt. Da die Nutzungsbedingungen die unverschlüsselte Übertragung über ein aus der Sicht der Musik- und Filmindustrie vermeintlich offenes Datennetz wie MOST verbieten, muss neuerdings auch eine verschlüsselte Datenübertragung bereitgestellt werden, bei der Sender und Empfänger sich authentifizieren und Schlüssel austauschen. Dazu wird *Digital Transmission Content Protection DTCP* eingesetzt, ein Verfahren das aus der Unterhaltungselektronik stammt und auch bei IEEE 1394-FireWire verwendet wird.

3.4.7 Beispiel für Systemstart und Audioverbindung

In Abb. 3.39 ist beispielhaft der Aufbau eines MOST25 Rings mit vier Teilnehmern dargestellt. Der *Network*, der *Connection* und der *Timing Master* werden in der Regel im zentralen Infotainment-Steuergerät implementiert, der sogenannten *Head Unit*, die auch als Schnittstelle zum Benutzer des Systems dient. Die anderen drei Teilnehmer (*Slaves*) stellen Dienste bereit, die von der *Head Unit* genutzt werden. Das Beispielnetz besteht aus einem Navigationssystem, einem CD-Spieler und einem Verstärker.

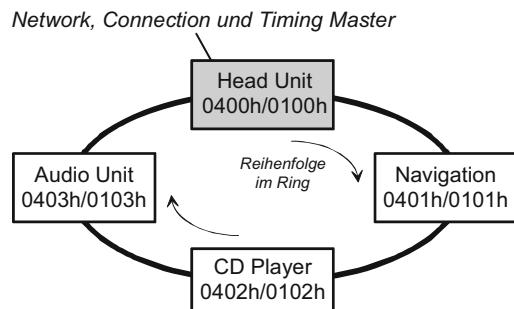
Die in Tab. 3.13 dargestellte Aufzeichnung von Steuerbotschaften zeigt den Startvorgang des Systems sowie den Aufbau einer synchronen Audioverbindung zwischen CD-Player und Audioverstärker in leicht vereinfachter Form.

Tab. 3.13 Steuerbotschaften beim Startvorgang des MOST-Rings nach Abb. 3.39

Von	Nach	Msg.	Steuerbotschaften
			Type
0100	0400	04	ResourceDeAllocate.DeallocateAll Beim Start eines MOST Netzwerkes setzt der <i>Network Master</i> die Belegung der synchronen Kanäle zurück und versendet die Systemnachricht <i>Resource Deallocation</i> . Systemnachrichten werden stets über den Pseudo-Funktionsblock 00h mit Zieladresse 400h verschickt und von den <i>Network Services</i> verarbeitet. Auf Applikationsebene ist diese Nachricht nicht sichtbar.
0100	0401	00	NetBlock.01.FBlockIDs.Get
0101	0100	00	NetBlock.01.FBlockIDs.Status. 520053000600
0100	0402	00	NetBlock.02.FBlockIDs.Get
0102	0100	00	NetBlock.02.FBlockIDs.Status. 31000600
0100	0403	00	NetBlock.03.FBlockIDs.Get
0103	0100	00	NetBlock.03.FBlockIDs.Status. 22000600 Diese Folge von Nachrichten stellt den sogenannten <i>System Scan</i> dar, bei dem der <i>Network Master</i> die <i>Slaves</i> der Reihe nach abfragt, um die unterstützten Funktionsblöcke zu ermitteln. Dabei wird die physikalische Adresse (400h + Position im Ring) verwendet, da die logische Adresse zu diesem Zeitpunkt noch undefiniert ist. Die Navigation meldet <i>FblockID</i> =52h (<i>Navigation System</i>), 53h (<i>TMC Decoder</i>) und 06h (<i>Diagnosis</i>), jeweils mit <i>InstID</i> =00h. Der CD-Spieler unterstützt die Funktionsblöcke 31h (<i>Audio Disk Player</i>) und 06h (<i>Diagnosis</i>). Der Verstärker bietet neben seiner Hauptfunktion <i>Audio Amplifier</i> (<i>FblockID</i> 22h), ebenfalls die Diagnosefunktion.
0100	0101	00	NetBlock.01.FBlockIDs.SetGet.060001
0101	0100	00	NetBlock.01.FBlockIDs.Status. 520053000601
0100	0102	00	NetBlock.02.FBlockIDs.SetGet.060002
0102	0100	00	NetBlock.02.FBlockIDs.Status. 31000602
0100	0103	00	NetBlock.03.FBlockIDs.SetGet.060003
0103	0100	00	NetBlock.03.FBlockIDs.Status. 2200060326002400 Innerhalb eines MOST Rings muss die Kombination aus <i>FBlockID</i> und <i>InstID</i> eindeutig sein. Dies ist im vorliegenden Beispiel für den Diagnosefunktionsblock 06h noch nicht gegeben, da alle <i>Slaves</i> dieselbe <i>InstID</i> =00h für diesen Block gemeldet haben. Der <i>Network Master</i> löst die Mehrdeutigkeit auf und weist den Slaves die neuen <i>InstIDs</i> 01, 02 und 03 für diesen Funktionsblock zu.
0100	03C8	00	NetworkMaster.00.Configuration.Status.OK Nachdem der <i>Network Master</i> die <i>Central Registry</i> mit den gemeldeten Funktionsblöcken erfolgreich aufgebaut hat, wird allen Teilnehmern über eine <i>Broadcast</i> Nachricht der Konfigurationsstatus <i>OK</i> mitgeteilt. Anschließend ist der Ring funktionsbereit.

Tab. 3.13 (Fortsetzung)

Von	Nach	Msg.	Steuerbotschaften
			Type
0100	0102	00	AudioDiskPlayer.01.Allocate.StartResult.01
0102	0400	03	ResourceAllocate.00010203
0102	0100	00	AudioDiskPlayer.01.Allocate.Result.010200010203 Die <i>Head Unit</i> fordert den CD-Spieler auf, Daten seiner logischen Datenquelle 01h zu übertragen. Quelle 01h entspricht in diesem Beispiel dem Stereo-Audiosignal der CD mit Abtastwerten von je 16 bit, d. h. vier physikalischen Kanälen zu je 8 bit. Die Applikation des CD-Spielers fordert diese Kanäle über die <i>Network Services</i> an. Die Systemnachricht <i>Resource Allocate</i> informiert alle Busteilnehmer, dass diese Kanäle belegt sind. Der CD-Spieler verbindet den Datenstrom der Quelle mit den reservierten MOST Kanälen und bestätigt die erfolgreiche Ausführung des <i>Allocate</i> , wobei als Parameter die Quelle (01), die Verzögerung der synchronen Daten relativ zum <i>Master</i> (02) und die Liste der belegten Kanäle (00 01 02 03) in der Bestätigung gesendet werden.
0100	0103	00	AudioAmplifier.00.Connect.StartResult.010000010203
0103	0100	00	AudioAmplifier.00.Connect.Result.01 Damit die übertragenen Audiodaten vom Audioverstärker verarbeitet werden, wird er vom <i>Network Master</i> über die Funktion <i>Connect</i> aufgefordert, seine logische Datensenke (<i>Sink</i>) 01, d. h. seine digitalen Audioeingänge, mit den synchronen Daten auf den Kanälen 0...3 zu verbinden.
0100	0102	00	AudioDiskPlayer.01.DeckStatus.Set.00 Bisher hat der CD-Spieler nur Leerdaten übertragen. Über die Funktion <i>DeckStatus</i> erhält er jetzt den Befehl, die CD tatsächlich abzuspielen (Parameter 00= <i>Play</i> , 01= <i>Stop</i> usw.).
0100	0103	00	AudioAmplifier.00.Volume.Set.1 F
0100	0103	00	AudioAmplifier.00.Mute.SetGet.0100
0103	0100	00	AudioAmplifier.00.Mute.Status. 0100 Hier wird die gewünschte Lautstärke (<i>Volume</i>) im Verstärker eingestellt und die Stummschaltung (<i>Mute</i>) aufgehoben, damit das Audiosignal tatsächlich über die Lautsprecher zu hören ist.

Abb. 3.39 Beispiel eines MOST25-Rings (mit physikalischen und logischen Adressen)

3.4.8 Zusammenfassung MOST

- Bitstrom-orientiertes Übertragungsprotokoll mit Kunststoff-Lichtwellenleiter für Infotainment-Anwendungen, seltener mit elektrischen Zwei-Draht- bzw. Koaxialkabel-Verbindungen, überwiegend in Ringstruktur.
- Bitrate ca. 25 Mbit/s (MOST25), zweite Generation mit 50 Mbit/s (MOST50), Weiterentwicklung mit 150 Mbit/s (MOST150).
- Kommunikation mit logischen Kanälen und reservierter Übertragungsbandbreite (bis zu 60 Byte je MOST25-Botschaft, 117 Byte je MOST50-Botschaft bzw. 372 Byte je MOST150-Botschaft bei einer Botschaftsrate von 48 kHz) mit TDMA-Buszugriff. Nutzdatenrate bei MOST25 bis 2,6 MB/s (MOST25), 5,6 MB/s (MOST50) und 17,8 MB/s (MOST150), wenn keine asynchronen Daten übertragen werden.
- Falls nicht die volle Bandbreite für synchrone Daten benötigt wird, steht der Rest der Bandbreite für eine paketorientierte, asynchrone Übertragung zur Verfügung. Für die paketorientierte Übertragung keine garantierte Botschaftslatenz. Absicherung durch CRC-Prüfsumme. Bei MOST150 können auch Ethernet-Botschaften ohne Umformierung über MOST übertragen werden.
- Zusätzlich Steuerbotschaften für die Netzwerkverwaltung und Anwendungen mit automatischer Fehlerbehandlung, aber auch ohne garantierte Botschaftslatenz. Datenraten um 50 KB/s [17].

3.5 Automotive Ethernet

In den letzten Jahren musste die Automobilindustrie lernen, dass eine Vielzahl inkompatibler und nahezu ausschließlich in der Automobilindustrie verwendeter Lösungen zu hohen Kosten und einem ständigen Weiterentwicklungsaufwand führt. Daher überlegt man im Hinblick auf den weiter wachsenden Bandbreitenbedarf, das im Bürobereich seit vielen Jahren etablierte Kommunikationskonzept Ethernet/IP einzuführen. Ethernet als *Bussystem* (ISO Layer 1 und 2) erscheint aufgrund seiner extrem hohen Stückzahlen kostengünstig und wird wegen seiner hohen Verbreitung auch ständig weiterentwickelt. Die Internetprotokolle IP, TCP, UDP usw. (ISO Layer 3 und 4) erlauben eine transparente Kommunikation, da sie heute von praktisch jedem computerähnlichen Gerät unterstützt werden. Damit vereinfacht sich die Integration von Consumergeräten wie Smartphones erheblich.

3.5.1 Ethernet nach IEEE 802.3

Ursprünglich war Ethernet ein Linienbussystem mit CSMA/CD Zugriffsverfahren (Kap. 2). In solchen Systemen steigt die Übertragungslatenz bei höheren Busbelastungen sehr stark an. Moderne Ethernet Netze (Abb. 3.40) werden deshalb in Stern-Topologie ausgelegt, wobei am Kopplungspunkt ein sogenannter *Switch* sitzt, der ankommende Botschaften an

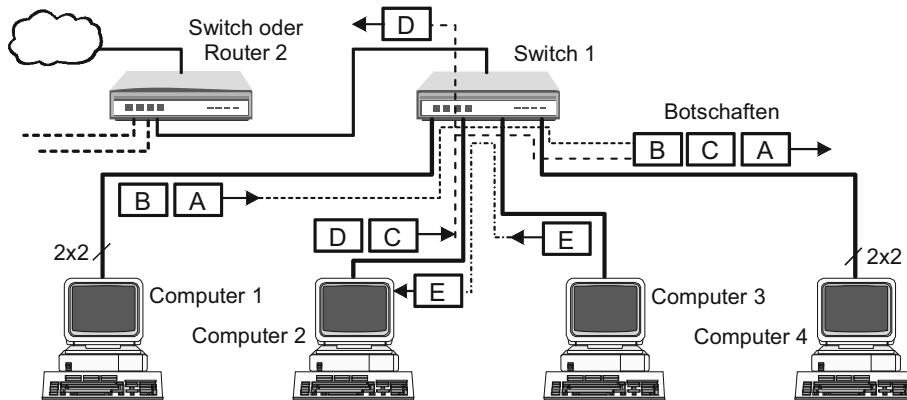


Abb. 3.40 Topologie eines Switched Ethernet Netzes

genau eines der angeschlossenen Geräte weiterversendet (*Switched Network*). Der Empfänger wird aus der in den Botschaften enthaltenen Ethernet-Adressen ermittelt. Lediglich bei unbekanntem Empfänger leitet der *Switch* eine Botschaft an alle angeschlossenen Geräte weiter. Welche Geräte angeschlossen sind, ermittelt der *Switch* im laufenden Betrieb aus den Ethernet-Adressen der ankommenden Botschaften selbstständig.

Jeder Computer ist an den *Switch* über ein Paar von Zwei-Draht-Leitungen als Punkt-zu-Punkt Verbindung angeschlossen. Damit ist eine Voll-Duplex-Kommunikation möglich. Die Kopplung zwischen den Leitungen im *Switch* erfolgt nach dem Prinzip des Kreuzschienenverteilers, so dass der *Switch* gleichzeitig Botschaften von verschiedenen Sendern zu mehreren Empfängern übertragen kann (in Abb. 3.40 z. B. Botschaft A und E). Das System arbeitet kollisionsfrei. Zu größeren Übertragungsverzögerungen kommt es nur, wenn am *Switch* gleichzeitig mehrere Botschaften für denselben Empfänger ankommen. In diesem Fall leitet der *Switch* die erste Botschaft direkt weiter, die restlichen Botschaften werden zwischengespeichert (in Abb. 3.40 z. B. Botschaft C, die gleichzeitig mit Botschaft A ankommt) und erst mit Verzögerung weitergesendet (*Store and Forward*). Botschaften gehen nur dann verloren, wenn der Pufferspeicher im *Switch* unterdimensioniert ist. Ein Netz kann mehrere *Switch* enthalten. Nach Möglichkeit sollte dabei eine hierarchische Baumstruktur verwendet werden, bei der es zwischen zwei angeschlossenen Geräten jeweils einen eindeutigen Verbindungsweg gibt. Ist dies nicht der Fall, können sich moderne *Switch* mit Hilfe des sogenannten *Spanning Tree Protocols* automatisch so konfigurieren, dass zumindest die logische Verbindungsstruktur eindeutig ist.

Zu Beginn einer Ethernet-Botschaft (Abb. 3.41) wird eine feste Bitfolge, die *Präambel* und der *Start Frame Delimiter SFD*, gesendet, die bei manchen *Physical Layer*n zur Taktsynchronisation notwendig sind. Ethernet arbeitet mit Geräteadressierung. Die jeweils 6 Byte langen Ziel- und Quelladressen (*Medium Access Control MAC* Adressen) sind den Ethernet-Kommunikationscontrollern fest zugeordnet und werden von den Chiphersteller weltweit eindeutig festgelegt. Neben den eindeutigen Adressen existieren *Multicast* Adres-

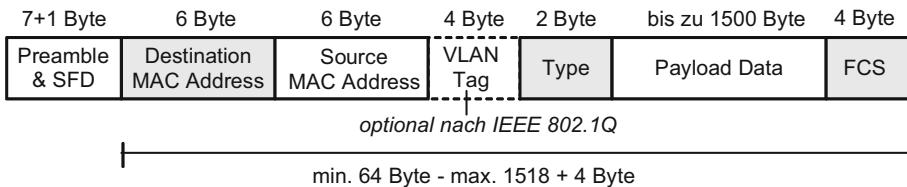


Abb. 3.41 Format einer Ethernet-Botschaft nach IEEE 802.3

sen für Gerätegruppen sowie eine *Broadcast* Adresse, um eine Botschaft an alle angeschlossenen Geräte gleichzeitig zu senden. Das optionale VLAN Tag erlaubt die Bildung von Unternetzen, den virtuellen LANs. Die Teilnehmer eines VLAN können miteinander kommunizieren, sind aber für andere Teilnehmer außerhalb des Unternetzes praktisch unsichtbar, weil ihre Botschaften nur innerhalb des eigenen Unternetzes weitergeleitet werden. Zusätzlich zur VLAN-Kennung kann das Tag-Feld auch eine Prioritätsforderung enthalten, die von einem *Switch* berücksichtigt wird, wenn mehrere ankommende Botschaften zur Weiterleitung anstehen.

Durch ein Typfeld wird der Inhalt des folgenden Nutzdatenfeldes gekennzeichnet. Enthält dieses eine IPv4-Botschaft (Internet Protokoll Version 4), so wird z. B. 800_H gesendet, bei IPv6 $86DD_H$. Das Datenfeld kann maximal 1500 Byte enthalten. Sollen nur wenige Daten versendet werden, muss das Nutzdatenfeld gegebenenfalls mit Leerdaten aufgefüllt werden, weil die Ethernet-Botschaft (ohne Präambel und SFD) mindestens 64 Byte lang sein muss. Am Ende der Botschaft folgt eine Prüfsumme, die *Frame Check Sequence*. Bei Übertragungsfehlern verwirft der Empfänger die Botschaft automatisch, eine Fehlermeldung oder Empfangsbestätigung erfolgt nicht. Zwischen zwei Ethernet-Botschaften muss ein Mindestabstand (*Inter Packet Gap* oder *Inter Frame Spacing*) von 96 Bittakten eingehalten werden.

Zum Erfolg von Ethernet hat die ausgeprägte Trennung zwischen *Data Link Layer* (im Ethernet-Jargon *Medium Access Control MAC*) und der Bitübertragungsschicht (*Physical Layer PHY*) beigetragen (Abb. 3.42). Beide sind in IEEE 802.3 standardisiert, wobei die MAC Schicht seit vielen Jahren praktisch unverändert ist, während die PHY Ebene laufend weiterentwickelt wird, um höhere Übertragungsraten abzudecken. Der Durchbruch gelang Ethernet mit einer Bitrate von 10 Mbit/s, die zunächst über Koaxialkabel und später über ungeschirmte, verdrillte Leitungspaare übertragen wurde (Ethernet-Bezeichnung 10-Base-T). Die nächste Ausbaustufe verwendet 100 Mbit/s (*Fast Ethernet* 100-Base-TX) mit zwei verdrillten Leitungspaaren, während heute im Bürobereich oft schon 1 Gbit/s (*Gigabit Ethernet* 1000-Base-T) mit vier verdrillten Leitungspaaren eingesetzt wird. Für noch höhere Bitraten (10 bzw. 100 Gbit/s und mehr) gibt es ebenfalls schon erste Komponenten. Daneben existieren verschiedene Untervarianten, aber auch Versionen für Glasfaserkabel, Varianten für Kleingeräte mit integrierter Spannungsversorgung über die Ethernet-Anbindung (*Power over Ethernet*) oder umgekehrt Ethernet-Verbindungen über das gewöhnliche 230 V Hausnetz (*Powerline Ethernet*).

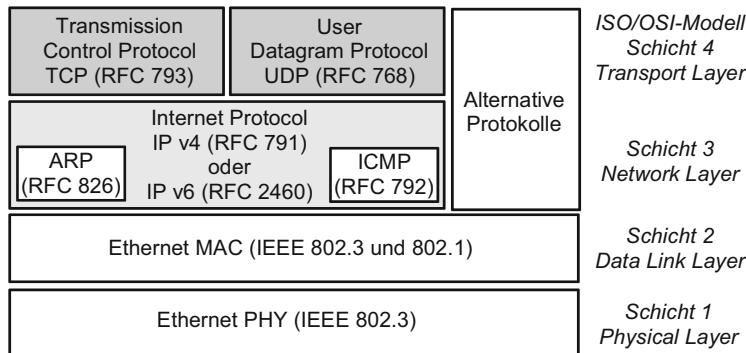


Abb. 3.42 Protokollstapel Ethernet, IP, TCP und UDP

Die Trennung zwischen MAC und PHY Schicht ist auch in der Hardwareimplementierung üblich (Abb. 3.43). Die PHY-Komponente besteht bei Ethernet nicht nur aus einem einfachen *Transceiver*, sondern enthält die komplette Logik zur Bitübertragung. Da die Leitungsschnittstelle galvanisch entkoppelt wird, sind zusätzlich zwei kleine Übertrager und einige passive Bauteile notwendig. Die meisten PHY-Komponenten können selbstständig erkennen, ob die Kommunikationspartner mit 10 Mbit/s oder mit 100 Mbit/s arbeiten (*Autonegotiation*). Die MAC Schicht ist oft bereits in den Mikrocontroller integriert. Die Signale zwischen MAC und PHY-Schicht sind als *Media Independent Interface MII* weitgehend standardisiert, so dass Komponenten von unterschiedlichen Herstellern kombiniert werden können.

3.5.2 Autotauglicher Physical Layer BroadR-Reach

Wegen des Kabels mit zwei Leitungspaaren und der Steckverbindertechnik gelten Standard-Ethernet-Komponenten weder bezüglich der Kosten noch im Hinblick auf die EMV-, Temperatur-, Schüttel- und Feuchtebelastung als direkt autotauglich. Bei der neuen Diagnoseschnittstelle *Diagnostic over Ethernet DoIP* (siehe Abschn. 4.6) wird die konventionelle Technik trotzdem eingesetzt, weil diese Anforderungen im Werkstatt- und Fertigungsumfeld unproblematisch sind. Für den Einsatz in der On-Board-Kommunikation

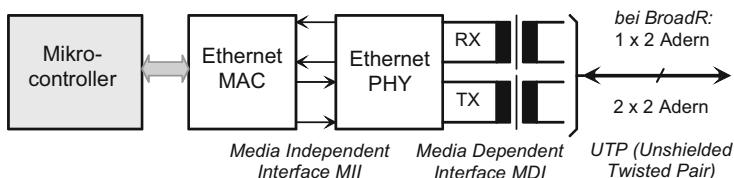


Abb. 3.43 Typischer Aufbau einer 10/100-BASE-T Ethernet-Schnittstelle (vereinfacht)

dagegen soll ein spezieller *Physical Layer* der Firma Broadcom verwendet werden. Dessen unter der Bezeichnung *BroadR-Reach* vermarktete PHY-Bausteine erlauben derzeit eine Bitrate von 100 Mbit/s im Voll-Duplex-Betrieb über eine einfache, verdrillte Zweidraht-Leitung. Die Anforderungen an das Kabel und die Steckverbinder sollen dabei nicht höher sein als bei einem FlexRay-System. Unter dem Dach der *One-Pair Ethernet (OPEN) Alliance* haben sich Broadcom und verschiedene Automobilhersteller und Zulieferer zu einem Konsortium zusammengeschlossen, das das Konzept vollständig standardisieren und in der Praxis erproben will. Mindestens einer der beteiligten Automobilhersteller hat bereits Serienprojekte auf dieser Basis angekündigt, wobei zunächst MOST und später auch FlexRay durch Ethernet ergänzt bzw. ersetzt werden soll.

Wie erfolgreich *BroadR-Reach* sein wird, hängt sicher nicht nur davon ab, ob die bisher proprietäre Technologie auch von anderen Halbleiterherstellern nachgebaut werden kann und darf. Bei MOST war ja genau dies ein erhebliches Hindernis. Entscheidend für Ethernet aber wird sein, ob es gelingt, ein kostengünstiges Konzept für die Echtzeitfähigkeit zu erarbeiten.

3.5.3 Echtzeitfähigkeit mit IEEE 802.1 Audio-Video-Bridging AVB

Die Dauer einer Ethernet-Botschaft lässt sich abschätzen zu

$$T_{\text{Frame}} \approx (n_{\text{Header+Trailer}} + n_{\text{Data}}) \cdot T_{\text{bit}}. \quad (3.25)$$

Der *Overhead* der Botschaft ist mit $n_{\text{Header+Trailer}} = 30$ Byte = 240 bit sehr groß, wenn man die Präambel und das optionale VLAN-Tag mit berücksichtigt. Die kürzestmögliche Ethernet-Botschaft (Präambel plus 64 Byte, davon 42 Byte Daten) wird bei einer Bitrate von $f_{\text{bit}} = 1 / T_{\text{bit}} = 100$ Mbit/s in knapp 6 μs übertragen. Bei $n_{\text{Data}} = 254$ Byte Daten, der Maximallänge von FlexRay, benötigt die Botschaft 23 μs und selbst die längste Ethernet-Botschaft mit $n_{\text{Data}} = 1500$ Byte lässt sich in nur 123 μs versenden.

Die maximale Bandbreite einer Ethernet-Verbindung ergibt sich zu

$$f_{\text{Data}} = \frac{n_{\text{Data}}}{T_{\text{Frame}} + 96 \cdot T_{\text{bit}}}, \quad (3.26)$$

wobei berücksichtigt wurde, dass zwischen den Botschaften eine Pause von mindestens 96 bit eingehalten werden muss. Ein 100 Mbit/s Netz hat damit bei einer Datenlänge zwischen 42 und 1500 Byte eine Bandbreite von 6 bis 12 MB/s.

Im Gegensatz zur Dauer einer Botschaft lässt sich die gesamte Übertragungsverzögerung T_{Latenz} zwischen Sender und Empfänger nicht so einfach ermitteln. Zur Dauer der Ethernet-Botschaft selbst addiert sich die Durchlaufverzögerung aller *Switches* auf dem Weg zwischen Sende- und Empfangssteuergerät.

$$T_{\text{Latenz}} = T_{\text{Frame}} + \sum T_{\text{Switch}} \quad (3.27)$$

Die Durchlaufverzögerung T_{Switch} hängt zunächst von deren interner Arbeitsweise ab. Jeder *Switch* muss die Botschaft mindestens solange verzögern, bis er die Zieladresse empfangen hat und erkennen kann, an welches Steuergerät er die Botschaft weiterleiten muss (*Cut Through*). Die theoretisch minimale Durchlaufzeit lässt sich daher zu $T_{\text{Switch,min}} = 112 \cdot T_{\text{bit}} = 1,2 \mu\text{s}$ abschätzen. Dieser Wert wird in der Praxis aber nicht erreichbar sein, da die Überprüfung der Adresse und das möglich Umkopieren der Botschaft zwischen verschiedenen Pufferspeichern zusätzliche Zeit kostet. Hersteller industrieller *Cut Through Switches* geben Werte im Bereich von $5 \mu\text{s}$ an. Manche *Switches* warten aber sogar mit der Weiterleitung, bis sie eine Botschaft vollständig empfangen haben (*Store and Forward*). In diesem Fall kann nämlich die Prüfsumme der empfangenen Botschaft überwacht und fehlerhafte Botschaften blockiert werden. Solche *Switches* haben eine Verzögerung von mindestens $T_{\text{Switch}} = T_{\text{Frame}}$.

Kommen an einem *Switch* gleichzeitig auch noch mehrere Botschaften für denselben Empfänger an, so werden diese Botschaften in jedem Fall vollständig zwischengespeichert und erst nacheinander weitergeleitet. Sowohl die zulässige Durchlaufverzögerung als auch die Reihenfolge bei der Weiterleitung sind im Ethernet-Standard IEEE 802.3 nicht geregelt, so dass Ethernet in seiner Grundform nicht echtzeitfähig ist.

Um trotzdem Anwendungen mit Echtzeit-ähnlichen Anforderungen wie beispielsweise *Voice-over-IP-Telefonie* oder *Live-Video-Streams* im Internet zu realisieren, behilft man sich mit speziellen Protokollen in den höheren Ebenen des Protokollstapels, z. B. dem *Real Time Transport Protocol RTP* nach RFC 3550. Diese Lösungen können aber nur eine ausreichende mittlere Bandbreite für die Datenübertragung sicherstellen. Das Problem der variablen Übertragungsverzögerung (*Jitter*) selbst können sie nicht beseitigen, sondern nur kaschieren. Der entscheidende *Trick* besteht darin, im Empfänger einen ausreichend großen Puffer einzurichten, in dem die mit variabler Verzögerung ankommenden Botschaften zwischengespeichert und mit konstanter Rate ausgelesen und weiterverarbeitet werden. Dadurch wird die Gesamtverzögerung zwischen Sender und Empfänger zwar größer, erscheint aber als konstante Latenz. Für Sprach- und Bildübertragungen ist dies unkritisch, zumal dort sogar sporadisch ausfallende Botschaften vom Menschen nicht wahrgenommen werden.

Bei zeitkritischen Steuerungsdaten oder gar in geschlossenen Regelkreisen muss die Übertragung aller Botschaften mit bekannter Maximalverzögerung gewährleistet werden. Nicht übertragene Botschaften können dort sogar sicherheitskritisch werden. Für Industriesteuerungen, bei denen Ethernet schon seit längerem eingesetzt wird, wurden daher eine Reihe von Echtzeitlösungen wie *ProfiNet*, *EtherCAT*, *EtherNet/IP*, *Modbus/TCP*, *Sercos III* oder *TTEthernet* entwickelt [19]. Hinter vielen dieser Lösungen steht allerdings nur ein einzelner großer Automatisierungshersteller oder ein Konsortium von kleineren Firmen. Ein einheitlicher Industriestandard fehlt. Gelegentlich werden sogar modifizierte *Data Link Layer* und damit herstellerspezifische Kommunikationscontroller verwendet. Um eine kostengünstige, für den Massenmarkt taugliche Lösung zu finden, haben sich *Consumer*-Geräte- und Computerhersteller zusammengeschlossen und ein Konzept für das Audio-Video-Bridging AVB erarbeitet, die Echtzeit-Übertragung von Audio- und Videodaten direkt über den Ethernet *Data Link Layer*. Zur Unterstützung des Konzepts, das

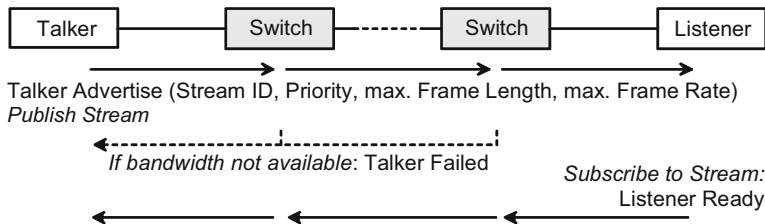


Abb. 3.44 Einrichten einer Echtzeitverbindung mit dem *Stream Reservation Protocol*

inzwischen unter IEEE 802.1 standardisiert ist, haben sich diese Hersteller in der *AVnu Alliance* organisiert. Diesem Konsortium haben sich mittlerweile auch die Automobilhersteller und Zulieferer der *OPEN Alliance* Initiative sowie einige Automatisierungshersteller angeschlossen, um Ethernet vollständig echtzeit- und automobiltauglich zu machen [20].

AVB ist als IEEE 802.1Q standardisiert, die Basisnorm enthält seit 2012 auch die früheren Teilenormen 802.1Qat und 802.Qav. IEEE 802.1Q spezifiziert u. a. das *Multiple Stream Reservation Protocol MSRP*, mit dem Übertragungskanäle (*Streams*) mit garantierter Bandbreite und maximaler Übertragungsverzögerung eingerichtet werden können. Die Norm nennt bis zu 75 % der Bandbreite, die für AVB-Botschaften reserviert werden sollen, und strebt eine Gesamtlatenz von unter 2 ms bei bis zu sieben Teilstrecken (*Hops*), also sechs *Switchen* zwischen Sender und Empfänger an.

Ein Gerät, das Echtzeitdaten senden möchte, in der Norm als *Talker* bezeichnet, kündigt seinen Bandbreitenbedarf mit Hilfe einer *Talker Advertise* Botschaft an (Abb. 3.44). Der angeschlossene *Switch* prüft, ob der erforderliche Pufferspeicher (*Queue*) verfügbar ist und leitet die Botschaft an alle anderen Geräte weiter, sofern er die Bandbreitenforderung erfüllen kann. Dabei ergänzt er die Botschaft um einen Schätzwert für seine eigene Durchlaufverzögerung, so dass der Empfänger eine Information über die gesamte Übertragungslatenz der späteren Echtzeitdaten bekommt. Falls die Bandbreitenforderung nicht erfüllbar ist, teilt der *Switch* dies durch eine *Talker Failed* Botschaft mit. Ein Steuergerät, das Echtzeitdaten empfangen möchte, in der Norm als *Listener* bezeichnet, meldet sich durch eine *Listener Ready* Botschaft beim *Talker* an. Im Fehlerfall wird eine *Listener Failed* Botschaft versendet. Sobald eine *Talker Advertise* Ankündigung durch mindestens eine *Listener Ready* Anmeldung bestätigt wurde, reservieren alle beteiligten Geräte die Bandbreite verbindlich und die Übertragung kann beginnen. Ein *Stream* bleibt aktiv, bis er wieder abgekündigt wird (*Publish-Subscriber* Konzept nach Kap. 2). Die Echtzeitbotschaften ebenso wie die Botschaften des MSRP-Protokolls sowie einiger weiterer Hilfsprotokolle werden als gewöhnliche Ethernet-Botschaften versendet (Abb. 3.41), wobei das *Type* und das *VLAN Tag* Feld verwendet werden, um zwischen den verschiedenen Protokollen, *Streams* und deren Prioritäten zu unterscheiden. Da die Echtzeitbotschaften in einem *Switch* priorisiert werden, können in einem IEEE 802.1Q kompatiblen Netz auch gewöhnliche Botschaften transportiert werden, ohne die Echtzeitkommunikation zu stören.

Das Problem, dass auch eine höherpriore Botschaft warten muss, wenn ein *Switch* gerade auf derselben ausgehenden Verbindung eine andere Botschaft überträgt, kann aber natürlich grundsätzlich nicht gelöst werden. Die einzige mögliche Abhilfe dafür ist, im gesamten Netz ein synchrones Übertragungsschema einzuführen, das solche Wartesituationen durch zeitversetztes Senden bereits an der Quelle vermeidet. FlexRay hat aus diesem Grund den beschriebenen zeitgesteuerten Kommunikationsplan (*Schedule*) mit fest zugewiesenen Zeitschlitten (*Slots*) und einer netzweiten Zeitbasis (*Cycles, Macro Ticks*). Ethernet dagegen verwendet wie CAN ein ereignisgesteuertes Sendekonzept und besitzt keine eigene netzweite Zeitbasis. Mit Hilfe der in IEEE 802.1AS definierten Untermenge des aus IEEE 1588 schon länger bekannten *Precision Time Protocol PTP* kann diese netzweite Zeitbasis nachgerüstet werden.

Für die Zeitsynchronisation wird das Netz mit Hilfe des *Best Master Clock Algorithmus* dynamisch in einer Baumstruktur organisiert, wobei als Wurzel der Teilnehmer mit der genauesten Zeitbasis festgelegt wird. Dieser Teilnehmer wird als *Grandmaster* bezeichnet. Auf dessen Zeitbasis synchronisieren sich nun die Mitglieder der nächsten Hierarchiestufe, die dabei als *Slaves* agieren. Gegenüber der nächsttieferen Hierarchiestufe dreht sich ihre Rolle um und sie werden zum *Master*. Auf diese Weise wird die Zeitbasis stufenweise von der Wurzel bis zu den Spitzen des Baums synchronisiert, wobei in jedem Teilschritt nur jeweils zwei Teilnehmer beteiligt sind, einer in der *Master*-, der andere in der *Slave*-Rolle. Kern des Verfahrens sind die Botschaften *Sync* und *DelayRequest* (Abb. 3.45), deren Sende- bzw. Empfangszeitpunkte jeweils mit Hilfe der lokalen Zeitbasen gemessen werden. Die vom Master ermittelten Werte werden dem *Slave* in zwei zusätzlichen Botschaften *FollowUp* und *DelayResponse* übermittelt. Aus den vier Messwerten kann der Slave nun sowohl den Zeitversatz seiner eigenen Uhr gegenüber der des Masters als auch die Übertragungszeit der Botschaften berechnen. Dabei wird angenommen, dass die Übertragungsdauer konstant und in beiden Richtungen gleich ist. Da die lokalen Uhren driften können, wird der Vorgang periodisch wiederholt. Die Genauigkeit des Verfahrens hängt entscheidend davon ab, wie genau der Sende- bzw. Empfangszeitpunkt der Ethernet-Botschaften tatsächlich gemessen werden kann. Erfolgt die Messung mit Hilfe von Software-Interrupts, so sind bei den meisten Mikrocontrollern Fehler von deutlich über 10 µs zu erwarten. Für geringfügig modifizierte Ethernet-Controller, die automatisch interne Hardware-Zeitstempel (*Timestamp*) erzeugen, geben Hersteller dagegen erreichbare Fehler bei der Zeitsynchronisation im unteren Mikrosekundenbereich an.

3.5.4 Höhere Protokollsichten IP, TCP und UDP

Theoretisch könnten Anwendungen ihre Daten direkt im Datenfeld der Ethernet-Botschaften verpacken. Da Ethernet aber keine Segmentierung und keine Fehlerbehandlung unterstützt und für Weitverkehrsnetze und Funkübertragungen nicht direkt einsetzbar ist, verwenden Anwendungen heute einige höhere Protokollesbenen.

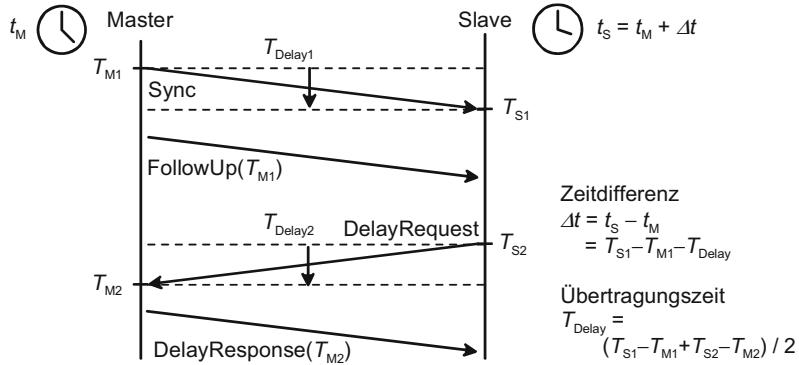


Abb. 3.45 Prinzip der Zeitsynchronisation nach IEEE 802.1AS

Auf Schicht 3 des ISO/OSI-Modells (Abb. 3.42) verwendet man das *Internet Protocol IP* nach RFC 791/2460. Dieses benutzt gerätebezogene Adressen, die fest oder mittels des Hilfsprotokolls *Dynamic Host Configuration Protocol DHCP* nach RFC 2131 auch dynamisch vergeben werden können. Damit Rechner weltweit miteinander kommunizieren können, sollten die Adressen eindeutig sein und werden daher von der Nutzerorganisation ICANN zentral an *Internet Service Provider ISP* und von diesen an die Endteilnehmer vergeben. Leider waren für diese Adressen ursprünglich nur 32 bit Werte vorgesehen, die anfangs zudem sehr großzügig verteilt wurden, so dass der Adressbereich mittlerweile ausgeschöpft ist. Aus diesem Grund wird die lange Zeit stabile Protokollversion IPv4 allmählich durch die neuere Version IPv6 abgelöst, die 128 bit IP Adressen vorsieht. Leider vergrößert sich die Länge des Botschaftsheaders dabei von 20 Byte auf 40 Byte. Die Zuordnung der IP Adressen zu den Ethernet MAC Adressen kann innerhalb eines Ethernet-Netzes mit den Hilfsprotokollen *Address Resolution ARP* bzw. *Neighbor Discovery Protocols NDP* (RFC 826/4861 NDP) ermittelt werden. Die numerischen IP-Adressen, z. B. 134.108.34.3, sind für Menschen recht unhandlich. Daher hat man Klarnamen wie www.hs-esslingen.de eingeführt, deren Zuordnung zu den IP-Adressen von Web-Browsern und ähnlichen Anwendungen im *Domain Name System DNS*, einer Art Telefonbuch für Internet-Adressen, abgefragt werden kann. Der Datenbereich einer IP-Botschaft kann bis zu 64 KB lang sein. Theoretisch können IP-Botschaften zwar bei der Weiterleitung segmentiert werden, in der

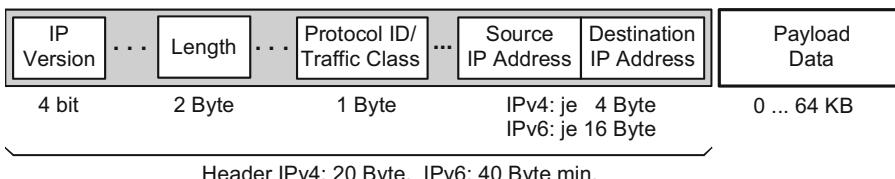


Abb. 3.46 Aufbau einer IP Botschaft (vereinfacht)

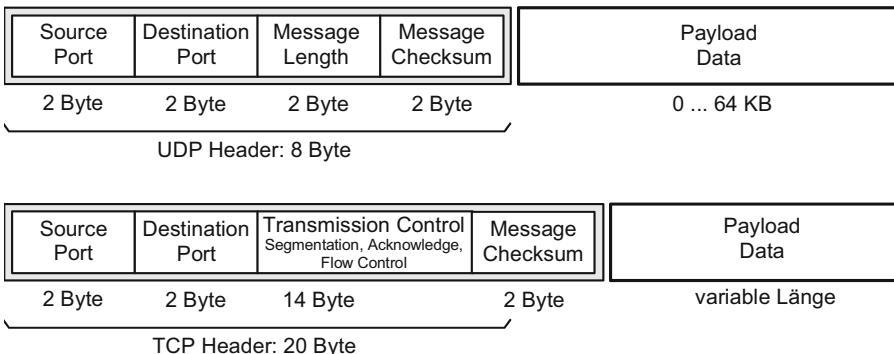


Abb. 3.47 UDP (oben) und TCP Botschaften (vereinfacht)

Praxis wird man die Länge aber meist so beschränken, dass die IP-Botschaft vollständig in das Datenfeld einer Ethernet-Botschaft passt. Falls eine IP-Botschaft nicht weitergeleitet werden kann, z. B. weil eine falsche IP-Adresse verwendet wurde, erhält der Sender von der Station, die den Fehler feststellt, eine Fehlermeldung mittels des Hilfsprotokolls *Internet Control Message Protocol*. Die weitere Fehlerbehandlung ist jedoch auf der IP Ebene genauso wenig definiert wie die Überwachung des Datenfelds durch eine Prüfsumme. IP erlaubt also lediglich eine verbindungslose ungesicherte Übertragung. Das *Protocol ID* Feld zeigt an, welches Protokoll die IP-Botschaft transportiert, z. B. 1 für ICMP, 6 für TCP oder 17 für UDP.

In der nächsthöheren Schicht 4 sind mit UDP und TCP zwei alternative Protokolle im Einsatz (Abb. 3.42). Zusätzlich zu den gerätebezogenen IP-Adressen der Schicht 3 werden in der Schicht 4 sogenannte *Ports* verwendet, die eine anwendungsbezogene Adressierung erlauben. Die Portnummern sind auf der Serverseite meist fest vergeben, z. B. 80_H für Web-Server (*Well-Known Port*), auf der Clientseite werden sie in der Regel dynamisch festgelegt. Durch die sogenannten *Sockets*, die Kombination der IP-Adresse und der Portnummer, kann eine Anwendung auf einem Rechner gezielt Verbindung zu einer Anwendung auf einem anderen Rechner aufnehmen.

Das *User Datagram Protocol UDP* nach RFC 768 erlaubt eine verbindungslose, unbestätigte Datenübertragung von Datenblöcken bis zu 64 KB, die über eine Prüfsumme gesichert werden (Abb. 3.47). Damit ist es möglich, fehlerhafte Botschaften auf der Empfangsseite zu erkennen und zu ignorieren, eine Rückmeldung an den Sender und eine automatische Wiederholung gibt es aber nicht.

Das *Transmission Control Protocol TCP* nach RFC 793 (Abb. 3.47) dagegen kann beliebig große Datenblöcke segmentiert übertragen, deren Empfang bestätigt und im Fehlerfall wiederholt wird [21]. Die Verbindung muss vor Beginn der Übertragung geöffnet und am Ende wieder abgebaut werden. Die Segmentgröße wird eventuell dynamisch angepasst, gegebenenfalls werden sogar mehrere Datenblöcke zwischengespeichert, bevor eine Botschaft versendet wird, um den relativen Overhead zu verringern. Alle diese Vorgänge erfolgen

Tab. 3.14 Bandbreitenschätzung für Ethernet/IP (Ethernet mit 100 Mbit/s)

Overhead in Byte je Botschaft	Max. Nutzdatenrate in MB/s		
	Nutzdaten je Botschaft		
	64 Byte	254 Byte	1024 Byte
Ethernet 100 Mbit/s	30	7,5	10,7
IPv4 auf Ethernet	30 + 20 = 50	6,3	10,0
IPv6 auf Ethernet	30 + 40 = 70	5,5	9,4
UDP/IPv6 auf Ethernet	30 + 40 + 8 = 78	5,2	9,2
TCP/IPv6 auf Ethernet	30 + 40 + 20 = 90	4,8 ^a	8,9 ^a
			11,4 ^a

^a Echtzeittauglichkeit siehe Text.

für die Anwendung weitgehend unsichtbar und sind von dieser nur wenig beeinflussbar. Das Zeitverhalten des TCP Protokolls ist daher nicht streng deterministisch und dürfte für Echtzeitanwendungen nur bedingt geeignet sein.

Ethernet bzw. UDP/IP auf Ethernet dagegen sind prinzipiell einsetzbar, wobei allerdings klar sein muss, dass die in Tab. 3.14 angegebenen Nutzdatenraten für Steuergeräte eher theoretischen Wert haben dürften. Noch deutlich mehr als schon bei FlexRay wird in der Praxis nicht das physikalische Übertragungssystem sondern der Software-Protokollstack in den Steuergeräten die tatsächlichen Latenzen und den erreichbaren Durchsatz bestimmen. Während auf der Ethernet-Ebene wie bei CAN und FlexRay die Botschaftsheader und Prüfsummen weitgehend automatisch durch die Hardware im Kommunikationscontroller erzeugt werden, müssen die IP, UDP und TCP Header sowie die eigentlichen Nutzdaten durch die Software zeitrichtig bereitgestellt und weiterverarbeitet werden. Ohne große RAM-Speicher zum Zwischenspeichern der Botschaften und schnelle Mikroprozessoren zur Berechnung und Überprüfung der Prüfsummen kann die mögliche Bandbreite nicht genutzt werden. Während solche Rechenleistungen bei High-End-Infotainmentsystemen auch im Fahrzeug schon verfügbar sind, müssen Steuergeräte im Antriebs- und Fahrwerksbereich noch deutlich aufgerüstet oder weite Teile des Protokollstacks in Hardware implementiert werden. Und in letzter Konsequenz sollten ähnlich wie bei Nutzfahrzeugen mit SAE J1939/71 (siehe Abschn. 4.5) endlich auch im PKW-Bereich die übertragenen Echtzeitdaten und nicht nur das Übertragungssystem standardisiert werden.

3.6 Normen und Standards zu Kapitel 3

CAN	Bosch: CAN Specification Version 2.0, 1991, www.can.bosch.com
	ISO 11898-1 Road Vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling, 2003 und 2006, www.iso.org
	ISO 11898-2 Road Vehicles – Controller area network (CAN) – Part 2: High-speed medium access unit, 2003, www.iso.org
	ISO 11898-3 Road Vehicles – Controller area network (CAN) – Part 3: Low-speed fault tolerant medium dependent interface, 2006, www.iso.org
	ISO 11898-4 Road Vehicles – Controller area network (CAN) – Part 4: Time triggered communication, 2004, www.iso.org
	ISO 11898-5 Road Vehicles – Controller area network (CAN) – Part 5: High-speed medium access unit with low-power mode, 2007, www.iso.org
	ISO 11898-6 Road Vehicles – Controller area network (CAN) – Part 6: High-speed medium access unit with selective wakeup-functionality, 2013, www.iso.org
	ISO 11992-1 bis -4 Road vehicles – Interchange of digital information on electrical connections between towing and towed vehicles, 2003 bis 2008
	SAE J2284/1 High Speed CAN for Vehicle Applications at 125 Kbps, 2002, www.sae.org
	SAE J2284/2 High Speed CAN for Vehicle Applications at 250 Kbps, 2002, www.sae.org
	SAE J2284/3 High Speed CAN for Vehicle Applications at 500 Kbps, 2010, www.sae.org
LIN	SAE J2411 Single Wire CAN Network for Vehicle Applications, 2000, www.sae.org
	ISO 15765 siehe Kap. 5
	SAE J1939 siehe Kap. 4
	Bosch: CAN with Flexible Data-Rate. Specification V1.0, 2012, www.can.bosch.com
	LIN Specification Package Revision 1.3. 2002, www.lin-subbus.org
	LIN Specification Package Revision 2.0. 2003, www.lin-subbus.org
	LIN Specification Package Revision 2.1, 2006, www.lin-subbus.org
MOST	LIN Specification Package Revision 2.2 A, 2010, www.lin-subbus.org
	ISO 17987-1 bis -7 Road vehicles – Local Interconnect Network (LIN), in Vorbereitung, www.iso.org
	SAE J2602/1 LIN Network for Vehicle Applications, 2012, www.sae.org
	SAE J2602/2 Conformance Testing for SAE J2602 LIN, 2012, www.sae.org
	SAE J2602/3 File Structures for a Node Capability File (NCF), 2010, www.sae.org
	MOST Specification Framework, Rev. 1.1, 1999, www.mostcooperation.com
	MOST Specification Rev. 2.5, 2006, www.mostcooperation.com
	MOST Specification Rev. 3.0, 2008, www.mostcooperation.com
	MOST Specification Rev. 3.0 E2, 2010, www.mostcooperation.com
MOST	MOST Dynamic Specification Rev. 3.0, 2013, www.mostcooperation.com
	MOST MAMAC Specification Rev. 1.1, 2005, www.mostcooperation.com
	MOST Core Compliance Test Specification Rev. 1.2, 2006, www.mostcooperation.com
	MOST Function Block Library für Netzketten mit Slave- oder Masterfunktion, Audio Verstärker, Disk Player usw. Jeweils einzeln versioniert. www.mostcooperation.com

FlexRay	<p><i>FlexRay Consortium</i> (www.flexray.com)</p> <p>FlexRay Communications System – Protocol Specification: Version 2.1 Revision A, 2005; Version 3.0.1, 2010</p> <p>FlexRay Communications System – Electrical Physical Layer Specification: Version 2.1 Revision B, 2006; Version 3.0.1, 2010</p> <p>FlexRay Communications System – Electrical Physical Layer Application Notes: Version 2.1 Revision B, 2006; Version 3.0.1, 2010</p> <p>FlexRay Communications System – Preliminary Node-Local Bus Guardian Specification Version 2.0.9, 2005, www.flexray.com</p> <p>FlexRay Communications System – Preliminary Central Bus Guardian Specification Version 2.0.9, 2005, www.flexray.com</p> <p><i>ISO</i> (www.iso.org)</p> <p>ISO 17458 Road vehicles - FlexRay communications system</p> <p>Part 1: General information and use case definition, 2013</p> <p>Part 2: Data link layer Specification, 2013</p> <p>Part 3: Data link layer conformance test specification, 2013</p> <p>Part 4: Electrical physical layer specification, 2013</p> <p>Part 5: Electrical physical layer conformance test specification, 2013</p> <p>ISO 10681 Road vehicles – Communication on FlexRay, siehe Kap. 5</p> <p>SAE J2813 FlexRay for Vehicle Applications, noch nicht veröffentlicht, www.sae.org</p>
Ethernet	<p>IEEE 802.3 Local and metropolitan area networks. Part 3: Carrier Sense Multiple Access with Collision Detection Access Method (CSMA/CD) and Physical Layer Specifications, 2008, www.ieee.org</p> <p>IEEE 802.1Q Local and metropolitan area networks – Media Access Control Bridges and virtual Bridged Local Area Networks, 2011, www.ieee.org</p> <p>IEEE 802.1AS Local and metropolitan area networks – Timing and Synchronisation for Time-Sensitive Applications in Bridged Local Area Networks, 2011</p> <p>IEEE 802.1BA Local and metropolitan area networks – Audio Video Bridging Systems, 2011, www.ieee.org</p> <p>IEEE 1588 Precision Clock Synchronisation Protocol for Networked Measurement and Control Systems, 2008, www.ieee.org</p> <p>IEEE 1722 Layer 2 Transport Protocol for Time-Sensitive Applications in a Bridged Local Area Network, 2011, www.ieee.org</p> <p>IEEE 1733 Layer 3 Transport Protocol for Time-Sensitive Applications in Local Area Networks, 2011, www.ieee.org</p>
TCP/IP	Internet Protokolle IP, TCP, UDP u. a. siehe Kap. 8
UDP	Internet Engineering Task Force IETF, www.ietf.org

Literatur

- [1] K. Etschberger: Controller Area Network. Hanser Verlag, 3. Auflage, 2006
- [2] W. Lawrenz, N. Obermöller: CAN Controller Area Network. VDE Verlag, 5. Auflage, 2011
- [3] Hartwich, F.: CAN with Flexible Data-Rate. 13. International CAN Conference 2012. www.can-bosch.com

-
- [4] K. Tindell, A. Burns, A. Wellings: Calculating CAN Message Response Times. *Control Engineering Practice*, Heft 8, 1995, S. 1163-1169
 - [5] A. Burns, R. Davis, R. Bril, J. Lukkien: CAN Schedulability Analysis: Refuted, Revised and Revisited. *Real-Time Systems Journal*, Springer Verlag, Heft 3, 2007, S. 239-272
 - [6] T. Nolte, H. Hansson, C. Norström, S. Punnekkat: Using Bit-stuffing Distributions in CAN Analysis. *IEEE Real-time Embedded Systems Workshop*, London, Dez. 2001
 - [7] S. Punnekkat, H. Hansson, C. Norström: Response Time Analysis under Errors for CAN. *IEEE Real-Time Technology and Applications Symposium*, Juni 2000, S. 258-265
 - [8] N. Navet, F. Simonot-Lion: *Automotive Embedded Systems Handbook*. CRC Press, 1. Auflage, 2009
 - [9] K. Tindell: Adding Time-Offsets to Schedulability Analysis. Technical Report YCS 221, University of York, 1994
 - [10] A. Grzemba, H.-C. von der Wense: *LIN-Bus*. Franzis Verlag, 1. Auflage, 2005
 - [11] J. Berwanger, M. Peteratzinger, A. Schedl: *FlexRay-Bordnetz für Fahrdynamik und Fahrerassistenzsysteme*. Sonderausgabe electronic automotive, 2008
 - [12] M. Rausch: *FlexRay. Grundlagen, Funktionsweise, Anwendung*. Hanser Verlag, 1. Auflage, 2007
 - [13] D. Paret: *FlexRay and its Applications*. John Wiley & Sons Verlag, 2. Auflage, 2012
 - [14] N.N.: *E-Ray, FlexRay IP-Module, User's Manual Rev. 1.2.7*. Robert Bosch GmbH, 2009, <http://www.bosch-semiconductors.de>
 - [15] N.N.: *FlexRay Kommunikationscontroller MFR 4310 und Mikrocontroller mit FlexRay-Controller MPC556x und S12XF*. www.freescale.com
 - [16] T. Pop, P. Pop, P. Eles, Z. Pong, A. Andrei: Timing Analysis of the FlexRay Communication Protocol. *Real-Time Systems Journal*, Springer Verlag, Heft 1-3, 2008, S. 205-235
 - [17] A. Grzemba (Hrsg.): *MOST – The Automotive Multimedia Network*. Franzis Verlag, 2. Auflage, 2011 und www.mostcooperation.com
 - [18] A. Meroth, B. Tolg: *Infotainmentsysteme im Kraftfahrzeug*. Vieweg+Teubner Verlag, 1. Auflage, 2007
 - [19] G. Schnell: *Bussysteme in der Automatisierungs- und Prozesstechnik*. Springer-Vieweg-Verlag, 8. Auflage, 2011
 - [20] R. Kreifeldt: *AVB for Automotive Use White Paper*. 2009, AVnu Resource Library, www.avnu.org
 - [21] R. Stevens: *TCP/IP Illustrated*. Addison-Wesley, 3 Bände, 2002
 - [22] E. Hall: *Internet Core Protocols*. O'Reilly, 2000

Alle aktuellen Bussysteme definieren Data Link Layer Botschaften mit begrenzter Nutzdatenlänge. So kann CAN nach ISO 11898 (Abschn. 3.1) maximal 8 Nutzdatenbytes, FlexRay nach ISO 17458 (Abschn. 3.3) maximal 254 Byte Daten je Botschaft übertragen. Die Übertragung größerer Datenblöcke sowie Adressierungsverfahren, die die Weiterleitung von Botschaften über Gateways zwischen verschiedenen Netzen erlauben, sind zunächst nicht spezifiziert. Für Diagnoseanwendungen oder zur Flash-Programmierung von Steuergeräten sind derartige Übertragungen aber notwendig.

Daher ist für CAN bzw. FlexRay ein Protokoll für die Transportschicht notwendig, d. h. für die Ebenen 3 und 4 des OSI-Schichtenmodells. Dessen Aufgaben sind:

- Übertragung von Datenblöcken, die größer sind als die *natürliche Blockgröße* des Data Link Layers (*Segmentierung, Desegmentierung*)
- Flusskontrolle (*Flow Control*), d. h. Steuerung des zeitlichen Abstands der Datenblöcke so, dass der Empfänger nicht überlastet wird, und zeitliche Überwachung der gesamten Kommunikation (*Timeout*)
- Weiterleiten von Botschaften in fremde Netze, d. h. in ein Netz mit einem anderen Adressraum oder einer anderen Adressierungsmethode als das lokale Netz.

Leider haben sich im Laufe der Zeit mehrere Lösungen etabliert. Neben dem in ISO 15765-2 standardisierten Transportprotokoll, das bei KWP 2000 on CAN und UDS on CAN verwendet wird, werden bei VW/Audi die auf einer frühen OSEK COM Version beruhenden Protokolle TP 1.6 bzw. TP 2.0 eingesetzt. Auch das bei Nutzfahrzeugen übliche SAE J1939 enthält eine eigene Transportschicht. Für FlexRay hat AUTOSAR ein inzwischen in ISO 10681-2 genormtes Transportprotokoll vorgeschlagen, das sich an ISO 15765-2 orientiert, aber mehrfach verändert wurde und nicht mehr wirklich aufwärts kompatibel ist. Für CAN FD ist zu erwarten, dass das bisherige CAN-Transportprotokoll ISO 15765-2 ebenfalls modifiziert werden muss.

4.1 Transportprotokoll ISO TP für CAN nach ISO 15765-2

Wie in Abschn. 3.1 dargestellt, können in einer CAN-Botschaft maximal 8 Datenbytes übertragen werden. Die Nachricht wird durch einen *Message Identifier* gekennzeichnet. Eine direkte Adressierung von Sender und Empfänger erfolgt nicht. Um das ursprünglich nur für K-Line-Verbindungen definierte KWP 2000-Diagnoseprotokoll, bei dem größere Datenblöcke sowie eine Sender- und Empfängeradressierung notwendig sind, auch über CAN-Verbindungen implementieren zu können, wurde in ISO 15765-2 ein entsprechendes Transportprotokoll (TP) definiert, das sich aber nicht nur für Diagnose- sondern auch für andere Aufgaben verwenden lässt. Das Protokoll beschreibt:

- Die Abbildung von 4 Transportprotokoll-Botschaftstypen auf CAN-Botschaften.
- Die Aufteilung (*Segmentierung*) von größeren Datenblöcken bis zu 4095 Byte auf einzelne CAN-Botschaften, die auf der Empfängerseite wieder zusammengesetzt werden (*Desegmentierung*).
- Ein Verfahren zur Flussteuerung zwischen Sender und Empfänger.
- Die Abbildung der KWP 2000-Sende-Empfänger-Adressen auf CAN-Message-IDs.
- Dienste (Services) zur Kommunikation zwischen Application und Transport Layer.

4.1.1 Botschaftsaufbau

Aus Sicht der Anwendung wird zwischen Adressierungsinformation (AI) und den eigentlichen Daten (*SDU... Service Data Unit*) unterschieden. Die Adressierungsinformation (*Normal Addressing*) wird bei der Übertragung auf den CAN-Identifier abgebildet (Abb. 4.1). Bei der so genannten erweiterten oder gemischten Adressierung (Extended und Mixed Addressing), die bei Netzen mit Gateways verwendet wird, enthält zusätzlich das erste Nutzdatenbyte der CAN-Botschaft eine weitere Adressinformation. Als erstes bzw. nächstes Byte der CAN-Botschaft fügt das Transportprotokoll ein zusätzliches Steuerbyte (*PCI... Protocol Control Information*) hinzu. Dieses zeigt an, wie die Nutzdaten zu interpretieren sind. In den restlichen Bytes der insgesamt max. 8 Byte langen CAN-Botschaft folgen die Nutzdatenbytes der Anwendung.

Falls alle Anwendungsdaten in die verbleibenden Datenbytes der CAN-Botschaft passen, wird ein sogenannter *Single Frame SF* gesendet. Dabei besteht die *Protocol Control Information PCI* aus einem Byte, wobei Bit 7...4 auf 0 gesetzt werden, Bit 3...0 enthalten die Anzahl der folgenden Nutzdatenbytes (*DL... Single Frame Data Length*).

Falls die Anwendungsdaten nicht in eine einzelne CAN-Botschaft passen, müssen die Anwendungsdaten auf mehrere CAN-Botschaften verteilt werden (*Segmentierung, Multi Frame Botschaften*). Zuerst wird ein sogenannter *First Frame FF* gesendet. Dabei besteht PCI aus zwei Bytes. Die oberen 4 bit des ersten PCI-Bytes enthalten den Wert 1 h, die unteren 4 bit des ersten PCI-Bytes und das gesamte zweite PCI-Byte, insgesamt 12 bit, enthalten die Länge der Nutzdaten (*FF_DL... First Frame Data Length*). Trotz des etwas verwirren-

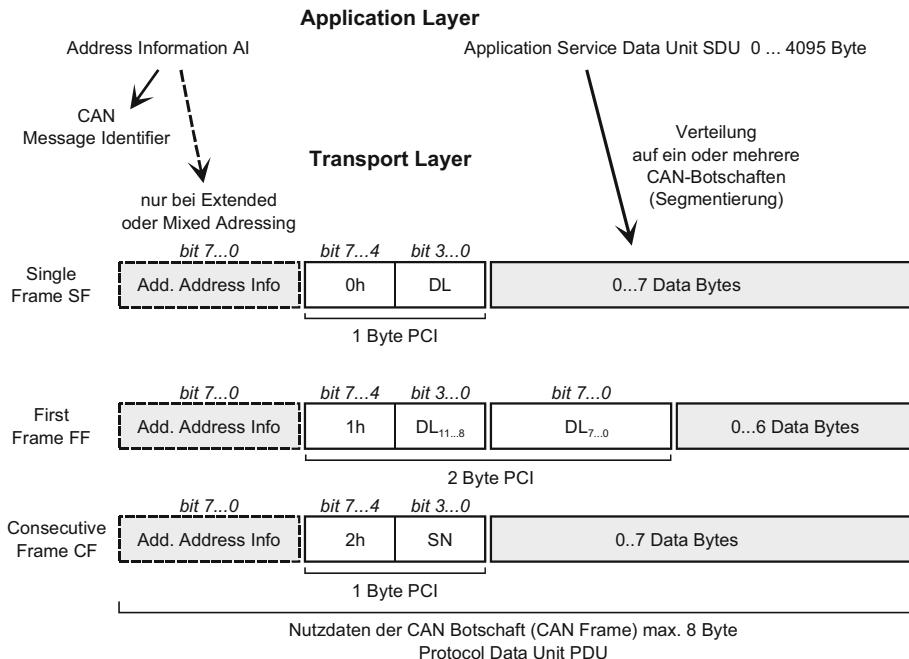


Abb. 4.1 Zuordnung zwischen Anwendungsdaten und CAN-Botschaften

den Namens *First Frame Data Length* ist dabei die Gesamtlänge der Anwendungsdaten und nicht nur die Länge der im *First Frame* selbst enthaltenen Daten gemeint. Anschließend folgen sogenannte *Consecutive Frames CF*, bis die gesamten Anwendungsdaten versendet worden sind. Bei einem *Consecutive Frame* besteht PCI aus einem Byte, dessen obere 4 bit den Wert 2 h und die unteren 4 bit die sogenannte Sequenznummer SN enthalten. Die zu einem Anwendungsdatensatz gehörenden CAN-Botschaften werden in aufsteigender Reihenfolge durchnummiert, wobei der *First Frame* die Sequenznummer 0, der erste *Consecutive Frame* die Nummer 1, der nächste Frame die Nummer 2 usw. erhält.

Da das Feld für die Sequenznummer innerhalb des PCI Bytes nur 4 bit lang ist, wird die Sequenznummer Modulo 16 übertragen.

Durch das bei jeder CAN-Botschaft übertragene, mindestens 1 Byte lange PCI-Feld reduziert sich die Nutzdatenrate auf im günstigsten Fall $7/8 = 87\%$ der in Abschn. 3.1 für CAN angegebenen Maximalwerte, d. h. ca. 26 KB/s (CAN mit einer Bitrate von 500 kBit/s und 29 bit Identifizieren).

4.1.2 Flusssteuerung, Zeitüberwachung und Fehlerbehandlung

Single Frame Botschaften werden vom Sender spontan gesendet. Da der CAN Data Link Layer die Fehlersicherung und Übertragungswiederholung selbstständig übernimmt, erfolgt auf der Ebene des Transportprotokolls keine Bestätigung des Empfängers. Der Mindestabstand zwischen aufeinanderfolgenden *Single Frame* Botschaften ist nicht spezifiziert.

Für segmentierte *Multi Frame* Botschaften ist der in Abb. 4.2 dargestellte Ablauf vorgesehen. Der Sender sendet die erste Datenbotschaft (*First Frame*) und wartet dann auf eine *Flow Control FC* Botschaft vom Empfänger (Abb. 4.3). In der *Flow Control* Botschaft teilt der Empfänger dem Sender mit, welche Anzahl BS (*Block Size*, $BS = 1 \dots 255$) von weiteren CAN-Botschaften (*Consecutive Frames*) er unmittelbar aufeinander empfangen kann und welche Mindestzeit ST_{min} (*Separation Time*) der Sender zwischen den einzelnen Botschaften pausieren muss. Nachdem der Sender die BS weiteren Botschaften gesendet hat, wartet er auf eine erneute *Flow Control* Botschaft vom Empfänger, bevor er weiter sendet. Dieser Ablauf wiederholt sich, bis alle Anwendungsdaten übertragen sind. Falls der Empfänger $BS = 0$ signalisiert, darf der Sender beliebig viele *Consecutive Frames* senden, ohne nochmals auf eine *Flow Control* Botschaft warten zu müssen. Statt einer *Flow Control* Botschaft mit $FS = 0\text{ h}$ (*Flow State Clear To Send*) kann der Empfänger auch $FS = 1\text{ h}$ (*Wait*) senden. Damit wird der Sender aufgefordert, mit dem Senden solange zu warten, bis er eine weitere *Flow Control* Botschaft erhält. Diese kann ihn dann zur Fortsetzung des Sendens oder zum weiteren Warten auffordern. Um ein endloses Warten zu verhindern, ist die Anzahl der aufeinanderfolgenden *Wait*-Anforderungen begrenzt. Ist die Datenlänge der segmentierten Botschaft für den Empfänger zu groß, meldet er dies mit $FS = 2\text{ h}$ (*Overflow*).

Der Sender führt beim Warten auf eine *Flow Control* Botschaft, der Empfänger beim Warten auf den nächsten *Consecutive Frame* eine Timeout-Überwachung (Defaultwert 1 s) durch. Im Fall eines Timeouts oder beim Empfang einer Botschaft mit einer falschen Sequenznummer erfolgt eine Fehlermeldung an den eigenen Application Layer, aber keine Fehlermeldung zur Gegenseite, d. h. falls Sender und Empfänger Fehlermeldungen austauschen sollen, muss dies auf der Ebene der Anwendung geschehen. Die fehlerhaft empfangenen Daten selbst werden vom Empfänger ignoriert.

Im Unterschied zu üblichen CAN-Botschaften für Steuer- und Regelaufgaben, die meist von mehreren Steuergeräten empfangen und weiterverarbeitet werden (1:n Kommunikation), ist dies hier nur bei *Single Frame* Botschaften möglich. Segmentierte *Multi Frame* Botschaften dagegen müssen in der Praxis über den CAN Identifier genau ein einziges Steuergerät als Empfänger ansprechen (1:1 Kommunikation), da die *Flow Control* Botschaften zur Flusssteuerung nur von einem einzigen Empfänger zurückgesendet werden dürfen (siehe auch Abschn. 4.1.5).

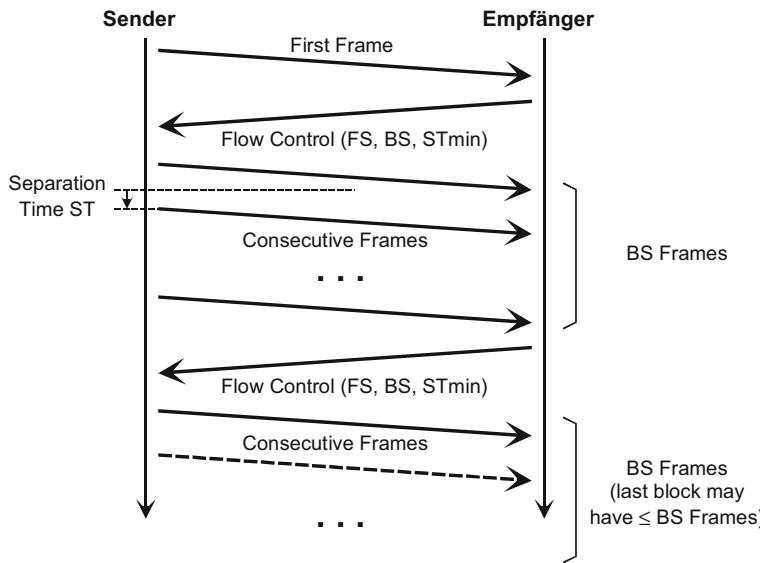


Abb. 4.2 Flusssteuerung bei Multi Frame Botschaften

4.1.3 Dienste für die Anwendungsschicht (Application Layer Services)

ISO 15765-2 spezifiziert zwar keine Programmier-API, definiert aber zumindest die in einer derartigen API mindestens zu implementierende Funktionalität (Abb. 4.4). Leider fehlen dabei präzise Angaben über die Handhabung des Sende- bzw. Empfangspuffers im Zusammenspiel zwischen Application Layer und Transport Layer:

- *Service Data.request*: Aufforderung zum Senden eines Datenblocks von bis zu 4095 Bytes an eine vorgegebene Adresse. Die Anwendung erhält eine Bestätigung bzw. Fehlermeldung *Data.confirm*, wenn der Block vollständig gesendet wurde, so dass der Sendepuffer für die nächste Botschaft wieder verwendet bzw. freigegeben werden kann. Durch geignetes, in der Norm aber nicht definiertes Zusammenspiel zwischen Application und

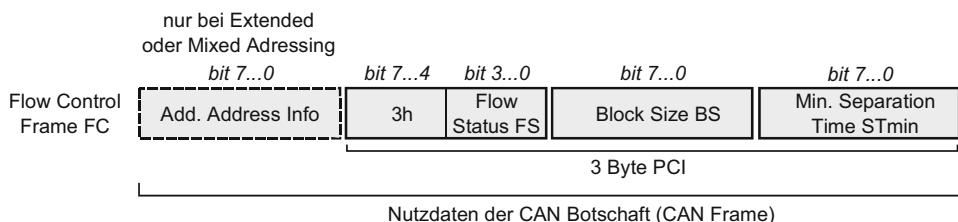


Abb. 4.3 Flow Control CAN-Botschaft (FS = CTS = 0 h Clear To Send)

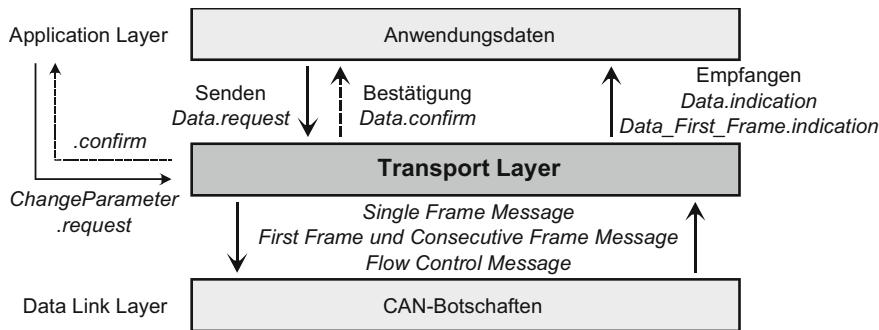


Abb. 4.4 Schnittstelle zwischen den Schichten

Transport Layer ist es auch möglich, Botschaften auf der Senderseite inkrementell zu erstellen und zu senden.

- *Service Data.indication:* Die Anwendung erhält eine Mitteilung, dass ein vollständiger Datenblock von bis zu 4095 Byte empfangen wurde. Zusätzlich erhält die Anwendung eine Information, wenn der erste Datenblock einer *Multi Frame* Botschaft empfangen wurde (*Data_First-Frame.indication*). Mit dem *First Frame* ist die Länge der zu erwartenden Botschaft bereits bekannt, so dass der Transport Layer oder der Application Layer einen ausreichend dimensionierten Empfangspuffer bereitstellen kann. Falls bei Datenblöcken im KB-Bereich kein ausreichend großer Empfangspuffer verfügbar ist, signalisiert der Empfänger dem Sender über die *Flow Control* Botschaften die tatsächliche Puffergröße. Er muss die Daten dann verarbeiten, sobald der Puffer gefüllt ist. Der dafür notwendige Dienst, mit dem der Transport Layer dem Application Layer mitteilen kann, dass der Empfangspuffer voll ist, wird in der Norm nicht spezifiziert. Problematisch bei der inkrementellen Verarbeitung auf der Empfangsseite ist, dass die Botschaft zu diesem Zeitpunkt noch nicht vollständig ist, so dass bei einem später folgenden Übertragungsfehler, z. B. durch einen Fehler auf der Senderseite, ein unvollständiger, aber schon weiter verarbeiteter Datensatz vorliegt. Dieses Problem, das zum Beispiel auftritt, wenn bei einem Flash-Programmievorgang die Übertragung der zu programmierenden Daten abgebrochen wird, obwohl bereits ein Teil des Flash-ROMs neu programmiert wurde, muss durch geeignete Maßnahmen im Application Layer gelöst werden.
- *Service ChangeParameter.request:* Die Anwendung kann die Parameter für BS und ST im Bereich von 1...255 verändern und erhält eine Bestätigung bzw. Fehlermeldung *ChangeParameter.confirm* vom Transport Layer. Die Gegenseite erfährt von dieser Änderung erst durch die nächste Flow Control Botschaft.

Die Timeout-Werte sind nach ISO 15765-2 konstant 1 s. Die Anzahl der maximal akzeptablen, aufeinander folgenden Flow Control Wait-Botschaften ist eine Applikationskonstante für den Sender und den Empfänger. Sie wird zwischen Sender und Empfänger nicht automatisch ausgehandelt.

Die Entscheidung, ob ein Steuergerät gleichzeitig senden und empfangen kann (Voll-Duplex) oder nicht (Halb-Duplex), bleibt der Implementierung überlassen. In jedem Fall soll ein Steuergerät aber in der Lage sein, gleichzeitig mit mehreren anderen Steuergeräten, d. h. Geräten mit unterschiedlichen Adressen, zu kommunizieren. Wobei ein Empfänger den Sender aber gegebenenfalls über *Flow Control Wait* Botschaften bremsen kann.

4.1.4 Protokoll-Erweiterungen

Die Norm definiert, wie dargestellt, 4 Botschaftstypen (High Nibble des PCI-Bytes 0 h...3 h). Die verbleibenden Werte sind für zukünftige Erweiterungen reserviert. Botschaften mit unbekanntem PCI-Wert oder anderen Fehlern wie einer falschen Sequenznummer sollen vom Empfänger ignoriert werden.

4.1.5 Adressierung bei KWP 2000/UDS – Zuordnung von CAN Identifiern

Beim Diagnoseprotokoll KWP 2000 werden im Application Layer der Sender (*Source Address*) und der Empfänger (*Target Address*) mit je einer 1 Byte großen Adresse gekennzeichnet. ISO 15765-2 und -3 geben Empfehlungen, wie diese logische Adressierung auf CAN Message Identifier abgebildet werden soll. Für die Diagnose abgasrelevanter Steuergeräte werden die Identifier in ISO 15765-4 fest vorgegeben, zu weiteren Einzelheiten siehe Abschn. 5.1.3.

Weiterhin enthält der Anhang der Norm einen Vorschlag, wie die CAN Message Identifier festgelegt werden sollen, wenn das ISO 15765-2 Transportprotokoll in einem CAN-Netz verwendet wird, in dem Steuergeräte zusätzlich mit Botschaften nach dem SAE J1939-Standard kommunizieren, wie er bei Nutzfahrzeugen üblich ist.

4.1.6 Bandbreite des ISO TP für CAN

Während in der On-Board-Kommunikation vor allem die Latenzzeiten (Abschn. 3.1.7 und 3.3.6) für die Datenübertragung wichtig sind, interessiert bei der Flash-Programmierung (Abschn. 9.4), dem Software-Update von Steuergeräten, vor allem die Bandbreite des Bussystems. Weil dabei große Datenblöcke übertragen werden müssen, kommt grundsätzlich ein Transportprotokoll zum Einsatz. Dadurch verringert sich die effektive Bandbreite des Bussystems gegenüber der maximalen Nutzdatenrate des reinen *Data Link Layers* [1].

Die Übertragungsdauer T_{Frame} einer einzelnen CAN-Botschaft hängt nach Gl. 3.2 neben der Bitrate von der Anzahl der Nutzdaten, der Länge des *Message Identifiers* und der Zahl der notwendigen *Stuff-Bits* ab. Für die *Stuff-Bits* können nur Minimal- und Maximalwert angegeben werden, da die tatsächliche Anzahl von den Werten des *CAN Identifiers*

und der Datenbytes der jeweiligen Botschaft abhängt. Bei der Übertragung großer Datenmengen kann angenommen werden, dass (eventuell mit Ausnahme der letzten Botschaft) alle CAN Botschaften $n_{\text{Data,DLL}} = 8$ Datenbyte enthalten und dass die mittlere Anzahl der Stuff-Bits zwischen dem Minimal- und dem Maximalwert nach Gl. 3.3 liegt. Bei einer Bitrate von 500 kbit/s ergibt sich bei 11 bit *Message Identifier* dann für die mittlere Übertragungsdauer einer einzelnen Botschaft $T_{\text{Frame}}^* = 246 \mu\text{s}$ und auf Ebene des CAN *Data Link Layers* eine mittlere Nutzdatenrate

$$f_{\text{Data,DLL}}^* = n_{\text{Data,DLL}} / T_{\text{Frame}}^* = 31,8 \text{ KB/s} \quad (4.1)$$

Dabei wird vorausgesetzt, dass der Sender CAN Botschaften schnellstmöglich sendet, d. h. im nach CAN Spezifikation zulässigen Minimalabstand von 3 Bit-Takten. In der Praxis benötigen CAN Kommunikationscontroller bzw. die zugehörige Ansteuersoftware oft länger. Außerdem wurde angenommen, dass keine anderen CAN Botschaften den Bus blockieren. Diese Voraussetzung ist in einem typischen Flash-Programmierszenario realisierbar, da die Diagnoseprotokolle KWP 2000 und UDS Diagnosedienste bieten, mit denen die normale Steuergerätekommunikation während des Programmievorgangs abgeschaltet werden kann (siehe Kap. 5).

Durch den Overhead des Transportprotokolls verringert sich die effektive Datenrate gegenüber Gl. 4.1. Würde man eine unsegmentierte Übertragung mit ISO TP *Single Frame* SF Botschaften (Abb. 4.1) verwenden, könnte man mit jeder Botschaft nur $n_{\text{Data,SF}} = 7$ Byte Nutzdaten senden, da jeweils zusätzlich das *PCI Byte* des Transportprotokolls mit übertragen werden muss. Die Nutzdatenrate aus Sicht der höheren Protokollschichten verringert sich somit auf

$$f_{\text{Data,unseg}} = n_{\text{Data,SF}} / n_{\text{Data,DLL}} \cdot f_{\text{Data,DLL}}^* = 7/8 \cdot f_{\text{Data,DLL}}^*. \quad (4.2)$$

Dabei wurde angenommen, dass für die Adressierung ausschließlich der CAN *Message Identifier* verwendet wird (*Normal Addressing*). Setzt man dagegen das *Mixed* oder *Extended Addressing* ein, verliert man ein weiteres Nutzdatenbyte je Botschaft.

Sinnvoll ist das Transportprotokoll aber eigentlich nur, wenn man statt der unsegmentierten eine segmentierte Übertragung mit einer Folge von *First Frame FF* und *Consecutive Frame CF* Botschaften verwendet (Abb. 4.2). Der FF kann nur 6, die CF-Botschaften wieder 7 Nutzdatenbyte enthalten. Damit lassen sich Datenblöcke bis zu $n_{\text{Datablock}} \leq 4095$ Byte übertragen. Die notwendige Anzahl von Botschaften ist

$$N_{\text{FF}} = 1 \quad \text{und} \quad N_{\text{CF}} = \lceil (n_{\text{Datablock}} - 6 \text{ Byte}) / 7 \text{ Byte} \rceil \quad (4.3)$$

Zusätzlich sind noch ein oder mehrere *Flow Control FC* Botschaften notwendig. Deren Anzahl hängt von der Puffergröße des Empfängers ab, die dieser dem Sender über den *Block Size BS* Parameter der FC Botschaft mitteilt:

$$\begin{aligned} \text{bei unbeschränktem Puffer BS} = 0 : \quad N_{\text{FC}} &= 1 \\ \text{bei endlicher Puffergröße BS} > 0 : \quad N_{\text{FC}} &= \lceil N_{\text{CF}} / BS \rceil \end{aligned} \quad (4.4)$$

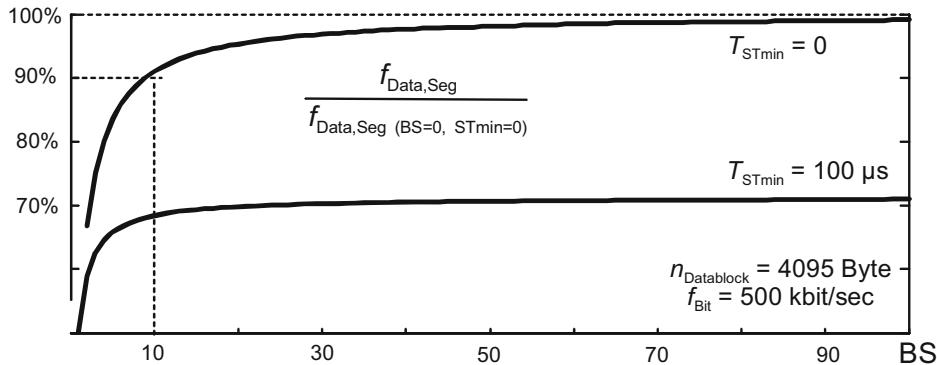


Abb. 4.5 Abhängigkeit der effektiven Datenrate von den *Flow Control* Parametern

Vernachlässigt man, dass die FC Botschaften und eventuell auch die letzte CF Botschaft etwas kürzer sind als die übrigen Botschaften, kann man die Nutzdatenrate für die segmentierte Übertragung auf der Ebene des Transportprotokolls abschätzen:

$$f_{\text{Data,seg}} \approx \frac{n_{\text{Datablock}}}{(N_{\text{FF}} + N_{\text{CF}} + N_{\text{FC}}) \cdot T_{\text{Frame}}^*} \quad (4.5)$$

Bei einem Datenblock von $n_{\text{Datablock}} = 4095$ Byte und einem unbeschränkten Puffer sind insgesamt 587 CAN Botschaften notwendig. Mit einem Bittakt von 500 kBit/s ergibt sich eine effektive Datenrate von 27,7 KB/s, d. h. das ISO Transportprotokoll erreicht im besten Fall etwa 87 % des Wertes des CAN *Data Link Layers* nach Gl. 4.1. Bei kleinen Puffergrößen BS auf der Empfängerseite bricht die Datenrate deutlich ein, für $BS > 10$ ist der Verlust mit weniger als 10 % dagegen noch akzeptabel (Abb. 4.5).

Bisher wurde angenommen, dass der Empfänger über den Parameter *Separation Time* T_{STmin} in der FC Botschaft keine Pause zwischen den einzelnen CF Botschaften eingefordert hat. Für $T_{\text{STmin}} > 0$ verlängert sich die Übertragung der CF Botschaften. Als Anzahl N_{ST} der einzufügenden Pausen ergibt sich

$$\begin{aligned} \text{für } BS = 0 : \quad N_{\text{ST}} &= N_{\text{CF}} - 1 \\ \text{für } BS > 0 : \quad N_{\text{ST}} &= N_{\text{CF}} - 1 - \lceil N_{\text{CF}}/BS - 1 \rceil \end{aligned} \quad (4.6)$$

Damit reduziert sich die Datenrate auf

$$f_{\text{Data,seg}} \approx \frac{n_{\text{Datablock}}}{(N_{\text{FF}} + N_{\text{CF}} + N_{\text{FC}}) \cdot T_{\text{Frame}}^* + N_{\text{ST}} + T_{\text{STmin}}} \quad (4.7)$$

Bereits bei $T_{\text{STmin}} = 100 \mu\text{s}$ bricht die Datenrate massiv ein (Abb. 4.5).

4.2 Transportprotokoll für FlexRay nach ISO 10681-2

Das bisher einzige Transportprotokoll für FlexRay-Bussysteme wurde innerhalb der AUTOSAR-Initiative entwickelt (siehe Kap. 8). Nachdem es ursprünglich vollständig aufwärts kompatibel zu ISO 15765-2 war, wurde es mittlerweile mehrfach modifiziert, um die Unterschiede zwischen CAN und FlexRay besser zu berücksichtigen. FlexRay sichert die Daten bei der Übertragung zwar mit einer CRC-Prüfsumme, liefert aber im Gegensatz zu CAN weder eine Empfangsbestätigung noch wird die Übertragung im Fehlerfall automatisch wiederholt. Diese Mechanismen wurden daher innerhalb des FlexRay Transportprotokolls nachgebildet. Inzwischen befindet sich das Protokoll als ISO 10681-2 im Standardisierungsprozess. Im Folgenden sollen vor allem die Unterschiede des aktuellen Standardvorschlags zu ISO TP für CAN beschrieben werden. Die früheren AUTOSAR-Varianten wurden in der zweiten und dritten Auflage dieses Buches erläutert.

4.2.1 Botschaftsaufbau und Adressierung

Im Gegensatz zu CAN, wo die Sender- und Empfängeradressierung auf den CAN-Message-Identifier abgebildet wird, muss die Adressinformation bei FlexRay innerhalb der Botschaft übertragen werden, da es nicht sinnvoll ist, jeder Verbindung einen separaten FlexRay-Zeitschlitz zuzuordnen. Danach folgen die *Protocol Control Information* (PCI) und anschließend die Nutzdaten.

Die Empfängeradresse (*Target Address TA*) und die Senderadresse (*Source Address SA*) werden dabei jeweils als zwei Byte lange Werte übertragen (Abb. 4.6). Die Mehr-Byte-Werte werden im Big-Endian-Format dargestellt, d. h. mit dem höchstwertigen Byte zuerst. Die Adressen liegen am Anfang des FlexRay-Nutzdatenfelds, d. h. in derjenigen Position, die bei FlexRay im dynamischen Segment als eine Art optionale *Message ID* fungiert, falls das *Payload Preamble Bit* im FlexRay Botschaftsheader gesetzt ist (siehe Abschn. 3.3.2).

Diese *Message ID* kann durch den Kommunikationscontroller automatisch ausgewertet werden kann. Auf diese Weise kann ein FlexRay Zeitschlitz für Übertragungen von einem Sender zu mehreren Empfängern verwendet werden, wobei die Empfänger über das Adressfeld ähnlich wie bei CAN eine schnelle, hardwaregestützte Akzeptanzfilterung durchführen können. Speziell für Gateways, die mit sehr vielen Geräten kommunizieren,

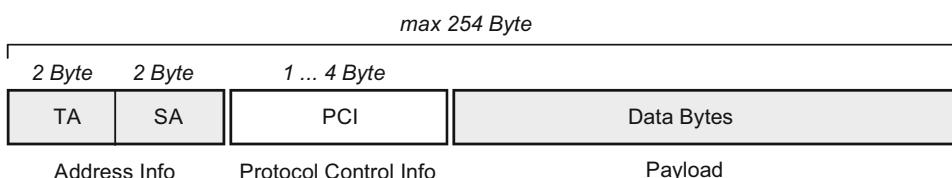


Abb. 4.6 PDU (Protocol Data Unit)-Format der FlexRay-TP-Botschaften

Tab. 4.1 Format des PCI-Felds

Frame-Typ	1. Byte	2. Byte	3. Byte	4. Byte
	Bit 7...4	Bit 3...0		
Start Frame STFU	4h	ACK=0	FPL	ML
Start Frame STFA	4h	ACK=1	FPL	ML
Consecutive Frame CF1	5h	SN	FPL	
Consecutive Frame CF2	6h	SN	FPL	
ConsecutiveFrame CFEOB	7h	SN	FPL	
Flow Control FC_CTS	8h	FS = 3	BC	BfS
Flow Control FC_ACK	8h	FS = 4	RET/ACK	BP
Flow Control FC	8h	FS	FS=5 Wait, FS=6 Abort, FS=7 Overflow	
Last Frame LF	9h	0	FPL	ML
FPL...Frame Payload Length ML ...Message Length SN ...Sequence Number BfS...Buffer Size ACK..Acknowledge				
BfS...Buffer Size FS...Flow Status BC...Bandwidth Control BP...Byte Position RET...Retry				

kann dadurch die Anzahl der FlexRay Zeitschlüsse reduziert werden, die z. B. für die Diagnosekommunikation, insbesondere für Softwareupdates (*Flashen*) von Steuergeräten reserviert werden müssen [1].

4.2.2 Verbindungsarten und Übertragungsablauf

Die von CAN bekannten *Single*, *Consecutive* und *Flow Control Frames* wurden für FlexRay modifiziert, der *First Frame* entfällt (Tab. 4.1). Es sind segmentierte und unsegmentierte Übertragungen ohne und mit Bestätigung möglich, wobei auch der Fall abgedeckt wird, dass die Gesamtlänge der Daten zu Beginn einer segmentierten Übertragung noch nicht bekannt ist. Der Ablauf ist wie folgt:

- Jede Übertragung beginnt mit einem *Start Frame*. Durch das ACK Bit wird signalisiert, ob der Sender eine Empfangsbestätigung (*Acknowledge*) wünscht (STFA mit ACK = 1) oder nicht (*Unacknowledged*, STFU mit ACK = 0). Segmentierte Übertragungen, die bei CAN zur Unterscheidung mit einem *First Frame* beginnen mussten, können anhand der Längenangaben von unsegmentierten Übertragungen unterschieden werden. Im FPL-(*Frame Payload Length*)-Feld steht die Länge der innerhalb des *Start Frames* über-

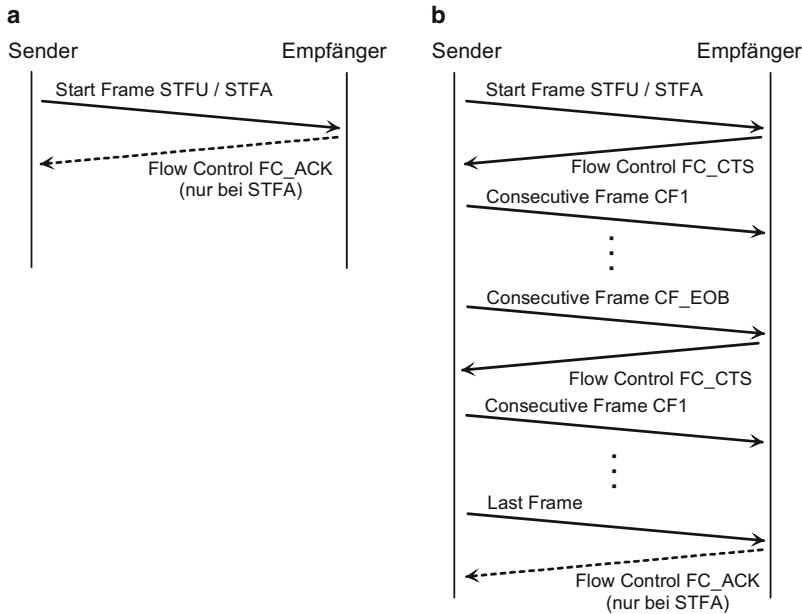


Abb. 4.7 Botschaftssequenzen bei ISO 10681-2 bei fehlerfreier Übertragung, **a** Unsegmented unacknowledged/acknowledged, **b** Segmented unacknowledged/acknowledged

tragenen Nutzdaten in Byte, im ML-(*Message Length*)-Feld die Länge der Nutzdaten der gesamten Botschaft. Sind beide Werte gleich, ist keine Segmentierung notwendig.

- Falls eine Bestätigung angefordert wurde, antwortet der Empfänger mit einem *Flow Control Frame FC_ACK* (Abb. 4.7a), der mit RET/ACK = 0 den erfolgreichen Empfang bestätigt. Mit RET/ACK = 1 kann der Empfänger den Sender auffordern, die Übertragung zu wiederholen. Dabei signalisiert er im BP Feld, ab welchem Byte die Daten fehlerhaft waren.
- Ist im *First Frame* der ML Wert größer als FPL, folgen auf den *First Frame* weitere Daten in *Consecutive Frames* (Abb. 4.7b). Die Maximallänge ist 64 kByte. Auf den *First Frame* muss der Empfänger zuerst mit einem *Flow Control Frame FC_CTS* antworten, in dem er gegebenenfalls mit dem Parameter BC eine Bandbreitensteuerung fordert (siehe Abschn. 4.2.3). Die Blockgröße, hier *Buffer Size BfS* genannt, gibt die maximale Anzahl von Bytes an, die in den folgenden *Consecutive Frames* übertragen werden darf, bevor der Sender auf einen weiteren *Flow Control Frame* warten muss. Dass der Sender einen weiteren *Flow Control Frame* erwartet, signalisiert er, indem er am Ende eines Blocks einen *End of Block Consecutive Frame CF_EOB* statt eines normalen *Consecutive Frames* sendet. Die Sequenznummer SN der *Consecutive Frames* hat dieselbe Aufgabe wie bei CAN.
- Zuletzt überträgt der Sender einen *Last Frame*. Falls dieser noch Nutzdaten enthält, wird deren Anzahl im FPL Feld angezeigt, während in ML nochmals die Datenlänge der ge-

Tab. 4.2 Bandwidth Control Parameter BC

Bit 7 ... 3	Bit 2 ... 0
Maximum Number of PDUs per Cycle MNPC	Separation Cycle Exponent

samten segmentierten Botschaft übertragen wird. Dies dient als Prüfmöglichkeit, zumal es auch möglich ist, im *Start Frame* zunächst $ML = 0$ zu senden, falls die Gesamtlänge zu Beginn der Übertragung noch nicht bekannt ist. Wenn der Sender mit dem *Start Frame* STFA eine Bestätigung angefordert hat, folgt diese am Ende der gesamten Sequenz als *Flow Control Frame* FC_ACK.

- ISO definiert zwei Typen von *Consecutive Frames*. Die segmentierte Übertragung beginnt stets mit dem Frame-Typ CF1. Wird eine Sendewiederholung vom Empfänger gefordert, so wechselt der Frame-Typ auf CF2. Bei einer weiteren Sendewiederholung wechselt der Frame-Typ zurück zu CF1 usw.
- Ähnlich wie bei CAN kann der Empfänger den Sender mit einem *Flow Control Frame* mit $FS = 5$ zum Warten auffordern, mit $FS = 7$ einen Pufferüberlauf melden oder mit $FS = 6$ die Übertragung abbrechen. Entsprechend sind wieder Zeitüberwachungen und maximale Übertragungswiederholungen definiert.

Die Werte für die PCI-Kennung sind so festgelegt, dass sie nicht mit den Werten nach ISO 15765-2 überlappen. Es ist daher, wie in den ersten AUTOSAR Versionen vorgesehen, auch weiterhin möglich, alternativ die CAN-kompatiblen PCI-Formate zu verwenden, wenn auf die FlexRay-spezifischen Erweiterungen verzichtet werden kann.

4.2.3 Bandbreitensteuerung

Durch die in Abschn. 4.2.1 beschriebene Adressierungsmethodik ist es möglich, die Bandbreite für die Kommunikation zwischen zwei Teilnehmern dynamisch zu verändern. Reserviert man in einem FlexRay-Zyklus mehrere Zeitschlüsse für Transportprotokollbotschaften, so kann man diese Zeitschlüsse bündeln, um z. B. bei der Flash-Programmierung große Datenmengen mit hoher Bandbreite an ein oder mehrere Steuergerät zu übertragen. Allerdings besteht die Gefahr, dass der Pufferspeicherbereich des Empfängers nicht ausreicht, wenn der Zeitabstand zwischen den einzelnen Botschaften zu kurz wird. Daher verwendet das FlexRay-Transportprotokoll wiederum eine Flusssteuerung über die *Flow Control Frames FC_CTS*, die aber gegenüber dem CAN-Transportprotokoll modifiziert wurde (Tab. 4.1). Neben seiner Pufferlänge BfS sendet der Empfänger den sogenannten *Bandwidth Control BC* Parameter, der eine ähnliche Rolle spielt wie die *Separation Time* bei CAN (Tab. 4.2).

Wird im BC Byte der Wert MNPC = 0 gesetzt, darf der Sender die maximale mögliche Bandbreite nutzen. Andernfalls gibt MNPC an, wie viele *Consecutive Frames* der Empfänger innerhalb ein und desselben FlexRay-Zyklus verarbeiten kann. Der Wert SCexp

definiert, welcher Mindestabstand zwischen den Frames liegen muss. Die geforderte Anzahl von FlexRay-Zyklen berechnet sich dabei zu $SC = 2^{SC_{exp}} - 1$. Durch geeignete Wahl dieser Werte lassen sich je nach Anforderungen sowohl eine kontinuierliche Busauslastung als auch ein Burst-Betrieb realisieren.

4.2.4 Fehlerbehandlung und Implementierungshinweise

Die Behandlung von Fehlern wie falscher Datenlänge, Sequenznummer oder Überschreitung von Zeitschranken erfolgt bei unbestätigten Übertragungen ähnlich wie bei ISO TP, indem der Empfänger die fehlerhafte Botschaft einfach ignoriert. Im Fall der bestätigten Übertragung erhält der Sender eine Rückmeldung über die *Flow Control FC_ACK* Botschaft und kann die Übertragung wiederholen.

ISO 10681-2 fordert, dass Sender und Empfänger in der Lage sein müssen, gleichzeitig mehrere Verbindungen abzuwickeln, solange sich diese in der Kombination von Sender- und Empfängeradresse unterscheiden. Die Norm definiert allerdings keine Prioritäten für die parallel aktiven Verbindungen. Eine effiziente Implementierungsmethode für die Bandbreitenzuweisung ist das *Round Robin Verfahren* (Abb. 4.8). Dabei werden die einzelnen Sendepuffer für die Botschaften als Warteschlangen (*Queue*) reihum bedient. Für die Pufferverwaltung schlägt die Norm zwei Steuergrößen vor. Jede Warteschlange besitzt eine Füllstandsanzeige (*Filling Level*), die anzeigt, wie viele Transportprotokoll-Botschaften je Verbindung anstehen. Ein weiterer Zähler (*TX Pending Counter*) dokumentiert, wie viele Botschaften von der Transportschicht (in der Norm als *Communication Layer* bezeichnet) an den *Data Link Layer* übergeben wurden, aber noch nicht versendet sind. Steht dieser Zähler zu Beginn eines FlexRay-Zyklus nicht auf Null, so wurden im vorhergehenden Zyklus nicht alle Botschaften dieser Warteschlange versendet. Dies kann hauptsächlich im dynamischen Segment vorkommen, wenn eine Botschaft am Ende des FlexRay-Zyklus durch Botschaften in früheren Zeitschlitzten soweit verzögert wird, dass sie nicht mehr im laufenden Zyklus versendet werden kann (siehe Abschn. 3.3.6). In diesem Fall muss die Transportschicht warten, bis der *TX Pending Counter* wieder den Wert 0 hat, da sonst möglicherweise *Consecutive Frames* in der falschen Reihenfolge versendet werden.

Wie die Schnittstelle zwischen Transportschicht und der übergeordneten Anwendung gestaltet werden kann und wie das Transportprotokoll in den gesamten AUTOSAR Protokollstapel eingebunden ist, wird im Kap. 8 beschrieben.

4.2.5 Bandbreite des FlexRay Transportprotokolls

Bei FlexRay hängt die Nutzdatenrate auf Ebene des *Data Link Layers* davon ab, mit welcher Periodendauer T_{Cycle} der Kommunikationszyklus (*Cycle*) arbeitet, wie viele Botschaften je

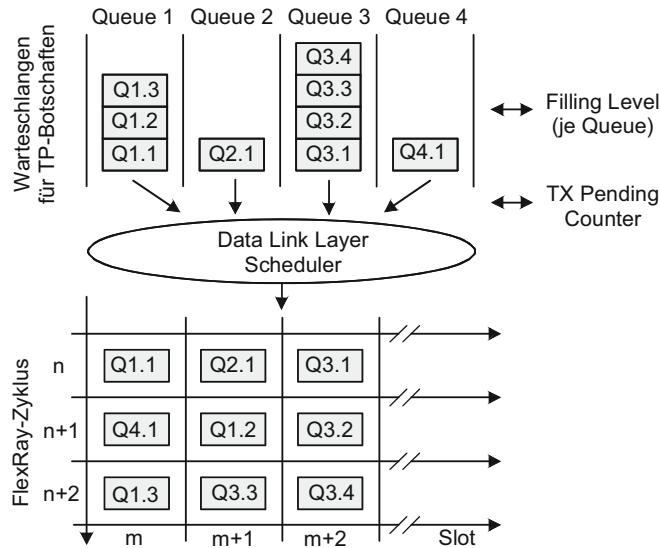


Abb. 4.8 Round Robin Scheduling von Transportprotokoll-Botschaften

Zyklus $N_{\text{FramesPerCycle}}$ für eine bestimmte Übertragung reserviert sind und wie viele Datenbytes $n_{\text{Data,DLL}}$ in einer Botschaft gesendet werden können:

$$f_{\text{Data,DLL}} = N_{\text{FramesPerCycle}} \cdot n_{\text{Data,DLL}} / T_{\text{Cycle}} \quad (4.8)$$

Die Anzahl $n_{\text{Data,DLL}}$ der Datenbytes einer Botschaft bestimmt umgekehrt die Dauer der zugehörigen *Slots* und damit zusammen mit der Aufteilung des Kommunikationszyklus in ein statisches und ein dynamisches Segment die Obergrenze für $N_{\text{FramesPerCycle}}$. Die gegenseitigen Abhängigkeiten wurden im Abschn. 3.3.6 dargestellt. Würde man beispielsweise den maximal möglichen Wert $n_{\text{Data,DLL}} = 254$ Byte je Botschaft wählen, wäre diese bei einer Bitrate von 10 Mbit/s nach Tab. 3.9 etwa 270 μs lang. In ein 2 ms langes dynamisches Segment passen dann $N_{\text{FramePerCycle}} = 7$ Botschaften je Zyklus, falls das Segment vollständig für diese Übertragung genutzt werden kann. Bei einer Zykluszeit von 5 ms ergibt sich dann eine Nutzdatenrate von 347 KB/s. Soll mit *Cycle Multiplexing* gearbeitet werden, ist für T_{Cycle} nicht die Periodendauer des Basis-Kommunikationszyklus sondern die effektive Multiplex-Periodendauer einzusetzen. Bereits beim *Multiplexen* mit zwei Zyklen würde sich die Datenrate halbieren.

Beim FlexRay-Transportprotokoll verliert man bei unsegmentierter Übertragung insgesamt 8 Byte je Botschaft für das Adress- und PCI-Feld des Transportprotokolls (*Start Frame* nach Abb. 4.6 und Tab. 4.1). Wegen des im Beispiel mit 254 Byte sehr großen FlexRay-Datenfelds verringert sich die Nutzdatenrate lediglich um 3 %:

$$f_{\text{Data,unseg}} = (n_{\text{Data,DLL}} - 8 \text{ Byte}) / n_{\text{Data,DLL}} \cdot f_{\text{Data,DLL}} = (254 - 8) / 254 \cdot f_{\text{Data,DLL}} \quad (4.9)$$

So lange Botschaften sind üblicherweise nur im dynamischen Segment möglich. Da im statischen Segment alle *Slots* gleich lang sein müssen, legt man dieses in der Regel nur für kurze Botschaften mit beispielsweise $n_{\text{Data,DLL}} = 16$ Byte aus. Dort ist das Transportprotokoll mit seinem *Overhead* von bis zu 8 Byte nicht mehr effektiv.

Auch die segmentierte Übertragung (Abb. 4.7) beginnt bei FlexRay mit einem *Start Frame SF*. In den folgenden *Consecutive Frames CF* gehen wegen des kürzeren PCI-Feldes 6 Byte verloren. In der letzten Botschaft, dem *Last Frame LF*, sind es wieder 8 Byte. Die Anzahl der notwendigen Botschaften für einen größeren Datenblock ist

$$\begin{aligned} N_{\text{FF}} &= 1 \\ N_{\text{LF}} &= 1 \\ N_{\text{CF}} &= \lceil (n_{\text{Datablock}} - 2 \cdot (n_{\text{Data,DLL}} - 8) \text{ Byte}) / (n_{\text{Data,DLL}} - 6) \text{ Byte} \rceil \end{aligned} \quad (4.10)$$

Vernachlässigt man zunächst die Flusssteuerung, so kann man die Anzahl der notwendigen Kommunikationszyklen für die Übertragung abschätzen zu

$$N_{\text{Cycles}} \approx \lceil \frac{N_{\text{FF}} + N_{\text{CF}} + N_{\text{LF}}}{N_{\text{Frames per cycle}}} \rceil \quad (4.11)$$

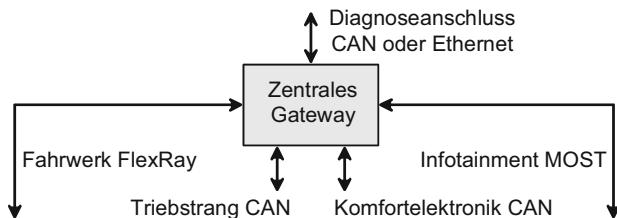
Als grober Wert für die Datenrate ohne Flusssteuerung ergibt sich dann

$$f_{\text{Data,seg}} \approx \frac{n_{\text{Datablock}}}{N_{\text{Cycles}} \cdot T_{\text{Cycle}}} \quad (4.12)$$

Um einen Datenblock von 64 KB zu übertragen, sind in der obigen FlexRay-Konfiguration 263 Botschaften notwendig, die in 38 Zyklen mit etwa 340 KB/s gesendet werden.

Berücksichtigt man die Flusssteuerung, hängt die Anzahl der notwendigen *Flow Control FC* Botschaften wieder von der Puffergröße des Empfängers ab. Da die FC Botschaften nur 8 Byte lang sind, ist ihre Übertragungsdauer mit $T_{\text{Frame,FC}} \approx 20 \mu\text{s}$ gegenüber der im Beispiel 254 Byte langen CF Datenbotschaften mit $T_{\text{Frame,CF}} \approx 270 \mu\text{s}$ fast vernachlässigbar. Viel kritischer ist die Frage, wo die *Slots* für die FC Botschaften innerhalb des Kommunikationszyklus relativ zu den *Slots* für die Datenbotschaften platziert werden und wie man die Blockgröße des Empfängers relativ zur Anzahl der Datenbotschaften je Zyklus wählt. Passen die Festlegungen nicht zusammen, z. B. eine Blockgröße des Empfängers von 4 Botschaften bei 7 möglichen Botschaften je Kommunikationszyklus, so verliert man durch das Warten auf die FC Botschaften unter Umständen vollständige Kommunikationszyklen. Wenn die einzelnen Steuergeräte dann auch noch unterschiedliche Blockgrößen oder Mindestabstände zwischen den Datenbotschaften anfordern und die Flash-Programmierung mit der für den normalen Fahrbetrieb optimierten Zyklus-Konfiguration erfolgen soll, ist die reale Datenrate oft erheblich schlechter als der theoretische Wert nach Gl. 4.12. Eine detaillierte Beschreibung dieser Probleme und möglicher Abhilfemaßnahmen findet sich in [1].

Abb. 4.9 Busstruktur aktueller Fahrzeuge



In der Regel erfolgt die Flash-Programmierung über den normalen Diagnosezugang des Fahrzeugs. Von den internen Bussystemen ist der Diagnosezugang meist durch ein zentrales *Gateway*-Steuergerät entkoppelt, an dem die internen Bussysteme des Fahrzeugs zusammenlaufen (Abb. 4.9). In einer solchen Kette bestimmt das Bussystem mit der niedrigen Datenrate die effektive Bandbreite. Die Übertragung zu einem Steuergerät am internen FlexRay-Bus erfolgt also bestenfalls mit der Datenrate des CAN-Busses, wenn ein gewöhnlicher CAN-Diagnoseanschluss verwendet wird. Umgekehrt beschleunigt ein Ethernet-Diagnoseanschluss die Datenübertragung zu einem Steuergerät am internen CAN-Datenbus nicht. Da viele dieser Busse unterschiedliche *Frame* Formate und Bitraten haben, können die ankommenden Botschaften nicht einfach durchgeleitet, sondern müssen im *Gateway* zwischengespeichert und umformatiert werden. Weil im Normalbetrieb oft sehr viele Botschaften verarbeitet werden und Gateways für die Flash-Programmierung oft nicht umkonfiguriert werden können, obwohl dabei die normale Kommunikation in der Regel abgeschaltet wird, stellen kleine Datenpuffer, d. h. kleine Blockgrößen für die Transportprotokolle, einen Flaschenhals dar. Die Verarbeitungszeit im *Gateway* ist für die Übertragung der eigentlichen Datenbotschaften (FF, CF) unkritisch [1], wirkt aber bei der Flusssteuerung wie eine vergrößerte *Separation Time* (siehe Abb. 4.5).

4.3 Transportprotokoll TP 2.0 für CAN

Im Gegensatz zum ISO-Transportprotokoll handelt es sich hierbei um ein verbindungsorientiertes Protokoll. Dabei werden dynamisch exklusive Verbindungen zwischen zwei CAN-Teilnehmern, sogenannte *Kanäle*, eingerichtet und nach Beendigung des Datenaustausches wieder aufgelöst. Die gesamte Kommunikation gliedert sich in drei Phasen:

- **Verbindungsaufbau** (Eröffnung eines Transportkanals)
- **Datenübertragung** innerhalb des Transportkanals
- **Verbindungsabbau** (Schließen des Transportkanals).

Für den Verbindungsaufbau, aber auch für einige andere Dienste, besteht darüber hinaus die Möglichkeit zur 1:n Kommunikation (Broadcast). Im Gegensatz zu ISO-TP, bei dem die CAN Identifier statisch konfiguriert sind, werden die CAN Identifier für die Kanäle dynamisch vergeben.

TP 2.0 betreibt ein sehr aufwendiges Verfahren zur Flusskontrolle. Dabei wird jede vollständig empfangene Botschaft oder ein Block von mehreren Botschaften vom Empfänger quittiert (Handshake). Weitere wesentliche Merkmale des Transportprotokolls TP 2.0 sind:

- Die Abbildung von 10 verschiedenen Transportprotokoll-Botschaftstypen auf CAN-Botschaften.
- Die Aufteilung (Segmentierung) von beliebig großen Datenblöcken auf einzelne CAN-Botschaften, die auf der Empfängerseite wieder zusammengesetzt werden (Desegmentierung).
- Automatische Sendewiederholung bei Timeout-Fehlern.

4.3.1 Adressierung und CAN Message Identifier

Jedes Steuergerät besitzt eine fahrzeugweit eindeutige logische Adresse, die nach einem festen Schema einem CAN Message Identifier zugeordnet ist. Zusätzlich gibt es logische Adressen, unter denen mehrere Steuergeräte, z. B. alle Steuergeräte des Antriebsstrangs, zu einer Steuergerätegruppe zusammengefasst werden. Auch derartige Gruppenadressen werden jeweils einem CAN-Identifier zugeordnet. Die derartig festgelegten CAN-Identifier werden für den Verbindungsaufbau sowie andere Broadcast-Dienste verwendet und als *Eröffnungs-ID* bezeichnet:

$$\text{Eröffnungs-ID} \quad = \quad \text{Basis-ID} \quad + \quad \text{Logische Adresse des Gerätes}$$

Eindeutige Geräteadresse bzw. Broadcast-Adresse für alle Steuergeräte am Antriebsstrang-Bus, am Infotainment-Bus, am Komfort-Bus oder alle Steuergeräte im Fahrzeug

Beim Aufbau einer dynamischen Verbindung (*Kanal*) handeln Sender und Empfänger individuelle CAN-Identifier aus, die sogenannten *Kanal IDs*, welche dann für die anschließende Datenübertragung bis zum Abbau der jeweiligen Verbindung verwendet werden.

4.3.2 Broadcast-Botschaften

Broadcast-Botschaften werden grundsätzlich mit der *Eröffnungs-ID* als CAN Message Identifier gesendet und haben alle denselben schematischen Aufbau (Abb. 4.10). Alle Botschaften besitzen nur sieben Datenbytes, das achte Byte wird nicht übertragen. Im ersten Datenbyte wird die logische Adresse des Ziel-Steuergerätes gesendet. Auf diese Weise ist die Weiterleitung der Botschaften über Gateways möglich. Der *Opcode* im zweiten Byte kennzeichnet den Typ der Botschaft, anschließend folgt ein als *Service-ID SID* bezeichnetes Byte, das einen der im Application Layer definierten Dienste auswählt, sowie dessen Parameter. Bei den Diensten kann es sich sowohl um herstellerspezifische Funktionen als

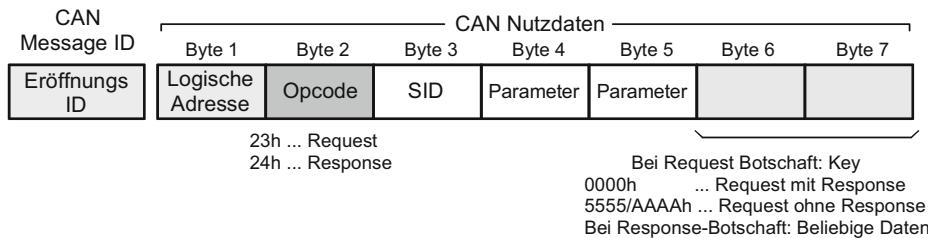


Abb. 4.10 Broadcast-Botschaften bei TP 2.0

auch um die durch die KWP 2000- bzw. UDS-Normen definierten Funktionen handeln, die in Kap. 5 beschrieben werden.

TP 2.0 unterscheidet Broadcast-Botschaften, bei denen eine Antwort (Response) vom Empfänger erwartet wird, und solche, bei denen keine Antwort notwendig ist.

Bei einer *Broadcast Request* Botschaft wird das Opcode-Byte auf den Wert 23 h gesetzt, bei der *Broadcast Response* Antwortbotschaft auf 24 h. Falls eine Antwort vom Empfänger erwartet wird, setzt der Sender im *Broadcast Request* die Bytes 6 und 7, den sogenannten *Key*, auf 0000 h. Der Sender sendet die *Request* Botschaft insgesamt fünf Mal innerhalb von 100 ms, der Empfänger führt den zugehörigen Dienst aus, sobald er die Botschaft einmal erkannt hat, und muss innerhalb von maximal 500 ms antworten. Die Reaktion bei Ausbleiben der Antwort ist nicht spezifiziert.

Broadcast Request Botschaften, bei denen der Sender keine Antwort erwartet, werden ebenfalls innerhalb von 100 ms fünf Mal gesendet, wobei in den Bytes 6 und 7 als *Key* alternierend 5555 h und AAAAh gesendet wird. Der Empfänger führt den Dienst erst dann aus, wenn er die Botschaft mit jedem der beiden *Keys* mindestens einmal innerhalb der 100 ms empfangen hat.

Bei Diensten, welche die Empfänger der Botschaft in einen besonderen Systemzustand versetzen, müssen die *Request* Botschaften gegebenenfalls anschließend in größerem Abstand, z. B. 1 s, wiederholt werden, um diesen Zustand aufrecht zu erhalten. Erkennt ein Steuengerät während dieser sogenannten Retrigger-Phase einen Timeout, fällt es aus dem Sondermodus wieder in den Normalmodus zurück.

4.3.3 Dynamischer Kanalaufbau und Verbindungsmanagement

Für die eigentliche Datenübertragung zwischen zwei Geräten müssen zunächst eine logische Verbindung, ein sogenannter Kanal, aufgebaut und die für die spätere Übertragung verwendeten CAN-Identifier ausgehandelt werden. Der dynamische Kanalaufbau erfolgt wiederum mit dem *Eröffnungs-ID* als CAN Message Identifier und stellt einen Spezialfall der Broadcast-Botschaften dar (Abb. 4.11).

Die beiden CAN-ID Felder RX ID und TX ID lassen nur Platz für 11 bit-CAN Identifier, 29 bit-Identifier werden gegenwärtig nicht unterstützt. Die unteren 8 Bit des Identifiers

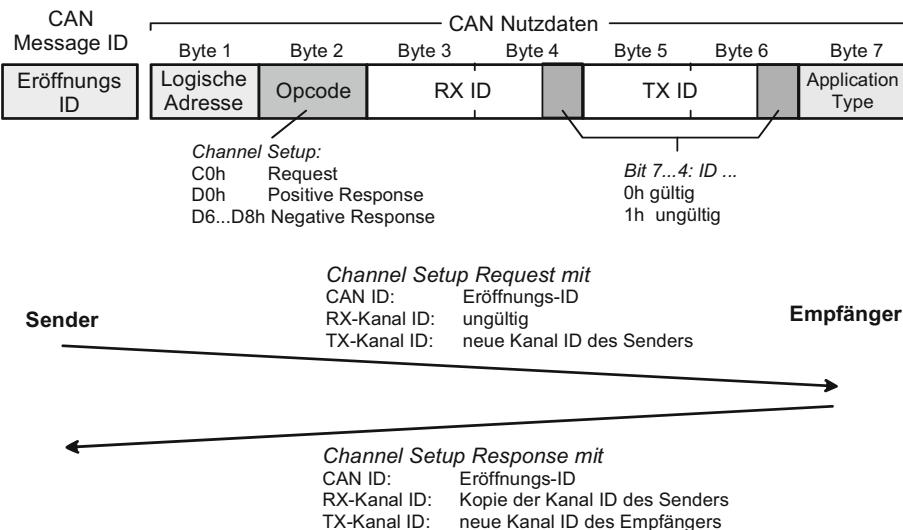


Abb. 4.11 Botschaften für den dynamischen Kanalaufbau bei TP 2.0

werden in Byte 3 bzw. Byte 5 gesendet, die oberen 3 Bit in der unteren Hälfte der Bytes 4 bzw. 6. Die oberen 4 Bit dieser beiden Bytes zeigen mit dem Wert 0 h an, dass die CAN-Identifier gültig sind, mit dem Wert 1 h, dass der jeweilige CAN-Identifier ungültig ist. Dies ist für den Handshake-Ablauf beim Verbindungsaufbau notwendig:

- Der Sender sendet einen *Channel Setup Request* (Opcode C0 h) mit derjenigen CAN-ID im Feld TX ID, mit der er zukünftig Daten empfangen will. Das Feld RX ID markiert er als ungültig.
- Der Sender antwortet, falls er die Kanaleröffnung akzeptiert, mit einer positiven *Channel Setup Response* (Opcode D0 h), wobei er im Feld TX ID jetzt diejenige CAN-ID zurücksendet, mit der er zukünftig Daten empfangen will. Außerdem wiederholt er im Feld RX ID die CAN-ID, die ihm der Sender gerade als seine zukünftige *Kanal-ID* mitgeteilt hat. Das Vertauschen der Werte in den RX ID und TX ID Feldern erscheint zunächst verwirrend. Es ist aber leicht zu verstehen, wenn man sich vergegenwärtigt, dass im TX ID Feld stets diejenige CAN ID steht, an die der jeweilige Empfänger seine Antworten oder eigenen Botschaften an den anderen Partner senden (TX ... transmit) muss.
- Falls der Empfänger die Verbindung ablehnt, sendet er die *Channel Setup Response* mit einem der Opcodes D6 h ... D8 h, um eine dauerhafte oder zeitweise Ablehnung der Verbindung zu signalisieren. Der Timeout beim Verbindungsaufbau beträgt 50 ms. Der Sender darf den Verbindungsversuch bis zu 10mal wiederholen. Im Feld *Application Type* wird definiert, welche Anwendungsfunktion des Steuergerätes angesprochen werden soll. Der Wert 01 h spezifiziert bei VW/Audi beispielsweise die KWP 2000-Diagnose.

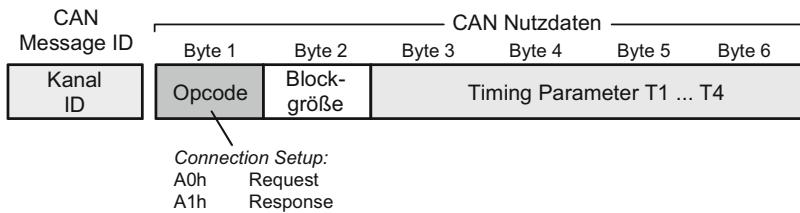


Abb. 4.12 Parametrierung von Verbindungskanälen bei TP 2.0

Nach erfolgreichem Verbindungsauftbau werden alle weiteren Botschaften bis zum Verbindungsabbau dann mit den beiden ausgetauschten *Kanal-IDs* übertragen.

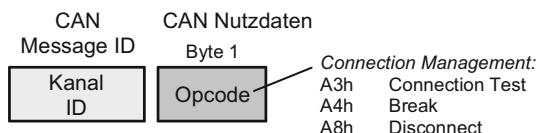
Als nächstes werden die Verbindungsparameter für den neuen Kanal ausgehandelt. Dazu sendet der Sender eine *Connection Setup Request* Botschaft mit den Verbindungsparametern des Senders (Abb. 4.12).

Das Byte 2 enthält dabei die Blockgröße, d. h. die Anzahl der CAN-Botschaften, die hintereinander empfangen werden sollen, bevor der Empfang durch eine ACK-Botschaft (siehe unten) bestätigt werden muss. Zulässige Blockgrößen sind 1...15. Die Bytes 3 bis 6 enthalten diverse Timeout-Parameter, z. B. die maximal zulässige Zeit T1 zwischen der letzten CAN-Botschaft eines Blocks und der zugehörigen Bestätigung durch die ACK-Botschaft, oder die minimal notwendige Zeit T3 zwischen zwei aufeinander folgenden CAN-Botschaften. Die Zeiten werden mit 6 bit Auflösung sowie einem 2 bit Skalierungsfaktor übertragen, der angibt, ob der Wert mit 100 µs, 1 ms, 10 ms oder 100 ms multipliziert werden muss. Die Zeiten T2 und T4 sind für Erweiterungen vorgesehen und werden im Normalfall als FFh übertragen. Der Empfänger antwortet in der *Connection Setup Response* Botschaft mit den entsprechenden Werten für seine Seite. Anschließend ist der Kanal bereit für die eigentliche Datenübertragung, die im folgenden Abschnitt beschrieben wird.

Der abschließende Abbau der Verbindung erfolgt durch eine *Disconnect* Botschaft (Abb. 4.13). Der Abbau wird durch den Empfänger bestätigt, in dem er seinerseits mit einer *Disconnect* Botschaft antwortet. Anschließend dürfen keine Botschaften mehr mit den dieser Verbindung zugeordneten *Kanal-IDs* gesendet und empfangen werden.

In komplexen Fahrzeugen werden häufig Gateways zur Anbindung des Diagnosetesters sowie zur Verbindung mehrerer Bussysteme eingesetzt. Da diese Gateways in der Regel nur Botschaften weiterleiten, die Botschaftsinhalte selbst aber nicht analysieren, sind sie auch nicht in der Lage, die für einen Kanal ausgehandelten Timeout-Zeiten zu überwachen. Sie besitzen daher ein festes Zeitraster für die Timeout-Überwachung. Dies kann unter Um-

Abb. 4.13 Weitere Botschaften für Verbindungsabbau und Verbindungssteuerung



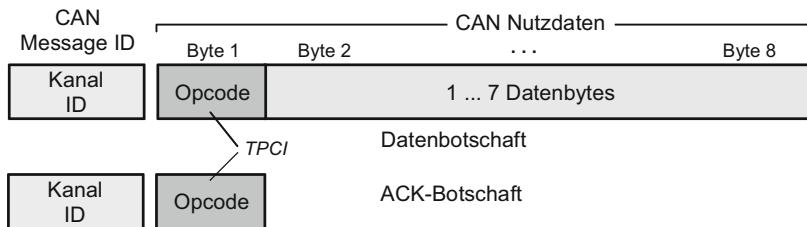


Abb. 4.14 Datenbotschaft und ACK-Botschaft bei TP 2.0

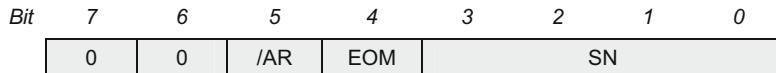


Abb. 4.15 Aufbau des Transport Control Bytes TPCI des Senders

ständen kleiner sein als die ausgehandelten Zeiten für den dynamischen Kanal. Es ist somit möglich, dass ein Gateway aufgrund dieser festen Zeiten einen Timeout für einen Kanal detektiert, obwohl die beiden Kommunikationspartner sich zeitlich korrekt verhalten. Für diesen Fall existieren *Connection Test* Botschaften, die im Zeitraster des Gateways gesendet werden. Der Empfänger antwortet mit einer *Connection Setup Response* Botschaft. Das Gateway erkennt ein aus seiner Sicht korrektes Timing und lässt den Kanal weiterhin geöffnet.

4.3.4 Datenübertragung

Die Datenübertragung erfolgt blockweise, wobei der Empfänger den Empfang jedes Blocks bestätigen muss. Ein Block besteht aus einer oder mehreren aufeinanderfolgenden CAN-Botschaften. Die maximale Blockgröße wurde beim Aufbau des Kanals mit den *Connection Setup* Botschaften ausgehandelt. Die Bestätigung durch den Empfänger erfolgt durch eine ACK-Botschaft, bevor der Sender den nächsten Block senden darf. Die Anzahl der aufeinanderfolgenden Blöcke ist beliebig.

Eine einzelne Datenbotschaft kann dabei maximal 7 Datenbyte enthalten (Abb. 4.14). Das erste Byte, das sogenannte *Transport Control Byte TPCI* des Senders, dient der Steuerung der Übertragung und hat den in Abb. 4.15 dargestellten Aufbau. Mit *Acknowledge Request/AR = 0* signalisiert der Sender, dass er als nächstes eine ACK-Botschaft als Bestätigung erwartet, mit *End of Message EOM = 1*, dass es sich um die letzte Botschaft eines Blockes handelt. Die Sequenznummer SN ist eine fortlaufende Zahl, mit der aufeinander folgende Datenbotschaften durchnummieriert werden. Da nur 4 bit zur Verfügung stehen, wird die Sequenznummer mod 16 übertragen. Beide Kommunikationspartner besitzen einen unabhängigen SN-Zähler, der jeweils mit 0 beginnt und kontinuierlich bei jeder gesendeten Botschaft inkrementiert wird.

Bit	7	6	5	4	3	2	1	0
	1	0	RS	1				SN

Abb. 4.16 Aufbau des Transport Control Bytes TPCI des Empfängers

Die ACK-Botschaft des Empfängers besteht nur aus dem TPCI-Byte (Abb. 4.16). SN enthält die Sequenznummer der zu quittierenden Datenbotschaft plus 1. Mit dem *Receive Status Bit RS = 1 (Receiver Ready)* bestätigt der Empfänger, dass er für den Empfang des nächsten Blocks bereit ist. Mit RS = 0 (*Receiver Not Ready*) wird die Übertragung weiterer *Frames* verhindert. Innerhalb der Zeit T1 muss der Empfänger dann eine weitere ACK-Botschaft senden. Ist dabei wiederum RS = 0 gesetzt, so wird die Überwachungszeit T1 neu gestartet. Wird keine weitere ACK-Botschaft gesendet, wiederholt der Sender der Datenbotschaft den zuletzt gesendeten Block. Nach einer bestimmten Anzahl solcher Versuche bricht der Sender die Übertragung ab und schließt den Kanal.

Falls der Empfänger eine Datenübertragung vorzeitig beenden will, sendet er eine *BREAK-Botschaft* (Abb. 4.13). Der Sender antwortet darauf mit einer Datenbotschaft mit EOM = 1. Die seit dem letzten ACK übertragenen Daten werden verworfen, der Kanal bleibt jedoch geöffnet. Der Sender wartet eine bestimmte Zeit und startet dann die Übertragung erneut. Nach einer bestimmten Anzahl von erfolglosen Versuchen bricht er die Übertragung ab und schließt den Kanal.

Für den Fall, dass auf der Ebene des Transportprotokolls mehrere CAN-Botschaften gleichzeitig zur Übertragung anstehen, soll die Protokollsoftware die Botschaften in der in Tab. 4.3 dargestellten Reihenfolge senden.

4.4 Transportprotokoll TP 1.6 für CAN

Das Transportprotokoll 1.6 (TP 1.6) ist der Vorläufer des im vorigen Kapitel beschriebenen Protokolls TP 2.0 und verwendet weitgehend dieselben Konzepte und Botschaften. Auch hier werden exklusive Verbindungen (Kanäle) zwischen zwei CAN-Teilnehmern dynamisch eingerichtet und nach Beendigung des Datenaustausches wieder aufgelöst. Die

Tab. 4.3 Botschaftsprioritäten

Höchste Priorität 1	Connection Setup Response Connection Test
	BREAK-Botschaft
	ACK-Botschaft
Niedrigste Priorität 4	Datenbotschaft Connection Setup Request Disconnect

Kanaleröffnung erfolgt mit fest konfigurierten CAN-Identifiern, den *Eröffnungs-IDs*. Die CAN-Identifier für die Datenübertragung, die *Kanal-IDs* werden beim Verbindungsauftbau ausgehandelt. Analog zu TP 2.0 betreibt auch TP 1.6 ein aufwendiges Verfahren zur Flusskontrolle. Dabei kann der Sender für jedes vollständig gesendete Telegramm oder einen Block von mehreren Telegrammen eine Quittung (Handshake) vom Empfänger anfordern. Nach jedem vollständig übertragenen Datenblock wechselt die Senderichtung. Das bedeutet, die Gegenstelle (ehemals der Empfänger) wird jetzt zum Sender. Dieser kann nun selbst Daten senden oder das Senderecht wieder zurückgeben. Damit können bidirektionale, gleichberechtigte Kanäle zwischen zwei Busteilnehmern aufgebaut werden.

Die wesentlichen Unterschiede zu TP 2.0 sind:

- keine Broadcast-Botschaften (TP 2.0: Opcode 23 h und 24 h),
- keine Botschaft zum Verbindungstest (TP 2.0: Opcode A3 h),
- keine Botschaft zur Unterbrechung der Datenübertragung (TP 2.0: Opcode A4 h).

4.4.1 Botschaftsaufbau

Wie bei TP 2.0 besitzt auch hier jedes Steuergerät eine fahrzeugweit eindeutige logische Adresse, die einem festen Anfrage- bzw. Antwortkanal zugeordnet ist. Der zugehörige CAN-Identifier wird wiederum als *Eröffnungs-ID* bezeichnet. Über diesen Kanal können Botschaften zur Anforderung eines dynamischen Kanals ausgetauscht werden.

Darüber hinaus besitzt jeder Busteilnehmer ein bis vier feste Adressen, die für die Kommunikation über einen dynamischen Kanal verwendet werden können. Für den Antriebsstrang sind die CAN-Identifier ab 740 h reserviert, für den Komfort-Bus ab 300 h und für den Infotainment-Bus ab 4E0 h.

Für den *Eröffnungs-ID* gilt:

$$\text{Eröffnungs-ID} = \text{Basis-ID} + \text{Logische Adresse des Gerätes}$$

Eindeutige Geräteadresse bzw. Broadcast-Adresse für alle Steuergeräte bzw. den Diagnosetester am Antriebsstrang-Bus, am Infotainment-Bus bzw. am Komfort-Bus

4.4.2 Dynamischer Kanalaufbau

Für die eigentliche Datenübertragung zwischen zwei Geräten muss, wie bei TP 2.0 im Abschn. 4.3.3 beschrieben, zunächst eine logische Verbindung, ein sogenannter Kanal, aufgebaut werden. Dabei werden die für die spätere Übertragung verwendeten CAN-Identifier ausgehandelt. Der dynamische Kanalaufbau erfolgt mit der *Eröffnungs-ID* als CAN-Identifier. Der Botschaftsaufbau bei TP 1.6 unterscheidet sich vom Aufbau bei TP 2.0 (Abb. 4.17).

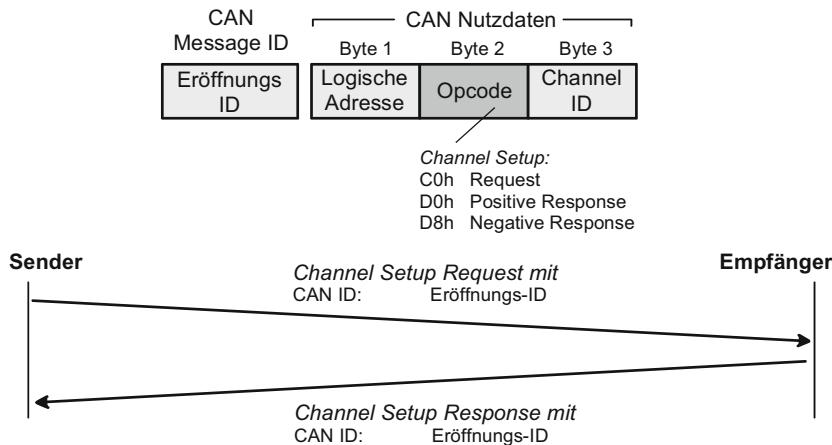


Abb. 4.17 Botschaften für den dynamischen Kanalaufbau bei TP 1.6

Das Steuergerät, das den Kanal aufbauen will, sendet in der *Channel Setup Request* Botschaft die zugehörige *Kanal-ID*, d. h. den CAN-Identifier, mit dem es zukünftig Datenbotschaften empfangen will. Das andere Steuergerät antwortet mit der *Kanal-ID*, mit der es seinerseits Datenbotschaften empfangen will.

4.4.3 Datenübertragung und Datenrichtungswechsel

Die Datenübertragung erfolgt mit den bereits im Abschn. 4.3.4 beschriebenen Datenbotschaften, die vom Empfänger jeweils durch ACK-Botschaften bestätigt werden, mit den jeweiligen *Kanal-IDs* als CAN-Identifier.

TP 1.6 fordert dabei, dass nach jedem abgeschlossenen Datenblock die Datenflussrichtung wechselt. Das Steuergerät, das die Verbindung aufgebaut hat, beginnt mit dem Senden eines Datenblocks. Sobald der Datenblock vollständig gesendet und in der letzten Botschaft des Datenblocks $EOM = 1$ gesetzt wurde, wird der bisherige Sender zum Empfänger und umgekehrt. Der neue Sender kann nun seinerseits einen Datenblock senden, in dessen letzter Botschaft wieder $EOM = 1$ gesetzt wird. Oder er gibt durch eine Datenbotschaft mit 0 Datenbytes und gesetztem $EOM = 1$ das Senderecht sofort wieder ab. Bei jedem Richtungswechsel wird SN wieder auf 0 gesetzt.

TP 1.6 unterscheidet darüber hinaus zwischen einem *schnellen* und einem *langsamem* Richtungswechsel für die Acknowledge-Botschaft. Beim *langsamem* Datenrichtungswechsel wird zunächst ein ACK gesendet, bevor der *neue* Sender seinerseits mit der Datenübertragung beginnt. Beim *schnellen* Datenrichtungswechsel wird kein ACK angefordert und der *neue* Sender beginnt sofort mit der Datenübertragung (Abb. 4.18).

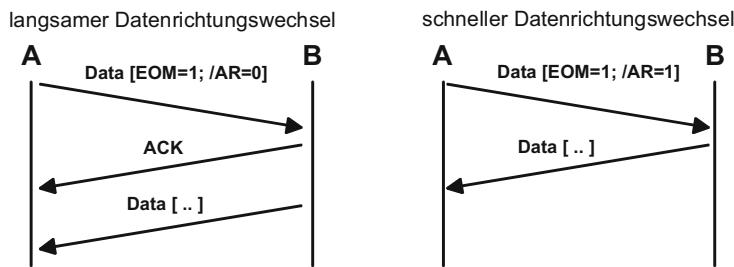


Abb. 4.18 Langsamer und schneller Datenrichtungswechsel bei TP 1.6

4.5 Transportprotokoll SAE J1939/21 für CAN

Das von der amerikanischen Society of Automotive Engineers SAE spezifizierte, auf CAN basierende Protokoll SAE J1939 wird hauptsächlich im Nutzfahrzeuggbereich verwendet. Dabei werden in mehreren Dokumenten die verschiedenen Ebenen des ISO/OSI-Referenzmodells beschrieben (Tab. 4.4). Bei der Definition wurde versucht, Strukturen des früher in amerikanischen Nutzfahrzeugen stark verbreiteten Busprotokolls nach SAE J1708 (zeichenorientiertes Busprotokoll mit 9600 bit/s und einer physikalischen Schnittstelle ähnlich RS485) mit der zugehörigen Applikationsschicht nach SAE J1587 auf CAN zu übertragen.

Tab. 4.4 Protokollstapel nach SAE J1939 (vereinfacht)

Dokument	Inhalt	ISO/OSI Referenzmodell	
SAE J1939/73 SAE J1939/71	Beschreibung der Dateninhalte für Off- und On-Board-Kommunikation	Application	7
		Presentation	6
		Session	5
SAE J1939/21	Transportprotokoll	Transport	4
SAE J1939/31	Spezifikation einer Bridge	Network	3
SAE J1939/21	Übertragung auf Basis von CAN 2.0B	Data Link	2
SAE J1939/14 SAE J1939/11, /15	Bitrate, Busankopplung, Verkabelung und Steckverbinder	Physical	1

Grundlage der Datenübertragung nach SAE J1939 bildet die Spezifikation CAN 2.0B, auf der auch ISO 11898 basiert. Dabei werden ausschließlich 29-Bit CAN-Identifier verwendet. Die elektrischen und mechanischen Eigenschaften der Busankopplung mit einer festen Bitrate von 250 kBit/s bzw. 500 kBit/s werden in SAE J1939/11, J1939/14 und J1939/15 definiert. Data Link Layer (Schicht 2) und Transport Layer (Schicht 4) werden leider nicht eindeutig unterschieden, sondern im Dokument SAE J1939/21 zusammengefasst.

Auf Anwendungsebene existiert sowohl eine Spezifikation für die Datenübertragung im Fahrbetrieb (On-Board-Kommunikation nach SAE J1939/71) als auch für die Diagnose (Off-Board-Kommunikation nach SAE J1939/73 und OBD nach SAE J1939/3).

Neben den in Tab. 4.4 dargestellten Protokollschichten existieren noch ein Schichtenübergreifendes Netzwerkmanagement (SAE J1939/81) sowie verschiedene Normergänzungen für Spezialanwendungen wie Landmaschinen (SAE J1939/2) oder Industriegeneratoren (SAE J1939/75).

4.5.1 Übertragungsarten, Adressierung und CAN Message Identifier

Alle Steuergeräte bei J1939 erhalten eine eindeutige, 8 bit große logische Steuergeräteadresse, wobei der Wert FFh als Broadcast-Adresse reserviert ist. Die Steuergeräteadressen sind im Allgemeinen statisch definiert. Die Netzwerkmanagement-Spezifikation SAE J1939/81 schlägt aber auch ein Verfahren vor, um die Steuergeräteadresse dynamisch zu konfigurieren.

SAE J1939 verwendet 29 bit CAN-Identifier. Diese sind in unterschiedliche Felder aufgeteilt, mit denen die Informationen zur Adressierung und zum Botschaftsinhalt codiert werden (Abb. 4.19). Viele dieser Felder haben ihren Ursprung im Vorläuferprotokoll J1708/J1587. Über die obersten 3 bit des CAN-Identifiers wird die Priorität der Botschaft codiert. Die unteren 8 bit des CAN-Identifiers geben die logische Adresse des Senders (*Source Address*) an. Den größten Teil des CAN-Identifiers nimmt die so genannte *Parameter Group Number PGN* ein. Deren wichtigster Teil ist das 8 bit große *PDU Format PF* Feld, das zwischen einer verbindungsorientierten und einer nachrichtenorientierten Datenübertragung unterscheidet. Bei der *verbindungsorientierten* Übertragung (*PDU 1 Format*) ist das Ziel der Botschaft ein einzelnes Steuergerät, dessen 8 bit große Zieladresse (*Destination Address*) dann innerhalb des PGN-Felds übertragen wird. Bei der *nachrichtenorientierten* Übertragung (*PDU 2 Format*) dagegen wird ein 8 bit Wert (*Group Extension GE*) übertragen, der den Inhalt der Nutzdaten auf der Anwendungsebene kennzeichnet. Das reservierte Bit und das *Data Page Bit* sind für zukünftige Erweiterungen vorgesehen und derzeit für Anwendungen von SAE J1939 bei Straßenfahrzeugen mit 0 belegt.

Die Werte des *PDU Format Bytes PF* und damit der Inhalt der Nutzdaten einer Botschaft wird im Wesentlichen durch die Normen für die Applikationsschicht J1939/71 und J1939/73 festgelegt. Einige wenige Werte im Bereich E8 h ... EFh sind für spezielle Aufgaben wie die im folgenden Abschnitt beschriebene Transportschicht für die segmentierte

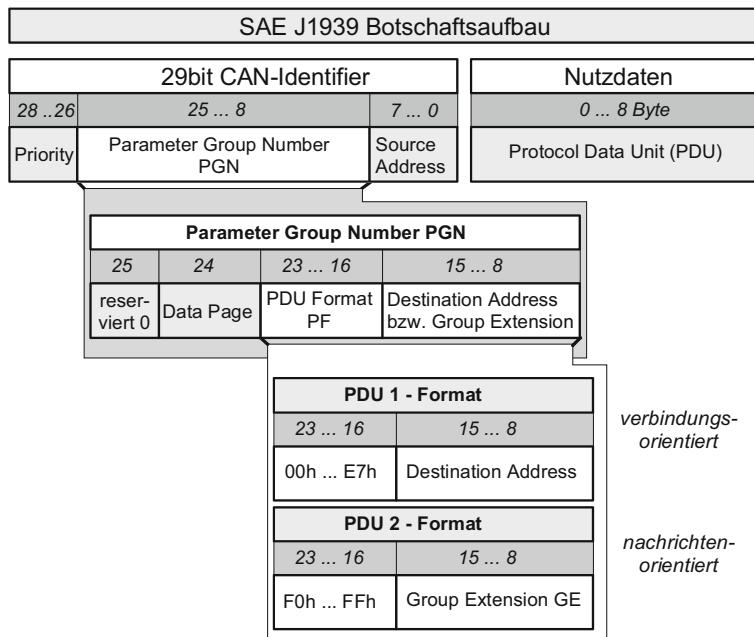


Abb. 4.19 Aufbau von SAE J1939-Botschaften und Struktur des CAN-Identifiers

Datenübertragung vorgesehen. Diese wird eingesetzt, wenn der zu übertragende Datenblock mehr als 8 Datenbytes umfasst und daher nicht mehr in eine einzelne CAN-Botschaft passt.

Als Beispiel für eine *nachrichtenorientierte* Botschaft soll die *Engine Temperature #1* Botschaft betrachtet werden, die periodisch jede Sekunde vom Motorsteuergerät gesendet wird und Temperaturinformationen enthält. Wie in SAE J1939/71 definiert, verwendet die Botschaft das *PDU 2 Format* PF = 254 = FEh und die *Group Extension* GE = 238 = EEh und enthält 8 Datenbytes mit den in Abb. 4.20 dargestellten Werten.

Die *Torque Speed Control #1* Botschaft dagegen, mit der das Motorsteuergerät bzw. die Motorbremse von übergeordneten Fahrzeugsystemen ferngesteuert werden, ist eine *verbindungsorientierte* Botschaft mit PF = 0. Sie wird alle 10 ms gezielt an das Motorsteuergerät bzw. alle 50 ms an die Motorbremse gesendet (Abb. 4.21) und enthält nur 4 Datenbytes. Im

Byte 1	Byte 2	Byte 3, 4	Byte 5, 6	Byte 7	Byte 8
Kühlwasser-temperatur	Kraftstoff-temperatur	Motoröl-temperatur	Turbolader-temperatur	Ladeluftkühlertemp.	LLK-Thermostatventil
- 40 ... +210 °C 1 K/bit	- 273 ... +1735 °C 0,03125 K/bit	- 40 ... +210 °C 1 K/bit	0 ... 100 % 0,4 %/bit		

Abb. 4.20 Aufbau der Engine Temperature #1 Botschaft (PF = FEh, GE = EEh)

Abb. 4.21 Aufbau der Torque Speed Control #1 Botschaft (PF = 00 h)

Byte 1	Byte 2, 3	Byte 4
Steuer- bits	Soll- drehzahl	Soll- drehmoment
0 ... 8032 min ⁻¹ 0,125 min ⁻¹ /bit	- 125 ... +125 %	1 %/bit

Abb. 4.22 Abfrage von Daten mit einer Request Botschaft (PF = EAh)

Byte 1, 2, 3
PGN der abzufragenden Daten, z.B. 00FEDAh

ersten Byte werden eine Reihe von Steuerbits zusammengefasst, mit denen festgelegt wird, ob der Motor drehzahl- bzw. drehmomentgeregelt betrieben, der Drehzahlregler für Leerlaufbetrieb, Fahrbetrieb oder Nebenabtriebe umgeschaltet werden soll. Außerdem wird die Priorität des externen Eingriffs gegenüber dem Leerlaufdrehzahlregler der Motorsteuerung bzw. dem Gaspedal festgelegt.

Botschaften, die mehr als 8 Datenbytes umfassen, erfordern die im folgenden Abschnitt beschriebene segmentierte Datenübertragung. Ein Beispiel ist die Abfrage der Software-Identifikations- und Versionsnummern. Dazu sendet das abfragende Gerät eine verbindungsorientierte *Request Botschaft* mit PF = 234 = EAh (Abb. 4.22) an das Steuergerät, dessen Identifikation ausgelesen werden soll. Die Botschaft enthält die 3 Byte lange PGN (PF und GE) der abzufragenden Informationen, im Beispiel die Werte PF = 254 = FEh, GE = 218 = DAh für die *Software Identification*. Die Antwort des Steuergerätes sind nachrichtenorientiert übertragene, segmentierte Botschaften mit einem Textstring variabler Länge im Datenteil (Abb. 4.23a, Details im folgenden Abschnitt).

Falls das Steuergerät die angeforderten Daten nicht liefern kann, antwortet es mit einer ACK-Botschaft (PF = 232 = E8 h), die im ersten Datenbyte den Wert 1 (*Not Acknowledge*)

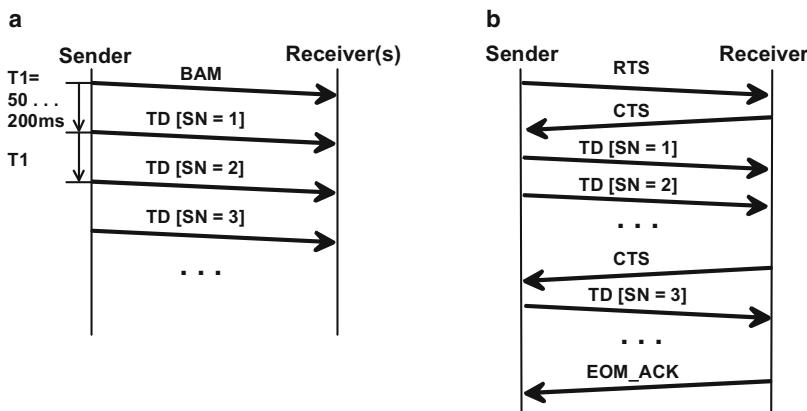


Abb. 4.23 Alternative Abläufe bei der segmentierten Datenübertragung, **a** nachrichtenorientierte Übertragung, **b** verbindungsorientierte Übertragung

	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
Transfer Data TD	Sequence Number					Daten		
<hr/>								
Request To Send RTS	10h	Anzahl Datenbytes	Anzahl Botschaften	Max. Anz. Resp. Bots.	Parameter Group Number			
Clear To Send RTS	11h	Anzahl Botschaften	Nächste Seq. Nr.	reserviert (FFh)	reserviert (FFh)	Parameter Group Number		
End Of Message EOM ACK	13h	Anzahl der empfangenen Datenbytes	Anz. empf. Botschaft.	reserviert (FFh)	Parameter Group Number			
Broadcast Announce Message BAM	20h	Anzahl Datenbytes	Anzahl Botschaften	reserviert (FFh)	Parameter Group Number			
Connection Abort CA	FFh	Grund für Abbruch		reserviert (FFh)	Parameter Group Number			

Abb. 4.24 Nutzdatenbereich der Botschaften für die segmentierte Übertragung

enthält. In einigen Fällen ist eine positive Bestätigung (*Acknowledge*) mit 0 im ersten Datenbyte erforderlich. Die letzten 3 Byte einer ACK-Botschaft enthalten die PGN, auf die sich die ACK-Botschaft bezieht.

Für herstellerdefinierte Botschaften sind PF = 239 = EFh und PF = 255 = FFh vorgesehen.

4.5.2 Segmentierte Datenübertragung (Multi Packet)

Wenn die segmentierte Datenübertragung an mehrere Empfänger gerichtet ist und daher *nachrichtenorientiert* erfolgt, kündigt der Sender den Empfängern die segmentierte Datenübertragung durch eine *Broadcast Announce Message BAM* an (Abb. 4.23a). Nach einer Pause von jeweils 50 ... 200 ms sendet er dann in den *Transfer Data TD* Botschaften die Nutzdaten. Dabei wird im ersten Byte eine bei 1 beginnende, mit jeder Datenbotschaft hochgezählte Sequenznummer SN übertragen, die übrigen 7 Datenbytes der CAN-Botschaft enthalten Nutzdaten. Die letzte Datenbotschaft wird gegebenenfalls mit FFh-Bytes aufgefüllt, so dass die CAN-Botschaften stets die volle Länge von 8 Datenbytes haben. Mit der BAM wurde die Gesamtzahl der Datenbytes sowie die Anzahl der zu erwartenden Botschaften übertragen (Abb. 4.24). Somit kann jeder Teilnehmer das Ende des Datenblocks erkennen, ohne dass dies besonders gekennzeichnet wird. Da die Sequenznummer nicht überlaufen darf, können auf diese Weise Datenblöcke mit maximal $255 \cdot 7 = 1785$ Byte übertragen werden. Wegen der großen Pausen von min. 50 ms zwischen den einzelnen CAN-Botschaften liegt die Nutzdatenrate bei dieser Art der Übertragung unter 160 B/s.

Erfolgt die segmentierte Datenübertragung dagegen *verbindungsorientiert*, d. h. mit einem einzigen Empfänger, so ist ein ähnlicher Handshake-Mechanismus vorgesehen wie

bei ISO 15765-2 (Abb. 4.23b). Der Sender sendet eine *Request To Send RTS* Botschaft an den Empfänger, in der er die Länge des Datenblocks (Gesamtzahl der Bytes und Anzahl der Segmente) ankündigt (Abb. 4.24). Der Empfänger antwortet mit einer *Clear To Send CTS* Botschaft, in der er mitteilt, wie viele CAN-Botschaften er ohne Pause hintereinander empfangen kann und welche Sequenznummer er als nächstes erwartet. Nachdem der Sender diese Anzahl von CAN-Botschaften gesendet hat, wartet er auf die nächste CTS Botschaft vom Empfänger, bevor er weitere Datenbotschaften sendet. Die letzte Datenbotschaft wird wieder mit FFh-Bytes aufgefüllt. Anschließend bestätigt der Empfänger mit einer *End Of Message Acknowledge EOM ACK* Botschaft den Empfang des gesamten Datenblocks. Will der Empfänger den Verbindungswunsch nicht annehmen oder die Übertragung vorzeitig abbrechen, so sendet er statt CTS eine *Connection Abort CA* Botschaft, die auch den Grund des aufgetretenen Problems enthält. Bei der verbindungsorientierten Datenübertragung dürfen die Datenbotschaften ohne Mindestabstand gesendet werden, so dass die volle CAN-Bandbreite ausgenutzt werden kann. Für die Maximalabstände existieren wiederum Zeitschränken im Bereich zwischen 750 ms und 1250 ms.

Für die *Transfer Data TD* Botschaft ist das PDU Format Byte PF = 235 = EBh reserviert. Die als *Connection Management* bezeichneten RTS, CTS, EOM, BAM und CA Botschaften verwenden PF = 236 = ECh. Zwischen ihnen wird mit unterschiedlichen Werten des ersten Nutzdatenbytes unterschieden. Zusätzlich wird bei diesen Botschaften in den letzten 3 Datenbytes die Parameter Group Number PGN des zugehörigen Nutzdatenblocks übertragen.

Empfänger müssen in der Lage sein, gleichzeitig mindestens eine verbindungsorientierte und eine nachrichtenorientierte Datenübertragung zu verarbeiten. Sender können gleichzeitig eine nachrichtenorientierte und gegebenenfalls mehrere verbindungsorientierte Übertragungen zu unterschiedlichen Empfängern handhaben.

4.6 Transportprotokoll DoIP nach ISO 13400

Mit der zunehmenden Nutzung von Fahrerassistenz- und Infotainmentsystemen steigt der Softwareumfang in Fahrzeugen und damit die Download-Zeit bei der End-of-Line-Programmierung (Flash Programmierung) in der Fertigung und in der Werkstatt immer weiter. Verwendet man dazu die übliche CAN Diagnoseschnittstelle mit einer maximalen Datenrate im Bereich um 30 KByte/s, so dauert die Übertragung von 100 MByte, wie sie bereits bei einem geringfügigen Update der Kartensoftware des Navigationssystems notwendig würde, knapp eine Stunde. Aus diesem Grund verwenden einige Oberklassefahrzeuge neuerdings eine zusätzliche Ethernet-Schnittstelle und führen die Übertragung mit dem Standard-Internetprotokoll TCP/IP durch (Abb. 4.25).

Die 10/100 Mbit/s Ethernet-Verbindung könnte zunächst freie Anschlüsse des OBD-Diagnosesteckers nutzen (Abb. 4.26). Mittelfristig soll eine neue, besser für die hohen Bitraten geeignete Steckverbindung eingeführt werden. Zusätzlich zu Ethernet ist ein Aktivierungssignal vorgesehen, mit dem die Verbindung freigeschaltet wird.

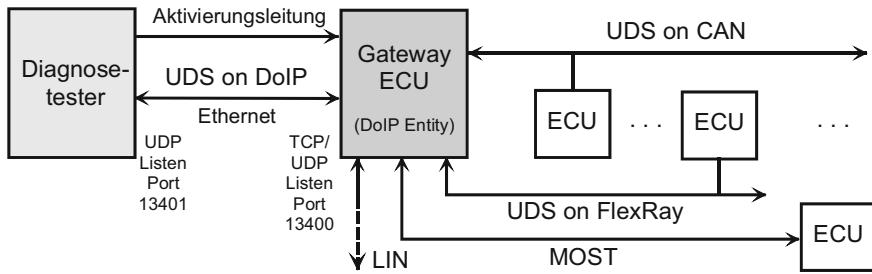
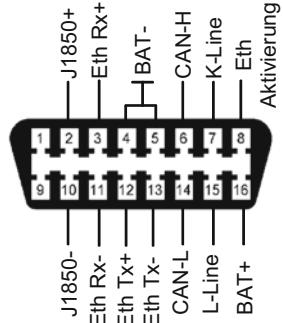


Abb. 4.25 Ethernet – TCP/IP – Schnittstelle zum Fahrzeug

Abb. 4.26 Erweiterung des OBD-Diagnosesteckers für Ethernet



Um existierende Diagnoseprotokolle wie UDS (siehe Kap. 5) sowohl auf der Steuergeräteseite als auch im Diagnosetester weiter nutzen zu können und die Anfragen des Diagnosestesters vom Gateway-Steuengerät direkt zu den internen CAN, FlexRay und LIN Busystemen durchleiten zu können, wurde mit *Diagnostics over IP* (DoIP) ein neues Transportprotokoll entwickelt. DoIP, das als ISO 13400 standardisiert wird, erlaubt es, gewöhnliche UDS-Botschaften in TCP/IP-Botschaften zu verpacken und diese über Ethernet oder WLAN zu übertragen (Abb. 4.27). Der Ethernet-Protokollstapel hat mit TCP und UDP, IP v4 bzw. v6 sowie dem Hilfsdienst DHCP den üblichen Umfang [2].

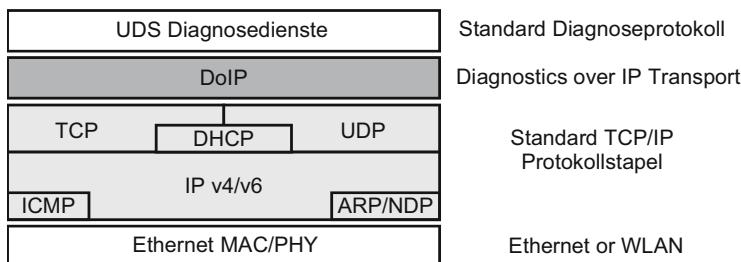
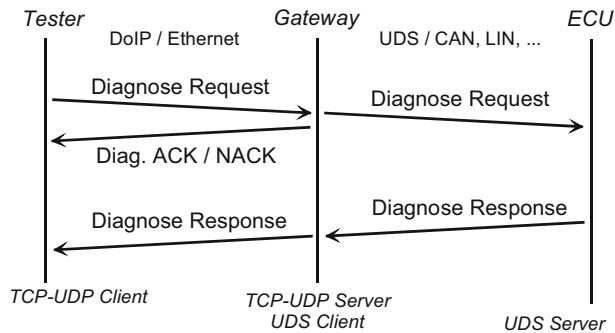


Abb. 4.27 Protokollstapel mit DoIP-Transportprotokoll

Abb. 4.28 DoIP Botschafts-
ablauf zwischen Tester,
Gateway und Steuergerät



Gegenüber dem Diagnosetester agiert das Gateway als TCP- bzw. UDP-Server mit vordefinierten IP-Ports. Weitere Ports können dynamisch vergeben werden. Die IP-Adressen werden über DHCP automatisch zugeordnet, wobei es feste Defaultwerte gibt, falls kein DHCP-Server vorhanden ist. Zusätzlich zu den diagnoseüblichen Request-Response-Botschaften zwischen Tester und Steuergerät tauschen das Gateway und der Tester auf der Ethernet-Seite Bestätigungsbotschaften (*Acknowledge ACK*) aus (Abb. 4.28). Damit muss das Gateway die erfolgreiche Weiterleitung einer Diagnosebotschaft zum internen Fahrzeugnetz bestätigen. Im Fehlerfall enthält diese Meldung (*Not Acknowledge NACK*) einen entsprechenden Fehlercode.

Die DoIP-Botschaften (Abb. 4.29 und Tab. 4.5) verwenden einen Header mit Steuerinformationen, mit dem verschiedene Nutzdatentypen (*Payload*) unterschieden werden. Enthält der Nutzdatenblock eine UDS Diagnosebotschaft (UDS Request bzw. Response) bzw. eine ACK/NACK-Bestätigungsmeldung, so werden die UDS *Source* bzw. *Target* Adressen von Steuergerät und Diagnosetester vorangestellt. Im Rest des Nutzdatenblocks folgen der SID *Service Identifier* und die notwendigen Parameter und Daten (vgl. Abschn. 5.1.3 und 5.2) bzw. bei Bestätigungsmeldungen ein 1 Byte langer Fehlercode. Diese Botschaften verwenden alle das TCP-Protokoll.

Bevor Diagnosebotschaften ausgetauscht werden, muss zunächst eine Art Verbindungsaufbau zwischen Diagnosetester und Fahrzeug-Gateway erfolgen. Sobald das Gateway nach dem Anschluss der Ethernet-Verbindung über DHCP mit einer gültigen IP-Adresse konfiguriert wurde, sendet es im Abstand von 500 ms über UDP eine *Vehicle Announce-*

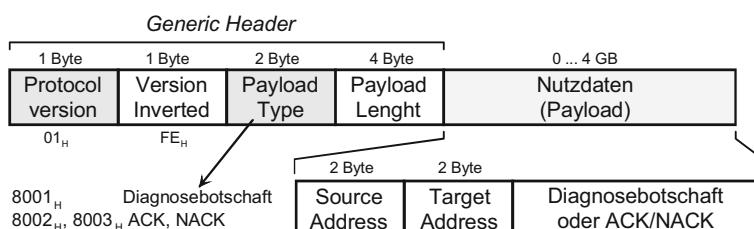


Abb. 4.29 DoIP Botschaftsformat (weitere Botschaften siehe Tab. 4.5)

Tab. 4.5 DoIP-Botschaftstypen

Bezeichnung	Payload Type	IP Protokoll	Anwendung
Generic DoIP Header NACK	0000 _H		Meldung des Gateways bei Fehlern im DoIP-Header, enthält Fehlercode
Vehicle Identification Request	0001 _H ... 0003 _H	UDP	Abfrage der Fahrzeugkennungen durch den Tester. Enthält keine Nutzdaten. Zusätzliche Untervarianten, falls der Tester die VIN und/oder die eindeutigen Gerätekennziffern kennt und gezielt ein Gateway ansprechen will.
Vehicle Announcement Message bzw. Vehicle Identification Response	0004 _H	UDP	Ankündigung des Fahrzeugs, dass ein DoIP- Gateway vorhanden ist. Enthält die Fahrgestellnummer VIN (17 Byte), die UDS-Diagnoseadresse des Gateways (2 Byte) und zwei je 6 Byte lange eindeutige Gerätekennziffern
Routing Activation Request/Response	0005 _H 0006 _H	TCP	Aktivieren der Weiterleitung von Diagnosebotschaften von/zum internen Fahrzeugnetz
Alive Check Request/Response	0007 _H 0008 _H	TCP	Aufrechterhalten der TCP/IP-Verbindung, falls über längere Zeit keine Diagnosebotschaften ausgetauscht werden
DoIP Entity Status Request/Response	4001 _H 4006 _H	UDP	Abfrage zum Typ des DoIP Gerätes (Gateway oder normales Steuergerät) und zur Anzahl der möglichen Verbindungen (TCP/IP-Sockets)
Diagnostic Power Mode Info Request/Response	4003 _H 4004 _H	UDP	Abfrage, ob die Fahrzeugnetzwerke aktiv sind (Zündung ein usw.)
Diagnostic Message Request	8001 _H	TCP	Austausch von Diagnosebotschaften, Einzelheiten siehe Text
Diag. ACK/NACK	8002 _H 8003 _H		

ment Message. Diese teilt dem Diagnosetester die Fahrgestellnummer (*Vehicle Identification Number VIN*), die UDS-Diagnoseadresse des Gateways sowie zwei Hardwarekennziffern mit, z. B. die MAC-Adresse. Durch die Hardwarekennziffern soll der Tester unterscheiden können, ob im Fahrzeug außer dem Gateway weitere Geräte direkt mit dem Ethernet-Zugang verbunden sind. Falls der Tester die *Vehicle Announcement* Informationen verpasst, kann er das Fahrzeug durch Senden einer *Vehicle Identification Request Message* jederzeit zur Wiederholung der Botschaften auffordern.

Im Anschluss an die Fahrzeugidentifikation fordert der Diagnosetester das Gateway durch einen *Routing Activation Request* auf, nachfolgende Diagnosebotschaften tatsächlich von und zum internen Fahrzeugnetz weiterzuleiten. Der *Routing Activation Request*

enthält dabei die UDS-Diagnoseadresse des Gateways. Dadurch kann in zukünftigen Fahrzeugen mit möglicherweise mehreren Gateways gezielt ein einzelnes Gateway ausgewählt werden. Über weitere Parameter dieser Botschaft soll zukünftig voraussichtlich auch eine Login/Authentifizierungsprozedur abgewickelt werden, um den Fahrzeugzugang freizuschalten. Während einer bestehenden Verbindung kann der Diagnosetester über verschiedene Botschaften, die hier nicht im Detail beschrieben werden sollen, Statusinformationen mit dem Gateway austauschen.

4.7 Transportprotokoll für CAN FD

Für die CAN-Weiterentwicklung CAN FD muss eine Erweiterung des bisherigen Transportprotokolls nach ISO 15765-2 (Abschn. 4.1) definiert werden. Da CAN FD bis zu 64 Byte in einer einzelnen CAN-Botschaft übertragen kann, ist das bisher nur 4 bit lange Datenlänge-Feld DL im Single Frame SF des Transportprotokolls nicht mehr ausreichend (Abb. 4.1). Weil das modifizierte Transportprotokoll aufwärtskompatibel bleiben soll, wird wohl ähnlich wie bei FlexRay (Tab. 4.1) ein modifizierter Single Frame mit einem mindestens 6 bit langen DL-Feld eingeführt.

Bei den übrigen Transportprotokoll-Botschaften besteht aus Sicht von CAN FD kein Änderungsbedarf. Allerdings kann darüber nachgedacht werden, auch eine Variante des First Frames FF einzuführen, falls angestrebt wird, die Blockgröße für segmentierte CAN-Übertragungen von bisher 4 KB in Richtung der beim FlexRay-Transportprotokoll möglichen 64 KB-Blöcke zu erweitern.

4.8 Normen und Standards zu Kapitel 4

ISO 15765 Road vehicles – Diagnostics on CAN, siehe Kap. 5
(ISO TP)

AUTOSAR siehe Kap. 8

ISO 10681 Road vehicles – Communication on FlexRay, siehe Kap. 5

VW TP VW CAN-Transportprotokoll TP 2.0, Version 1.1, 2002, internes Dokument
VW CAN-Transportprotokoll, Version 1.6.1, 2000, internes Dokument

SAE J1939	SAE J1939 Serial Control and Communications Heavy Duty Vehicle Network, 2012, www.sae.org SAE J1939/1 On-Highway Equipment Control and Communications Network, 2011, www.sae.org SAE J1939/2 Agricultural and Forestry Off-Road Machinery Control and Communications Network, 2013, www.sae.org SAE J1939/11 Physical Layer 250 kbit/s, Shielded Twisted Pair, 2012 SAE J1939/14 Physical Layer 500 kbit/s, 2011, www.sae.org SAE J1939/15 Reduced Physical Layer 250 kbit/s, Unshielded Twisted Pair, 2008 SAE J1939/13 Off-Board Diagnostic Connector, 2011, www.sae.org SAE J1939/21 Data Link Layer, 2010, www.sae.org SAE J1939/31 Network Layer, 2010, www.sae.org SAE J1939/81 Network Management, 2011, www.sae.org SAE J1939/71 Vehicle Application Layer, 2012, www.sae.org SAE J1939/73 Application Layer - Diagnostics, 2010, www.sae.org SAE J1939/3 On-Board Diagnostics Implementation Guide, 2008 SAE J1939/74 Application Layer – Configurable Messaging, 2010 SAE J1939/75 Application Layer – Generator Sets and Industrial, 2011
SAE J1587	SAE J1587 Electronic Data Interchange Between Microcomputer Systems in Heavy-Duty Vehicle Applications, 2013, www.sae.org
SAE J1708	SAE J1708 Serial Data Communications Between Microcomputer Systems in Heavy-Duty Vehicle Applications, 2010, www.sae.org
ISO 13400 DoIP	ISO 13400 Road vehicles – Diagnostic communication over Internet Protocol (DoIP), www.iso.org Part 1: General information and use case definition, 2011 Part 2: Transport protocol and network layer services, 2012 Part 3: Wired vehicle interface based on IEEE 802.3, 2011 Weitere Literaturangaben zu Internet Protokollen siehe Kap. 8, zu Ethernet siehe Kap. 3

Literatur

- [1] R. Schmidgall: Automotive Embedded Systems Software Reprogramming, 2012, PhD-Thesis, School of Engineering and Design, Brunel University London, <http://bura.brunel.ac.uk/handle/2438/7070>
- [2] R. Stevens: TCP/IP Illustrated. Addison-Wesley, 3 Bände, 2002

In den 1990er Jahren wurde die Notwendigkeit erkannt, nicht nur die Datenübertragung sondern auch die Applikationsschicht der Protokolle zu standardisieren, um den Aufwand und die Pflege der Diagnoseschnittstellen in den Steuergeräten und bei den Diagnosetestern zu begrenzen. Getrieben wurde diese Standardisierung zum einen von den gesetzlichen Vorschriften, die für die Überprüfung zumindest der abgasrelevanten Systemfunktionen eine relativ einheitliche Schnittstelle erzwangen (On-Board-Diagnose OBD). Zum anderen durch die zunehmende Kooperation der Fahrzeughersteller im Zuge der Globalisierung. In deren Rahmen werden Komponenten oder ganze Fahrzeuge teilweise gemeinsam von mehreren Herstellern entwickelt und/oder eingesetzt. Leider begann dieser Standardisierungsprozess relativ spät, so dass bereits viele proprietäre, inkompatible Einzellösungen existierten. Zudem verlief der Standardisierungsprozess langsamer als die technische Weiterentwicklung [1]. Man merkt den heutigen Standards daher ihre historische Entwicklung und das Bemühen an, ältere Lösungen zu integrieren. Es gibt diverse, sich überdeckende Normen. Viele Festlegungen werden mehrfach wiederholt, unterscheiden sich aber doch in Details oder lassen verschiedene Varianten zu, die in Wirklichkeit untereinander inkompatibel sind. Oft wird nur der kleinste gemeinsame Nenner festgelegt, so dass viele Punkte weiter hersteller- und implementierungsabhängig bleiben.

Zurückblickend ergibt sich etwa der folgende historische Verlauf bis heute (Abb. 5.1):

- Der erste herstellerübergreifende Standard für die Diagnoseschnittstelle war **ISO 14230**, das **K-Line-basierte KWP 2000 Protokoll**, wobei die Norm den gesamten Protokollstapel umfasst. Allerdings wurden im Wesentlichen nur das Kommunikationsmodell und die Übertragungsdienste (*Diagnostic Services*) definiert, mit denen Anfragen und Antworten zwischen Diagnosetester und Steuergeräten ausgetauscht werden. Die Bedeutung und Formatierung der Diagnosedaten dagegen wurde nicht weiter festgelegt. Die Norm schließt abgasrelevante Diagnosedienste (OBD) ausdrücklich aus, reserviert aber einen Bereich von Diagnosebotschaften dafür und verweist für OBD-Diagnosedienste auf die Norm ISO 15031-5. Für die unteren Protokollsichten dagegen werden im Zu-

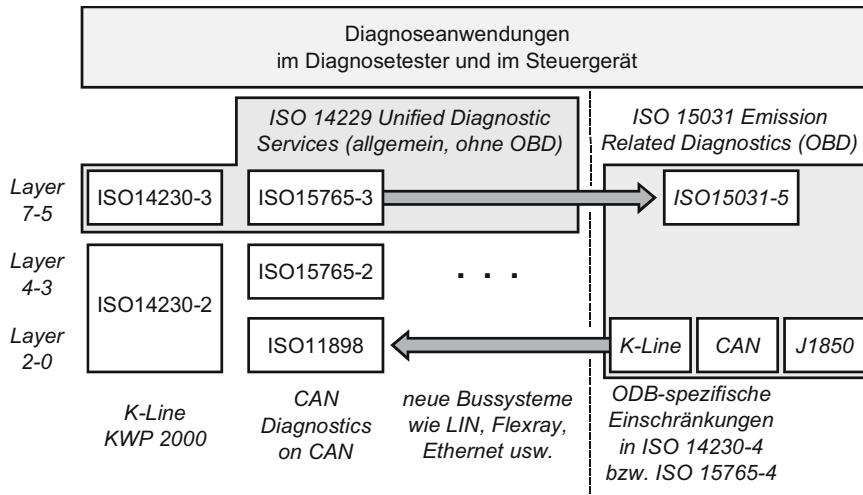


Abb. 5.1 Protokollfamilie für Diagnoseschnittstellen in europäischen Steuergeräten

sammenhang mit OBD in ISO 14230-4 verschiedene Protokollvarianten ausgeschlossen und verschiedene Punkte präzisiert.

- Mit der als **KWP 2000 on CAN** bekannt gewordenen Vornorm **ISO/DIS 15765-3** wurde der **KWP 2000**-Application Layer weitgehend unverändert auf CAN übertragen. Bezuglich des Inhaltes der Diagnoserdienste referenziert und kommentierte ISO/DIS 15765-3 im Wesentlichen lediglich ISO 14230-3. Inhaltliche Unterschiede im Application Layer gab es lediglich bei einigen wenigen Diensten, die zur Parametrierung der Datenübertragungsschichten in den Layern 4 und darunter dienten. Weil sich der Standardisierungsprozess lange hinzog, wurde der Normentwurf ISO/DIS 15765-3 von vielen Herstellern bereits in verschiedenen Zwischenstadien implementiert und dürfte das heute in der Praxis noch immer am häufigsten anzutreffende Diagnoseprotokoll sein.
- Bevor **ISO 15765** dann jedoch formal verabschiedet werden konnte, kam es zu einer Abkehr vom **KWP 2000 on CAN**-Konzept. Um neue Bussysteme wie LIN oder FlexRay leichter integrieren zu können, wurde versucht, in der Norm **ISO 14229** die Applicationsschicht unabhängig vom darunter liegenden Bussystem einheitlich zu beschreiben (**Unified Diagnostic Services UDS**). Im Grundaufbau der Botschaften sind die in ISO 14229 spezifizierten Dienste mit den KWP 2000-Diensten identisch, allerdings wurden die Dienste teilweise neu gruppiert, neue Identifier vergeben und der Parameteraufbau in vielen Details verändert, so dass UDS nur funktional, aber nicht in der konkreten Implementierung zu KWP 2000 aufwärts kompatibel ist. Außerdem wurden einige Dienste ergänzt, die bei KWP 2000 wegen der begrenzten Leistungsfähigkeit des K-Line-Bussystems nicht sinnvoll gewesen wären. UDS muss daher als eigenständiges, wenn auch zu KWP 2000 nahe verwandtes Protokoll betrachtet werden.

- ISO 15765-3 wurde in **UDS on CAN** umbenannt und beschreibt nun „nur“ noch die konkrete Implementierung der UDS Dienste für das Bussystem CAN. CAN-spezifisch ist im Wesentlichen die Transportschicht nach ISO 15765-2 (Layer 4), die erforderlich ist, weil die CAN-Botschaftslänge von maximal 8 Nutzdatenbytes für viele der im Application Layer definierten Dienste nicht ausreicht, sowie einige Diagnosedienste, mit denen direkt aus dem Application Layer heraus Parameter des Bussystems beeinflusst werden können. Für OBD-Anwendungen wurden im vierten Teil der Norm ISO 15765-4 einige Punkte der unteren Protokollsichten präzisiert. Bei Neuentwicklungen erfolgt heute in der Regel ein Übergang von *KWP 2000 on CAN* auf *UDS on CAN*, das nun durch das Normenbündel ISO 14229-1 und ISO 15765-1 bis -4 gemeinsam definiert wird. *UDS on CAN* ist allerdings ein relativ komplexer Standard, der einige redundante Dienste enthält. Zusätzlich enthält das Protokoll einige Möglichkeiten, z. B. die *Response on Event* Dienste, die sehr aufwendig in der Implementierung sind. Es ist daher nicht überraschend, wenn Hersteller jeweils lediglich eine Teilmenge des Standards implementieren.
- Für die Diagnose abgasrelevanter Komponenten haben die Gesetzgeber zunächst in USA und dann in Europa Vorschriften erlassen, die u. a. das Diagnoseprotokoll definieren. Während auf den Ebenen 0–4 dabei die ganze Sammlung gängiger Diagnose-Bussysteme alternativ zulässig war (Abb. 5.1) und erst seit 2008 (in USA) eine Vereinheitlichung mit CAN erfolgt, konnte man sich auf der Applikationsebene erfreulicherweise auf ein einheitliches Protokoll einigen, das in ISO 15031-5 und der inhaltsgleichen Norm SAE J1979 beschrieben wird. Dieses *OBD-Protokoll* erlaubt Behörden und Institutionen wie dem TÜV den Zugriff auf die wichtigsten Diagnosedaten (Fehlerspeicher, Messwerte, Ergebnis von Überwachungstests) für die unmittelbar abgasrelevanten Funktionen (Lambdaonde, Katalysator, Zündung, Einspritzung). Als kleinstes gemeinsamer Nenner wird dabei aber nur ein minimaler Funktionsumfang unterstützt, so dass Werkstätten praktisch immer zusätzliche Diagnoseinformationen über KWP 2000 oder andere Protokolle verwenden müssen und sich einige Funktionen, z. B. das Auslesen und Löschen des Fehlerspeichers, überlappen, d. h. in beiden Protokollen für denselben Zweck in unterschiedlicher Form existieren.
- Früher als in Europa wurde in USA neben der Diagnose der abgasrelevanten Komponenten auch die allgemeine Diagnose mit SAE J2190 standardisiert. Die spätere europäische KWP 2000 Standardisierung nahm diese Norm zum Vorbild, so dass die Mehrzahl der Dienste und ihrer Parameter gleich sind, auch wenn leider vielfach unterschiedliche Bezeichnungen verwendet werden (der KWP 2000 *Service Identifier* z. B. entspricht dem J2190 *Test Mode*). Aus diesem Grund wird auf SAE J2190 im Folgenden nicht mehr weiter eingegangen.

Da die Diagnoseprotokolle alle dieselben Grundkonzepte verwenden und da UDS ohnehin eine direkte Weiterentwicklung darstellt, beginnt die folgende Beschreibung bei KWP 2000, das immer noch vielfach im Einsatz ist. Anschließend werden dann die Unterschiede der Kommunikation nach UDS und OBD dargestellt.

5.1 Diagnoseprotokoll KWP 2000 (ISO 14230-3)

Das derzeit am weitesten verbreitete Diagnoseprotokoll in europäischen Fahrzeugen ist das zunächst mit K-Line und später mit CAN-Bussystemen realisierte Keyword 2000 Protokoll (KWP 2000), das in ISO 14230-3 für K-Line offiziell und in der Vornorm ISO/DIS 15765-3 inoffiziell definiert ist.

5.1.1 Überblick

Das Kommunikationsmodell der KWP 2000-Diagnose sieht vor, dass die gesamte Kommunikation vom Tester ausgeht (Abb. 5.2). Die Diagnose-Anwendung des Testers sendet eine Botschaft mit einer Diagnose-Anfrage (*Request*) über das Netz an das Steuergerät. Der *Application Layer* informiert die Steuergeräte-Anwendung (*Indication*). Die Antwort (*Response*) des Steuergerätes wird über das Netz an den Tester übertragen, dessen *Application Layer* die Antwort (*Confirm*) auf die ursprüngliche Anfrage an die Tester-Anwendung übergibt. In der Norm werden der Diagnosetester als *Client*, das Steuergerät als *Server* und die vordefinierten Anfragen als *Services* (Dienste) bezeichnet.

Die Dienste der Anwendungsschicht bestehen aus folgenden Teilen (Abb. 5.3):

- *Adressinformation AI*, bestehend aus *Source Address SA*, *Target Address TA* sowie gegebenenfalls *Remote Address RA*, jeweils 1 Byte,
- *Service Identifier SID*, kennzeichnet den ausgewählten Dienst (1 Byte),
- Parameter, Anzahl abhängig vom jeweiligen Dienst.

Die Adressinformationen werden im Header abgebildet, wie in Abb. 3.15 für K-Line und in Abb. 4.1 für CAN bereits dargestellt ist. Der *Service Identifier SID* wird im ersten Nutzdatenbyte, die Parameter in den folgenden Nutzdatenbytes übertragen. Bei Antworten

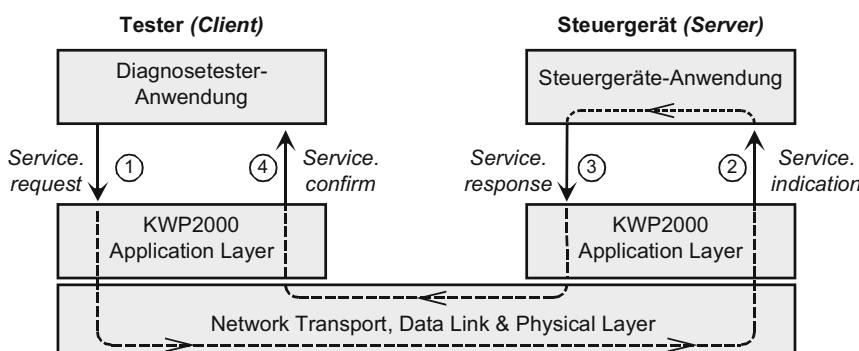
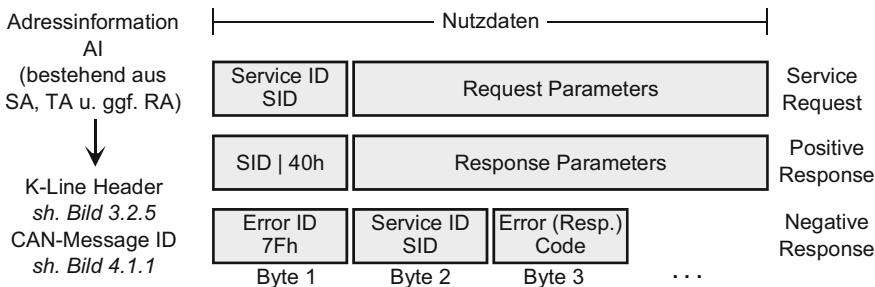


Abb. 5.2 Kommunikationsmodell der KWP 2000 – Anwendungsschicht

**Abb. 5.3** Aufbau KWP 2000 Application Layer Botschaften

wird zwischen einer positiven (*Positive Response*) und einer negativen Antwort (*Negative Response*) unterschieden. Tabelle 5.1 zeigt die in den Normen definierten *Service Identifier SID*.

Die Service Identifier für positive Antworten entsprechen den Service Identifiern der Anfrage, wobei das Bit 6 auf 1 gesetzt wird (dies entspricht einer logischen ODER Verknüpfung mit 40 h). Bei den negativen Antworten wird SID = 7Fh und anschließend der Service Identifier des fehlgeschlagenen Dienstes sowie ein Fehlercode (1 Byte) gesendet (Tab. 5.2). Im Transport bzw. Data Link Layer werden diese Botschaften dann mit den entsprechenden Header- und Trailer-Informationen versehen.

Die KWP 2000 Diagnosedienste lassen sich in Funktionsgruppen unterteilen, wie in Tab. 5.3 dargestellt.

Die Zeitbedingungen für die K-Line-Kommunikation wurden bereits in Tab. 3.7 angeben. Beim CAN-Bus muss das Steuergerät eine Testeanfrage innerhalb von 50 ms beantworten.

Tab. 5.1 Überblick Service Identifier (nicht aufgeführte Werte sind reserviert)

	Request	Positive Response	Negative Response	Definiert durch
OBD kompatible Dienste	00 ... 0Fh	40 ... 4Fh	7 F 00 ... 7 F 0Fh	SAE J1979 (ISO 15031-5)
Allgemeine Dienste (für K-Line und CAN)	10 ... 3Eh	50 ... 7Eh	7 F 10 ... 7 F 3Eh	ISO 14230-3 (in ISO/DIS 15765-3 wiederholt)
Escape Code (ESC)	80 h	C0 h	7 F 80 h	dito
K-Line-spezifische Dienste	81 ... 83 h	C1 ... C3 h	7 F 81 ... 7 F 82 h	ISO 14230-2
CAN-spezifische Dienste	84 ... 85 h	C4 ... C5 h	7 F 84 ... 7 F 85 h	ISO/DIS 15765-3
Herstellerspezifische Dienste	A0 ... BEh	E0 ... FEh	7 F A0 ... 7 F BEh	Fahrzeug- bzw. Systemhersteller

Tab. 5.2 Typische Response Error Codes

10 h	General reject (allgemeiner Fehler)
11 h	Service not supported
12 h	Subfunction not supported
21 h	Busy, repeat request
78 h	Response pending
33 h	Security access denied
35 h	Invalid key
...	...

worten (P2 Timeout in Tab. 3.7). Der Tester darf anschließend sofort die nächste Anfrage senden. Bei den Diagnosediensten, bei denen er keine Antwort erhält oder bei denen mehrere Steuergeräte antworten können, muss zwischen den Anfragen ein Mindestabstand von ebenfalls 50 ms (P3 Timeout) liegen.

5.1.2 Diagnosesitzungen (Diagnostic Management)

Unter dem Begriff *Diagnosesitzung* (*Diagnostic Session*, andere Bezeichnung: *Diagnostic Mode*) versteht man einen Betriebszustand des Steuergerätes, in dem ein bestimmter Satz von Diagnosediensten unterstützt wird. Im normalen Betrieb befindet sich das Steuergerät in der *Default Diagnostic Session*, in der üblicherweise aus Sicherheitsgründen nur ein sehr kleiner Teil der Diagnosedienste unterstützt wird. Beim K-Line-Bus muss dem Start einer Diagnosesitzung ein Verbindungsaufbau vorausgehen, wie in Abschn. 2.2.3 dargestellt. Die verschiedenen unterstützten Diagnosesitzungen werden durch Nummern identifiziert (Tab. 5.4).

Ausgehend von der Default Diagnostic Session kann der Diagnosetester das Steuergerät durch eine *Start Diagnostic Session* Botschaft (Tab. 5.5) auffordern, eine spezielle Diagnosesitzung zu eröffnen.

Tab. 5.3 Überblick über die KWP 2000-Diagnosedienste

Diagnostic Management	Verwaltung der Diagnosesitzung
Network Layer Protocol Control (Communication Management)	Steuerung des Data Link Layers, z. B. Modifikation von Timeout-Parametern
Data Transmission	Einzelne Steuergeräte-Daten lesen und schreiben
Stored Data Transmission	Fehlerspeicher lesen und löschen
Input/Output Control	Ansteuerung von Steuergeräte-Ein-/Ausgängen
Remote Activation of Routines	Starten von Programmen im Steuergerät
Upload/Download	Programm- und Datenblöcke auslesen und speichern (<i>Flashen</i>)

Tab. 5.4 Diagnostic Session Nummern nach ISO/DIS 15765-3

Nummer	Typ	Bemerkung
81 h	Default Diagnostic Session	Grundzustand
85 h	Programming Session	Programmieren des Programm- und Datenspeichers (<i>Flashen</i>)
86 h	Development Session	Sondermodus für die Steuergeräte- und Systementwicklung
87 h	Adjustment Session	Applikation von Steuergeräte-Parametern
89 h – FEh	Herstellerspezifische Session	

Der Fahrzeughersteller legt fest, unter welchen Bedingungen eine bestimmte Diagnosesitzung eröffnet werden kann und welche Diagnosedienste in dieser Sitzung unterstützt werden. Übliche Bedingungen sind:

- Fahrzeug und Motor müssen sich in einem bestimmten Betriebszustand befinden, z. B. muss beim *Flashen* das Fahrzeug stehen und der Motor abgestellt sein. Beim Test von bestimmten Aktuatoren (*Stellertest*) muss das Fahrzeug stehen und der Motor im Leerlauf arbeiten usw.
- Der Diagnosetester muss sich beim Steuergerät mit Hilfe eines Schlüsselaustausches (*Seed and Key*) anmelden (*authentifizieren*), um Zugriff auf bestimmte Diagnosedienste zu erhalten (*Unlock ECU*). Dazu sendet der Tester eine *Security Access – Request Seed* Botschaft. Das Steuergerät antwortet mit einem Initialisierungswert (*Seed*), oft einer Zufallszahl, aus dem der Tester einen Schlüsselwert (*Key*) berechnet und mit einer *Security Access – Send Key* Botschaft an das Steuergerät zurückschickt. Falls der vom Tester gesendete *Key*-Wert mit dem vom Steuergerät erwarteten Wert übereinstimmt, sendet

Tab. 5.5 KWP 2000-Dienste für die Verwaltung der Diagnosesitzungen

Service	SID	Parameter/Bemerkung
Start Diagnostic Session	10 h	Session-Nummer (1 Byte)
Stop Diagnostic Session	20 h	Nur bei K-Line. Bei CAN wird die aktive Sitzung gestoppt, indem mit SID 10 h auf die Defaultsitzung 81 h umgeschaltet wird.
Security Access	27 h	01 h: Request Seed 02 h: Send Key weitere herstellerspezifische Werte und Parameter möglich
Tester Present	3Eh	Botschaft zum Aufrechterhalten einer Diagnosesitzung (Vermeiden eines Timeout, auch als <i>Keep Alive</i> bezeichnet)
ECU Reset	11 h	Steuergerät zurücksetzen
Read ECU Identification	1Ah	Parameter und Steuergeräteantwort herstellerspezifisch

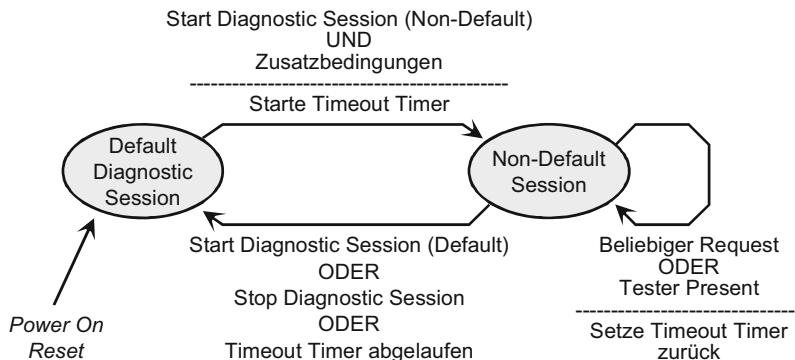


Abb. 5.4 KWP 2000 Diagnose-Sitzungen

das Steuergerät eine positive Antwort und schaltet auf die neue Diagnosesitzung um. Die Länge der *Seed*- und *Key*-Werte sowie die Algorithmen, mit denen sie berechnet werden, sind in den KWP 2000-Normen selbst nicht definiert sondern dem Hersteller überlassen. Üblich sind Sitzungen mit unterschiedlichen Schlüsseln für den Steuergerätehersteller, den Fahrzeugherstellern sowie verschiedenen Arten von Vertrags- und anderen Werkstätten usw. Die Unterscheidung erfolgt über den herstellerspezifischen Parameter der *Security Access* Botschaften.

Solange sich das Steuergerät in einer speziellen Diagnosesitzung befindet, wird ein Timeout-Mechanismus aktiviert (Abb. 5.4). Wenn das Steuergerät nicht spätestens alle 5 s eine Diagnose-Botschaft vom Tester erhält, stoppt es diese Diagnosesitzung und kehrt zur Default Session zurück. Für den Fall, dass der Tester keine *echte* Diagnosebotschaft zu senden hat, sendet er (aus Sicherheitsgründen bereits spätestens nach 4 s) eine *Tester Present* Botschaft, um die Verbindung aufrecht zu erhalten.

Normalerweise wird eine Diagnosesitzung vom Diagnosetester durch eine *Stop Diagnostic Session* Botschaft beendet. Darauf hin kehrt das Steuergerät zur *Default Diagnostic Session* zurück. Alternativ kann der Tester auch direkt eine andere Diagnosesitzung mit Hilfe einer weiteren *Start Diagnostic Session* Botschaft anfordern. Dadurch wird die gerade aktive Sitzung ebenfalls beendet, da immer nur genau eine Diagnosesitzung aktiv sein kann. Bei CAN sollte laut Norm die *Stop Diagnostic Session* Botschaft nicht verwendet werden, sondern die spezielle Diagnosesitzung durch eine weitere *Start Diagnostic Session* Botschaft mit der Sitzungsnummer 81 h (*Default Diagnostic Session*) beendet werden.

Durch eine *ECU Reset* Botschaft kann der Diagnosetester das Steuergerät auch zu einem vollständigen Neustart auffordern, d. h. das Steuergerät verhält sich dann wie beim Einschalten der Spannungsversorgung.

Mit Hilfe der *Read ECU Identification* Botschaft kann der Diagnosetester Kenndaten des Steuergerätes abfragen. Die Antwort des Steuergerätes ist herstellerspezifisch. Üblich sind

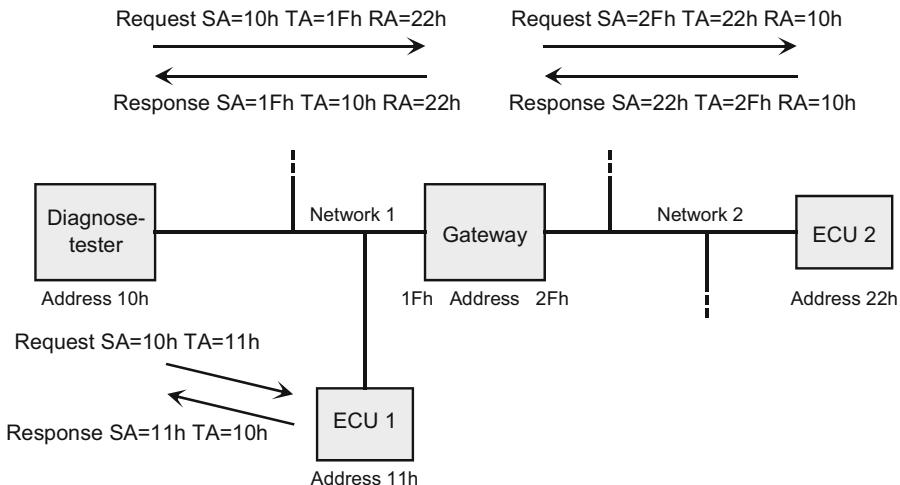


Abb. 5.5 Geräteadressierung nach KWP 2000

Herstellername, Typ, Hardware- und Softwareversion, Seriennummer, Fertigungsdatum usw.

5.1.3 Adressierung der Steuergeräte nach KWP 2000 und UDS

KWP 2000 Diagnoseanwendungen verwenden zur Kennzeichnung von Steuergeräten eine 1 Byte große Kennziffer, die so genannte Geräteadresse (Abb. 5.5). Beim Aufruf der Diagnosedienste durch den Tester (*Request*) sowie bei den Antworten durch das Steuergerät (*Response*) werden Sender und Empfänger der zugehörigen Daten durch diese Adressen identifiziert (*Source Address SA* ... Adresse des Senders, *Target Address TA* ... Adresse des Empfängers, Beispiel in Abb. 5.5: Kommunikation zwischen Diagnosetester und ECU 1).

Die Geräteadressen müssen vom Hersteller so festgelegt werden, dass sie zumindest innerhalb eines einzelnen Datennetzes, möglichst aber im Gesamtfahrzeug, eindeutig sind. In diesem Fall spricht man von *physischer Adressierung*. Im Anhang A von ISO 14230-2 wird empfohlen, dass Diagnosetester den Adressbereich F0 h ... FDh verwenden, Motorsteuergeräte 10 h ... 17 h, Getriebesteuergeräte 18 h ... 1Fh, ABS, ASR und ESP-Geräte 28 h ... 2Fh usw.

Nachteilig dabei ist, dass der Diagnosetester die Adresse des Steuergerätes vorab kennen muss, mit dem er kommunizieren will. Für herstellerunabhängige Tester, wie sie z. B. von den Behörden zur Überwachung der Abgasemissionen verwendet werden (*OBD Scan Tool*), sind abhängig von der Gerätefunktion für die Empfängeradresse TA feste Werte vorgeschrieben (*funktionale Adressierung*). Steuergeräte, die abgasrelevante Funktionen enthalten, müssen in jedem Fall für die in den OBD-Normen vorgeschriebenen Diagnose-

dienste (SID 00 h ... 0Fh) die funktionale Adressierung unterstützen. Der Diagnosetester muss dabei die physikalische und funktionale Adresse F1 h verwenden. Alle abgasrelevanten Steuergeräte müssen dieselbe funktionale Adresse 33 h, aber eine jeweils eindeutige physikalische Adresse haben. Funktionale Adressen werden nur für die Empfängeradresse TA bei der Testeranfrage verwendet. Die Senderadresse SA ist immer eine physikalische Adresse, damit der Sender eindeutig identifiziert werden kann. Auf eine funktional adressierte Testeranfrage können dabei mehrere Steuergeräte antworten, falls ein Fahrzeug mehrere abgasrelevante Steuergeräte besitzt. Aus den physikalischen Adressen in der Steuergeräteantwort erkennt der Tester, welche Steuergeräte vorhanden sind und welche physikalischen Adressen diese Steuergeräte haben. Weitere Anfragen kann der Tester dann wiederum funktional an alle Geräte oder über die physikalische Adresse gezielt an ein einzelnes Steuergerät richten.

Wird als physikalisches Bussystem der K-Line-Bus verwendet, so werden diese Adressen nach ISO 14230-2 direkt in den Header der übertragenen Botschaft übernommen (vgl. Abschn. 2.2).

Bei CAN als physikalischem Bussystem schlägt die Norm ISO 15765-2 vor, jedem Paar von Sender- und Empfängeradresse, d. h. jeder Verbindung zwischen Tester und Steuergerät einen eindeutigen CAN-Identifier zuzuordnen (in ISO 15765-2 als *Normal Addressing* bezeichnet). Für abgasrelevante Steuergeräte (OBD) werden in ISO 15765-4 die 11 bit CAN-Identifier 7DFh, 7E0 h ... 7EFh vorgeschrieben.

Falls 29 bit CAN Identifier verwendet werden, schlägt ISO 15765-2 vor, für allgemeine Steuergeräte die Identifier 0CDAXYYh (bei physikalischer Adressierung) und 0CDBXXYYh (bei funktionaler Adressierung) zu verwenden, wobei im Feld XX die Target und im Feld YY die Source Adresse einzusetzen ist (in ISO 15765-2 als *Normal Fixed Addressing* bezeichnet). Für abgasrelevante Steuergeräte werden nach ISO 15765-4 die Identifier 18DAXYYh und 18DBXXYYh vorgegeben.

Alternativ kann bei allgemeinen Steuergeräten auch nur die Senderadresse im Identifier codiert werden, während die Empfängeradresse im ersten Nutzdatenbyte der CAN-Botschaft gesendet wird. Diese Adressierungsart, bei der ein Nutzdatenbyte „verschwendet“ wird, wird in der Norm als *Extended Addressing* bezeichnet. Bei Fahrzeugen, in denen mehrere Datennetze über Gateways gekoppelt werden (Abb. 5.5), empfiehlt die Norm eine als *Mixed Addressing* bezeichnete Adressierung, falls Diagnosetester und Steuergerät sich nicht im selben Datennetz befinden. Dabei sind SA und TA grundsätzlich die Adressen des lokalen Datennetzes, in dem die Botschaft gerade übertragen wird. Das heißt, eine der beiden Adressen ist die Adresse des Gateways, über das der andere Kommunikationspartner im entfernten Datennetz (*Remote Network*) erreichbar ist (vgl. im Abb. 5.5: Kommunikation zwischen Diagnosetester und ECU 2). Als dritte Adresse muss dann die *Remote Address RA* (Bezeichnung nach ISO/DIS 15765-3, im Teil ISO 15765-2 leider etwas verwirrend auch als *Address Extension AE* bezeichnet), übertragen werden, d. h. die Adresse des Kommunikationspartners im entfernten Datennetz. Auch bei dieser Adressierungsart werden SA und TA über den CAN Message Identifier codiert, während RA im ersten Nutzdatenbyte der CAN-Botschaft gesendet wird (vgl. dazu auch Abb. 4.1).

Tab. 5.6 K-Line spezifische Dienste nach ISO 14230-2

Service	SID	Parameter/Bemerkung
Start Communication Service Request	81 h	Verbindungsaubau und Abbau sowie Einstellung der Timeout-Parameter des K-Line
Stop Communication Service Request	82 h	Data Link Layers (siehe Abschn. 2.2)
Access Timing Parameter Request	83 h	

Tab. 5.7 CAN spezifische Dienste nach ISO/DIS 15765-3

Service	SID	Parameter/Bemerkung
Network Configuration	84 h	Über diese Botschaft kann der Tester abfragen, welches Adressformat (physikalische, funktionale oder erweiterte Adressen), welcher Typ und welche Werte von CAN Message Identifizieren (11 bit oder 29 bit) für die Diagnosesitzungen verwendet werden können. Für die Anfrage selbst werden vordefinierte CAN-Identifier verwendet (Tester: 7D0 ... 7D3 h, Steuergeräte: 7D4 ... 7DEh), danach kann nach einem vom Hersteller festzulegenden Verfahren auf die anderen CAN-Identifier umgeschaltet werden. In der Regel erfolgt diese Abfrage nur an das Diagnose-Gateway (soweit vorhanden), während die einzelnen Steuergeräte mit festen Werten arbeiten.
Disable Normal Message Transmission	28 h	Der Tester kann das Steuergerät auffordern, das Senden normaler CAN-Botschaften, d. h. der nicht-diagnosebezogenen On-Board-Kommunikation zwischen den Fahrzeugsteuergeräten einzustellen und nur noch mit dem Diagnosetester zu kommunizieren.
Enable Normal Message Transmission	29 h	Kommunikation zwischen den Fahrzeugsteuergeräten einzustellen und nur noch mit dem Diagnosetester zu kommunizieren.
Control DTC Setting	85 h	Der Tester kann das Steuergerät auffordern, das Speichern von Fehlercodes (für alle oder einzelne Fehler herstellerabhängig implementierbar) ein- oder auszuschalten. Abschalten ist z. B. beim Stellertest oder beim Abziehen von Kabeln im Werkstatttest sinnvoll, um überflüssige Fehlermeldungen zu vermeiden. Der Service wäre bei K-Line ebenfalls sinnvoll, ist dort aber nicht definiert.

5.1.4 Bussystem-abhängige Dienste (Network Layer Protocol Control)

Ein Teil der Diagnosedienste ist abhängig vom jeweiligen Bussystem. Bei K-Line werden die Dienste aus Tab. 5.6 verwendet, die schon in Abschn. 2.2 beschrieben wurden. CAN benutzt die Dienste in Tab. 5.7.

Diese Dienste werden in der Regel unmittelbar vor oder nach Einleiten einer Diagnosesitzung verwendet. Bei K-Line ist es üblich, dass vor dem Programmieren des Steuergeräte-Flash-Speichers die Timeout-Werte und der Zeichenabstand auf die Minimalwerte gesetzt werden, um einen höheren Datendurchsatz und damit kürzere Programmierzeiten zu erhalten.

Tab. 5.8 KWP 2000-Dienste für das Lesen und Löschen des Fehlerspeichers

Service	SID	Parameter/Bemerkung
Read Diagnostic Trouble Codes	13 h	Auslesen der gespeicherten Fehlercodes im Steuergerät mit oder ohne zugehörige Statusinformationen.
Read Diagnostic Trouble Codes By Status	18 h	Dabei können entweder alle Fehlercodes oder nur die zu bestimmten Funktionsgruppen gehörenden Fehlercodes ausgelesen werden.
Read Status Of Diagnostic Trouble Codes	17 h	
Read Freeze Frame Data	12 h	Lesen der zu einem oder mehreren Fehlern gespeicherten <i>Umgebungsdaten</i> (Freeze Frame)
Clear Diagnostic Information	14 h	Löschen des gesamten Fehlerspeichers oder der Fehler zu vorgegebenen Funktionsgruppen.

5.1.5 Fehlerspeicher lesen und löschen (Stored Data Transmission)

Zu den wichtigsten Diensten gehört das Lesen und Löschen des Fehlerspeichers nach Tab. 5.8. Die KWP 2000-Normen definieren keine Einzelheiten zu den Fehlercodes (*Diagnostic Trouble Code DTC*) sowie den zugehörigen Statusinformationen und abgespeicherten Umgebungsbedingungen, sondern bezeichnen diese Werte als herstellerspezifisch. In der Praxis müssen sich aber zumindest die Hersteller abgasrelevanter Komponenten an die in den OBD und EOBD-Normen ISO 15031 bzw. SAE J2012 (PKW), J1587 und J1939 (NKW) festgelegten Formate und Fehlercodes halten (siehe Abschn. 5.3.2).

5.1.6 Daten lesen und schreiben (Data Transmission), Ansteuern von Steuergeräte-Ein- und Ausgängen (Input/Output Control)

Mit Hilfe der *Read Data*, *Write Data*, *Read Memory* und *Write Memory* Botschaften kann der Tester Werte aus dem Speicher des Steuergerätes lesen oder in den Speicher schreiben (Tab. 5.9). Welche Werte gelesen oder geschrieben werden, wird in der Botschaft entweder über eine vom Hersteller definierte 1 Byte (*Local Identifier*) oder 2 Byte (*Common Identifier*) lange Kennzahl oder durch Angabe einer 24 bit Speicheradresse (*Memory Address*) und einer 1 Byte Längenangabe (*Memory Size*) vorgegeben. Beim Lesen wird außerdem vorgegeben, ob die Ausgabe durch das Steuergerät einmalig erfolgen oder bis zu 255 mal wiederholt werden soll. Bei wiederholter Ausgabe kann der Tester die Ausgaberate in 3 Stufen (*slow, medium, fast*) auswählen. Der zugehörige konkrete Wert der Datenrate kann vom Tester über eine *Set Data Rates* Botschaft vorher aus einem vom Hersteller festgelegten Satz von Datenraten ausgewählt werden. Erhält das Steuergerät während der periodischen Ausgabe vom Tester eine neue *Request* Botschaft, wird die periodische Ausgabe abgebrochen.

Tab. 5.9 KWP 2000-Dienste für das Auslesen von Steuergerätedaten und das Ansteuern von Steuergeräte-Ein- und Ausgängen

Service	SID	Parameter/Bemerkung
Read Data By Local Identifier	21h	Auslesen von Steuergerätedaten im Speicher des Steuergerätes. Die Werte werden über Kennziffern (1 Byte oder 2 Byte) oder Speicheradresse (3 Byte) und Längenangabe (1 Byte) ausgewählt. Die Ausgabe kann einmalig oder bis zu 255 mal mit stufenweise vorgebbarer Wiederholrate erfolgen.
Read Data By Common Identifier	22h	
Read Memory By Address	23h	
Set Data Rates	26h	Auswahl der Datenrate für die periodische Ausgabe aus einer vom Hersteller vordefinierten Liste von Werten.
Write Data By Local Identifier	3Bh	Schreiben von Werten in den Speicher des Steuergerätes.
Write Data By Common Identifier	2Eh	Auswahl über Kennziffer oder Speicheradresse.
Write Memory By Address	3Dh	
Dynamically Define Local Identifier	2Ch	Dynamische Zusammenstellung von Datenwerten zu einem Datensatz und Zuordnung zu einer Kennziffer
Input Output Control By Local Identifier	30h	Überschreiben eines Steuergeräte-Eingangs oder Direktansteuerung eines Steuergeräte-Ausgangs. Details der Ansteuerung sind über herstellerspezifische Parameter der Botschaft vorgebar.
Input Output Control By Common Identifier	2Fh	

Bedeutung, Formatierung der Werte (Datensätze, *Records*) und der zugehörigen Kennziffern (*Identifier*) werden herstellerspezifisch festgelegt. Mit Hilfe der *Dynamically Define Local Identifier* Botschaft kann der Tester im Steuergerät eine neue Kennziffer für die aktuelle Diagnosesitzung dynamisch festlegen. Durch die Parameter der Botschaft werden dieser Kennziffer einzelne Datenwerte aus Datensätzen existierender Kennziffern oder durch Angabe von Speicheradresse und Länge aus dem Datenspeicher des Steuergerätes zugeordnet. Diese Vorgehensweise ist insbesondere für periodische Messaufgaben sinnvoll, wenn der Tester auswählen will, welche Messwerte angezeigt werden sollen und für diese Zusammenstellung noch keine vordefinierte Kennziffer existiert.

Die *Input Output Control* Botschaften sind zum zeitweiligen Überschreiben von Steuergeräte-Eingangssignalen oder zur Direktansteuerung von Steuergeräteausgängen vorgesehen. Über die herstellerspezifischen Parameter dieser Botschaften lassen sich dabei beliebige Details wie Ansteuerdauern, Zykluszahlen usw. vorgeben. Da auch bei den *Write Data* Botschaften die Botschaftsparameter herstellerspezifisch sind, lässt sich mit diesen Botschaften prinzipiell dieselbe Wirkung erzielen.

Tab. 5.10 KWP 2000-Dienste für das Übertragen größerer Datenblöcke

Service	SID	Parameter/Bemerkung
Request Download	34 h	Initialisierung der Übertragung von größeren Datenblöcken vom Diagnosetester zum Steuergerät
Request Upload	35 h	Initialisierung der Übertragung von größeren Datenblöcken vom Steuergerät zum Diagnosetester
Transfer Data	36 h	Eigentliche Übertragung der Datenblöcke
Request Transfer Exit	37 h	Beenden der Datenübertragung

5.1.7 Speicherblöcke auslesen und speichern (Upload, Download)

Mit den folgenden Botschaften (Tab. 5.10) lassen sich größere Datenmengen zwischen Tester und Steuergerät oder umgekehrt übertragen. Vorgesehen sind diese Botschaften, um den Steuergerätespeicher auszulesen oder neue Programme und Datensätze in das Steuergerät zu laden. Diese Botschaften stellen die Kernbotschaften für das *Flashen* oder die *End of Line*-Programmierung dar.

Die Einzelheiten der Übertragung, z. B. Blockgrößen, Checksummen und Handshaking auf der Anwendungsebene sind herstellerabhängig. Üblich ist, dass der Tester über die *Request Download* bzw. *Request Upload* Botschaft die Übertragung anfordert und dem Steuergerät mitteilt, welche Speicherbereiche und wie viele Daten übertragen werden sollen. Das Steuergerät antwortet, welche Blockgröße es erwartet. Die eigentlichen Daten werden dann als Parameter der *Transfer Data* Botschaft bzw. der zugehörigen Steuergeräteantwort übertragen. Mit der *Request Transfer Exit* Botschaft beendet der Tester die Übertragung.

5.1.8 Start von Programmen im Steuergerät (Remote Routine Activation)

Über die Dienste in Tab. 5.11 lassen sich Programmroutine im Steuergerät starten, stoppen und Ergebnisse abfragen. Dabei lassen sich wiederum herstellerabhängig beliebige Parameter übergeben.

Typische Anwendungen bestehen beispielsweise in so genannten Stellgliedtests, d. h. dem Starten bestimmter Routinen, mit denen z. B. die Druckdichtheit eines Einspritzsystems geprüft wird. Eine weitere Anwendung ist der Start einer Programmierroutine für den Flash-Speicher, nachdem über die *Download*-Funktionen ein neues Programm in das Steuergerät geladen wurde. Auch das Ausführen von Prüfroutinen im Fertigungstest, die mit den *Download*-Funktionen geladen wurden, wird so realisiert.

Erhält das Steuergerät eine neue *Request* Botschaft vom Diagnosetester, während es noch mit dem Ausführen einer Routine, einem *Up*- oder *Download* oder dem Löschen

Tab. 5.11 KWP 2000-Dienste für das Starten von Programmen im Steuergerät

Service	SID	Parameter/Bemerkung
Start Routine By Local ID	31 h	Starten und Stoppen vordefinierter oder in das Steuergerät geladener Routinen durch den Diagnosetester.
Start Routine By Address	38 h	
Stop Routine By Local ID	32 h	
Stop Routine By Address	39 h	(ID Identifier ... Kennziffer)
Request Routine Results By Local Identifier	33 h	Abfrage der Ergebnisse der Routine
Request Routine Results By Address	3Ah	

Tab. 5.12 Realisierung zusätzlicher KWP 2000-Dienste

Service	SID	Parameter/Bemerkung
Escape Code Request	80 h	Beliebige herstellerdefinierte Services, die über die weiteren Parameter der Botschaft ausgewählt werden.

des Fehlerspeichers beschäftigt ist, kann es die neue Botschaft mit einer Fehlermeldung, z. B. *Response pending* oder *Busy – repeat request* ablehnen (Tab. 5.2).

5.1.9 Erweiterte Dienste (Extended Services)

Mit der *Escape Code Request* Botschaft können beliebige, in keine der in der Norm vordefinierten Kategorien passende Dienste realisiert werden (Tab. 5.12). Sämtliche Parameter der Botschaft sind herstellerabhängig.

Beispiele für den Einsatz von Diensten für das Auslesen von Diagnoseinformationen werden in Abschn. 5.3.7 und Beispiele für die Flash-Programmierung von Steuergeräten in Abschn. 9.4 vorgestellt.

5.2 Unified Diagnostic Services UDS nach ISO 14229/15765-3

Mit ISO 14229 wird der Versuch unternommen, die mit der KWP 2000-Diagnose eingeführten Prinzipien zu verallgemeinern und von dem darunter liegenden, realen Busprotokoll unabhängig zu machen. Die Implementierung von UDS in CAN-Bussystemen wird in ISO 15765-3 beschrieben. Die gesamten in Abschn. 5.1 dargestellten Prinzipien gelten, mit Ausnahme der nachfolgend dargestellten Änderungen, weiter.

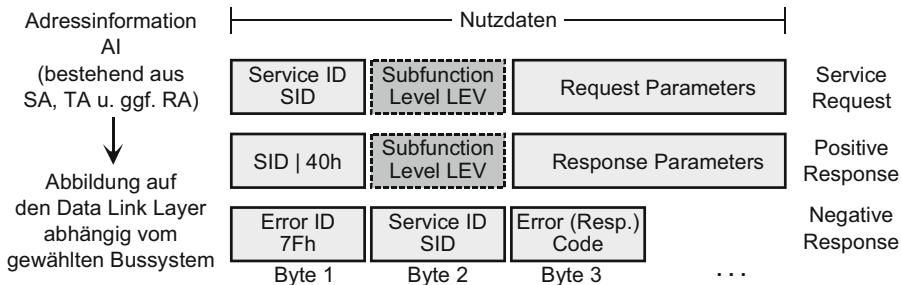


Abb. 5.6 Aufbau UDS Application Layer Service Botschaften

5.2.1 Unterschiede zum KWP 2000 Diagnoseprotokoll

Die Menge der *Service Identifier SID* wurde gestrafft, indem einige KWP 2000-Dienste bei UDS als Unterfunktionen zu einem einzigen Dienst mit einer gemeinsamen SID zusammengefasst wurden. Die Auswahl der Unterfunktion erfolgt bei diesen *Requests* über den neuen Parameter *Subfunction Level LEV* (Abb. 5.6). Bei den positiven Antwortbotschaften wird meist zusätzlich zur SID und den eigentlichen Antwortdaten auch dieser Parameter LEV zurückgesendet. Die übrigen Parameter der *Request* und *Response* Botschaften sind im Vergleich zu den KWP 2000 Normen etwas präziser spezifiziert. Es besteht aber weiter herstellerspezifischer Freiraum für die Implementierung. LEV darf die Werte 00 h ... 7Fh aufweisen. Das Bit 7 des LEV-Bytes dagegen hat eine Funktion für die Kommunikationssteuerung. Wenn es in einer *Request* Botschaft auf 1 gesetzt ist, darf die sonst für jede *Request* Botschaft notwendige *Positive Response* Botschaft durch das Steuergerät entfallen, falls beim Ausführen des Dienstes kein Fehler aufgetreten ist. Im Fehlerfall dagegen muss das Steuergerät in jedem Fall eine *Negative Response* Botschaft senden.

Bei KWP 2000 wurden Datensätze oder Funktionen im Steuergerät entweder durch deren Speicheradresse oder durch Kennziffern identifiziert. Dabei wurden 8 bit Kennziffern (*Local Identifier*) und 16 bit Kennziffern (*Common Identifier*) verwendet. UDS verwendet ausschließlich 16 bit Kennziffern, die Begriffe *Local* und *Common* entfallen. Einige Kennziffern, z. B. für den Applikationsdatensatz, für Geräteserien- und Versionsnummern, Herstelleridentifikation, OBD-relevante Daten oder periodisch zu sendende Daten sind in der Norm vorgegeben. Mit Hilfe der *Dynamically Define Data Identifier* Botschaft können wieder eigene Datensätze aus vorhandenen Datensätzen oder aus Daten, deren Speicheradressen bekannt sind, zusammengestellt werden.

5.2.2 Überblick über die UDS-Diagnosedienste

Das Management der Diagnosesitzungen erfolgt wie bei KWP 2000 (Tab. 5.13). Die verschiedenen Sitzungstypen wurden allerdings neu nummeriert (*Default Session* 01 h,

Tab. 5.13 UDS-Dienste für das Management der Diagnosesitzungen

Service	SID	LEV	Parameter/Bemerkung	Ersetzt SID KWP 2000
Diagnostic Session Control	10 h	01 h	Starten einer Diagnosesitzung	10 h
		02 h	Default Session	
		03 h	Programming Session	
		40 h	Extended Diagnostic Session	
		...7Eh	Herstellerspezifische Sitzungen	
Security Access	27 h	01 h	Request Seed	27 h
		02 h	Send Key	
		...	Weitere herstellerspez. Werte	
Secured Data Transmission	84 h		Verschlüsselung und Entschlüsselung der übertragenen Daten nach ISO 15764	NEU
ECU Reset	11 h	01 h	Rücksetzen des Steuergerätes	11 h
		02 h	Hard Reset	
		04 h	Zündung aus – Zündung ein	
		40 h	Soft Reset	
		...7Eh	Herstellerspezifische Reset-Sequenzen	
Tester Present	3Eh		Keep-Alive-Botschaft, um eine Diagnose-Session aufrecht zu erhalten, solange keine anderen Diagnosebotschaften vorliegen.	3Eh

Programming Session 02 h usw.) und es wurde festgelegt, welche Diagnosedienste in der *Default Diagnostic Session* unterstützt werden müssen (Tab. 5.14).

Alle anderen Dienste, z. B. zum Lesen oder Neuprogrammieren des gesamten Steuergerätespeichers, dürfen lediglich in speziellen Sitzungen verwendet werden. Diese können in der Regel nur unter besonderen Bedingungen, z. B. nach einer Login-Prozedur mit einer *Security Access* Botschaft, gestartet werden.

Neu ist die Möglichkeit, die Datenübertragung über das Bussystem verschlüsselt durchzuführen. Dabei wird eine Verschlüsselung nach ISO 15764 verwendet. Die Verschlüsselung und Entschlüsselung erfolgt im Sender und Empfänger auf einer zusätzlichen Protokollsicht (*Security Layer*), die zwischen die Diagnose- bzw. Steuergeräte-Anwendung und den UDS-Application Layer eingefügt wird (Abb. 5.7). Die Anwendung selbst sieht weiterhin normale UDS-Dienste im *offenen Klartext*, die aber gegenüber dem UDS-Protokollstapel verschlüsselt und als *Secured Data Transmission* Botschaften mit SID 84 h übertragen werden.

Die Botschaften zur Steuerung der Transport- und Datenübertragungsschicht (Tab. 5.15) hängen naturgemäß weiterhin stark vom verwendeten Bussystem ab.

Die vielleicht interessanteste Neuerung bei UDS stellt der *Response on Event* Dienst dar. Er erlaubt dem Tester, im Steuergerät einen oder mehrere Ereignistrigger zu installieren.

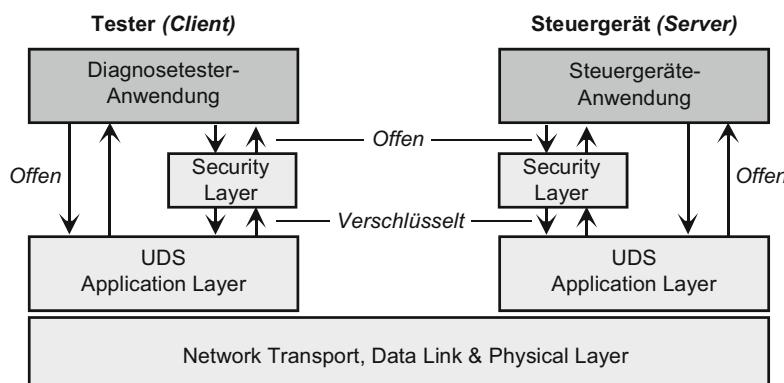
Tab. 5.14 Unterstützte Dienste in der UDS Default Diagnostic Session

Geforderte Diagnosedienste in der Default Diagnostic Session	
10 h, 3Eh	Starten und Aufrechterhalten einer Diagnosesitzung
11 h	Steuergeräte-Reset
14 h, 19 h	Fehlerspeicher lesen und löschen
Optionale Dienste in der Default Diagnostic Session	
86 h	Ereignisgesteuerte Botschaften (neu)
Bei den folgenden Diensten kann der Hersteller festlegen, auf welche Daten in der Default Diagnostic Session und auf welche nur in einer speziellen Diagnosesitzung zugegriffen werden kann:	
22 h, 23 h	Lesen und Schreiben von Steuergeräte-Daten
24 h, 2Ch	(basierend auf Speicheradressen oder Kennziffern).
2Eh, 3Dh	Starten von Programmen im Steuergerät
31 h	

Beim Eintreffen des Ereignisses (Triggern des Events) sendet das Steuergerät dann spontan entsprechende Antwortbotschaft an den Tester, ohne auf eine weitere Testerabfrage warten zu müssen (Tab. 5.16).

Die Dienste zum Zugriff auf den Fehlerspeicher (Tab. 5.17) sind detaillierter beschrieben als bei KWP 2000. Fehlercodes, der Aufbau des Fehlerspeichers und die zu einem Fehler gespeicherten Statusinformationen und Umgebungsbedingungen sind aber weiterhin nicht definiert. Für den Fehlercode sind 3 Byte vorgesehen, im Übrigen wird auf ISO 15031-6, SAE J1939-73 oder eine herstellerspezifische Festlegung verwiesen.

Im Vergleich zu KWP 2000 wurden die einmalige und die periodischen Datenausgabe in zwei verschiedene Dienste aufgespalten (Tab. 5.18). Die Datenrate für die periodische Ausgabe kann weiterhin in 3 Stufen verstellt werden, die im Steuergerät fest vordefiniert sind. Eine Feineinstellung in mehr als 3 Stufen, wie sie über die KWP 2000-Botschaft *Set Data Rates* möglich war, ist bei UDS nicht mehr vorgesehen.

**Abb. 5.7** Verschlüsselte und unverschlüsselte (offene) UDS-Dienste

Tab. 5.15 UDS-Dienste für das Management der Buskommunikation

Service	SID	LEV 6...0	Parameter/Bemerkung	Ersetzt SID KWP 2000
Communication Control	28 h	...	Der Tester kann das Steuergerät auffordern, das Senden und/oder Empfangen normaler Botschaften, d. h. der nicht-diagnosebezogenen On-Board-Kommunikation zwischen den Fahrzeugsteuergeräten einzustellen und nur noch mit dem Diagnosetester zu kommunizieren.	28 h, 29 h
Access Timing Parameter	83 h	01 h	Auslesen und Setzen der Timing-Parameter des Steuergerätes; Details abhängig vom verwendeten Bussystem	83 h
		02 h	Auslesen der vom Steuergerät unterstützten Werte	
		03 h	Zurücksetzen auf die Default-Werte	
		04 h	Auslesen der aktuell eingestellten Werte	
			Setzen neuer Timing-Parameter	
Link Control	87 h	...	Umschalten der Bitrate; Details abhängig vom verwendeten Bussystem. Der Tester teilt dem Steuergerät mit, dass er auf eine andere, vordefinierte oder in der Anfrage angegebene Bitrate umschalten möchte und führt bei positiver Antwort des Steuergerätes die Umschaltung durch. Die Umschaltung ist nur in speziellen Diagnosesitzungen möglich, die Default Diagnostic Session arbeitet immer mit der festen Standard-Bitrate.	NEU

Die bei KWP 2000 übliche Unterscheidung von 1 Byte Kennziffern (*Local Identifier*) und 2 Byte Kennziffern (*Common Identifier*) zur Auswahl von Steuergerätewerten oder Steuergeräteprogrammen entfällt. Bei UDS haben Kennziffern (*Identifier*) durchgängig 2 Byte.

Neu ist die Möglichkeit, über *Read Scaling Data* die Normierung und Skalierung der steuergeräteinternen Daten, d. h. den Zusammenhang zwischen den im Gerät gespeicherten Hexadezimalzahlen und den realen physikalischen Werten abzufragen. Dabei werden die Zahlendarstellung (Ganze Zahl, BCD- oder Gleitkommazahl, Anzahl der Bytes), die verwendeten Einheiten (Meter, Feet, km/h, Meilen/h usw.) und die Skalierungsfaktoren

Tab. 5.16 UDS-Ereignisgesteuerte Ausführung eines Dienstes

Service	SID	LEV 6...0	Parameter/Bemerkung	Ersetzt SID KWP 2000
Response on Event	86 h	...	Installieren eines Diensts, der bei Eintreffen eines Triggerereignisses im Steuergerät ausgeführt wird. Einzelheiten siehe Abschn. 5.2.3	NEU

ermittelt, d. h. Steigung und Offset der folgenden Umrechnungsformel:

$$\text{Physikalischer Wert} = \text{Hexadezimalwert} \times \text{Steigung} + \text{Offset}$$

Die Dienste zum Übertragen größerer Datenblöcke (Tab. 5.19) sind funktional mit den KWP 2000-Diensten identisch, wobei die Parameterformate für Speicheradresse und Speichergröße sowie die Blocklänge und Blockzähler jetzt präzise definiert sind. Außerdem können für die Übertragung herstellerdefinierte Kompressions- und Verschlüsselungsformate angefordert werden. Die beispielhafte Anwendung dieser Dienste zur Flash-Programmierung wird im Abschn. 9.4 beschrieben.

Wiederum können vordefinierte oder vorher geladene Steuergerätefunktionen gestartet werden (Tab. 5.20). Das Format der Parameter, die an die zu startende Routine übergeben werden, ist ebenso wie das Format der Ergebnisse weiterhin herstellerabhängig. Die Routinen werden über Kennziffern (*Identifier*) ausgewählt, wobei sowohl vordefinierte Kennziffern, z. B. die so genannten OBD *Test IDs*, als auch herstellerdefinierte Werte möglich sind. Die bei KWP 2000 vorhandene Möglichkeit, Routinen direkt über ihre Speicheradressen zu starten, existiert nicht mehr, doch kann über eine vorherige *Dynamically Define Data Identifier* Botschaft einer Speicheradresse eine Kennziffer zugewiesen werden.

Ob eine Routine durch eine Stopp-Botschaft des Diagnosetesters beendet wird oder selbstständig nach einer bestimmten Zeit oder Anzahl von Durchläufen endet, ist abhängig von Art und Funktion der Routine.

5.2.3 Response on Event Dienst

Der Ablauf bei der ereignisgesteuerten Ausführung von UDS-Diensten ist wie folgt (Abb. 5.8):

- Installieren mit Angabe des auszuführenden Dienstes, der Triggerbedingung und des Trigger-Zeitfensters (LEV = 01 h – 03 h oder 07 h, siehe unten)
- Starten des Event-Dienstes (*Start Response on Event* LEV = 05 h), Beginn des Trigger-Zeitfensters
- Senden von Antworten bei Eintreffen des Ereignisses vom Steuergerät zum Diagnosetester

Tab. 5.17 UDS-Dienste für das Lesen und Löschen des Fehlerspeichers

Service	SID	LEV	Parameter/Bemerkung	Ersetzt SID KWP 2000
Read DTC Information	19 h	6...0	Auslesen des Fehlerspeichers Liste aller möglichen Fehlercodes des Steuergerätes und den zugehörigen aktuellen Status	
	0Ah	01 h	Anzahl und Liste der Fehler mit einer bestimmten Fehlermaske	12 h, 13 h,
	02 h	02 h	Anzahl und Liste der Fehler mit einer bestimmten Fehlermaske	17 h, 18 h
	07 h	07 h	Anzahl und Liste der Fehler mit einer bestimmten Fehlerschwere	
	...09 h	12 h	Anzahl und Liste aller ODB-relevanten Fehler	
	13 h	0Bh	Ältesten bzw. jüngsten Fehlerspeicher-Eintrag lesen	
	0Bh	...0Eh	Auslesen der gespeicherten Umgebungsbedingungen und erweiterten Statusinformationen	
	03 h	...06 h	Auslesen der gespeicherten Umgebungsbedingungen und erweiterten Statusinformationen	
	0Fh	0Fh	Fehler aus dem Fehlerspiegel auslesen	
	...11 h	...11 h		
Clear Diagnostic Information	14 h		Löschen des gesamten Fehlerspeichers, der Fehler zu einer vorgegebenen Funktionsgruppe (z. B. OBD-relevante Fehler, Fehler des Motorsteuergerätes usw.) oder zu einem einzelnen Fehlercode	14 h
Control DTC Setting	85 h	...	Der Tester kann das Steuergerät auffordern, das Speichern von Fehlercodes (für alle oder einzelne Fehler herstellerabhängig implementierbar) ein- oder auszuschalten. Abschalten ist z. B. beim Stellertest oder beim Abziehen von Kabeln im Werkstatttest sinnvoll, um überflüssige Fehlermeldungen zu vermeiden.	85 h

- Stoppen des Event-Dienstes durch das Steuergerät nach Ablauf des Trigger-Zeitfensters oder explizites Stoppen durch den Diagnosetester (*Stop Response on Event* LEV = 00 h)
- Deinstallieren des Event-Dienstes (*Clear Response on Event* LEV = 06 h)

Das Trigger-Zeitfenster (*Event Window*) definiert den Zeitbereich (ab Starten des Dienstes), in dem auf das Trigger-Ereignis gewartet wird. Nach Ablauf dieses Zeitfensters beendet das Steuergerät das Warten auf das Triggerereignis selbstständig. Das explizite Stoppen des Event-Dienstes ist nur notwendig, wenn das Trigger-Zeitfenster unendlich groß gewählt

Tab. 5.18 UDS-Dienste für das Auslesen von Steuergerätedaten und das Ansteuern von Steuergeräte-Ein- und Ausgängen

Service	SID	LEV	Parameter/Bemerkung	Ersetzt SID KWP 2000
			6...0	
Read Data By Identifier	22 h	–	Einmaliges Auslesen von Steuergerätwerten (Einzelwerte oder Datensätze) im Speicher des Steuergerätes.	21 h, 22 h Single
Read Memory By Address	23 h	–	Die Werte werden über eine oder mehrere 2 Byte Kennziffern oder Speicheradresse und Längenangabe (mit wählbarem Adressformat) ausgewählt. Vordefinierte Identifier existieren z. B. zur Abfrage der unterstützten CAN IDs.	23 h Single
Read Data By Periodic Identifier	2Ah		Periodisches Auslesen von Steuergerätwerten. Die periodisch zu sendenden Daten müssen über spezielle Kennziffern identifiziert werden. Optional kann dabei das Transportprotokoll umgangen werden, um mehr Nutzdaten je CAN-Botschaft übertragen zu können.	21 h, 22 h Periodic
	01 h		Senden mit niedriger Datenrate	
	02 h		Senden mit mittlerer Datenrate	
	03 h		Senden mit hoher Datenrate	
	04 h		Senden beenden	
Write Data By Identifier	2Eh	–	Schreiben von Werten in den Speicher des Steuergerätes.	2Eh, 3Bh
Write Memory By Address	3Dh	–	Auswahl erfolgt über Kennziffer oder Speicheradresse	3Dh
Read Scaling Data By Identifier	24 h	–	Auslesen von Skalierungs- und Normierungsinformationen für Einzelwerte oder Datensätze.	NEU
Dynamically Define Data Identifier	2Ch		Dynamische Zusammenstellung von Datenwerten zu einem Datensatz und Zuordnung zu einer Kennziffer.	2Ch
	01 h		Auswahl aus vorhandenem Datensatz	
	02 h		Auswahl über Speicheradresse	
	03 h		Löschen von Kennziffern	
Input Output Control By Identifier	2Fh	–	Überschreiben eines Steuergeräte-Eingangs oder Direktansteuerung eines Steuergeräte-Ausgangs. Details der Ansteuerung sind über die herstellerspezifischen Parameter der Botschaft vorgebbar. Über spezielle Parameter kann ein Wert verändert, der veränderte Wert eingefroren, auf einen Defaultwert umgeschaltet oder die Kontrolle über den Ein-/Ausgang an das Steuergerät zurückgegeben werden.	2Fh, 30 h

Tab. 5.19 UDS-Dienste für das Übertragen größerer Datenblöcke

Service	SID	LEV 6...0	Parameter/Bemerkung	Ersetzt SID KWP 2000
Request Download	34 h	–	Initialisierung der Übertragung von größeren Datenblöcken vom Diagnosetester zum Steuergerät	34 h
Request Upload	35 h	–	Initialisierung der Übertragung von größeren Datenblöcken vom Steuergerät zum Diagnosetester	35 h
Transfer Data	36 h	–	Eigentliche Übertragung der Datenblöcke	36 h
Request Transfer Exit	37 h	–	Beenden der Datenübertragung	37 h

wurde oder wenn das Zeitfenster vorzeitig beendet werden soll. Der mögliche Wertebereich für das Zeitfenster ist herstellerspezifisch.

Folgende Ereignisse können als Triggerbedingung im Steuergerät verwendet werden:

- bei Änderung eines Fehlerspeichereintrags (*On DTC Status Change* LEV = 01 h),
- bei einem Zeitgeber-Interrupt (*On Timer Interrupt* LEV = 02 h), erlaubt periodische Triggerung,
- bei Änderung eines Datenwertes (*On Change of Data Identifier* LEV = 03 h),
- Vergleich eines Datenwertes mit einer Konstanten (größer, kleiner, Gleichheit, Ungleichheit, ggf. mit Hysterese, *On Comparison of Values* LEV = 07 h).

Folgende Dienste können durch das Event im Steuergerät ausgeführt werden:

- Auslesen von Steuergerätewerten (*Read Data by Identifier* SID = 22 h),
- Auslesen des Fehlerspeichers (*Read DTC Information* SID = 19 h),
- Ausführen von Programmrutinen im Steuergerät (*Routine Control* SID = 31 h),
- Ansteuern von Steuergeräte-Ein- und Ausgängen (*Input/Output Control by Identifier* SID = 2Fh).

Es ist möglich, mehrere Event-Dienste gleichzeitig zu aktivieren. Mit LEV = 04 h kann abgefragt werden, welche Event-Dienste aktuell installiert sind. Während ein oder meh-

Tab. 5.20 UDS-Dienste für das Starten von Programmen im Steuergerät

Service	SID	LEV 6...0	Parameter/Bemerkung	Ersetzt SID KWP 2000
Routine Control	31 h		Ausführen vordefinierter oder in das Steuergerät geladener Routinen	
	05 h		Starten	31 h, (38 h)
	00 h		Stoppen	32 h, (39 h)
	03 h		Abfragen der Ergebnisse	33 h, (3Ah)

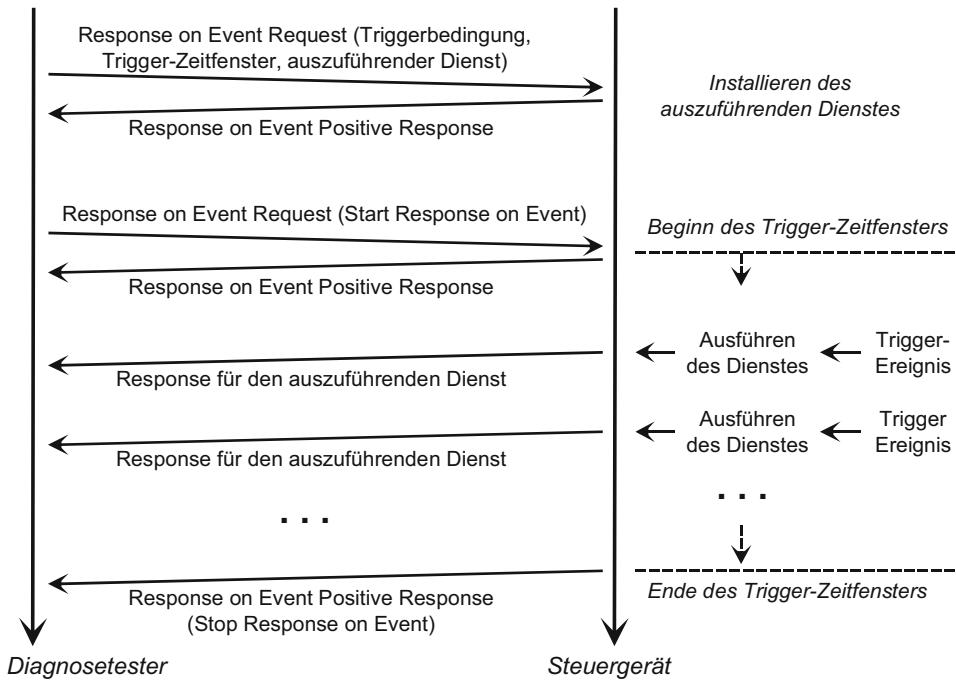


Abb. 5.8 Ablauf beim UDS-Dienst *Response on Event*

reere Event-Dienste installiert und gestartet sind, muss das Steuergerät beliebige weitere Botschaften vom Diagnosetester (mit Ausnahme von SID = 28 h, 2Ch, 31 h, 34 h bis 37 h) verarbeiten können.

Mit Hilfe von Bit 6 des LEV-Parameters kann eine Triggerbedingung auch so installiert werden, dass sie nach einem Steuergeräte-Reset weiter aktiv bleibt (*Store Event*).

5.3 On-Board-Diagnose OBD nach ISO 15031/SAE J1979

Zur Überwachung von abgasrelevanten Systemen wie z. B. Motorsteuergeräten, schreiben gesetzliche Vorschriften in USA und Europa die so genannte On-Board-Diagnose (OBD, EOBD) vor. Die für die Übertragung der zugehörigen Diagnosedaten relevanten Spezifikationen wurden im Normensatz ISO 15031 zusammengefasst, der mit den entsprechenden amerikanischen SAE-Normen nahezu inhaltsgleich ist (Tab. 5.21).

Als unterlagertes Bussystem lässt die Norm K-Line-Busse mit KWP 2000 und dessen Vorläufer ISO 9141-2 CARB, CAN nach ISO 15765-2/-4 sowie die vor allem in amerikanischen Fahrzeugen verbreiteten SAE J1850-Busse in PWM und VPWM-Ausführung zu. Während ein Fahrzeug dabei an seiner Diagnoseschnittstelle in der Regel nur eines dieser Bussysteme unterstützt, muss ein normgerechter Diagnosetester (*OBD Scan Tool*) alle zu-

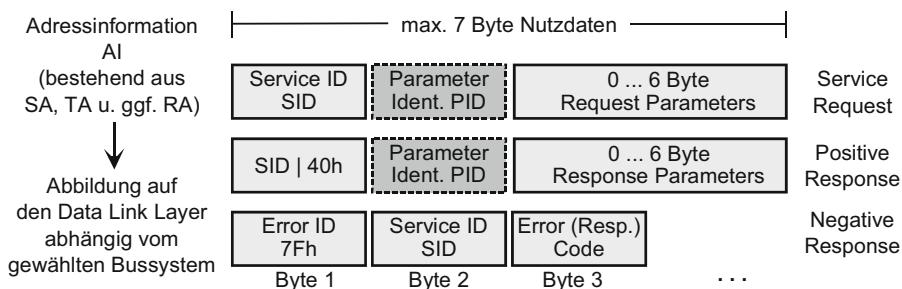
Tab. 5.21 Normen für OBD Diagnoseprotokolle

Thema	ISO-Norm	SAE-Norm
Allgemeines	ISO 15031-1	-
Begriffsdefinitionen und Abkürzungen	ISO 15031-2	J1930
Steckverbinder	ISO 15031-3	J1962
Diagnosetester (OBD Scan Tool)	ISO 15031-4	J1978
Diagnosedienste	ISO 15031-5	J1979
Fehlercodes (Diagnostic Trouble Codes)	ISO 15031-6	J2012
Data Link Security	ISO 15031-7	J2186
Alternativen für das Bussystem	ISO 9141-2 K/L-Line CARB ISO 14230 K-Line KWP 2000 ISO 15765-2/-4 CAN SAE J1850 PWM und VPWM	

lässigen Bussysteme unterstützen und automatisch erkennen. In USA ist seit 2007 für alle Fahrzeuge CAN vorgeschrieben. Mittlerweile soll OBD als ISO 27145 weltweit in einheitlicher Form standardisiert werden.

5.3.1 Überblick OBD-Diagnosdienste

Die OBD *Request* und *Response* Botschaften besitzen das bereits in Abschn. 5.1.1 beschriebene KWP 2000 Format (Abb. 5.9), wobei der Diagnosetester die in Abschn. 5.1.3 beschriebene funktionale Adressierung verwenden muss. Der Diagnosetester verwendet die funktionale und physikalische Adresse F1 h, die OBD-relevanten Steuergeräte die funktionale Adresse 33 h. SAE J1850 verwendet andere Adresswerte. Während es stets nur einen Diagnosetester geben darf, werden in der Regel mehrere OBD-relevante Steuergeräte auf diese funktionale Adresse reagieren. Der Diagnosetester muss daher stets mit mehreren Antworten rechnen, die er anhand der in den Antwortbotschaften enthaltenen, stets eindeutigen physikalischen Adresse des sendenden Steuergerätes unterscheiden kann.

**Abb. 5.9** OBD-Botschaftsformat (bei CAN längere Antwortbotschaft möglich)

Die Botschaften bestehen neben der SID aus maximal 6 Datenbytes, d. h. insgesamt maximal 7 Byte. Bei CAN dürfen *Response*-Botschaften länger sein. Das erste Datenbyte nach der SID dient bei den meisten *Request* Botschaften zur Auswahl eines Parameters (*Parameter Identifier PID*), z. B. eines Datenwertes im Steuergerät, und wird in der Antwortbotschaft ebenfalls als Echo zurückgesendet.

In den SAE-Normen werden die Diagnosedienste, die bei ISO als *Services* bezeichnet werden, auch *Test Modes* oder einfach nur *Modes* genannt (Tab. 5.22).

5.3.2 Auslesen des Fehlerspeichers und von Steuergeräterewerten

Zu Beginn eines Diagnosetests wird der Diagnosetester üblicherweise mit dem *Service 09 h Request Vehicle Information* zunächst die Serien- und Versionsnummern des Fahrzeugs und der Steuergerätesoftware abfragen, um das Fahrzeug eindeutig zu identifizieren.

Danach wird der Diagnosetester mit dem *Service SID = 01 h PID = 01 h Request Current Power Train Diagnostic Data* den Zustand der Fehlerlampe im Armaturenbrett (*Malfunction Indicator Lamp MIL*) und die Anzahl der im Steuergerät gespeicherten Fehler abfragen. Das erste Byte der Steuergeräte-Antwort enthält im obersten Bit den Zustand der MIL-Lampe (Bit 7 = 1 entspricht MIL = ein), die unteren 7 bit enthalten die Anzahl der gespeicherten Fehler. Das zweite Byte zeigt an, welche Überwachungsfunktionen im Steuergerät implementiert sind (Bit 0 = 1 Zündaussetzerüberwachung, Bit 1 = 1 Einspritzsystemüberwachung, Bit 2 = 1 sonstige Überwachung, Bit 4, 5 und 6 sollen anzeigen, ob die jeweiligen Überwachungsfunktionen vollständige Testzyklen durchführen konnten, d. h. ob die zugehörigen Überwachungsdaten gültig sind). Das dritte Byte zeigt, welche Komponenten jeweils überwacht werden (von Bit 0 ausgehend Katalysator, Katalysator-Heizung, Tankentlüftung, Sekundärluftsystem, Klimaanlagen-Kältemittel, Lambdasonde, Lambdasondenheizung, Abgasrückführung). Das vierte Byte zeigt wiederum bitweise an, ob die jeweilige Überwachung einen vollständigen Testzyklus ausführen konnte.

Anschließend wird der Diagnosetester mit dem *Service 03 h Request Emission-Related Diagnostic Trouble Codes* den Fehlerspeicher für die OBD-relevanten Fehler abfragen. Die Antwortbotschaft des Steuergerätes enthält grundsätzlich drei 16 bit-Fehlercodes, die so codiert sind, wie im folgenden Abschn. 5.3.4 beschrieben. Falls gar kein oder weniger Fehler gespeichert sind, wird 0000 h für den Fehlercode gesetzt. Falls mehr als drei Fehler gespeichert sind, antwortet das Steuergerät bei K-Line ohne erneute Anfrage mit mehreren Antwortbotschaften, bei CAN lässt die Transportschicht ISO 15765-2 ohnehin längere Antworten zu. Aus der vorigen Anfrage mit SID = 01 h PID = 01 h weiß der Diagnosetester, wie viele Fehler gespeichert sind und damit, wie viele Antwortbotschaften vom Steuergerät zu erwarten sind. Der Service 03 h liefert nur als *endgültig defekt* eingestufte Fehler, während man mit dem Service 07 h, der ansonsten gleich aufgebaut ist, auch Fehlercodes für Fehler abfragen kann, die lediglich als *vorläufig defekt* eingestuft wurden.

Zu jedem Fehlercode kann der Diagnosetester mit dem *Service 02 h Request Power Train Freeze Frame Data* im Steuergerät gespeicherte Umweltbedingungen (*Freeze Frame*) ab-

Tab. 5.22 OBD-Diagnosedienste

Service	SID	Parameter/Bemerkung
<i>Fehlerspeicher</i>		
Request emission-related diagnostic trouble codes	03 h	Auslesen der abgasrelevanten Fehler aus dem Fehlerspeicher (kein PID). Es werden nur als <i>endgültig defekt</i> eingestufte Fehler (siehe Abschn. 5.3.4) zurückgemeldet.
Request emission-related diagnostic trouble codes detected during current or last completed driving cycle	07 h	Wie SID 03, dabei werden aber auch als nur <i>vorläufig defekt</i> eingestufte Fehler zurückgemeldet.
Request powertrain freeze frame data	02 h	Abfrage der Umgebungsdaten für einen im Fehlerspeicher gespeicherten Fehlercode
Clear emission-related diagnostic information	04 h	Löschen des Fehlerspeichers (Fehlercodes, Umgebungsbedingungen, Status der verschiedenen Tests), kein PID.
<i>Test abgasrelevanter Komponenten</i>		
Request oxygen sensor monitoring test results	05 h	Überwachung der Lambda-Sonde
Request on-board monitoring test results for non-continuously monitored systems	06 h	Überwachung Katalysatoren, Abgasrückführung, Sonden- und Katalysatorheizung, Zünd- und Einspritzsystem
Request control of on-board system, test or component	08 h	Test der Tankentlüftung
<i>Auslesen von Steuergerätewerten</i>		
Request current powertrain diagnostic data	01 h	Abfrage von Steuergeräte-Datenwerten, die über Kennziffern (Parameter Identifier PID) ausgewählt werden.
Request vehicle information	09 h	Auslesen der Fahrzeugseriennummer (Vehicle Identification Number, 17 ASCII-Zeichen, die – außer bei CAN – auf 5 Antwortbotschaften verteilt werden, PID = 02 h) oder der Software-/Datensatzserien- oder Versionsnummer (Calibration Identification, 16 ASCII-Zeichen, PID = 04 h) bzw. der zugehörigen Prüfsummen (Calibration Verification Number, 4 Byte Hexwert, PID = 06 h). Zuvor kann der Diagnosetester mit PID = 01 h, 03 h bzw. 05 h abfragen, wie viele Antwortbotschaften er vom Steuergerät jeweils erwarten muss.
0 A-0Fh		Reserviert

Tab. 5.23 Auswahl verschiedener OBD-Messwerte über PIDs

PID	Bedeutung	Datengröße	Wertebereich (min ... max)
04 h	Motorlast	8 bit	0 ... 100 %
05 h	Kühlwassertemperatur	8 bit	– 40 ... + 215 °C
06 h ... 09 h	Kraftstoffmengenkorrektur der Einspritzventile (Ausgabe für je 2 Zylinder)	8 bit	– 100 ... + 99,2 %
0Bh	Druck im Ansaugrohr	8 bit	0 ... 255 kPa
0Ch	Motordrehzahl	16 bit	0 ... 16383,75 1/min
0Dh	Fahrgeschwindigkeit	8 bit	0 ... 255 km/h
0Eh	Zündwinkel (für Zylinder 1)	8 bit	– 64 ... + 63,5 °
0Fh	Ansauglufttemperatur	8 bit	– 40 ... + 215 °C
10 h	Luftmasse	16 bit	0 ... 655,35 g/s
11 h	Gaspedalstellung	8 bit	0 ... 100 %
14 h	Lambda-Sonden-Spannung	8 bit	0 ... 1,275 V
... 1Bh	(siehe auch PID = 24 h ... 2Bh bzw. 34 h ... 3Bh je nach Sondentyp)	(16 bit)	
2Ch	Abgasrückführrate	8 bit	0 ... 100 %
31 h	Fahrstrecke seit Löschen des Fehlerspeichers	16 bit	0 ... 65535 km
4Eh	Betriebsdauer seit Löschen des Fehlerspeichers	16 bit	0 ... 65535 min
21 h	Fahrstrecke seit Einschalten der Fehlerlampe (MIL)	16 bit	0 ... 65535 km
4Dh	Betriebsdauer seit Einschalten der Fehlerlampe	16 bit	0 ... 65535 min
...

fragen. Dazu prüft der Diagnosetester mit PID = 02 h zuerst, welchem Fehler (DTC) die gespeicherten Daten zugeordnet sind. Da das Steuergerät unter Umständen mehrere Sätze von Umgebungsbedingungen abgespeichert hat, muss der Tester bei dieser und bei den folgenden Abfragen die *Frame* Nummer angeben. Die Nummer des *OBD-Default-Frame*s ist 0. Ob weitere *Frames* gespeichert werden, ist herstellerspezifisch. Anschließend kann der Tester mit demselben Service 02 h, aber weiteren PIDs (Tab. 5.23) die eigentlichen gespeicherten Messwerte abfragen.

Alle gespeicherten Werte sind Hexadezimalwerte (mit $n = 8$ bzw. $n = 16$ bit), die mit Hilfe des in Tab. 5.23 angegebenen Wertebereichs in den tatsächlichen physikalischen Wert umgerechnet werden müssen:

$$\text{Physikalischer Wert} = \frac{\text{Hexadizimalwert}}{2^n - 1} (\text{Maximalwert} - \text{Minimalwert}) + \text{Minimalwert}$$

Beispiel Kühlwassertemperatur:

gespeicherter Hexadezimalwert 3 Ah = 58

$$\text{physikalischer Wert } \frac{58}{2^8 - 1} \cdot (215^\circ\text{C} - (-40^\circ\text{C})) + (-40^\circ\text{C}) = 18^\circ\text{C}$$

Während der Fehlersuche lassen sich dann mit dem *Service 01 h Request Current Powertrain Diagnostic Data* mit Hilfe derselben PIDs (Tab. 5.23) die jeweils aktuellen Messwerte abfragen. Dieser Dienst kann auch während der Applikationsphase eines Fahrzeugs verwendet werden, wobei durch zyklische Anfragen sogar eine Messwertaufzeichnung im Fahrbetrieb möglich ist.

Nach Beseitigung des Fehlers kann der Fehlerspeicher dann mit Hilfe des *Service 04 h Clear Emission-related Diagnostics Information* gelöscht werden.

Wie dargestellt verwenden die meisten Dienste den PID-Parameter zur Auswahl einer Unterfunktion bzw. eines Messwertes. Die zu verwendenden PID-Werte sind in der Norm festgelegt, aber nicht jedes Steuergerät muss alle PIDs unterstützen. Bei praktisch allen Diagnosiediensten (SIDs) liefert eine Abfrage mit $\text{PID} = 0\text{ h}$ eine 32 bit-Bitmaske zurück, die angibt, welche PIDs von einem Steuergerät unterstützt werden und welche nicht. Da mit dieser Bitmaske nur 32 Werte spezifiziert werden könnten, ist ein Erweiterungsmechanismus vorgesehen, mit dem über $\text{PID} = 20\text{ h}, 40\text{ h}, 60\text{ h}, \dots$ die Unterstützung weiterer PIDs abgeprüft werden kann. Bevor der Diagnosetest einen bestimmten Service weiter verwendet, fragt er daher für diesen Service mit $\text{PID} = 0\text{ h}$ zunächst ab, welche PIDs für diesen Service unterstützt werden und baut mit der Steuergeräteantwort eine interne Tabelle auf. In der Regel verwendet der Diagnosetest diese Tabelle dann dazu, dem Bediener nur diejenigen Dienste und Messwerte anzubieten, die vom Steuergerät tatsächlich unterstützt werden.

5.3.3 Abfrage der Testergebnisse für abgasrelevante Komponenten

Die gesetzlichen Vorschriften sehen vor, dass die abgasrelevanten Komponenten im Fahrbetrieb laufend überwacht werden. Der Zustand dieser Überwachungen kann mit den Services 05 h, 06 h und 08 h abgefragt werden. Im Gesetz ist dabei nicht im Detail geregelt, wie die Überwachungen durchzuführen sind, sondern lediglich, welche Abweichungen von den gesetzlichen Abgasvorschriften erkannt werden müssen. Da die Überwachungen in der Regel nur in bestimmten Motorbetriebszuständen, z. B. erst ab einer bestimmten Motortemperatur, durchgeführt werden können, speichert das Steuergerät jeweils die Ergebnisse der letzten vollständig durchgeföhrten Überwachung. ISO 15031 und die zugrunde liegenden OBD-Normen definieren dabei – unabhängig von den eigentlichen gesetzlichen Vorschriften zu den Abgasgrenzwerten und Überwachungsfunktionen – lediglich das Protokoll, mit dem die Ergebnisse abgefragt werden. Mit dem *Service 05 h Request Oxygen Sensor Monitoring Test Results* beispielsweise wird das Ergebnis der Überwachung der

Lambda-Sonden abgefragt. Die Anfrage enthält die PID (in der Norm in diesem Fall leider als Test ID bezeichnet), um das Testergebnis auszuwählen, sowie die Nummer der Sonde. Abgefragt werden können zum Beispiel die Grenzwerte (als Sensorspannungen) für Magerbetrieb (PID = 03 h) und Fettbetrieb (PID = 04 h), die größte und die kleinste gemessene Sensorspannung (PID = 07 h bzw. 08 h), die Schaltzeiten für das Durchlaufen des Bereichs zwischen den Schwellwerten sowie die Periodendauer eines Lambda-Regelzyklusses (PID = 0Ah) und herstellerspezifisch definierte Werte. Wo und wie viele Lambda-Sonden eingebaut sind, d. h. die Sondernummer, erfährt der Diagnosetester zuvor über die Anfrage SID = 01 h, PID = 13 h bzw. PID = 1Dh.

Mit dem *Service 06 h Request On-Board Monitoring Test Results for Non-Continuously Monitored Systems* können zusätzlich zu den Lambda-Sonden auch die Überwachungen für andere Komponenten wie die Sondenheizung, die Katalysatoren und die Katalysatorenheizung, die Abgasrückführung, die Tankentlüftung, das Zündsystem und die Einspritzanlage für die einzelnen Zylinder abgefragt werden. An dieser Stelle unterscheidet die Norm abhängig vom zugrunde liegenden Busprotokoll. Bei K-Line und J1850-Bussen sind die PIDs und die Skalierung der Messwerte aus historischen Gründen weitgehend herstellerspezifisch frei wählbar, während bei CAN beides präzise vorgegeben wird.

Mit dem *Service 08 h Request Control of On-Board System, Test or Component* soll der Diagnosetester einen Test im Steuergerät starten und stoppen können. Derzeit ist dies nur für den Test der Tankentlüftung vorgesehen.

Welche Tests von den Diensten 05 h, 06 h und 08 h jeweils unterstützt werden und ob gültige Testergebnisse vorliegen, d. h. der Motor seit dem Löschen des Fehlerspeichers lange genug in einem Betriebszustand betrieben wurde, in dem dieser Test vollständig durchgeführt werden konnte, kann der Tester der Abfrage SID = 01 h, PID = 01 h entnehmen (siehe oben). Die Abfrage, ob die Tests vollständig durchgeführt wurden sowie die Tests selbst werden häufig als *Readiness Tests* bezeichnet.

5.3.4 OBD-Fehlercodes

Im Steuergerät werden die Fehlercodes als 16 bit Hexadezimalzahl gespeichert, die im Diagnosetester oder in der Dokumentation in eine fünfstellige alphanumerische Darstellung umgewandelt werden. Der Wert 0000 h wird als *kein Fehler* interpretiert, ansonsten gilt folgende Zuordnung (vgl. Abb. 5.10):

Über den Buchstaben wird zwischen Fehlern in folgenden Bereichen unterschieden:

Antriebsstrang	Powertrain	P
Fahrwerk	Chassis	C
Karosseriebereich	Body	B
Datennetz, Bussystem	Network	U

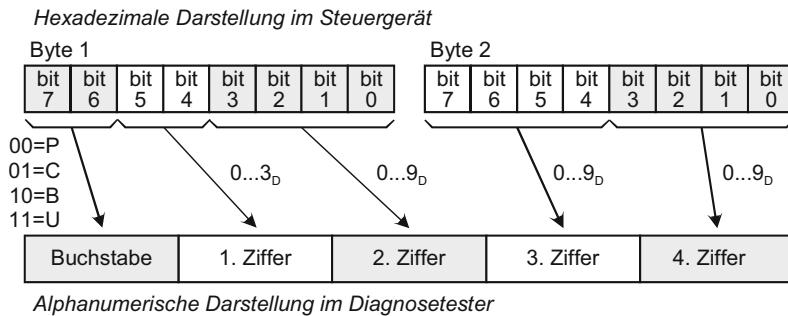


Abb. 5.10 OBD Fehlercode-Format

Die erste Ziffer zeigt an, wer den Fehlercode festgelegt hat:

- 0 Fehlercodes nach ISO 15031-6 bzw. SAE J2012
- 1, 2 Herstellerdefinierte Fehlercodes
- 3 Reserviert

Bei den von ISO/SAE festgelegten Fehlercodes für den Antriebsstrang unterscheidet die 2. Ziffer, welche Komponente fehlerhaft ist, z. B.

- P01 ..., P02 ... Einspritzsystem
- P03 ... Zündung
- P04 ... Abgasrückführung und andere Hilfskomponenten
- P05 ... Fahrgeschwindigkeits- und Leerlaufregelung
- P06 ... Steuergeräteinterne Fehler
- P07 ..., P08 ... Getriebe

Über die weiteren beiden Ziffern werden die Komponente und die Fehlerart näher klassifiziert. So empfiehlt die Norm, die folgenden Fehlerarten zu unterscheiden:

- Allgemeine Fehlfunktion
- Unzulässiger Wert oder schlechte Leistung
- Eingang dauernd ein
- Eingang dauernd aus
- Wackelkontakt
- ...

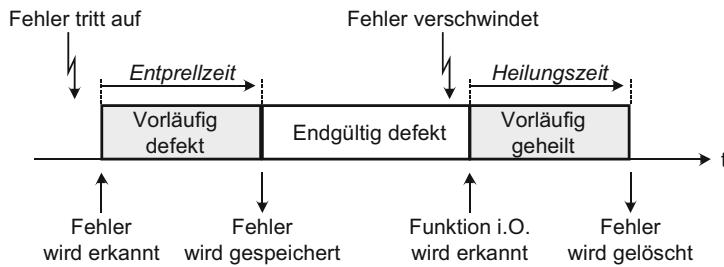


Abb. 5.11 Fehlerentprellung und Fehlerheilung

Beispiele für Fehlercodes:

P0100	Luftmassenmesser	Allgemeiner Fehler
P0101	Luftmassenmesser	Unzulässiger Wert
P0104	Luftmassenmesser	Wackelkontakt
P0130	Erste Lambda-Sonde	Allgemeiner Fehler
P0201	Einspritzventil Zylinder 1	Allgemeiner Fehler
P0301	Zündung Zylinder 1	Fehlzündung erkannt
P0400	Abgasrückführung	Allgemeiner Fehler
P0506	Leerlaufdrehzahl	Wert zu niedrig
...

Fehler werden in der Regel *entprellt*, bevor sie gespeichert werden, und können auch wieder *geheilt* werden (Abb. 5.11). Ein Fehler muss für eine bestimmte Mindestzeit (*Entprellzeit*) erkannt worden sein, bevor er als *endgültig defekt* eingestuft und in den permanenten Fehlerspeicher des Steuergerätes, in der Regel ein EEPROM, eingetragen wird. Als *Wackelkontakt* wird ein Fehler erkannt, wenn er während der Entprellzeit zwar nicht dauerhaft, aber doch in kurzen Zeitabständen wiederholt erkannt wird. Verschwindet der Fehler wieder, so muss die zugehörige Funktion für eine weitere Mindestzeit (*Heilungszeit*) als einwandfrei erkannt werden, bevor der Fehlereintrag endgültig gelöscht wird.

Während der Entprell- und Heilungszeiten kann der Status des im RAM des Steuergerätes vorläufig gespeicherten Fehlers von den im EEPROM gespeicherten Werten abweichen. Die durch die OBD-Dienste gelesenen Fehlercodes und zugehörigen Umgebungsbedingungen beziehen sich in der Regel ausschließlich auf die im permanenten Fehlerspeicher gespeicherten Fehler. Wahlweise können aber auch die als *vorläufig defekt* eingestuften Fehler gelesen werden.

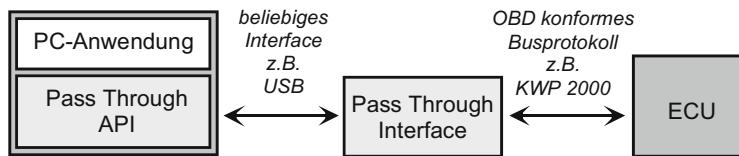


Abb. 5.12 Pass-Through-Programmierung nach SAE J2534/ISO 23248

5.3.5 Data Link Security

Der OBD-Standard definiert denselben *Seed and Key*-Mechanismus zur Zugangskontrolle wie die KWP 2000-Protokolle (vgl. Abschn. 5.1.2). Wie dort wird der Algorithmus zur Berechnung von *Seed* und *Key* und deren Länge selbst nicht spezifiziert und auch nicht vorgegeben, welche Dienste zu schützen sind.

5.3.6 Pass-Through-Programmierung

Ähnlich wie bei der Diagnose der abgasrelevanten Fahrzeugkomponenten hat der amerikanische Gesetzgeber die Notwendigkeit erkannt, die Flash-Programmierung der Steuergeräte zu normieren. Dadurch soll gewährleistet werden, dass für Werkstätten kostengünstige Werkzeuge bereitgestellt werden, mit denen die Steuergeräte-Software auf einen neuen Stand gebracht werden kann. Die Standardisierung nach SAE J2534, die mittlerweile innerhalb der ASAM-Initiative als ISO 22900 (siehe Abschn. 6.8) in angepasster Form auch in Europa übernommen werden soll, umfasst sowohl die Hardware-Schnittstelle zwischen einem handelsüblichen PC und der Diagnoseschnittstelle des Fahrzeugs als auch die Software-Programmierschnittstelle für eine Windows-basierte Anwendungssoftware (Abb. 5.12). Als Diagnoseprotokolle muss die Hardwareschnittstelle die OBD-üblichen Protokolle ISO 9141, KWP 2000 über K-Line und CAN sowie beide Varianten von SAE J1850 enthalten. Das im Nutzfahrzeubereich übliche SAE J1939 war zunächst nicht verbindlich, soll zukünftig aber ebenfalls unterstützt werden. Das neuere UDS-Protokoll ist, ebenso wie die Bussysteme FlexRay, LIN und MOST, die im Diagnosebereich allgemein noch vernachlässigt werden, derzeit noch nicht enthalten.

Die Software-Programmierschnittstelle enthält nicht nur Funktionen zum Lesen und Programmieren des Steuergeräte-Flash-Speichers über die Diagnoseprotokolle, sondern erlaubt über Funktionen wie *PassThruReadMsgs()*, *PassThruWriteMsgs()* oder *PassThruStart/StopPeriodicMsg()* und programmierbare Botschaftsfilter das Versenden und Empfangen beliebiger Botschaften, so dass über die Programmierschnittstelle praktisch Zugriff auf das gesamte Diagnoseprotokoll besteht.

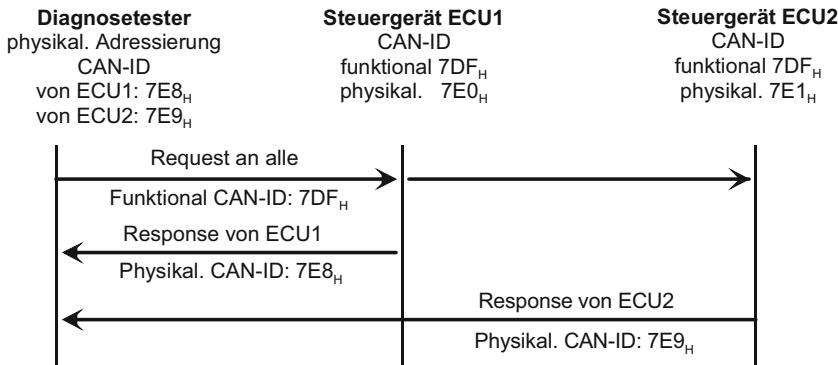


Abb. 5.13 Steuergerätedressierung bei der ODB-Diagnose

5.3.7 Beispiel

Im Folgenden sollen einige Botschaftssequenzen für typische OBD-Diagnoseanfragen dargestellt werden. Dabei wird angenommen, dass ein CAN-Bussystem mit 11 bit *Identifizieren* und dem Transportprotokoll nach ISO 15765-2 eingesetzt wird. Der Diagnosetester sendet seine Anfragen (*Request*) gemäß ISO 15765-4 funktional adressiert mit dem *CAN Identifier* 7DF_H (Abb. 5.13). Die Antwort (*Response*) des ersten OBD-relevanten Steuergerätes, in der Regel das Motorsteuergerät, erwartet der Tester mit dem *CAN Identifier* 7E8_H (physikalische Adressierung). Für Antworten des zweiten Steuergerätes, z. B. dem Getriebesteuergerät, ist der *CAN Identifier* 7E9_H vorgesehen. Nach ISO 15765-4 darf die Bitrate 250 kbit/s oder 500 kbit/s betragen. Der Tester muss die Bitrate sowie die Verwendung von 11bit oder 29bit *CAN Identifier* nach einem in der Norm vorgegebenen Algorithmus automatisch erkennen.

Im ersten Szenario fragt der Diagnosetester nach der Motordrehzahl und Fahrgeschwindigkeit. Das Motorsteuergerät liefert die Drehzahl, das Getriebesteuergerät die Geschwindigkeit. Sowohl die *Request* als auch die *Response* Botschaften enthalten nur wenige Bytes, so dass sie als *Single Frame* Botschaften der ISO 15765-2 Transportschicht übertragen werden können:

CAN Identifier Adressierung	ISO TP Protocol Control Information PCI	OBD Service Identifier SID	OBD PID und Daten
<i>Request Diagnosetester → alle Steuergeräte</i>			
7DF _H funktional	03 h Single Frame 3 Datenbytes	01 h SID Messdaten lesen	0Ch PID Motordrehzahl 0Dh PID Fahrgeschwindigkeit

CAN Identifier Adressierung	ISO TP Protocol Control Information PCI	OBD Service Identifier SID	OBD PID und Daten
<i>Response Diagnosetester ← Motorsteuergerät</i>			
7E8 h physikalisch	04 h Single Frame 4 Datenbytes	41 h Positive Response SID 01 h	0Ch PID Motordrehzahl 0E 74 h Messwert 925 min ⁻¹
<i>Response Diagnosetester ← Getriebesteuergerät</i>			
7E9 h physikalisch	03 h Single Frame 3 Datenbytes	41 h Positive Response SID 01 h	0Dh PID Fahrge- schwindigkeit 32 h Messwert 32 h = 50 km/h

Im zweiten Szenario fragt der Diagnosetester nach der Fahrgestellnummer. Das Motorsteuergerät antwortet mit einer *First Frame* Botschaft und zwei weiteren *Consecutive Frame* Botschaften, nachdem es die *Flow Control* Botschaft des Diagnosetesters erhalten hat. Während die *Requests* des Testers grundsätzlich funktional adressiert werden, muss die *Flow Control* Botschaft nach ISO 15765-4 die physikalische Adresse verwenden.

Im Beispiel hat das Fahrzeug genau eine Fahrgestellnummer (erstes Antwortbyte 01 h) und besteht aus 17 ASCII-Zeichen, beginnend ab 57 h = „W“. Das Getriebesteuergerät kennt diese Information nicht und antwortet daher gar nicht:

CAN Identifier/ Adressierung	ISO TP Protocol Control Information PCI	OBD Service Identifier SID	OBD PID und Daten
<i>Request Diagnosetester → alle Steuergeräte</i>			
7DFh funktional	02 h Single Frame 2 Datenbytes	09 h SID Fahrzeug- daten lesen	02 h PID Fahrgestell- nummer
<i>Response Diagnosetester ← Motorsteuergerät</i>			
7E8 h physikalisch	10 14 h First Frame 14 h = 20 Daten- bytes folgen	49 h Positive Response SID 09 h	02 h PID Fahrgestell- nummer 01 57 41 55 h 1 Datensatz „W A U“
<i>Flusssteuerung Diagnosetester → Motorsteuergerät</i>			
7E0 h physikalisch	30 00 00 h Flow Control Frame		
<i>Response Diagnosetester ← Motorsteuergerät</i>			
7E8 h physikalisch	21 h Consecutive Frame Sequenz- nummer 1		5 A 5 A 5 A 38 45 37 37 h „Z Z 8 E 7 7“

CAN Identifier/ Adressierung	ISO TP Protocol Control Information PCI	OBD Service Identifier SID	OBD PID und Daten
<i>Response Diagnosetester ← Motorsteuergerät</i>			
7E8 h physisch	21 h Consecutive Frame Sequenz- nummer 2		41 30 xx xx xx 32 h „A 0 2“

In Wirklichkeit senden Tester und Steuergeräte bei OBD gemäß ISO 15765-4 immer vollständige, 8 Byte lange CAN-Botschaften, wobei die restlichen Bytes einer Botschaft mit beliebigen, vom Empfänger ignorierten Werten gefüllt werden.

5.4 Weiterentwicklung der Diagnose

Mittelfristig geht man davon aus, dass die externe Fahrzeugschnittstelle auf Ethernet/IP umgestellt wird (Abschn. 4.6). Diese im Bürobereich übliche Technik bietet zum einen eine sehr hohe, ständig steigende Bandbreite, zum anderen ist sie sehr kostengünstig. Die Echtzeitfähigkeit und Robustheit bei EMV und sonstigen Umwelteinflüssen, die für den Einsatz in der On-Board-Kommunikation (Abschn. 3.5) noch nicht vollständig gelöst sind, gelten im Off-Board-Bereich als unkritisch (Abb. 5.14).

Vor diesem Hintergrund wurden und werden die Diagnosenormen überarbeitet, um eine saubere Trennung der verschiedenen Schichten des ISO/OSI-Modells in den Normschriften zu erreichen (Abb. 5.15). Die Anwendungsebene der Diagnose wird durch UDS nach ISO 14229 abgedeckt. Auch die Abgasdiagnose OBD wird in ISO 27145 als World Wide-Harmonized OBD auf eine Auswahl von UDS-Botschaften umgestellt. In den überarbeiteten Normen werden die Diagnosedienste und die Behandlung der Diagnosesitzungen in den Standards besser getrennt. Für die darunter liegenden Bussysteme werden die Transport- und Netzwerkschicht einheitlich beschrieben und vom Data Link Layer deutlicher abgegrenzt.

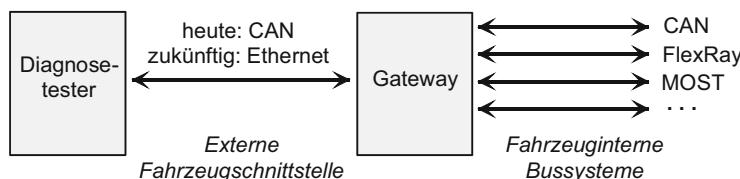


Abb. 5.14 Externe und interne Datennetze

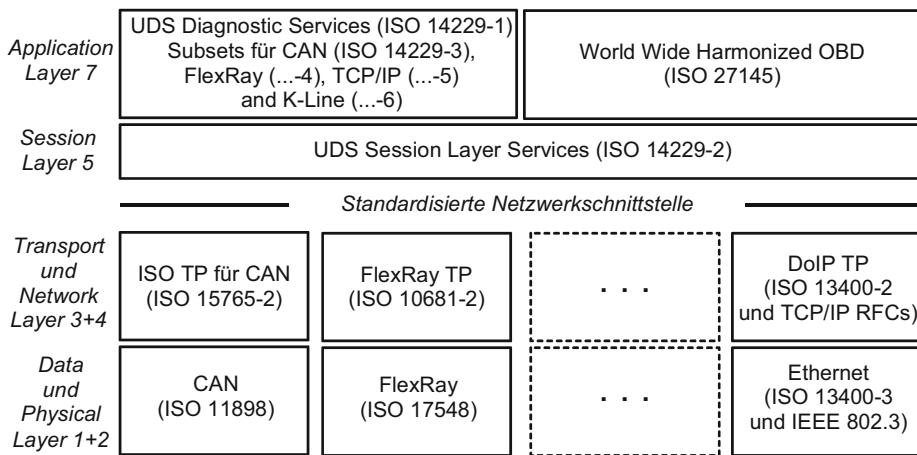


Abb. 5.15 Zukünftige Struktur des Diagnose-Protokollstapels

5.4.1 World-Wide Harmonized On-Board Diagnose nach ISO 27145

Mit Einführung der WWH-OBD im NKW- und später auch im PKW-Bereich soll versucht werden, den Zusatzaufwand für das OBD-spezifische Diagnoseprotokoll zu eliminieren. Wie bei der Vorstellung von OBD in Abschn. 5.3 klar wurde, sind die Kommunikationsfunktionen der Abgasdiagnose logisch eine kleine Untermenge der allgemeinen UDS-Werkstattdiagnose, verwenden aber bisher eigene Diagnosedienste und müssen daher separat implementiert werden. ISO 27145 ersetzt nun sämtliche OBD-spezifischen Dienste durch vorhandene UDS-Dienste (Tab. 5.24).

Tab. 5.24 WWH-OBD-Diagnosedienste

UDS-SID für WWH-OBD	Diagnosedienst	Ersetzt OBD-SID aus Tab. 5.22
22 h	Read Data By Identifier	01 h, 09 h
19 h	Read DTC Information Unterfunktion SFID = 04 h Report DTC Snapshot Record By DTC Number Unterfunktion SIFD = 06 h Report DTC Extended Data Record By DTC Number Unterfunktion SFID = 42 h Report WWH-OBD DTC By Mask Record	02 h, 03 h, 07 h
14 h	Clear Diagnostic Information Unterscheidung der OBD-relevanten Daten durch den Parameter groupOfDTC = FFFF33 h	04 h
31 h	Routine Control Unterfunktion 01 h – Start Routine	05 h, 06 h, 08 h

Tab. 5.25 Auswahl verschiedener OBD-Messwerte über DIDs

UDS-DID für WWH-OBD	Bedeutung	Ersetzt OBD-PID aus Tab. 5.23
F802 h	Fahrzeugseriennummer (Vehicle Identification Number VIN, OBD Dienst 09 h)	02 h (SID 09 h)
F804 h	Applikationsdatensatzkennnummer (Calibration Verification Number, OBD Dienst 09 h)	06 h (SID 09 h)
F806 h		
F404 h	Motorlast	04 h
F467 h	Kühlwassertemperatur	05 h
F470 h	Druck im Ansaugrohr	0Bh
F40Ch	Motordrehzahl	0Ch
F40Dh	Fahrgeschwindigkeit	0Dh
F468 h	Ansauglufttemperatur	0Fh
F466 h	Luftmasse	10 h
F44Ah	Gaspedalstellung	11 h
F491 h	Status der Fehleranzeigelampe MIL	
F430 h	Betriebsdauer seit Löschen des Fehlerspeichers	4Eh
F490 h	Betriebsdauer seit Einschalten der Fehlerlampe	4Dh
...

Im Vergleich zur ursprünglichen UDS-Norm heißen die Auswahlwerte für Unterfunktionen, die in Abschn. 5.2 als LEV bezeichnet wurden, *Sub Function Identifier SFID*. Die bekannten OBD *Parameter Identifier PID* zum Auslesen von Steuergerätedaten werden jetzt *Data Identifier DID* genannt und sind nun 16 bit breit (Tab. 5.25).

Die bekannten OBD Fehlercodes (*Diagnostic Trouble Code DTC*) aus ISO 15031-6/SAE J2012 (Abschn. 5.3.4) gelten weiterhin. Der Fehlercode besteht aus drei Byte. Die ersten beiden Bytes weisen auf die fehlerhafte Komponente, wie in Abb. 5.10 bereits beschrieben. Das dritte Byte, das *Failure Type Byte FTB*, grenzt die Art des Fehlers weiter ein (Tab. 5.26). Alternativ zu diesen im PKW-üblichen Fehlercodes sind auch die im NKW-Bereich eingeführten Fehlercodes (*Failure Mode Identifier FMI*) aus SAE J1939/73 zulässig.

Als physikalische Fahrzeugschnittstelle schreibt WWH-OBD weiterhin CAN nach ISO 11898 mit dem Transportprotokoll nach ISO 15765-2 vor. Mittelfristig soll aber auch bei OBD Ethernet/IP nach ISO 13400 als Diagnoseschnittstelle verwendet werden. Innerhalb des Fahrzeugs selbst können die Diagnosebotschaften natürlich auch über FlexRay oder LIN übertragen werden.

Tab. 5.26 Beispiele für *Failure Type Bytes FTB*

FTB	Bedeutung
11 h	Kurzschluss nach Masse
12 h	Kurzschluss zur Batteriespannung
13 h	Verbindung unterbrochen
16 h	Spannung zu klein
17 h	Spannung zu groß
1Fh	Wackelkontakt
36 h	Signalfrequenz zu klein
37 h	Signalfrequenz zu groß
64 h	Signal nicht plausibel
...	...

5.5 Normen und Standards zu Kapitel 5

KWP 2000	ISO 14230, Road vehicles diagnostic systems – Keyword Protocol 2000
K-Line	Part 1: Physical Layer, 1999, www.iso.org Part 2: Data Link Layer, 1999, www.iso.org Part 3: Application Layer, 1999, www.iso.org Part 4: Requirements for emissions-related systems, 2000, www.iso.org Überarbeitete Versionen: ISO 14230: Road vehicles – Diagnostic communication over K-Line (DoK-Line) Part 1: Physical Layer, 2012, www.iso.org Part 2: Data Link Layer, 2013, www.iso.org
KWP 2000	ISO/DIS 15765-3: Road vehicles – Diagnostics on Controller Area Networks
CAN	(CAN) – Part 3: Application Layer Services, 1999, www.iso.org . Als KWP 2000 on CAN bekanntgewordener Normentwurf, offiziell zurückgezogen und durch UDS on CAN ersetzt.
UDS	ISO 14229 Road vehicles – Unified diagnostic systems (UDS) Part 1: Specification and requirements, 2013, www.iso.org Part 2: Session layer services, 2013, www.iso.org Part 3: Unified diagnostic services on CAN (UDS on CAN), 2012, www.iso.org Part 4: Unified diagnostic services on FlexRay (UDS on FR), 2012 Part 6: Unified diagnostic services on K-Line implementation (UDS on K-Line), 2013, www.iso.org
UDS	ISO 15765 Road vehicles – Diagnostics communication over Controller Area Networks (DoCAN), www.iso.org
CAN	Part 1: General information and use case definition, 2011 Part 2: Transport protocol and network layer services, 2011 Part 3: Implementation of unified diagnostic services (UDS on CAN), 2004, Neufassung siehe ISO 14229-3 Part 4: Requirements for emissions-related systems, 2011 Part 4/AMD1: Amendment 1, 2013

UDS	ISO 10681 Road vehicles – Communication on FlexRay, www.iso.org
FlexRay	Part 1: General Information and use case definition, 2010
	Part 2: Communication layer services, 2010
OBD	<p>ISO 15031 Road vehicles – Communication between vehicle and external equipment for emissions-related diagnostics, www.iso.org</p> <p>Part 1: General information and use case definition, 2010</p> <p>Part 2: Guidance on terms, definitions, abbreviations and acronyms, 2010, (entspricht SAE J1930, www.sae.org)</p> <p>Part 3: Diagnostic connector and related electrical circuit, 2004, (entspricht SAE J1962, www.sae.org)</p> <p>Part 4: External test equipment, 2005, (entspricht SAE J1978)</p> <p>Part 5: Emissions-related diagnostic services, 2011, (entspricht SAE J1979, www.sae.org)</p> <p>Part 6: Diagnostic trouble code definitions, 2010, (entspricht SAE J2012, www.sae.org)</p> <p>Part 7: Data link security, 2011, (entspricht SAE J2186)</p> <p>ISO 23248 Road vehicles – Pass-through programming, 2004 und 2006, www.iso.org (entspricht SAE J2534, www.sae.org), inzwischen: ISO 22900</p> <p>SAE J2012 Diagnostic Trouble Code Definitions, 2013, www.sae.org</p>
WWH-OBD	<p>ISO 27145 Road vehicles – Implementation of World Wide Harmonized On-Board Diagnostics (WWH-OBD) communication requirements, www.iso.org</p> <p>Part 1: General information and use case definition. 2012</p> <p>Part 2: Common data dictionary. 2012</p> <p>Part 3: Common message dictionary. 2012</p> <p>Part 4: Connection between vehicle and test equipment. 2012</p> <p>Part 5: Conformance test. 2009</p> <p>Part 6: External test equipment. 2009</p>
DoIP	ISO 13400 Road vehicles Diagnostic communication over Internet Protocol
Ethernet	siehe Kap. 4

Literatur

- [1] C. Marscholik, P. Subke: Datenkommunikation im Automobil. Hüthig Verlag, 1. Auflage, 2007

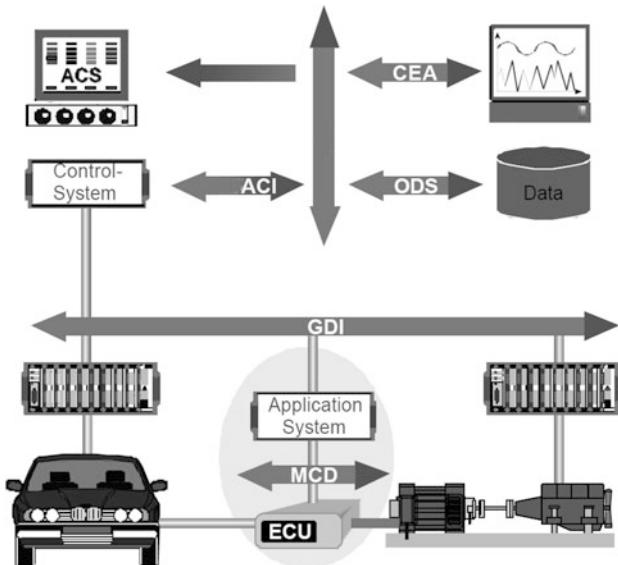
6.1 Einführung

ASAM, die Association for Standardization of Automation and Measuring Systems (ursprünglich ASAP Arbeitskreis zur Standardisierung von Applikationssystemen), ist eine Initiative der europäischen Automobilhersteller und ihrer Zulieferer, um die Applikation und den Test elektronischer Systeme in der Entwicklungs- und Fertigungsphase zu vereinfachen. In der Applikationsphase werden bei der Erprobung des Fahrzeugs und seiner Komponenten auf verschiedenen Prüfständen sowie im Fahrversuch Messwerte aus den Steuergeräten erfasst (*Measure*) und die Steuergeräteinternen Parameter angepasst (*Calibrate*). In der Fertigungsphase wird der korrekte Einbau und die korrekte Funktion der Komponenten auf Fertigungsprüfständen erprobt und gegebenenfalls Feinjustage der Steuergerätedaten vorgenommen. In beiden Fällen werden Messdaten gesammelt (*Measure*), Diagnoseinformationen ausgewertet (*Diagnose*) und Steuergeräteparameter verstellt (*Calibrate*). Die Steuerung der Abläufe sowie die Datenauswertung und Datenhaltung erfolgt dabei durch eine Reihe von vernetzten Rechnersystemen, die unter anderem mit den Steuergeräten im Fahrzeug kommunizieren müssen. Innerhalb ASAM wird versucht, die Schnittstellen zwischen den einzelnen Prüfständen sowie die Datenaustauschformate zu standardisieren.

ASAM ist unterteilt in eine Reihe von Standards für die unterschiedlichen Ebenen des Gesamtsystems (Abb. 6.1) von den Steuergeräten und deren Applikations- und Diagnoseschnittstellen (MCD, neuerdings auch als Automotive Electronics AE bezeichnet), den Schnittstellen zwischen den Prüfstandskomponenten (GDI) bis zur Leitebene der übergeordneten Prüfstandssteuerung (ACI), Datenhaltung (ODS) und Datenanalyse (CEA). Im Folgenden soll lediglich die AE MCD-Ebene aus Sicht der Steuergeräte näher betrachtet werden.

In der Begriffswelt der Informatik würde man ASAM-MCD als *Middleware* bezeichnen, d. h. als eine Zwischenschicht, die eine Kopplung zwischen den Steuergeräten auf der untersten Ebene und den übergeordneten Test- und Diagnoseanwendungen herstellt

Abb. 6.1 Überblick über ASAM (Quelle: ASAM-Dokument: Introduction ASAM-MCD); MCD: Measure, Calibrate, Diagnose (AE ... Automotive Electronics); GDI: Generic Device Interface; ODS: Open Data Service; ACI: Automatic Calibration Interface; CEA: Components for Evaluation and Analysis; ECU: Electronic Control Unit



(Abb. 6.2). Die Aufgabe der Zwischenschicht besteht darin, nach oben eine einheitliche, abstrahierte Schnittstelle zur Verfügung zu stellen und Implementierungsdetails der Steuengeräte zu kapseln. Die ASAM-MCD Middleware-Software läuft dabei auf dem Testsystem oder Diagnosetester, d. h. einem PC-artigen Rechnersystem mit den üblichen Schnittstellen zu anderen Rechnersystemen (Ethernet, USB usw.) und kommuniziert mit den Steuengeräten über die Kfz-üblichen Bussysteme (K-Line, CAN usw.). Häufig findet man aber auch für Applikationsaufgaben modifizierte Steuengeräte, bei denen der Steuengeräte-Flash-ROM-Speicher erweitert und durch RAM-Speicher ergänzt wurde (*Emulations-Tastkopf ETK*), um die Programme und Daten leichter ändern und schnell zwischen unterschiedlichen Werten umschalten zu können. Eine wesentliche Rolle bei ASAM MCD spielt außerdem die einheitliche, herstellerunabhängige Beschreibung der Steuengeräteeigenschaften in einer Datenbank mit standardisiertem Datenaustauschformat. Die Datenbank enthält Informationen über die im Steuengerät verfügbaren Funktionen und Variablen, z. B. deren Speicheradressen, die Parameter der Busschnittstelle sowie zur Umrechnung der steuengeräteinternen Daten in physikalische Werte. Insgesamt definiert ASAM-MCD drei Schnittstellen

- ASAM 3: Schnittstelle zu den übergeordneten Test- und Diagnoseanwendungen,
- ASAM 2: Schnittstelle zur Steuengerätedatenbank,
- ASAM 1: Schnittstelle zu den Steuengeräten.

Leider widerspricht die real existierende Situation in der MCD-Welt der sauberen Grundstruktur nach Abb. 6.2 und spiegelt sehr stark die historische Entwicklung sowohl in der Kfz-Elektronik als auch in der Informationstechnik seit Mitte der 90er Jahre wieder.

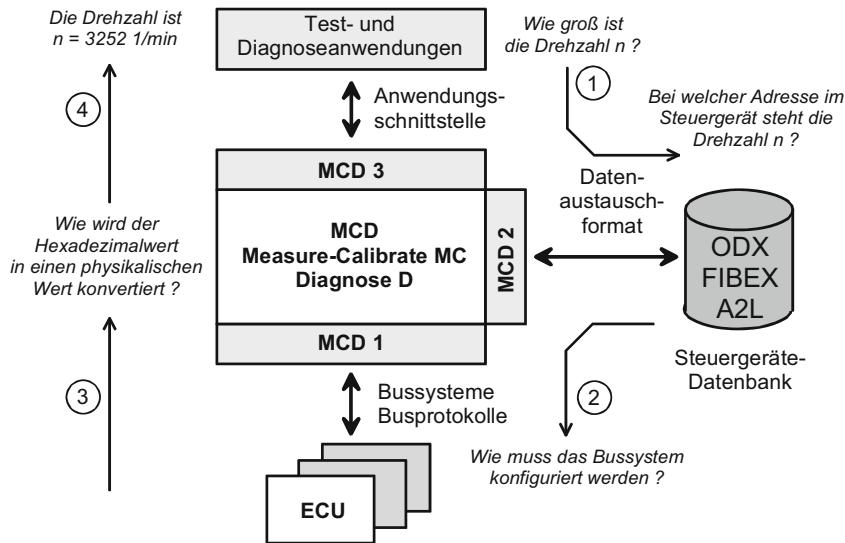


Abb. 6.2 ASAM AE MCD-Schnittstellen

Die Initiatoren der ASAM-Vorgängervereinigung ASAP beschäftigten sich zunächst lediglich mit Applikationssystemen und damit dem Bereich Messen und Kalibrieren (MC). Der Bereich Diagnose (D) war ausgeklammert. Die Diagnoseschnittstellen wurden parallel zur Arbeit von ASAP in ISO-Gremien normiert. Als ab Anfang 2000 versucht wurde, die Diagnoseaspekte mit zu berücksichtigen, musste auf die auf beiden Seiten existierenden, nur teilweise kompatiblen Lösungen Rücksicht genommen werden. Daher existiert heute für jede der drei Schnittstellen ASAM 1, 2, 3 jeweils eine -MC und eine -D Ausführung, die zwar so konzipiert sind, dass sie koexistieren können, die aber noch keineswegs vereinheitlicht sind.

Gleichzeitig spiegelt sich in den Standards die Weiterentwicklung in der Informations-technik wieder (Tab. 6.1). Das Busprotokoll zu den Steuergeräten (ASAM 1MC) für den Bereich Messen-Kalibrieren war ursprünglich das CAN Calibration Protokoll CCP. Dieses wird allmählich durch das XCP-Protokoll abgelöst, das neben CAN und FlexRay auch andere Netzwerkvarianten (USB, Ethernet mit TCP/IP und UDP/IP) berücksichtigt. Für Diagnoseaufgaben dagegen sieht ASAM keine eigenen Standards vor, sondern verweist auf die etablierten ISO-Normen für KWP 2000 bzw. UDS.

Als Datenaustauschformat (ASAM 2) wurde ursprünglich ein proprietäres Textformat (ASAM MCD 2MC) definiert, auch als ASAM2 Meta Language AML oder in neueren Dokumenten als ASAP Classic bezeichnet. Dieses könnte zukünftig durch ein XML-basiertes Format ersetzt werden. Bereits XML-basiert sind das Format für die Beschreibung der Diagnosedaten (ASAM MCD 2D), als ODX (Open Data Exchange)-Format bekannt, das OTX-Format zur Beschreibung von Diagnoseabläufe sowie der als FIBEX bezeichnete Standard für die Beschreibung der Steuergeräte-Onboard-Kommunikation.

Tab. 6.1 ASAM-Varianten innerhalb des AE MCD Standards

Busprotokolle für Messen-Kalibrieren (MC)	
ASAM AE MCD 1MC CCP	CAN Calibration Protocol CCP
ASAM AE MCD 1MC XCP	Universal Measurement and Calibration Protocol XCP
Busprotokolle für Diagnose (D)	
Modular Vehicle Communication Interface (ISO 22900-1, -2 MVCI, D-PDU)	Hardware- und Softwareinterface zum Bussystem
KWP 2000 (K-Line oder CAN)	Verweis auf die ISO-Normen, keine Normierung innerhalb ASAM
UDS (CAN, FlexRay, Ethernet)	
Datenbeschreibungsformate	
ASAM AE MCD 2MC ASAP2	Proprietäres textbasiertes Datenformat für Messen-Kalibrieren (ASAM <i>Classic</i> , ASAM2 Meta Language AML)
ASAM AE MCD 2MC CDF	XML-basiertes Datenformat für Messen-Kalibrieren <i>CDF Calibration Data Format</i> , soll mittelfristig AML ablösen/ergänzen
ASAM AE MCD 2D ODX ISO 22901-1, -2	XML-basiertes Datenformat für Diagnosedaten <i>ODX Open Data Exchange</i>
ASAM AE MCD 2NET FIBEX	XML-basiertes Datenformat für die Steuergeräte-Onboard-Kommunikation <i>FIBEX Field Bus Exchange Format</i>
Anwendungsschnittstelle	
ASAM AE MCD 3 (ISO 22900-3, nur D-Server)	Einheitliche Schnittstelle für Mess-, Kalibrier- und Diagnoseanwendungen
ISO 13209 OTX	Open Test Sequence Exchange

Inzwischen teilweise noch als ASAP bezeichnete Dokumente nach ASAM überführt und einige ASAM-Vorschläge als ISO-Standards genormt. So ist ODX mittlerweile als ISO 22901 standardisiert. Die ASAM MCD 3D-Schnittstelle inklusive der Hard- und Software-Schnittstelle zum Diagnose-Bussystem, dort als *MVCI Modular Vehicle Communication Interface* und *D-PDU Diagnostic Protokoll Data Unit Interface* bezeichnet, wurde als ISO 22900 genormt.

6.2 Busprotokolle für Aufgaben in der Applikation (ASAM AE MCD 1MC)

Die für Mess- und Kalibrieraufgaben verwendete unterste ASAM-Schicht wird nochmals in zwei Teile untergliedert (Abb. 6.3). Der Teil 1a definiert die Busprotokolle CCP und XCP, der Teil 1b die funktionale Schnittstelle (Interface) zur übergeordneten Schicht. Für beide Teile sind Konfigurationsdateien notwendig.

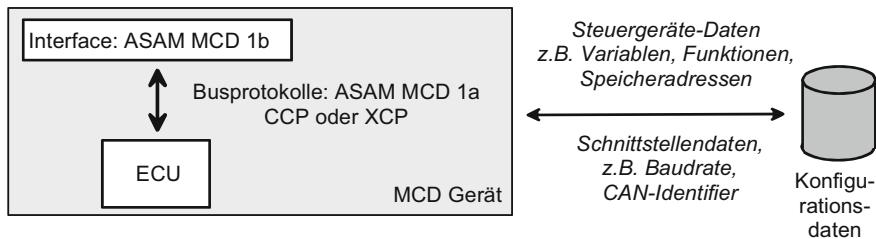
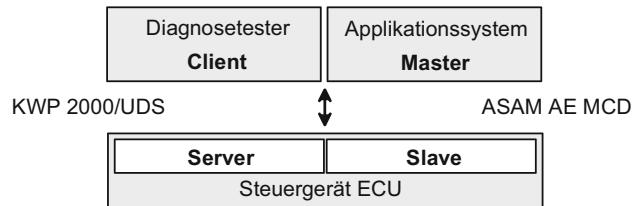


Abb. 6.3 ASAM-Schnittstelle zu den Steuergeräten für MC-Anwendungen

Das *Universal Measurement and Calibration-Busprotokoll XCP* basiert auf dem älteren *CAN Calibration Protocol CCP*, erweitert und modifiziert dieses allerdings so stark, dass es zu diesem nicht aufwärts kompatibel ist. Weiterhin erlaubt XCP außer CAN auch die Verwendung anderer Bussysteme von SPI (Serial Peripheral Interface, wird Steuergeräte-intern verwendet) bis zu Ethernet und USB. Diese Bussysteme werden heute zwar nicht von den eigentlichen Steuergeräten eingesetzt, in der Entwicklungsphase werden aber häufig modifizierte Geräte mit so genannten Applikationsadapters verwendet. Die Applikationsadapter ersetzen den ROM-Speicher der Steuergeräte durch RAM-Speicher, um die Programme und Daten schneller ändern zu können, oder machen die Mikroprozessor-internen Register- und Speicherbereiche durch Emulator- oder Debugschnittstellen zugänglich. Um die übliche In-Vehicle-Kommunikation der Steuergeräte nicht zu stören, werden die Applikationsadapter in der Regel über ein zusätzliches unabhängiges Bussystem angebunden, wobei dann auch Ethernet, USB oder Firewire zum Einsatz kommen können. Aufgrund der historischen Ableitung aus CCP und weil CAN bezüglich der möglichen Botschaftsgröße die stärkste Beschränkung aufweist, orientiert sich das XCP-Protokoll nahezu durchgängig an CAN. Die anderen Busvarianten stellen lediglich „Tunnel“ dar, die die CAN-typischen, max. 8 Byte großen Nutzdatenblöcke über ein anderes Netzwerk durchleiten.

CCP und XCP verwenden eine verbindungsorientierte, logische Punkt-zu-Punkt-Verbindung zwischen dem ASAM-MCD-Applikationssystem und dem Steuergerät, wobei das Applikationssystem gleichzeitig mehrere Verbindungen zu unterschiedlichen Steuergeräten bedienen kann, während ein einzelnes Steuergerät nur genau eine Verbindung unterstützt. Das Konzept ist dem KWP 2000/UDS-Kommunikationsmodell nach Abschn. 5.1.1 und dem zugehörigen Single-Frame/Multi-Frame Transportprotokoll nach Abschn. 4.1 sehr ähnlich, doch verwenden die CCP/XCP-Normen leider andere Bezeichnungen (Abb. 6.4). Während das Steuergerät bei KWP 2000/UDS in der Kommunikation als *Server* und der Diagnoserechner als *Client* bezeichnet wird, ist das Steuergerät bei ASAM MCD der *Slave* und der Applikationsrechner der *Master*. Inhaltlich verbirgt sich dahinter exakt dieselbe Rollenverteilung: Alle Aktionen gehen vom externen Rechner (Diagnosetester, Applikationssystem) aus, der die Verbindung zum Steuergerät (*Slave*) aufbaut, Anfragen an das Steuergerät sendet und auf jede Anfrage eine Antwort erhält, bevor die nächste Anfrage gesendet wird (*Request-Response*-Verfahren).

Abb. 6.4 Benennung der Kommunikationspartner bei Diagnose und Applikation



Vergleicht man CCP/XCP mit KWP 2000/UDS, so stellt man fest, dass KWP 2000/UDS bei Verwendung von CAN praktisch dieselbe Funktionalität bietet. Lediglich der Protokoll-Overhead bei KWP 2000/UDS ist höher, so dass die Messdatenerfassung bzw. Stimuluserzeugung mit CCP/XCP höhere Datenraten erreicht und die zeitliche Konsistenz von Datenblöcken besser gewährleistet ist. Historisch gesehen wurde CCP zu einem Zeitpunkt entwickelt, als KWP 2000 ausschließlich über die für Applikationszwecke viel zu langsame K-Line verfügbar war. Als KWP 2000 schließlich auch über CAN eingesetzt wurde, war CCP im Applikationsbereich so fest etabliert, dass den Betroffenen anscheinend eher eine Weiterentwicklung zu dem verwandten, aber auch zu CCP nicht kompatiblen XCP statt einer Modifikation von KWP 2000/UDS für Applikationszwecke sinnvoll erschien. Unter diesem Nebeneinander von in der Ausführung vollständig inkompatiblen, aber in der Grundfunktionalität stark verwandten Protokollen leidet die Zusammenführung der MC- und der D-Funktionen unter dem Dach von ASAM durchgehend.

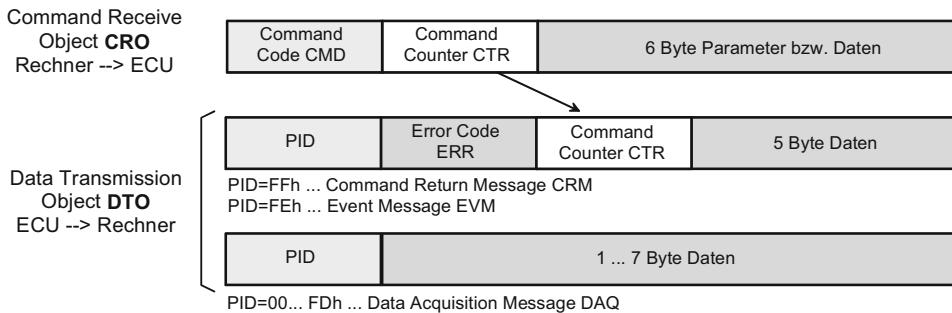
6.2.1 CAN Calibration Protocol CCP

Bei CCP erfolgt die gesamte Kommunikation mit Hilfe von zwei CAN-Botschaften mit jeweils eigenem CAN-Identifier. Die Botschaften werden bei ASAM als Objekte bezeichnet:

- **Command Receive Object CRO** (Befehlsbotschaft) vom Applikationssystem zum Steuergerät; überträgt Befehle zum Steuergerät.
- **Data Transmission Object DTO** (Datenbotschaft) vom Steuergerät zum Applikationssystem; enthält die Steuergeräteantwort.

Eine *Befehlsbotschaft CRO* ist grundsätzlich 8 Byte groß, d. h. umfasst die maximal mögliche Nutzdatenlänge einer CAN-Botschaft, und hat ein festes Format (Abb. 6.5). Das erste Byte enthält den Befehlscode *CMD* (*Command Code*), das zweite Byte eine laufende Nummer *CTR* (*Command Counter*), die mit jeder CRO-Botschaft inkrementiert wird, sowie bis zu 6 Byte Daten bzw. Parameter des Befehls.

Für die *Datenbotschaft DTO* (*Data Transmission Object*) des Steuergerätes gibt es zwei Formate. Bei direkten Antworten *CRM* (*Command Return Message*) auf einen vorigen Befehl ist die DTO-Botschaft ebenfalls grundsätzlich 8 Byte lang. Als erstes Byte wird die CRM-Paket-Kennung (*Packet Identifier*) *PID = FFh* zurückgesendet, im zweiten Byte ein

**Abb. 6.5** CCP-Botschaftsformat

Fehlercode *ERR* (0 h bei fehlerfreier Ausführung des Befehls) und als drittes Byte die laufende Botschaftsnummer *CTR* der Befehlsbotschaft, zu der diese Antwort gehört. Die restlichen 5 Bytes enthalten Antwortdaten.

Dasselbe Format verwenden auch die *Event-Botschaften*, dabei wird aber *PID* = FEh gesetzt. Event-Botschaften werden vom Steuergerät asynchron gesendet, wenn Fehler auftreten, die sich nicht auf einen unmittelbar zuvor empfangenen Befehl beziehen.

Das zweite Format wird für die Messdatenübertragung *DAQ* (*Data Acquisition*) verwendet. Das *PID*-Byte bezieht sich dabei auf die Nummer einer so genannten Datenobjekttabelle *ODT* (*Object Descriptor Table*), die weiter unten beschrieben wird. Die weiteren Bytes enthalten eine variable Anzahl von Daten (max. 7 Byte). Nachdem die Messdatenübertragung vom Applikationssystem konfiguriert und gestartet wurde, erfolgt die Datenübertragung dann aus Sicht des Steuergeräts spontan, d. h. periodisch oder beim Eintreffen eines steuergeräteinternen Ereignisses, ohne dass weitere Anfragen vom Applikationssystem notwendig sind.

Die verfügbaren CCP-Befehle lassen sich in folgende Funktionsgruppen einteilen:

- Verbindungsaufbau und Steuerung,
- Zugriff auf den Steuergerätespeicher,
- Diagnose,
- Flash-Programmierung PGM,
- Kalibrieren CAL,
- Messdatenerfassung DAQ.

CCP-Verbindungsauftbau und Steuerung

CMD	Name	Zweck
01 h	CONNECT	Aufbau der logischen Verbindung zum Steuergerät durch das Applikationssystem. Die Botschaft enthält als Parameter eine 16 bit Kennziffer, die das Steuergerät eindeutig identifiziert. Das angesprochene Steuergerät antwortet mit einer positiven Antwort, alle übrigen Steuergeräte antworten nicht. Auf alle anderen Botschaften reagiert das Gerät nur, solange eine Verbindung besteht.
05 h *	TEST	Diese Botschaft entspricht der CONNECT-Botschaft, baut aber selbst keine Verbindung auf, sondern dient nur der Abfrage, ob ein bestimmtes Steuergerät verfügbar ist. Auf diese Weise kann das Applikationssystem ermitteln, welche Steuergeräte am Bus angeschlossen sind.
07 h	DISCONNECT	Abbau der logischen Verbindung. Auch hier wird als Parameter die 16 bit Kennziffer mitgesendet, die das Steuergerät eindeutig identifiziert. Über einen weiteren Parameter kann die Verbindung auch nur zeitweise beendet werden, in diesem Fall nimmt das Steuergerät keine weiteren Befehle mehr entgegen (außer CONNECT), sendet aber weiterhin Messdaten.
12 h *	GET_SEED	Anfordern eines Initialisierungswertes für den Schlüsselalgorithmus
13 h *	UNLOCK	und Zurücksenden des ermittelten Schlüsselwertes als Login-Prozedur für den Zugriff auf geschützte Funktionen. Die Länge des Schlüsselwertes und der Schlüsselalgorithmus selbst sind wie bei KWP 2000 nicht in der Norm definiert (daher erfordert die MCD 1b-Schicht des Applikationssystems in der Regel die Einbindung eines herstellerspezifischen Schlüsselalgorithmus).
0Ch *	SET_S_STATUS	Setzen und Abfragen von Statusinformationen, die anzeigen, ob das
0Dh *	GET_S_STATUS	Applikationssystem im Steuergerät bestimmte Initialisierungsschritte, z. B. das Initialisieren der Kalibrierdaten oder das Initialisieren der Messdatenerfassung abgeschlossen hat. Außerdem kann das Steuergerät aufgefordert werden, diese Initialisierungsinformationen permanent zu speichern, so dass z. B. nach einem Steuergeräte-Reset automatisch eine Messdatenübertragung gestartet wird.
18 h	GET_CCP_VERSION	Mit dieser Botschaft kann das Applikationssystem abfragen, welche CCP-Version, z. B. CCP 2.1, das Steuergerät unterstützt.
17 h	EXCHANGE_ID	Austausch von Konfigurationsinformationen. Das Steuergerät liefert eine 16 bit Bitmaske zurück, in der es anzeigt, welche der Funktionsblöcke Kalibrieren CAL, Messdatenerfassung DAQ und Flash-Programmierung PGM es unterstützt und für welche dieser Funktionen ein <i>Seed and Key</i> -Login-Vorgang notwendig ist. Der Befehl ermöglicht zusätzlich die Übertragung herstellerabhängiger Konfigurationsinformationen, evtl. mit Hilfe zusätzlicher UPLOAD-Botschaften.

(mit * markierte Befehle sind optional)

CCP-Zugriff auf den Steuergerätespeicher

CMD	Name	Zweck
02 h	SET_MTA	Setzen eines der beiden steuergeräteinternen Speicherpointer MTA0 bzw. MTA1. Die Adresse ist ein 32 bit + 8 bit-Wert, wobei der 8 bit-Teil zur Auswahl einer Speicherseite oder eines Segments dient. Vor jedem Datentransfer von oder zum Steuergerät muss zunächst mit diesem Befehl die Transferadresse gesetzt werden (Ausnahme: SHORT_UP). Bei weiteren Transfers auf aufeinander folgende Adressen ist dies nicht notwendig, da die Transferbefehle bei den Transfers den Pointer automatisch inkrementieren. MTA1 wird nur beim MOVE-Befehl verwendet.
03 h	DNLOAD	Schreiben von 1 bis 5 Datenbytes in den Steuergerätespeicher ab der Steuergeräte-Speicheradresse, auf die der Pointer MTA0 zeigt. Neben den Datenbytes wird die Länge des Datenblocks als Parameter übergeben. MTA0 wird nach der Übertragung automatisch um die entsprechende Anzahl von Bytes inkrementiert.
23 h *	DNLOAD_6	Schreiben von 6 Datenbytes, ansonsten wie DNLOAD. Durch die feste Länge muss keine Längenangabe übertragen werden, so dass ein Datenbyte mehr gesendet werden kann.
04 h	UPLOAD	Lesen von 1 bis 6 Datenbytes aus dem Steuergerätespeicher ab der Steuergeräte-Speicheradresse, auf die der Pointer MTA0 zeigt. MTA0 wird nach der Übertragung automatisch um die entsprechende Anzahl von Bytes inkrementiert.
0Fh *	SHORT_UP	Lesen von 1 bis 5 Datenbytes, dabei wird neben der gewünschten Länge die 32 bit + 8 bit Speicheradresse mitübertragen. Die Pointer MTA0 und MTA1 werden nicht verwendet, d. h. es ist kein vorheriger SET_MTA-Befehl notwendig.
19 h	MOVE	Kopieren einer im Befehl angegebenen Anzahl (32 bit Wert) von Bytes innerhalb des Steuergerätespeichers von der Adresse, auf die MTA0 zeigt, zu der Adresse, auf die MTA1 zeigt.

Im Gegensatz zu KWP 2000 und UDS verwendet CCP für den Zugriff auf Steuergerätedaten stets deren absolute Speicheradressen, d. h. das Applikationssystem benötigt exakte Informationen über die Speicherverteilung des Steuergerätes und der aktuellen Steuergerätesoftware. Für den Zugriff muss die CCP-Protokollsoftware im Steuergerät zwei Pointer MTA0 bzw. MTA1 verwalten, die über einen SET_MTA-Befehl vom Applikationssystem gesetzt werden, bevor über einen folgenden DNLOAD, UPLOAD oder MOVE-Befehl auf den Speicher zugegriffen wird.

CCP-Funktionsgruppe Flash-Programmierung PGM

CMD	Name	Zweck
10 h *	CLEAR_MEMORY	Löschen einer bestimmten Anzahl von Bytes im Flash-ROM des Steuergerätes. Die Anzahl wird im Befehl übergeben, die Anfangsadresse des zu löschenen Speicherbereiches steht im Pointer MTA0 und muss mit einem vorherigen SET_MTA-Befehl initialisiert werden.
18 h *	PROGRAM	Programmiert 1 bis 5 Byte ab der Speicheradresse, auf die der Pointer MTA0 zeigt, in das Flash-ROM des Steuergerätes. Die Länge und die zu programmierenden Daten werden mit diesem Befehl übertragen. MTA0 wird anschließend um die entsprechende Anzahl von Bytes inkrementiert.
22 h *	PROGRAM_6	Wie PROGRAM, aber feste Länge von 6 Byte.
0Eh *	BUILD_CHECKSUM	Berechnen und Auslesen einer Prüfsumme über einen Speicherbereich, dessen Länge im Befehl angegeben wird. Die Anfangsadresse steht in MTA0. Der Algorithmus zur Berechnung der Prüfsumme (1 bis 4 Byte) ist herstellerspezifisch. Checksummen werden üblicherweise verwendet, um die Konsistenz eines Datensatzes und den Erfolg des Flash-Programmierungsvorgangs zu prüfen.

CCP-Funktionsgruppe Diagnose

CMD	Name	Zweck
20 h *	DIAG_SERVICE	Starten eines Diagnoserdienstes im Steuergerät. Mit dem Befehl wird die Service ID des Dienstes sowie 1 bis 4 Parameterbyte übergeben. Die Ergebnisse des Diagnoserdienstes werden in den Steuergerätespeicher ab Adresse MTA0 kopiert, die Länge des Ergebnisses wird in der Steuergeräteantwort zurückübertragen. Die Ergebnisse selbst können über einen anschließenden UPLOAD-Befehl ausgelesen werden.
21 h *	ACTION_SERVICE	Wie DIAG_SERVICE, nur dass eine herstellerspezifische Funktion im Steuergerät gestartet wird.

CCP-Funktionsgruppe Kalibrieren CAL

CMD	Name	Zweck
11 h *	SELECT_CAL_PAGE	Umschalten auf eine neue Speicherseite. Die Anfangsadresse der neuen Speicherseite muss vorher über eine SET_MTA-Botschaft in den Pointer MTA0 geladen werden.
09 h *	GET_ACTIVE_CAL_PAGE	Abfrage, welche Speicherseite gerade aktiv ist. Die Antwort liefert die 32 bit + 8 bit Anfangsadresse der Speicherseite zurück.

Für die Applikation werden häufig speziell umgerüstete Steuergeräte eingesetzt, deren ROM-Speicher durch RAM-Speicher ersetzt wurde. Um im laufenden Betrieb ganze Datensätze, z. B. den gesamten Parametersatz eines Reglers oder ein Kennfeld, in einem Schritt

aktivieren zu können, verwenden die Applikationssteuergeräte häufig so genannte Speicherseiten, von denen genau eine aktiv ist, d. h. für die Steuer- und Regelfunktionen des Steuergerätes verwendet wird, während die andere(n) Speicherseite(n) im Hintergrund vom Applikationssystem verändert werden können. Der Anfang einer Speicherseite wird wieder durch eine 32 bit + 8 bit-Adresse festgelegt.

Das Herunterladen der Datensätze erfolgt mit DNLOAD-Befehlen. Die Gültigkeit des Applikationsdatensatzes kann über BUILD_CHECKSUM geprüft werden. Als Anzeigemöglichkeit, ob der Applikationsdatensatz gültige Daten enthält oder nicht, kann das CAL-Bit im Steuergeräte-Statusbyte verwendet werden, das vom Applikationssystem mit SET_S_STATUS bzw. GET_S_STATUS geschrieben bzw. gelesen werden kann.

CCP-Funktionsgruppe Messdatenerfassung DAQ

CMD	Name	Zweck
14 h	GET_DAQ_SIZE	Abfrage, wie viele ODTs eine bestimmte DAQ Liste enthalten kann (Beschreibung von ODTs und DAQ Listen siehe unten). Die Nummer der Liste muss im Befehl angegeben werden. Durch die Abfrage wird die Liste automatisch gelöscht und eine gegebenenfalls laufende Übertragung gestoppt. Im Befehl wird zusätzlich angegeben, mit welchem CAN-Identifier die Messdaten gesendet werden sollen. Die Unterstützung unterschiedlicher CAN-Identifier ist optional. Neben der Anzahl der ODTs wird in der Steuergeräteantwort auch der PID-Wert des ersten Listeneintrags zurückgeliefert. Alle folgenden PIDs innerhalb der Liste müssen fortlaufend sein.
15 h	SET_DAQ_PTR	Dieser Befehl initialisiert einen steuergeräteinternen Pointer als Vorbereitung für WRITE_DAQ. Im Befehl muss die Nummer der DAQ Liste, die Nummer der ODT und die Nummer des Eintrags in der ODT angegeben werden.
16 h	WRITE_DAQ	Erstellt einen Eintrag in einer ODT. Die Auswahl des Eintrags erfolgt über einen vorherigen SET_DAQ_PTR-Befehl. Der Eintrag enthält eine 32 bit + 8 bit-Adresse eines Steuergerätespeicherbereichs sowie dessen Länge.
06 h	START_STOP	Startet oder stoppt die fortlaufende Messdatenübertragung für eine einzelne DAQ-Liste. Neben der Nummer der Liste und der Anzahl der ODTs in der Liste muss die Häufigkeit der Übertragung vorgegeben werden (Einzelheiten siehe unten). Statt die Messdatenübertragung direkt zu starten, kann der Start auch nur vorbereitet werden.
08 h *	START_STOP_ALL	Startet die Messdatenübertragung für alle DAQ-Listen, die durch einen vorherigen START_STOP-Befehl für den Start vorbereitet wurden.

Mit Hilfe der Funktionsgruppe *Messdatenübertragung* werden in DAQ-Botschaften periodisch Daten aus dem Steuergerät zum Applikationssystem gesendet. Welche Datenbytes in einer CAN-Botschaft übertragen werden, wird für jede DAQ-Botschaft über eine Zuordnungstabelle, die *Object Descriptor Table ODT*, festgelegt (Abb. 6.6). Da in der Regel mehrere Datentabellen ODT verwendet werden, werden die Datentabellen durchnumme-

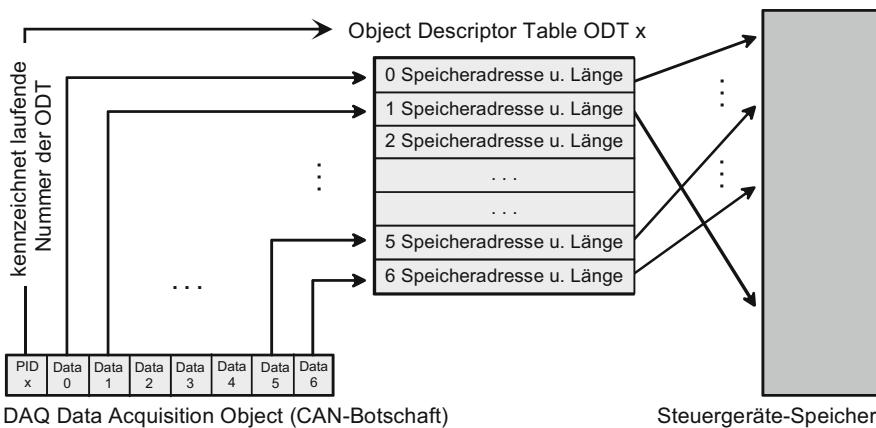


Abb. 6.6 Zuordnung von Messdaten in CAN-Botschaften zu Speicheradressen

riert und die Nummer der Tabelle im PID-Byte, dem ersten Datenbyte der CAN-Botschaft, mit übertragen. Die Einträge in einer ODT legen zu jedem der restlichen 7 Datenbytes der CAN-Botschaft die zugehörige Speicheradresse im Steuergerät fest. Für Daten, die mehr als ein Byte benötigen, kann die Tabelle auch so formatiert sein, dass statt der Adresse jedes Bytes die Anfangsadresse eines Datenwertes und dessen Länge eingetragen werden. Die Gesamtlänge aller in einer einzelnen ODT eingetragenen Daten darf maximal 7 Byte betragen. Da in der Regel mehr als 7 Datenbyte als konsistenter Messdatensatz gemessen und in mehreren DAQ-Botschaften übertragen werden müssen, lassen sich mehrere *Object Descriptor Tabellen ODTs* zu einem zusammengehörigen Datensatz, einer so genannten *DAQ-Liste* zusammenfassen (Abb. 6.7). Auch hier sind mehrere DAQ-Listen möglich, die

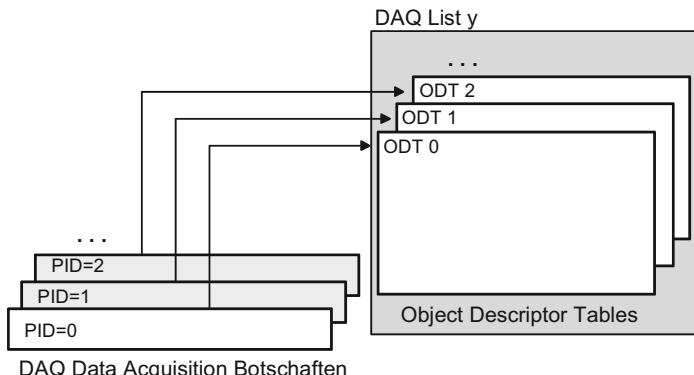


Abb. 6.7 Zusammenfassung von mehreren ODTs zu einer DAQ-Liste

ebenfalls durchnummiert werden. Dabei kann optional für die Übertragung jeder DAQ-Liste ein eigener CAN-Identifier vorgesehen werden.

Bevor eine Messdatenübertragung gestartet wird, muss das Applikationssystem die DAQ-Listen und die ODT-Einträge (in einer Schleife) über die Befehle GET_DAQ_SIZE, SET_DAQ_PTR, WRITE_DAQ initialisieren.

Die eigentliche Datenerfassung und anschließende Datenübertragung im Steuergerät wird für jede DAQ-Liste durch ein Ereignis im Steuergerät ausgelöst. Als Ereignisquellen im Steuergerät können für periodische Übertragungen Zeitgeberinterrupts oder für spontane Übertragungen das Eintreffen von Signalen o. ä. festgelegt sein. Die CCP-Norm macht keinerlei Vorgaben, sondern schreibt lediglich vor, dass die möglichen Ereignisquellen (Event Channels) in den Steuergerätebeschreibungsdateien anzugeben und zu nummerieren sind. Mit dem START_STOP Befehl wird die Nummer der vom Applikationssystem ausgewählten Ereignisquelle an das Steuergerät übertragen. Zusätzlich wird ein *Vorteiler-Faktor N* übertragen, mit dem festgelegt werden kann, dass die Messdatenübertragung nur bei jedem N-ten Ereignis erfolgen soll.

Als Überwachung und Fehlerbehandlung definiert die Norm für alle Befehle Zeitschränken (Timeout) für die Steuergeräteantwort, in der Regel 25 ms. Größere Werte sind bei aufwendigeren Aufgaben wie der Flash-Programmierung oder dem Ausführen von Diagnosediensten möglich.

6.2.2 Extended Calibration Protocol XCP

XCP verwendet dasselbe verbindungsorientierte Request-Response-Kommunikationsschema und weitgehend dieselben Dienste wie CCP. Das Botschaftsformat und die Befehlscodes sind jedoch, wenn auch nur im Detail, unterschiedlich, so dass konkrete XCP-Implementierungen nicht aufwärts kompatibel zu CCP sind. Mit wenigen Ausnahmen sind die Botschaften unabhängig vom gewählten physikalischen Bussystem. In der CAN-Variante werden für jedes Steuergerät zwei feste, in einer Konfigurationsdatei vorgegebene CAN-Identifier verwendet. Einer dient zum Empfangen von Botschaften vom Applikationssystem zum Steuergerät, einer zum Senden von Botschaften vom Steuergerät zum Applikationssystem. Das heißt, im Gegensatz zu CCP verwendet XCP für jedes Steuergerät zwingend ein eigenes Paar von CAN-Identifiern. Über einen zusätzlichen Broadcast-CAN-Identifier kann das Applikationssystem mit Hilfe einer GET_SLAVE_ID-Botschaft die CAN-Identifier abfragen und optional für die Messdatenübertragung mit Hilfe von GET_DAQ_ID/SET_DAQ_ID-Botschaften zusätzliche CAN-Identifier dynamisch festlegen. Auf die wenigen spezifischen Botschaften für die anderen, alternativen Bussysteme (Ethernet, USB usw.) soll hier nicht eingegangen werden. Die XCP-Version für FlexRay wurde erstmals 2006 veröffentlicht. Im Vergleich zu anderen Bussystemen ist, wie am Ende dieses Kapitels dargestellt wird, ein höherer Aufwand notwendig, um eine ausreichende Bandbreite für die Applikationsaufgaben bedarfsabhängig zu garantieren, ohne diese dauerhaft für die eigentlichen Steuer- und Regelaufgaben zu blockieren.

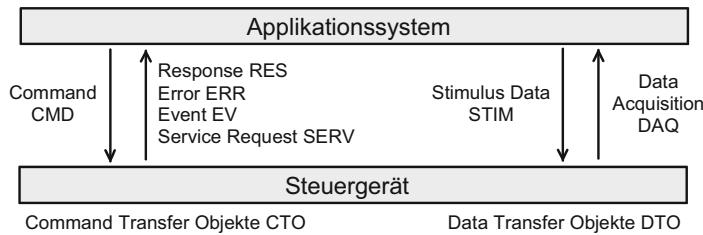


Abb. 6.8 XCP-Botschaftsgruppen

Wie bei CCP werden die Botschaften grob in zwei Gruppen unterteilt (Abb. 6.8):

- **Command Transfer Objekte CTO**

Befehle (*Command Packet CMD*) vom Applikationssystem zum Steuergerät sowie Antworten (*Response Packet RES*) bzw. Fehlermeldungen des Steuergerätes (*Error Packet ERR*) sowie spontane Ereignisbotschaften (*Event Packet EV*) und Dienstanforderungen (*Service Request Packet SERV*) vom Steuergerät zum Applikationssystem.

- **Data Transfer Objekte DTO**

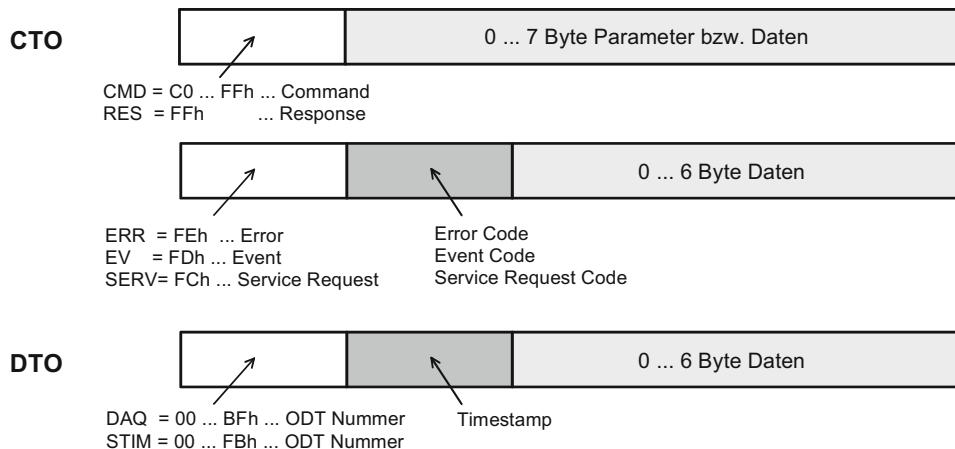
Datenbotschaften (*Data Acquisition Packet DAQ*) für Messdaten vom Steuergerät zum Applikationssystem bzw. Stimulus-Daten (*Stimulus Data Packet STIM*) vom Applikationssystem zum Steuergerät.

Der STIM-Mechanismus wurde bei XCP neu eingeführt und stellt das Gegenstück zur schnellen Messdatenübertragung DAQ dar. Er ermöglicht die Realisierung von Bypass-Systemen, bei denen in der Entwicklungsphase Teile der Steuergerätefunktionalität in ein Applikations- bzw. Simulationssystem ausgelagert werden. DAQ und STIM zusammen ermöglichen dann im laufenden Steuergerätebetrieb Daten in beiden Richtungen schnell zu übertragen.

Die verwendeten XCP-Botschaftsformate (Abb. 6.9) enthalten im Gegensatz zu CCP (vgl. Abb. 6.5) keinen Botschaftszähler. Das erste Byte dient zur eindeutigen Unterscheidung des Botschaftsinhalts. ERR, EV und SERV-Antworten kodieren die Art des Fehlers, Ereignisses bzw. der Dienstanforderung im zweiten Byte. DAQ und STIM-Botschaften enthalten im zweiten Byte optional einen Zeitstempel. Dies ist im Vergleich zu CCP ebenfalls eine Neuerung.

Die XCP-Befehle lassen sich in dieselben Funktionsgruppen unterteilen wie bei CCP:

- Verbindungsaufbau und Steuerung sowie Zugriff auf den Steuergerätespeicher, zusammengefasst als Standardfunktionen STD bezeichnet,
- Flash-Programmierung PGM,
- Messdatenerfassung DAQ und Stimulation STIM,
- Kalibrieren CAL und Speicherseitenumschaltung PAG.

**Abb. 6.9** XCP-Botschaftsformate

Die Konzepte entsprechen weitgehend den bereits im Abschn. 6.2.1 für CCP erläuterten Mechanismen, so dass in den folgenden Tabellen zwar alle XCP-Befehle aufgelistet, im jeweils anschließenden Text aber nur auf die Unterschiede zu CCP eingegangen wird. Die Funktionsgruppe Diagnose ist bei XCP entfallen, da hierfür mittlerweile die Standardprotokolle KWP 2000 und UDS verwendet werden.

Wiederum sind eine Reihe von Befehlen optional, die in den folgenden Tabellen mit * markiert wurden. Bevor einer dieser optionalen Befehle verwendet wird, muss das Applikationssystem mit Hilfe eines GET_..._INFO-Befehls für die zugehörige Funktionsgruppe zunächst abfragen, welche Optionen unterstützt werden.

XCP-Standardfunktionen für Verbindungsaufbau und -steuerung STD Teil 1

CMD	Name	Zweck
FFh	CONNECT	Aufbau der logischen Verbindung zum Steuergerät durch das Applikationssystem. Im Gegensatz zu CCP wird das angesprochene Steuergerät dabei durch die CAN-ID der Botschaft identifiziert, nicht durch eine in der Botschaft enthaltene Kennziffer. Auf alle übrigen Botschaften reagiert ein Steuergerät nur, solange eine logische Verbindung besteht. In der zugehörigen Antwort teilt das Steuergerät mit, welche Funktionsgruppen (Messdatenübertragung, Stimulation, Kalibrieren, Flash-Programmierung) unterstützt werden, welche XCP-Version implementiert ist, wie groß CTO und DTO Botschaften maximal sein dürfen (bei CAN stets 8 Byte), welche Bytefolge (Little bzw. Big Endian) und Adressgranularität (Byte, Word, Double Word) das Steuergerät verwendet und ob das Steuergerät nicht nur Einzelbotschaften sondern bei UPLOAD-Befehlen Blockübertragungen ähnlich dem Transportprotokoll ISO 15765-2 unterstützt.

CMD	Name	Zweck
FBh *	GET_COMM_MODE_INFO	Abfrage, ob das Steuergerät für PROGRAM_NEXT und DOWNLOAD_NEXT-Befehle eine Blockübertragung ähnlich ISO 15765-2 unterstützt.
FDh	GET_STATUS	Setzen und Abfragen von Statusinformationen, die anzeigen, ob ein Speichervorgang der Kalibrierdaten bzw. der Konfiguration der Messdatenerfassung im Flash-ROM oder die Messdatenübertragung aktiv ist. Weiterhin wird abgefragt, welche der Funktionsgruppen Messdatenübertragung, Stimulation, Kalibrieren, Flash-Programmierung durch einen Seed and Key-Mechanismus geschützt sind.
F8 h *	GET_SEED	Anfordern eines Initialisierungswertes für einen Schlüsselalgorithmus und Zurücksenden des ermittelten Schlüsselwertes als Login-Prozedur für den Zugriff auf geschützte Funktionen. Die Länge des Schlüsselwertes und der Schlüsselalgorithmus selbst sind wie bei KWP 2000/UDS nicht in der Norm definiert (daher erfordert die MCD 1b-Schicht des Applikationssystems in der Regel die Einbindung eines herstellerspezifischen Schlüsselalgorithmus). Der Schlüssel muss für jede Funktionsgruppe (Messdatenübertragung, Stimulation, Kalibrieren, Flash-Programmierung) separat ausgetauscht werden.
F7 h *	UNLOCK	Das Protokoll erlaubt das Austauschen von Schlüsselwerten mit mehr als 6 Byte mit Hilfe mehrerer aufeinander folgender CAN-Botschaften.
FAh *	GET_ID	Abfrage der Steuergeräte-Identifikation. Dabei kann es sich um einen ASCII-Text mit dem entsprechenden Inhalt oder um den Verweis (URL) auf eine entsprechende Beschreibungsdatei handeln. Falls die Informationen nicht in die Antwortbotschaft passen, wird ein steuergeräteinterner Pointer gesetzt, von dem aus die Daten mit Hilfe von UPLOAD-Befehlen abgeholt werden können.
FCh	SYNC	Neuinitialisierung nach einem Timeout
FEh	DISCONNECT	Abbau der logischen Verbindung
F1 h	USER_CMD	Übertragung herstellerdefinierter Befehle. Üblicherweise wird über das zweite Byte der Befehl spezifiziert.
F2 h	TRANSPORT_LAYER_CMD	Bussystem-abhängiger Befehl. Der Befehl wird über das zweite Byte ausgewählt. Für CAN sind folgende Befehle definiert: GET_SLAVE_ID (FFh): Abfrage der CAN-Identifier aller am Bus angeschlossenen, XCP-fähigen Steuergeräte. Diese Botschaft wird mit dem Broadcast-CAN-Identifier gesendet. GET_DAQ_ID (FEh) und SET_DAQ_ID (FDh): Abfrage und gegebenenfalls Neusetzen des CAN-Identifiers für eine bestimmte Liste von Messdaten (DAQ-Liste).
F9 h *	SET_REQUEST	Starten eines Speichervorgangs der Kalibrierdaten bzw. der Konfiguration der Messdatenerfassung im Flash-ROM. Der Fortschritt des Speichervorgangs kann über GET_STATUS abgefragt werden.

(mit * markierte Befehle sind hier und in den weiteren Tabellen jeweils optional)

Zu den wesentlichen Neuerungen von XCP gehört, dass zusätzlich zum bekannten Request-Response-Schema, bei dem das Applikationssystem eine Befehlsbotschaft an das Steuergerät sendet und genau eine Antwortbotschaft zurückerhält, zusätzlich als Option der Blockmodus nach ISO 15765-2 (ISO-TP) unterstützt wird. Dabei wird unterschieden zwischen dem so genannten *Slave Block Mode*, bei dem das Steuergerät auf eine einzelne Befehlsbotschaft mit mehreren aufeinander folgenden Antwortbotschaften reagieren kann, und dem *Master Block Mode*, bei dem das Applikationssystem auf eine Befehlsbotschaft unmittelbar weitere Botschaften mit zugehörigen Befehlsparametern bzw. Daten senden darf. Der *Slave Block Mode* wird hauptsächlich in Verbindung mit dem Auslesen größerer Speicherbereiche des Steuergerätes (UPLOAD) verwendet, der *Master Block Mode* beim Herunterladen von größeren Blöcken von Kalibrier- und Programmierdaten (DOWNLOAD). Die Unterstützung der beiden Block-Modi durch das Steuergerät ist optional und wird dem Applikationssystem durch die Antworten auf die Verbindungsaufnahme (CONNECT) bzw. durch explizite Abfrage (GET_COMM_MODE_INFO) mitgeteilt. Dabei erfährt das Applikationssystem auch, welche der Funktionsgruppen CAL/PAG, DAQ, STIM und PGM das Steuergerät unterstützt. Über GET_ID gibt das Steuergerät Klar- textinformationen über sich selbst oder die URL einer zugehörigen Beschreibungsdatei bekannt.

Über GET_STATUS erfährt das Applikationssystem, welche Funktionsgruppen über einen Seed and Key-Mechanismus geschützt sind und kann diese Funktionsgruppen über GET_SEED und UNLOCK freischalten.

Kommt es bei den Steuergeräte-Antworten zu einem Timeout oder antwortet ein Steuergerät auf eine Anfrage mit einer Fehlermeldung, so definiert der Standard, wie das Applikationssystem reagieren soll. Die Timeout-Werte selbst sind in der Schnittstellenkonfigurationsdatei des Steuergerätes festgelegt. Die Mehrzahl der Fehlermeldungen zeigt lediglich an, dass bestimmte Funktionen nicht unterstützt werden oder Parameter außerhalb des zulässigen Bereiches liegen. Bei Timeout-Fehlern wird versucht, die Kommunikation mit einer SYNC-Botschaft neu zu synchronisieren. Falls dies misslingt oder bei anderen schweren Fehlern wird ein neuer Verbindungsaufbau notwendig.

Über einen der Transport Layer Befehle kann das Applikationssystem mit Hilfe einer Broadcast-Nachricht alle am Bus angeschlossenen Steuergeräte auffordern, ihm die vom jeweiligen Steuergerät für die Kommunikation mit dem Applikationssystem verwendeten CAN-Identifier mitzuteilen. Diese Botschaften sind die einzigen Botschaften, auf die alle Steuergeräte ohne vorheriges CONNECT antworten.

Über USER_CMD können herstellerspezifische Befehle implementiert werden.

Wie bereits bei CCP kann das Steuergerät Kalibrierdaten und Konfigurationen für die Messdatenübertragung für einen folgenden Neustart (Reset) speichern. Die Speicherung erfolgt in der Regel im Flash-ROM bzw. EEPROM und wird über SET_REQUEST ausgelöst. Ob die Speicherung erfolgreich abgeschlossen wurde, wird über GET_STATUS abgefragt, bevor ein Reset ausgelöst wird.

XCP-Funktionen für den Zugriff auf den Steuergerätespeicher STD Teil 2

CMD	Name	Zweck
F6 h	SET_MTA	Setzen eines steuergeräteinternen Speicherpointers MTA (Memory Transfer Address). Die Adresse ist ein 32 bit + 8 bit-Wert, wobei der 8 bit-Teil z. B. zur Auswahl einer Speicherseite oder eines Segments dient. Vor jedem Datentransfer von oder zum Steuergerät muss zunächst mit diesem Befehl die Transferadresse gesetzt werden. Bei weiteren Transfers von/zu aufeinander folgenden Adressen ist dies nicht notwendig, da die Transferbefehle bei den Transfers den verwendeten Pointer automatisch inkrementieren.
F0 h EFh *	DOWNLOAD DOWNLOAD_NEXT	Schreiben von Datenbytes in den Steuergerätespeicher ab der Steuergeräte-Speicheradresse, auf die der Pointer MTA zeigt. Neben den Datenbytes wird die Länge des Datenblocks als Parameter übergeben. Wenn das Steuergerät nur Einzel-Botschaften unterstützt, können maximal 6 Byte geschrieben werden, bei Blockübertragung bis zu 255 Byte.
EEh *	DOWNLOAD_MAX	Schreiben von 7 Datenbytes, ansonsten wie DOWNLOAD. Durch die feste Länge muss keine Längenangabe übertragen werden, so dass ein Datenbyte mehr übertragen werden kann.
F5 h *	UPLOAD	Lesen von Daten aus dem Steuergerätespeicher ab der Steuergeräte-Speicheradresse, auf die der Pointer MTA zeigt. Wenn das Steuergerät nur Einzel-Botschaften unterstützt, können maximal 7 Byte gelesen werden, sonst bis zu 255 Byte.
F4 h *	SHORT_UPLOAD	Lesen von 1 bis 7 Datenbytes, dabei wird neben der gewünschten Länge die 32 bit + 8 bit Speicheradresse mitübertragen. Der Pointer MTA wird nicht verwendet, d. h. es ist kein vorheriger SET_MTA-Befehl notwendig.
ECh *	MODIFY_BITS	Setzen oder Löschen von Bits innerhalb eines 32 bit-Datenwortes ab der Steuergeräte-Speicheradresse, auf die der Pointer MTA zeigt.

Der Zugriff auf den Steuergerätespeicher erfolgt wieder über Steuergeräte-Speicheradressen (32 bit + 8 bit), wobei über SET_MTA zunächst ein steuergeräteinterner Pointer gesetzt werden muss, bevor auf die Adressen zugegriffen wird. Die UPLOAD und DOWNLOAD-Befehle und ihre Varianten werden in der Regel in Verbindung mit den nachfolgend beschriebenen Funktionsgruppen PGM und CAL verwendet.

XCP-Funktionsgruppe Flash-Programmierung PGM

CMD	Name	Zweck
D2 h	PROGRAM_START	Schaltet das Steuergerät in den Betriebszustand zum Programmieren des Flash-ROMs bzw. EEPROMs. Erfordert in der Regel ein vorheriges Freischalten über den Seed and Key-Mechanismus. Das Steuergerät teilt in der Antwort mit, ob es für den Download bzw. Upload eine Blockübertragung nach ISO 15765-2 unterstützt.
CCh *	PROGRAM_PREPARE	Falls der Programmieralgorithmus bzw. Treiber für das Flash-ROM nicht bereits im Steuergerät gespeichert ist, bereitet dieser Befehl das Steuergerät auf das Herunterladen der Programmierdaten vor. Das Steuergerät muss einen ausreichend großen Speicherblock als Zwischenspeicher reservieren. Das eigentliche Herunterladen erfolgt anschließend über DOWNLOAD-Botschaften.
CEh *	GET_PGM_PROCESSOR_INFO	Abfrage, welche Optionen der Flash-ROM-Programmieralgorithmus des Steuergerätes unterstützt (Absolute oder Functional Address Mode, Kompression, Verschlüsselung, Zahl der Speichersektoren, wahlfreie Programmierung usw.)
CDh *	GET_SECTOR_INFO	Abfrage der Anfangsadresse bzw. Länge eines Flash-ROM-Sektors (kleinster lösch- und programmierbarer Speicherbereich) sowie Abfrage, in welcher Reihenfolge die Sektoren gelöscht bzw. programmiert werden müssen, falls der Flash-ROM-Baustein keine beliebige Sektor-Reihenfolge (wahlfreie bzw. sequenzielle Programmierung) zulässt.
CBh *	PROGRAM_FORMAT	Informiert das Steuergerät, ob die Daten für den folgenden Programmervorgang komprimiert und/oder verschlüsselt übertragen werden, wählt einen anwendungsspezifischen Programmieralgorithmus aus und schaltet zwischen Absolute und Functional Address Mode um (siehe unten).
D1 h	PROGRAM_CLEAR	Löschen einer bestimmten Anzahl von Bytes im Flash-ROM oder EEPROM des Steuergerätes. Die Anzahl wird im Befehl übergeben, die Anfangsadresse des zu löschenen Speicherbereiches steht im Pointer MTA (Absolute Address Mode) und muss mit einem SET_MTA-Befehl vorher initialisiert werden. Alternativ kann der zu löschenen Bereich (Kalibrierdatensatz, Programmcode, EEPROM usw.) über eine Bitmaske ausgewählt werden (Functional Address Mode).
D0 h	PROGRAM	Programmiert 1 bis 6 Byte in das Flash-ROM bzw. EEPROM des Steuergerätes. Die Länge und die zu programmierenden Daten werden mit diesem Befehl übertragen. Die Anfangsadresse entspricht im Absolute Address Mode dem aktuellen Wert des Pointers MTA oder wird im Functional Address Mode implizit festgelegt. Welcher Modus verwendet wird, wird durch den vorherigen PROGRAM_CLEAR-Befehl definiert. MTA wird anschließend entsprechend inkrementiert.

CMD	Name	Zweck
CAh *	PROGRAM_NEXT	Wie PROGRAM. Diese Botschaft wird für die folgenden Programmierdaten verwendet, falls die Blockdatenübertragung verwendet wird.
C9h *	PROGRAM_MAX	Wie PROGRAM, aber feste Länge von 7 Byte.
C8h *	PROGRAM_VERIFY	Fordert das Steuergerät auf, einen ausgewählten Teil des Flash-ROM-Speichers zu verifizieren. Der Prüfalgorithmus ist herstellerspezifisch. Dies ist eine Alternative zur Checksummenprüfung.
CFh	PROGRAM_RESET	Beendet eine Programmiersequenz. Das Steuergerät bricht die Verbindung ab und führt im Allgemeinen einen Reset durch.
F3h *	BUILD_CHECKSUM	Berechnen und Auslesen einer Prüfsumme über einen Speicherbereich, dessen Länge im Befehl angegeben wird. Die Anfangsadresse steht in MTA. Der Algorithmus zur Berechnung der Prüfsumme (1 bis 4 Byte) ist herstellerspezifisch. Vordefiniert sind Speicherwortsummen bzw. verschiedene CRC-Algorithmen. Checksummen werden üblicherweise verwendet, um die Konsistenz eines Datensatzes und den Erfolg eines Flash-Programmievorgangs zu prüfen.

Mit PROGRAM_START wird das Steuergerät in den Programmiermodus geschaltet. Der eigentliche Programmieralgorithmus kann Teil des Steuergeräteprogrammes in einem Bereich des Flash-ROM-Speichers sein, der nicht gelöscht wird, oder vor dem Programmievorgang mit PROGRAM_PREPARE und DOWNLOAD in das Steuergerät geladen werden. Über die GET_..._INFO- und PROGRAM_FORMAT-Befehle beschafft sich das Applikationssystem Informationen über die Anzahl und Länge der Speichersektoren und darüber, ob die Daten in komprimierter oder verschlüsselter Form in das Steuergerät geladen werden sollen. Ein Sektor ist die kleinste löscharbe Einheit eines Flash-ROM-Speichers und wird entweder durch seine Speicheradresse (Absolute Address Mode) oder durch die Sektornummer (Functional Address Mode) identifiziert. Das Löschen eines Sektors erfolgt durch PROGRAM_CLEAR, bevor dann mit einer Reihe von PROGRAM_...-Botschaften die Daten heruntergeladen und sofort programmiert werden. Anschließend werden die programmierten Daten gegebenenfalls durch eine Checksummenprüfung oder einen steuergerätespezifischen Verifikationsalgorithmus überprüft und schließlich der Programmievorgang mit PROGRAM_RESET abgeschlossen.

XCP-Funktionsgruppe Kalibrieren CAL und Speicherseitenumschaltung PAG

CMD	Name	Zweck
Ebh	SET_CAL_PAGE	Umschalten auf eine neue Speicherseite so, dass die Seite entweder für die Steuergeräte-Applikation oder für das Applikationssystem oder für beide zugänglich ist. Die Speicherseite wird durch eine Segment- und Seitennummer ausgewählt.
EAh	GET_CAL_PAGE	Abfrage, welche Speicherseite in einem vorgegebenen Speichersegment gerade für die Steuergeräte-Applikation bzw. für das Applikationssystem aktiviert ist. Die Antwort liefert die Seitennummer der Speicherseite zurück.
E9 h *	GET_PAGE_PROCESSOR_INFO	Abfrage, wie viele Speichersegmente verfügbar sind und ob der Speicher eingefroren werden kann.
E8 h *	GET_SEGMENT_INFO	Abfrage der Anfangsadresse und Länge eines Speichersegments sowie der Anzahl der Seiten des Segments.
E7 h *	GET_PAGE_INFO	Abfrage, ob das Steuergerät bzw. das Applikationssystem lesend bzw. schreibend auf eine Seite zugreifen können, ob das Steuergerät gleichzeitig mit dem Applikationssystem oder nur exklusiv auf eine Speicherseite zugreifen kann und ob es sich um die Seite mit den Initialisierungsdaten für diese Seite handelt.
E6 h *	SET_SEGMENT_MODE	Einfrieren eines Speichersegmentes.
E5 h *	GET_SEGMENT_MODE	Abfrage, ob ein Speichersegment eingefroren ist.
E4 h *	COPY_CAL_PAGE	Kopieren einer Speicherseite aus einem Speichersegment in ein anderes Speichersegment.

Für Applikationszwecke wird der Speicher eines Steuergerätes häufig ergänzt oder erweitert, indem in ein und denselben physikalischen Adressbereich (Segment) alternativ mehrere Speicherseiten (Page) eingeblendet werden können (Abb. 6.10). Dabei ist in jedem Segment genau eine Speicherseite in den physikalischen Adressraum des Steuergerätes eingeblendet, während die anderen Speicherseiten des Segments lediglich im Hintergrund liegen und nur für das Applikationssystem zugänglich sind. Häufig existiert für jedes Segment eine Seite im Flash-ROM, die das Standardprogramm bzw. die Standarddaten enthält, sowie eine oder mehrere Seiten, die vom Applikationssystem mit geänderten Daten geladen werden. Durch den SET_CAL_PAGE-Befehl kann das Applikationssystem für jedes Segment umschalten, welche Seite im Steuergerätedressraum eingeblendet ist (Active ECU Page) und welche Seite für das Applikationssystem zugänglich ist (Active XCP Page), wobei zwischen *nur lesbar* und *les- und schreibbar* unterschieden wird. Abhängig vom Hardwareaufbau des Speichers ist es gegebenenfalls möglich, dass das Applikationssystem gleichzeitig mit dem Steuergerät auf dasselbe Segment zugreift. Hierdurch kann das Applikationssystem z. B. den Parametersatz eines Reglers im Hintergrund verändern und dann auf den neuen Parametersatz umschalten. Informationen über Segmente und Seiten er-

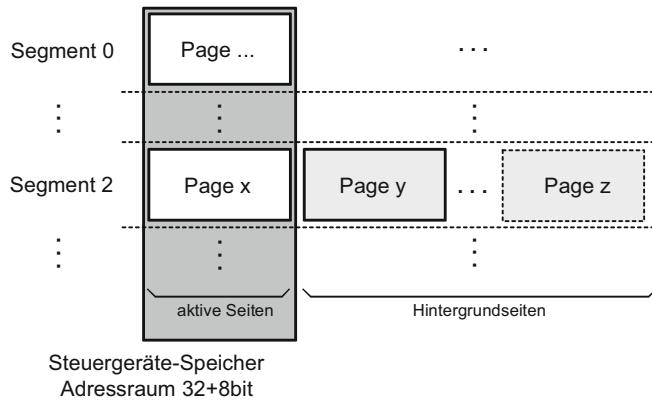


Abb. 6.10 XCP-Speicherstruktur für die Funktionsgruppe CAL/PAG

hält das Applikationssystem über die GET_..._INFO-Abfragen. Mit COPY_CAL_PAGE können Speicherseiten im selben Segment oder in ein anderes Segment kopiert und mit SET_SEGMENT_MODE eingefroren, d. h. gegen weiteres Verändern geschützt und nötigenfalls in die zugehörige Flash-ROM-Seite kopiert werden.

XCP-Funktionsgruppe Messdatenerfassung DAQ und Stimulation STIM

CMD	Name	Zweck
E2 h	SET_DAQ_PTR	Dieser Befehl initialisiert einen steuergeräteinternen Pointer als Vorbereitung für WRITE_DAQ. Im Befehl muss die Nummer der DAQ-Liste, die Nummer der ODT und die Nummer des Eintrags in der ODT angegeben werden (siehe unten).
E1 h	WRITE_DAQ	Erstellt einen Eintrag in einer ODT. Die Auswahl des Eintrags erfolgt über einen vorherigen SET_DAQ_PTR-Befehl. Der Eintrag enthält eine 32 bit + 8 bit-Adresse eines Steuergerätespeicherbereichs sowie dessen Länge. Nach dem Auslesen wird der DAQ-Pointer automatisch auf den nächsten Eintrag weitergeschaltet.
DBh *	READ_DAQ	Auslesen eines Eintrags in einer ODT. Komplementäre Funktion zu WRITE_DAQ.
DEh	START_STOP_DAQ_LIST	Startet oder stoppt die laufende Messdatenübertragung für eine einzelne DAQ-Liste. Statt die Messdatenübertragung direkt zu starten, kann eine Liste auch nur zum Start oder Stop vorausgewählt werden.
DDh	START_STOP_SYNC	Startet oder stoppt die Messdatenübertragung für alle DAQ-Listen gleichzeitig, die zuvor durch START_STOP_DAQ_LIST ausgewählt wurden.
E3 h	CLEAR_DAQ_LIST	Löschen einer DAQ-Liste, Stoppen der zugehörigen Datenübertragung, falls diese aktiviert war.

CMD	Name	Zweck
E0 h	SET_DAQ_LIST_MODE	Setzt Parameter für die Datenübertragung: Übertragungsrichtung Messdaten (DAQ)/Stimuli (STIM), Zeitstempel ein/aus, Übertragung der ODT-Nummer ein/aus (nur möglich, wenn die Daten der DAQ-Liste mit einem eindeutigen CAN-Identifier übertragen werden). Auswahl des Ereigniskanals, der die eigentliche Datenübertragung triggert, des Teilerfaktors für die Wiederholrate und der Priorität, mit der die Daten übertragen werden, falls Daten aus mehreren DAQ-Listen gleichzeitig zur Übertragung anstehen.
DFh	GET_DAQ_LIST_MODE	Abfrage der mit SET_DAQ_LIST_MODE gesetzten Werte für eine DAQ-Liste.
DCh *	GET_DAQ_CLOCK	Abfrage des aktuellen Standes des Steuergeräte-Timers, mit dem die Zeitstempel der DAQ-Botschaften erzeugt werden. Wird in der Regel zur Synchronisation mit der Zeitbasis des Applikationssystems verwendet.
DAh *	GET_DAQ_PROCESSOR_INFO	Abfrage verschiedener DAQ-Optionen: Dynamisch eingerichtete DAQ-Listen, Resume-Modus (Speichern von DAQ-Listen im Flash-ROM und Start einer automatischen Datenübertragung nach Steuergeräte-Reset), Unterstützung von Zeitstempeln, Abschalten der Übertragung von ODT-Nummern, Anzahl der möglichen DAQ-Listen und Unterscheidung von statisch konfigurierten und dynamisch einrichtbaren Listen, Anzahl der verfügbaren Ereigniskanäle und andere.
D9 h *	GET_DAQ_RESOLUTION_INFO	Abfrage weiterer DAQ-Optionen: Größe der ODT-Einträge (1, 2, 4 Byte), max. zulässige Länge des Speicherbereichs, auf den ein ODT-Eintrag zeigt, Länge und Zeitauflösung des Zeitstempels.
D8 h *	GET_DAQ_LIST_INFO	Abfrage der Optionen einer DAQ-Liste: Statisch/dynamisch konfigurierte Liste, fester/konfigurierbarer Ereigniskanal, Anzahl der zugeordneten ODTs und Anzahl der Einträge je ODT.
D7 h *	GET_DAQ_EVENT_INFO	Abfrage der Einstellungen eines bestimmten Ereigniskanals: Wiederholrate, Priorität, Anzahl der DAQ-Listen, die durch das Ereignis getriggert werden können. Die Antwort enthält außerdem eine Information, ob eine Klartextbeschreibung für den Kanal verfügbar ist, die dann über anschließende UPLOAD-Befehle gelesen werden kann.
D6 h *	FREE_DAQ	Gibt alle dynamisch eingerichteten DAQ-Listen frei. Muss gesendet werden, bevor dynamische DAQ-Listen mit ALLOC_DAQ eingerichtet werden.
D5 h *	ALLOC_DAQ	Reserviert Speicherplatz für eine bestimmte Anzahl von DAQ-Listen (so genannte dynamische DAQ-Listen).
D4 h *	ALLOC_ODT	Reserviert Speicherplatz für eine bestimmte Anzahl von ODTs und ordnet sie einer bestimmten DAQ-Liste zu.
D3 h *	ALLOC_ODT_ENTRY	Reserviert Speicherplatz für eine bestimmte Anzahl von ODT-Einträgen und ordnet Sie einer ODT zu.

Die automatische, periodische Datenübertragung wird in derselben Weise wie bei CCP durch in DAQ-Listen zusammengefasste Object Descriptor Tabellen ODT mit Hilfe von SET_DAQ_PTR und WRITE_DAQ konfiguriert (Abb. 6.6 und 6.7). Neu bei XCP ist, dass nicht nur Messdaten vom Steuergerät zum Applikationssystem übertragen werden können (DAQ), sondern dass auch Stimulidaten in umgekehrter Richtung übertragen werden dürfen (STIM). Außerdem können zusammen mit den Daten Zeitstempel übertragen werden, die sich auf den steuergeräteinternen Zeitgeber beziehen und anzeigen, wann die Messdaten im Steuergerät erfasst bzw. wann die Stimulidaten im Steuergerät wirksam werden sollen. Ob das Steuergerät den STIM-Betrieb unterstützt und welche Optionen bei DAQ und STIM im Einzelnen möglich sind, kann über GET_DAQ_..._INFO-Botschaften abgefragt werden.

Das Starten und Stoppen der eigentlichen Datenübertragung erfolgt wiederum mit Hilfe von START_STOP...-Befehlen.

Während bei CCP die Menge der gleichzeitig übertragbaren Messwerte durch die in der Steuergerätesoftware vorgegebene Anzahl von DAQ-Listen und ODT-Einträgen fest vorgegeben war, können XCP-taugliche Steuergeräte optional das dynamische Anfordern weiterer DAQ-Listen und ODT-Einträge mit Hilfe von ALLOC_...-Botschaften zulassen, wenn sie über ausreichend freien RAM-Speicher verfügen und dessen dynamische Verwaltung erlauben.

Neben den Antworten auf unmittelbare Anfragen des Applikationssystems und der periodischen Messdatenübertragung kann das Steuergerät spontan *EV-Botschaften* an das Applikationssystem senden, wenn spezielle Ereignisse eintreten. Dazu gehört z. B., wenn der Speichervorgang von Kalibrierdaten oder die Konfiguration von DAQ-Listen im Flash-ROM abgeschlossen ist, wenn die Ausführung eines Befehls länger dauert und die Timeout-Überwachung neu gestartet werden muss, wenn die Messdatenübertragung das Steuergerät überlastet ist, wenn das Steuergerät von sich aus die Verbindung beenden will oder wenn ein herstellerdefiniertes Ereignis auftritt. Mit Hilfe einer *SERV-Botschaft* kann das Steuergerät das Applikationssystem sogar auffordern, das Steuergerät zurückzusetzen. Dies erfordert in der Regel ein Abschalten und Wiedereinschalten der Spannungsversorgung.

XCP über FlexRay: XCP-Botschaften sind sowohl im statischen als auch im dynamischen Segment eines FlexRay-Kommunikationszyklus zulässig (vgl. Abschn. 3.3). Die zugehörigen Zeitschlitzte werden bei der Konfiguration des FlexRay-Bussystems für XCP reserviert, wobei die beiden FlexRay-Kanäle unabhängig voneinander belegt werden dürfen. Zeitschlitzte im statischen Segment müssen eindeutig einem Steuergerät zugeordnet werden, während Zeitschlitzte im dynamischen Segment von mehreren Steuergeräten im Multiplex-Betrieb genutzt werden können. Dieses *Slot Multiplexing* erfolgt über den Zykluszähler der FlexRay-Botschaften. Auf diese Weise kann z. B. festgelegt werden, dass mehrere Steuergeräte denselben Zeitschlitz verwenden, wobei ein Steuergerät A nur in jedem zweiten Kommunikationszyklus sendet, während zwei andere Geräte B und C sich in den verbleibenden Zeitschlitzten abwechseln (Abb. 6.11).

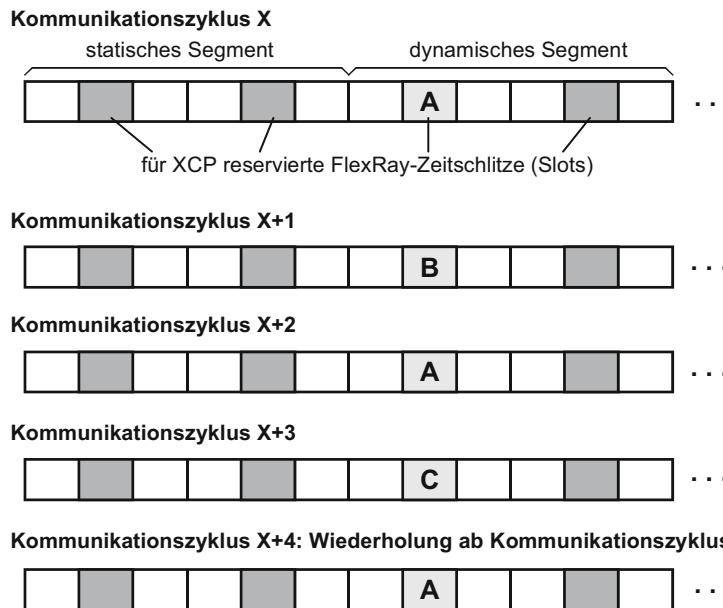


Abb. 6.11 Beispiel für die Reservierung von FlexRay-Zeitschlitten (Slots) für XCP

XCP-Botschaften werden grundsätzlich nur ereignisgesteuert gesendet. Falls keine Daten zu senden sind, bleibt der entsprechende Zeitschlitz einfach leer. Die Lage, Länge und Zuordnung dieser Zeitschlitzte wird mit Hilfe von AML/A2L- oder FIBEX-Konfigurationsdateien (siehe Abschn. 6.2.3 bzw. 6.3) festgelegt und kann, soweit die Steuergeräte dies zulassen, vom Applikationssystem mit Hilfe der FlexRay-spezifischen XCP-Botschaften FLX_ASSIGN, FLX_ACTIVATE bzw. FLX_DEACTIVATE dynamisch konfiguriert und aktiviert werden. Wieweit dies tatsächlich genutzt werden kann, wird durch die verfügbaren FlexRay-Kommunikationscontroller jedoch erheblich eingeschränkt. Bei vielen derzeitigen Kommunikationscontrollern ist die Zuordnung von Zeitschlitzten und die Veränderung der Botschaftslänge nur nach einer Neuinitialisierung und damit einer Unterbrechung der Netzwerkkommunikation möglich, so dass die notwendige Bandbreite für XCP-Anwendungen oft dauerhaft reserviert werden muss, obwohl sie eigentlich nur kurzzeitig benötigt wird. Eine ähnliche Problematik ergibt sich derzeit übrigens auch bei der Flash-Programmierung über FlexRay.

Zur eindeutigen Unterscheidung der Steuergeräte beim *Slot Multiplexing* erhält jedes Gerät eine netzweit eindeutige, 1 Byte lange Knotenadresse NAX (*Network Address for XCP*), die z. B. mit der physikalischen Steuergeräteadresse für die KWP 2000- bzw. UDS-Diagnose identisch sein kann. Bei den vom Applikationssystem gesendeten XCP-Botschaften dient die Knotenadresse zur Festlegung des Empfänger-Steuergerätes, bei den vom Steuergerät zum Applikationssystem gesendeten XCP-Botschaften zur Identifikation des Senders. Die Knotenadresse NAX und einige weitere Steuerbytes werden in einem

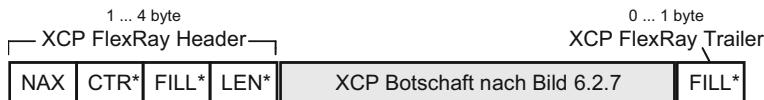


Abb. 6.12 XCP on FlexRay Botschaftsformat (* optionale Bytes)

XCP FlexRay Header gesendet (Abb. 6.12), dem dann die eigentliche XCP-Botschaft mit dem üblichen Format nach Abb. 6.9 folgt. Optional kann im Header ein Zählbyte CTR (*Counter*) mitgesendet werden, mit dem Applikationssystem und Steuergeräte ihre jeweils gesendeten Botschaften unabhängig voneinander durchnummerieren können, so dass der Empfänger fehlende Botschaften leichter erkennen kann. Ebenfalls optional ist ein Längenbyte LEN (*Length*), das die Länge der folgenden, eigentlichen XCP-Botschaft nach Abb. 6.9 angibt. Dies ist vor allem dann sinnvoll, wenn die Länge der XCP-Botschaften dynamisch variiert.

Der Header muss gegebenenfalls durch Füll-Bytes aufgefüllt werden, falls der Kommunikationscontroller beispielsweise aus Geschwindigkeitsgründen nur Datenblöcke verarbeiten kann, die 2 oder 4 Byte lang sind. Dasselbe gilt für die gesamte Botschaft. Da FlexRay-Botschaften immer nur eine geradzahlige Anzahl von Bytes enthalten dürfen, muss die Botschaft eventuell mit einem Füll-Byte am Ende (XCP FlexRay Trailer) aufgefüllt werden. Der Inhalt der Füll-Bytes ist beliebig. Welches Header-Format jedes Steuergerät verwendet, wird bei der XCP Konfiguration eindeutig festgelegt.

Bei Bedarf können mehrere XCP Botschaften (von einem Sender zu genau einem Empfänger) innerhalb derselben FlexRay-Botschaft zusammengefasst werden. Dabei enthält lediglich die erste XCP-Botschaft den vollständigen Header. Bei den weiteren XCP-Botschaften wird nur noch das LEN-Byte gesendet.

6.2.3 AML-Konfigurationsdateien für XCP und CCP

Die CCP- und XCP-Standards definieren Konfigurationsdateien, mit denen die Fähigkeiten und Parameter eines konkreten Steuergerätes beschrieben werden können. Bei den Beschreibungsdateien handelt es sich um Textdateien, die eine Syntax verwenden, die als ASAP2/ASAM *Meta Language AML* bezeichnet wird. Die Konfigurationsdateien werden in der Regel mit der Endung *.A2L* gespeichert.

Die Beschreibung erfolgt hierarchisch in aufeinander folgenden oder verschachtelten/`begin .../end` Blöcken (Tab. 6.2). Ein hierarchisch übergeordneter Block definiert allgemeine Projektdaten, z. B. Name und Versionsnummer des Projekts. Auf einer tieferen Ebene folgt eine Definition der vom Steuergerät unterstützten optionalen XCP-Befehle und getrennt davon der für das konkrete Bussystem erforderlichen Parameter wie z. B. der CAN-Identifier.

Tab. 6.2 Prinzip der XCP-Konfigurationsdateien

```

/begin PROJECT XCP
"Projekt 28147891" /* Allgemeine Angaben zum Projekt */
/begin HEADER
"EDC 21 for Customer X Model Y"
VERSION "V3.1"
.

/end HEADER
/begin MODULE
/begin A2ML           /* AML Header-Dateien*/
/include XCP_Common.aml
/include XCP_on_CAN.aml
/end A2ML
/begin IF_DATA XCP
/begin PROTOCOL_LAYER /* Protokollkonfiguration */
BYTE_ORDER_MSB_FIRST
ADDRESS_GRANULARITY_BYTE
OPTIONAL_CMD_GET_ID
OPTIONAL_CMD_SET_REQUEST
.

/end PROTOCOL_LAYER
.

/begin XCP_ON_CAN      /* Bussystem-Konfiguration */
0x0100                /* XCP on CAN version 1.0 */
CAN_ID_BROADCAST 0x0100 /* Broadcast CAN Id */
CAN_ID_MASTER 0x0200   /* Application Sys. CAN Id*/
CAN_ID_SLAVE 0x0300    /* ECU CAN Id */
BAUDRATE 500000        /* CAN Baudrate */
.

/end XCP_ON_CAN
/end IF_DATA
/end MODULE
/end PROJECT

```

Im Sinne der Hierarchie ist es möglich, in einem hierarchisch höher stehenden Block Defaultwerte für bestimmte Parameter anzugeben, die dann in einem niedriger stehenden Block überschrieben werden.

Sinnvollerweise werden die Konfigurationsdaten für den Bussystem-unabhängigen und für den Bussystem-abhängigen Teil in separate Dateien ausgelagert, die mit Hilfe von/include Anweisungen in die zentrale Konfigurationsdatei eingebunden werden.

Welche Parameter konfiguriert werden können, die dabei zu verwendeten Schlüsselworte sowie die zulässigen Werte sind in Definitionsdateien festgelegt, die eine C-artige Syntax verwenden und häufig mit der Endung .AML gespeichert werden (Tab. 6.3). Auf diese Weise kann mit Hilfe eines Softwarewerkzeuges sowohl eine Bedienoberfläche geschaffen werden, die nur diejenigen Parameter und Optionen zur Konfiguration anbietet, die zu verändern sind, als auch eine automatisierte Überprüfung der Syntax der Konfigurationsdateien durchgeführt werden.

AML stammt noch aus einer Zeit, in der solche Pseudo-Textdateien mit anwendungsabhängig definierter Struktur und Syntax zur Konfiguration von Software allgemein üblich

Tab. 6.3 Ausschnitt aus der Definitionsdatei für XCP über CAN

```

/* A2ML data type description */
/* uchar    unsigned 8 Bit */
/* ulong    unsigned integer 32 Bit */

. . .

/***** start of CAN *****/
struct CAN_Parameters
{ uint;           /* XCP on CAN version e.g. "1.0" = 0x0100 */
  taggedstruct
  { "CAN_ID_BROADCAST" ulong;    /* Broadcast CAN Id */
    "CAN_ID_MASTER"     ulong;    /* Application Sys. CAN Id */
    "CAN_ID_SLAVE"      ulong;    /* ECU CAN Id */
    "BAUDRATE"          ulong;    /* CAN Baudrate */
  };
. . .
};

/***** end of CAN *****/

```

waren. Leider führte dies auch im Fall von AML zu einer individuellen Lösung und erfordert daher speziell für AML entwickelte Werkzeuge. Die Erstellung mit einem einfachen Texteditor ist zwar theoretisch möglich, in der Praxis aber zu aufwendig und fehlerträchtig. Mittlerweile bietet die Informatik mit XML (Extended Markup Language, siehe Kasten unten) ein verallgemeinertes Grundkonzept für solche Beschreibungs- und Konfigurationsdateien an. Für deren Erstellung und Überprüfung existieren gute Werkzeuge, die eine höhere Verbreitung besitzen und daher ausgereifter und zukunftssicherer sein dürfen. Aus diesem Grund ist zu erwarten, dass die XCP-Definitionsdateien zukünftig als XML Schema vorliegen und die Konfigurationsdateien in XML erstellt werden. Ein möglicher Kandidat dafür ist das *Calibration Data Format CDF* (siehe Abschn. 6.5.2), ergänzt durch FIBEX für die Buskommunikation. Es kann davon ausgegangen werden, dass die Werkzeughersteller Konvertierungswerkzeuge anbieten werden, die die Umsetzung von AML-Dateien nach XML automatisiert durchführen und für eine Übergangszeit eine Koexistenz beider Formate zulassen werden.

6.2.4 Interface zwischen Bus und Applikationssystem ASAM MCD 1b

Um die darüberliegenden Schichten des Applikationssystems vom darunterliegenden Treiber für das Busprotokoll (CCP, XCP, ...) zu entkoppeln (vgl. Abb. 6.3), wird eine Zwischenschicht mit standardisierten Funktionen eingesetzt (Abb. 6.13).

Ein Messvorgang wird durch Aufruf der Funktion INIT_READ() initialisiert, die Messung durch SYNC() gestartet. Das Lesen der Messwerte erfolgt durch ein oder mehrfachen Aufruf der Funktion READ(). Mit STOP() wird die Messung wiederum beendet. Der Zugriff auf Applikationsparameter eines Steuergerätes bei einem Kalibriervorgang wird mit INIT_ACCESS() eingeleitet und mit ACCESS() ausgeführt. In beiden Fällen wird der Vor-

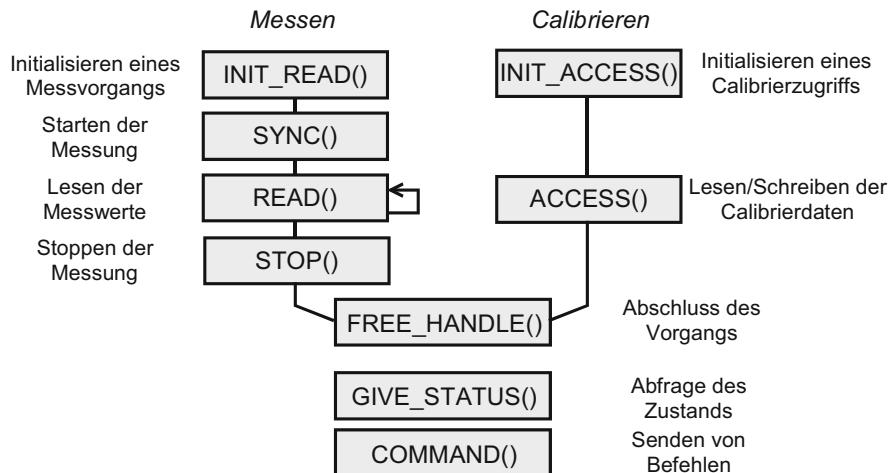


Abb. 6.13 Standardisierte Funktionen der ASAM MCD 1b Schnittstelle

gang mit der Funktion FREE_HANDLE() abgeschlossen. Mit Hilfe der Funktion COMMAND() können jederzeit herstellerspezifische Befehle an das Steuergerät gesendet werden, z. B. zur Auswahl einer Speicherseite oder zum Zurücksetzen des Steuergerätes. Alle Funktionen erwarten einen komplexen Parametersatz, der über verkettete Strukturen übergeben wird. Die Mehrzahl dieser Parameter, z. B. die Speicheradressen, auf die im Steuergerät zugegriffen werden soll, oder die Größen und das Datenformat der zugehörigen Variablen erhält der ASAM MCD 1-Treiber aus einer ASAP2-Konfigurationsdatei (siehe Abschn. 6.2.3).

Bei neueren Werkzeugen wird die MCD 1b-Schnittstelle voraussichtlich durch die MVCI/D-PDU-Schnittstelle abgelöst werden, die in Abschn. 6.8 beschrieben wird.

Extensible Markup Language XML & Unified Modeling Language UML

XML ist ein Mitte der 90er Jahre zunächst für Internet-Anwendungen vorgeschlagenes Textformat, mit dem Informationen strukturiert werden können (Abb. 6.14).

Die in XML-Dateien enthaltene Information wird durch so genannte Tags strukturiert. Tags sind in <...> geschriebene, vordefinierte Bezeichner, z. B. <Bussysteme>, die kennzeichnen, welche Art von Information folgt. Die Information selbst, als Wert des Elements bezeichnet, steht hinter dem in <...> geschriebenen Bezeichner und wird mit dem in </...> wiederholten Bezeichner abgeschlossen, wie z. B. der Name PowertrainCAN des CAN-Bussystems in Abb. 6.14. Bei hierarchisch untergeordneten Informationen werden die Tags verschachtelt, z. B. <Bussysteme> und <Bussystem>. Bei hierarchisch gleichwertigen Informationen werden die Tags einfach aneinander gereiht, z. B. <Bussysteme> und

<Steuergeräte>, wobei dasselbe Tag auch mehrfach wiederholt werden darf. Auf der obersten Hierarchieebene muss und darf es genau ein Element geben, das sogenannte Wurzelement, im Beispiel <Fahrzeug>. Darunter können Elemente beliebig aneinander gereiht oder geschachtelt werden. Optisch wird die Verschachtelung durch Einrückungen der Elemente in der Datei visualisiert.

```

<Fahrzeug>
  <name> AudiVolksPorscheBenzMotorenWagen T520 </name>
  <Bussysteme . . . >
    <Gateway-Ref ID="ecu1" />
    <Bussystem ID="bus1" type="CAN" . . . >
      <name> PowertrainCAN </name>
      . . .
    </Bussystem>
    <Bussystem ID="bus2" type="LIN" . . . >
      <name> LeftDoorLIN </name>
      <Gateway-Ref ID="ecu5" />
      . . .
    </Bussystem>
  </Bussysteme>
  . . .
  <Steuergeräte . . . >
    <Steuergerät ID="ecu1" . . . >
      <name> EngineECU </name>
      <Bussystem-Ref ID-REF="bus1" />
      . . .
    </Steuergerät>
    . . .
    <Steuergerät ID="ecu5" . . . >
      <name> DashboardECU </name>
      <Bussystem-Ref ID-REF="bus1" />
      <Bussystem-Ref ID-REF="bus2" />
    </Steuergerät>
  </Steuergeräte>
</Fahrzeug>

```

Abb. 6.14 Beispiel einer XML-Datei

Zusätzlich zu den Werten der Elemente können den Tags auch so genannte Attribute direkt zugeordnet werden, die einen Namen und einen Wert aufweisen. In solchen Attributen können weitere Informationen untergebracht werden, vorzugsweise Sortier- und Klassifikationswerte, im Beispiel etwa `ID="bus1"` oder `type="CAN"`.

Die Beschreibung des Datenmodells, das den XML-Konfigurationsdateien zugrunde liegt, erfolgt in neueren Spezifikationen wie FIBEX oder ODX, die im folgenden beschrieben werden, häufig in einer UML-artigen Darstellung (Abb. 6.15). Die **Unified Modeling Language UML** ist eine standardisierte grafische Notation, mit der objektorientierte Softwaresysteme beschrieben werden. Zu den wesentlichen Konzepten, die dabei immer wieder auftauchen, gehören Klassen und

Objekte sowie deren Beziehungen untereinander. Im dargestellten Beispiel enthält das Objekt **Fahrzeug** ein oder mehrere Objekte der Klassen **Bussystem** und **Steuergerät**. Die Zuordnung „enthält“ wird in der Informatik als *Aggregation* bezeichnet und in der UML-Notation mit einer Raute an der Verbindungsline zwischen den Objekten dargestellt. Da ein Steuergerät an mehrere Bussysteme angeschlossen sein kann, wird die Beziehung zwischen **Steuergerät** und **Bussystem** nicht durch eine *Aggregation* sondern durch eine *Assoziation*, d. h. in der UML-Grafik eine einfache Linie zwischen zwei Objekten, dargestellt. Die zulässige Anzahl der Objekte wird durch *Kardinalitäts-* oder *Multiplizitätszahlen* beschrieben, wobei im Beispiel 1 **Fahrzeug** eine beliebige Anzahl $0 \dots *$ von **Bussystemen** bzw. **Steuergeräten** enthalten kann, während 1 **Bussystem** mindestens zwei oder mehr ($2 \dots *$) **Steuergeräte** verbindet. Bussysteme existieren in verschiedenen Ausprägungen, z. B. CAN und FlexRay, die einige logische Eigenschaften teilen, z. B. die Bitrate, aber auch unterschiedliche Eigenschaften haben, z. B. *Sync Frames*, die es nur bei FlexRay gibt. In der Informatik spricht man dabei von *Eltern-Kind-Beziehung*, wobei in der UML-Notation die *Eltern*-Seite durch ein Dreieck gekennzeichnet wird.

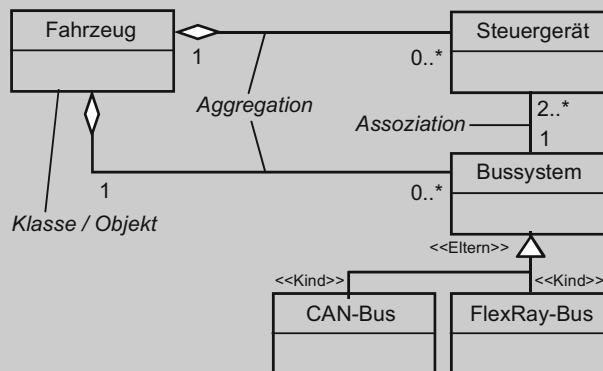


Abb. 6.15 UML-Darstellung von Objektbeziehungen

Bei der Umsetzung solcher UML-Datenmodelle in die XML-Beschreibung werden *Aggregationen* in der Regel durch ineinander verschachtelte XML-Elemente, *Assoziationen* durch Referenzen auf andere Objekte abgebildet. In Abb. 6.14 etwa verweist der Eintrag für das Steuergerät `DashboardECU` über das Tag `<Bussystem-Ref>` auf das Bussystem `bus2`. Auch das Konzept der *Vererbung* von Eigenschaften wird in XML abgebildet. So wird auf der übergeordneten Ebene der Bussysteme in Abb. 6.14 mit `Gateway-Ref ID="ecu1"` ein Steuergerät als Default-Gateway festgelegt. Dieses wird von allen hierarchisch untergeordneten Bussen verwendet (*geerbt*), die nicht selbst eine eigene Gateway-Definition enthalten. Das untergeordnete Bussystem `bus2` dagegen wiederholt dieses Tag

Gateway-Ref, verweist aber für diesen Bus auf das Steuergerät ecu5 als Gateway.

XML ist primär ein Textformat, das selbstverständlich auch Zahlen in Textform enthalten kann. Aber auch binäre Informationen, z. B. compilierter Programmcode, lassen sich aus einer XML-Beschreibung heraus als externe Datei referenzieren. Die Auswertung der binären Information oder deren direkte Bearbeitung in XML dagegen ist nicht möglich und erfordert externe Werkzeuge.

Die Syntax von XML hat einige Ähnlichkeit mit der Auszeichnungssprache HTML, mit der die Informationen im Internet dargestellt werden. Dort werden Überschriften verschiedener Ebenen z. B. mit Tags wie `<H1>...</H1>` oder Textpassagen mit `<P>...</P>` gekennzeichnet. Attribute wie `font="Helvetica"` werden für die Festlegung von Schriftarten oder Schriftgrößen verwendet. Im Unterschied zu HTML, bei dem die Tags vor allem die optische Darstellung der Information festlegen, beschreiben die Tags bei XML vor allem die inhaltliche Bedeutung der Information. Weit entscheidender ist jedoch, dass die Tags und ihre Bedeutung bei HTML unveränderlich vorgegeben sind, während sie bei XML frei definierbar sind. Die Definition erfolgt in der Regel in einem so genannten **XML-Schema**. Dort wird festgelegt, welche Tags in einem Dokument an welcher Stelle und in welcher Häufigkeit verwendet werden können und müssen, welche Attribute ein Tag haben darf und welche Art von Werten (Text, Zahlen, Wertebereich usw.) für die Elemente und Attribute zulässig ist. Erst in Verbindung mit einer derartigen, für eine bestimmte Aufgabenstellung standardisierten XML-Schema-Definition sind Daten auf XML-Basis wirklich austauschbar.

Obwohl XML ein textbasiertes Format und damit theoretisch mit einfachsten Mitteln lesbar ist, werden XML-Dokumente schnell unübersichtlich und bei Änderungen fehleranfällig. In der Praxis sind daher stets geeignete Werkzeuge notwendig, die den Anwender bei der Suche nach Informationen unterstützen und bei der Bearbeitung eine sofortige Überprüfung durchführen. Insoweit unterscheiden sich in XML erstellte Beschreibungen für den Anwender nicht von Beschreibungen in AML, CanDB, OIL oder anderen Industriestandards. Der wesentliche Unterschied zu diesen proprietären Formaten ist, dass es für die Toolhersteller bei XML leichter ist, solche Werkzeuge zu erstellen, weil es für XML bereits viele Softwarekomponenten aus anderen Bereichen der Informatik gibt, die verhältnismäßig einfach in die Werkzeuge für den Automobilbereich integriert werden können.

6.3 Field Bus Exchange Format FIBEX

Mit der FIBEX-Spezifikation versucht ASAM, ein einheitliches XML-Beschreibungsformat für die Kommunikation über die verschiedenen Bussysteme von CAN über LIN, FlexRay bis zu MOST und Erweiterungsmöglichkeiten für weitere Busse zu definieren. FIBEX soll die bisher üblichen proprietären Formate wie CANdb (DBC Dateien) oder LIN Configuration Language und Node Capability File (LDF und NCF Dateien) mittelfristig ablösen. Die schnellste Einführung dieses Formats erfolgte bei FlexRay, da dafür noch kein anderes Beschreibungsformat etabliert war.

Der Schwerpunkt von FIBEX liegt auf der Beschreibung der On-Board-Kommunikation im normalen Fahrzeugbetrieb und soll in Spezifikations-, Test- und Simulationswerkzeugen verwendet werden. Wie bei vielen anderen derartigen Standardisierungsbestrebungen bleibt aber auch bei FIBEX seltsam unklar, ob es sich wirklich um einen universellen Standard handeln wird, der von allen Werkzeugherstellern unterstützt wird. Prinzipiell wäre es wohl möglich, etwa die Beschreibung von CCP und des ebenfalls vom ASAM-Konsortium spezifizierten XCP in FIBEX zu integrieren, wenn dort entsprechende Erweiterungen oder eine klar definierte Schnittstelle vorgesehen würden. Dies ist genauso wenig erkennbar wie die Antwort auf die Frage, welche Bedeutung FIBEX behalten wird, wo AUTOSAR (Kap. 8) doch erheblich mächtigere Dateiformate definiert hat, aber eben doch nicht alle Informationen abdeckt, die in FIBEX festgehalten werden können.

Die Syntax der FIBEX-Beschreibungsdateien ist in einem frei verfügbaren XML-Schema allgemein spezifiziert, wobei die Einbeziehung herstellerspezifischer Erweiterungen ausdrücklich vorgesehen ist. Die wesentlichen Elemente der hierarchischen XML-Baumstruktur sind in Abb. 6.16 dargestellt. Im Hauptelement ELEMENTS werden einerseits die Struktur des Netzes, d. h. die verwendeten Bussysteme (CLUSTERS, CHANNELS) sowie Steuergeräte (ECUS, GATEWAYS), und andererseits die in diesem Netz ablaufende Kommunikation, d. h. die Botschaften (FRAMES) und deren Dateninhalt (SIGNALS), beschrieben. In der FIBEX-Nomenklatur bezeichnet der Begriff CLUSTER ein einzelnes Bussystem, das etwa bei FlexRay aus mehreren parallelen Kanälen (CHANNELS) bestehen kann, sowie den daran angeschlossenen Steuergeräten (ECUS), von denen einige auch die Funktion eines GATEWAYS zu einem der anderen Bussysteme übernehmen können. Mit FIBEX 3.0 wurden als neues Element sogenannte *Protocol Data Units* PDUS als Zwischenschicht zwischen FRAMES und SIGNALS eingeführt, um die zentralen Datenelemente des AUTOSAR-Protokollstapels (vgl. Kap. 8) abzubilden. Da FIBEX 2.0 aber bereits eingesetzt wurde und die Änderungen nicht vollständig kompatibel sind, wird zunächst das ältere Format beschrieben und anschließend auf die Änderungen eingegangen.

Als Beispiel für eine FIBEX-Beschreibung wird das in Abb. 6.17 dargestellte System betrachtet, bei dem zwei Busse über ein Gateway-Steuengerät miteinander verbunden sind. Dabei soll das linke Steuergerät über das Gateway eine Botschaft an das rechte Steuergerät senden. Bei beiden Bussen wird CAN eingesetzt. Jedes Bussystem hat jeweils einen Kanal. Die Bussysteme sowie die daran angeschlossenen Steuergeräte werden in der FIBEX-Beschreibung (Tab. 6.4) als CLUSTER bezeichnet (hier clMotorCAN {1} bzw. clKarosse-

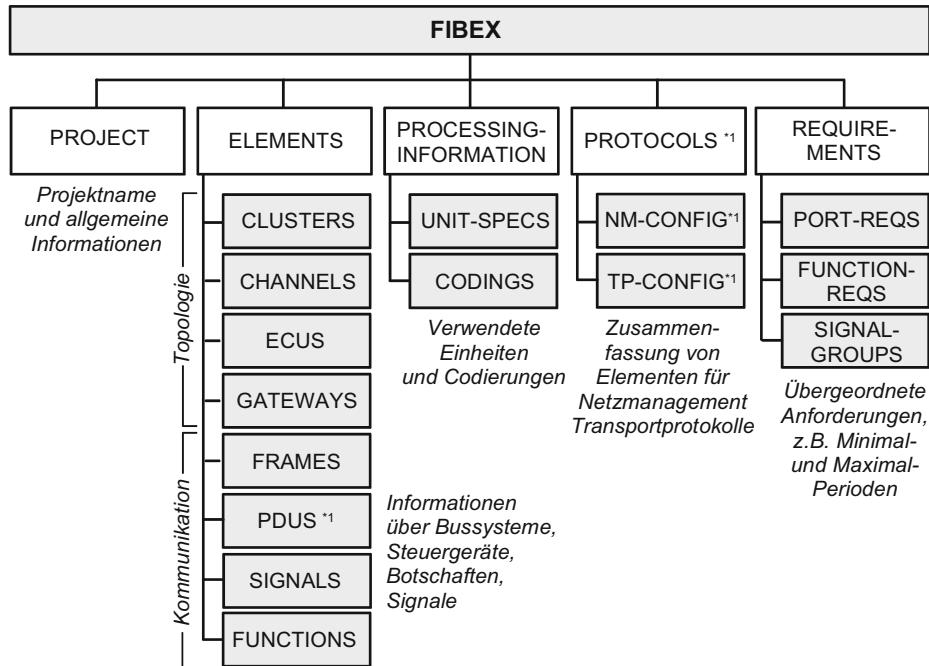


Abb. 6.16 Hauptelemente einer FIBEX-Beschreibungsdatei; ^{*1} neu seit FIBEX 3.0

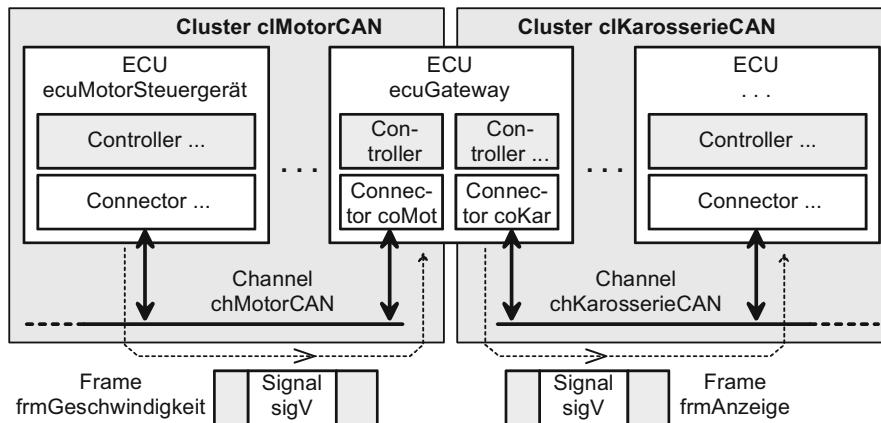


Abb. 6.17 Beispielsystem mit zwei über ein Gateway gekoppelten CAN-Bussystemen

rieCAN {2}). Die hier angegebenen Nummern {1}, {2} usw. beziehen sich auf die entsprechenden Stellen in den XML-Dateien.

Im CLUSTER-Abschnitt wird unter PROTOCOL {3} der Typ des Bussystems und unter SPEED dessen Bitrate aufgelistet. Unter CHANNEL-REFS {4} wird auf den zugehö-

Tab. 6.4 FIBEX-Beschreibungsdatei des Systems nach Abb. 6.17 – Teil 1

Beschreibung der Netzstruktur (CLUSTERS, CHANNELS)

```

<?xml version="1.0"?>
<FIBEX xmlns="http://www.asam.net/xml/fbx" . . . VERSION="2.0.0b">
  <fx:PROJECT>
    <ho:SHORT-NAME>CAN_Beispiel</ho:SHORT-NAME>
    <ho:DESC>Beispielprojekt . . . </ho:DESC>
  </fx:PROJECT>
  <fx:ELEMENTS>
    <fx:CLUSTERS>
      <fx:CLUSTER ID="clMotorCAN"> {1}
        <ho:SHORT-NAME>Motorbus</ho:SHORT-NAME>
        <ho:DESC>Motorseitiger High Speed CAN Bus</ho:DESC>
        <fx:PROTOCOL xsi:type="can:PROTOCOL-TYPE">CAN</PROTOCOL> {3}
        <fx:SPEED>500000</fx:SPEED>
        . . .
        <fx:CHANNEL-REFS>
          <fx:CHANNEL-REF ID-REF="chMotorCAN"/> {4}
        </fx:CHANNEL-REFS>
      </fx:CLUSTER>
      <fx:CLUSTER ID="clKarosserieCAN"> {2}
        <ho:SHORT-NAME>Karosseriebus</ho:SHORT-NAME>
        . . .
      </fx:CLUSTER>
    </fx:CLUSTERS>
    <fx:CHANNELS>
      <fx:CHANNEL ID="chMotorCAN"> {5}
        . . .
        <fx:FRAME-TRIGGERINGS>
          <fx:FRAME-TRIGGERING ID="ftgGeschwindigkeit">
            <fx:IDENTIFIER>
              <fx:IDENTIFIER-VALUE>125</fx:IDENTIFIER-VALUE>
            </fx:IDENTIFIER>
            <fx:FRAME-REF ID-REF="frmGeschwindigkeit"/> {7}
          </fx:FRAME-TRIGGERING>
        </fx:FRAME-TRIGGERINGS>
      </fx:CHANNEL>
      <fx:CHANNEL ID="chKarosserieCAN"> {6}
        . . .
        <fx:FRAME-TRIGGERINGS>
          <fx:FRAME-TRIGGERING ID="ftgAnzeige">
            <fx:IDENTIFIER>
              <fx:IDENTIFIER-VALUE>32</fx:IDENTIFIER-VALUE>
            </fx:IDENTIFIER>
            <fx:FRAME-REF ID-REF="frmAnzeige"/> {8}
          </fx:FRAME-TRIGGERING>
        </fx:FRAME-TRIGGERINGS>
      </fx:CHANNEL>
    </fx:CHANNELS>
  </FIBEX>

```

riegen CHANNEL (hier chMotorCAN {5} bzw. chKarosserieCAN {6}) verwiesen. Unter CHANNEL wird aufgeführt, welche Botschaften über den jeweiligen Bus übertragen werden (FRAME-REF, hier frmGeschwindigkeit {7} bzw. frmAnzeige {8}) und welche CAN-Identifier verwendet werden.

Die Steuergeräte werden im Abschnitt ECUS aufgelistet (Tab. 6.5). Jedes Steuergerät {9} hat einen oder mehrere *Stecker* (CONNECTOR) {10, 11}, die unter CHANNEL-REF {12, 13} auf das Bussystem verweisen, an das das Gerät über diesen *Stecker* angeschlossen ist. Hier wird unter OUTPUTS – FRAME-TRIGGERING-REF bzw. INPUTS – SIGNAL-INSTANCE-REF auch angegeben, welche Botschaften von diesem Steuergerät gesendet bzw. empfangen werden. Im Beispielsystem empfängt das Gateway-Steuergerät das Signal sigV {14} vom Motor-CAN (das Teil der Botschaft frmGeschwindigkeit ist, siehe unten) und sendet eine Botschaft ftgAnzeige {15}.

Anstatt auf die Botschaften im CONNECTOR-Abschnitt direkt zu verweisen, ist alternativ ein hier nicht verwendeter FUNCTION-Abschnitt möglich. Im Unterabschnitt CONTROLLER {16} können der Typ des eingesetzten Kommunikationscontrollers und dessen Konfigurationsparameter aufgeführt werden. Steuergeräte, die eine Gateway-Funktion zwischen verschiedenen Bussystemen haben, werden im Abschnitt GATEWAY unter ECU-REF referenziert {17}.

Die Details der Botschaften werden im Abschnitt FRAME beschrieben (Tab. 6.6). Hier wird angegeben, wie lange eine Botschaft ist (BYTE-LENGTH), aus welchen Signalen sie sich zusammensetzt (SIGNAL-INSTANCE, SIGNAL-REF) und an welcher Bitposition (BIT-POSITION) innerhalb der Botschaft die Signale jeweils beginnen. Die Botschaft frmGeschwindigkeit {18} im Beispiel ist zwei Byte lang {19} und enthält ab Bitposition 0 {20} in Little-Endian-Reihenfolge das Signal sigV {21, 22} mit der Codierung CODING-speed {23}.

Diese Codierung wird im Abschnitt CODING {24} beschrieben (Tab. 6.7), wobei im Beispiel als Einheit (UNIT-REF {25} und UNIT-SPEC {26}) für die Fahrgeschwindigkeit km/h verwendet wird und ein linearer Zusammenhang {27} zwischen dem auf dem Bus übertragenen 16 bit-Wert und dem physikalischen Wert mit 0 als kleinstem und 400 km/h als größtem Wert festgelegt wird {28}.

Wie man bereits an diesem einfachen Beispiel sieht, ist die fehlerfreie Erstellung und Pflege einer derartigen Konfigurationsdatei ohne die Hilfe eines geeigneten Werkzeuges, das entsprechende Eingabemasken bereitstellt, die Struktur des Bussystems sowie die Verweise zwischen den einzelnen Elementen visualisiert und die Konsistenz der Daten sicherstellt, nahezu unmöglich.

Seit Version 2.0 des FIBEX-Standards wird erstmals ein XML-Schema für Time-Triggered CAN bereitgestellt. Als wesentliche Erweiterung wird nun auch MOST offiziell unterstützt. Die Dienst-orientierte Struktur der MOST *Network Services* (siehe Abschn. 3.4) mit ihrer Objekthierarchie zwang zu einigen Erweiterungen des bisherigen Datenschemas im Vergleich zu der Signal-orientierten Beschreibung von CAN, LIN und FlexRay. Ein MOST-Funktionsblock wird in FIBEX als FBLOCK abgebildet, der die zugeordneten MOST-Funktionen als FIBEX-Function enthält. Die von MOST bekannten Kennziffern

Tab. 6.5 FIBEX-Beschreibungsdatei des Systems nach Abb. 6.17 – Teil 2

Beschreibung der Steuergeräte (ECUS, GATEWAYS)

```

<fx:ECUS>
  <fx:ECU ID="ecuMotorSteuergerät">
    .
    .
    <fx:CONNECTORS>
      <fx:CONNECTOR ID=". . .">
        <fx:CHANNEL-REF ID-REF="chMotorCAN"/>
        <fx:OUTPUTS ID=". . .">
          <fx:FRAME-TRIGGERING-REF ID-REF="ftgGeschwindigkeit"/>
        </fx:OUTPUTS>
      </fx:CONNECTOR>
    </fx:CONNECTORS>

    <fx:CONTROLLERS>
      <fx:CONTROLLER xsi:type="can:CONTROLLER-TYPE" ID=". . .">
        .
        .
        <fx:CHIP-NAME>SJA1000</fx:CHIP-NAME>
        .
        .
      </fx:CONTROLLER>
    </fx:CONTROLLERS>
  </fx:ECU>

  <fx:ECU ID="ecuGateway"> {9}
    .
    .
    <fx:CONNECTORS>
      <fx:CONNECTOR ID="coMot"> {10}
        <fx:CHANNEL-REF ID-REF="chMotorCAN"/> {12}
        <fx:INPUTS>
          <fx:INPUT-PORT ID=". . ."> {14}
            <fx:SIGNAL-INSTANCE-REF ID-REF="Instance-sigV"/>
          </fx:INPUT-PORT>
        </fx:INPUTS>
      </fx:CONNECTOR>
      <fx:CONNECTOR ID="coKar"> {11}
        <fx:CHANNEL-REF ID-REF="chKarosserieCAN"/> {13}
        <fx:OUTPUTS ID=". . .">
          <fx:FRAME-TRIGGERING-REF ID-REF="ftgAnzeige"/> {15}
        </fx:OUTPUTS>
      </fx:CONNECTOR>
    </fx:CONNECTORS>
  </fx:ECU>
</fx:ECUS>

<fx:GATEWAYS>
  <fx:GATEWAY ID="gGateway">
    .
    .
    <fx:ECU-REF ID-REF="ecuGateway"/> {17}
  </fx:GATEWAY>
</fx:GATEWAYS>

```

Tab. 6.6 FIBEX-Beschreibungsdatei des Systems nach Abb. 6.17 – Teil 3

Beschreibung der Botschaften (FRAMES, SIGNALS)	
<fx:FRAMES>	
<fx:FRAME ID="frmGeschwindigkeit">	{18}
<ho:SHORT-NAME>Fahrgeschwindigkeit</ho:SHORT-NAME>	
<fx:BYTE-LENGTH>2</fx:BYTE-LENGTH>	{19}
<fx:FRAME-TYPE>APPLICATION</fx:FRAME-TYPE>	
<fx: SIGNAL-INSTANCES>	
<fx: SIGNAL-INSTANCE ID="Instance-sigV">	
<fx:BIT-POSITION>0</fx:BIT-POSITION>	{20}
<fx:IS-HIGH-LOW-BYTE-ORDER>true</IS-HIGH-LOW-BYTE-ORDER>	
<fx: SIGNAL-REF ID-REF="sigV"/>	{21}
</fx: SIGNAL-INSTANCE>	
</fx: SIGNAL-INSTANCES>	
</fx: FRAME>	
<fx:FRAME ID="frmAnzeige">	
. . .	
</fx: FRAME>	
</fx:FRAMES>	
<fx: SIGNALS>	
<fx: SIGNAL ID="sigV">	{22}
. . .	
<fx: CODING-REF ID-REF="CODING-speed"/>	{23}
</fx: SIGNAL>	
</fx: SIGNALS>	
</fx: ELEMENTS>	

FB1ckID, FktID usw. tauchen in der FIBEX-Beschreibung als XML-Elemente wieder auf. Der MOST OPType, der präzise angibt, welche MOST-Funktion im Detail ausgeführt wird und letztlich auf eine MOST-Steuerdaten-Botschaft abgebildet wird, referenziert schließlich einen FIBEX Frame, d. h. die bekannte Datenstruktur für Busbotschaften. Zur Darstellung der Parameter der MOST-Funktionen, d. h. der „Signale“ in den MOST-Botschaften, wurden einige neue Typen eingeführt, unter anderem Array und Stream.

Zu den weiteren Änderungen ab Version 2.0 gehört die durchgängige Verwendung von XML-Namensräumen, wobei einige bisherige Namensräume umbenannt wurden. Für die XML-Beschreibung von Gateways wurden einige Datenelemente umorganisiert und das Mapping von Signalen zwischen den verschiedenen Bussystemen in der Dokumentation ausführlicher dargestellt. Darüber hinaus wurde die Darstellung aller Datenstrukturen in der Dokumentation vollständig auf UML-Diagramme umgestellt und damit an die Form der ODX-Dokumentation angepasst.

Die Austauschbarkeit der Dateninhalte mit AUTOSAR war Hauptziel der FIBEX Version 3.0. Dazu wurden die *Protocol Data Units* Elemente als Zwischenschicht zwischen FRAMES und SIGNALS eingeführt. Diese Änderung ist nicht vollständig rückwärts kompatibel zur Version 2.0. Eine FIBEX PDU entspricht der *Interaction Layer I-PDU* der AUTOSAR COM Schicht, während ein FIBEX FRAME praktisch der AUTOSAR *Data Link*

Tab. 6.7 FIBEX-Beschreibungsdatei des Systems nach Abb. 6.17 – Teil 4

```

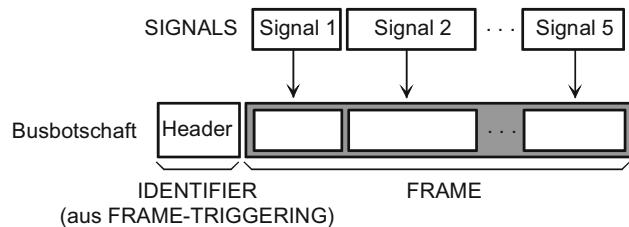
Beschreibung der Signal-Einheiten und -Codierung (UNITS, CODINGS)

<fx:PROCESSING-INFORMATION xmlns="http://www.asam.net/xml">
  <ho:UNIT-SPEC>
    <ho:PHYSICAL-DIMENSIONS>
      <ho:PHYSICAL-DIMENSION ID="UNIT-speed"> {26}
        <ho:SHORT-NAME>speed</ho:SHORT-NAME>
        <ho:DESC>Fahrgeschwindigkeit in km/h</ho:DESC>
      </ho:PHYSICAL-DIMENSION>
    </ho:PHYSICAL-DIMENSIONS>
  </ho:UNIT-SPEC>
  <fx:CODINGS>
    <fx:CODING ID="CODING-speed"> {24}
      .
      .
      <ho:CODED-TYPE ho:BASE-DATA-TYPE="A_UINT16" ... ENCODING="UNSIGNED">
        <ho:BIT-LENGTH>16</ho:BIT-LENGTH>
      </ho:CODED-TYPE>
      <ho:COMPU-METHOD>
        .
        .
        <ho:UNIT-REF ID-REF="UNIT-speed"/> {25}
        <ho:CATEGORY>LINEAR</ho:CATEGORY> {27}
        <ho:COMPU-INTERNAL-TO-PHYS>
          <ho:COMPU-SCALES>
            <ho:COMPU-SCALE>
              <ho:LOWER-LIMIT INTERVAL-TYPE="CLOSED">0 {28}
              </ho:LOWER-LIMIT>
              <ho:UPPER-LIMIT INTERVAL-TYPE="CLOSED">400
              </ho:UPPER-LIMIT>
            <ho:COMPU-CONST>
              <ho:V>1.0</ho:V>
            </ho:COMPU-CONST>
            <ho:COMPU-SCALES>
              <ho:COMPU-DEFAULT-VALUE>
                <ho:V>0.0</ho:V>
              </ho:COMPU-DEFAULT-VALUE>
            </ho:COMPU-INTERNAL-TO-PHYS>
          </ho:COMPU-METHOD>
        </fx:CODING>
      </fx:CODINGS>
    </fx:PROCESSING-INFORMATION>
  </fx:FIBEX>

```

Layer L-PDU entspricht, also der tatsächlich versendeten Busbotschaft (siehe Abschn. 8.4, insbesondere Abb. 8.12). Die I-PDU besteht aus einem oder mehreren Signalen, d. h. Steuergeräterewerten wie z. B. der Motordrehzahl, der Kühlwassertemperatur oder der Fahrgeschwindigkeit aus dem obigen FIBEX-Beispiel. Bei CAN oder LIN, deren Busbotschaften maximal 8 Byte Nutzdaten enthalten können, kann eine I-PDU nur relativ wenige Signale enthalten. Dort wird die I-PDU praktisch direkt auf den Nutzdatenbereich der eigentlichen

Abb. 6.18 Beschreibung einer Busbotschaft ab FIBEX 2.0



Busbotschaft, des FRAMES, abgebildet. Eine Zwischenebene war nicht wirklich notwendig. In dieser Weise war die Beschreibung bei FIBEX 2.0 aufgebaut (Abb. 6.18).

Bei FlexRay dagegen, dessen Botschaften bis zu 254 Byte lang sein können, werden aus Effizienzgründen möglicherweise mehrere I-PDUs im Nutzdatenbereich einer einzelnen Busbotschaft, einem FRAME, zusammengefasst. Um auf den oberen Ebenen weitgehend unabhängig vom eigentlichen Bussystem zu sein, wird die Abbildung von Signalen auf I-PDUs und von dort aus dann auf den Nutzdatenbereich von Busbotschaften bei AUTOSAR durchgängig bei allen Bussystemen verwendet. Dieses Konzept wurde ab FIBEX 3.0 übernommen (Abb. 6.19).

Die im obigen Beispiel notwendigen Änderungen zur Anpassung an FIBEX 3.0 sind überschaubar (Tab. 6.8). Im Wesentlichen wird im FRAMES-Abschnitt aus Tab. 6.6 der Begriff FRAME durch PDU ersetzt {29} und im neuen FRAMES-Abschnitt mit derselben Syntax auf die PDUs statt auf die Signale verwiesen {30}.

Das wahlweise Multiplexen von PDUs im selben Frame wird über die neuen Elemente PDU-MULTIPLEXING und das Zeitverhalten von PDUs über PDU-TRIGGERING und I-TIMING beschrieben, die im vorliegenden Beispiel nicht verwendet wurden.

Bei FIBEX-Version 3.1 wurde das Beschreibungsschema für LIN überarbeitet. Hauptneuerung aber war die Aufnahme weiterer AUTOSAR-Elemente. Damit können nun auch das Transportprotokoll AUTOSAR TP und das Netzmanagement AUTOSAR NM in FIBEX beschrieben werden.

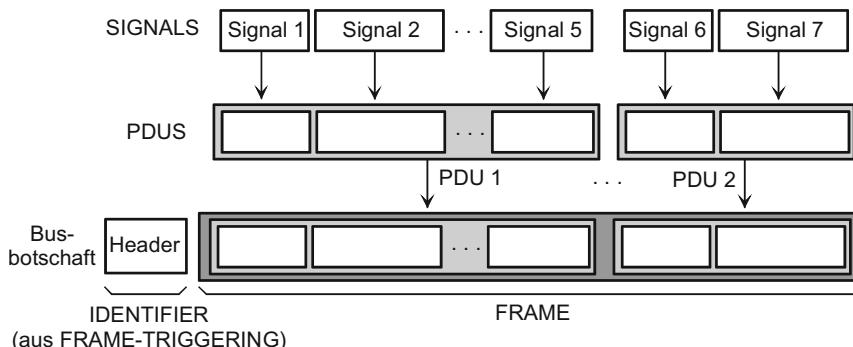


Abb. 6.19 Erweiterte Beschreibung einer Busbotschaft ab FIBEX 3.x

Tab. 6.8 Prinzipiell notwendige Anpassung des Beispiels an FIBEX 3.x

```

<FIBEX xmlns="http://www.asam.net/xml/fbx" . . . VERSION="3.0.0">
  . . .
  <fx:ELEMENTS>
    . . .
    <fx:PDUS> entspricht dem alten FRAME-Abschnitt aus Tabelle 6.3.3 {29}
      <fx:PDU ID="pduGeschwindigkeit">
        <ho:SHORT-NAME>Fahrgeschwindigkeit</ho:SHORT-NAME>
        <fx:BYTE-LENGTH>2</fx:BYTE-LENGTH>
        <fx:PDU-TYPE>APPLICATION</fx:PDU-TYPE>
        <fx:SIGNAL-INSTANCES>
          <fx:SIGNAL-INSTANCE ID="Instance-sigV">
            <fx:SIGNAL-REF ID-REF="sigV"/>
          . . .
        </fx:SIGNAL-INSTANCES>
      </fx:PDU>
    </fx:PDUS>

    <fx:FRAMES> neuer FRAME-Abschnitt, verweist auf PDUs {30}
      <fx:FRAME ID="frmGeschwindigkeit">
        <ho:SHORT-NAME>Fahrgeschwindigkeit</ho:SHORT-NAME>
        <fx:BYTE-LENGTH>2</fx:BYTE-LENGTH>
        <fx:FRAME-TYPE>APPLICATION</fx:FRAME-TYPE>
        <fx:PDU-INSTANCES>
          <fx:PDU-INSTANCE ID=" . . . ">
            <fx:PDU-REF ID-REF="pduGeschwindigkeit"/>
            <fx:BIT-POSITION>0</fx:BIT-POSITION>
            <fx:IS-HIGH-LOW-BYTE-ORDER> . . .
          </fx:PDU-INSTANCE>
        </fx:PDU-INSTANCES>
      </fx:FRAME>
    </fx:FRAMES>
  <fx:ELEMENTS>

```

Die Grundstruktur der FIBEX Beschreibung des in Abschn. 4.2 vorgestellten Transportprotokolls für FlexRay ist in Abb. 6.20 dargestellt. Im Abschnitt TP-CONFIG werden die FlexRay-Kanäle sowie die angeschlossenen Steuergeräte als *Knoten* definiert. Die für das Transportprotokoll vorgesehenen Botschaften im statischen bzw. dynamischen Segment werden über TP-PDU-USAGE bzw. bei Flow Control Botschaften über TP-FC-PDU-USAGE referenziert und als normale PDUs beschrieben.

Das AUTOSAR Netzmanagement, das in Abschn. 8.5 dargestellt wird, kann mittels der FIBEX Struktur nach Abb. 6.21 beschrieben werden. Die am Netzmanagement eines einzelnen Bussystems beteiligten Netzknoten werden im Abschnitt NM-CLUSTERS definiert, der über NM-PDU-USAGE auf die Beschreibung der zugehörigen PDUs verweist. Für Gateways sind die Abschnitte NM-CLUSTER-COUPLING bzw. NM-COORDINATOR vorgesehen, falls das Netzmanagement übergreifend über mehrere Bussysteme erfolgt und koordiniert wird.

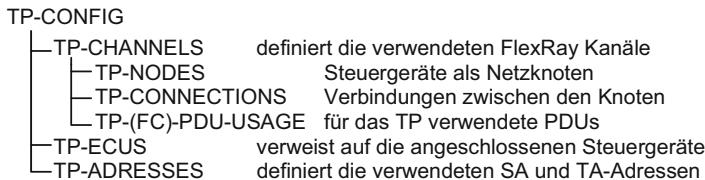


Abb. 6.20 Struktur der AUTOSAR TP Beschreibung in FIBEX

In zukünftigen FIBEX-Versionen ist mit der Integration weiterer AUTOSAR-kompatibler Beschreibungselemente zu rechnen. Um die Aufwärtskompatibilität neuer Systeme, die mit AUTOSAR-Methoden und Werkzeugen (siehe Kap. 8) entworfen werden, zu vorhandenen Werkzeugen für die Entwicklung und Analyse der Buskommunikation zu gewährleisten, stellen die Werkzeughersteller Konvertierungsprogramme bereit, mit denen die AUTOSAR XML-Beschreibungsformate in etablierte Datenformate wie FIBEX, CANdb oder LDF umgewandelt werden können. Da in diesen Formaten aber einige semantische Elemente fehlen, die in AUTOSAR enthalten sind, müssen die älteren Formate erweitert, damit keine Definitionen verloren gehen und die Umwandlung gegebenenfalls auch wieder in umgekehrter Richtung möglich ist.

6.4 Überblick über ASAM AE MCD 2 und MCD 3

ASAM AE MCD 3 definiert die Schnittstelle zu übergeordneten Mess-, Kalibrier- und Diagnosewerkzeugen, die diese sowohl von den Details der darunterliegenden Kommunikationsprotokolle und Bussysteme als auch von der kompletten Datenhaltung für die zugehörigen Beschreibungsdaten entkoppelt (vgl. Abb. 6.2). Das Format der Beschreibungsdateien ist in den ASAM MCD 2 Spezifikationen vorgegeben.

Die Grundidee besteht darin, Diagnosetester und Mess- und Kalibrierwerkzeuge nicht für jedes Fahrzeug und Steuergerät neu zu programmieren, sondern ein universelles Programm, den sogenannten MCD 3 Server (Abb. 6.22) einzusetzen, der sich selbstständig über MCD 2-Beschreibungsdateien auf das jeweilige Fahrzeug und seine Steuergeräte parametriert und konfiguriert (*datengetriebener Tester*).

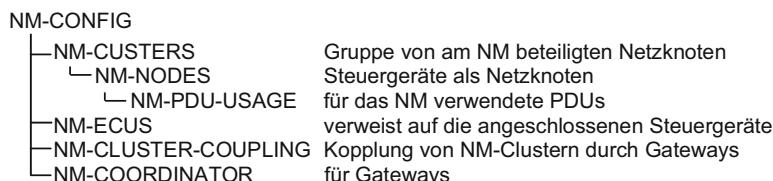


Abb. 6.21 Struktur der AUTOSAR NM Beschreibung in FIBEX

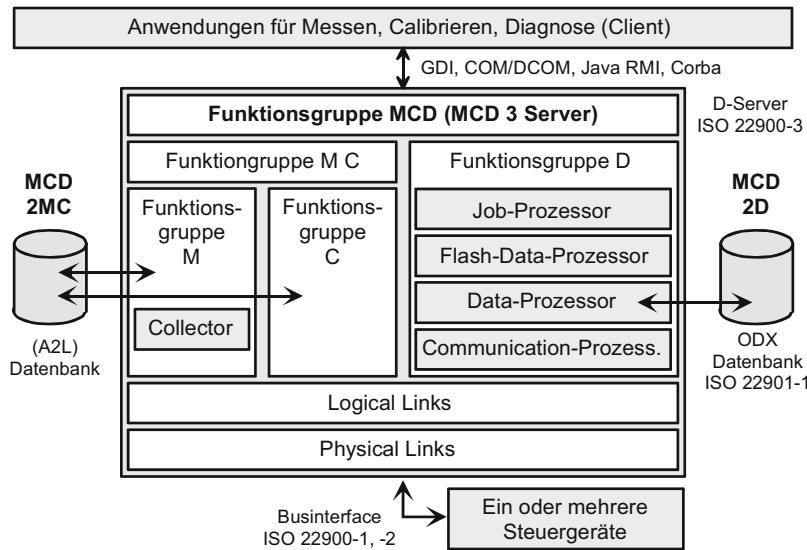


Abb. 6.22 Aufbau eines ASAM-Systems

Der für den Benutzer sichtbare Teil des Systems, bei ASAM als Client bezeichnet, enthält die Mess-, Kalibrier- und/oder Diagnoseanwendung mit der Bedienoberfläche, wobei mehrere solcher Clients gleichzeitig aktiv sein können. Der Client selbst ist in ASAM MCD 3 ausdrücklich nicht spezifiziert, um auf den individuellen Einsatzfall zugeschnittene Lösungen zu erlauben und den Wettbewerb zwischen den Werkzeugherstellern nicht unnötig einzuschränken. Die Schnittstelle zum Server dagegen und die Struktur des Servers, des ASAM MCD 3 – Laufzeitsystems, wird detailliert beschrieben. Aber auch wenn die geforderte Funktionalität an der Schnittstelle durch MCD 3 exakt vorgegeben ist, darf die Kommunikation zwischen Client und Server von den Herstellern mit verschiedenen Programmiersprachen und Standard-Technologien implementiert werden. Die MCD 3 Spezifikation erwähnt Java und Java Remote Method Invocation (RMI), das meist in Verbindung mit C/C++ und neuerdings C# eingesetzte Microsoft COM/DCOM (Active X), das aus der UNIX-Welt stammende Corba und das ASAM-eigene GDI als Implementierungsvarianten. Trotz formaler MCD 3 Kompatibilität werden Client und Server verschiedener Hersteller daher nur dann zusammenarbeiten, wenn sie auch für die Implementierung der Schnittstelle dieselbe Technologie verwenden.

Wie in der Einführung zu diesem Kapitel bereits dargestellt, existieren im Bereich Messen-Kalibrieren einerseits und Diagnose andererseits eine große Anzahl von unterschiedlichen, zueinander inkompatiblen Standards und Datenformaten, die seit Langem etabliert sind und vermutlich noch lange koexistieren werden. Diese Unterschiede versucht man durch die in Version 2 neu konzipierte, einheitlich objektorientierte MCD 3 Schnittstelle zu verdecken. Bei den bis Anfang der 90er Jahre zurückreichenden, aber in realen

Produkten nur selten eingesetzten Vorgängerversionen waren die MC- und die D-Welt noch streng getrennt und die ältere MC-Schnittstelle war prozedural definiert. Völlig durchgängig ist die Integration auch weiterhin nicht, die Funktionsgruppen M, C und D werden immer noch unterschieden (Abb. 6.22) und die Datenformate für die Datenablage von MC-Informationen einerseits und D-Informationen andererseits sind historisch bedingt unterschiedlich. Im MC-Bereich wird das im folgenden Abschn. 6.5 beschriebene ASAP2-Format (A2L, AML) verwendet, im D-Bereich ODX (Abschn. 6.6).

Die objektorientierte Schnittstelle verwendet für beide Teilbereiche dieselben Konzepte und versucht, die Details der Datenformate zu kaschieren, so dass für den Anwender nur dort Unterschiede sichtbar werden, wo dies durch die unterschiedliche Aufgabenstellung bedingt ist. Insgesamt ist ASAM MCD 3 allerdings sehr komplex. Die Spezifikation allein umfasst über 1600 Seiten, die Beschreibung der ASAP2- und ODX-Datenformate benötigt weitere 700 Seiten. ASAM MCD 2 und MCD 3-kompatible Systeme kommen daher nur schrittweise auf den Markt und implementieren in der Regel zunächst nur Teilmengen der gesamten Spezifikation. Die Spezifikation lässt ausdrücklich zu, dass nur einer der Funktionsblöcke M, C oder D oder eine Kombination davon implementiert wird, wobei die Implementierung eines Funktionsblocks aber in jedem Fall vollständig sein muss. Ein Beispiel dafür wird in Abschn. 9.7 vorgestellt. Angesichts der Komplexität ist auf den folgenden Seiten nur ein grober, stark vereinfachter Überblick möglich. Dabei sollen zunächst die beiden Datenformate für den Bereich MC und für den Bereich D beschrieben werden, bevor anschließend auf Interna der MCD 3-Schicht eingegangen wird.

6.5 Applikationsdatensätze nach ASAM MCD 2 MC

6.5.1 ASAP2/A2 L-Applikationsdatensätze

Mess- und Kalibrieraufgaben fallen hauptsächlich während der Entwicklung von Geräten, Aggregaten und des Gesamtfahrzeuges an. Hierbei ist der möglichst flexible Zugriff auf alle Steuergeräteinternen Daten und leichte Verstellbarkeit wichtig.

Die Beschreibung, welche internen Größen des Steuergerätes messbar und welche Parameter veränderbar sind (vgl. Abb. 6.2), erfolgt mit derselben Syntax, die bereits für die Beschreibung der XCP- bzw. CCP-Schnittstelle zwischen Applikationssystem und Steuergerät verwendet wird und im Abschn. 6.2.3 erörtert wurde. Die Grundstruktur der Beschreibung vollständiger Applikationsdatensätze, häufig nach dem früheren Namen von ASAM als ASAP2-Format bezeichnet, ist in Abb. 6.23 dargestellt.

Ein Projekt kann ein einzelnes Steuergerät oder ein vollständiges Fahrzeug umfassen (Tab. 6.9). Da ein Fahrzeugprojekt in der Regel mehrere Geräte enthalten wird, kann die Beschreibung der einzelnen Steuergeräte (MODULE) in separaten Dateien erfolgen, die mit Hilfe von/include-Anweisungen in die Projektdatei eingebunden werden. In derselben Weise wird meist auch die schon aus Abschn. 6.2.3 bekannte Beschreibung der Kommu-

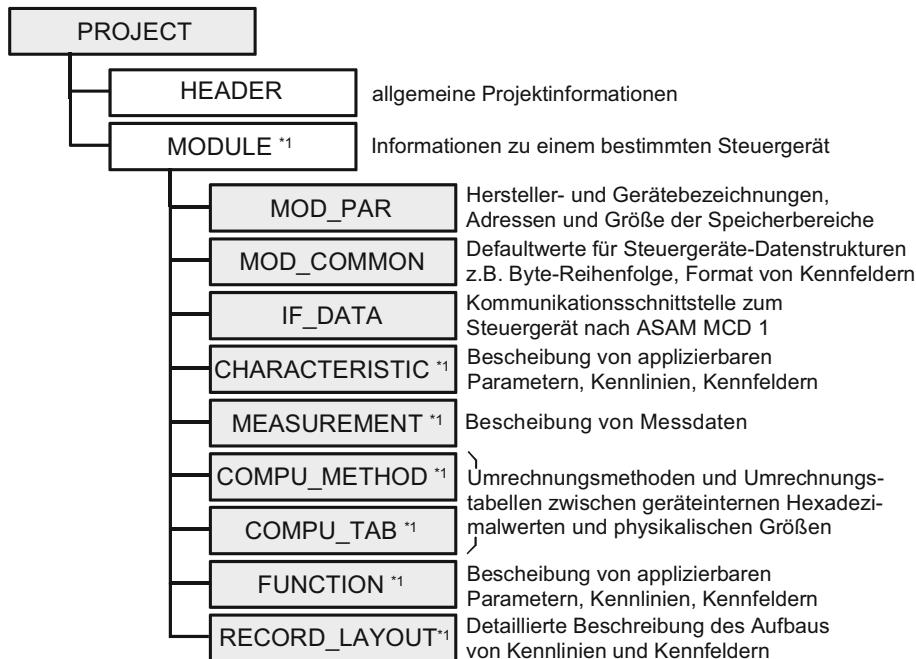


Abb. 6.23 Struktur einer ASAM MCD 2MC-Beschreibung; ^{*1} Abschnitte dürfen mehrfach vorkommen

nikationsschnittstelle IF_DATA zum Steuergerät nach ASAM MCD 1 eingebunden (vgl. Tab. 6.2).

Der Abschnitt MOD_PAR enthält allgemeine Informationen über das Steuergerät wie den Hersteller (SUPPLIER, CUSTOMER, USER), Gerätetyp (ECU) und Versionsnummer (VERSION), Typ der Steuergeräte-CPU (CPU_TYPE), Anzahl der Kommunikations-schnittstellen, Anfangsadressen und Größen der Speicherbereiche des Steuergerätepro-gramms und der Applikationsdaten (MEMORY_LAYOUT), globale Konstanten (SYS-TEM_CONSTANT) sowie herstellerabhängige Parameter für die Applikationsschnittstelle (CALIBRATION_METHOD).

Im folgenden Abschnitt MOD_COMMON finden sich Hinweise zu steuergeräteinter-nen Datenstrukturen wie der Byte-Reihenfolge (BYTE_ORDER), der Standard-Daten-wortbreite und der Datenausrichtung (DATA_SIZE, ALIGNMENT,...) und dem Stan-dardaufbau von Tabellen für Kennlinien und Kennfelder (DEPOSIT, S_REC_LAYOUT).

Für jede applizierbare Datenstruktur ist ein CHARACTERISTIC-Abschnitt vorhanden, der einen Klartext-Bezeichner für die Datenstruktur, die Adresse sowie den Aufbau der Struktur angibt. Dabei kann es sich um Einzelwerte, Textstrings, Arrays, Kennlinien oder Kennfelder handeln. Für Details zum Aufbau von Kennlinien und Kennfeldern wird auf den entsprechenden RECORD_LAYOUT-Eintrag verwiesen. Zusätzlich können in einge-

Tab. 6.9 Auszug aus einer ASAP2-Konfigurationsdatei

```

/begin PROJECT XCP
. . .
/include "ABS_ECU.a2l" /* Einbinden eines weiteren Steuergerätes*/
. . .
/begin MODULE           /* Beschreibung eines Steuergerätes */
  /begin MOD_PAR
    SUPPLIER "Muster AG"
    CUSTOMER "ToyCarProductions"
    ECU „Electronic Diesel Engine Management EDC 24“
    CPU_TYPE "FreeIntelfineon CoreMpcTri"
  . . .
  /end MOD_PAR
. . .
/begin MOD_COMMON  Allgemeines_Datenformat
. . .
  BYTE_ORDER BIG_ENDIAN
  DATA_SIZE 8           /* Standard-Datengröße 8 bit */
  ALIGNMENT_BYTE        /* Datenausrichtung an Byte-Grenzen */
. . .
/end MOD_COMMON
. . .
/begin CHARACTERISTIC Maximale_Einspritzmenge
. . .
  VALUE                /* Typ Konstante */
  0x7140               /* Adresse im Steuergerätespeicher */
. . .
  FUEL_QUANTITY        /* Verweis auf Umwandlungsvorschrift */
  0.0     80.0          /* Minimalwert, Maximalwert */
. . .
/end CHARACTERISTIC
. . .
/begin MEASUREMENT Motordrehzahl
. . .
  UWORLD                /* Datentyp 16bit unsigned */
  N_RULE                /* Verweis auf Umwandlungsvorschrift */
  4                     /* Auflösung in bit */
  1.0                  /* Genauigkeit in % */
  0.0     6000.0         /* Minimalwert, Maximalwert */
. . .
/begin FUNCTION_LIST N_CTRL INJECTION
/end FUNCTION_LIST /* Referenz auf Funktionsgruppen */
. . .
/end MEASUREMENT
. . .
/begin COMPU_METHOD N_RULE /* Umwandlung Hex-Wert -> phys.Wert*/
. . .
  RAT_FUNC              /* Gebrochen rationale Funktion
    Y =  $\frac{a x^2 + b x + c}{d x^2 + e x + f}$  y...hex., x... phys*/
    "%4.0"               /* Formatierung für Bildschirmschirmdarstellung*/
    "1/min"               /* Dimension */
    COEFFS 0.0 255.0 0.0 /* Koeffizienten a,b,...,f der Funktion*/
    0.0 5800.0 0.0
/end COMPU_METHOD
. . .
/begin FUNCTION N_CTRL "speed control"
/end FUNCTION /* Funktionsgruppe Drehzahlregelung
/begin FUNCTION INJECTION "injection control"
/end FUNCTION /* Funktionsgruppe Einspritzsteuerung */
. . .
/end MODULE
/end PROJECT

```

schränkter Form auch Abhängigkeiten von anderen Datenstrukturen und Grenzwerte für den Verstellbereich angegeben werden.

Für jeden Messwert, d. h. alle internen Größen des Steuergerätes, die im Applikationswerkzeug angezeigt und aufgezeichnet werden können, gibt es einen MEASUREMENT-Abschnitt. Neben dem Datentyp und der Speicheradresse werden Unter- und Obergrenzen, Auflösung und Genauigkeit des Messwertes angegeben und auf eine Umrechnungsformel in einem COMPU_METHOD-Abschnitt verwiesen, mit der der steuergeräteinterne Hexadezimalwert in einen physikalischen Messwert umgerechnet werden kann. Umrechnungsmethoden können Formeln, Umrechnungstabellen oder, z. B. bei Bitwerten, eine Klartextbedeutung sein. Bei Umrechnungstabellen wird dabei auf einen entsprechenden COMPU_TAB-Eintrag verwiesen.

Über FUNCTION-Abschnitte können die applizierbaren Parameter bzw. Messdaten zu Funktionsgruppen zusammengefasst werden, indem z. B. eine Funktionsgruppe Leerlaufregler definiert und auf alle Parameter und Messdaten verwiesen wird, die für diese Funktionsgruppe relevant sind. Da Motorsteuerungen heute oft mehrere Tausend applizierbare Datenstrukturen und Messwerte besitzen, ist eine derartige Gruppierung für eine sinnvolle Benutzerführung im Applikationswerkzeug unabdingbar.

Bei praktisch jedem Abschnitt der Beschreibung können in einem Unterabschnitt IF_DATA herstellerspezifische Parameter für die ASAM 1-Schicht, d. h. für den eigentlichen Zugriff auf das Steuergerät definiert werden. Das Format der Parameter, z. B. Strings, Hexadezimalzahlen usw., hängt von der Aufgabenstellung ab. Aus Sicht von ASAM 2 handelt es sich dabei um Wertefolgen, die Binary Large Objects BLOB, die ohne weitere Interpretation an die darunterliegende Schicht 1 weitergegeben werden.

6.5.2 Calibration Data Format CDF und Meta Data Exchange MDX

Das XML-basierte, ebenfalls vom ASAM-Konsortium spezifizierte CDF ergänzt die AS-AP2/A2 L-Datensätze um übergeordnete Informationen und könnte es in Verbindung mit MDX langfristig vielleicht ablösen.

Eine CDF-Beschreibung (Abb. 6.24) enthält für jedes Steuergerät ein SW-SYSTEM-Element, unterhalb dessen die einzelnen Applikationsdaten als SW-INSTANCE-Knoten abgebildet werden. Dabei können alle im etablierten A2 L-Format vorhandenen Datentypen wie Skalare (VALUE), Strings (ASCII), Kennlinien (CURVE) oder Kennfelder (MAP) abgebildet werden. Neben dem eigentlichen Wert (SW-VALUES-PHYS) kann die physikalische Einheit (UNIT-DISPLAY-NAME) angegeben werden.

Im Vergleich zum A2 L-Format erlaubt es CDF nun, den Qualitätsstand (*Quality Meta Data*) der Applikationsdaten mit Hilfe von SW-CS-HISTORY Elementen zu beschreiben. Darin wird einem Datenelement das Kalenderdatum (DATE) der letzten Änderung, der Name des verantwortlichen Applikationsingenieurs (CSUS) sowie der Qualitätsstand (STATE) und weitere Informationen zugeordnet werden. Unter Qualitätsstand wird angegeben, ob das Datenelement noch gar nicht appliziert ist, gegenüber einem früheren Stand

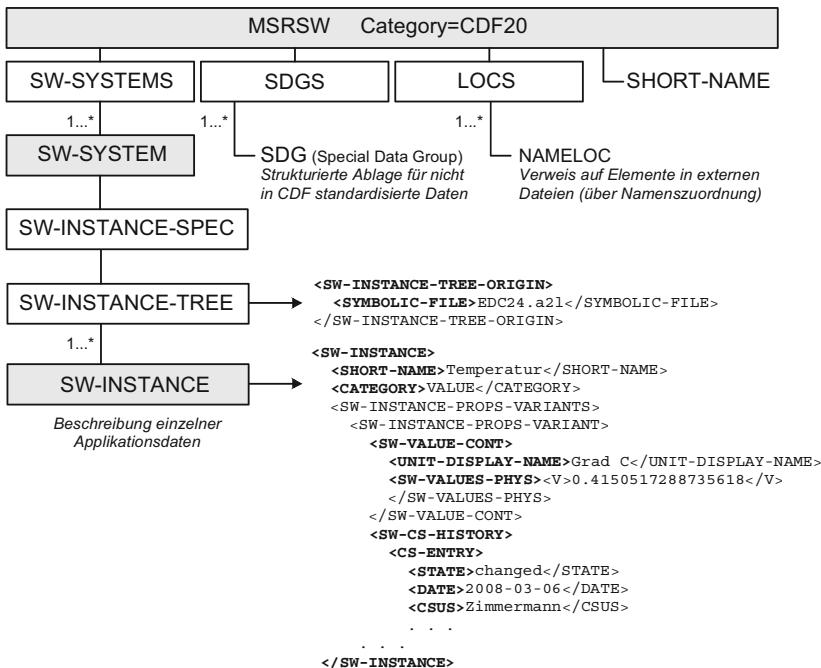


Abb. 6.24 Vereinfachte Grundstruktur einer CDF-Beschreibung

verändert (*changed*) wurde, ob der neue Wert nur vorläufig (*prelimCalibrated*) oder bereits endgültig festgelegt (*calibrated*) wurde und ob er bereits gegengeprüft (*checked*) bzw. für die Serie freigegeben (*completed*) wurde. Dieses Konzept wurde aus dem *Parameter Content (PaCo)* Standardentwurf des MSR-Konsortiums übernommen, das sich mittlerweile an ASAM angeschlossen hat.

CDF speichert grundsätzlich die physikalischen Werte der Applikationsdaten. Umrechnungsformeln zwischen den physikalischen und steuergeräteinternen Werten, wie die COMPU-METHOD in A2 L, oder die Speicheradresse, unter der die Daten im Steuergerät gespeichert sind, werden nicht abgebildet. In der derzeitigen Form kann CDF die bekannten A2 L-Datensätze für die Steuergeräte-Applikation deshalb nicht ersetzen, sondern nur ergänzen. Daher enthält die CDF-Datei unter SYMBOLIC-FILE in der Regel einen Verweis auf die zugehörige A2 L-Datei. Die Zuordnung setzt voraus, dass die Datenwerte in der A2 L- und in der CDF-Datei denselben Kurznamen (SHORT-NAME) verwenden oder dass mit Hilfe von NAMELOC Definitionen ein Namensmapping vorgenommen wird.

Die bei CDF fehlenden Informationen über die steuergeräteinternen Werte könnten statt aus der A2 L-Datei auch aus einer MDX-Datei bezogen werden. Das *Meta Data Exchange Format for Software Module Sharing* soll die Steuergerätesoftware so beschreiben, dass Softwaremodule zwischen verschiedenen Projekten oder zwischen Softwarezulieferer und Geräte- bzw. Fahrzeughersteller leichter ausgetauscht werden können. Das Format

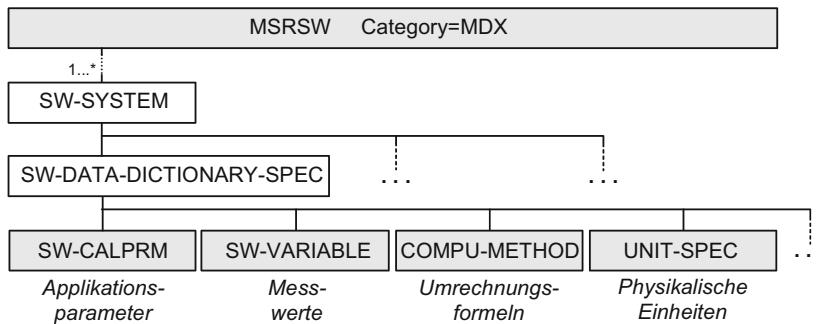


Abb. 6.25 Wichtige Elemente einer MDX-Beschreibung für Applikationsdatensätze

wurde im selben MSR-Konsortium definiert wie CDF und ähnelt diesem in der Struktur sehr stark (Abb. 6.25). Ob sich MDX für die Beschreibung von Softwaremodulen auf dem Markt durchsetzt, wird stark davon abhängen, ob und wie es in die Beschreibungswelt von AUTOSAR integriert wird (siehe Kap. 8).

6.6 ODX-Diagnosedatensätze nach ASAM AE MCD 2D

Open Diagnostic Data Exchange (ODX) spezifiziert ein Datenmodell für die Haltung bzw. den Austausch aller diagnoserelevanten Daten eines Steuergerätes oder eines Fahrzeugs [1]. Unter Diagnose werden Aufgaben verstanden, die bei der Wartung und Fehlersuche in Werkstätten aber auch in der Fertigung von Fahrzeugen anfallen:

- Auslesen und Löschen des Fehlerspeichers,
- Durchführung von Funktionstests, Abfrage von Mess- und Stellgrößen des Fahrzeugs,
- Auslesen von Identifikationsdaten,
- Variantendefinition, Variantenauswahl und Variantenkonfiguration,
- Programmierung neuer Steuergerätesoftware und Datensätze (*Flashen*).

Die Umsetzung erfolgt mit den in Kap. 5 beschriebenen Diagnoseprotokollen UDS oder KWP 2000. Diese Protokolle definieren jedoch nur das Botschaftsformat der Diagnosedienste. Die innerhalb der Botschaften übertragenen Parameter unterscheiden sich je nach Fahrzeughersteller (OEM), Steuergerätehersteller und Gerätetyp. Insbesondere in der Art und Weise, wie, wann und in welchem Umfang Fehler in den Fehlerspeicher eingetragen werden, gibt es sehr unterschiedliche Herstellerphilosophien, die durch gesetzliche Anforderungen wie OBD nur teilweise harmonisiert werden.

Ein nicht minder wichtiges Themengebiet stellt die sogenannte Variantenverwaltung und -identifikation dar. Dahinter verbirgt sich die Möglichkeit, dass Steuergeräte des gleichen Typs unterschiedliche Ausprägungen ihrer Funktionalität besitzen können. So wird

zum Beispiel in einer Variante eines Tachometer-Steuergerätes für Europa die Geschwindigkeit im km/h angezeigt, in der Variante für die USA hingegen in mph, d. h. die Rohdaten der Geschwindigkeitsmessung werden je nach Variante anders interpretiert. Ein zweiter Ansatz zur Verwendung von Varianten stellt die Dokumentation von Entwicklungs- und Serienständen eines Steuergerätes dar. Die jeweilige Variante des Steuergerätes lässt sich anhand der Identifikationsdaten erkennen. Zur Vermeidung von redundanter Datenhaltung (*Single-Source-Prinzip*) und zur Vereinfachung der Datenpflege besteht an das Austauschformat die Forderung, alle Varianten ausgehend von einer Basis-Variante innerhalb eines einzigen Datensatzes zu halten.

Der oben beschriebene Diagnoseumfang eines datengetriebenen Diagnosetesters muss durch ein Datenmodell vollständig abgebildet werden. Die Datensätze müssen innerhalb von Entwicklung, Produktion und Service zwischen OEM, Zulieferer, Tool-Hersteller und Testeinrichtung, z. B. in einer Werkstatt, ausgetauscht werden können, ohne dass weitere Informationen notwendig sind.

ASAM definierte zunächst ein SGML-basiertes Datenformat, das sich für die Praxis als zu komplex erwies. Infolgedessen wurde eine vereinfachte Version 1.1.4 unter dem Namen ASAM MCD 2D Basic herausgegeben, für die sich die Bezeichnung ODX (*Open Diagnostic Data Exchange*) einbürgerte. Das Datenformat basiert auf XML (siehe Kasten am Ende von Abschn. 6.2) und wurde zu dem nachfolgend beschriebenen ASAM-MCD 2D (ODX) weiterentwickelt und in der Version ODX V2.1 als ISO 22901-1 standardisiert. Es basiert auf einem objektorientierten Ansatz und verwendet zur visuellen Darstellung in der Dokumentation die in der Informatik mittlerweile üblichen UML-Diagramme (*Unified Modeling Language*).

6.6.1 Aufbau des ODX-Datenmodells

Das ODX-Datenmodell ist unterteilt in acht große Kategorien (ODX-CATEGORY) (Abb. 6.26) für unterschiedliche Aufgabenbereiche:

Beschreibung der Kommunikationsprotokolle und des Fahrzeugnetzes

- **VEHICLE-INFO-SPEC**

Definiert alle Informationen für die Fahrzeugidentifikation und den Fahrzeugzugang. Dabei wird auch die Netzwerk-Topologie beschrieben, welche für den Zugriff auf die Steuergeräte über die Bussysteme und Gateways relevant ist.

- **COMPARAM-SPEC**

Fasst die Kommunikationsparameter für einen Protokoll-Stapel (PROT-STACK) bestehend aus mehreren COMPARAM-SUBSETs zusammen.

- **COMPARAM-SUBSET** (ab ODX 2.1)

Beschreibt die Kommunikationsparameter für eine bestimmte Schicht im ISO/OSI-Referenzmodell, z. B. das Zeitverhalten für die physikalische Schicht oder das Transport-Protokoll.

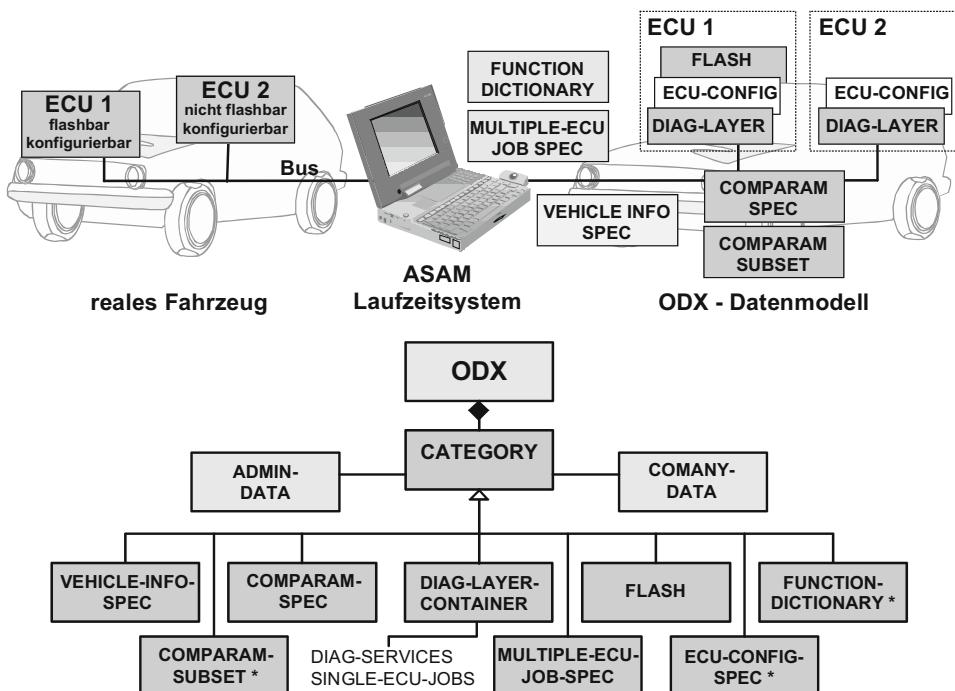


Abb. 6.26 Komponenten des ODX-Datenmodells (* ab ODX V2.1)

Hierarchische Beschreibung der Diagnosestruktur, Diagnosedienste und Abläufe

- **DIAG-LAYER-CONTAINER**

Beschreibt mit den Objekten BASE-VARIANT, ECU-VARIANT, PROTOCOL, FUNCTIONAL-GROUP und ECU-SHARED-DATA den hierarchisch aufgebauten Datensatz eines Steuergeräte-Typs und dessen Varianten aus Diagnosesicht. Die PROTOCOL-Schicht verweist auf Diagnosedienste (DIAG-SERVICES) und Diagnoseabläufe (SINGLE-ECU-JOB), zusammen als DIAG-COMM bezeichnet, mit den notwendigen Daten und Parametern, sogenannten Datenobjekten DOP.

- **MULTIPLE-ECU-JOB-SPEC**

Beschreibt sogenannte *Diagnostic Jobs* (Abläufe oder Makros), welche sich auf mehrere Steuergeräte beziehen und daher nicht innerhalb der eigentlichen Steuergeräte-Beschreibungen (DIAG-LAYER) dargestellt werden können.

Spezielle Aufgabenbereiche

- **FLASH**

Hier werden in ECU-MEM-Objekten alle Informationen abgelegt, die für das Programmieren von Code oder Daten in ein Steuergerät notwendig sind.

- **ECU-CONFIG-SPEC** (ab ODX 2.1)
Beinhaltet die Informationen zur Steuergerätekonfiguration (mitunter als *Variantencodierung* bezeichnet).
- **FUNCTION-DICTIONARY** (ab ODX 2.1)
Beinhaltet Informationen zur funktionsorientierten Diagnose.

Wird für eine dieser ODX-Kategorien ein Datensatz (Instanz) angelegt, so spricht man von einem *ODX-Dokument*. Ein *ODX-Dokument* beinhaltet immer nur eine der oben aufgeführten ODX-Kategorien. Die ODX-Kategorisierung erfolgte unter dem Aspekt, dass die Dateninhalte für die vollständige Beschreibung der Diagnoseumfänge eines Fahrzeuges aus unterschiedlichen Quellen stammen und unterschiedliche Lebenszeiten besitzen. So kann der Umfang der notwendigen Diagnosedienste (DIAG-LAYER-CONTAINER) für ein Steuergerät am besten von dessen Hersteller definiert werden. Dieser hat aber keine Informationen darüber, wie das Steuergerät in das Netzwerk des Fahrzeuges integriert wird und der externe Zugriff darauf erfolgt (VEHICLE-INFO-SPEC). Bestimmte Kommunikationsparameter einer entsprechenden Schicht im ISO/OSI-Referenzmodell (COMPARAM-SUBSET), z. B. die Baudrate der physikalischen Schicht „CAN“, sind für alle Steuergeräte an einem bestimmten Bus gleich. Dasselbe gilt für ganze Protokollstapel bestehend aus physikalischer, Transport- und Diagnoseschicht (COMPARAM-SPEC).

Um eine redundante Datenhaltung zu vermeiden, besteht innerhalb von ODX-Dokumenten die Möglichkeit, auf Datensätze anderer ODX-Dokumente zu referenzieren und so indirekten Zugriff auf diese Datensätze zu bekommen.

Viele ODX-Dateninhalte werden während der Projektlebenszeit durch unterschiedliche Personen und Firmen Änderungen unterzogen, die gegebenenfalls nachvollzogen werden müssen. Zu vielen ODX-Elementen lassen sich daher zusätzlich ADMIN-DATA (Versionierung, Datum und Änderungstext) und COMPANY-DATA Informationen ablegen, mit denen die Änderungshistorie beschrieben werden kann.

Im Folgenden wird zunächst die Hierarchiestruktur der DIAG-LAYER beschrieben. Danach werden die ODX-Beschreibungselemente für die Kommunikationsverbindung und die Diagnosedienste dargestellt. Abschließend folgen die Beschreibungsmechanismen für Sonderaufgaben wie die Flash-Programmierung.

6.6.2 DIAG-LAYER: Hierarchische Diagnosebeschreibung

Im einfachsten Fall besteht die ODX-Beschreibung aus drei Teilen (Abb. 6.27), der Beschreibung der Fahrzeugtopologie, dem Parametersatz des Bussystems und als wichtigster Teil für jedes im Fahrzeug eingebaute Steuergerät genau einem DIAG-LAYER-CONTAINER. Die Beschreibung der Fahrzeugtopologie (VEHICLE-INFO-SPEC) enthält Informationen über den Hersteller und Typ des Fahrzeugs und beschreibt, welche Steuergeräte im Fahrzeug verbaut sind, über welche Gateways und Bussysteme sie für den Diagnosetester erreichbar sind und verweist auf den Datensatz des jeweiligen Steuergeräts

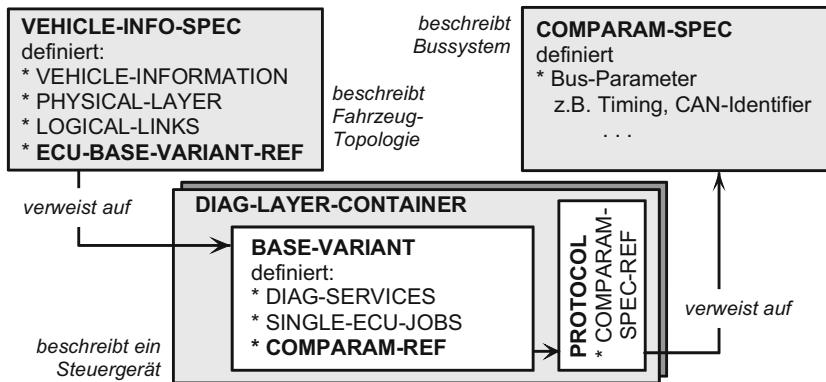


Abb. 6.27 Minimal-Struktur einer ODX-Beschreibung

im DIAG-LAYER-CONTAINER. Dieser enthält im einfachsten Fall im Unterabschnitt BASE-VARIANT sämtliche Diagnosedienste und Diagnoseabläufe eines Steuergerätes und verweist über das PROTOCOL auf die COMPARAM-SPEC, in der die Timing-Parameter, CAN-Identifier usw. des Bussystems definiert sind, über das kommuniziert wird.

In der Praxis existieren für jedes Steuergerät verschiedene Varianten. In der einfachen Struktur nach Abb. 6.27 müsste man jede Gerätevariante in einem eigenen BASE-VARIANT-Datensatz abbilden, der alle Diagnosedienste vollständig beschreibt, die das Gerät unterstützt. Um die Komplexität der Datensätze zu reduzieren und die Konsistenz der Datensätze leichter sicherzustellen, ist es besser, die gemeinsamen Eigenschaften aller Gerätevarianten nur einmal zu beschreiben, und für jede Gerätevariante nur die Unterschiede darzustellen. Dies wird durch das Konzept der Diagnoseschichten (*Diagnostic Layer*) in ODX ermöglicht (Abb. 6.28).

Die Steuergeräte-Beschreibung kann eine gemeinsame Bibliothek (ECU-SHARED-DATA) sowie bis zu vier Schichten, die DIAG-LAYER, enthalten, wobei lediglich die BASE-VARIANT-Schicht verbindlich vorgeschrieben ist. Die Verwendung des Schichtenmodells soll an einem Beispiel erläutert werden. Betrachtet werden die Steuergeräte für die Türen eines Fahrzeugs, die an das Karosserie-Bussystem angeschlossen sind. Das Türsteuergerät wird im Fahrzeug zweimal eingebaut, wobei die Variante für die Front-Türen zusätzlich die Steuerung der Außenspiegel übernimmt.

Diese Funktionen sind im Steuergerät für die Heck-Türen aus Kostengründen nicht enthalten. Die zugehörige ODX-Beschreibung könnte folgenden Aufbau haben (Abb. 6.28 und Tab. 6.10):

- In der obersten Schicht, dem PROTOCOL-LAYER (1) werden die für das verwendete Diagnoseprotokoll relevanten Daten und Dienste spezifiziert werden. Dort könnten z. B. die UDS Diagnosedienste wie *Read Data by Identifier* (vgl. Abschn. 5.2) definiert und auf die COMPARAM-SPEC des Bussystems verwiesen werden.

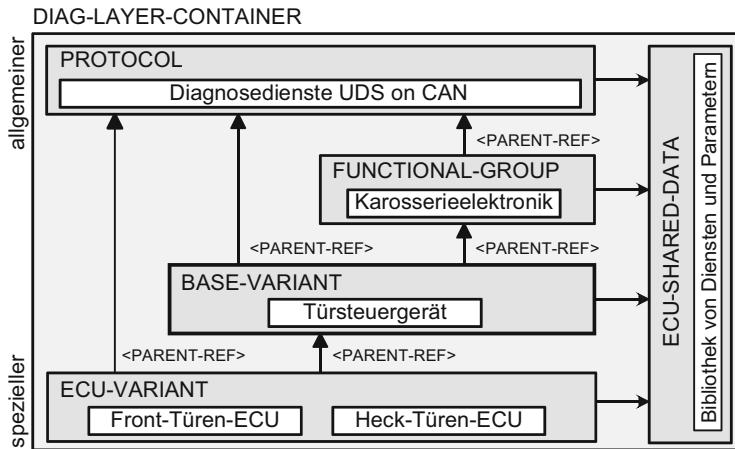


Abb. 6.28 Hierarchische Beschreibung (Diagnostic Layer) für Steuergeräte

- Die zugehörigen Parameter der Dienste, z. B. die *Identifier*, die bei jeder Steuergerätefamilie andere Werte haben, würde man dagegen sinnvollerweise in der ECU-BASE-VARIANT-Schicht definieren (2).
- Manche dieser *Identifier* allerdings werden nicht nur von den Türsteuergeräten verwendet, sondern sind vom Fahrzeughersteller vielleicht für alle Steuergeräte im Bereich der Karosserieelektronik einheitlich vorgegeben. ODX bietet die Möglichkeit, solche für alle Geräte einer bestimmten Anwendungsklasse gültigen Daten in einer Funktionsgruppe, der FUNCTIONAL-GROUP-Schicht, zusammenzufassen.
- Die Unterschiede zwischen der Front- und der Heck-Variante des Türsteuergerätes werden in der ECU-VARIANT-Schicht beschrieben (3). Dort wird dargestellt, welche zusätzlichen oder geänderten Diagnosedienste bzw. Daten das jeweilige Steuergerät im Vergleich zur Basisvariante besitzt und/oder welche Dienste das Steuergerät nicht unterstützt.

Die Beschreibungshierarchie erlaubt im Informatik-Jargon ein *Vererben* von Daten (*Value inheritance*) von oben nach unten, d. h. ein spezielles Steuergerät (untere Schicht oder *Kind-Schicht*) wird durch seine eigenen spezifischen Daten und durch alle in den höheren Schichten (*Eltern-Schicht*) definierten Daten beschrieben. Falls ein bestimmter Datenwert mehrfach definiert wird, so hat der auf der weiter unten liegende, d. h. speziellere Wert, Vorrang vor dem weiter oben stehenden, d. h. allgemeineren Wert. Auf diese Weise kann eine bestimmte Ausführung eines Steuergerätes einen Defaultwert des standardisierten Diagnoseprotokolls durch einen gerätespezifischen Wert überschreiben. Ausgedrückt wird die *Vererbung* in der zugehörigen XML-Beschreibungsdatei (Tab. 6.10) durch ein PARENT-REF-Element, das auf die zugehörige *Eltern-Schicht* verweist (4). Umgekehrt kann eine untere Schicht auch Daten einer höheren Schicht, die überhaupt nicht benötigt oder un-

Tab. 6.10 Hierarchische ODX-Beschreibung einer Steuergerätefamilie

```

<DIAG-LAYER-CONTAINER>
  <SHORT-NAME>ECU1</SHORT-NAME>
  <PROTOCOLS>                                ← Protokoll-Schicht          (1)
    <PROTOCOL ID="P.UDS.ID" TYPE="ISO_15765_3_on_ISO_15765_2">
      <DIAG-COMMS>
        <DIAG-SERVICE . . .>           ← Diagnosedienste des UDS-Protokolls
        . . .
        <COMPARAM-SPEC-REF ID-REF="COMPARAM.ISO15765.ID" . . . />      (6)
      </PROTOCOL>
    </PROTOCOLS>
    <BASE-VARIANTS>                            ← Grundvariante der Türsteuergeräte (2)
      <BASE-VARIANT ID="Door_ECU">
        <DIAG-COMMS>                      ← Spezielle Diagnosedienste und Daten
                                              der Türsteuergeräte
        . . .
        <PARENT-REFS>                      ← Erben der Dienste der Protokoll-Schicht
          <PARENT-REF xsi:type="PROTOCOL-REF" ID-REF="P.UDS.ID" />      (4)
        </PARENT-REFS>
      </BASE-VARIANT>
    </BASE-VARIANTS>
    <ECU-VARIANTS>                            ← Varianten der Türsteuergeräte
      <ECU-VARIANT ID="Rear_Door_ECU">  ← Variante Heck-Tür-Steuergerät (3)
        <DIAG-COMMS>                      ← Spezielle Diagnosedienste und Daten
                                              des Heck-Tür-Steuergerätes
        . . .
        <PARENT-REFS>                      ← Erben der Dienste der Grundvariante
          <PARENT-REF xsi:type="BASE-VARIANT-REF" ID-REF="Door_ECU">
            <NOT-INHERITED-DIAG-COMMS>  ← Vom Heck-Tür-Steuergerät nicht (5)
              <NOT-INHERITED-DIAG-COMM>  unterstützen Dienst der Grundvariante
              <DIAG-COMM-SNREF SHORT-NAME="SeedAndKey" />
            . . .
        </PARENT-REFS>
      </ECU-VARIANT>
    </ECU-VARIANTS>
  </DIAG-LAYER-CONTAINER>

```

terstützt werden, mit Hilfe von NOT-INHERITED-Elementen gezielt von der Vererbung ausblenden (5).

6.6.3 VEHICLE-INFO-SPEC: Fahrzeugzugang und Bustopologie

Das VEHICLE-INFO-SPEC-Dokument gibt an, für welches Fahrzeugmodell der ODX-Datensatz gültig ist, welche Steuergeräte verbaut sind, über welche Schnittstelle der Diagnosetester Zugang zum Fahrzeug erhält und wie die interne Topologie der Bussysteme im Fahrzeug beschaffen ist (Abb. 6.29).

Im Beispiel nach Abb. 6.29 bzw. Tab. 6.11, auf die sich die nachfolgend in () angegebenen Ziffern beziehen, erfolgt die *Fahrzeugidentifikation* (1,2) schrittweise ausgehend

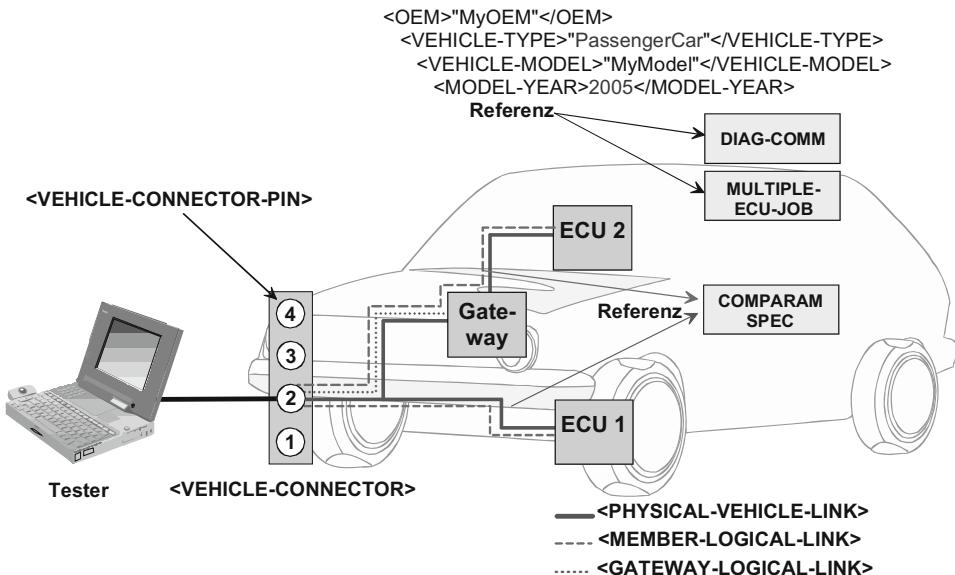


Abb. 6.29 Informationen aus dem ODX-Dokument VEHICLE-INFO-SPEC

vom Hersteller (OEM), über die Fahrzeugklasse (VEHICLE-TYPE), das Fahrzeugmodell (VEHICLE-MODEL) bis zum Modelljahr (MODEL-YEAR). Falls die Selektion im Laufzeitsystem nicht manuell durch den Bediener erfolgt, kann zusätzlich ein Verweis auf das Ergebnis eines Diagnosedienstes (DIAG-COMM oder MULTIPLE-ECU-JOB) angegeben werden, mit dem die Fahrzeugidentifikation gelesen wird.

Für den Fahrzeugzugang, z. B. über den OBD-Stecker, wird zunächst der Stecker selbst als VEHICLE-CONNECTOR (3) beschrieben. Ein VEHICLE-CONNECTOR-PIN definiert durch den PHYSICAL-VEHICLE-LINK (5) und dessen Typ, z. B. CAN, die physikalische Verbindung vom Diagnosetestern zum Fahrzeugstecker und von dort zum Steuergerät.

Ein LOGICAL-LINK (4) hingegen repräsentiert einen logischen Pfad vom Diagnosetestern zu einem Steuergerät. Falls ein Steuergerät mit mehreren Bussystemen verbunden ist, existieren möglicherweise über die verschiedenen physikalischen Busverbindungen mehrere logische Pfade zu einem Steuergerät. ODX unterscheidet bei einem LOGICAL-LINK zwischen einer Verbindung zu einem Gateway (GATEWAY-LOGICAL-LINK) oder zu einem regulären Steuergerät (MEMBER-LOGICAL-LINK). Jede logische Verbindung kann wiederum auf andere logische Verbindungen referenzieren. Um nun, wie im Abb. 6.29 dargestellt, eine reale Verbindung zum Steuergerät ECU 2 zu bekommen, muss der LOGICAL-LINK „rückwärts“ ausgewertet werden: Zunächst gibt es eine logische Verbindung von ECU 2 zum Gateway, von dort zum Pin 2 des Diagnosesteckers. Das ASAM-Laufzeitsystem kann nun eine Verbindung über die physikalische Schicht, z. B. CAN an Pin 2 des Diagnosesteckers, zum Gateway aufbauen. Das Gateway leitet die Information dann zum Steuergerät weiter.

Tab. 6.11 ODX-Beschreibung des Systems nach Abb. 6.29

```

<VEHICLE-INFO-SPEC ID="COMPACT-CLASS">
  <SHORT-NAME>CompactCar</SHORT-NAME>
  ...
  <INFO-COMPONENTS>
    <INFO-COMPONENT ID="ID_OEM">
      <SHORT-NAME>OEM</SHORT-NAME>
      <EXPECTED-VALUE>MyOEM</EXPECTED-VALUE>
    </INFO-COMPONENT> (1)
    <INFO-COMPONENT ID="ID_VEHICLE-TYPE">
      <SHORT-NAME>VEHICLE-TYPE</SHORT-NAME>
      <EXPECTED-VALUE>PassengerCar</EXPECTED-VALUE>
    </INFO-COMPONENT> (7)
    ...
  </INFO-COMPONENTS> (8)

  <VEHICLE-INFORMATION>
    <SHORT-NAME>VEHICLE-INFO</SHORT-NAME>
    ...
    <INFO-COMPONENT-REFS>
      <INFO-COMPONENT-REF ID-REF="ID_OEM"/> (2)
      <INFO-COMPONENT-REF ID-REF="ID_VEHICLE-TYPE"/> (9)
      ...
    </INFO-COMPONENT-REFS>
    ...
    <VEHICLE-CONNECTOR> (3)
      <SHORT-NAME>OBD-Connector</SHORT-NAME>
      <VEHICLE-CONNECTOR-PINS>
        <VEHICLE-CONNECTOR-PIN TYPE="HI" ID="ID_CAN-HI">
          <SHORT-NAME>CAN-HI</SHORT-NAME>
          <PIN-NUMBER>2</PIN-NUMBER>
        </VEHICLE-CONNECTOR-PIN>
        ...
      </VEHICLE-CONNECTOR-PINS>
    </VEHICLE-CONNECTOR>
    <LOGICAL-LINKS> (4)
      <LOGICAL-LINK ID="LL_ECU-1" xsi-type="MEMBER-LOGICAL-LINK">
        <SHORT-NAME>ECU-1</SHORT-NAME>
        <PHYSICAL-VEHICLE-LINK-REF ID-REF="PL-CAN-1"/>
        <BASE-VARIANT-REF ID-REF="ECU-1"/> (6)
        ...
      </LOGICAL-LINK>
      <LOGICAL-LINK ID="LL_GATEWAY" xsi-type="MEMBER-LOGICAL-LINK">
        <SHORT-NAME>GATEWAY</SHORT-NAME>
        <PHYSICAL-VEHICLE-LINK-REF ID-REF="PL-CAN-1"/>
        <BASE-VARIANT-REF ID-REF="GATEWAY-ECU"/>
        ...
      </LOGICAL-LINK>
      <LOGICAL-LINK ID="LL_ECU-2" xsi-type="GATEWAY-LOGICAL-LINK">
        <SHORT-NAME>ECU-2</SHORT-NAME>
        <GATEWAY-LOGICAL-LINK-REF ID-REF="GATEWAY"/>
        <PHYSICAL-VEHICLE-LINK-REF ID-REF="PL-CAN-2"/>
        <BASE-VARIANT-REF ID-REF="ECU-2"/>
        ...
      </LOGICAL-LINK>
    </LOGICAL-LINKS>
  
```

```

<PHYSICAL-VEHICLE-LINKS> (5)
  <PHYSICAL-VEHICLE-LINK ID="PL-CAN-1">
    <SHORT-NAME>PHYSICAL-LINK-CAN-1</SHORT-NAME>
    <VEHICLE-CONNECTOR-PIN-REFS>
      <VEHICLE-CONNECTOR-PIN-REF ID="CAN-HI">
      ...
    </VEHICLE-CONNECTOR-PIN-REFS>
  </PHYSICAL-VEHICLE-LINK>
  <PHYSICAL-VEHICLE-LINK ID="PL-CAN-2">
  ...
</PHYSICAL-VEHICLE-LINK>
</PHYSICAL-VEHICLE-LINKS>
</VEHICLE-INFO-SPEC>
</VEHICLE-INFO-SPEC>

```

Damit das Laufzeitsystem auch die korrekten Parameter für die Verbindungen, z. B. Baudrate und Zeitverhalten, einstellen kann, enthält der LOGICAL-LINK einen Verweis auf das DIAG-LAYER-Dokument (hier als BASE-VARIANT) für das jeweilige Steuerrgerät (6). Dort wird das für die Verbindung eingesetzte Busprotokoll definiert und auf die COMPARAM-SPEC verwiesen (vgl. (6) in Tab. 6.10).

An diesem Beispiel wird auch das Konzept von ODX deutlich, Daten möglichst nur an einer Stelle zu definieren (*Single Source* Prinzip) und von anderen Stellen aus durch Verweise zugänglich zu machen. Dazu erhalten ODX-Elemente als Attribut einen *Identifier ID*, eine Kennung, die für den jeweiligen Elementtyp im gesamten Datensatz eindeutig sein muss (7). Die ID wird von entsprechenden ODX-Autorenwerkzeugen häufig automatisch vergeben. Daher erhält das Element zusätzlich auch noch eine Klartext-Kurzbezeichnung SHORT-NAME (8). Auf ein derartig gekennzeichnetes Element kann von einer anderen Stelle aus nun entweder mit Hilfe des Identifiers über einen ODX-Link ID-REF (9) oder mit Hilfe von SN-REF über die Kurzbezeichnung verwiesen werden. Zusätzlich zum Identifier ID und zum Kurznamen SHORT-NAME, die bei praktisch allen ODX-Elementen vorgeschrieben sind, kann einem Element auf Wunsch auch ein längerer Name LONG-NAME und eine ausführliche Beschreibung DESCRIPTION zugeordnet werden, um dem Anwender eines ODX-Werkzeugs oder dem Benutzer des Diagnosetesters weitere Informationen anzuzeigen.

6.6.4 COMPARAM-SPEC und COMPARAM-SUBSET: Busprotokoll

Aus der COMPARAM-SPEC entnimmt das Laufzeitsystem Angaben über das Kommunikationsprotokoll. Die entsprechenden Parameter sind dabei nach ISO/OSI-Schichten getrennt in einzelne COMPARAM-SUBSETS gegliedert, welche wiederum zu einem Protokollstapel (PROT-STACK) zusammengefasst werden können (Abb. 6.30). Für jedes eingesetzte Protokoll existiert seit ODX V2.1 ein eigenes COMPARAM-SUBSET mit dessen spezifischen Kommunikationsparametern. So sind Bus- und Adressinfor-

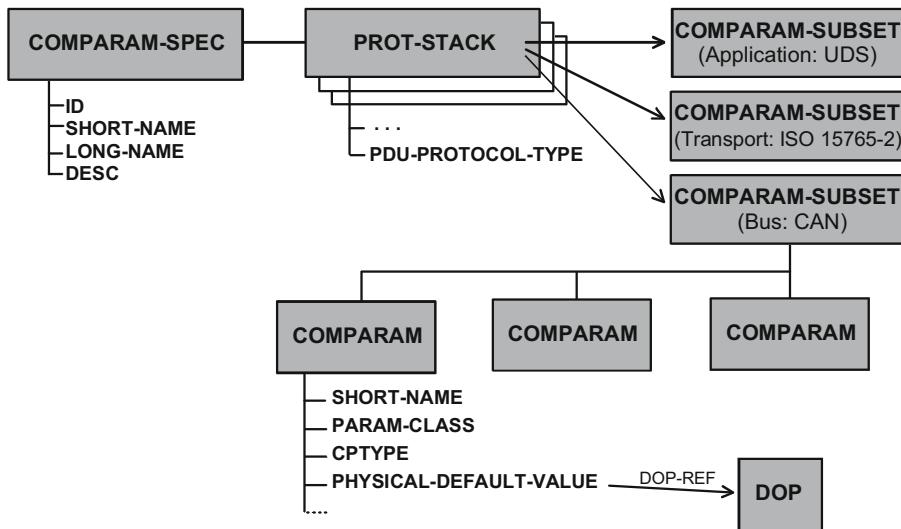


Abb. 6.30 Struktur der Parameterbeschreibung für das Busprotokoll

mationen des Steuergerätes (z. B. Baudrate oder CAN Identifier) und Angaben zum Aufbau einer Kommunikationsverbindung (z. B. Wakeup-Pattern) in einem Subset für die physikalische Schicht abgelegt. Transportprotokoll-Parameter (z. B. ST_{min} bei ISO-15765-2) oder Diagnoseprotokollparameter (z. B. P2 bei UDS) besitzen eigene Subsets. Auch Parameter wie etwa die Anzahl der Sendewiederholungen im Fehlerfall sind hier definiert. Ein Protokollstapel (PROT-STACK) fasst nun die einzelnen Protokollsichten zusammen, indem auf die jeweiligen COMPARAM-SUBSETS referenziert wird. Die COMPARAM-SPEC selbst beinhaltet einen oder mehrere solcher PROT-STACKs. ISO 22901-1 (entspricht ODX V2.1) in Verbindung mit ISO 22900-2 (D-PDU-API) definiert die ODX-Datensätze für die Diagnoseprotokolle KWP 2000 und UDS, das Transportprotokoll ISO 15765-2 und das Bussystem CAN (ISO 11898) sowie die entsprechenden Protokollstack-Kombinationen (PDU-PROTOCOL-TYPE), z. B. *UDS on CAN* als ISO_15765_3_on_ISO_15765_2. Welches Steuergerät welchen Protokollstapel verwendet, wird durch Verweis auf einen PROT-STACK im PROTOCOL-Feld der Diagnostic-Layer-Beschreibung des jeweiligen Steuergerätes festgelegt (siehe (6) in Tab. 6.10).

Jeder einzelne Kommunikationsparameter wird über einen eigenen COMPARAM-Abschnitt in der ODX-Datei beschrieben. Zur Identifikation des Parameters dient sein SHORT-NAME, der Parameter hat einen PHYSICAL-DEFAULT-Wert und verweist auf ein Datenobjekt Data-Object-Property DOP. Über die COMPU-METHOD des DOP kann der physikalische Wert in das interne Format des Bussystems umgerechnet werden. Der allgemeine Aufbau von DOP und COMPU-METHOD wird in Abschn. 6.6.6 beschrieben. Die Art der Kommunikationsparameter wird durch die PARAM-CLASS unterschieden (Tab. 6.12).

Tab. 6.12 Klassen von Kommunikationsparametern

PARAM-CLASS	Beschreibung
TIMING	Zeiten für den Botschaftsverkehr auf dem Bussystem
INIT	Initialisierungsparameter
COM	Allgemeine Kommunikationsparameter (z. B. CAN-Identifier)
ERRHDL	Parameter für die Fehlerbehandlung (z. B. Sende-Wiederholung)
BUSTYPE	Bussystem-spezifische Parameter (z. B. Baudrate)
UNIQUE_ID	Eindeutige Identifier

Tab. 6.13 Weitere Unterteilung von Kommunikationsparametern

CPTYPE	Beschreibung
STANDARD	Der Kommunikationsparameter ist fester Bestandteil eines standardisierten Protokolls und muss von jedem Laufzeitsystem unterstützt werden, welches dieses Protokoll implementiert.
OEM-SPECIFIC	Der Kommunikationsparameter ist fester Bestandteil eines nicht standardisierten, OEM-spezifischen Protokolls. Der Parameter muss vom Laufzeitsystem unterstützt werden, wenn das OEM-spezifische Protokoll implementiert wird.
OPTIONAL	Diese Parameter müssen vom Laufzeitsystem nicht unterstützt werden. Wird ein solcher Parameter trotzdem verwendet, kann das Laufzeitsystem diesen ignorieren und die Kommunikation ohne diesen Parameter weiterführen.

Für das Laufzeitsystem werden alle Kommunikationsparameter nochmals in drei Kategorien unterteilt (Tab. 6.13), mit denen zwischen Standardwerten, herstellerspezifischen und optionalen Werten unterschieden wird.

Im Beispiel nach Tab. 6.14 wird der Parameter P2 min nach ISO 15765 definiert, der dort den Mindestbotschaftsabstand zwischen Testeranfrage und Steuergeräteantwort angibt (vgl. Abschn. 2.2.3). Dieser Parameter (1) besitzt den Initialwert 25 und referenziert ein Datenobjekt DOP namens DOP_P2MIN (2). Durch die Umrechnungsfunktion IDENTICAL des DOPs und die Einheit (UNIT) Millisekunden kann das Laufzeitsystem den Wert für den Kommunikationsparameter P2 min auf 25 ms einstellen.

6.6.5 DIAG-COMM und DIAG-SERVICE: Diagnosedienste

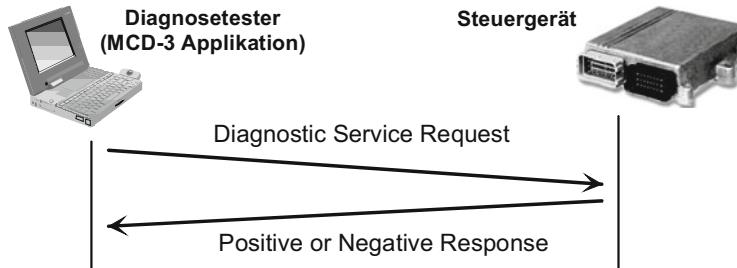
Ein KWP 2000- oder UDS-Diagnosedienst setzt sich aus einer Diagnose-Anforderung vom Tester an das Steuergerät (Request) sowie ein oder mehreren positiven oder negativen Antwortbotschaften vom Steuergerät zum Tester (Response) zusammen (Abb. 6.31, vgl. Kap. 5). Um beispielsweise einen Temperaturwert aus dem Steuergerät auszulesen, könnte der UDS-Diagnosedienst *Read (Data) Memory By Address* SID = 23 h verwendet werden:

Tab. 6.14 Ausschnitt aus einem COMPARAM-SUBSET

```

<COMPARAM-SUBSET ID="COMPARAM-SUBSET.ISO15765" >
  <SHORT-NAME>ISO_15765_COMPARAM</SHORT-NAME>
  <COMPARAMS>
    <COMPARAM CPTYPE="STANDARD" (1)
      ID="COMPARAM-SUBSET.ISO15765.p2min"
      PARAM-CLASS="TIMING"
      <SHORT-NAME>P2min</SHORT-NAME>
      <PHYSICAL-DEFAULT-VALUE>25</PHYSICAL-DEFAULT-VALUE>
      <DATA-OBJECT-PROP-REF>
        ID-REF="COMPARAM-SUBSET.ISO15765.P2MIN"
      </DATA-OBJECT-PROP-REF>
    </COMPARAM>
    ...
  </COMPARAMS>
  <DATA-OBJECT-PROP ID="P2MIN" (2)
    <SHORT-NAME>DOP_P2min</SHORT-NAME>
    ...
    <COMPU-METHOD>
      <CATEGORY>IDENTICAL</CATEGORY>
    </COMPU-METHOD>
    <UNIT ID="Unit_MILLISECOND" >
      <SHORT-NAME>Unit_MilliSecond</SHORT-NAME>
      <DISPLAY-NAME>ms</DISPLAY-NAME>
    </UNIT>
    ...
  </DATA-OBJECT-PROP>
</COMPARAM-SUBSET>

```

**Abb. 6.31** Diagnosedienst

- Falls der Temperaturwert im Steuergerätespeicher ab der Adresse 0524 h steht und 2 Byte lang ist, würde die Anforderungsbotschaft folgendermaßen aussehen:

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4
SID = 23 h	12 h	05 h	24 h	02 h

- Im Erfolgsfall (Positive Response) erhält der Tester die Antwort

Byte 0	Byte 1	Byte 2
63 h	04 h	E5 h

dabei ist 63 h die vom Steuergerät wiederholte SID, die in der Antwort protokollgemäß mit gesetztem Bit 6 zurückgesendet wird. Der aktuelle Temperaturwert sei 04E5 h, der wie die Speicheradresse in High-Low-Byte-Reihenfolge (*Big Endian*) übertragen werden soll.

- Im Fehlerfall (Negative Response) könnte die Antwort beispielsweise lauten

Byte 0	Byte 1	Byte 2
7Fh	23 h	31 h

wobei 7Fh anzeigt, dass es sich um eine Fehlermeldung handelt, 23 h ist die fehlerhafte SID und 31 h steht für den UDS-Fehlercode *Request out of Range*.

Die Struktur der ODX-Beschreibung für eine derartige Botschaft ist in Abb. 6.32 dargestellt. Hauptelement der Diagnosebeschreibung sind DIAG-COMM Objekte, mit denen Diagnosedienste DIAG-SERVICE und Diagnoseabläufe SINGLE-ECU-JOB beschrieben werden. Über die Attribute AUDIENCE und SECURITY-ACCESS-LEVEL kann die Verwendung des Diagnosedienstes für bestimmte Anwendergruppen (z. B. Hersteller, Entwicklung, Fertigung, Werkstatt) zugelassen bzw. verboten werden und von einer bestimmten Sicherheitsstufe abhängig gemacht werden. Wie das Laufzeitsystem die Berechtigung des Anwenders überprüft, ist allerdings im MCD 2/3-Standard nicht definiert. Außerdem lassen sich die Diagnosedienste über DIAGNOSTIC-CLASS so klassifizieren, dass dem Anwender in einer bestimmten Situation nur bestimmte in diesem Kontext sinnvolle Dienste angeboten werden.

Im DIAG-SERVICE-Abschnitt des zugehörigen ODX-Dokumentes wird auf die Anforderungs- und die Antwortbotschaften für den Diagnosedienst verwiesen (Tab. 6.15). Die Botschaften selbst werden in den entsprechenden REQUEST, POS-RESPONSE und NEG-RESPONSE-Abschnitten definiert (1 bis 3). Die in einer Botschaft vorhandenen Daten werden durch PARAM-(Parameter)-Abschnitte beschrieben (Tab. 6.16), wobei ein Daten-element selbst durch seine Byte- und Bit-Position innerhalb der Botschaft sowie einen Querverweis auf ein entsprechendes Datenobjekt DOP (siehe unten) definiert wird, das die zugehörigen Einzelheiten enthält.

Bei der Definition der positiven Antwort (Tab. 6.16) auf die oben beschriebene Temperaturabfrage wird beispielsweise angegeben, dass das Byte 0 der Antwort den Wert $99 = 63 \text{ h} + 23 \text{ h} + 40 \text{ h}$ haben muss. Da dieser Wert nicht weiter von Interesse ist, wird er vollständig im PARAM-Feld (4) beschrieben. Für den eigentlichen Temperaturwert (5) dagegen wird auf ein separates Datenobjekt DOP verwiesen (6). Im PARAM-Feld selbst

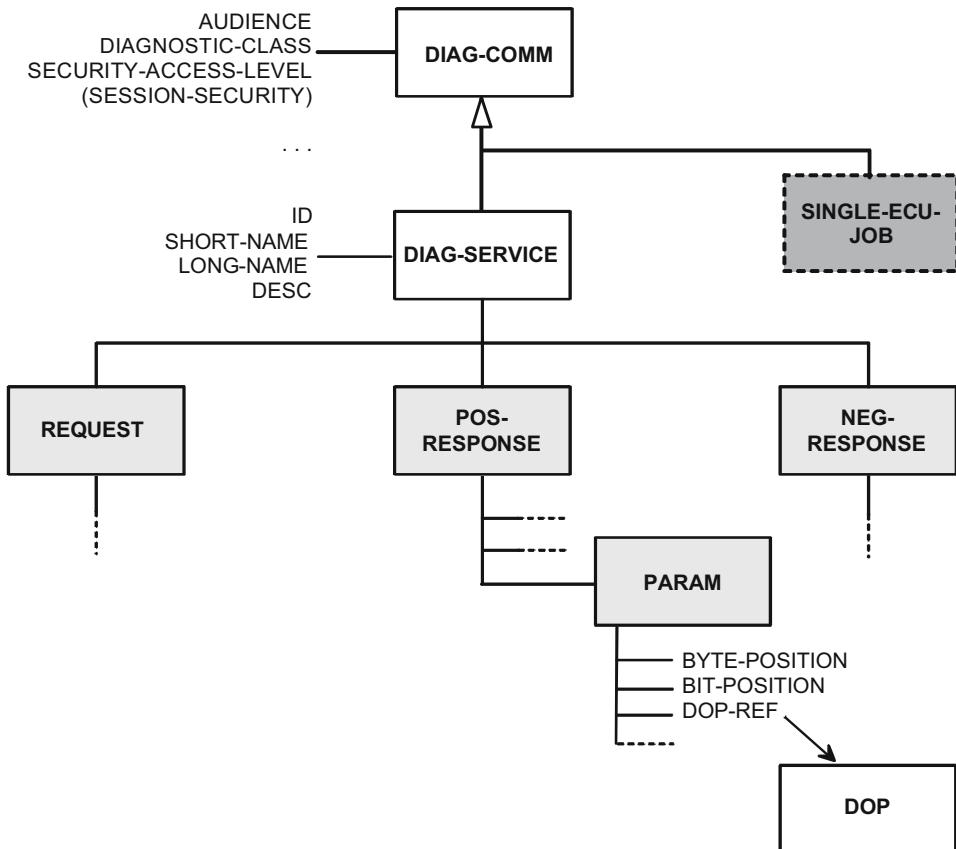


Abb. 6.32 Struktur der ODX-Beschreibung für Diagnosedienste

wird lediglich angegeben, dass der Temperaturwert ab Bit 0 (BIT_POSITION) des Bytes 1 (BYTE_POSITION) beginnt.

Mit Hilfe von Parametern kann das ASAM-Laufzeitsystem somit die Anforderungsbotschaft eines Diagnosedienstes aus DOP-Elementen zusammensetzen bzw. die Antwortbotschaft wieder auflösen. In Tab. 6.17 sind die vordefinierten PARAM-Typen aufgeführt.

DIAG-VARIABLE: Diagnosevariable Beim Messen und Kalibrieren wird direkt auf Größen im Speicher des Steuergerätes zugegriffen. Für den Zugriff wird dabei der steuergeräteinterne Name der Größe verwendet, die mit Hilfe einer vom Compiler/Linker der Steuergerätesoftware erstellten Mapping-Datei einer Speicheradresse im Steuergerät zugeordnet wird. ODX bietet mit den Diagnosevariablen (DIAG-VARIABLE) einen ähnlichen Mechanismus (Tab. 6.18). Im Beispiel wird eine Diagnosevariable DV_PHY_TMOT definiert (1), die der Variablen Tmot in der Steuergerätesoftware entspricht. Das eigentliche Lesen und Schreiben der Variable erfolgt durch die Diagnosedienste DS_ReadMotorTemperature

Tab. 6.15 ODX-Beschreibung eines Diagnosedienstes

```

<DIAG-SERVICE ID="DS_ReadMemoryByAddress">
  <SHORT-NAME>DS_ReadMemoryByAddress</SHORT-NAME>
  .
  .
  <REQUEST-REF ID-REF="REQUEST_ReadMemoryByAddress"/> (1)
  <POS-RESPONSE-REFS>
    <POS-RESPONSE-REF ID-REF="RESP_ReadMemoryByAddress"/> (2)
  </POS-RESPONSE-REFS>
  <NEG-RESPONSE-REFS>
    <NEG-RESPONSE-REF ID-REF="NEGRESP_IllegalFormat"/> (3)
  </NEG-RESPONSE-REFS>
  .
  .
</DIAG-SERVICE>

```

Tab. 6.16 Beschreibung einer Antwortbotschaft in ODX

```

<POS-RESPONSE ID="RESP_ReadMemoryByAddress">
  <SHORT-NAME>RESP_ReadMemoryByAddress</SHORT-NAME>
  .
  .
  <PARAMS>
    <PARAM SEMANTIC="SERVICE-ID" xsi:type="CODED-CONST"/> (4)
      <SHORT-NAME>POS-RESP-ReadMemoryByAddress</SHORT-NAME>
      <BYTE-POSITION>0</BYTE-POSITION>
      <CODED-VALUE>99</CODED-VALUE>
      <DIAG-CODED-TYPE BASE-DATA-TYPE="A_UINT32"/>
      <BIT-LENGTH>8</BIT-LENGTH>
    </PARAM>
    <PARAM xsi:type="VALUE"/> (5)
      <SHORT-NAME>Temperature</SHORT-NAME>
      <BYTE-POSITION>1</BYTE-POSITION>
      <BIT-POSITION>0</BIT-POSITION>
      <DOP-REF ID-REF="DOP_Temperature"/> (6)
    </PARAM>
  </PARAMS>
  .
  .
</POS-RESPONSE>

```

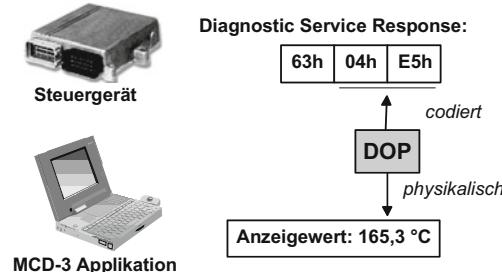
bzw. DS_WriteMotorTemperature, auf die in der Beschreibung verwiesen wird (2, 3). Darauf verbergen sich z. B. die UDS-Diagnosedienste *Read Memory By Address* oder *Write Memory By Address*. Eine MCD-Anwendung kann transparent auf die Diagnosevariable wie auf eine lokale Variable zugreifen, während das MCD-Laufzeitsystem im Hintergrund die Datenübertragung von und zum Steuergerät abwickelt.

6.6.6 Einfache und komplexe Datenobjekte

Simple DOP: Einfache Datenobjekte Kernelement der ODX-Datensätze sind die *einfachen Datenobjekte* (Data Object Property DOP). Sie dienen dazu, die in den Parametern eines Diagnosedienstes verpackten und codierten Informationen eines Steuergerätes korrekt zu interpretieren und die Inhalte für den Menschen verständlich zu machen (Abb. 6.33).

Tab. 6.17 Parametertypen von Diagnosebotschaften

Typ	Bemerkung
VALUE	Normaler Datenwert, verweist auf ein DOP.
RESERVED	Wird verwendet, wenn das Laufzeitsystem den Parameter ignorieren soll.
CODED-CONST	Konstante, z. B. für Service Identifier (SID) verwendet.
PHYS-CONST	Wie CODED-CONST, enthält aber zusätzlich eine Umrechnung in einen physikalischen Wert.
LENGTH-KEY	Wird verwendet, wenn die Parameter-Länge von einem anderen Parameter abhängt.
MATCHING-REQUEST-PARAM	Wird verwendet, wenn ein Parameter der Antwort mit einem Wert der Anforderung verglichen werden muss (z. B. die Local-ID bei einem Diagnose-Dienst).
TABLE-KEY	Wird bei komplexen Datenobjekten (siehe Abschn. 6.6.6) verwendet, wenn Datenstrukturen über Kennwerte (Data Identifier) identifiziert werden wie bei den KWP 2000/UDS-Diagnosediensten <i>Read/Write Data By Identifier</i> .
TABLE-STRUCT	
TABLE-ENTRY	
DYNAMIC	Dieser Typ zeigt dem Laufzeitsystem an, dass der Parameter dynamisch definiert wird.
SYSTEM	Dieser Parameter wird verwendet, wenn ein Wert abhängig von System-Informationen berechnet werden soll (z. B. aus der aktuellen Systemzeit)
NRC-CONST	Wie CODED-CONST...

Abb. 6.33 Verwendung eines DOP

Bei den einfachen DOPs unterscheidet ODX zwei Typen:

- Der gewöhnliche DOP bildet die Basis aller anderen Datenobjekte.
- Der DTC-DOP (Diagnostic Trouble Code) beschreibt Fehlerspeichereinträge.

Jedem DOP wird zur Unterscheidung ein eindeutiger Identifier (ID) und eine Kurzbezeichnung (SHORT-NAME) zugeordnet. Mit einem optionalen LONG-NAME und einer DESCRIPTION wird ein DOP im Klartext beschrieben. Ein DOP besteht aus den in Tab. 6.19 aufgeführten Elementen.

Abbildung 6.34 zeigt als Beispiel ein DOP für einen Temperaturwert. Für das codierte, d. h. Steuergeräte-interne Format (DIAG-CODED-TYPE) wurde im Beispiel eine vor-

Tab. 6.18 Beschreibung einer Diagnosevariablen

```

<DIAG-VARIABLES>
  <DIAG-VARIABLE ID="DV_PHY_TMOT" IS-READ-BEFORE-WRITE="false"> (1)
    <SHORT-NAME>DV_PHY_TMOT</SHORT-NAME>
    <SW-VARIABLES>
      <SW-VARIABLE>
        <SHORT-NAME>Tmot</SHORT-NAME>
        <ORIGIN>A</ORIGIN>
      </SW-VARIABLE>
    </SW-VARIABLES>
    <COMM-RELATIONS>
      <COMM-RELATION VALUE-TYPE="CURRENT">
        <RELATION-TYPE>READ</RELATION-TYPE>
        <DIAG-COMM-SNREF SHORT-NAME="DS_ReadMotorTemperature"/> (2)
          <OUT-PARAM-IF-SNREF SHORT-NAME="STAT_PHY_TMOT"/>
        </COMM-RELATION>
      <COMM-RELATION VALUE-TYPE="CURRENT">
        <RELATION-TYPE>WRITE</RELATION-TYPE>
        <DIAG-COMM-SNREF SHORT-NAME="DS_WriteMotorTemperature"/> (3)
          <IN-PARAM-IF-SNREF SHORT-NAME="ARG_PHY_TMOT"/>
        </COMM-RELATION>
      </COMM-RELATIONS>
    </DIAG-VARIABLE>
  </DIAG-VARIABLES>

```

Tab. 6.19 Elemente eines einfachen ODX-Datenobjekts DOP

DIAG-CODED-TYPE	Datentyp des codierten Wertes eines Datums.
INTERNAL-CONSTR	Grenzwerte für das Datum im codierten (internen) Format, z. B. Monatsangabe in hexadezimaler Schreibweise. Bsp.: 01 h ≤ Monat ≤ 0Ch, alle anderen Werte sind ungültig.
COMPU-METHOD	Umrechnungsmethode zwischen dem codierten Wert und der physikalischen Größe.
PHYSICAL-TYPE	Datentyp für den physikalischen Wert eines Datums.
UNIT	Einheit zum physikalischen Wert (°C, m/s, km/h, ...).

zeichenlose 32 bit Ganzzahl des vordefinierten Datentyps A_UINT32 (unsigned integer 32 bit) gewählt. Mittels der Umrechnungsformel (COMPU-METHOD) kann das codierte (interne) Format in das physikalische Format (PHYSICAL-TYPE) umgerechnet werden, für das eine 32 bit Gleitkommazahl des Typs A_FLOAT32 verwendet wird. Die Einheit (UNIT) des physikalischen Wertes ist °C. Bei der Wandlung zwischen codiertem Format und physikalischem Wert können zusätzlich noch Grenzen (INTERNAL-CONSTR) für den Wertebereich der codierten Größe festgelegt werden. In Abb. 6.34 beträgt der steuergeräteinterne Wert 4E5 h. Dieser Wert liegt innerhalb des zulässigen Wertebereichs von 0 bis 2400 = 960 h. Er kann über die Umrechnungsformel $y = 40 + 0.1 \cdot x = 40 + 0.1 \cdot 4E5$ h korrekt in den physikalischen Wert 165,3 umgerechnet und in der Einheit °C angezeigt werden.

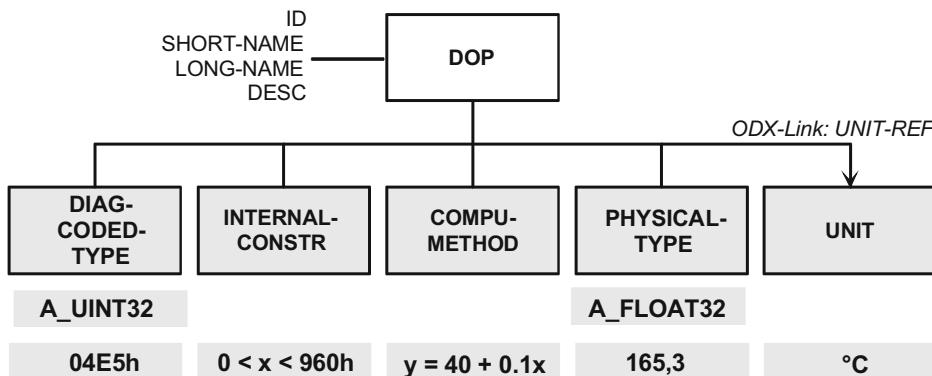


Abb. 6.34 Beispiel eines einfachen Datenobjekts DOP

Tabelle 6.20 zeigt den zugehörigen ODX-Datensatz, dessen Details im Folgenden erläutert werden sollen. Die in () stehenden Ziffern beziehen sich auf die entsprechenden Nummern im Datensatz.

- (1.) Aus Sicht des ASAM-Laufzeitsystems bilden alle zwischen Steuergerät und Tester in einer Diagnosebotschaft enthaltenen Daten zunächst einen Datenstrom, der zur Interpretation in seine Bestandteile zerlegt werden muss. Dazu benötigt das Laufzeitsystem die Information, wo ein Datum innerhalb des Datenstroms beginnt und wie lang es ist. Die Information über die Position im Datenstrom wurde bereits bei Tab. 6.16 erläutert. Die Länge bzw. die Art der Längenberechnung des Datums selbst wird über entsprechende LENGTH-INFO-TYPES festgelegt (Abb. 6.35). Durch diese Informationen werden die Daten aus dem Datenstrom separiert und im entsprechenden internen Datenformat (DIAG-CODED-TYPE) zwischengespeichert. Folgende Datentypen stehen dabei zur Verfügung:

Tab. 6.20 ODX-Datensatz für das Beispiel nach Abb. 6.34

```

<DATA-OBJECT-PROP ID="DOP_Temperature">
  <SHORT-NAME>MotTemp</SHORT-NAME>
  <LONG-NAME>Motortemperatur</LONG-NAME>
  <DESC>Kühlwassertemperatur in Grad Celsius</DESC>
  <DIAG-CODED-TYPE> (1)
    BASE-DATA-TYPE="A_INT32"
    xsi:type="STANDARD-LENGTH-TYPE"
    IS-HIGH-LOW-BYTE-ORDER="true"
    <BIT-LENGTH>16</BIT-LENGTH>
  </DIAG-CODED-TYPE>

  <INTERNAL-CONSTR> (2)
    <LOWER-LIMIT>0</LOWER-LIMIT>
    <UPPER-LIMIT>2400</UPPER-LIMIT>
  </INTERNAL-CONSTR>

  <PHYSICAL-TYPE> (3)
    BASE-DATA-TYPE="A_FLOAT32" DISPLAY-RADIX="DEC"
    <PRECISION>1</PRECISION>
  </PHYSICAL-TYPE>

  <COMPU-METHOD> (4)
    <CATEGORY>LINEAR</CATEGORY>
    <COMPU-INTERNAL-TO-PHYS>
      <COMPU-SCALES>
        <COMPU-SCALE>
          <COMPU-RATIONAL-COEFFS>
            <COMPU-NUMERATOR>
              <V>400</V>
              <V>1</V>
            </COMPU-NUMERATOR>
            <COMPU-DENOMINATOR>
              <V>10</V>
            </COMPU-DENOMINATOR>
            <COMPU-RATIONAL-COEFFS>
          </COMPU-SCALE>
        </COMPU-SCALES>
      </COMPU-INTERNAL-TO-PHYS>
    </COMPU-METHOD>

    <UNIT-SPEC> (5)
      <UNITS>
        <UNIT ID="Unit_Celsius">
          <SHORT-NAME>Unit_Celsius</SHORT-NAME>
          <DISPLAY-NAME>°C</DISPLAY-NAME>
        </UNIT>
        ...
      </UNITS>
    <UNIT-SPEC>
  </DATA-OBJECT-PROP>

```

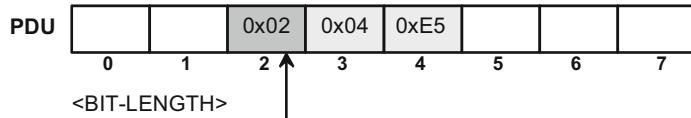
LENGTH-INFO-TYPE	Beschreibung
STANDARD-LENGTH-TYPE	Die Länge des Datenwerts wird explizit durch BIT-LENGTH spezifiziert.
LEADING-LENGTH-INFO-TYPE	Die Längeninformation steht im Datenbyte vor dem eigentlichen Datenwert innerhalb der PDU.
PARAM-LENGTH-INFO-TYPE	Die Längeninformation steht in einem anderen Parameter. Auf diesen zweiten Parameter wird hier nur referenziert.
MIN-MAX-LENGTH-TYPE	Für Datenwerte variabler Länge, wobei ein als TERMINATION definiertes Byte den Datenwert abschließt. Außerdem wird die minimale (MIN-LENGTH) und maximale (MAX-LENGTH) Datenlänge angeben. Mögliche Werte für TERMINATION sind: ZERO: Der zu extrahierende Bytestrom endet mit 00 h HEX-FF: Der zu extrahierende Bytestrom endet mit FFh END-OF-PDU: Es gibt keinen festgelegten Wert für das Ende des Bytestromes. Das Ende ist durch das Ende der Botschaft gegeben.

Wenn der Datenwert mehrere Bytes umfasst, muss die Reihenfolge der Datenbytes angegeben werden. Die beiden möglichen Formate High-Byte-First (Motorola Big-Endian-Format) und Low-Byte-First (Intel Little-Endian-Format) werden mit dem Parameter IS-HIGH-LOW-BYTE-ORDER (TRUE/FALSE) unterschieden.

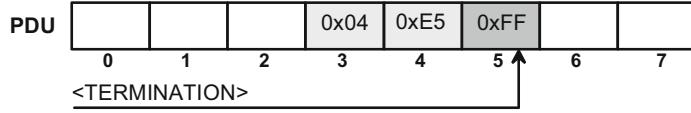
- (2.) Der Gültigkeitsbereich von internen Werten (DIAG-CODED-TYPE) kann durch Grenzwerte (INTERNAL-CONST) eingeschränkt werden. Diese Einschränkung umfasst sowohl einen unteren Grenzwert (LOWER-LIMIT) als auch einen oberen Grenzwert (UPPER-LIMIT). Darüber hinaus können Sub-Intervalle (SCALE-CONST) definiert werden, mit deren Hilfe gültige, ungültige, nicht definierte oder nicht verfügbare Bereiche festgelegt werden. Jeder interne Wert wird zunächst gegen die Grenzwerte getestet. Die Umrechnung in den physikalischen Wert erfolgt nur, wenn der Wert als gültig erkannt wird. Im Beispiel ist der Wertebereich der gültigen Temperaturwerte 0 ... 2400 = 960 h und somit ein geschlossenes Intervall (INTERVAL-TYPE = "CLOSED").
- (3.) PHYSICAL-TYPE beschreibt den Datentyp für die physikalische Darstellung des internen Wertes. Die optionale Information DISPLAY-RADIX gibt an, in welchem Zahlenformat der Wert angezeigt werden soll. Möglich ist dabei eine hexadezimale (HEX), eine dezimale (DEC), eine oktale (OCT) oder binäre (BIN) Darstellung. PRECISION definiert die Anzahl der Nachkommastellen.
- (4.) Die Umrechnung des internen Wertes DIAG-CODED-TYPE in den physikalischen Wert PHYSICAL-TYPE erfolgt über die im Feld COMPU-METHOD angegebene Umrechnungsmethode. Insgesamt gibt es acht Methoden CATEGORY:

CATEGORY	Beschreibung
IDENTICAL	Interner und physikalischer Wert sind identisch, keine Umrechnung erforderlich.
LINEAR	Linearer Zusammenhang nach der Formel: $y = \frac{v_{N0} + v_{N1} \cdot x}{v_{D0}}$, Einzelheiten siehe RAT-FUNC
SCALE-LINEAR	Abschnittweise lineare Funktionen. Jeder Abschnitt wird als COMPU-SCALE mit Unter- und Obergrenze spezifiziert.
TAB-INTP	Wertetabelle mit linearer Interpolation
TEXTTABLE	Umsetzung eines Zahlenwertes in einen Textwert.
COMPUCODE	Wird verwendet, wenn eine in der Programmiersprache JAVA geschriebene Funktion die Umrechnung durchführen soll.
RAT-FUNC	Allgemeine rationale Funktion nach der Formel $y = \frac{v_{N0} + v_{N1} \cdot x + v_{N2} \cdot x^2 + \dots + v_{Nm} \cdot x^m}{v_{D0} + v_{D1} \cdot x + v_{D2} \cdot x^2 + \dots + v_{Dn} \cdot x^n}$ dabei ist x der interne und y der physikalische Wert. Die Parameter $v_{N0}, v_{N1}, \dots, v_{Nm}$ des Zählerpolynoms (COMPU-NUMERATOR) und $v_{D0}, v_{D1}, \dots, v_{Dn}$ des Nennerpolynoms (COMPU-DENOMINATOR) sind Konstanten.
SCALE-RAT-FUNC	Abschnittweise rationale Funktionen.

LEADING-LENGTH-INFO-TYPE:



MIN-MAX-LENGTH-TYPE:



STANDARD-LENGTH-TYPE:

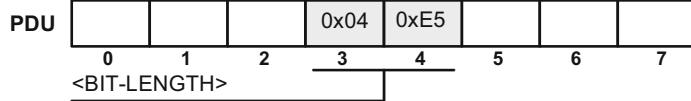


Abb. 6.35 Festlegung der Länge eines Datenfeldes in einer Diagnosebotschaft

Im obigen Beispiel für den Temperaturwert wird eine lineare Umrechnung mit

$$y = 40 + 0.1x = \frac{400 + 1x}{10}$$

verwendet. Die Zählerkoeffizienten sind 400 und 1, der Nennerkoeffizient ist 10.

- (5.) Im Abschnitt UNIT-SPEC wird dem physikalischen Wert eine physikalische Einheit zugeordnet. ODX bietet die Möglichkeit, denselben physikalischen Wert in unterschiedlichen Einheiten darzustellen, z. B. km/h und mph. Hierfür wird eine Größe zunächst in den SI-Basis единицах, z. B. m/s, dargestellt. Andere Einheiten werden daraus über einen Faktor und einen Offsetwert abgeleitet:

$$\text{Einheit} = \text{SI - Basis единица} \times \text{FACTOR-SI-TO-UNIT} + \text{OFFSET-SI-TO-UNIT}$$

Beispiel: Umrechnung von km/h in mph über die SI-Einheit m/s

$$\begin{aligned} [\text{m/s}] &= [\text{km/h}] \times 1/3.6 \\ \text{mph} &= [\text{m/s}] \times 2,23741 + 0 \end{aligned}$$

DISPLAY-NAME beschreibt, wie die Einheit in der Diagnoseanwendung über die MCD 3-Schnittstelle visuell dargestellt werden soll. Physikalische Werte, die nicht als SI-Einheit sondern direkt gespeichert sind, wie im Beispiel die Temperatur in °C, verwenden nur den DISPLAY-NAME.

DTC-DOP: Datenobjekte für Fehlerspeichereinträge In der Regel legen Steuergeräte beim Auftreten eines internen Fehlers neben dem Fehlercode (*Diagnostic Trouble Code DTC*) zusätzliche Informationen über die Umgebungsbedingungen wie Motordrehzahl oder Motortemperatur ab. Damit soll der aufgetretene Fehler möglichst exakt dokumentiert werden. In ODX werden die Fehlercodes in *Diagnostic Trouble Code Datenobjekten* (DTC-DOP) beschrieben (Tab. 6.21), die neben den Informationen eines gewöhnlichen DOP noch zusätzliche DTC-spezifische Informationen wie den internen Fehlercode (TROUBLE-CODE), den für die Anzeige zu verwendenden Fehlercode (DISPLAY-TROUBLE-CODE), eine Fehlermeldung im Klartext (TEXT) und einen Wert (LEVEL) für die Klassifizierung des Fehlers nach Schwere oder Art enthalten. Die zugehörigen Umgebungsbedingungen werden als komplexe Datenobjekte (siehe unten) gespeichert.

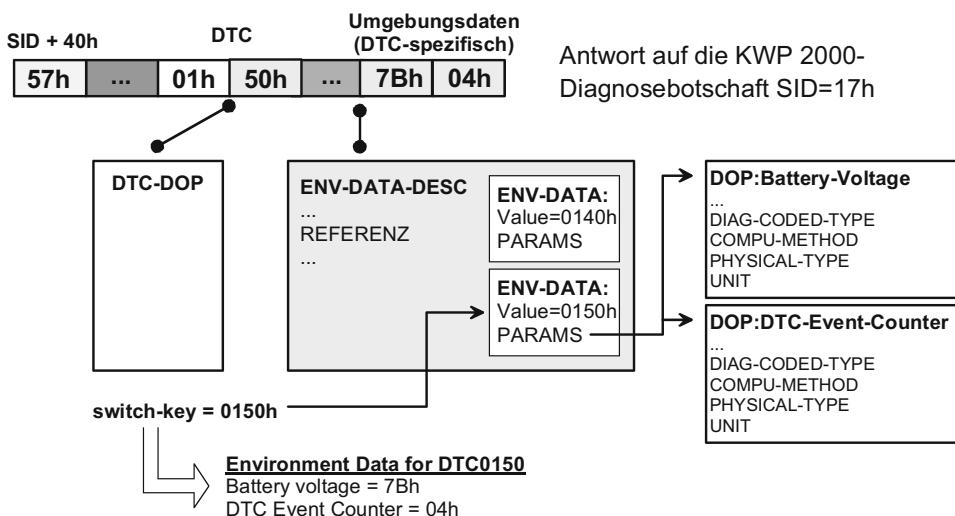
Complex DOP: Komplexe Datenobjekte Komplexe Datenobjekte sind aus einfachen Datenobjekten zusammengesetzte Strukturen (Tab. 6.22). Sie werden für die Definition und Interpretation von komplexen Parametern von Diagnosediensten verwendet, wenn die Art und die Zusammensetzung der Parameter erst zur Laufzeit des Systems bestimmt werden können. Complex DOPs werden in derselben Weise durch PARAM-Abschnitte referenziert wie gewöhnliche DOPs.

Tab. 6.21 Beispiel eines DTC-DOP

```

<DTC-DOP ID="DTCDOP_ALL">
  <SHORT-NAME>DTCDOP_ALL</SHORT-NAME>
  .
  .
  <DTCS>
    <DTC ID="DTC0140">
      <SHORT-NAME>DTC0140</SHORT-NAME>
      <TROUBLE-CODE>320</TROUBLE-CODE>
      <DISPLAY-TROUBLE-CODE>Error_140</DISPLAY-TROUBLE-CODE>
      <TEXT>Ashtray Illumination Defect</TEXT>
      <LEVEL>2</LEVEL>
    </DTC>
  .
  .
</DTCS>
</DTC-DOP>

```

**Abb. 6.36** Auslesen von Umgebungsbedingungen aus dem Fehlerspeicher

Das nachfolgende Beispiel zeigt den Aufbau eines komplexen Datenobjekts für die Dokumentation eines Fehlerspeichereintrages (DTC – Diagnostic Trouble Code). Ein Steuergerät legt für die spätere Analyse eines Fehlers die aktuell herrschenden Umgebungsbedingungen mit ab. Diese Daten sind eine Ansammlung unterschiedlicher physikalischer Größen, die strukturiert abgelegt werden müssen. Als Beispiel zeigt Abb. 6.36, wie der Diagnosetester die Umgebungsbedingungen zum Fehler mit der Fehlernummer DTC = 336 = 150 h ausliest. Die einzelnen Umgebungsbedingungen, in diesem Fall die Batteriespannung und ein Fehlerzähler (Event Counter), werden vom Steuergerät als Parameter hintereinander in der Antwortbotschaft zurückgesendet.

In Tab. 6.23 ist die zugehörige ODX-Beschreibung dargestellt. ODX sieht für Umgebungsbedingungen einen ENV-DATA-DESC-Abschnitt mit ENV-DATA-Feldern vor. Die-

Tab. 6.22 Typen von komplexen Datenobjekten in ODX

Type	Beschreibung
STRUCTURE	Vergleichbar mit einer Struktur innerhalb der Programmiersprache C, Zusammenfassung mehrerer DOPs (simple oder complex). Bsp.: Lese den letzten Eintrag des Steuergeräte-Historienspeichers für die Reprogrammierung.
STATIC-FIELD	Wird verwendet, wenn sich eine STRUCTURE innerhalb eines Bytestroms mehrfach mit fester Anzahl wiederholt. Bsp.: Lese die letzten drei Einträge des Steuergeräte-Historienspeichers für die Reprogrammierung.
DYNAMIC-LENGTH-FIELD	Wird verwendet, wenn sich eine STRUCTURE innerhalb eines Bytestroms wiederholt und die Anzahl der Wiederholungen nur dynamisch, d. h. erst zur Laufzeit, bekannt ist. Bsp.: Lese alle Einträge des Steuergeräte-Historienspeichers für die Reprogrammierung.
DYNAMIC-ENDMARKER-FIELD	Ein DYNAMIC-ENDMARKER-FIELD wird verwendet, wenn sich eine STRUCTURE innerhalb eines Bytestroms solange wiederholt, bis ein definierter TERMINATION-VALUE erkannt wird. Bsp.: Lese alle Einträge des Steuergeräte-Historienspeichers für die Reprogrammierung, wobei das Steuergerät als Abschlusskennzeichnung den Wert FFh sendet.
MUX	Ein Multiplexer (MUX) wird verwendet, wenn die Interpretation eines Datenstroms von einem bestimmten Parameter (SWITCH-KEY) abhängt. Über den SWITCH-KEY wird ein entsprechender CASE selektiert, der wiederum auf eine Parameterstruktur verweist. So können für alle möglichen Sub-Identifier eines Diagnosedienstes die jeweils definierten Parameterstrukturen modelliert werden.
END-OF-PDU-FIELD	Ein END-OF-PDU-FIELD wird verwendet, wenn sich eine STRUCTURE innerhalb eines Bytestroms bis zu dessen Ende wiederholt.
TABLE	Wird verwendet, wenn Datenstrukturen über Kennwerte (Data Identifier) identifiziert werden wie bei den KWP 2000/UDS-Diagnosediensten <i>Read/Write Data By Identifier</i> . Eine TABLE besteht aus einer KEY- und einer STRUCTURE-Spalte. Die KEY-Spalte beinhaltet die entsprechenden Data Identifier des Diagnosedienstes. Die STRUCTURE-Spalte verweist auf die zugehörigen Datenstrukturen, deren Inhalte in entsprechenden STRUCTURE-Abschnitten definiert sind.

ses komplexe Datenobjekt enthält die Umgebungsbedingungen für mehrere Fehler und ist daher als Multiplexer (MUX, vgl. Tab. 6.22) aufgebaut. Zur Auswahl (SWITCH-KEY) dient die Fehlernummer, die über DTC-VALUE mit $336 = 150$ h festgelegt ist. Die ODX-Datei gibt an, dass zu diesem Fehlercode die Umgebungsparameter Batteriespannung und Fehlerzähler gehören und verweist (DOP-REF) auf die zugehörigen einfachen Datenobjekte, über die die Umrechnung zwischen hexadezimalem und physikalischem Wert, der Wertebereich, die Einheit usw. festgelegt werden. Am Beispiel der DTC-DOPs lässt sich wieder

Tab. 6.23 ODX-Beschreibung zu Abb. 6.36

```

<ENV-DATA-DESC ID="ENVDESC_EnvDataDesc1">
  <SHORT-NAME>ENVDESC_EnvDataDesc1</SHORT-NAME>
  <PARAM-SNREF SHORT-NAME="SwitchKeyDTC">
  <ENV-DATAS>
    <ENV-DATA ID="ED_1">
      <SHORT-NAME>ED_DTC0140</SHORT-NAME>
      .
      .
    </ENV-DATA>

    <ENV-DATA ID="ED_2">
      <SHORT-NAME>ED_DTC0150</SHORT-NAME>
      <DTC-VALUES>
        <DTC-VALUE>336<DTC-VALUE>
      </DTC-VALUES>
      <PARAMS>
        <PARAM xsi:type="VALUE">
          <SHORT-NAME>BatteryVoltage</SHORT-NAME>
          <BYTE-POSITION>0</BYTE-POSITION>
          <DOP-REF ID-REF="DOP_BatteryVoltage">
        </PARAM>
        <PARAM xsi:type="VALUE">
          <SHORT-NAME>EventCounter</SHORT-NAME>
          <BYTE-POSITION>1</BYTE-POSITION>
          <DOP-REF ID-REF="DOP_EventCounter">
        </PARAM>
      </PARAMS>
    </ENV-DATA>
  </ENV-DATAS>
</ENV-DATA-DESC>

```

das Single-Source-Prinzip von ODX erkennen. Sind Fehlercodes bereits an einer Stelle definiert, so wird an einer anderen Stelle nur auf die dortige Definition verwiesen (Referenz).

Special Data Groups: Herstellerspezifische Daten Über sogenannte SDG-Datenobjekte kann ein Hersteller Datensätze in ODX-Beschreibungen aufnehmen, für die der Standard keine Datenobjekte vorgesehen hat. Standard-ODX-Werkzeuge dürfen solche herstellerspezifischen Erweiterungen ignorieren.

6.6.7 SINGLE-ECU-JOB und MULTIPLE-ECU-JOB: Diagnoseabläufe

Häufig müssen bei der Diagnose bestimmte Abläufe ausgeführt werden, die aus der sequenziellen Abarbeitung verschiedener Diagnosebotschaften bestehen, wobei die Parameter oder die Art der Folgebotschaften von den Ergebnissen der vorigen Botschaften abhängt. Ein Beispiel dafür ist der *Seed-and-Key*-Mechanismus, mit dem der Zugriff auf bestimmte Steuergerätefunktionen freigeschaltet wird. Dabei fordert der Diagnosetester vom Steuergerät mit einer ersten Botschaft einen Initialisierungswert an, berechnet daraus dann

Tab. 6.24 Single-ECU-Job: ODX-Beschreibung und Java-Programm

ODX-Beschreibung: <pre> <SINGLE-ECU-JOB ID="ID_734711" DIAGNOSTIC-CLASS="COMMUNICATION"> <PROG-CODES> <PROG-CODE> //Verweis auf JAVA-Programm <CODE-FILE>Beispiel.java</CODE-FILE> . </PROG-CODE> </PROG-CODES> <INPUT-PARAMS> <INPUT-PARAM> <SHORT-NAME>Address</SHORT-NAME> <DOP-BASE-REF ID-REF="DOP-3BYTE-IDENTICAL" /> </INPUT-PARAM> . </INPUT-PARAMS> <OUTPUT-PARAMS> <OUTPUT-PARAM ID="ID_734712"> <SHORT-NAME>Returned_Value</SHORT-NAME> <DOP-BASE-REF ID-REF="DOP-ByteArray" /> </OUTPUT-PARAM> </OUTPUT-PARAMS> </SINGLE-ECU-JOB> </pre>
--

einen Schlüssel und sendet diesen an das Steuergerät zurück. Für derartige Szenarien sieht ODX sogenannte *Jobs* vor, mit denen ein solcher Ablauf definiert werden kann.

Falls ein solcher *Job* Botschaften an mehrere Steuergeräte umfasst, muss er im separaten ODX-Dokument MULTIPLE-ECU-JOB definiert werden, ansonsten wird er als SINGLE-ECU-JOB einem DIAG-COMM-Element zugeordnet (Abb. 6.32). Bei einem MULTIPLE-ECU-JOB muss das Laufzeitsystem dann mehrere gleichzeitige logische Verbindungen (Logical Links) zu den einzelnen Steuergeräten aufbauen. Ein typischer Anwendungsfall für einen MULTIPLE-ECU-JOB ist die Identifikation aller Geräte eines Fahrzeugs. Dabei wird gleichzeitig mit allen Steuergeräten kommuniziert, um deren Identifikationsdaten auszulesen. Anstatt vom Diagnosesystem nacheinander alle Geräte abzufragen und die Informationen in der Bedienebene oberhalb des Laufzeitsystems zu sammeln, kann durch einen MULTIPLE-ECU-JOB bereits im Laufzeitsystem eine Bündelung der Daten erfolgen.

Der eigentliche Ablauf eines *Jobs* wird in einer separaten Datei in der Programmiersprache Java beschrieben, auf die in der ODX-Datei verwiesen wird (Tab. 6.24). Darüber hinaus definiert die ODX-Datei, welche Parameter an die Java-Funktion zu übergeben und wie deren Rückgabewerte zu interpretieren sind. Für derartige Java-Abläufe stellt ASAM ein Java-Paket asam.d.* zur Verfügung, das die im Beispielprogramm eingesetzten, mit MCD_... beginnenden Hilfsfunktionen und Klassendefinitionen enthält. Optional kann ein ASAM-Laufzeitsystem herstellerabhängig auch andere Programmiersprachen unterstützen, z. B. in C/C++ erstellte Windows-DLL-Funktionen.

JAVA-Programm in Datei Beispiel.java:

```

import asam.d.*;

public class JobExample implements JobTemplate
{
    ...
    public void execute( MCDRequestParameters inputParameters,
                         MCDJobApi jobHandler, MCDLogicalLink link,
                         MCDSingleEcuJob apiJobObject
                         ) throws MCDEception
    {
        MCDService service;
        MCDResult serviceResult, jobResult;
        MCDResultState resultState;
        MCDResponse serviceResponse, jobResponse;
        MCDResponseParameter value;

        // Create the ReadMemoryByAddress diagnostic service to be executed
        service = link.createService(link.getDbObject()
                                      .getDBLocation.getServices()
                                      .getItemByName("ReadDataByAddress"));

        // Set variable request parameters of the service
        service.setParameterValue("Address", inputParameters
                                      .getItemByName("Address").getValue());
        ...

        resultState = service.executeSync(); // Execute the service
        serviceResult = resultState.getResults().getItemByIndex(1);
        serviceResponse = serviceResult.getResponses()
                                      .getItemByIndex(1));
        // Create the job result and response and copy the second service
        // response parameter into the job response
        jobResult = apiJobObject.createResult(...);
        jobResponse = jobResult.getResponses().add(...);
        jobResponse.getResponseParameters.addElementWithContent(
            serviceResponse.getResponseParameters().getItemByIndex(2));
        link.deleteComPrimitive(service); // Delete the service
        jobHandler.sendFinalResult(jobResult); // Return result
        return;
    }
}

```

6.6.8 STATE-CHART: Diagnosesitzungen

ODX bildet das Konzept der Diagnosesitzungen (vgl. Abschn. 5.1.2) auf einen Zustandsautomaten (STATE-CHART) ab, der Teil eines DIAG-LAYER-Elements (Abschn. 6.6.2) ist. Für diesen Zustandsautomaten werden Zustände (STATE) und Übergänge zwischen den Zuständen (STATE-TRANSITION) definiert (Abb. 6.37)

Bei den Diagnosediensten und -abläufen (DIAG-COMM-Elemente) kann durch einen Verweis (STATE-TRANSITION-REF) auf einen entsprechenden Zustandsübergang angegeben werden, dass der Zustandsautomat in einen anderen Zustand wechseln soll, wenn der Diagnosedienst erfolgreich ausgeführt wurde. Umgekehrt kann mit Hilfe von PRE-CONDITION-STATE-REF-Verweisen auf einen oder mehrere Zustände festgelegt werden,

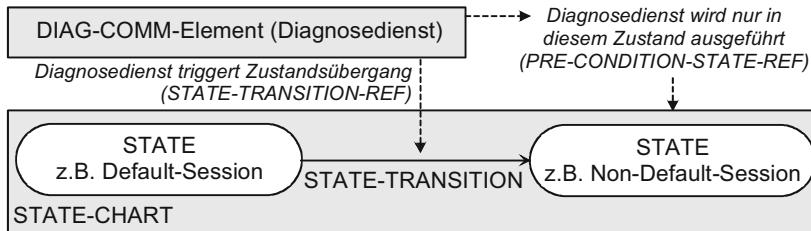


Abb. 6.37 Zustandsautomat für Diagnosesitzungen

dass der Diagnosedienst nur ausgeführt werden darf, wenn sich das Steuergerät in den dabei definierten Zuständen befindet.

6.6.9 ECU-CONFIG: Beschreibung der Steuergeräte-Konfiguration

Unter der Konfiguration oder Parametrisierung eines Steuergerätes (umgangssprachlich als *Codieren* bezeichnet) versteht man die Aktivierung bestimmter interner Einstellungen oder Funktionalitäten eines Steuergerätes. So kann zum Beispiel bei einem Steuergerät für das Außenlicht konfiguriert werden, ob ein Glühfaden-Scheinwerfer oder ein Xenon-Scheinwerfer angesteuert werden soll. Bei der Konfiguration eines Steuergerätes wird mittels eines Diagnosedienstes (z. B. UDS *WriteDataByAddress*, siehe Kap. 5) ein bestimmter Datensatz an eine definierte Adresse im Steuergerät geschrieben. Das Steuergerät wertet dann zur Laufzeit das Bitmuster des Datensatzes aus und setzt intern die entsprechende Konfiguration. Ein solcher Datensatz besteht aus der seriellen Anordnung der einzelnen Informationen und kann, je nach Anzahl der zu konfigurierenden Parameter, zwischen einem Bit und mehreren Byte lang sein. Das ODX-Datenmodell bietet abhängig vom Anwendungsfall zwei Möglichkeiten für den Umgang mit solchen Datensätzen.

Während der Entwicklung werden die Konfigurationsdatensätze je nach gewünschter Funktionalität einzeln zusammengesetzt (Abb. 6.38). Daher muss in ODX die Bedeutung jedes einzelnen Elementes (CONFIG-ITEM) des Datensatzes (CONFIG-RECORD) hinterlegt sein.

In der Produktion sind die Datensätze für ein Steuergerät in einem bestimmten Fahrzeug in der Regel bereits erstellt. Die Konfiguration (CONFIG-RECORD) wird hier als kompletter Datensatz (DATA-RECORD) mit einem Diagnosedienst ins Steuergerät geschrieben.

Mit einem weiteren Diagnosedienst kann die Konfiguration auch aus dem Steuergerät ausgelesen werden. Durch die Wandlung der einzelnen CONFIG-ITEMs in ihren physikalischen Wert (DOP) kann der Konfigurationsstand des Steuergerätes ermittelt werden.

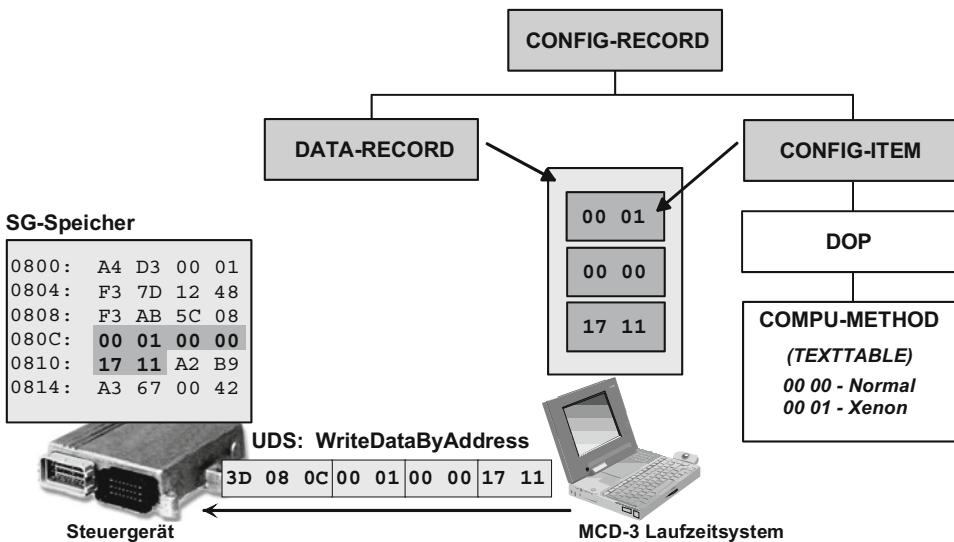


Abb. 6.38 Vereinfachte Darstellung für die Steuergerätekonfiguration

6.6.10 ECU-MEM: Beschreibung der Flash-Programmierung

Wie oben bereits erwähnt, ist das Reprogrammieren von Steuergeräten, umgangssprachlich als *Flashen* bezeichnet, ebenfalls Bestandteil der Steuergeräte-Diagnose. Dabei wird durch eine definierte Abfolge von Diagnosiediensten der Flash-Speicher des Steuergerätes gelöscht und neue Software programmiert. Eine ausführliche Beschreibung dieses Vorgangs erfolgt in Abschn. 9.4. ODX beschreibt innerhalb des ECU-MEM-Dokumentes einen Datencontainer, in dem alle relevanten Informationen für die Durchführung eines solchen Programmierungsvorgangs auf der Seite des Diagnosetesters abgelegt sind. Innerhalb dieses Datencontainers definiert ODX sogenannte SESSIONS, mit denen die zu programmierenden Daten verwaltet werden. Abbildung 6.39 zeigt den prinzipiellen Aufbau dieser Verwaltung am Beispiel des Tacho-Steuergerätes mit der Variante für Europa (km/h) und USA (mph).

Die Steuergerätesoftware besteht aus dem Codeteil (Applikation) und dem Datenteil (Parametersatz). Die Applikation ist für beide Varianten gleich, der Parametersatz dagegen ist in der mph- und in der km/h-Variante unterschiedlich. Zusätzlich benötigt das Steuergerät zur Durchführung der Programmierung noch einen Flash-Treiber, der dynamisch in das Steuergerät geladen wird und dort den eigentlichen Programmierungsvorgang durchführt. Diese vier Dateien werden in der ODX-Beschreibung durch das Anlegen von zwei SESSIONS (Programmiersitzungen) logisch gegliedert. In der *Session 1* wird für die USA-Variante auf die Dateien *Parametersatz mph*, *Applikation* und *Flash-Treiber* referenziert, *Session 2* (Europa-Variante) dagegen verweist auf *Parametersatz km/h*, *Applikation* und *Flash-Treiber*. Das Laufzeitsystem hat nach Auswahl der entsprechenden SESSION Zugriff auf alle relevanten Informationen.

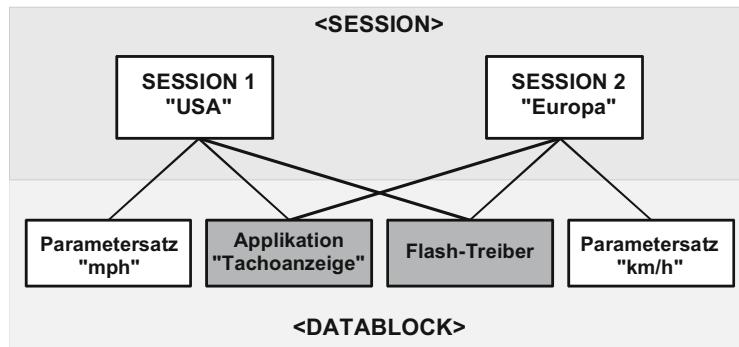


Abb. 6.39 Programmierdatensätze für ein Tachometer-Steuergerät

Abbildung 6.40 zeigt stark vereinfacht den Aufbau des Datencontainers. Ein DATA-BLOCK beschreibt die Struktur des Adressbereiches, der zu programmieren ist. Dazu werden zusammenhängende Datenblöcke der Programmierdaten (FLASHDATA) innerhalb von ODX auf SEGMENTe abgebildet und durch SOURCE-START-ADDRESS und SOURCE-END-ADDRESS abgegrenzt. Jeder DATA-BLOCK besitzt mindestens ein SEGMENT. Tabelle 6.25 zeigt die zugehörige ODX-Beschreibung.

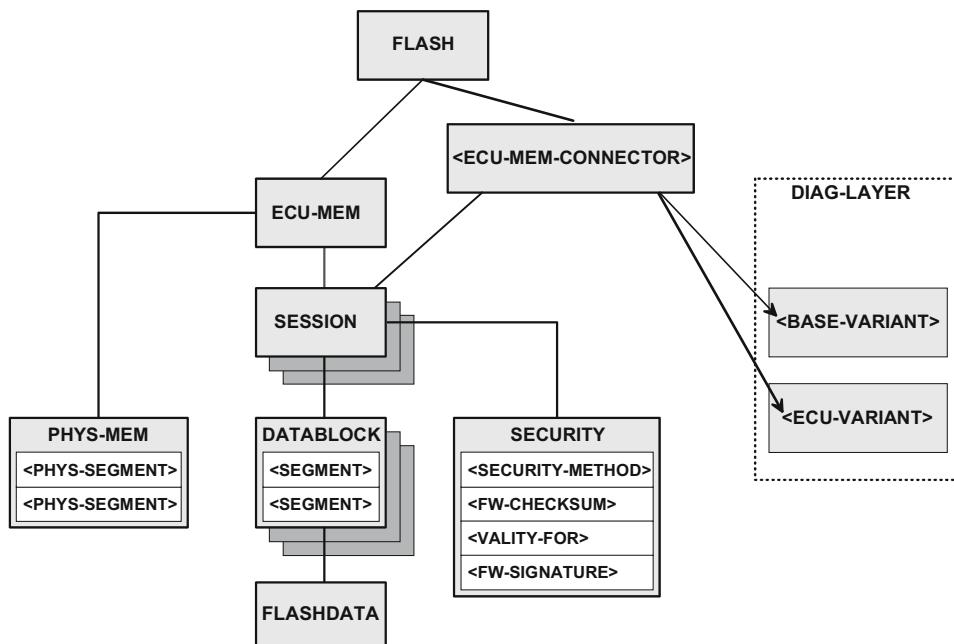


Abb. 6.40 ODX-Beschreibungsstruktur für die Flash-Programmierung (vereinfacht)

Tab. 6.25 ODX-Beschreibung von Speichersegmenten

```

<SEGMENTS>
  <SEGMENT ID="SEGMENT01">
    <SHORT-NAME>SEGMENT01</SHORT-NAME>
    .
    <SOURCE-START-ADDRESS>0 000</SOURCE-START-ADDRESS>
    <SOURCE-END-ADDRESS>3 FFF</SOURCE-END-ADDRESS>
  </SEGMENT>
  <SEGMENT ID="SEGMENT02">
    <SHORT-NAME>SEGMENT02</SHORT-NAME>
    .
    <SOURCE-START-ADDRESS>4 000</SOURCE-START-ADDRESS>
    <SOURCE-END-ADDRESS>4 FFF</SOURCE-END-ADDRESS>
  </SEGMENT>
.
</SEGMENTS>

```

Die Flash-Programmierung wird durch mehrere Prüfmechanismen abgesichert. Um unbefugte Manipulationen zu verhindern, wird ein Zugriffsschutz implementiert (z. B. *Seed and Key*). Der Algorithmus für die Berechnung des *Keys* aus dem *Seed* muss nicht nur dem Steuergerät sondern auch dem Laufzeitsystem bekannt sein. Ebenso werden die Informationen zur Verifikation der Programmierung wie Prüfsummen und Signaturen benötigt. Diese Daten werden innerhalb von ODX in SECURITY-Elementen abgelegt und einer Session bzw. einem Datenblock zugewiesen.

Innerhalb von PHYS-MEM wird die real existierende physikalische Speichertopologie des Steuergerätes beschrieben. Ein Laufzeitsystem kann somit prüfen, ob die logischen Segmente der Flash-Daten mit der tatsächlich existierenden Speichertopologie identisch sind. Dazu wird der Flash-Speicher durch PHYS-SEGMENTe bestehend aus START-ADDRESS, END-ADDRESS und BLOCKSIZE modelliert.

Der eigentliche Programmervorgang erfolgt durch die sequenzielle Abarbeitung von definierten Diagnosedielen (Programmiersequenz). Wie oben bereits erwähnt, beschreibt ODX eine solche Abfolge von Diagnosedielen in einem Job. Damit die Diagnosedielen des Flash-Jobs auf die Flash-Daten zugreifen können, muss eine Verbindung zwischen dem Flash-Job im Dokument DIAG-LAYER und einer SESSION im Dokument ECU-MEM hergestellt werden. Diese Verbindung wird durch den ECU-MEM-CONNECTOR vollzogen.

6.6.11 FUNCTION-DICTIONARY: Funktionsorientierte Diagnose

Die funktionsorientierte oder funktionsbezogene Diagnose stellt eine abstrakte Sichtweise der Diagnose dar. Dabei werden innerhalb eines Fahrzeugs nicht die einzelnen Steuergeräte betrachtet, sondern eine bestimmte Funktionalität im Gesamtsystem.

An der Umsetzung einer Funktion im Fahrzeug können durchaus mehrere Steuergeräte beteiligt sein. Die Funktionalität *Außenbeleuchtung* besteht beispielsweise aus einem

Steuergerät für die vorderen und einem für die hinteren Lampen. Die Schalterstellung am Armaturenbrett wird dabei von einem weiteren Steuergerät eingelesen. Für eine Funktion „LampenTest“, bei der die gesamte Außenbeleuchtung ein- und wieder ausgeschaltet wird, müssen also beide Lampensteuergeräte mit dem entsprechenden Diagnosedienst für das Ein- bzw. Ausschalten angesprochen werden. Ein Anwender muss daher exakte Kenntnisse haben, welche Steuergeräte bei der Umsetzung der Funktion involviert sind und welche Diagnosedienste die gewünschte Funktionalität erzeugen. Bei der funktionsorientierten Diagnose sind diese Informationen gekapselt. Der Anwender startet die Funktion „LampenTest“ und das Laufzeitsystem ermittelt über das ODX-Dokument FUNCTION-DICTIONARY die beteiligten Komponenten und die notwendigen Diagnosedienste für die Funktionsausführung. Speziell bei komplexen Systemen mit vielen beteiligten Komponenten (z. B. Fahrwerkregelung bzw. Assistenzsysteme wie ESP) bietet diese Sichtweise einen großen Vorteil. So werden bei einem System durch „Fehlerspeicher_Lesen“ alle Fehlerspeicher der beteiligten Komponenten ausgelesen. Durch entsprechende Analyse der Einträge kann so ein Fehlerbild auf Systemebene erstellt werden.

Die einzelnen Funktionen werden in FUNCTION-NODES organisiert (Abb. 6.41). Über Referenzen auf ausführbare (EXECUTABLE) Dienste (DIAG-COM) und MULTIPLE-ECU-JOBs innerhalb des DIAG-OBJECT-CONNECTORS werden die Verbindungen zu den einzelnen Steuergeräten hergestellt.

6.6.12 Packaged ODX und ODX-Autorenwerkzeuge

ODX-Datensätze werden von verschiedenen Stellen bei Fahrzeugherstellern und Zuliefern erstellt und gepflegt, wobei jeder Datensatz aus einer ganzen Reihe von XML- und gegebenenfalls Binärdateien besteht. Außerdem „leben“ die Datensätze, d. h. sie werden laufend weiterentwickelt, so dass Änderungsstände dokumentiert und versioniert werden müssen.

ASAM hat zu diesem Zweck unter der Bezeichnung *Packaged ODX* (PDX) ein Archivformat definiert, mit dem verschiedene, zu einem Projekt gehörende ODX-Dateien zusammengefasst werden können. Im Wesentlichen handelt es sich dabei um das aus der Rechnerwelt bekannte ZIP-Format zur komprimierten Speicherung von Verzeichnis und Dateistrukturen ergänzt um den *PDX-Catalogue*, eine XML-Datei, die das Inhaltsverzeichnis der PDX-Datei enthält.

Die Erstellung von konsistenten ODX-Datensätzen für ein Fahrzeug ist eine komplexe Aufgabe, die ohne entsprechende Autorenwerkzeuge nicht sinnvoll bewältigt werden kann. Beispiele solcher Werkzeuge werden in Abschn. 9.6 dargestellt.

6.6.13 ODX Version 2.2 und ISO 22901

Die aktuellste ODX-Version, mittlerweile als ISO 22901 standardisiert, beschreibt insbesondere Anwendungsszenarien für die Flash-Programmierung und die Handhabung von

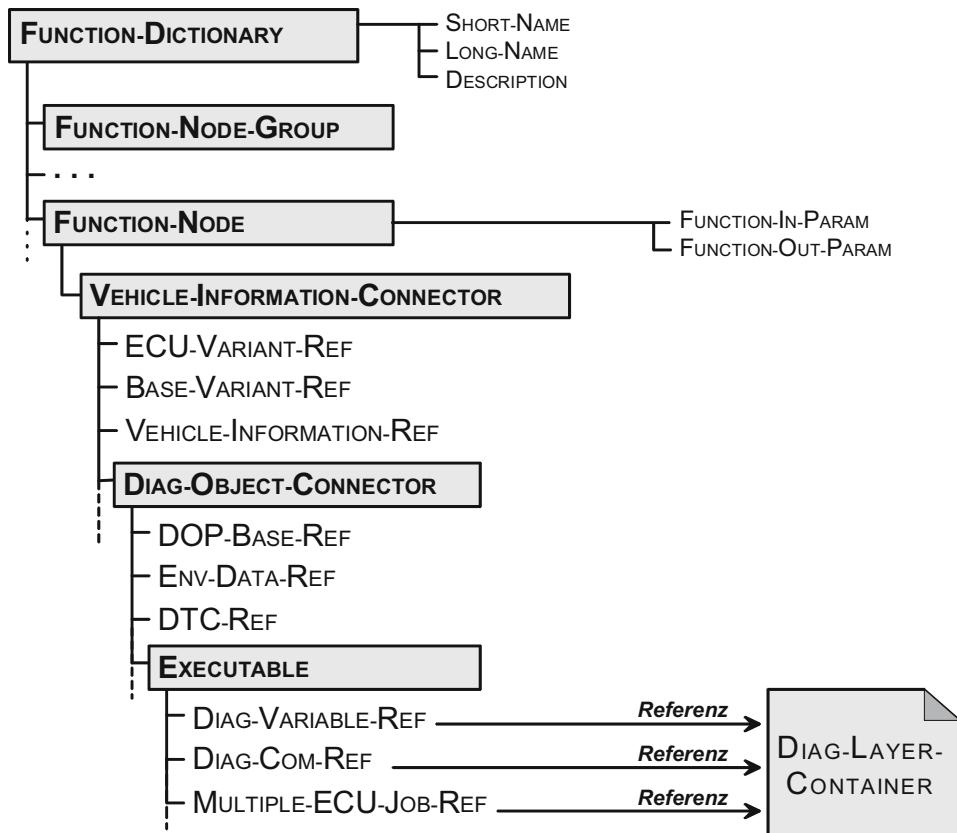


Abb. 6.41 Aufbau des FUNCTION-DICTIONARY-Dokumentes (vereinfacht)

Varianten ausführlicher als die Vorgänger. Zu den wenigen echten Neuerungen gehören LIBRARY-Elemente für ECU-JOBS, um Programmbibliotheken anzulegen und eine einfache Versions- und Zugriffsverwaltung für diese Bibliotheken.

6.7 ASAM AE MCD 3-Server

Die Grundstruktur eines MCD 3-Server wurde bereits in Abb. 6.22 dargestellt. Die Datenbasis des Servers ist in sogenannten Projekten organisiert, in denen beispielsweise die Daten eines bestimmten Fahrzeugmodells zusammengefasst werden.

Der MCD 3-Server kennt drei Grundzustände (Abb. 6.42). Zu Beginn wählt der Anwender über den entsprechenden Bedienclient eines der Projekte aus und kann dann zunächst die in der Datenbank für dieses Projekt gespeicherten Werte einsehen. Sobald die Verbindung zu einem Steuergerät hergestellt ist, können dann auch aktuelle Werte ausgelesen oder verändert werden.

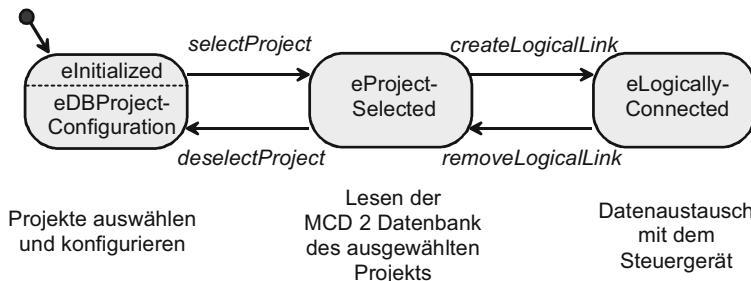


Abb. 6.42 Grundzustände des MCD 3-Servers (vereinfacht)

Wie Projekte innerhalb der Datenbank zu organisieren und über den Server zu verwalten sind, wird in der MCD 2/3-Spezifikation im Einzelnen nicht geregelt, sondern ist herstellerabhängig.

Das MCD 3-Objektmodell sieht eine strenge Trennung zwischen Datenobjekten (*Database Object DB*) und Laufzeitobjekten (*Runtime Object*) vor (Abb. 6.43). Die Datenobjekte repräsentieren den Inhalt der MCD 2-Datenbanken und sind statisch, d. h. sie können nicht verändert werden, während die Laufzeitobjekte die Interface-Methoden für den oder die Bedienclients bereitstellen. Die statischen Objekte bilden im Wesentlichen die Elemente der ODX-Diagnose- bzw. ASAP2-Applikationsdatensätze ab.

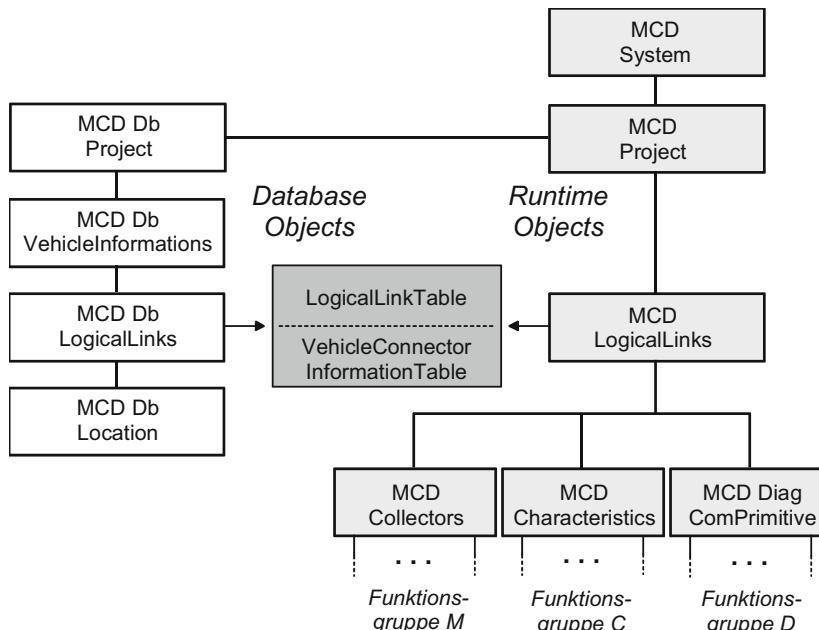


Abb. 6.43 Vereinfachte Grundstruktur des MCD 3-Objektmodells

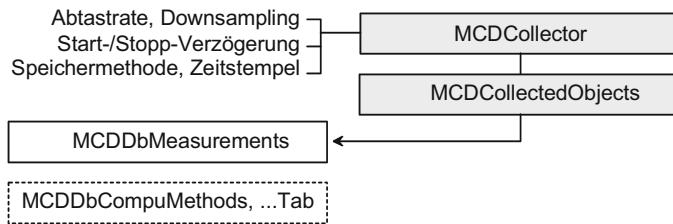


Abb. 6.44 Hauptelemente der Funktionsgruppe M – Messen

Viele der Objekte existieren mehrfach und werden in Listen (*Collections*) verwaltet, auf die über einen laufenden Index bzw. den Namen (ODX-Element **SHORT-NAME**) der Objektinstanz zugegriffen werden kann. Als Name einer solchen Liste wird üblicherweise der Name der enthaltenen Elemente im Plural verwendet, z. B. *MCDDbLogicalLinks* für die Liste aller im Projekt vorhandenen *MCDDbLogicalLink* Elemente aus der ODX-Datei (vgl. Abschn. 6.6.3).

Clients können über Schnittstellenmethoden der Laufzeitobjekte nicht nur selbst Werte abfragen und setzen sondern sich durch das Installieren von sogenannten Event-Handlern auch über auftretende Ereignisse, z. B. die Unterbrechung einer logischen Verbindung, automatisch vom Laufzeitsystem benachrichtigen lassen.

6.7.1 Funktionsgruppe M – Messen

Zentrales Objekt dieser Funktionsgruppe ist der *Collector* (Abb. 6.44). Seine Aufgabe besteht darin, selbstständig Messungen durchzuführen und Messwerte abzuspeichern. Alle innerhalb eines Collector-Objektes durchgeführten Messungen werden mit derselben Abtastrate durchgeführt, wobei jeder Messwert in einem Measurement-Abschnitt der ASAP2-Datei (vgl. Abschn. 6.5) definiert sein muss. Innerhalb eines Collectors können nur Messwerte über einen einzigen Logical Link, d. h. aus einem einzigen Steuergerät, erfasst werden, es ist aber möglich, mehrere Collector-Objekte gleichzeitig anzulegen. Die Messwerte können wahlweise mit einem Zeitstempel abgespeichert werden, so dass nachträglich eine zeitliche Korrelation mehrerer Messdatensätze möglich ist. Nachdem ein Collector-Objekt erzeugt und die gewünschten Messungen konfiguriert wurden, werden die Messungen aktiviert und starten bzw. stoppen nach Vorliegen der in der ASAP2-Datenbank definierten Triggerbedingungen sowie der über Collector-Attribute einstellbaren Verzögerungszeiten automatisch.

Die Messwerte werden in einem Ringpuffer mit einstellbarer Größe abgelegt und können vom Client durch Polling oder mit Unterstützung von Events fortlaufend ausgelesen werden. Alternativ kann das Laufzeitsystem die Messdaten auch selbstständig in einer Datei ablegen. Die Umrechnung zwischen geräteinternen und physikalischen Werten über-

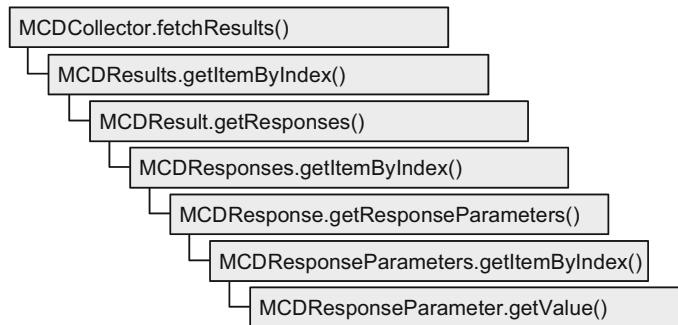


Abb. 6.45 Zugriff auf Messergebnisse über *MCDResult* und *MCDResponse*

nehmen die in der ASAP2-Datenbank als *CompuMethods* und *CompuTabs* definierten Umrechnungsformeln.

Optional können von einem ASAM MCD 3-Laufzeitsystem auch Kennlinien und Kennfelder (*Characteristics*) gemessen werden (im Abb. 6.44 nicht dargestellt).

Der Zugriff auf die Messergebnisse erfolgt über eine Kette von Funktionsaufrufen (Abb. 6.45). `MCDCollector::fetchResults()` liefert eine Liste der Ergebnisse aller Messungen des Collector-Objekts, aus denen dann das Ergebnis einer einzelnen Messung (*MCDResult*), daraus dann eine Liste der zum selben Abtastzeitpunkt erfassten Messwerte (*MCDResponse*) und damit schließlich ein einzelner Wert (*MCDResponseParameter*) extrahiert werden kann.

6.7.2 Funktionsgruppe C – Kalibrieren

Die Funktionsgruppe C erlaubt den Zugriff auf die steuergeräteinternen Parameter (Abb. 6.46). Diese sind in der ASAP2-Datenbank (vgl. Abschn. 6.5) als skalare Werte (*Scalar*), Kennlinien (*Curve*) und Kennfelder (*Map*) beschrieben.

Die unabhängigen Größen werden dabei als Achsen (*Axis*) bezeichnet, die abhängigen als Werte (*Value*). Bei Kennlinien und Kennfeldern bestehen beide aus mehreren Daten-

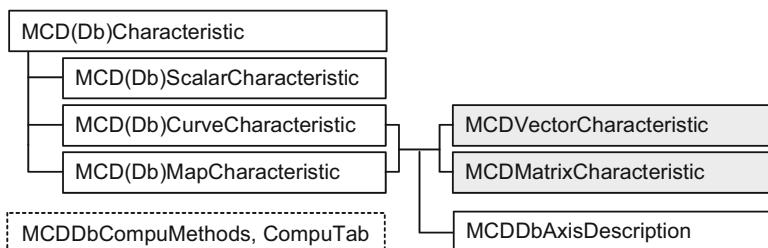
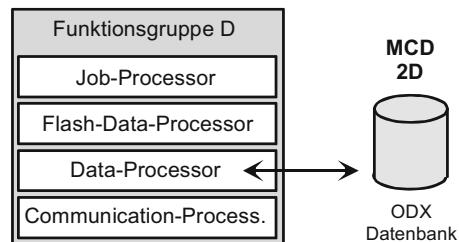


Abb. 6.46 Hauptelemente der Funktionsgruppe C – Kalibrieren

Abb. 6.47 Struktur der Funktionsgruppe D – Diagnose



werten, die zur einfacheren Handhabung zu Vektoren (*Vector*) oder Matrizen (*Matrix*) zusammengefasst werden. Für jeden in der Datenbank vordefinierten Wert, jede Kennlinie oder jedes Kennfeld kann ein Laufzeitobjekt angelegt werden. Dessen Eigenschaften können über *get...()*-Methoden abgefragt und dessen Werte über *read()* bzw. *write()*-Methoden gelesen und gesetzt werden. Die Umrechnung zwischen geräteinternen und physikalischen Werten übernehmen die in der ASAP2-Datenbank als *CompuMethods* und *CompuTabs* definierten Umrechnungsformeln.

6.7.3 Funktionsgruppe D – Diagnose

Da diese Funktionsgruppe wesentlich komplexer ist als die beiden anderen, gibt die MCD 3-Spezifikation nicht nur die äußere Schnittstelle sondern auch den inneren Aufbau präziser vor (Abb. 6.47). Der Diagnose-Block besteht aus sogenannten Prozessoren, die jeweils eindeutige Aufgaben besitzen:

- Der **Communication Processor** erzeugt Diagnosebotschaften, sendet diese zum Steuergerät und verarbeitet dessen Antworten.
- Der **Data Processor** bildet die einzige Schnittstelle zur ODX-Datenbank. Sämtliche Datenabfragen erfolgen über den Daten-Processor. Außerdem ist er für die Umwandlung der steuergeräteinternen Werte in die physikalischen Werte und umgekehrt zuständig.
- Der **Job Processor** führt vom Anwender bereitgestellte Funktionen aus, mit denen Sequenzen von Diagnosebotschaften erzeugt und deren Ergebnisse verarbeitet werden (vgl. Abschn. 6.6.7). Die Funktionen selbst werden als Java-Quellprogramme (*Java File*), als vorübersetzte Java-Bytecode-Programme (*Class File*) oder als Java-Archiv (*JAR File*) über die ODX-Datenbank bereitgestellt.
- Der **Flash Data Processor** bildet das Laufzeitsystem für die Abwicklung von Flash-Programmieraufgaben, wie bereits in Abschn. 6.6.10 beschrieben. Die Programmierdatensätze werden wiederum über die ODX-Datenbank bereitgestellt.

Hauptelement der Funktionsgruppe D ist *MCBDiagComPrimitive*, von dem, wie von den meisten anderen Objekten, sowohl ein Datenbank- als auch ein Laufzeitobjekt existiert (Abb. 6.48). Dahinter verbergen sich Diagnosedienste (*MCDDiagService*) und Jobs (*MCD-*

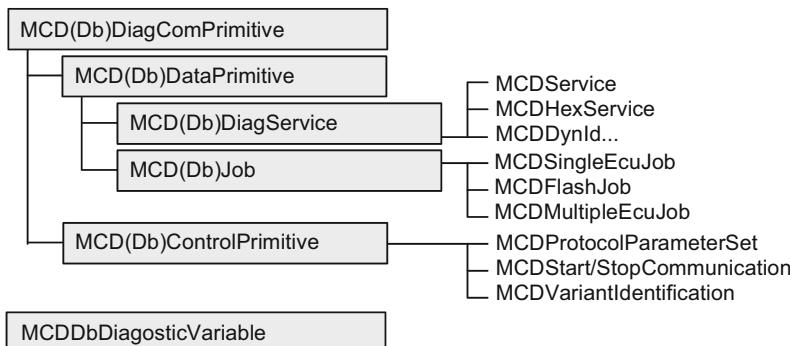


Abb. 6.48 Hauptelemente der Funktionsgruppe D

Job) sowie Schnittstellen zur Steuerung der Kommunikation zu den Steuergeräten (*MCD-ControlPrimitive*). Diagnosedienste können mit Hilfe der Methoden *Execute(A)Sync()* bzw. *StartRepetition()* einfach, mehrfach oder periodisch ausgeführt werden. Die einfache Ausführung erfolgt in der Regel synchron, d. h. nach dem Starten des Dienstes wartet der MCD 3-Server, bis der Diagnoserdienst ausgeführt wurde und liefert das Ergebnis als Rückgabewert. Die mehrfache und die periodische Ausführung müssen asynchron erfolgen, d. h. der Server arbeitet nach dem Start des Dienstes sofort weiter und der Client wird mit Hilfe eines Events nach jeder Ausführung des Diagnoserdienstes über das aktuelle Ergebnis informiert.

Auf die Parameter der Request Botschaft eines Diagnoserdienstes kann über *MCDService::GetRequest()* und *MCDRequest::GetRequestParameters()* zugegriffen werden, so dass die in der ODX-Datenbank enthaltenen Werte verändert werden können. Das Auslesen der Ergebnisse erfolgt über ein *MCDResult*-Objekt wie bereits in Abb. 6.45 dargestellt, wobei statt des nur in der Funktionsgruppe M vorhandenen *MCDCollector*-Objekts das *MCD-DiagService*-Objekt als Ausgangspunkt für die *fetchResults()*-Methode dient.

Alternativ zu Diagnoserdiensten kann auf einzelne Steuergerätegrößen auch direkt zugegriffen werden, sofern diese in der ODX-Beschreibung als Diagnosevariable (DIAG-VAR) definiert sind (vgl. Abschn. 6.6.5).

Jobs werden vom MCD 3-Server im Prinzip wie einfache synchrone Diagnoserdienste aufgerufen, d. h. der Job wird gestartet und der MCD 3-Server wartet, bis der Job vollständig ausgeführt ist. Jobs müssen in der Programmiersprache Java geschrieben werden, wobei aber nur ein kleiner Teil der Java-Bibliotheken (Java.lang, Java.util, Java.math, Java.text) verwendet werden darf und der gesamte Ablauf innerhalb eines Jobs sequenziell erfolgen muss. Die Übergabe der Ein- und Ausgangsparameter erfolgt wie bei den Diagnoserdiensten über *MCDRequest* bzw. *MCDResult*-Strukturen. Hierfür und für das Zusammenspiel mit dem MCD 3-Laufzeitsystem stellt ASAM das Java-Package *asam.d.** zur Verfügung, das jeder Java-Job importieren muss. Komplexe Aufgaben können in mehrere Jobs zerlegt werden, wobei ein *äußerer* Job dann den Aufruf der *inneren* Jobs veranlasst. Falls das

Client-Programm ebenfalls in Java realisiert ist und die Verbindung zwischen Client und Server das Java-RMI-(Remote-Method-Invocation)-Verfahren einsetzt, ist es zu Testzwecken in der Entwicklungsphase möglich, einen Job nicht auf dem Server sondern im Client auszuführen.

Flash-Jobs sind aus Sicht des Laufzeitsystems gewöhnliche (Single-ECU)-Jobs, denen über *MCDDbFlashSession* und weitere *MCDDbFlash...*-Objekte zusätzlich die Informationen aus der ODX-Beschreibung ECU-MEM über den Aufbau des Steuergerätespeichers und des Flash-Programmierdatensatzes zur Verfügung steht.

6.8 MVCI-Schnittstelle für Diagnosetester nach ISO 22900

Ähnlich dem amerikanischen Ansatz der *Pass-Through Programmierung* nach SAE J1534 (siehe Abschn. 5.3.6) wird auch in Europa versucht, den Aufbau von Diagnosetestsystemen zu vereinheitlichen. Unter dem Begriff *Modular Vehicle Communication Interface* (MVCI) sind die ASAM-Diagnosekonzepte als ISO 22900 bzw. ISO 22901 standardisiert. Das Testsystem besteht aus dem Diagnosetestrechner und einem für verschiedene Bus-systeme modular aufgebauten Businterface für die Diagnoseschnittstellen des Fahrzeugs (Abb. 6.49). Das Businterface übernimmt die Umsetzung der protokoll- und busunabhängigen *Diagnostic Protocol Data Unit* (D-PDU-API) Programmierschnittstelle auf das vom jeweiligen Fahrzeug genutzte Diagnoseprotokoll und Bussystem. Die eigentliche Diagnoseanwendung kann theoretisch direkt auf der D-PDU-Schnittstelle aufsetzen, soll aber vorzugsweise die darüber liegende *Diagnostic Server* (D-Server) Schicht verwenden, die gleichzeitig die Schnittstelle zu den ODX-Diagnosedaten kapselt. Die Normen für ODX (ISO 22901) und D-Server (ISO 22900-3) entsprechen im Wesentlichen den ASAM-Spezifikationen MCD 2D (siehe Abschn. 6.6) und MCD 3D (siehe Abschn. 6.7). Bei der D-PDU API (ISO 22900-2) dagegen handelt es sich um einen Neuentwurf, wobei die prinzipielle Aufgabenstellung der ASAM MCD 1 Schnittstelle entspricht (siehe Abschn. 6.1).

Die hardwareseitigen Vorgaben für das MVCI Businterface sind relativ vage. Das Businterface soll eines oder mehrere der derzeit gängigen oder zukünftigen Diagnoseprotokolle und Bussysteme unterstützen und über den bekannten OBD-Diagnosestecker nach ISO 15031-3 mit dem Fahrzeug verbunden werden. Als Verbindung zum Diagnosetester wird Ethernet bzw. USB vorgeschrieben, optional dürfen beliebige andere Schnittstellen, z. B. WLAN, unterstützt werden. Diese Vorgaben können relativ großzügig gehalten werden, da sämtliche Implementierungsunterschiede auf der Hardwareseite durch die D-PDU API, die der Hersteller zum Businterface mitliefern muss, verborgen bleiben. Die Schnittstelle ist so angelegt, dass mehrere MVCI Module und damit mehrere Protokolle und Bussysteme parallel unterstützt werden können.

Die D-PDU API unterstützt das gängige *Request-Response*-Kommunikationsmodell, bei dem der Diagnosetester eine Diagnoseanfrage sendet und auf eine Diagnoseantwort wartet (siehe Kap. 5). Die übergeordnete Anwendung fordert das Versenden einer Diagnosebot-

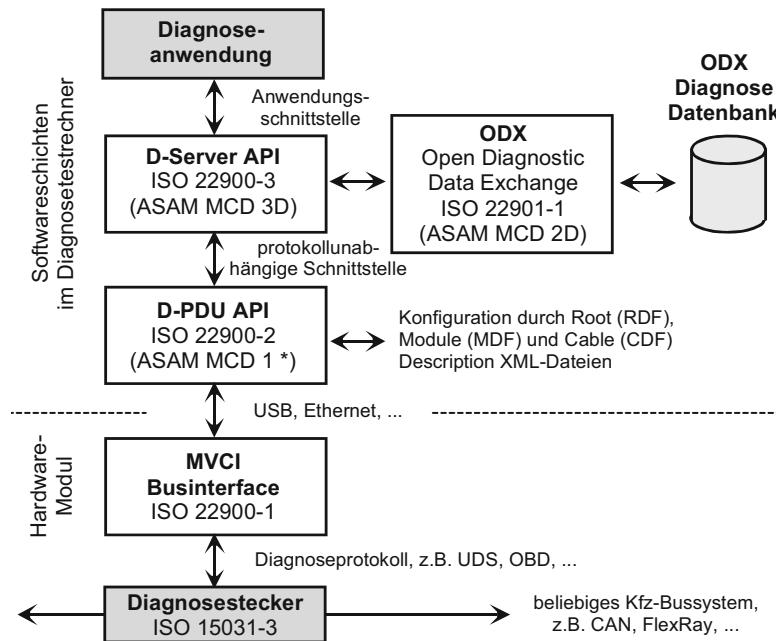


Abb. 6.49 Struktur eines MVCI-Diagnosetestsystems

schaft an (*Communication Primitive*) und wartet dann entweder in einer Abfrageschleife (*Polling*), bis die Diagnoseantwort empfangen wurde, oder lässt sich durch eine Rückruffunktion (*Callback Event Notification*) über die empfangene Antwortbotschaft informieren. Außerdem kann die *D-PDU* Schicht Empfangsbotschaften zwischenspeichern, um Kommunikationsszenarien behandeln zu können, bei denen mehrere Antwortbotschaften erwartet werden, z. B. wenn bei der funktionalen Adressierung bei KWP 2000 oder UDS mehrere Steuergeräte auf eine einzelne Anfrage antworten oder wenn ein Steuergerät spontan Botschaften an den Tester sendet wie beim UDS-Diagnosedienst *Response on Event*. Umgekehrt kann die *D-PDU* Schicht Botschaften, z. B. *Tester Present*, auch selbstständig mehrfach oder periodisch versenden.

Botschaften und sonstige Ereignisse werden beim Empfangen und Versenden im Businterface mit einem Zeitstempel versehen, so dass die darüber liegende Anwendung die zeitlichen Abläufe nachvollziehen kann.

Die *D-PDU API* benutzt das bereits von ODX bekannte Konzept der *logischen Verbindungen* (LOGICAL-LINK, siehe Abschn. 6.6.3) und der Definition von Protokollparametern (COMPARAM, siehe Abschn. 6.6.4). Beim Aufbau einer Verbindung (Tab. 6.26) werden die notwendigen Daten aus den MVCI-Konfigurationsdateien entnommen. Die zentrale Konfigurationsdatei, das *Module Description File MDF*, beschreibt, welche Busysteme und Protokolle das Businterface unterstützt und legt Minimal-, Maximal- und Defaultwerte für die einzelnen Protokollparameter fest. Die Defaultwerte werden dann bei

Tab. 6.26 Vereinfachter Ablauf einer Verbindung mit *D-PDU API* Funktionen

<i>Verbindungsaufbau</i>	
<code>PDUConstruct()</code>	Laden der <i>D-PDU API</i> Bibliothek, Erkennen der angeschlossenen Businterface-Module
<code>PDUREgisterEventCallback()</code>	Registrieren von Rückruffunktionen (optional)
<code>PDUCreateComLogicalLink()</code>	Initialisieren einer logischen Verbindung
<code>PDUGet/SetComParam()</code>	Abfragen und Setzen von Protokollparametern
<code>PDUConnect()</code>	Herstellen der Verbindung vom Tester zum Steuergerät
<i>Senden und Empfangen von Daten mit Hilfe von ComPrimitives</i>	
<code>PDUSTartComPrimitive()</code>	Senden einer Diagnosebotschaft
<code>PDUGetEventItem()</code>	Lesen der Antwortbotschaft
<i>Verbindungsabbau</i>	
<code>PDUDisconnect()</code>	Beenden der Verbindung zum Steuergerät
<code>PDUDestroyComLogicalLink()</code>	Löschen der logischen Verbindung
<code>PDUDEstruct()</code>	Entladen der <i>D-PDU API</i> Bibliothek

Bedarf durch diejenigen Parameter überschrieben, die für die spezielle Fahrzeugkonfiguration gültig sind und in der zugehörigen ODX COMPARAM-SPEC des Fahrzeugs festgelegt wurden.

Das Senden und Empfangen von Botschaften erfolgt mit Hilfe der sogenannten *Communication Primitives*. Dabei ergänzt die D-API beim Senden die von der übergeordneten Anwendung gelieferten „Nutzdaten“ (inkl. *Service Identifier*) der Diagnosebotschaften um die entsprechenden Protokollheader und liefert beim Empfangen nach Entfernen der Protokollinformationen die entsprechenden „Nutzdaten“ zurück. Eine Interpretation der Nutzdaten findet nicht statt, sondern erfolgt in der übergeordneten Anwendung z. B. mit Hilfe der ODX DIAG-SERVICE Definitionen (siehe Abschn. 6.6.5). Spezielle *Communication Primitives* führen protokollspezifische Abläufe wie etwa die KWP 2000 *Fast Initialisation* durch (siehe Abschn. 2.2.3).

Die *D-PDU API* übernimmt also im Normalfall die Abwicklung des Übertragungsprotokolls inklusive der Einhaltung der entsprechenden Zeitbedingungen. Im Sonderfall, dem sogenannten *Raw* oder *Pass-Through Mode*, dagegen wickelt die übergeordnete Anwendung das Diagnoseprotokoll selbst ab und verwendet die *D-PDU API* lediglich zum Versenden und Empfangen der Busbotschaften. Somit kann das Businterface auch für Protokolle genutzt werden, die nicht direkt durch MVCI unterstützt werden, z. B. XCP. Das Protokoll muss dann im Diagnosetestrechner selbst implementiert werden, was bei den üblichen, nicht echtzeitfähigen Betriebssystemen allerdings die Einhaltung der Timing-Bedingungen gegebenenfalls erheblich erschwert.

6.9 OTX-Beschreibung von Testabläufen nach ISO 13209

Diagnoseabläufe bestehen aus aufeinanderfolgenden Diagnosebotschaften, die üblicherweise von einem menschlichen Benutzer über die Bedienoberfläche des Diagnosetesters ausgelöst und deren Ergebnisse visualisiert werden. Nachfolgende Botschaftssequenzen hängen häufig von den Resultaten der vorigen Sequenz sowie weiteren Eingaben des Benutzers ab. ODX erlaubt es, derartige Abläufe über Java-Jobs zu realisieren. Ein Java-Job ist allerdings bereits die in eine spezielle Programmiersprache umgesetzte Implementierung eines derartigen Ablaufs. In der Praxis ist der Diagnoseentwickler allerdings ein Fachmann für Diagnoseprozesse, der solche Aufgaben auf einer abstrakteren Ebene spezifiziert und sich nicht mit programmiertechnischen Details befassen will und kann. Hierfür stellt ODX leider keine geeigneten Hilfsmittel bereit.

Um diese Lücke zu schließen, wurde mit dem *Open Test Sequence eXchange Format* OTX nach ISO 13209 ein Datenformat zur abstrakten Beschreibung von Diagnoseabläufen definiert. Logisch ist OTX im ASAM-Diagnosemodell nach Abb. 6.22 oberhalb der MCD 3 Schicht angeordnet.

Wie bei ODX soll der Anwender nicht direkt mit den XML-Datensätzen von OTX arbeiten, sondern seine Aufgaben mit Hilfe eines geeigneten Eingabewerkzeugs erledigen, welches das XML-Datenformat erstellt und daraus die notwendige Implementierung generiert. In diesem Kapitel werden die Grundstrukturen von OTX dargestellt. Ein typisches OTX-Werkzeug wird dann in Abschn. 9.7 vorgestellt.

6.9.1 Grundkonzepte und Aufbau von OTX

OTX unterscheidet das Grundsystem und die Systemerweiterungen (Abb. 6.50):

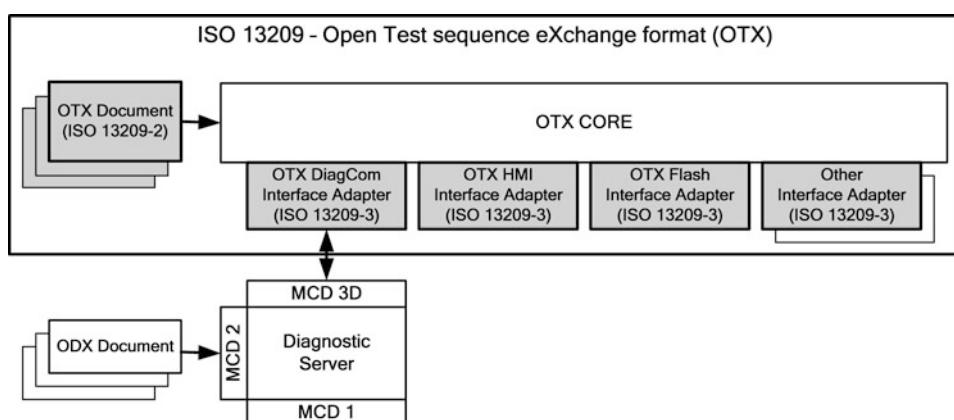


Abb. 6.50 Struktur eines OTX-Diagnosetestsystems

- Das OTX-Core-Datenmodell in ISO 13209-2 beschreibt die Grundelemente einer OTX-Testsequenz. Dabei werden die typischen Elemente einer Programmiersprache wie Datentypen, Deklarationen, Zuweisungen, Operatoren, Kontrollflusstrukturen und Methoden als XML-Strukturen definiert.
- Die in ISO 13209-3 beschriebenen OTX-Extensions definieren die Schnittstellen zur Außenwelt, z. B. den Zugriff auf Diagnosdienste und ODX-Daten (DiagCom), die Flash-Programmierung oder die Anbindung von Benutzerschnittstellen (HMI).

OTX beruht auf vier Grundkonzepten, die das Prozess- und Variantenmanagement bei der Diagnoseentwicklung vereinfachen sollen:

- Specification and Realisation*

Testsequenzen werden häufig in einem dreistufigen Prozess entwickelt. In der *Spezifikationsphase* werden die Testfälle allgemein spezifiziert. Testsequenzen können bereits logisch erfasst, in einzelne Schritte aufgeteilt und beschrieben werden, ohne die exakten Details zur Realisierung zu kennen. Bereits zu diesem Zeitpunkt ist ein Austausch der Testfallbeschreibungen zwischen Fahrzeughersteller, Zulieferern und Prüforganisationen wie TÜV oder DEKRA möglich.

In der *Zwischenphase* wird die Spezifikation schrittweise implementiert. Dabei entstehen *Aktivitäten*, ein Ablaufcode im XML-Format, der Steuerkonstrukte klassischer Programmiersprachen abstrakt nachbildet. Ein Interpreter für OTX-Abläufe kann diese Sequenzen simulieren und mit einem Codegenerator in realen Programmcode für ein Testsystem übersetzt. In dieser Phase ist ein Testen im Mischbetrieb möglich. Bereits implementierte Anteile können real ausgeführt, noch nicht realisierte Teile in der OTX-Laufzeitumgebung (*OTX-Runtime*) simuliert werden.

In der *Realisierungsphase* schließlich sind alle Testsequenzen vollständig implementiert und die Entwicklung ist abgeschlossen.

- Context*

Nicht alle für einen Test- und Prüfablauf relevanten Informationen können aus einem Steuergerät ausgelesen werden. Ein Steuergerät kann nicht erkennen, ob es sich im Entwicklungs-, Produktions- oder Serviceumfeld befindet. Um die Testsequenz im richtigen Kontext ausführen zu können, müssen solche Informationen durch das OTX-Laufzeitsystem durch sogenannte Kontextvariablen zur Verfügung gestellt werden. Kontextvariablen werden vom Diagnoseautor definiert und können Fahrzeugdaten, Benutzereingaben oder Informationen über den Diagnosetester enthalten.

- Validity*

OTX kann Testabläufe oder Teile davon in Abhängigkeit von bestimmten Bedingungen variabel ausführbar gestalten. Das *Validity*-Konzept ist vergleichbar mit den aus Programmiersprachen bekannten Compiler-Direktiven wie `#ifdef`, mit denen Programmteile ausgeblendet werden können. In Verbindung mit den Kontextvariablen können so Testschritte festgelegt werden, die nur in der Produktion oder nur in der Werkstatt ausgeführt werden. Das *Validity*-Konzept für ganze Testsequenzen ist übersichtlicher als

wenn solche Entscheidungen über viele einzelne Programmverzweigungen implementiert werden müssen.

- *Signature*

Eine Signatur ist die Schnittstelle einer OTX-Testsequenz. Sie entspricht dem Funktionsprototyp einer Prozedur in einer gewöhnlichen Programmiersprache. Über den Signaturnamen und die Übergabe der entsprechenden Parameter kann eine Testsequenz aus einer anderen Testsequenz heraus aufgerufen werden, ohne dass Implementierungsdetails bekannt sind. In Verbindung mit *Validity*-Informationen lassen sich so generische Testsequenzen erzeugen, die sich zur Laufzeit an die entsprechende Umgebung anpassen.

6.9.2 OTX-Core-Datenmodell

Ein OTX-Datensatz wird in einem *OTX-Dokument* angelegt, das mit dem XML-Wurzelement `<otx>` beginnt und in seinem Aufbau der Struktur eines Java-Programms ähnelt (Abb. 6.51). Am Anfang stehen Verwaltungsinformationen wie `Name` (`name`), Identifikationskennung (`id`), Versions- (`version`) und Zeitangaben (`timestamp`), eine Inhaltsbeschreibung (`specification`) und Historien- und Metainformationen (`adminData`). OTX unterstützt die Strukturierung, indem inhaltlich zusammengehörende OTX-Dokumente zu Paketen (`package`) gruppiert werden.

Der Import von Elementen aus anderen OTX-Dokumenten erfolgt über eine `imports`-Sektion. Die Kombination aus `package` und `name` stellt die eindeutige Bezeichnung der importierten OTX-Dokumente sicher und ermöglicht den Zugriff via eines *OTX-Links*. Die Definition eines kurzen `prefix`-Namens erleichtert den späteren Zugriff auf die importierten Elemente. Über diesen Mechanismus werden auch die Erweiterungen aus ISO 13209-3 eingebunden.

Innerhalb der XML-Struktur werden globale oder lokale Deklarationen von Variablen, Konstanten und Parametern in entsprechenden `declarations`-Sektionen angelegt. Auf der obersten Ebene werden globale Elemente wie Context-Variablen deklariert, die vom Laufzeitsystem gesetzt und innerhalb des OTX-Dokuments nur gelesen werden. Eine typische Context-Variable ist z. B. die Fahrzeugidentifikationsnummer VIN, die vom Laufzeitsystem über einen Diagnosedienst vom Fahrzeug abgefragt wird. Für die Realisierung lauffähiger Programme muss den Konstanten, Variablen oder Ausdrücken im `realisation`-Segment ein Datentyp wie `BOOLEAN`, `INTEGER`, `FLOAT`, `STRING` usw. und gegebenenfalls ein Initialisierungswert zugeordnet werden. Tabelle 6.27 zeigt ein Beispiel für eine Variablen-Deklaration. Die Sichtbarkeit von Variablen und Funktionen bestimmt OTX mit den Ebenen `PRIVATE`, `PACKAGE` und `PUBLIC`.

Nach den Variablen Deklarationen folgen die bereits oben beschriebenen `validities` für die bedingte Ausführung von Testschritten. Unter `signatures` werden Funktionsprototypen und anschließend unter `procedures` die eigentlichen Funktionen definiert (Tab. 6.28).

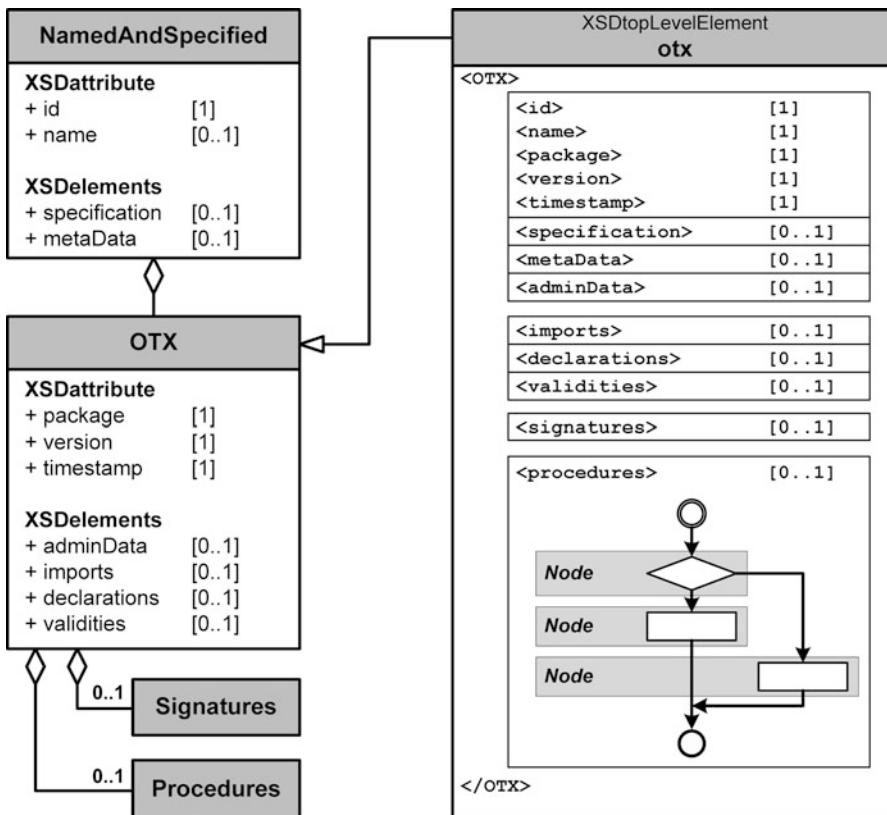


Abb. 6.51 Aufbau eines OTX-Dokumentes

6.9.3 OTX-Core-Programmelemente

Die von Programmiersprachen bekannten Elemente wie Zuweisungen, Blöcke und Kontrollflussstrukturen werden im OTX-Datenmodell als *Nodes* bezeichnet (Abb. 6.52). Eine Testprozedur besteht aus vielen *Nodes*, die nacheinander abgearbeitet werden. Jeder *Node* hat eine *id* zur Identifikation, optional sind ein *Name* (*name*) und eine Beschreibung (*specification*). Durch einen Schalter (*disabled*) kann ein *Node* deaktiviert werden. Dies entspricht dem *Auskommentieren* in einer klassischen Programmiersprache. OTX unterscheidet nach einfachen *AtomicNodes* und Elementen, die weitere *Nodes* enthalten (*CompoundNodes*).

Sogenannte *Action-Nodes* implementieren Zuweisungen und Funktionsaufrufe. Funktionen (OTX-Prozeduren) werden durch *ProcedureCall-Node*-Elemente mit den entsprechenden Argumenten aufgerufen. Tabelle 6.29 zeigt beispielhaft eine Zuweisung (*assignment*). Als Ergebnis (*result*) wird hier einer Variablen der Rückgabewert eines Ausdrucks (*term*) zugewiesen. Ausdrücke können mathematische Formeln, Lite-

Tab. 6.27 Vergleich einer Deklaration in einer Programmiersprache und in OTX

Deklaration in der Programmiersprache Java:
<pre>int i = 42; // A simple Integer</pre>
Deklaration in OTX:
<pre><declarations> <variable> <name="i" id="001"> <specification>A simple Integer</specification> <realisation> <dataType xsi:type="Integer"> <init value="42"/></dataType> </realisation> </variable> </declarations></pre>

rale, logische oder relationale Anweisungen, Konvertierungen, Dereferenzierungen und anderes enthalten.

Kontrollflusselemente in *Compound-Nodes* steuern den sequentiellen Ablauf von Programmen (Abb. 6.53). Das OTX-Core-Datenmodell bietet die Kontrollflusselemente Gruppierung (group) mit und ohne wechselseitigem Ausschluss (mutex), Schleifen (loop), Verzweigungen (branch), Parallelisierung (parallel) und Ausnahmebehandlung (handler) an. *End-Nodes* beenden einen Programmzweig, z. B. eine Schleife, vorzeitig und kehren auf eine höhere Ebene des Programmflusses zurück. Interessant ist dabei vor allem die Möglichkeit, mit *Parallel-Node* parallele Verarbeitungsstränge (lanes) zu erzeugen. Ein solcher Node wird erst verlassen, wenn alle parallelen Verarbeitungsstränge komplett durchlaufen sind. Ein einzelner Strang beendet sich normalerweise mit `return`, die gesamte Parallelstruktur kann mit `terminateLanes` vorzeitig abgebrochen werden. Kritische Bereiche (*critical section*) und wechselseitiger Ausschluss (*mutual exclusion*) können definiert werden, Ausnahmebehandlungen sind mit `throw` möglich.

Neben den vom OTX-Autor definierten Ausnahmebehandlungen (`UserException`) bietet OTX mit Bereichsüberwachungen bei Arrays und Listen (`OutOfBoundsException`), Typprüfungen (`TypeMismatchException`), der Erkennung von Fehlern in Formeln (`ArithmeticException`) oder der Überprüfungen auf nicht initialisierte Variable (`InvalidReferenceException`) schon standardmäßig eine große Zahl von Überwachungen, um Laufzeitfehler abzufangen.

6.9.4 OTX-Erweiterungen

Die OTX-Erweiterungen definieren die Schnittstellen des OTX-Kernsystems zur Außenwelt. Die wichtigsten Schnittstellen sind (Abb. 6.50):

Tab. 6.28 Attribute und Elemente von Prozeduren und Signaturen (Auswahl)

Attribut/Element	Prozedur, Si- gnatur	Multi- plizität	Beschreibung
name	Prozedur Signatur	1	Name der Prozedur oder Signatur innerhalb eines OTX-Dokumentes. main signalisiert die Einstiegsprozedur einer Testsequenz.
visibility	Prozedur Signatur	1	Definiert die Sichtbarkeit der Prozedur oder Signatur (PRIVATE, PACKAGE, PUBLIC).
implements	Prozedur	0..1	Verweis (OTX-Link) zwischen Signatur und Prozedur.
validFor	Prozedur	0..1	Definiert die Bedingungen, unter denen die Prozedur ausführbar ist. Die Ausführbarkeit kann in Abhängigkeit des Wertes einer globalen Konstanten (constant), einer context-Variablen oder eines validity-Wertes eingeschränkt werden.
specification	Prozedur Signatur	0..1	Für den Menschen lesbare Beschreibung der Prozedur oder Signatur.
realisation	Prozedur Signatur	0..1	Implementierungsblock
parameters	Prozedur Signatur	0..1	Deklarationsblock für Parameter von Prozeduren und Signaturen mit den Ein- und Ausgangsparametern inParam, inOutParam und outParam
declarations	Prozedur	0..1	Lokale Konstanten und Variablen
flow	Prozedur	1	Legt die einzelnen Programmelemente (Nodes) einer Prozedur fest.
...

Tab. 6.29 Beispiel einer Zuweisung in OTX

```

<action id="A-001">
  <specification>Example for an assignment action node</specification>
  <realisation xsi:type="Assignment">
    <result xsi:type="IntegerVariable" name="i" />
    <term  xsi:type="IntegerLiteral" value="1711174" />
  </realisation>
</action>

```

- *DiagDataBrowsing*, *DiagCom* und *DiagComRaw* erlauben den Zugriff auf ODX-Daten und die Abwicklung von Diagnosediensten (Abb. 6.54). Die Erweiterungen kapseln den Fahrzeugzugang, die physikalische oder funktionale Adressierung der Geräte, den Umgang mit Request- und Response-Parametern sowie die Variantenidentifikation. Grundsätzlich kann jede Art von Diagnose-Laufzeitsystem unterstützt werden, empfohlen jedoch wird ein standardisierter ODX/MVCI-Server.

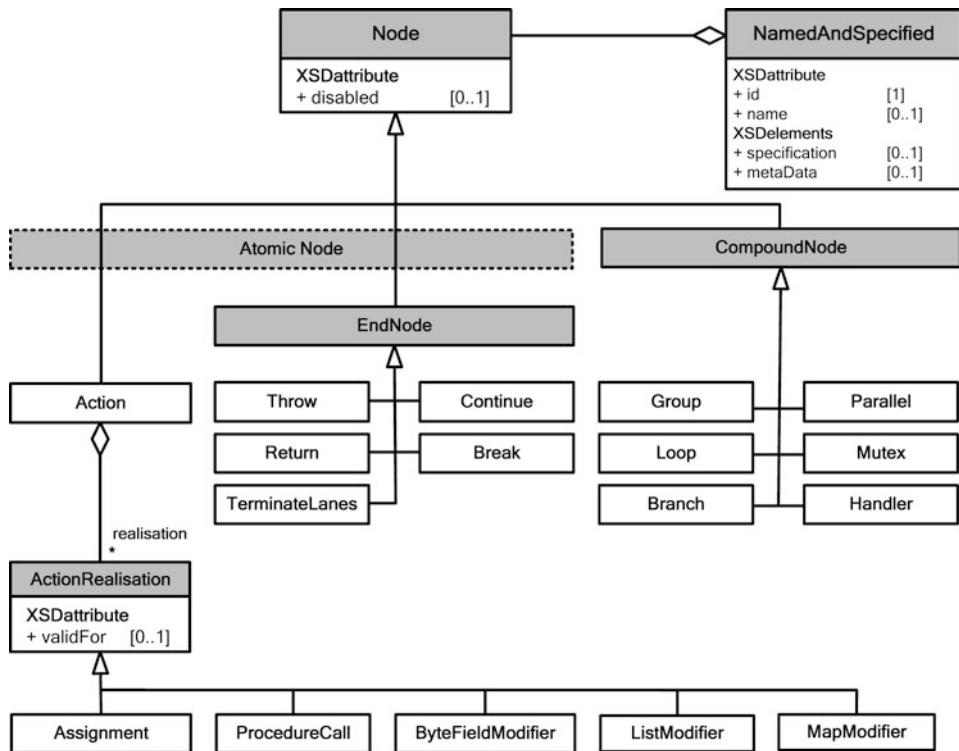


Abb. 6.52 Aufbau der Programmelemente (Nodes)

- *Human Machine Interface HMI* stellt Elemente zum Aufbau einer Bedienoberfläche für den Diagnosetester bereit. Die Erweiterung enthält Standarddialoge sowie frei parametrierbare Bildschirmausgaben und Tastatureingaben. OTX unterstützt im Grundumfang zwei Ausgabemasken. Der *BasicScreen* bietet modale Dialoge, mit denen der

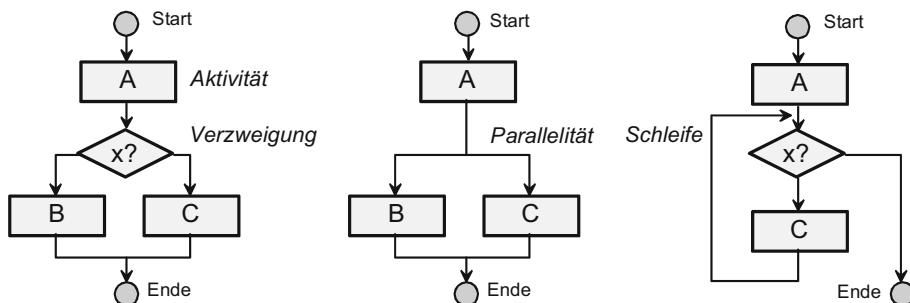


Abb. 6.53 Typische Kontrollflussstrukturen

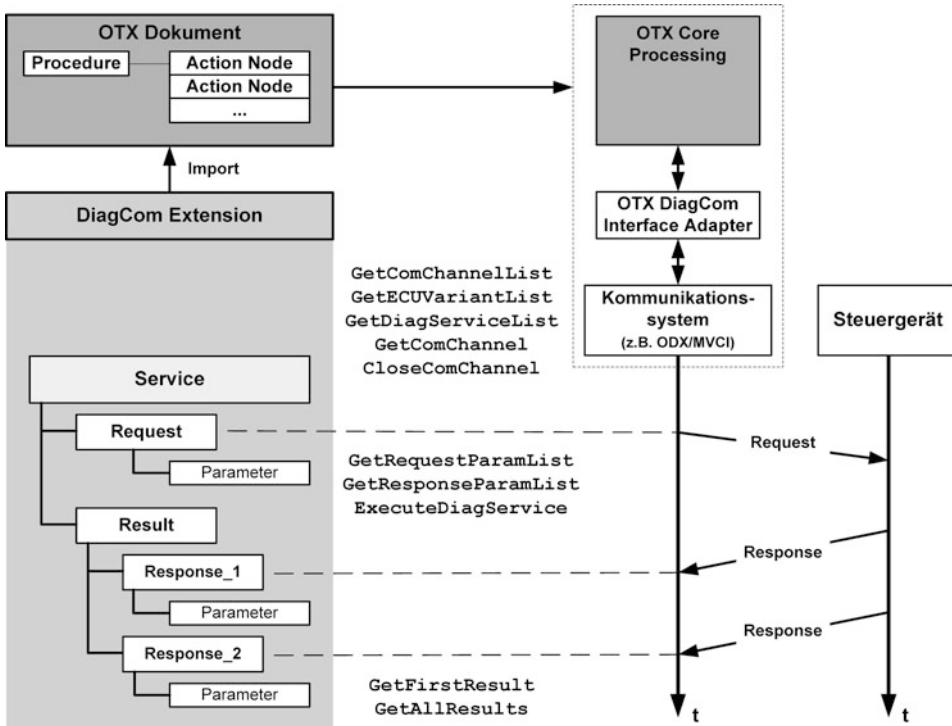


Abb. 6.54 Diagnose Request und Response in OTX

Anwender zu einfachen Eingaben aufgefordert werden kann bzw. Meldungen vom System erhält. Mit dem *CustomScreen* können beliebig komplexe Oberflächen gebaut werden. Da sich die verschiedenen Diagnosetester vom einfachen Handheld-Gerät über Geräte mit Touchscreens bis zum vollwertigen Computer mit hochauflösendem Farbbildschirm aber sehr stark unterscheiden, sieht OTX keine direkte Beschreibung des Bildschirmlayouts vor, sondern benötigt hierfür Tool-spezifische Erweiterungen wie beispielsweise das im Abschn. 9.7 beschriebene *Open Diagnostic Framework ODF*, dessen Daten standardkonform als OTX-Metadaten integriert werden.

- *Flash* enthält die Funktionen für die Flash-Programmierung von Steuergeräten einschließlich des Managements der Diagnosesitzungen und der Authentifizierung und Verifikation (s. h. Abschn. 9.4). Das OTX-Datenmodell (Abb. 6.55) ist an den ODX ECU-MEM Container (Abb. 6.40) angelehnt.
- *Measure* kapselt die Durchführung von Mess- und Steueraufgaben, die nicht ausschließlich über die Diagnosekommunikation sondern unter Einbindung externer Messgeräte durchgeführt werden.
- *EventHandling* ergänzt die sequenzielle Ablaufsteuerung des OTX-Kerns um Elemente zur Ereignisbehandlung, z. B. die Reaktion auf Benutzeraktionen wie Mausklicks, Tas-

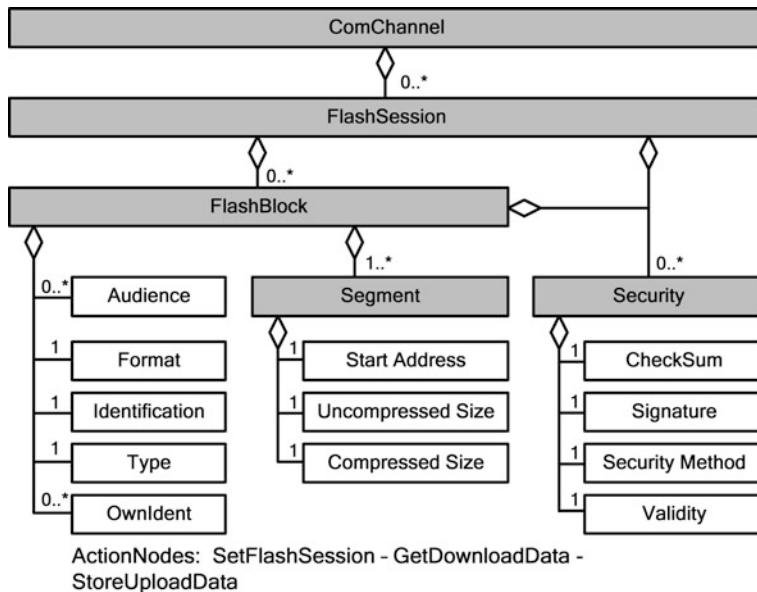


Abb. 6.55 OTX Flash Erweiterung

tatureingaben oder den Ablauf von Timern, aber auch die Veränderung von Variablen oder das Überschreiten eines Grenzwertes. Die Ereignisbehandlung erfolgt dabei synchron, d. h. die Ereignisquelle (*EventSource*) wird durch OTX definiert, danach wird auf das Auftreten des Ereignisses gewartet (*WaitForEvent*) und sequenziell weitergearbeitet, sobald das Ereignis (*Event*) auftritt.

- Mit *Internationalization*, *StringUtil*, *Math*, *DateTime* oder *Quantities*, die bei der Anpassung an unterschiedliche Landessprachen oder beim allgemeinen Umgang mit Texten, Umrechnungsformeln und Einheiten unterstützen, wird die praktische Implementierung der Diagnosefunktionen erleichtert. Darüber hinaus kann der Anwender oder Toolhersteller OTX um eigene Bibliotheken erweitern.

6.10 Normen und Standards zu Kapitel 6

ASAM 1	<p>ASAP Standard CAN Calibration Protocol CCP Version 2.1, 1999, www.asam.net</p> <p>ASAP2 Meta Language für CCP – AML Version 2.6, 2002, www.asam.net</p> <p>ASAP Interface Specification Interface 1b Version 1.2, 1998, www.asam.net</p> <p>XCP The Universal Measurement and Calibration Protocol Family – Part 1: Overview Version 1.0, 2003, www.asam.net</p> <p>XCP The Universal Measurement and Calibration Protocol Family – Part 2: Protocol Layer Specification Version 1.0, 2003, www.asam.net</p> <p>XCP The Universal Measurement and Calibration Protocol Family – Part 3: XCP on CAN Transport Layer Specification Version 1.0, 2003, www.asam.net</p> <p>XCP The Universal Measurement and Calibration Protocol Family – Part 3: XCP on FlexRay Transport Layer Specification Version 1.0, 2006, www.asam.net</p> <p>XCP The Universal Measurement and Calibration Protocol Family – Part 3: XCP on USB Transport Layer Specification Version 1.0, 2004, www.asam.net</p> <p>XCP The Universal Measurement and Calibration Protocol Family – Part 3: XCP on Ethernet Transport Layer Specification Version 1.0, 2003, www.asam.net</p> <p>XCP The Universal Measurement and Calibration Protocol Family – Part 3: XCP on SxL Transport Layer Specification Version 1.0, 2003, www.asam.net</p> <p>XCP The Universal Measurement and Calibration Protocol Family – Part 4: Interface Specification Version 1.0, 2004, www.asam.net</p> <p>XCP The Universal Measurement and Calibration Protocol Family – Part 5: Example Communication Sequences Version 1.0, 2003, www.asam.net</p> <p>XCP Version 1.1 – What's new, 2008, www.asam.net. Vollständige Spezifikationen der Version 1.1 derzeit nur für ASAM-Mitglieder zugänglich</p>
ASAM 2	<p>MCD 2 FIBEX Field Bus Exchange Format Version 2.0, 2006, www.asam.net</p> <p>MCD 2 NET FIBEX Field Bus Exchange Format Version 3.1.1, 2010</p> <p>MCD 2 NET FIBEX Field Bus Exchange Format Version 4.1, 2013</p> <p>MOST Fibex4MOST Cookbook Version 1.0, 2009, www.mostcooperation.com</p> <p>ASAM MCD 2MC/ASAP2 Interface Specification Interface 2 Version 1.6, 2009, www.asam.net</p> <p>ASAM MCD 2MC Calibration Data Format User's Guide Version 2.0, 2006, www.asam.net</p> <p>ASAM MCD 2MC Calibration Data Format Reference Guide Version 2.0, 2006, www.asam.net</p> <p>ASAM AE Meta Data Exchange Format for Software Sharing MDX Version 1.0, 2006, www.asam.net</p> <p>ASAM MCD 2D (ODX) Data Model Specification Version 2.0.1, 2005, www.asam.net</p> <p>ASAM MCD 2D (ODX) Data Model Specification Version 2.1.0, 2006</p> <p>ASAM MCD 2D (ODX) Data Model Specification Version 2.2, 2008, www.asam.net</p> <p>ASAM MCD 2D (ODX) Authoring Guidelines Version 1.0, 2011, www.asam.net ODX ist mittlerweile als ISO 22901 standardisiert (siehe unten)</p>

ASAM 3	ASAM MCD 3 Application programmer's interface specification Version 2.2.0, 2008, www.asam.net ASAM MCD 3 Programmer's reference guide Version 2.2.0, Teil 1: MCD, MD und D, Teil 2: MCD, MC, M und C, 2008, www.asam.net ASAM MCD 3 Application programming interface specification Version 3.0.0, Teil D und Teil MC, 2013, www.asam.net MCD3 ist mittlerweile als ISO 22900 standardisiert (siehe unten)
ASAM MDF	ASAM MDF Measurement Data Format – Programmer's guide Version 4.1.0, 2012, www.asam.net
ISO	ISO 22900-1 Road vehicles - Modular vehicle communication interface (MVCI) Part 1: Hardware design requirements. 2008, www.iso.org ISO 22900-2 Road vehicles - Modular vehicle communication interface (MVCI) Part 2: Diagnostic protocol data unit application programmer interface (D-PDU API). 2009, www.iso.org ISO 22900-3 Road vehicles - Modular vehicle communication interface (MVCI) Part 3: Diagnostic server application programmer interface (D-Server API). 2012, www.iso.org ISO 22901-1 Road vehicles - Open diagnostic data exchange (ODX) – Part 1: Data model specification. 2008, www.iso.org ISO 22901-2 Road vehicles - Open diagnostic data exchange (ODX) – Part 2: Emissions-related diagnostic data in ODX format. 2011, www.iso.org ISO 22901-3 Road vehicles - Open diagnostic data exchange (ODX) – Part 3: Fault symptom exchange description (FSD). Noch nicht veröffentlicht ISO 13209-1 Road vehicles – Open test sequence exchange format (OTX) – Part 1: General information and use cases. 2011, www.iso.org ISO 13209-2 Road vehicles – Open test sequence exchange format (OTX) – Part 2: Core data model specification and requirements. 2012, www.iso.org ISO 13209-3 Road vehicles – Open test sequence exchange format (OTX) – Part 3: Standard extensions and requirements. 2012, www.iso.org

Literatur

- [1] C. Marscholik, P. Subke: Datenkommunikation im Automobil. VDE-Verlag, 2. Auflage, 2011

7.1 Einführung

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

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

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

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

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

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

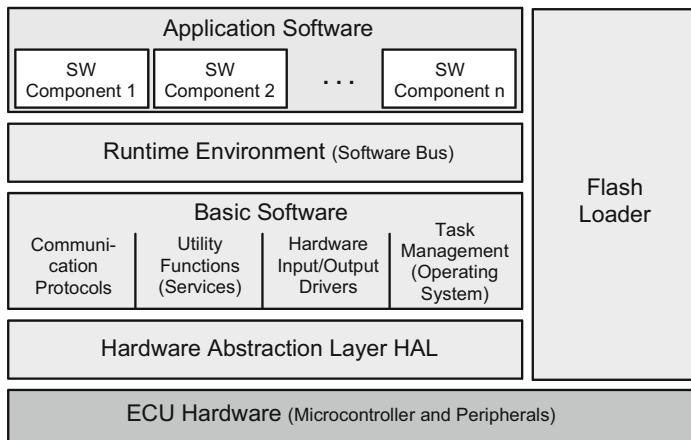


Abb. 7.1 Software-Grundarchitektur zukünftiger Steuergeräte nach AUTOSAR/HIS

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

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

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

7.2 OSEK/VDX

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

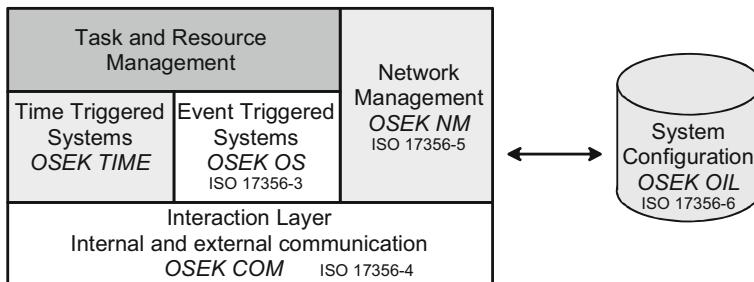


Abb. 7.2 Grundkomponenten des OSEK/VDX-Systems

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

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

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

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

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

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

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

7.2.1 Ereignisgesteuerter Betriebssystemkern OSEK/VDX OS

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

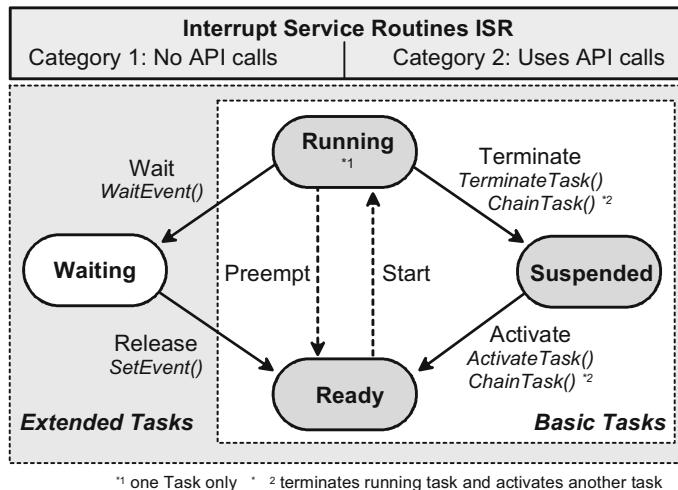


Abb. 7.3 Zustandsmodell für OSEK/VDX Tasks und zugehörige API-Funktionen

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

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

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

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

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

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

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

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

```

DeclareTask(myTaskName) ;

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

ISR(myIsrName)
{   . . .
}

```

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

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

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

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

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

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

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

Tab. 7.3 Beispiel einer OIL-Konfigurationsdatei

```

OIL_VERSION = "2.0";

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

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

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

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

    RESOURCE myResource; // *** Define a resource ****
    EVENT triggerEvent{ // *** Defines an event ****
        MASK = AUTO;
    };

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

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

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

```

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

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

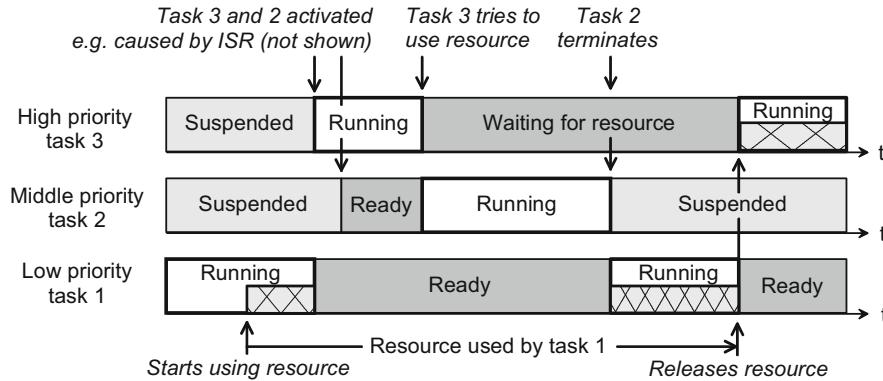


Abb. 7.4 Prioritätsinversion bei Systemen ohne dynamische Prioritätserhebung

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

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

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

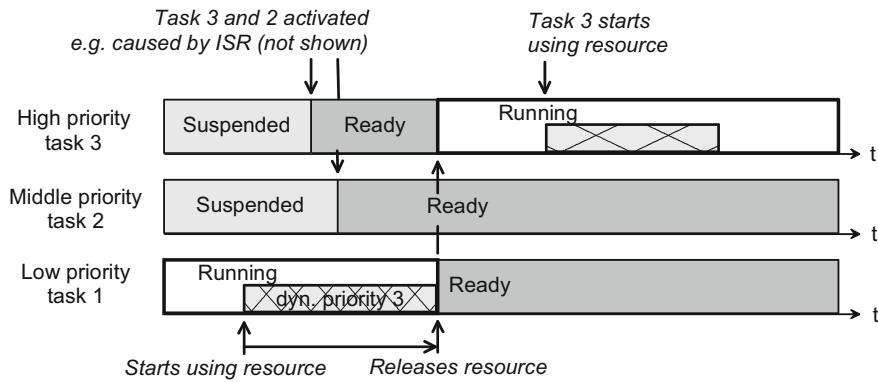


Abb. 7.5 Dynamische Prioritätserhöhung bei der Verwendung von Ressourcen bei OSEK (Annahme: Priorität der gemeinsamen Ressource sei 3)

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

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

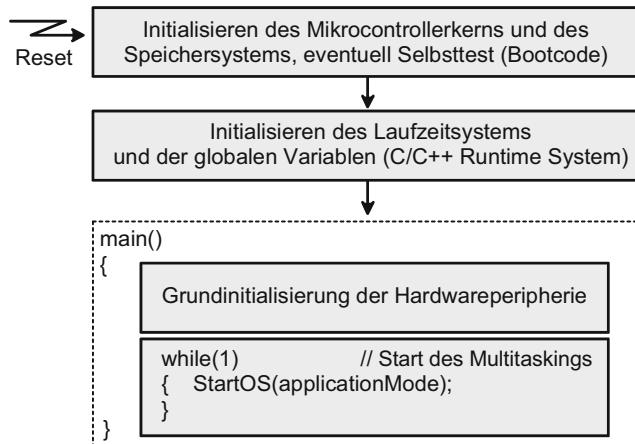


Abb. 7.6 Hochlauf des Systems

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

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

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

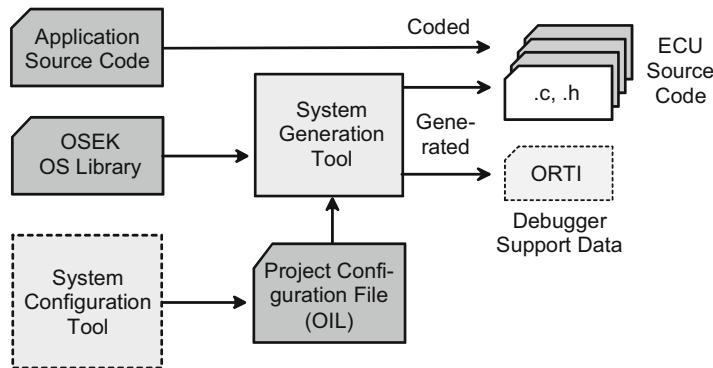


Abb. 7.7 OSEK/VDX Entwicklungsmodell

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

7.2.2 Kommunikation in OSEK/VDX COM

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

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

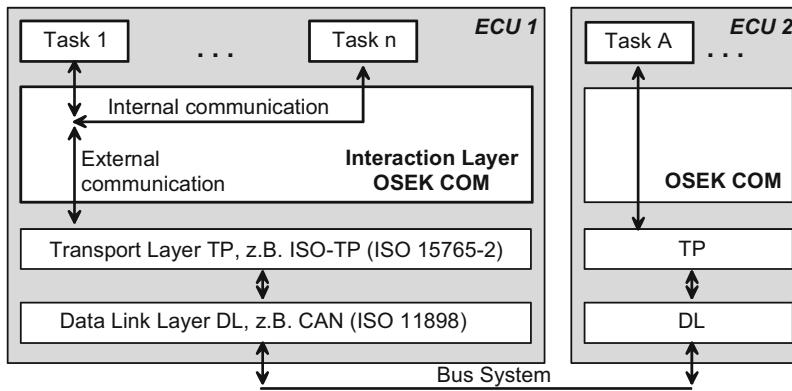


Abb. 7.8 Struktur von OSEK COM

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

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

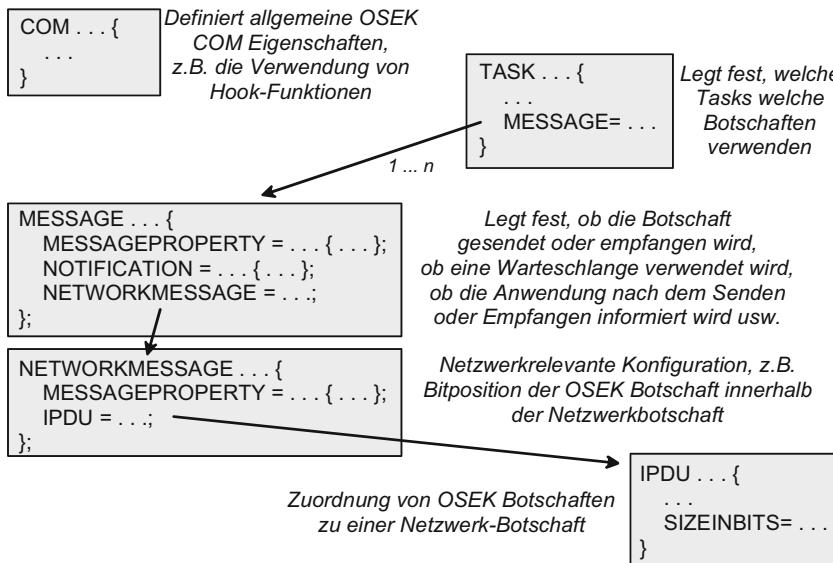


Abb. 7.9 Struktur der OIL-Konfigurationsdatei für OSEK-COM

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

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

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

Tab. 7.4 OSEK COM Ausbaustufen (COM Conformance Classes)

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

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

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

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

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

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

7.2.3 Netzmanagement mit OSEK/VDX NM

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

Man unterscheidet

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

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

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

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

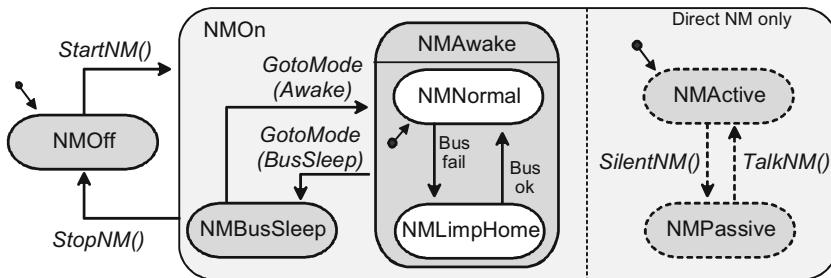


Abb. 7.10 Vereinfachtes Zustandsmodell von OSEK NM ohne transiente Zwischenzustände (gestrichelte Zustände nur beim direkten Netzmanagement)

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

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

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

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

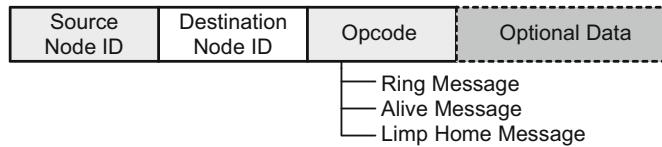


Abb. 7.11 Format der Botschaften für das direkte Netzmanagement

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

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

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

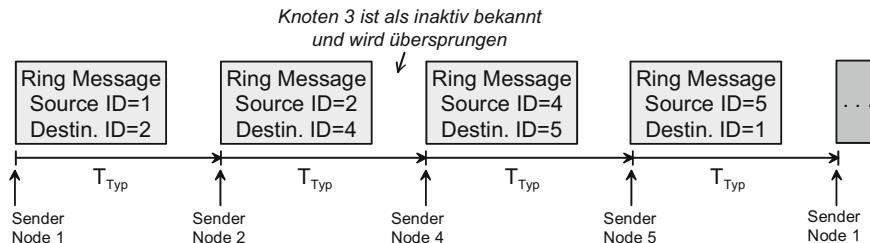


Abb. 7.12 Ring-Botschaften bei *stabilem Betrieb* eines Netzes mit 5 Knoten, wobei der Knoten 3 gerade nicht aktiv ist

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

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

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

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

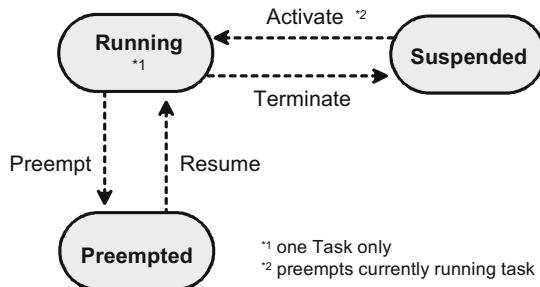
Mit OSEK *Time* wurde eine zeitgesteuerte Betriebssystem-Erweiterung des ereignisgesteuerten OSEK OS veröffentlicht und durch OSEK *FTCOM* (*Fault Tolerant COM*) ergänzt, die zu OSEK *Time* passende Variante von OSEK *COM*. Die Version 1.0 ist allerdings bis heute unverändert die einzige öffentlich zugängliche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

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

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

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

Abb. 7.13 Zustandsmodell der OSEK Time Tasks



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

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

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

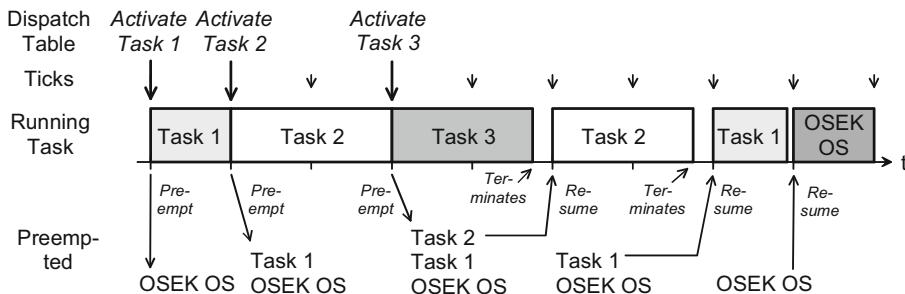


Abb. 7.14 Beispiel eines Task-Ablaufs bei OSEK Time

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

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

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

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

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

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

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

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

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

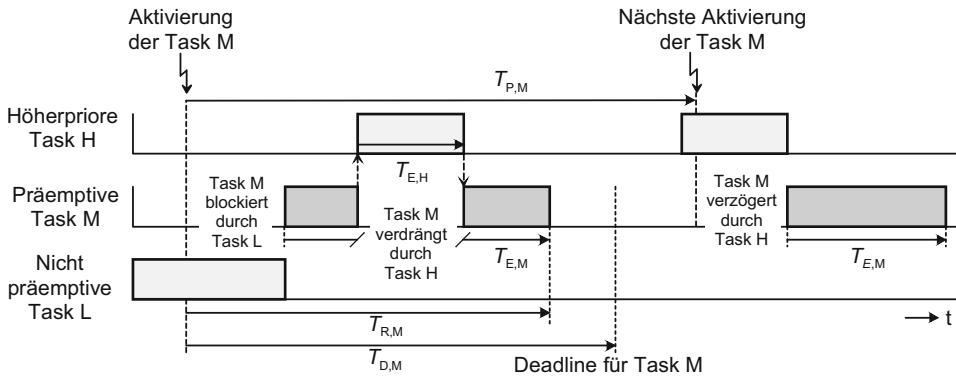


Abb. 7.15 Szenario zur Berechnung der Task-Antwortzeit

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

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

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

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

$$T_{R,M,min} = T_{E,M} \quad (7.2)$$

Die maximale Antwortzeit kann wie folgt abgeschätzt werden:

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

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

Tab. 7.5 Einfaches OSEK OS System

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

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

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

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

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

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

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

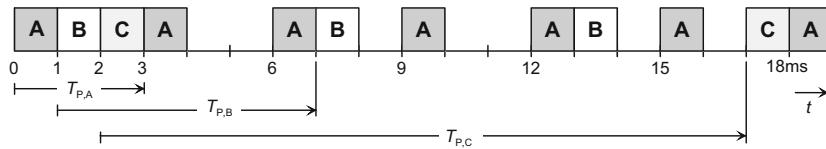


Abb. 7.16 Optimierter Task-Ablauf

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

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

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

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

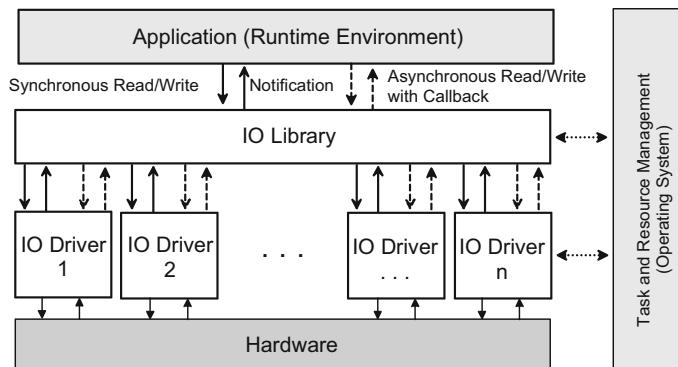
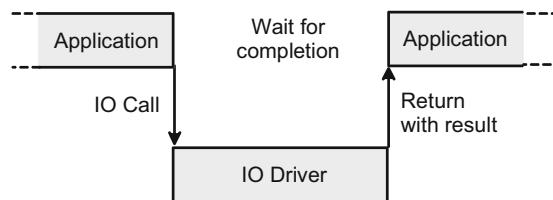


Abb. 7.17 Architektur für Hardwarezugriffe nach HIS

Abb. 7.18 Synchrone Ein-Ausgabe-Schnittstelle



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

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

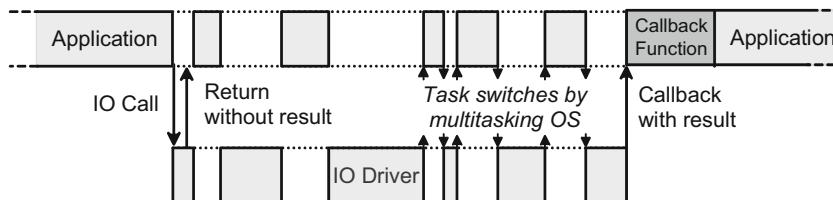


Abb. 7.19 Asynchrone Ein-Ausgabe-Schnittstelle

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

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

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

7.5 HIS Flash-Lader

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

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

7.6 Normen und Standards zu Kapitel 7

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

Literatur

- [1] J. Schäuffele, T. Zurawka: *Automotive Software Engineering*. Springer-Vieweg Verlag, 5. Auflage, 2013
- [2] M. Homann: *OSEK. Betriebssystem-Standard für Automotive und Embedded Systems*. Mitp-Verlag, 1. Auflage, 2005
- [3] J. Lemieux: *Programming in the OSEK/VDX environment*. CMP Books-Verlag, 2001
- [4] N. Audsley, A. Burns, M. Richardson, K. Tindell, A. Wellings: *Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling*. Software Engineering Journal, Heft 5, 1993, S. 284–292
- [5] W. Lei, W. Zhaojun, Z. Mingde: *Worst-Case Response Time Analysis for OSEK/VDX Compliant Real-time Distributed Control Systems*. Proceedings der IEEE International Computer Software and Applications Conference, 2004, S. 148–153
- [6] J.C. Palencia, M.G. Harbour: *Schedulability Analysis for Tasks with Static and Dynamic Offsets*. Proceedings des IEEE Real-Time Systems Symposiums, 1998, S. 26–37
- [7] R. Racu, R. Ernst, K. Richter, M. Jersak; *A Virtual Platform for Architectural Integration and Optimization in Automotive Networks*. SAE Transactions Vol 116: Journal of Passenger Car Electronic and Electrical Systems, 2007, S. 372–380

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-

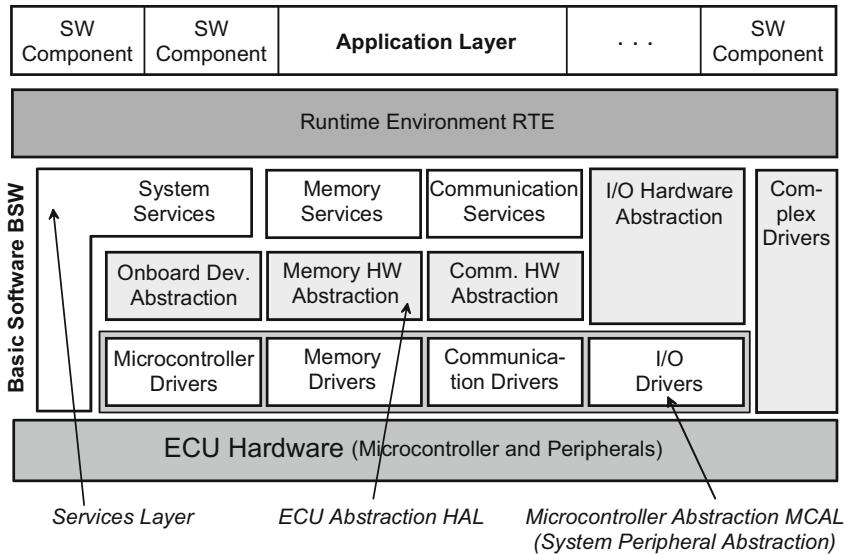


Abb. 8.1 Schichtenmodell der AUTOSAR-Basissoftware

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

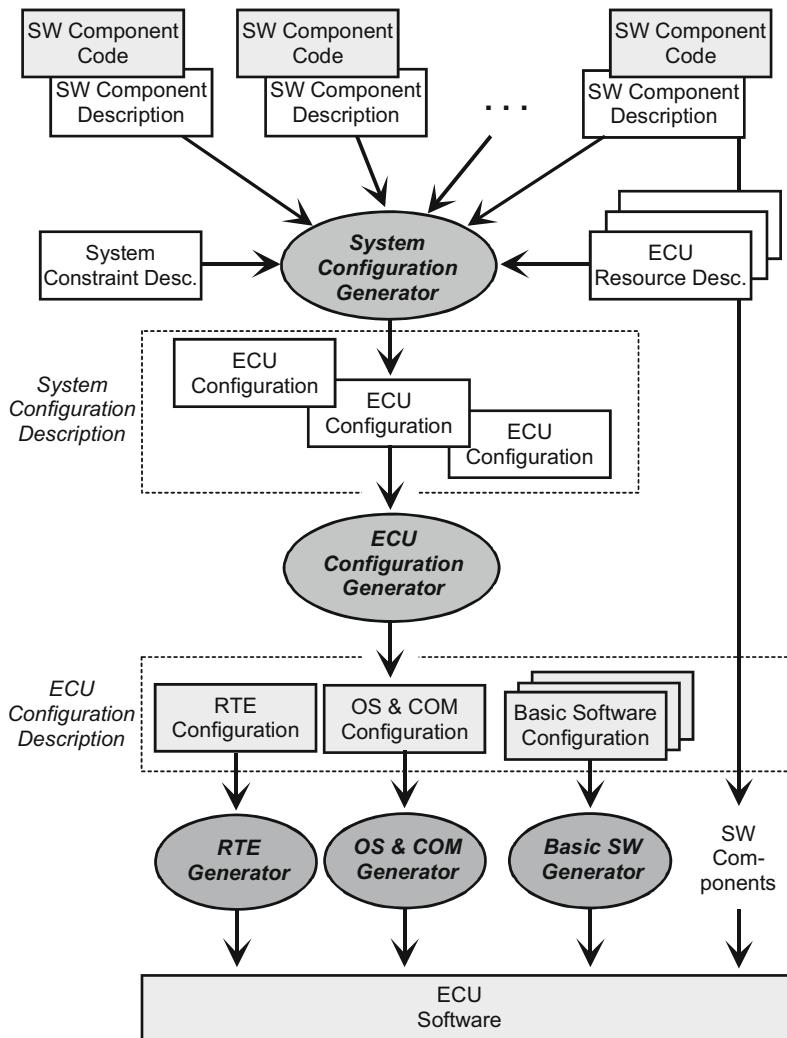


Abb. 8.2 AUTOSAR-Werkzeugkette

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 Gerätelei-

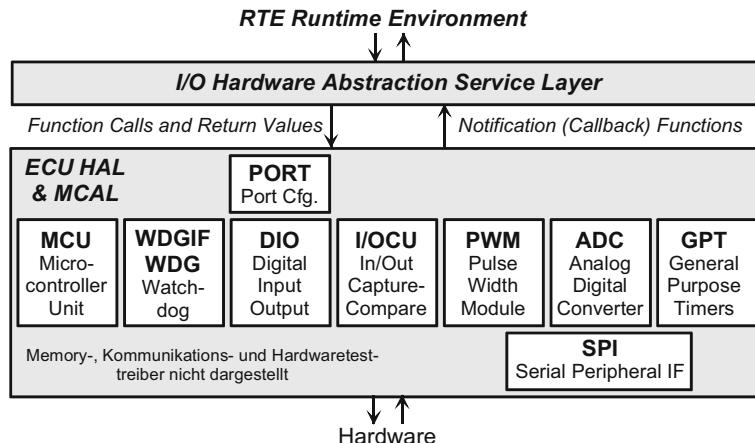


Abb. 8.3 Abstraktionsschichten für die Hardwareperipherie

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 Diagnosdiensten 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 (<i>Clock, PLL</i>), 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 <i>Startup Code</i> 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 (<i>Channel</i>), mehrerer Ein/Ausgänge (<i>Group</i>) oder eines ganzen 8 bzw. 16bit Microcontroller- <i>Ports</i> (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 (<i>Notification Callback</i>) 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 (<i>Notification Callback</i>) bei steigender und/oder fallender Signalflanke an einem Eingang bzw. nach Vorliegen von Messwerten.
Zeitgeber (General Purpose Timer GPT)	Starten und Stoppen von einmaligen (<i>One Shot</i>) oder periodischen (<i>Continuous</i>) Zeitgebern. Abfrage der abgelaufenen Zeit bzw. der Zeit bis zum Ablauf. Rückruffunktion (<i>Notification Callback</i>) beim Ablauf eines Zeitgebers.
Analoge Eingänge (ADC Analog to Digital Converter Inputs)	Messen eines einzelnen (<i>Channel</i>) analogen Eingangssignals oder einer Gruppe von Eingangssignalen (<i>Group</i>). Die Messung kann als Einzelmessung (<i>One Shot</i>) oder kontinuierlich (<i>Continuous</i>) durchgeführt werden. Das Starten der Messung kann durch einen Software-Funktionsaufruf (<i>On Demand</i>), durch das Ereignis (<i>Trigger</i>) eines Timer-Kanals oder eines Hardware-Eingangssignals erfolgen. Rückruffunktion (<i>Notification Callback</i>), wenn Messwerte vorliegen bzw. der Messwertpuffer gefüllt ist.
Watchdog (WDG, WDGIF)	Konfigurieren und Triggern des Watchdogs (WDG). Das <i>Watchdog Interface</i> (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 <i>Watchdog Manager</i> .

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 (<i>Level</i>), die nur einen oder aber mehrere Kommunikationsaufträge (<i>Job, Sequence</i>) gleichzeitig abwickeln können und wahlweise interne Datenpuffer (<i>Buffer</i>) bereitstellen.
Speicherbausteine	
Ansteuerung von internen und externen Flash-ROM und EEPROM-Bausteinen siehe Abschnitt <i>Memory Services</i>	
Kommunikationsschnittstellen	Treiber für interne und externe CAN-, LIN-, FlexRay- oder Ethernet-Controller siehe Abschnitt <i>Kommunikationsstack</i>

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 Fehler-speicher bildet.

Zwischen den Hardwaredreibern 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 Initialisierungsroutine `IoHwAb_Init...` () sowie `IoHwAb_Get...` () und `IoHwAb_Set...` () Funktionen für den Zugriff auf die Signale bereitstellen sollen.

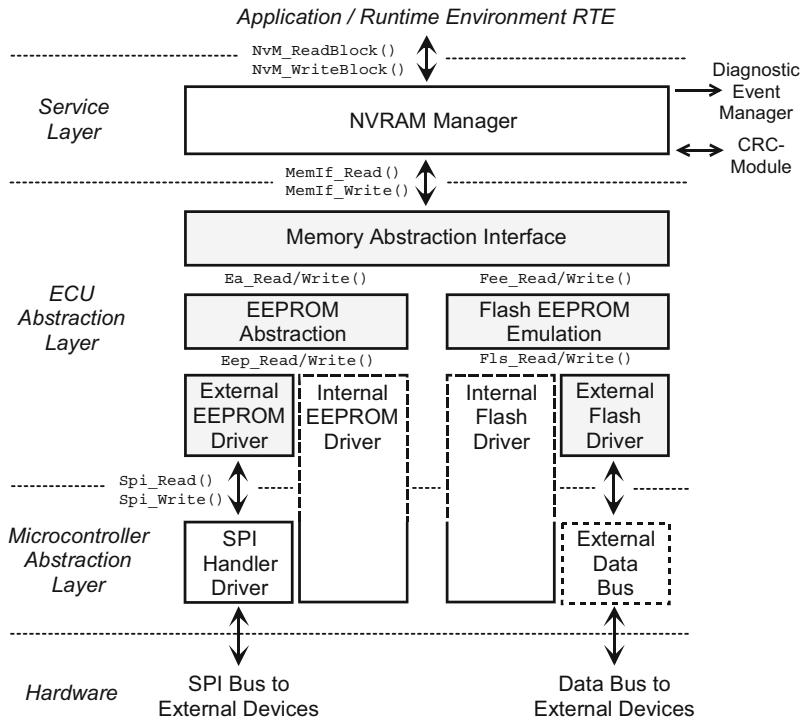


Abb. 8.4 Aufgabenverteilung am Beispiel der *Memory Services*

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

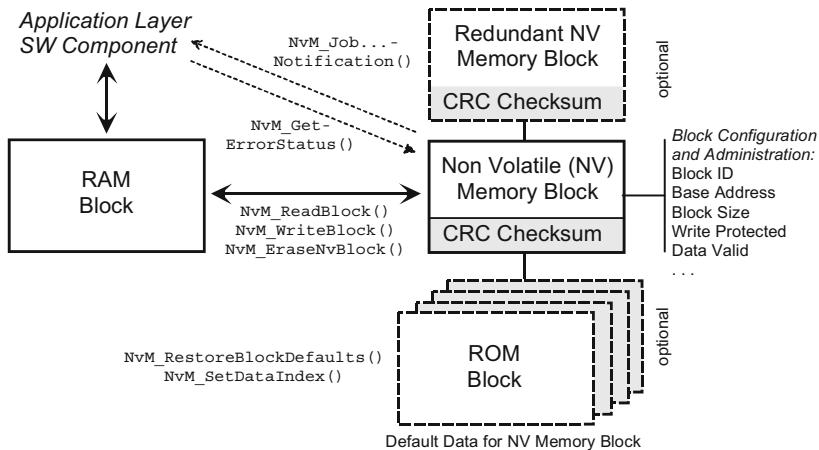


Abb. 8.5 Verwaltung von Daten im nicht-flüchtigen Speicher

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 Bausteinwahl 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

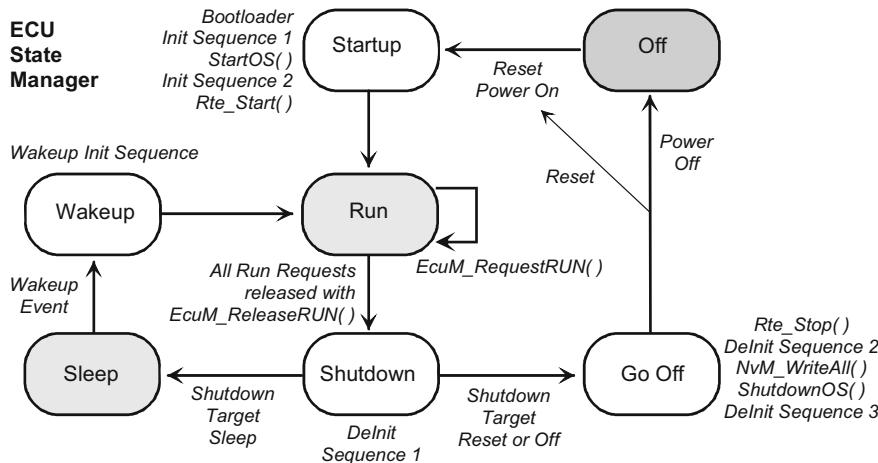


Abb. 8.6 Betriebszustände des Steuergerätes (vereinfacht)

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 Hardwaredtreibern 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 Hardwaredreiber 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 Diagnosiedienste 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-

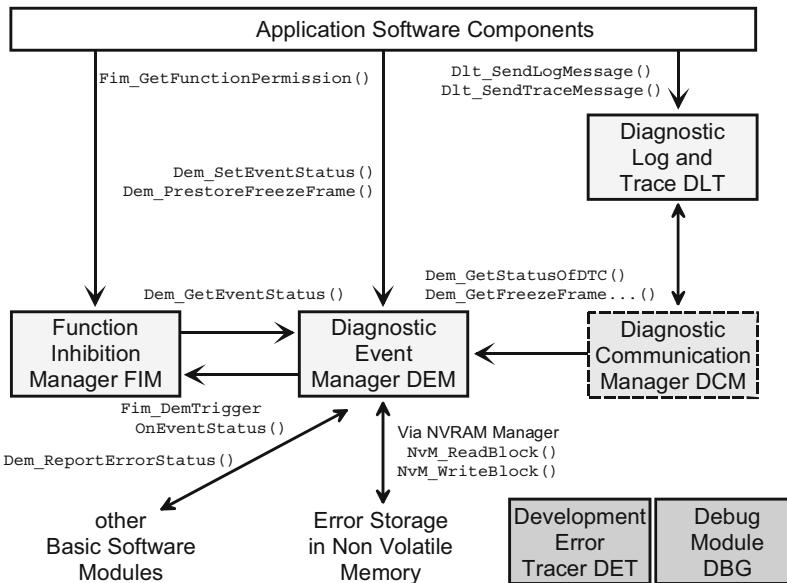


Abb. 8.7 Zentraler Fehlerspeicher und Funktionsauswahl

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 Ausführungsduer 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 Sequenzzählervwert (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

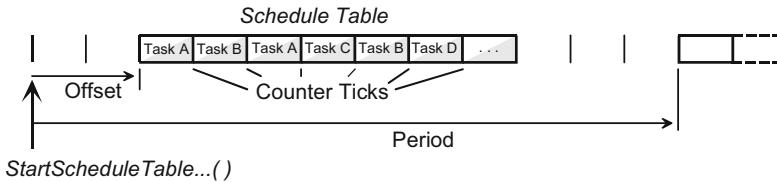


Abb. 8.8 Schedule Tabellen

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 (*Execution 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 Alarne, 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, Alarne 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 Initialisierungs-, 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 Nutzfahrzeubereich weitverbreiteten SAE J1939 Protokolls (Abschn. 4.5) für CAN. Zu dessen Unterstützung sind parallel zum Standard DCM ein spezieller *J1939 DCM*, ein

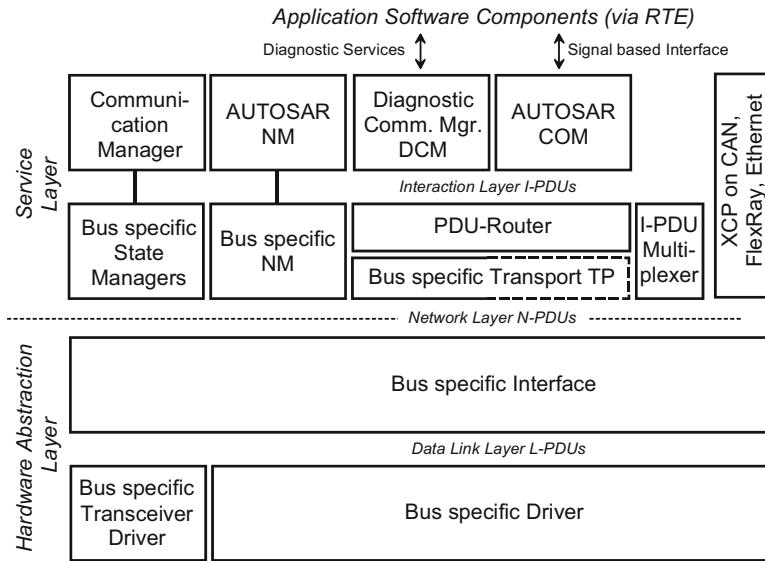


Abb. 8.9 Kommunikationsstack mit jeweils busspezifischen Komponenten für CAN, FlexRay, LIN und Ethernet

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 Hardwaredreiber 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 Managers* 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/pассив* 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-Diagnosesbotschaften 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

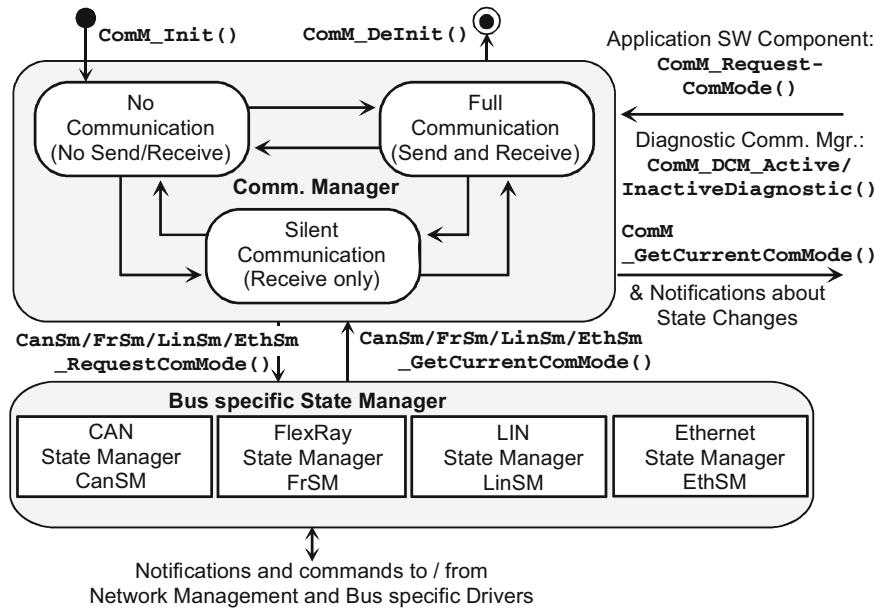


Abb. 8.10 Buszustände des *Communication Manager* (vereinfacht)

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-

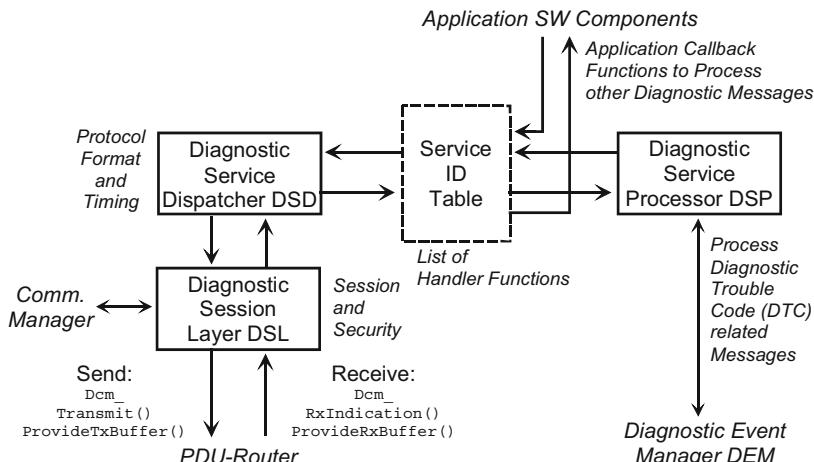


Abb. 8.11 Struktur des Diagnostic Communication Managers DCM

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 sendeseitige 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

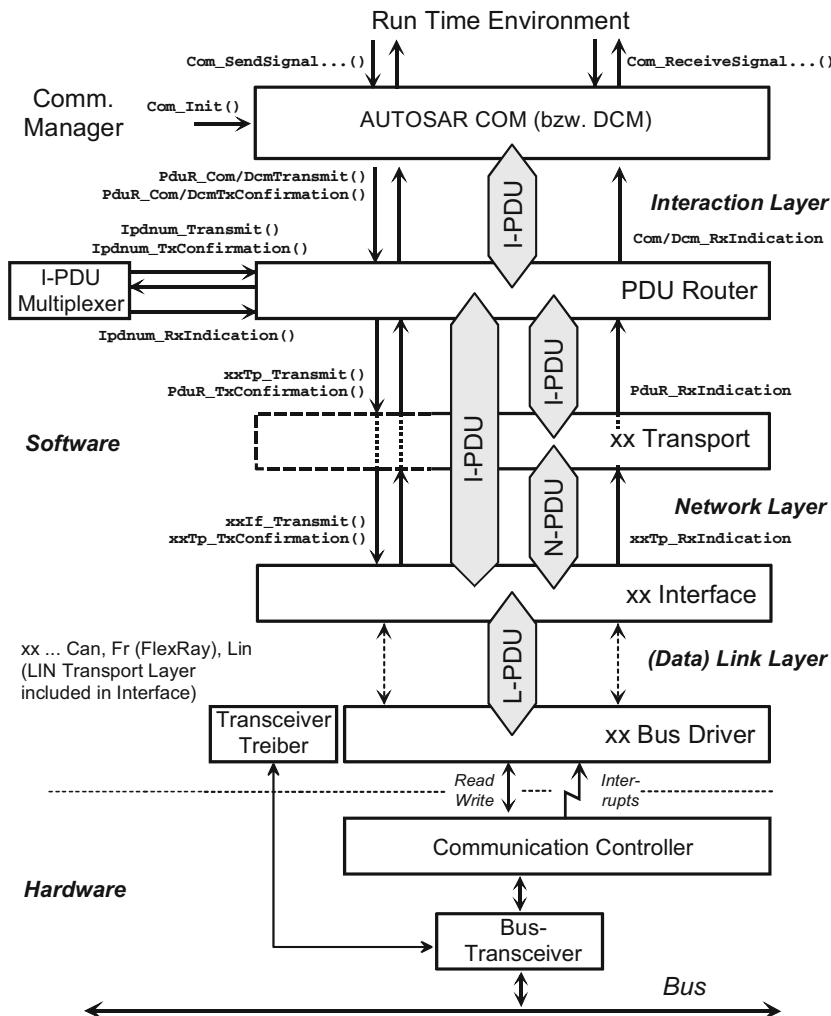


Abb. 8.12 Prinzip des AUTOSAR-Protokollstapels

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 signal-orientierte 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 Aufrufschaltung. 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.

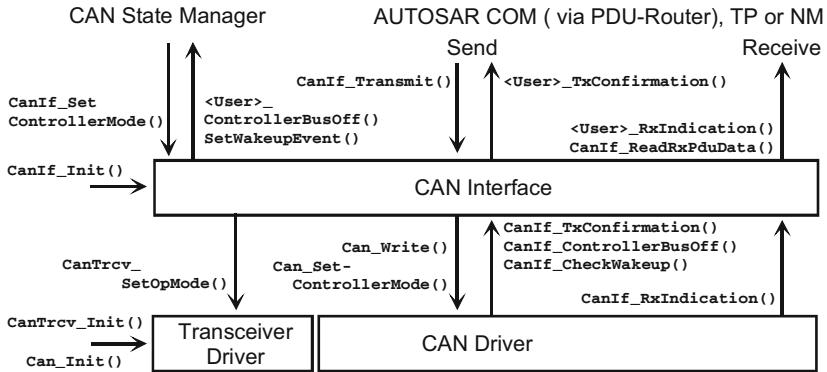


Abb. 8.13 CAN-Protokollstapel mit einer Auswahl von API-Funktionen

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

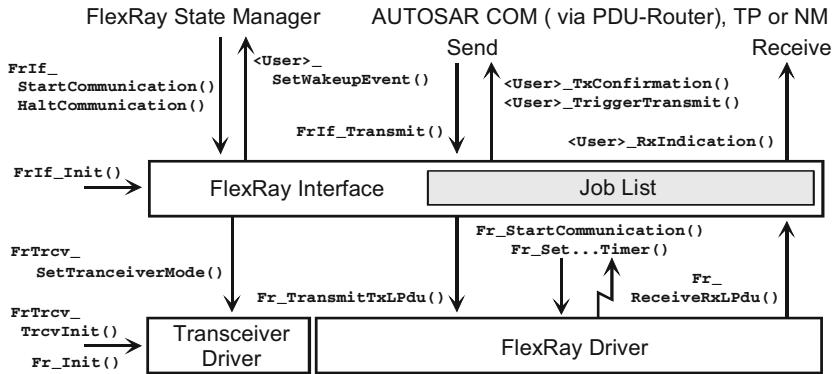


Abb. 8.14 FlexRay-Protokollstapel mit einer Auswahl von API-Funktionen

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 Bitrate-tenumschaltung 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*

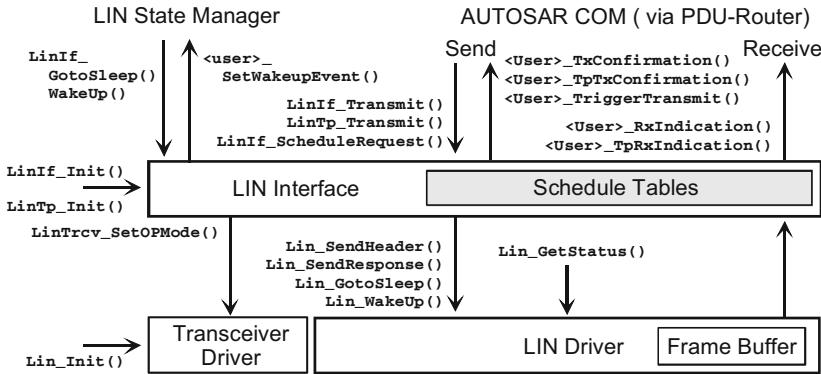


Abb. 8.15 LIN-Protokollstapel mit einer Auswahl von API-Funktionen

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-Steuengeräten (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.

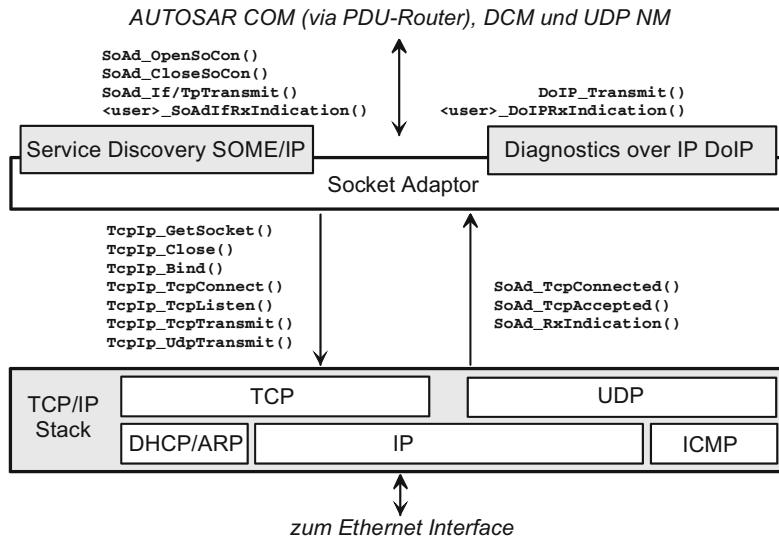


Abb. 8.16 Oberer Teil des TCP/IP-Ethernet-Protokollstapels

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 Manager*en wird der Betrieb des eigenen Steuerge-

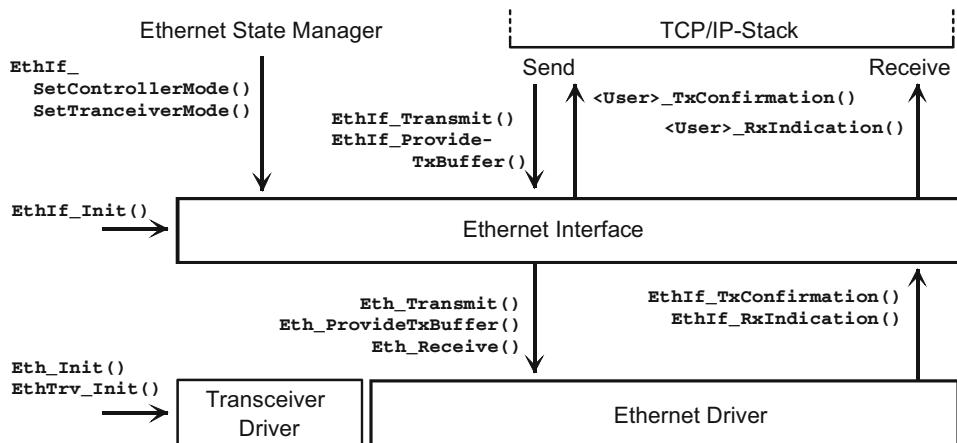


Abb. 8.17 Unterer Teil des TCP/IP-Ethernet-Protokollstapels

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):

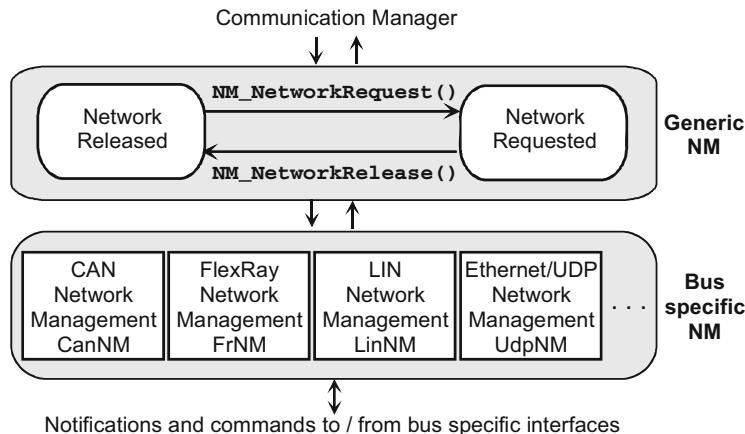
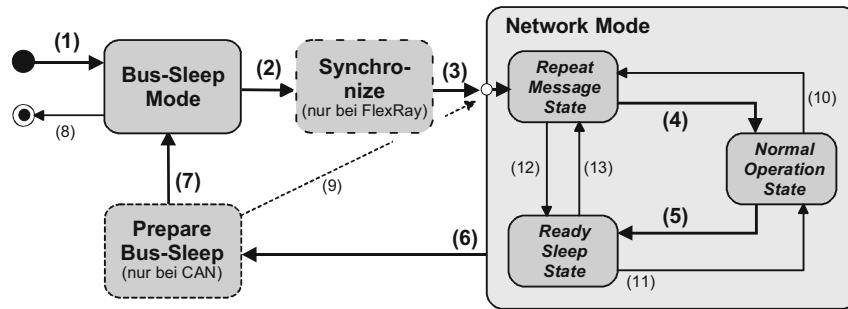


Abb. 8.18 Struktur der Netzmanagement-Komponenten

**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 Zustandsmodell, 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 zwis-

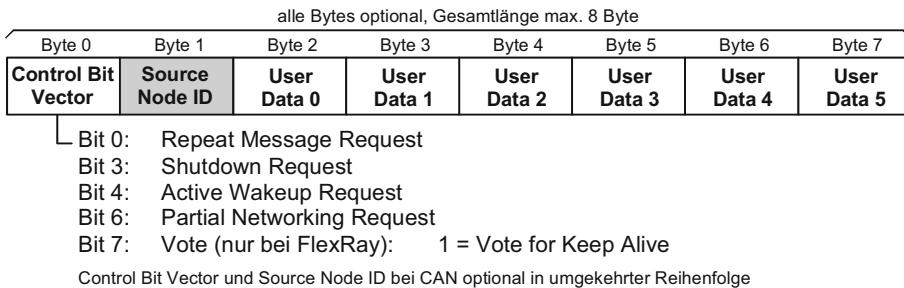


Abb. 8.20 Botschaftsaufbau für das Netzmanagement (NM PDUs)

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 Netzeilnehmer 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 Bus- system 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 Netzeilnehmer NM-Botschaften empfangen werden. Sendet kein anderer Netzeilnehmer 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 komplett 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 Softwareteilelfunktionen 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*).

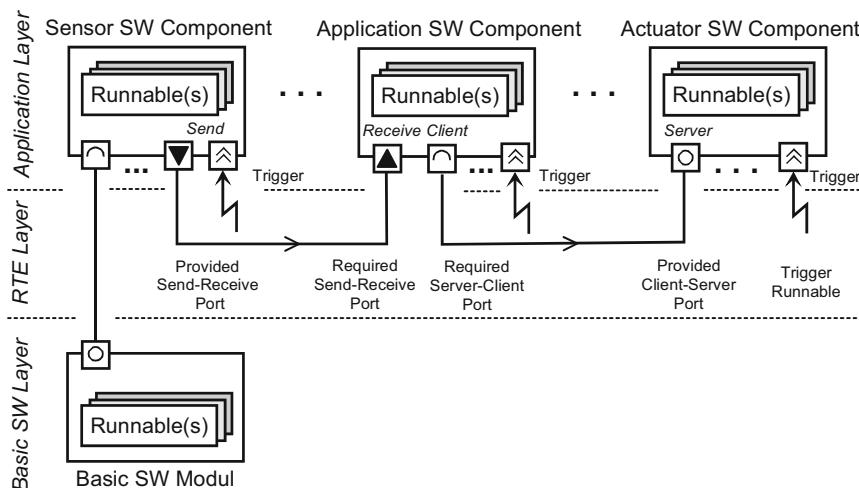


Abb. 8.21 Kommunikation von Softwarekomponenten über das RTE

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()	Lesen bzw. Schreiben eines Datenwertes (ohne FIFO-Queue, d. h. der aktuelle Wert wird gelesen bzw. überschrieben).
Rte_Write()	Datum als ungültig kennzeichnen bzw. auf Initialisierungswert zurücksetzen
Rte_Receive()	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 (<i>Data Received Event oder Data Receive Error Event</i>). Beim Senden wird nur der Sendauftrag an AUTOSAR COM erteilt, aber nicht gewartet, bis das Senden tatsächlich erfolgt ist
Rte_Send()	
Rte_Feedback()	Warten, bis das Senden eines Datums nach Aufruf von Rte_Send() tatsächlich erfolgt ist (<i>Data Send Completed Event</i>)

Implizites Lesen und Schreiben

Rte_IRead()	Lesen eines Datenwerts
Rte_IWrite()	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 (<i>Asynchronous Server Call Returns Event</i>)

Kommunikation und Synchronisation innerhalb einer Softwarekomponente

Rte_Irv(I)Read()	Explizites oder implizites Lesen und Schreiben von <i>Interrunnable Variablen</i>
Rte_Irv(I)Write()	
Rte_Enter()	Synchronisation durch kritische Abschnitte (<i>Exclusive Areas, Critical Sections</i>)
Rte_Exit()	

Betriebsmodi des Steuergerätes

Rte_Mode()	Lesen oder Umschalten eines der Betriebsmodi über den <i>ECU State Manager, Communication Manager</i> oder eine andere Komponente
Rte_Switch()	

Allgemeiner Ablauf

Rte_Start()	Initialisieren und Beenden des RTE durch den <i>ECU State Manager</i>
Rte_Stop()	

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 einer Zeitgebers (OSEK OS Alarm) für das periodische Triggern einer <i>Runnable Entity</i>	Ja	Nein
Mode Switch Event	Nach Umschalten eines der Steuergerätebetriebsmodi	Ja	Nein
<i>Ereignisse bei einem Sender/Receiver Port-Interface</i>			
Data Received	Empfang einer Datenbotschaft	Ja	Ja
Data Receive Error	Empfangsfehler Beim Empfang von Daten kann optional ein Datenfilter konfiguriert werden, so dass der Wertebereich von Daten o. Ä. überprüft werden kann.	Ja	Nein
Data Send Completed	Sendebestätigung einer Botschaft	Ja	Ja
<i>Ereignisse bei einem Client-Server Port-Interface</i>			
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 Signalisierungseignis 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-

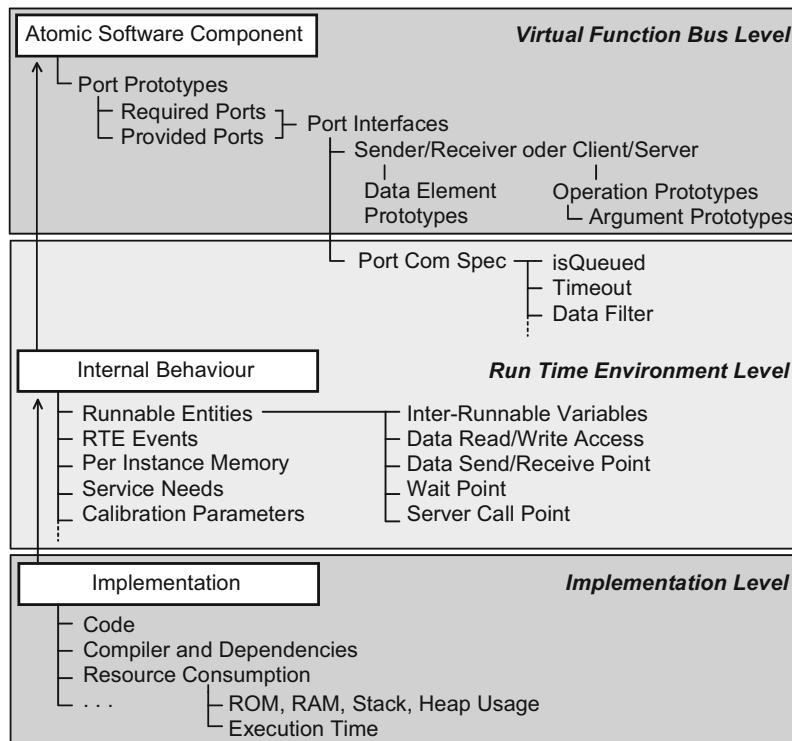


Abb. 8.22 Konfiguration einer AUTOSAR-Softwarekomponente (vereinfacht)

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 Generierungswerzeuge mehr oder weniger automatisch in entsprechenden Mechanismen und API-Aufrufe von AUTOSAR OS und AUTOSAR COM abgebildet.

Diese Generierungswerzeuge, 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, Alarne 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 Anzeigegeräte, 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

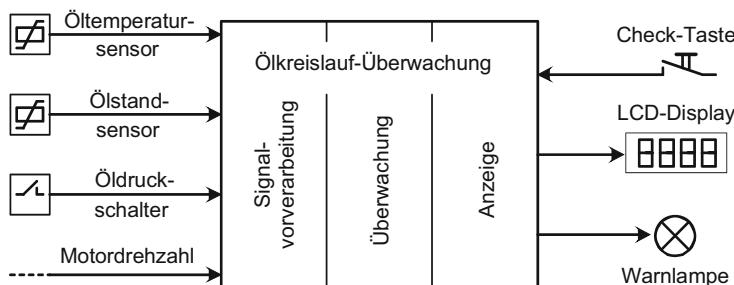


Abb. 8.23 Ölkreislauf-Überwachung

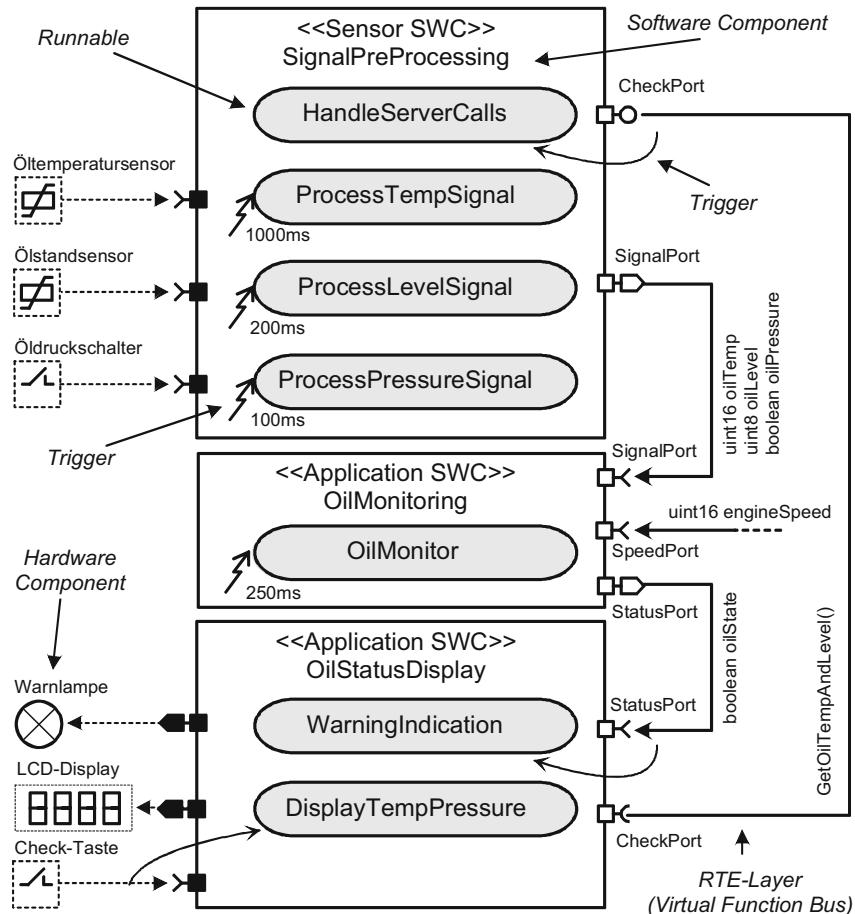


Abb. 8.24 AUTOSAR-Modell der Ölkreislauf-Überwachung

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 Hardwaredtreiber 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 Arbeitspakte 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. In soweit 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

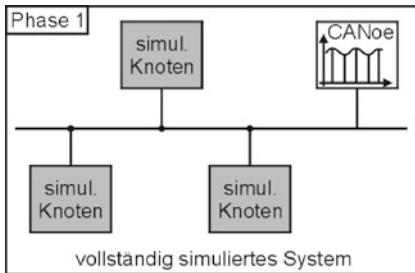
Während in den vorigen Kapiteln vor allem die standardisierten Bussysteme und Protokolle beschrieben wurden, sollen in diesem Abschnitt typische Anwendungen und die zugehörigen Werkzeuge im Vordergrund stehen.

9.1 Entwurf und Test der On-Board-Kommunikation

Entwurf und Erprobung der Kommunikation zwischen den Steuergeräten ist heute einer der aufwendigsten und für die Zuverlässigkeit wichtigsten Entwicklungsschritte. Die Aufgabe kann nur mit durchgängiger Werkzeugunterstützung sinnvoll bewältigt werden. Die typischen Entwicklungsschritte sollen hier exemplarisch anhand der weit verbreiteten Werkzeugkette *CANoe* der Firma Vector Informatik dargestellt werden.

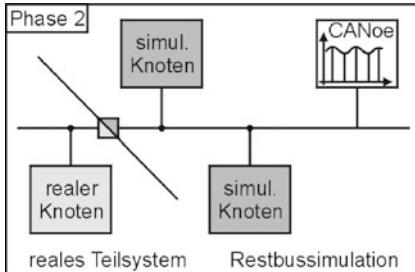
9.1.1 Entwicklungsprozess mit *CANoe* von Vector Informatik

CANoe unterstützt den kompletten Entwicklungsprozess für Steuergeräte und Netzwerke für die Bussysteme CAN, CAN FD, LIN, MOST, FlexRay und Ethernet/IP. Dabei gliedert sich der Prozess in drei Phasen. Diese werden sowohl beim Fahrzeughersteller (OEM) als auch bei den Zulieferern durchlaufen und basieren durchgängig auf denselben Datenbanken und Simulationsmodellen. OEM-spezifische Ausprägungen der Kommunikationsmodelle werden ebenso unterstützt wie unterschiedliche Implementierungen für die Transportprotokolle und das Netzmanagement. Die Phasen sind:



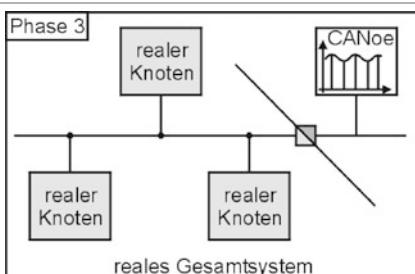
Phase 1: Netzwerkdesign und Simulation

In der ersten Phase der Entwicklung legt der OEM die komplette Kommunikation zwischen den einzelnen Netzknopen fest. Durch eine Simulation in CANoe kann verifiziert werden, ob der Entwurf vollständig ist und Latenzzzeiten sowie Busauslastung die Vorgaben einhalten.



Phase 2: Restbussimulation, Test und Analyse einzelner Steuergeräte

In der zweiten Phase der Entwicklung wird der CANoe-Datensatz aus Phase 1 als ausführbare Spezifikation an die Zulieferer für die Entwicklung der Steuergeräte übergeben. Jeder Zulieferer testet das Kommunikationsverhalten seines eigenen Gerätes dabei gegen die mit CANoe simulierten anderen Netzknopen. Am Ende der Phase 2 stehen fertig implementierte und beim Zulieferer vollständig getestete Steuergeräte.



Phase 3: Integration und Test des gesamten Netzwerks

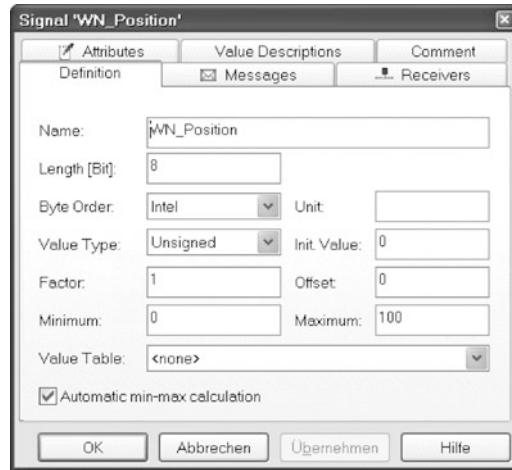
In der letzten Phase wird beim OEM das gesamte Netzwerk integriert. Stück für Stück werden die einzelnen Steuergeräte zu einem realen Gesamtsystem zusammengefügt und die endgültige Kommunikation sowie die einzelnen Funktionen geprüft.

9.1.2 Netzwerkdesign mit dem Network Designer

Typischerweise wird bei den Fahrzeugherrstellern heute für das System- und Netzwerkdesign eine übergeordnete Datenbank-Lösung eingesetzt, die alle Informationen zu den Steuergeräten, Bussystemen, Signalen und Varianten der Fahrzeugplattformen enthält. Für die einzelnen Projekte können daraus die unterschiedlichen Kommunikationsmatrizen für die jeweiligen Bussysteme automatisch generiert werden. Je nach Hersteller sind dies klassische Formate wie *DBC*, *LDF* oder *FIBEX*, aber immer häufiger auch *AUTOSAR System Description* oder *AUTOSAR ECU Extract*. Da diese Datenbanklösungen sehr komplex sind, wird der Designprozess hier zunächst anhand eines Stand-alone-Werkzeugs vorgestellt, einen Überblick über eine vollständige Werkzeugkette gibt dann das folgende Abschn. 9.2.

In der ersten Phase der Entwicklung eines Netzwerks wird die Kommunikationsmatrix festgelegt. Diese beschreibt, welche Informationen in einem Fahrzeugnetz zwischen

Abb. 9.1 Definition eines Signals



den einzelnen Netzknoten ausgetauscht werden. An dieser Stelle kommt der *Network Designer* zum Einsatz, der sämtliche Festlegungen in einer Datenbank für den weiteren Entwicklungsprozess speichert. Der Datenbankinhalt kann jederzeit im CANdb-, LDF- oder FIBEX-Format exportiert werden, so dass alle Werkzeugketten unterstützt werden, die auf diesen Formaten aufsetzen.

Zunächst werden die Netznoten und die einzelnen Signale definiert. Signale sind diejenigen Informationen, die die Funktionen in den verschiedenen Netznoten untereinander austauschen wollen. Signale werden im Wesentlichen durch ihre Länge und ihren Datentyp beschrieben, zusätzlich lassen sich ein Initialisierungswert und eine Umrechnungsformel zwischen dem physikalischen und dem hexadezimalen Wert zuordnen (Abb. 9.1). Im nächsten Schritt erfolgt die Zuweisung der Signale zu den jeweiligen Sendeknoten. Jedes Signal wird von genau einem Sendeknoten versendet (*Transmit TX*). In einem weiteren Schritt werden die Empfänger-Knoten (*Receive RX*) für die einzelnen Signale festgelegt. Die so entstandenen Sende- und Empfangsbeziehungen beschreiben den Datenaustausch auf der logisch-funktionalen Ebene und sind zunächst unabhängig vom verwendeten Busystem.

Anschließend folgt die Abbildung (*Mapping*) der Signale auf die eigentlichen Busbotschaften (Abb. 9.2) sowie deren Verteilung auf die Zeitschlitzte (*Slots*) der Kommunikationszyklen bei LIN und FlexRay (*Schedule*). Dabei werden spezielle Kommunikationsanforderungen wie Latenzzeiten, Zykluszeiten oder die Wichtigkeit der Information, aber auch die im System bereitgestellte Busbandbreite berücksichtigt.

Mit dem *Network Designer* können je nach Ausbau CAN-, LIN- und/oder FlexRay-Systeme entworfen werden. Durch die Integration in einem Werkzeug können Signalbeschreibungen in den unterschiedlichen Netzen wieder verwendet werden. Beim Umfang der busspezifischen Festlegungen unterscheiden sich die einzelnen Systeme.

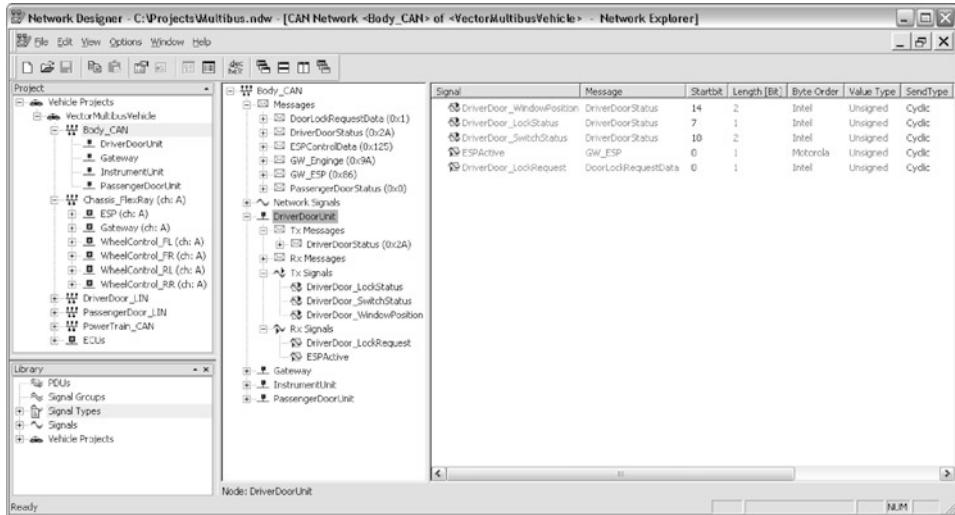


Abb. 9.2 Mapping von Signalen für einen Knoten im *Network Designer*

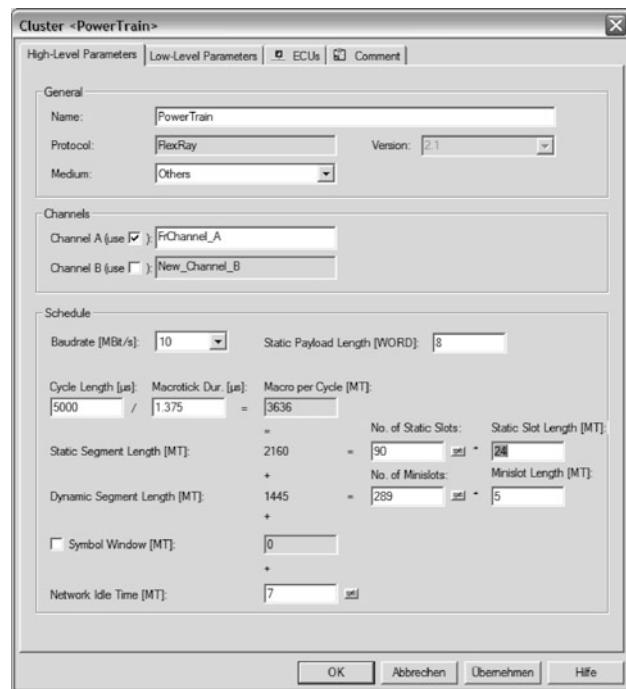
CAN-System Da die Baudrate des Busses in der Regel festliegt, bleiben bei der Verteilung der Signale auf CAN-Botschaften nur die Parameter Botschaftslänge, *Message Identifier* und Sendeverhalten (zyklisch oder ereignisgesteuert) übrig, um die Anforderungen an Latenz und Bandbreite zu erfüllen. Dabei ist abzuwägen, ob man mit längeren Botschaften viele Signale auf einmal überträgt, den Bus aber vergleichsweise lang blockiert, oder nur wenige Signale in kurzen Botschaften sendet, was den Protokoll-Overhead in die Höhe treibt, aber schnellere Reaktionen auf höherpriore Botschaften zulässt.

Das Sendeverhalten hängt vom Anwendungsgebiet und der Designphilosophie des OEM ab. Im Antriebsstrang- und Fahrwerksbereich (*Powertrain*, *Chassis*) werden Informationen oft zyklisch übertragen, während im Karosseriebereich (*Body*) meist ereignisgesteuert gesendet wird. Diese Zusatzinformationen werden in Datenbankattributen abgelegt und dienen der Parametrierung der *CANoe*-Bussimulation sowie der Erstellung von Testfällen wie der Überwachung der Zykluszeit. Außerdem können diese Daten bei der Entwicklung der Steuergerät-Software und der automatischen Code-Generierung verwendet werden.

FlexRay-System Im Gegensatz zu CAN ist das Design einer FlexRay-Architektur deutlich komplexer. Das fängt mit der Definition der Protokollparameter an. Danach gilt es, die Zuweisung der Signale zu den FlexRay Frames festzulegen und zu definieren, ob sie im statischen oder dynamischen Segment und in jedem Zyklus übertragen werden müssen, oder ob aus Gründen der Bandbreitenoptimierung *Cycle Multiplexing* verwendet wird.

Der *Network Designer* führt den Anwender schrittweise durch die FlexRay-Systemdefinition. Im ersten Schritt werden die Grundparameter des FlexRay Clusters von der Län-

Abb. 9.3 Grundparameter für ein FlexRay-Cluster



ge der Mikro- und Makroticks bis zur Gesamtlänge des Kommunikationszyklus definiert (Abb. 9.3). Dabei wird auch festgelegt, wie lang das statische und das dynamische Segment sowie die *Network Idle Time* sind und wie viele statische oder Mini-Slots es gibt. Der *Network Designer* prüft jeweils, ob die eingegebenen Werte mit den übrigen Daten konsistent sind, zeigt Fehler an und schlägt gegebenenfalls korrigierte Werte vor. Falls z. B. die Länge des statischen Segments und die Anzahl der enthaltenen Slots bereits eingestellt sind und anschließend die Slotlänge so gewählt wird, dass die Gesamtlänge überschritten würde, gibt die Überprüfung dem Anwender die Möglichkeit, die Anzahl bzw. Länge der Slots oder die Gesamtlänge des statischen Segments abzuändern und schlägt jeweils passende Werte vor.

Der zweite Schritt unterscheidet sich dann nicht wesentlich von einem CAN-System. Hier werden wieder die Signale, die sendenden und empfangenden Knoten sowie das Mapping der Signale auf die FlexRay Frames festgelegt.

Der abschließende dritte Schritt ist typisch für ein *Schedule*-basiertes System. In diesem Schritt werden die einzelnen Frames den Slots und bei *Cycle Multiplexing* den jeweiligen Kommunikationszyklen zugeordnet (Abb. 9.4).

LIN-System Signale und Botschaften werden im Wesentlichen wie bei CAN konfiguriert, die Scheduling Tabellen wie bei FlexRay, wobei aber weit weniger Parameter festzulegen sind.

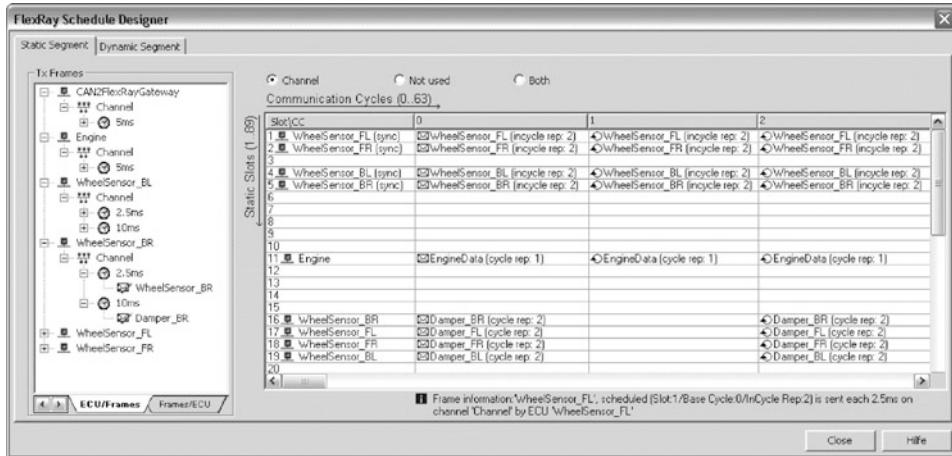


Abb. 9.4 Schedule eines FlexRay-Systems

CAN FD-System Für die neuen Bussysteme CAN FD und Ethernet/IP gibt es noch kein verbindliches Vorgehen bei der Definition der Kommunikation, solange sich diese Projekte bei den Herstellern noch im Vorentwicklungsstadium befinden. Bis eine Standardisierung erfolgt ist wird je nach Hersteller und Werkzeug eine Erweiterung von DBC, FIBEX oder der AUTOSAR-Formate bevorzugt, um schnell Ergebnisse zu erhalten.

Bei CANoe kann der *CANdb++-Editor* auch für die Netzwerkbeschreibung von CAN FD verwendet werden. Dabei ist das Vorgehen prinzipiell dasselbe wie bei konventionellen CAN-Systemen. Nachdem die Entscheidung getroffen ist, welche der CAN FD-Neuerungen (Botschaftslänge bis zu 64 Byte, höhere Datenrate) verwendet werden, müssen die zusätzlichen Informationen in der Datenbank eingetragen werden. Für die erhöhte Datenrate wird die dort verwendete Bitrate festgelegt, bei den CAN-Botschaften die Datenlänge DLC. Das Signal-*Mapping* erfolgt wie bei konventionellen CAN-Systemen.

Ethernet/IP-System Bei Ethernet/IP-Netzen hat sich noch kein allgemeingültiger Standard herausgebildet. Hier geht der Trend in Richtung AUTOSAR-Beschreibung.

CANoe bietet im Moment die Lösung, eigene signalbasierte Protokolle in DBC zu beschreiben und im Programm auf diese Signale zuzugreifen. Für die Interpretation und Simulation von Netzwerken die den SOME/IP-Standard verwenden, erfolgt das Netzwerkdesign in FIBEX.

9.1.3 Simulation des Gesamtsystems in CANoe

Basierend auf der Datenbasis ist mit CANoe eine komplette Simulation der Kommunikation möglich. Dabei wird das Kommunikationsverhalten der Knoten gegen das Bussystem

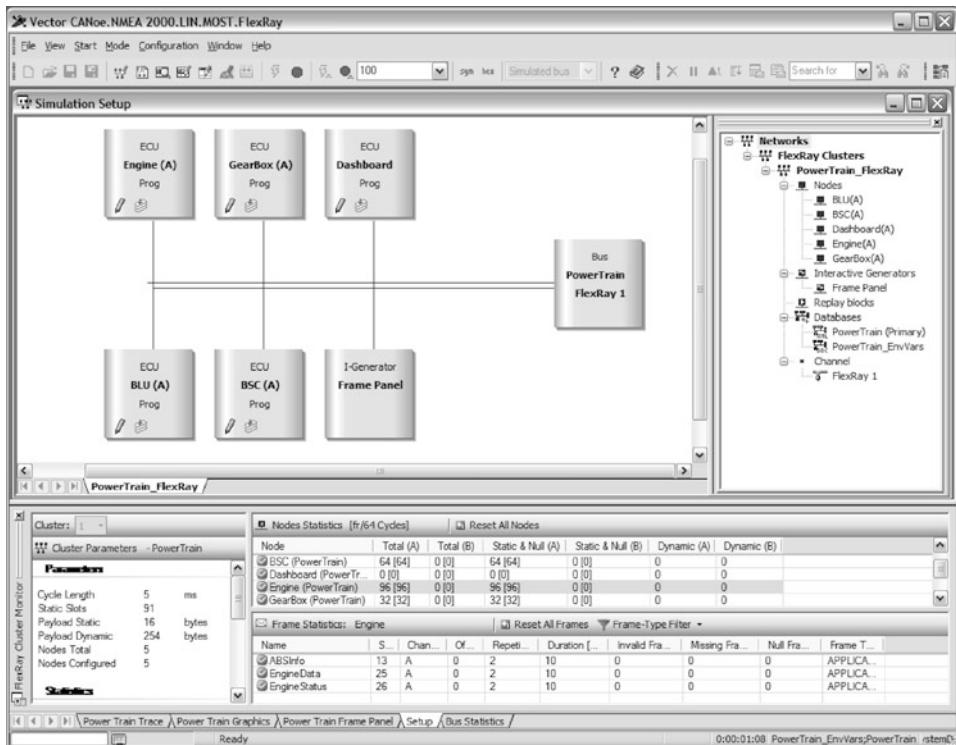


Abb. 9.5 FlexRay-Simulation in CANoe

simuliert. Eine vollständige Simulation der Applikation ist hier nicht erforderlich und auf Basis der Datenbankinformationen alleine auch gar nicht möglich. Für CANoe spielt es bei der Simulation keine Rolle, ob die Kommunikation in DBC, LDF oder FIBEX beschrieben ist oder als AUTOSAR System Description vorliegt.

In einem FlexRay- oder LIN-System ist das Kommunikationsverhalten bereits durch die *Schedule*-Tabelle weitgehend festgelegt. Mit der Auswahl der zugehörigen Datenbank ist die Systemsimulation in CANoe damit bereits vollständig konfiguriert. Bei einem CAN-System ist das Sendeverhalten dagegen von der Kommunikationsphilosophie des OEM abhängig. In CANoe ist dies durch unterschiedliche OEM-AddOns gelöst. Diese werden zum Standard-Produkt installiert und steuern dann OEM-abhängig die automatische Generierung der Restbussimulation. Nach der Auswahl der Datenbank wird auf Basis der Knoten-, Botschafts-, Signal- und Attributinformationen das komplette System in CANoe erstellt. Dabei werden die einzelnen Busse angelegt, die Knoten zugeordnet und das OEM-spezifische Netzwerk-Management eingerichtet. Zur Bedienung der einzelnen Netzknoten werden gleichzeitig Bedienpanels erzeugt, mit denen der Anwender dann die Sendesignale der einzelnen Knoten ändern oder die Empfangssignale ablesen kann (Abb. 9.5).

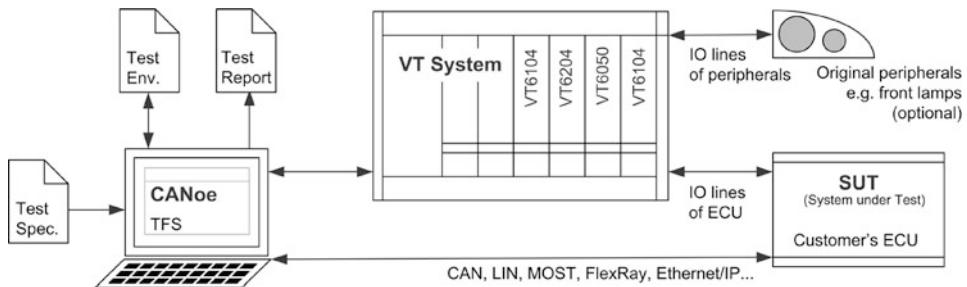


Abb. 9.6 Komplette CANoe-Testumgebung für die Steuergeräteentwicklung

Die Simulation darf mehrere Netze mit unterschiedlichen Bussystemen enthalten. Im simulierten Modell kann geprüft werden, ob alle Botschaften und Signale korrekt und mit der richtigen Zykluszeit übertragen werden und ob die Buslast im gewünschten Rahmen bleibt. Werden Fehler in der Kommunikationsmatrix entdeckt, so können diese im *Network Designer* behoben und in der Simulation erneut überprüft werden. Die fehlerfreie Simulation dient dann in den weiteren Phasen allen Partnern als verlässliche Basis für die Geräteentwicklung und Systemintegration.

9.1.4 Restbussimulation als Entwicklungsumgebung für Steuergeräte

Basierend auf der Datenbasis aus Phase 1 kann jeder Zulieferer nun die Restbussimulation für seine Geräteentwicklung erzeugen. Dazu schaltet er in der Simulation denjenigen Knoten ab, den er selbst entwickelt, und schließt diesen als reales Gerät über geeignete Bussystem-Hardware an das CANoe-System an (Abb. 9.6).

Ohne einen derartigen virtuellen Systemverbund lassen sich Steuergeräte im seriennahen Zustand praktisch überhaupt nicht in Betrieb nehmen, da die geräteinternen Überwachungsmechanismen fehlende Kommunikationspartner sofort erkennen und die Gerätefunktion einschränken. CAN-Steuergeräte etwa brauchen für den fehlerfreien Betrieb in der Regel ein funktionierendes Netzmanagement und einen gewissen Satz an Botschaften auf dem Bus. FlexRay-Steuergeräte benötigen mindestens einen, falls sie selbst nicht als Kaltstart- und Synchronisations-Knoten konfiguriert sind, sogar mindestens zwei weitere Steuergeräte am Bus.

Während der Geräteentwicklung werden dem Steuergerät über die Restbussimulation die nötigen Signalinformationen für die Prüfung der Algorithmen im Steuergerät geliefert. Solche Tests kann der Anwender im einfachsten Fall interaktiv durchführen, indem er die Signale über die Bedienpanels einstellt und dann im Trace-, Daten- oder Graphik-Fenster prüft, ob das Steuergerät die richtigen Werte zurücksendet. Damit können zunächst allerdings nur solche Funktionen geprüft werden, deren Ein- und Ausgangssignale als Busbotschaften verfügbar sind. Um eine komplette Testumgebung aufzubauen, kann

CANoe neben dem reinen Buszugang auch beliebige I/O-Hardware wie Signalgeneratoren oder Messgeräte integrieren (Abb. 9.6). So kann dann z. B. beim Test einer Fensterheber-Funktion auch geprüft werden, ob der Motor des Fensters richtig angesteuert oder der Bedienschalterzustand korrekt auf den Bus gelegt wird. Mit dem *VT-System* existiert eine speziell auf den Automotive-Anwendungsfall zugeschnittene Test-Hardware, die die I/O-Pins eines Steuergeräts stimulieren und messen kann. Fehlende Sensoren kann man über CANoe durch Diagnose- oder CCP/XCP-Botschaften simulieren, die den erforderlichen Signalwert direkt in den Speicher des Steuergerätes schreiben.

Manuelle Tests sind nur schwer dokumentier- und reproduzierbar. Die zuverlässige Prüfung des Zeitverhaltens und der zeitlichen Reihenfolge von Signalen und Botschaften über einen längeren Zeitraum hinweg ist nahezu unmöglich. Darüber hinaus sind bei komplexen Steuergeräten so viele Testfälle notwendig, dass eine mehrfache manuelle Wiederholung des vollständigen Tests bei jeder Softwareänderung in den verschiedenen Projektphasen praktisch unmöglich wäre.

Daher bietet CANoe mit dem *Test Feature Set* eine Automatisierung von Testabläufen an. Dazu beschreibt der Anwender mit den im *Test Feature Set* integrierten *Test Pattern* die einzelnen Testfälle in formaler Form. *Test Pattern* legen die anzulegenden Bus- und sonstigen Eingangssignale (*Eingangsvektor*) sowie die vom Steuergerät erwarteten Reaktionen in Form von Busbotschaften und Ausgangssignalen (*Ausgangsvektor*) sowie den zeitlichen Ablauf vom Anlegen der Signale bis zum Vorhandensein der Ausgangswerte fest. Gleichzeitig ist es möglich, Nebenbedingungen zu definieren, die etwa die Zykluszeiten aller Botschaften eines Knotens überwachen. So kann auch erkannt werden, wenn eine getestete Funktion selbst zwar korrekt arbeitet, aber eine andere Funktion stört. Idealerweise entstehen die einzelnen Testfälle bereits während der Entwicklung der Software und können am Ende zu einem Abschluss-Test zusammengefasst werden. Die Testergebnisse werden in XML-Dateien protokolliert und über XML-Stylesheets in leicht lesbare Formate konvertiert.

Mit dem Autorenwerkzeug *vTESTstudio* (Abb. 9.7) lassen sich Tests in allen von CANoe unterstützten Sprachen, z. B. dem integrierten CAPL oder der Standardsprache C#, programmieren oder über den *Table Text Editor TTE* ohne Programmierung graphisch beschreiben. Die drei Ansätze können beliebig gemischt werden. *vTESTstudio* kann Tests für Steuergerätevarianten erstellen, die aus einem Grundtest durch einfache Parametrierung erzeugt werden, oder mehrere Instanzen eines Tests mit unterschiedlichen Parametern generieren.

Verwaltung und Planung umfangreicher Tests in größeren Teams kann mit dem Werkzeug *vTESTcenter* (Abb. 9.7) erfolgen. Nachdem die Anforderungen an den Prüfling beschrieben oder aus anderen Werkzeugen wie z. B. DOORS importiert wurden, werden konkrete Tests für die einzelnen *Requirements* abgeleitet und Testschritte definiert. Dies können automatische Tests aber auch einfache manuelle Testlisten sein. Nachdem die automatisierten Tests mit *vTESTstudio* implementiert wurden, erfolgt die konkrete Testplanung. Der Anwender definiert die Testumfänge und weist sie unterschiedlichen Bearbeiter zu. Diese führen die Tests mit Hilfe von CANoe aus und importieren die Testergebnisse

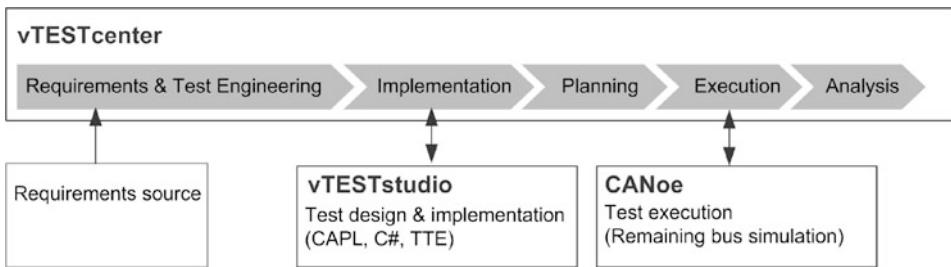


Abb. 9.7 Testplanung und Testdurchführung

wiederum automatisch in *vTESTcenter*. Verschiedene Auswertungen zeigen dem Anwender jederzeit, welche Testfälle ausgeführt wurden, ob die Tests erfolgreich waren und ob alle Anforderungen durch mindestens einen Test abgedeckt wurden.

9.1.5 Integration des Gesamtsystems

Nachdem die einzelnen Zulieferer ihre Steuergeräte fertig entwickelt und getestet haben, erfolgt die Systemintegration beim OEM. Dazu werden zunächst alle elektrischen Komponenten inklusive Kabelstrang, Verbraucher und Bedienelementen in einem Brettaufbau zusammengesetzt. Schritt für Schritt werden die virtuellen Steuergeräte in der Simulation abgeschaltet und real dem System hinzugefügt. Wenn alle Geräte gegen dieselbe Restbus-simulation entwickelt wurden, funktioniert die Buskommunikation meist sofort und die Geräte können ins Fahrzeug integriert werden.

In dieser abschließenden Phase wird *CANoe* vor allem als Analysewerkzeug eingesetzt (Abb. 9.8). Dabei werden die auf dem Bus übertragenen Bits und Bytes mit Zeitrelation aufgezeichnet (Trace). Entsprechende Filterfunktionen erlauben es, die relevanten Botschaften aus einer kompletten Busaufzeichnung z. B. durch Auswahl der CAN-Identifier oder der FlexRay-Slotnummern zu selektieren. Besonders komfortabel wird die Analyse, wenn nicht nur die hexadezimalen Botschaftsdaten angezeigt, sondern die Botschaften gemäß den höheren Protokollsichten decodiert oder Signale direkt als physikalische Werte graphisch dargestellt werden.

9.2 System- und Softwareentwurf für Steuergeräte

Für die in Kap. 4 und 5 beschriebenen Transport- und Diagnoseprotokolle sowie die in Kap. 7 und 8 beschriebenen Betriebssysteme und Basissoftware-Funktionen existieren heute für die im Kfz-Bereich verbreiteten Mikroprozessoren und Entwicklungsumgebungen käufliche Softwaremodule verschiedener Zulieferer (siehe Verzeichnis im Anhang). Diese Module können als Bestandteil der Steuergerätesoftware in die entsprechenden Projekte

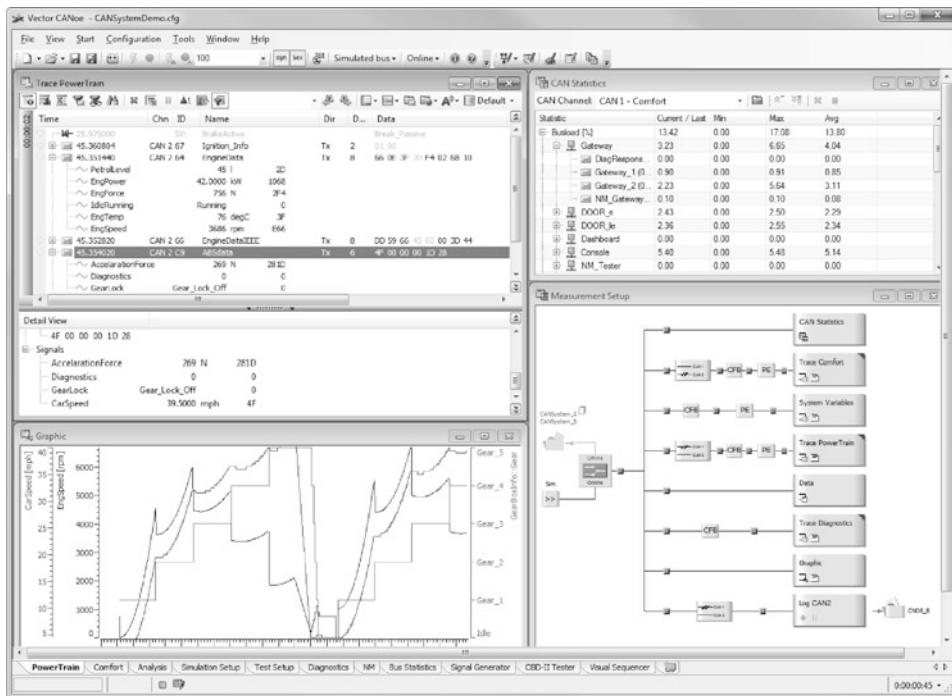


Abb. 9.8 Analyse der Buskommunikation mit CANoe

eingebunden werden. Der Einsatz von bewährten Protokollstapeln sowie Betriebssystemen, Flash-Ladern usw. in Form von standardisierten Modulen verringert nicht nur den Programmieraufwand, sondern reduziert für den Anwender auch den hohen Testumfang, der bei einer Neuimplementierung notwendig wird. Die Hersteller der Module weisen die Konformität ihrer Software in der Regel durch Testprotokolle und Zertifikate (*Conformance Test*) nach.

Für die Auswahl eines Protokollstapels oder Betriebssystems spielen neben den rein technischen Anforderungen wie Speicherplatz- und Laufzeitbedarf, die sich bei den meisten Implementierungen nur wenig unterscheiden, auch andere Aspekte eine Rolle. So muss geprüft werden, ob die beim Anwender zum Einsatz kommenden Software-Module mit relativ geringem Aufwand für neue Mikroprozessoren bzw. neue Entwicklungsumgebungen angepasst werden können. Aufgrund der immer noch vorhandenen regionalen Besonderheiten, z. B. dem Einsatz von SAE- statt ISO-Protokollen in USA, oder herstellerspezifischen Eigenheiten wie dem Einsatz der Diagnoseprotokolle mit unterschiedlichen Transportprotokollen oder Bussystemen, sollte der Zulieferer Protokollstapel für verschiedene Protokolle anbieten, die gegeneinander ausgetauscht werden oder nebeneinander koexistieren können.

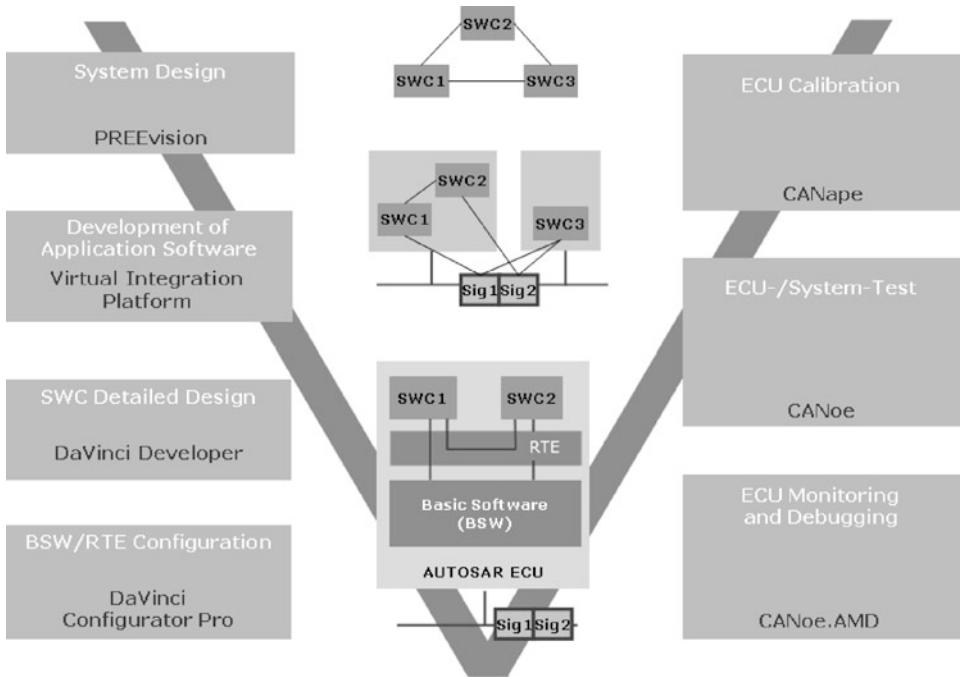


Abb. 9.9 Werkzeugkette für den AUTOSAR-Entwicklungsprozess

Mit der Verbreitung von AUTOSAR wird die Bedeutung solcher Softwarekomponenten (*Intellectual Property*) noch wesentlich wachsen, weil AUTOSAR standardisierte Schnittstellen und Konfigurationsmöglichkeiten für Komponenten als Teil eines durchgängigen Systementwurfs definiert. Werkzeugketten für die Entwicklung von AUTOSAR-Systemen werden beispielsweise als *EB tresos* von Elektrobit oder von Vector Informatik angeboten. Exemplarisch soll hier der Entwicklungsprozess mit den Werkzeugen von Vector Informatik dargestellt werden (Abb. 9.9).

9.2.1 Systementwurf mit *PREEvision* von Vector Informatik

Mit der Einführung der AUTOSAR-Methodik (Kap. 8) hat sich der Entwicklungsprozess nicht grundlegend verändert. Die Integration von AUTOSAR-Steuergeräten in bestehende Netze ist möglich und die Werkzeuge für die Netzwerksimulation und Analyse bleiben gleich. Bei der Systemdefinition und bei den verwendeten Beschreibungsformaten allerdings gibt es größere Änderungen. Der klassische Designansatz geht von kompletten Steuergeräten mit vorgegebenem Funktionsumfang aus, während die Datenverbindung zwischen den Geräten primär als Verkabelungsproblem betrachtet wurde. AUTOSAR dagegen geht von genau beschriebenen Einzelfunktionen mit definierten Schnittstellen aus,

die dann Steuergeräten zugeordnet werden, woraus sich letztlich dann auch die notwendige Kommunikation zwischen den Geräten ergibt.

Das Vector-Werkzeug *PREEvision* deckt diese verschiedenen Entwurfs- und Modellierungsebenen ab:

- Die *System-Software-Architektur* wird in Form von AUTOSAR SWCs (Softwarekomponenten) modelliert, wie in Abschn. 8.6 beschrieben.
- Im Rahmen der *Hardware-Netzwerk-Architektur* werden die Steuergeräte, Sensoren, Aktuatoren und Bussysteme erfasst.
- Die *Kommunikation* zwischen den Steuergeräten wird in Form von Signalen, Botschaften (PDUs, Frames) und Kommunikationszyklen (Schedules) beschrieben.
- Durch ein *Mapping* werden diese Modellierungsebenen in Bezug gesetzt, indem die SWCs einzelnen Steuergeräten zugeordnet und Signale einzelnen Busbotschaften zugeordnet werden.

Über AUTOSAR hinausgehend kann *PREEvision* aber auch Produktziele und Produktanforderungen erfassen, die gesamte logische Architektur eines Fahrzeugs mit Funktionsnetzen beschreiben oder wichtige Details wie Kabelbäume definieren.

9.2.2 Entwicklung der Anwendungssoftware im AUTOSAR-Prozess

Anwendungssoftware wird entweder manuell programmiert oder mit automatischer Codegenerierung modellbasiert entwickelt. Die *Virtual Integration Platform* (Abb. 9.9) unterstützt diese folgenden Phasen der Entwicklung:

- Während der *Systementwicklung* werden die SWCs unabhängig vom konkreten Steuergerät oder Netzwerk in einer virtuellen Umgebung ausgeführt, z. B. dem Entwickler-PC. Die *Virtual Integration Platform* dient in dieser Phase als VFB (Virtual Function Bus) Simulator für die Entwicklung der Fahrzeugfunktionen.
- Während der *Steuergeräteentwicklung* werden die SWCs sowie die HW-unabhängigen Teile der BSW (Basissoftware) zunächst ebenfalls in der virtuellen Umgebung ausgeführt. Die *Virtual Integration Platform* dient in dieser Phase als Integrationsumgebung. Soweit nicht bereits mit *PREEvision* erledigt, können Details der Softwarekomponenten wie Port-Verknüpfungen oder die Zuweisung von Runnables mit den graphischen Editoren des *DaVinci Developers* definiert werden (Abb. 9.10). Kombiniert man die *Virtual Integration Platform* mit CANoe, kann man sowohl Unit-Tests einzelner Softwarekomponenten oder Gruppen von Softwarekomponenten (*Compositions*) als auch Softwareintegrationstests einschließlich des Tests der Buskommunikation in der simulierten Umgebung durchführen. Bei den Unit Tests werden die Ports der SWC und die Runnables in Form von Programmierschnittstellen (API) bereitgestellt, die gezielt mit Eingangsdaten versorgt, ausgeführt und deren Ergebnisdaten ausgewertet werden können (Abb. 9.11).

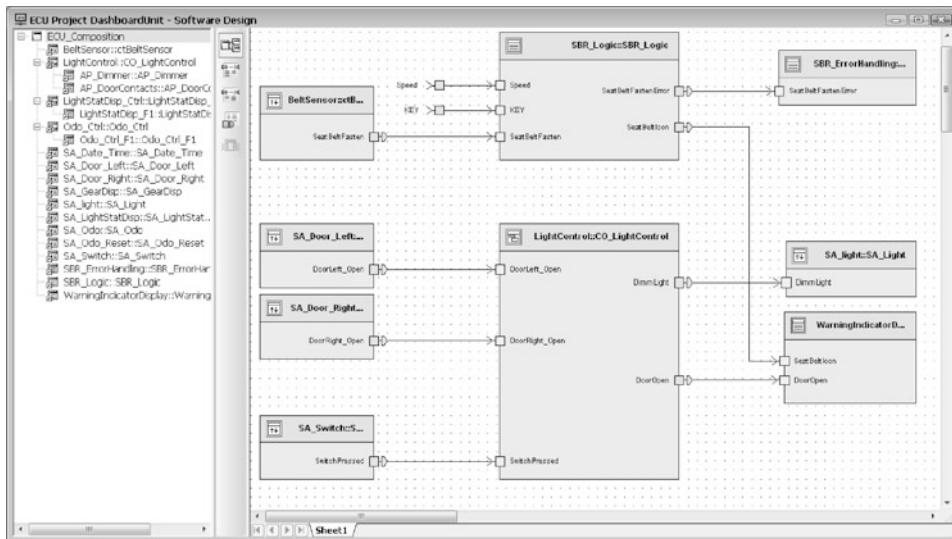


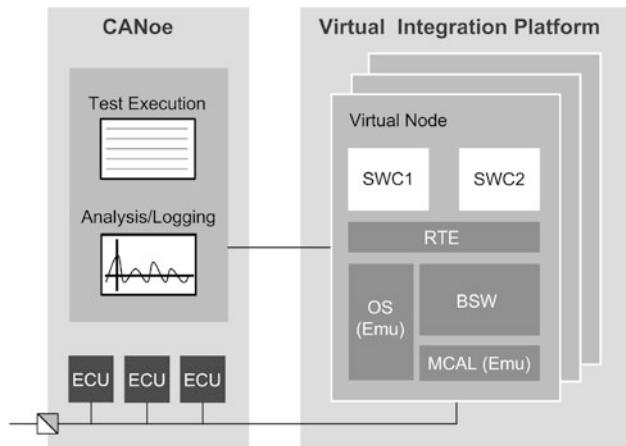
Abb. 9.10 Verknüpfung von Softwarekomponenten im *DaVinci Developer*

- Die *Konfiguration der Basissoftware* (AUTOSAR BSW) und der zwischen Anwendungssoftware und Basissoftware liegenden RTE-Schicht erfolgt im *DaVinci Configurator Pro*. Die initiale Konfiguration wird aus der Systembeschreibung abgeleitet, wobei Änderungen der Systembeschreibung durch eine Update-Funktion im Projektverlauf ständig nachgezogen werden können. Die Basissoftware wird über Editoren und Assistenzfunktionen konfiguriert, die die einzelnen BSW-Domänen wie Base Services, Memory Management, Mode Management oder Runtime System gezielt unterstützen. Eine Validierungsfunktion analysiert die Konfiguration, meldet Inkonsistenzen und schlägt Lösungen vor. Zur Integration von BSW-Modulen verschiedener Hersteller lassen sich die Beschreibungsdateien externer Module importieren und diese dann in einem generischen Konfigurationseditor bearbeiten.
- Sobald die reale Hardware vorhanden ist, kann man die virtuelle Plattform parallel zur realen Plattform betreiben. Die Software wird auf die reale Plattform übertragen, getestet und die Testergebnisse mit der simulierten Umgebung verglichen.

9.2.3 Systemtest und Applikation

Beim Debugging der Steuergeräte und im anschließenden Systemtest wird wiederum vor allem CANoe eingesetzt, wie bereits in Abschn. 9.1 beschrieben. Dessen Option AMD erlaubt einen AUTOSAR-artigen Blick ins Innere des Steuergerätes, weil über das XCP-Protokoll (Abschn. 6.2.2) direkt Ports, die Ausführung von Runnables oder der Zustand von BSW-Komponenten beobachtet werden kann.

Abb. 9.11 Testumgebung für die AUTOSAR-Softwarekomponenten



Die anschließende Applikation, d. h. die Datenanpassung der Anwendungssoftware an ein konkretes Fahrzeug (Abb. 9.9), kann mit *CANape* erfolgen, das im folgenden Abschnitt beschrieben wird.

9.3 Werkzeuge zur Applikation von Steuergeräten

In der Applikationsphase, d. h. der Feinjustierung der Parameter der elektronischen Systeme eines Fahrzeugs, wird der Applikationsingenieur durch PC-basierte Anwendungen unterstützt, die nicht nur die Analyse der Buskommunikation sondern den direkten Zugriff auf steuergeräteinterne Werte ermöglichen.

Typische Applikationswerkzeuge sind z. B. *INCA* der Firma ETAS (Abb. 9.12) oder *CANape* von Vector Informatik, mit denen steuergeräteinterne Werte nicht nur in Zahlen- darstellung sondern auch in graphischer Form dargestellt und verändert werden können. Der Zugriff auf die Werte im Steuergerät erfolgt über das reguläre Bussystem mit einem der ASAM-Protokolle CCP bzw. XCP (vgl. Abschn. 6.2) oder ein proprietäres Interface oder Protokoll wie ETAS ETK oder Bosch McMess. Neben den Mess- und Kalibrierarbeiten muss das Werkzeug auch Steuergeräte-Beschreibungen verwalten, die z. B. im ASAM ASAP2-(A2L)-Format dargestellt werden (Abschn. 6.5).

Über XCP lassen sich auch sogenannte Bypass-Strukturen aufbauen (Abb. 9.13), bei denen eine Teilfunktion des Systems zu Experimentierzwecken nicht im Steuergerät sondern in einem Echtzeit-Simulationsprogramm, z. B. *ASCET* von ETAS, *Target Link* von dSpace oder *Matlab/Simulink* von Mathworks implementiert wird.

Dabei werden Eingangssignale und interne Daten des Steuergerätes über DAQ-Botschaften vom realen Steuergerät zum Prototyping-System und dessen Berechnungsergebnisse über STIM-Botschaften zurück an das Steuergerät übertragen. Auf diese Weise lassen sich im Rapid Prototyping-System Funktionen zunächst ohne Rücksicht auf die be-

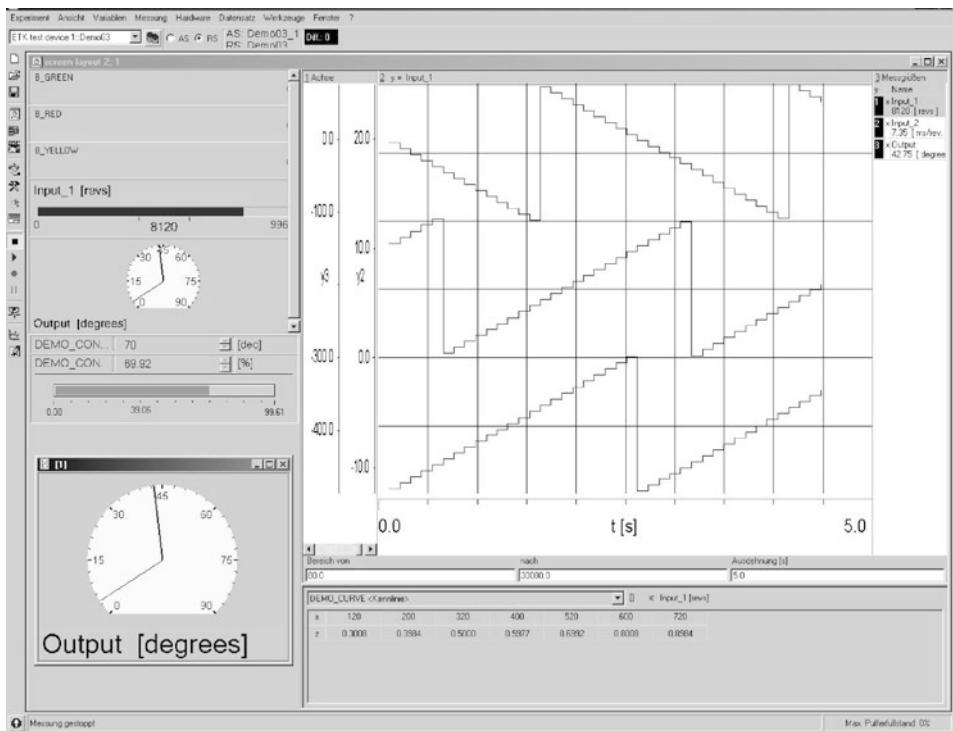


Abb. 9.12 Typisches Applikationswerkzeug zur Erfassung von Messwerten und Verstellung von steuergeräteinternen Parametern (ETAS INCA)

schränkten Ressourcen realer Steuergeräte entwickeln und anschließend, oft unter Verwendung automatischer Code-Generierungswerkzeuge, in das reale Steuergerät integrieren. In ähnlicher Weise werden Steuergeräte auch in Hardware- oder Software-in-the-Loop-Simulatoren (HIL) integriert.

9.3.1 Steuergeräte-Applikation mit **CANape** von Vector Informatik

Das Mess-, Kalibrier- und Diagnosewerkzeug **CANape** von Vector Informatik ist ein universelles Werkzeug zur Applikation von Steuergeräten (Abb. 9.14). Es bietet neben den Mess- und Verstellmöglichkeiten über die ASAM-Protokolle CCP und XCP auch den Zugriff auf Diagnosedaten und Diagnosedienste inklusive der OBD-Daten und liefert eine komplett Sicht auf die steuergeräteinternen Abläufe. Darüber hinaus unterstützt es durch die Schnittstelle zu Matlab/Simulink die modellgestützte Entwicklung.

Bei der Steuergeräte-Applikation spielen drei Typen von Beschreibungsdateien eine wesentliche Rolle:

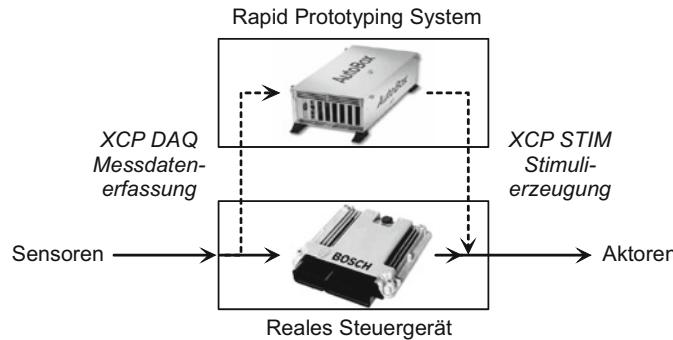


Abb. 9.13 Steuergerät mit Rapid-Prototyping-System in Bypass-Struktur

- Beschreibungsdateien für steuergeräteinterne Größen, z. B. ASAP2,
- Beschreibungsdateien der verschiedenen Kommunikationsnetze, z. B. DBC, LDF oder FIBEX,
- Beschreibungsdateien für Diagnoseservices und -daten sowie den Flash-Vorgang, z. B. ODX-Datensätze.

Um die Beschreibungsdateien anzuzeigen und zu editieren, enthält *CANape* verschiedene Editoren. Mit dem integrierten ASAP2-Editor werden die Applikationsdatensätze

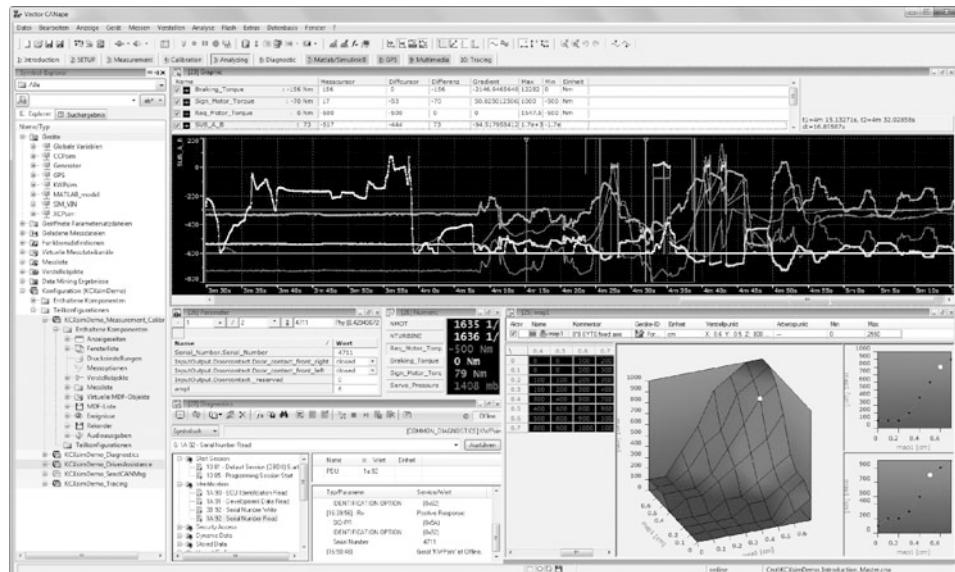


Abb. 9.14 Applikationswerkzeug *CANape* von Vector Informatik

erstellt und modifiziert. Weitere Editoren ermöglichen den Umgang mit DBC-, FIBEX-, LDF- und ODX-Dateien.

Für eine optimale Parametrierung der Steuergeräte werden Daten von verschiedenen Quellen erfasst und aufgezeichnet. Dazu benötigt der Entwickler ein zuverlässiges System, das synchron und zeitgenau arbeitet. *CANape* erlaubt den Zugriff auf:

- Variablen, Parameter, Kennlinien und Kennfelder im Steuergerät,
- Bus-Informationen von CAN, LIN und FlexRay,
- analoge und digitale Signale aus unterschiedlichen externen Messsystemen,
- aufgezeichnete Video- und Audiodaten,
- GPS-Informationen.

CANape kann Steuergeräteparameter sowohl im laufenden Betrieb im Speicher des Steuergerätes als auch offline im Flash-Programmierdatensatz verstellen. Automatisierte Verstolloperationen lassen sich per Skript realisieren. Das in *CANape* integrierte Kalibrierdatenmanagement kann die Parameterdatensätze nicht nur verstellen, sondern erlaubt auch eine leistungsfähige Verwaltung von Versionen und Varianten, wie sie im Laufe eines Entwicklungsprojektes entstehen.

Die Parameter werden in symbolischen, adressunabhängigen Datensätzen gespeichert. Die Verarbeitung ist dadurch unabhängig vom Steuergeräte-Programmstand, mit dem sie erzeugt wurden. Messdaten werden im MDF-Format abgespeichert. Weitere Formate wie ASCII oder Matlab sind über Konverter möglich. MDF ist ein weit verbreiteter Industriestandard, der mittlerweile durch ASAM gepflegt wird (Kap. 6). Die Visualisierung und Analyse der Messdaten erfolgt in vielfältigen Darstellungsfenstern. Data-Mining-Funktionen erlauben das Auswerten und Durchsuchen großer Datenmengen.

Bei der Entwicklung von Steuergeräten spielen die Diagnosefunktionen und der Diagnosetest eine bedeutende Rolle. *CANape* erlaubt den symbolischen Zugriff auf alle Steuergerätedaten und -funktionen, die über Diagnoseprotokolle zugänglich sind. Ein integrierter Viewer zeigt die Diagnosebeschreibungsdateien an und erlaubt den Zugriff auf die OBD-Daten.

Das während der Steuergeräte-Applikation notwendige Flashen neuer Programmversionen erfolgt über CCP/XCP oder eines der Diagnoseprotokolle mit oder ohne ODX-Flash-Container.

Im Rahmen der modellbasierten Software-Entwicklung werden Funktionen in einem iterativen Prozess zuerst in einer PC-Simulation konzipiert und dann in einer Bypass-Umgebung oder direkt im Steuergerät im Echtzeitbetrieb erprobt. Als Simulationswerkzeug dient häufig *Matlab* mit den Erweiterungen *Simulink*, *Stateflow* und der automatischen Codegenerierung durch den *Real Time Workshop*. Als Ablaufumgebung für die neu entwickelten Funktionen ist während der Entwicklungsphase oft der Einsatz von Standard-PCs anstelle kostenintensiver Rapid-Prototyping-Hardware ausreichend. *CANape* kann auch dann über XCP auf alle modellinternen Messgrößen und Parameter zugreifen, wenn die Funktionen nicht im Steuergerät, sondern auf dem PC ablaufen. Bei

Tab. 9.1 Vergleich der verschiedenen Schnittstellen zur Messdatenerfassung

ECU Interface	ECU Software-Änderung	ECU RAM-Bedarf	Maximale Messdatenrate	ECU Laufzeit-Einfluss	Bypass-Latenzzeit
CCP/XCP on CAN	CCP/XCP-Treiber	1 ... 2 KB	50 KB/s	Mittel	Hoch
XCP on FlexRay	XCP-Treiber Software	2 ... 16 KB	50 ... 400 KB/s	Groß	Mittel
XCP on JTAG/SPI	Tabellen für DAQ-Transfer	4 ... 16 KB	200 ... 400 KB/s	Groß	Mittel
Daten-Trace VX1000	Gering	Keiner	5000 KB/s	sehr gering	Gering

höheren Anforderungen kann aber auch eine Rapid-Prototyping-Hardware eingesetzt werden. Mit Hilfe des *Simulink Model Explorers* visualisiert *CANape* die *Simulink*- und *Stateflow*-Modelle. Durch Kopplung zwischen Modell und ASAP2/A2L-Beschreibungsdatei können alle Parameter und Messwerte unabhängig von der Ablaufumgebung verstellt und aufgezeichnet werden.

Beim Steuergerätezugriff über die standardisierten Mess- und Kalibrierprotokolle CCP oder XCP on CAN, FlexRay, JTAG oder SPI ist ein im Steuergerät integrierter Treiber für das zyklische Senden und Empfangen der gewünschten Signalwerte verantwortlich. Entsprechend der zu messenden Datenmenge benötigt der Treiber Hauptspeicher- und Rechenzeit, die aber nur im begrenzten Maße zur Verfügung stehen. Zusätzlich entsteht eine erhöhte Buslast, die sich negativ auf die Steuergerätekommunikation auswirken kann. Die möglichen Messdatenraten reichen dabei von 50 KB/s bei CAN bis hin zu Werten von 400 KB/s bei FlexRay, JTAG und SPI (Tab. 9.1). Bei Fahrerassistenzsystemen, Motorsteuergeräten oder bei der Entwicklung von Hybrid- oder Elektroantrieben werden inzwischen aber Messdatendurchsätze von 5 MByte/s und Messraster von unter 20 µs bei minimalem Einfluss auf die Laufzeiten des Steuergeräts gefordert. Derartige Anforderungen lassen sich einhalten, wenn die Ankopplung nicht über die Standard-Bussysteme, sondern über die mikrocontroller-spezifischen Daten-Trace- und Debug-Schnittstellen erfolgt, wie dies beispielsweise mit dem VX1000 Mess- und Kalibriersystem von Vector möglich ist. Das Mess- und Kalibriersystem selbst wird dann über XCP on Ethernet mit dem PC verbunden.

9.4 Flash-Programmierung von Steuergeräten

Programme und Daten von Steuergeräten werden heute in der Regel in Flash-ROM-Speichern abgelegt (Tab. 9.2). Speicher dieses Typs lassen sich im Gegensatz zu ROM-Speichern (Read Only Memory) oder den lange Zeit üblichen EPROM-Speichern (Erasable ROM) im eingebauten Zustand relativ einfach löschen und neu programmieren. Im Ver-

Tab. 9.2 Überblick HalbleiterSpeicher für Steuergeräte

Speichertyp	Programmieren	Löschen	Eignung, Sonstige Eigenschaften
ROM Read Only Memory	Beim IC-Hersteller	Nein	Feste Programme und Daten Geringe Kosten je Bit Vorlaufzeiten > 1 Monat Hohe Mindeststückzahlen
EPROM Erasable ROM	Im ausgebauten Zustand beim Gerätehersteller		Feste Programme und Daten Mittlere Kosten
Flash-ROM	Im Gerät blockweise (4...64 KB), einige 100.000-mal		Selten veränderliche Programme und Daten Mittlere Kosten
EEPROM Electrical Erasable ROM	Im Gerät byteweise einige 100.000-mal		Veränderliche Daten, z. B. Betriebs- stundenzähler Sehr hohe Kosten
RAM Random Access Memory	Nicht erforderlich		Variable Daten Dateninhalt geht beim Abschalten verloren

gleich zu EEPROM-Speichern (Electrical Erasable ROM) sind sie zwar nur blockweise lösch- und programmierbar, so dass der Normalbetrieb des Steuergerätes dazu unterbrochen werden muss, doch sind sie wesentlich billiger und zuverlässiger. EEPROMs werden daher praktisch ausschließlich für das dauerhafte Speichern von wenigen, aber häufiger veränderlichen Daten wie Kilometerständen, Betriebsstundenzählern oder Fehlerspeicherreinträgen verwendet. Bei aktuellen Mikrocontrollern werden EEPROMS häufig durch ein Flash-ROM emuliert. Dadurch kann das zusätzliche EEPROM eingespart werden.

Der Lösch- und anschließende Programmervorgang eines Flash-Speichers wird in diesem Buch als *Flashen* bezeichnet.

Die Flash-Funktion wird in der Regel in einer eigenständigen Software-Komponente im Steuergerät gekapselt und der übrigen Software über standardisierte Schnittstellen zur Verfügung gestellt (vgl. Abschn. 7.1 und 7.5). Für diese Softwarekomponente hat sich der Begriff *Flash-Lader* (Flash Loader), gelegentlich auch *Boot-Lader* etabliert.

9.4.1 Rahmenbedingungen

Der konsequente Einsatz der Flash-ROM-Technologie bietet im gesamten Produktlebenszyklus von der Entwicklung und Applikation über die Produktion bis hin zum Service und der Produktbetreuung (After Sales Market) viele Vorteile. Besonders deutlich wird der Nutzen in der Produktion und im Service. Obwohl in der Entwicklungs- und Applikationsphase schon immer Programmcode und Applikationsdatensatz getrennt gehalten wurden, um eine einfachere Änderbarkeit der Daten zu erreichen, wurden beide in der Vergangenheit spätestens während der Fertigung des Steuergerätes beim Geräteherstel-

ler zusammengeführt, in einen EPROM-Speicherbaustein programmiert und eingebaut. Beim Fahrzeughersteller wurden anschließend nur noch einige wenige Einstellparameter im EEPROM-Speicher angepasst. Unterschiedliche Modell- und Ländervarianten mussten entweder durch eine Variantencodierung im Steuergerät abgedeckt werden, was zu einem deutlich erhöhten Speicherplatzbedarf führt, oder wurden gleich durch Gerätevarianten abgedeckt. Selbst bei durchschnittlichen Fahrzeugbaureihen kam es so schnell zu Dutzenden von Gerätevarianten, die sich bei praktisch baugleicher Hardware lediglich in den unterschiedlichen Speicherinhalten unterschieden. Da eine nachträgliche Änderung des Speicherinhalts technisch aufwendig und daher wirtschaftlich praktisch unmöglich war, führte dies zu einem extrem hohen logistischen Aufwand vom Geräte-, über den Fahrzeughersteller bis zu den Werkstätten. Durch die Flash-ROM-Technologie kann die Hardware zumindest innerhalb einer Motoren- und Fahrzeugbaureihe, zunehmend sogar übergreifend, sehr viel besser standardisiert werden. Die Anpassung an das konkrete Modell oder gar einzelne Fahrzeug wird zu einem beliebigen späteren Zeitpunkt im Fertigungsprozess durchgeführt und kann selbst danach jederzeit wieder geändert werden. Durch die einfachere Gerätelogistik und die erheblich verkürzten Rüstzeiten bei Produktionsumstellungen ergeben sich deutliche Kostenvorteile. Gleichzeitig muss der Fahrzeughersteller in seiner Logistikkette bis zur Werkstatt allerdings sicherstellen, dass die Softwarestände in sämtlichen Steuergeräten zum einzelnen Fahrzeug passt.

Ob und welche Teile eines Steuergerätes bereits beim Zulieferer innerhalb von dessen Produktion oder erst beim Automobilhersteller am Band programmiert werden, wird oft intensiv diskutiert und von den verschiedenen Herstellern sehr unterschiedlich gehandhabt (Tab. 9.3). Der Trend geht aber zur Programmierung am Band des Fahrzeugherstellers.

Im Service und im After-Sales-Markt wird durch die Flash-Technologie die Möglichkeit geschaffen, Kosten einzusparen. Im Feld festgestellte Schwachstellen eines Fahrzeugs, die früher zu einem Austausch von Fahrzeugkomponenten geführt haben, können heute durch eine Update-Programmierung vor Ort kostengünstiger behoben werden. Darüber hinaus besteht die Möglichkeit, im Rahmen eines normalen Kundendienstes ältere Fahrzeuge auf den neuesten Stand zu aktualisieren, Funktionen nachzurüsten oder in begrenztem Umfang sogar an geänderte gesetzliche oder versicherungstechnische Vorschriften, etwa Abgasgrenzwerte, anzupassen.

Während in der Steuergerätefertigung noch spezielle, schnelle Schnittstellen für die Flash-Programmierung eingesetzt werden können (Tab. 9.4), kann der Zugang im Fahrzeug nur noch über den verhältnismäßig langsamen Diagnoseanschluss erfolgen. In der Regel ist dies das zentrale Gateway-Steuergerät (vgl. Abb. 1.1 und 9.15) mit den aus Kap. 4 und 5 bekannten Diagnoseprotokollen, z. B. KWP 2000, UDS oder SAE J1939. Von dort wird auf alle primären Bussysteme im Fahrzeug verzweigt. Vereinzelt erfolgt der Zugang ins Fahrzeug noch über K-Line, die Regel ist allerdings CAN. Übertragungsraten und Datenmengen differieren je nach Bussystem, eingestellter Baudrate und verwendetem Transportprotokoll. Da die Busübertragungszeiten neben den eigentlichen Programmierzeiten des Flash-Speichers den Durchsatz am Band des Fahrzeugherstellers nennenswert beeinflussen und die Datenmengen stetig steigen, führen erste Fahrzeugh-

Tab. 9.3 Typische Stationen der Flash-Programmierung

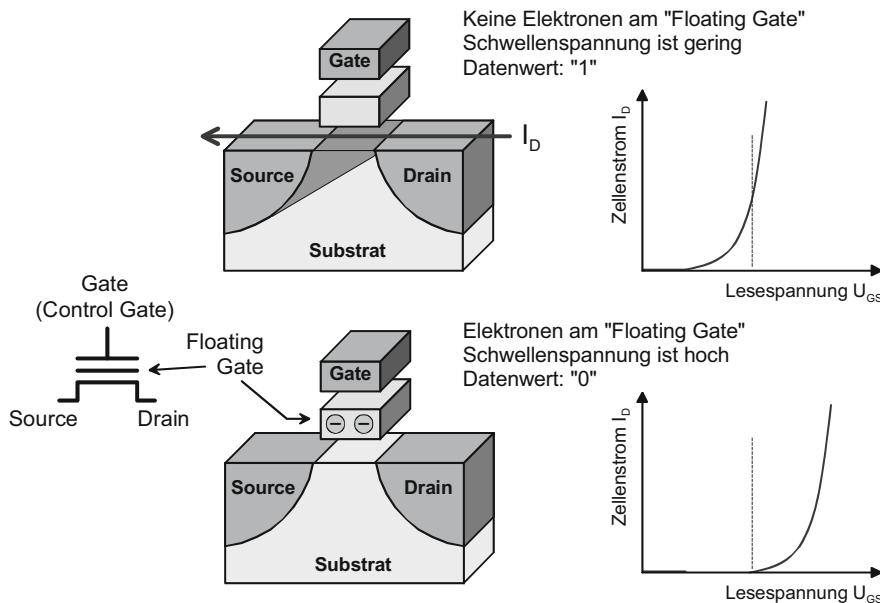
Wo wird programmiert?	Wer ist verantwortlich?	Was wird programmiert?	Erläuterung
Programmierstation	Steuergeräte-Hersteller	Anwendungssoftware inklusive Betriebssystem und Diagnoseprotokoll	Schnelle, parallele Programmierung der Flash-Speicher im nicht eingebauten Zustand über spezielle Programmiergeräte mit definierten und protokollierten Umgebungsbedingungen (Temperatur, Versorgungsspannung)
Steuergeräte-Fertigung (End of Line EOL)	Steuergeräte-Hersteller	Modellspezifische Applikationsdatensätze (z. B. Anpassung an 4 oder 6 Zylinder-Motor)	Bandende-Bedatung, z. B. Abgleichdaten; Diversifikation der Steuergeräte durch Softwareapplikation, Hardware im Gleichteilekonzept
Fahrzeug-Fertigung (EOL)	Fahrzeughersteller	Fahrzeug-Konfigurationsdaten	Freischaltung von Funktionen für das spezifische Fahrzeug und Feinjustierung
Werkstatt (Service)	Fahrzeug- und/oder Steuergeräte-Hersteller	Neue Anwendungssoftware, Fahrzeugkonfiguration	Fehlerbeseitigung, Kompensation von Alterung und Verschleiß, Nachrüstung neuer Funktionen

steller für den Programmervorgang parallel zur CAN-Diagnoseschnittstelle das schnellere Ethernet/DoIP-Interface ein.

Damit immer die richtigen Daten im passenden Steuergerät landen und dort dauerhaft gespeichert bleiben, ist eine stringente Datenhaltung und Pflege notwendig (Tab. 9.5). Dies bedeutet zunächst eine transparente Software-Versionskontrolle beim Automobilhersteller und die eindeutige Identifikation des Steuergerätes in der Werkstatt. Mit diesen Informationen lässt sich verhindern, dass nicht freigegebene Software-Stände ins Fahrzeug oder falsche Software in ein Steuergerät gelangen. Trotz aller logistischen Maßnahmen müssen aber Kontrollmechanismen im Steuergerät selbst, speziell im *Flash-Lader* implementiert sein. Dazu zählen der Zugriffsschutz des Steuergerätes vor unbefugten Dritten, die Überprüfungen der Hardware-Software-Kompatibilität, die Speicherbereichsüberwachung vor einer Programmierung und die Verifikation neu programmierte Software. Erkennt der *Flash-Lader* einen unzulässigen Zustand, wird die aktuelle Programmierung abgebrochen und muss komplett neu aufgesetzt werden. Somit wird verhindert, dass ein Steuergerät nach einem Fehler bei der Programmierung die fehlerhafte Software trotzdem zu starten versucht und danach keine Kommunikation mehr möglich ist. In diesem Fall müsste das Steuergerät getauscht werden und die Vorteile der Flash-Technologie blieben ungenutzt.

Tab. 9.4 Kenndaten für die Flash-Programmierung

Flash-Station	Programmierzugang	Typ. Datenmenge	Typ. Programmierdauer
Programmierstation	JTAG-Schnittstelle mit bis zu 10 MBit/s Datenrate	Body: 256 KB Powertrain: 1 ... 4 MB Multimedia: 5 ... 20 MB	30 sec ... 5 min
Steuergeräte-Fertigung	JTAG, zunehmend CAN	1 ... 64 KB (EOL)	ca. 10 sec
Fahrzeug-Fertigung (EOL)	Diagnoseschnittstelle (K-Line, CAN)	1 ... 64 KB (EOL)	ca. 10 sec
Werkstatt-Tester	Diagnoseschnittstelle (K-Line, CAN)	Neue Anwendungssoftware wie Programmierstation Neue Fahrzeugkonfiguration wie EOL	5 ... 10 min je Gerät, komplettes Fahrzeug im Stundenbereich Ziel: max. 15 min für Fahrzeug-Update

**Abb. 9.15** Aufbau einer Flash-Speicherzelle

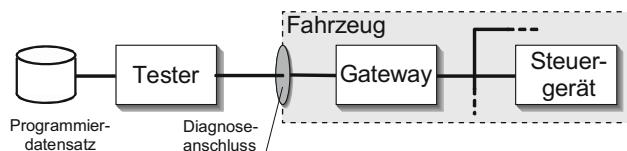
9.4.2 Flash-Speicher

Flash-Speicher nutzen MOS-Feldeffekttransistoren mit *Floating Gate*. Dabei werden Elektronen in einer isolierten Schicht zwischen dem Gate und dem Source-Drain-Kanal eingeschlossen, dem *Floating Gate*. Die kleinste Einheit eines Flash-Speichers ist die Speicherzelle. Sie besteht in der Regel aus einem einzigen Transistor (Abb. 9.16).

Tab. 9.5 Wichtige Anforderungen an die Flash-Programmierung

Programmieralgorithmus	Um eine sichere Datenhaltung von mindestens 10 bis 15 Jahren im Flash-ROM-Speicher sicherzustellen, müssen die Vorgaben des Flash-ROM-Herstellers (Versorgungsspannung, Temperatur, Zeitverlauf der Programmierbefehle/-signale) eingehalten werden.
Kompatibilitätsprüfung der Programmierdaten	Überprüfung, ob der neue Datensatz tatsächlich zur Hardware des Steuergerätes sowie zur restlichen Software passt, z. B. mit Hilfe des UDS-Dienstes <i>Routine Control - Check Programming Dependencies</i> .
Zugangsschutz	Sicherstellen, dass das Steuergerät nur von autorisierten Stellen programmiert werden kann, indem der Zugang zum Steuergerät und Start der Programmierroutinen nur nach einem Login-Vorgang z. B. mit einem Seed- und Key-Mechanismus freigeschaltet wird. Gegebenenfalls Ausschalten von <i>Hintertüren</i> , z. B. Sperren des Zugriffs über JTAG, nach erfolgreichem Programmervorgang beim Gerätehersteller. Verwendung von Signaturen und Zertifikaten zur Prüfung, ob der Programmierdatensatz manipuliert wurde.
Robustheit gegen Fehlprogrammierung	Der <i>Flash-Lader</i> muss auch dann ansprechbar sein, wenn während des Programmervorgangs ein Fehler auftritt und der Programmervorgang abgebrochen wird. Dazu muss der <i>Flash-Lader</i> selbst gegen unbeabsichtigtes Überschreiben gesichert werden, die Konsistenz der programmierten Daten bei jedem Einschalten des Geräts überprüft werden. Nur bei <i>korrekten</i> Daten wird der normale Betrieb (<i>Fahrfunktion</i>) des Gerätes aufgenommen, ansonsten verbleibt die Software im <i>Flash-Lader</i> .

Durch die eingelagerten Elektronen verschiebt sich die Schwellspannung des Transistors (*Threshold Voltage*), d. h. die Spannung, die am Gate des Transistors mindestens angelegt werden muss, damit der Transistor durchgeschaltet wird. Beim Auslesen des Speicherinhalts wird an dieses Gate eine positive Spannung gelegt (Lesespannung), die größer ist als die Schwellspannung bei ungeladenem Floating Gate. Bei Transistoren, die eine logische 1 speichern sollen, befinden sich auf dem Floating Gate keine Elektronen, so dass der Transistor beim Lesen leitend wird. Bei Transistoren, die eine logische 0 speichern sollen, wird das Floating Gate während des Programmervorgangs so mit Elektronen geladen und dadurch die Schwellspannung soweit erhöht ist, dass die Transistoren beim Lesen nicht leitend sind.

Abb. 9.16 Hauptkomponenten des Flash-Programmierprozesses

tend werden. Das Entladen und Laden des Floating-Gates beim Lösch- und Programmiervorgang erfolgt mit Hilfe von Avalanche- und Quanteneffekten wie dem Fowler-Nordheim-Tunneleffekt, die die Halbleiterstrukturen stark beanspruchen und die Isolationsschichten schädigen können. Daher dürfen die Lösch- und Programmervorgänge nicht beliebig oft durchgeführt werden, sondern typischerweise nur einige 100.000 Mal. Andernfalls wird die Zuverlässigkeit der Speicherzelle beeinträchtigt. Wie bei allen Halbleitern sinkt die Zuverlässigkeit annähernd exponentiell, wenn das Bauteil häufig bei hohen Temperaturen betrieben wird.

Für das Laden und Entladen der Floating Gates während des Programmervorgangs sind hohe Spannungen notwendig, die mit Hilfe von Ladungspumpen innerhalb des Flash-ROMs aus der normalen Versorgungsspannung erzeugt und mithilfe zusätzlicher Schaltungsstrukturen an die Speicherzellen angelegt werden. Um den hierfür notwendigen Aufwand zu begrenzen, lassen sich nicht einzelne Speicherzellen sondern nur ganze, relativ große Speicherblöcke, sogenannte Sektoren, am Stück löschen, beispielsweise Blöcke von 64 KB bei einem insgesamt 2 MB großen Speicher. Der Lösch- und Programmervorgang ist relativ langsam. Während das Auslesen eines Speicherwortes nur wenige 10 ns dauert, benötigt das Löschen eines Speicherblocks im Bereich von einigen hundert Millisekunden bis zu einigen Sekunden. Das Programmieren eines einzelnen Bytes dauert einige 10 μ s.

Der Lösch- und Programmervorgang wird über den normalen Daten- und Adressbus mit den üblichen Schreib- und Lese-Steuersignalen eingeleitet. Zum Schutz gegen unbeabsichtigtes Löschen sind bestimmte Steuerkommandofolgen mit vorgeschriebenem Zeitablauf einzuhalten. Da der *Programmieralgorithmus* hersteller- und typabhängig ist, wird er innerhalb des *Flash-Laders* üblicherweise in einer als *Flash-Treiber* bezeichneten Komponente mit definierter Schnittstelle gekapselt.

Während des Lösch- und Programmervorgangs ist bei vielen Bausteinen nicht nur der betroffene Speicherblock sondern ein großer Teil oder sogar der gesamte Baustein nicht mehr zugänglich, so dass der Programmieralgorithmus aus einem anderen Speicherbaustein heraus ausgeführt werden muss. Der Programmieralgorithmus wird daher oft in das RAM des Steuergerätes kopiert und von dort ausgeführt. Um sicherzustellen, dass der Programmervorgang nicht durch Unbefugte oder aufgrund einer Fehlfunktion der Steuergerätesoftware unbeabsichtigt ausgeführt wird, sichert man den Algorithmus durch bestimmte Maßnahmen oder lädt ihn ausschließlich für den Flash-Vorgang über die Diagnoseschnittstelle von außen in das Steuergerät. Bei neueren Bausteinen kann außerdem häufig einer der Speicherblöcke, gelegentlich als *Boot Block* bezeichnet, nach dem erstmaligen Programmieren dauerhaft gegen Löschen und Neuprogrammieren geschützt werden. Dieser Block eignet sich ganz besonders gut für die Boot-Software des Steuergerätes, d. h. denjenigen Programmteil, der beim Einschalten des Gerätes ausgeführt wird und die Grundroutinen zur Initialisierung des Rechnerkerns, den Test des Speichers und gegebenenfalls die Kernsoftware für den Flash-Programmervorgang enthält.

9.4.3 Flash-Programmierprozess

Am Flash-Programmierprozess im Fahrzeug sind mehrere Komponenten beteiligt (Abb. 9.16). Der Tester steuert den gesamten Flash-Programmierprozess. Er hält die zu programmierenden Daten vor und ist über den Diagnoseanschluss und gegebenenfalls ein Gateway mit dem internen Bussystem des Fahrzeugs und dem daran angeschlossenen Steuergerät verbunden, das zu programmieren ist.

Für das grundsätzliche Verständnis des Flash-Prozesses ist es notwendig, das Steuergerät sowohl aus Sicht der Hardware als auch der Software zu betrachten. Wie oben bereits beschrieben, muss zur Programmierung des Flash-Speichers ein *Flash-Lader* existieren, der diesen Vorgang durchführt. Der *Flash-Lader* teilt sich den vorhandenen Steuergerätespeicher mit der übrigen Steuergerätesoftware, im Weiteren als Anwendungssoftware oder Applikation bezeichnet.

Start des Flash-Laders nach HIS Die in Abb. 9.17 dargestellte Grundstruktur basiert auf der HIS-Spezifikation, die bereits in Abschn. 7.5 kurz beschrieben wurde. Ein Steuergerät besteht demnach aus den drei Softwareblöcken *Boot-Manager*, *Flash-Lader* und *Anwendungssoftware*:

- Unmittelbar nach einem Einschalten des Steuergerätes (*Reset*) verzweigt das Steuergerätesprogramm zunächst in den *Boot-Manager*. Dieser prüft, ob sich bereits eine verifizierte Anwendungssoftware im Steuergerätespeicher befindet.
- Ist keine gültige Anwendungssoftware programmiert oder ist der Programm- oder Datenspeicher in einem inkonsistenten Zustand, etwa weil ein früherer Programmierversuch vorzeitig abgebrochen wurde, so wechselt das Steuergerät nach dem Einschalten in den *Flash-Lader* und wartet auf Programmierbefehle des Testers. Damit der *Flash-Lader* über das Bussystem des Fahrzeugs mit dem Tester kommunizieren kann, muss er einen Diagnoseprotokollstapel mit der für den Programmervorgang notwendigen Untermenge von Botschaften enthalten.
- Ist eine gültige Anwendungssoftware programmiert, so prüft der *Boot-Manager* in einem zweiten Schritt, ob ein Wechsel aus der Anwendungssoftware in den *Flash-Lader* initiiert wurde (siehe unten). Andernfalls verzweigt das System in die Anwendungssoftware und das Steuergerät führt seine regulären Aufgaben für den Fahrbetrieb, z. B. die Motorsteuerung, aus.
- In der Anwendungssoftware kann eine Diagnoseanforderung *Start Diagnostic Session* bzw. *Diagnostic Session Control – Start Programming Session* (siehe Abschn. 5.1.2 und 5.2.2) jederzeit den Wechsel in den *Flash-Lader* initiieren. Die Anwendungssoftware speichert dabei die Anforderungsinformation für den Wechsel und löst einen Reset aus, der dann zum Start des *Flash-Laders* führt.
- In der Regel ist während des Löschens oder Programmierens eines Flash-Speicherblocks auch kein Zugriff auf die anderen Blöcke möglich. Daher werden die Routines zum Löschen und Programmieren des Flash-Speichers in das RAM des Steuergerätes kopiert

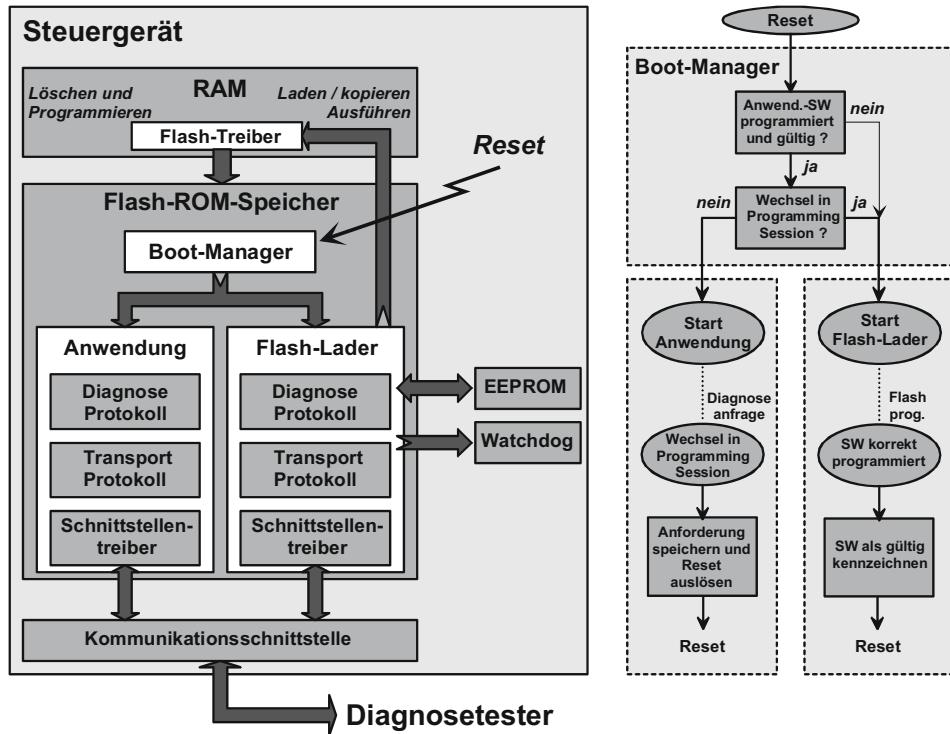


Abb. 9.17 Grundstruktur der Steuergerätesoftware für die Flash-Programmierung

und von dort ausgeführt. Umgangssprachlich hat sich dafür die Bezeichnung „die Flash-Funktionen laufen im RAM“ eingeprägt. Die Verzweigung zu der Routine im RAM wird als *Einsprung in die Flash-Routinen* bezeichnet.

- Nach einer erfolgreichen und korrekten Programmierung der neuen Anwendungssoftware in den Flash-Speicher kennzeichnet der *Flash-Lader* diese als gültig. Im Anschluss wird ein Software-Reset ausgelöst. Der *Boot-Manager* erkennt die gültige Anwendungssoftware und wechselt in die neue Software.

Flash-Ablauf nach HIS Als Flash-Ablauf oder Programmiersequenz wird die Aneinanderreihung von speziellen Diagnosediensten bezeichnet. Innerhalb der Diagnoseprotokolle (KWP 2000, UDS usw.) sind für die Programmierung spezielle Dienste spezifiziert, wie in Kap. 5 beschrieben. Abstrakt betrachtet ist der Flash-Ablauf für Steuergeräte immer gleich. Tatsächlich unterscheiden sich die einzelnen Fahrzeughersteller und Zulieferer im Bezug auf die verwendeten Diagnosedienste bisher aber erheblich. Innerhalb HIS wurde versucht, die Abläufe zu vereinheitlichen, wobei immer noch einige Schritte optional sind, so dass dennoch unterschiedliche Varianten möglich bleiben. Der von HIS vorgeschlagene Flash-

Ablauf ist in Tab. 9.6 dargestellt, wobei HIS davon ausgeht, dass UDS-Diagnosedienste verwendet werden.

Wie eingangs dargestellt, erfolgt die Flash-Programmierung häufig an unterschiedlichen Stellen, z. B. beim Gerätehersteller, beim Fahrzeugherrsteller oder in der Werkstatt. Dabei unterscheiden sich die zu programmierenden Datenmengen, die Geschwindigkeitsanforderungen und die Sicherheitsbedürfnisse erheblich, so dass es sinnvoll und notwendig ist, mehrere unterschiedliche Flash-Abläufe im selben Gerät zu unterstützen. Dazu werden die unterstützten Diagnosedienste und deren Abarbeitungsreihenfolge als autarke Abläufe im *Flash-Lader* definiert und in einer Diagnosesitzung gekapselt, die durch den entsprechenden Diagnose-Protokolldienst, bei UDS beispielsweise *Session Control*, aufgerufen wird. Ein Beispiel für die Notwendigkeit unterschiedlicher Diagnosesitzungen ist das Auslesen der Steuergeräteidentifikation. Während in der Gerätefertigung vor allem die Identifikationsdaten des Geräteherstellers ausgelesen werden, soll die Werkstatt primär die Identifikationsdaten des Fahrzeugherrstellers abfragen und unter Umständen die internen Informationen des Geräteherstellers gar nicht sehen. Dies kann durch unterschiedliche Diagnosesitzungen und/oder unterschiedliche Diagnosedienste *Read Data by Identifier* für die einzelnen Informationen erreicht werden.

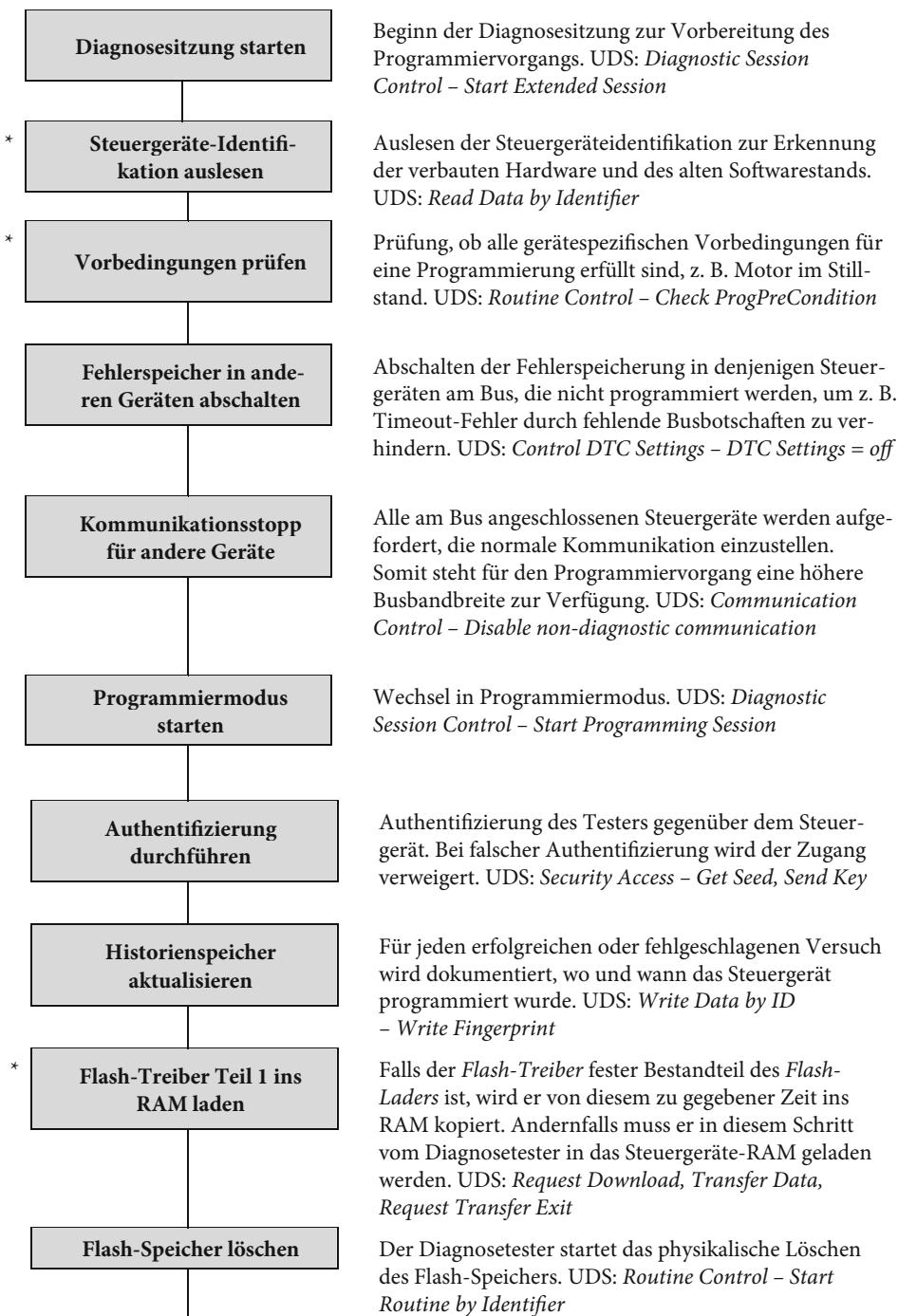
Authentifizierung und Validierung Zum Schutz eines Steuergerätes vor Manipulation werden verschiedene Maßnahmen getroffen. Dabei unterscheidet man zwischen *Authentifizierung* und *Validierung*.

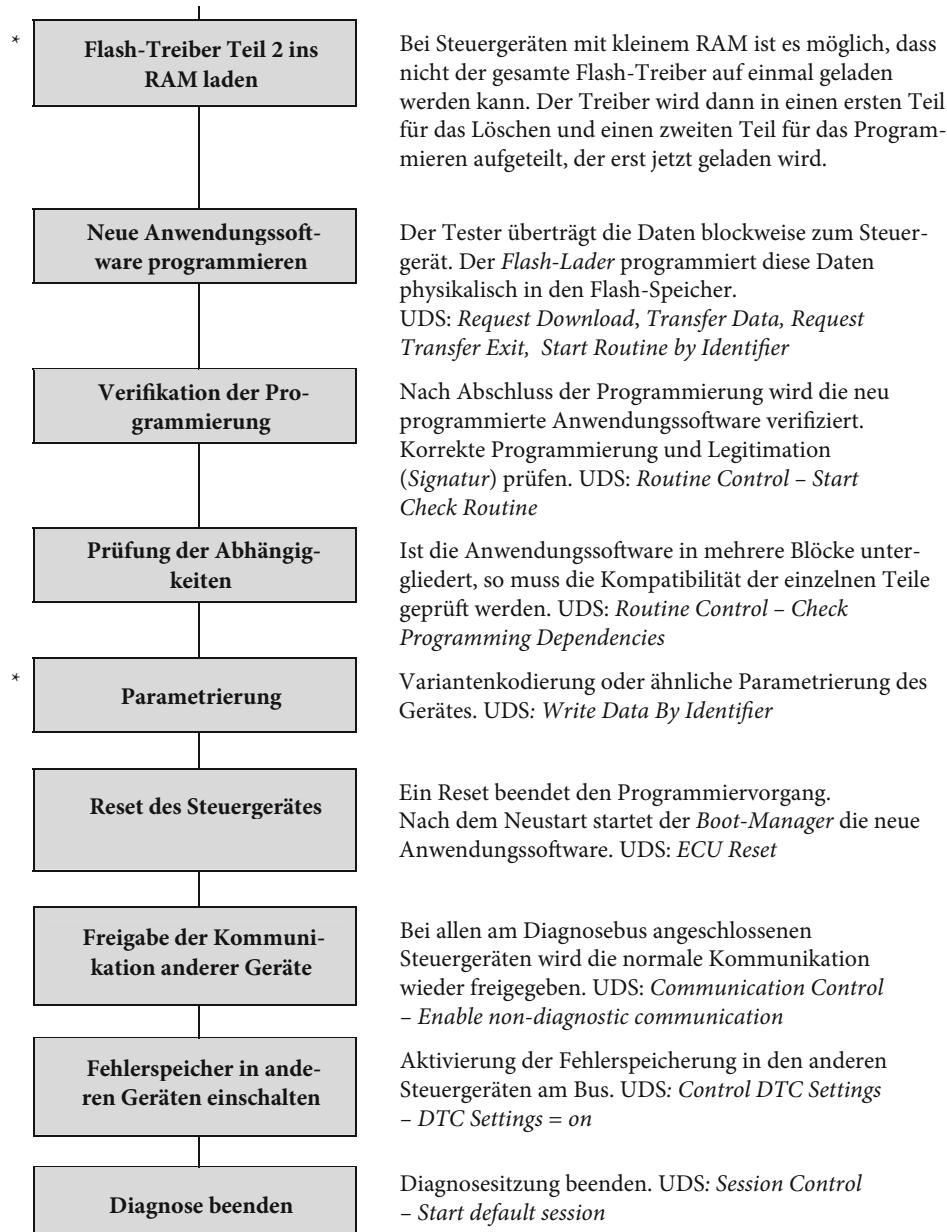
Um einen Zugriff auf das Steuergerät durch Unbefugte zu verhindern, muss sich der Tester zunächst gegenüber dem Steuergerät *authentifizieren*. Dabei fordert der Tester vom Steuergerät eine Zufallszahl als Initialisierungswert (*Seed*) an, berechnet daraus einen Schlüsselwert (*Key*) und sendet diesen zurück zum Steuergerät. Der geheime Algorithmus zur Schlüsselberechnung ist sowohl dem Tester als auch dem Steuergerät bekannt. Somit kann das Steuergerät ebenfalls aus dem *Seed*- den *Key*-Wert berechnen und diesen mit dem Wert des Testers vergleichen. Sind die jeweiligen Werte unterschiedlich, verweigert das Steuergerät dem Tester den weiteren Zugang. Eine Programmierung ist dann nicht möglich.

Die Berechnung von *Seed* und *Key* kann beliebig komplexe mathematische Algorithmen verwenden. Allerdings bietet dieses Verfahren keinen vollkommenen Schutz vor unbefugtem Zugriff, da der Berechnungsalgorithmus in jedem Werkstatttester vorhanden sein muss und somit relativ einfach zugänglich ist. Es stellt lediglich eine erste Hürde dar, die durch geeignete Zusatzmethoden, z. B. den Einbau zunehmender Wartezeiten bei Fehlversuchen den Angriff auf ein Steuergerät verzögern kann.

Den weitaus wichtigeren Anteil zum Verhindern von Manipulationen am Steuergerät bildet die *Validierung* der Software nach einer Programmierung. Dabei wird zum einen die fehlerfreie Programmierung der Software in den Flash-Speicher überprüft, zum anderen die Herkunft der Software verifiziert.

Dafür werden symmetrische oder asymmetrische kryptographische Verfahren verwendet (Abb. 9.18). Berücksichtigt man die Sicherheitsrisiken und den Aufwand bei der

Tab. 9.6 Flash-Programmierablauf nach HIS (* optionale Schritte)

Tab. 9.6 (Fortsetzung)

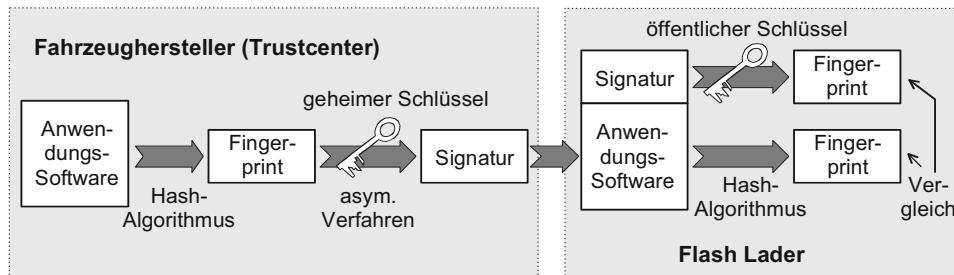


Abb. 9.18 Validierung einer Software

Verwaltung und Verteilung der notwendigen Schlüssel, so bietet die *asymmetrische* Verschlüsselung nach heutigem Stand die höchste Sicherheitsstufe. Bei der *asymmetrischen* Verschlüsselung wird beim Verschlüsseln ein *geheimer Schlüssel* und ein mathematischer Algorithmus, eine sogenannte Einwegfunktion, verwendet, die nicht umkehrbar ist. Die Entschlüsselung erfolgt durch eine weitere Funktion mithilfe eines *öffentliche Schlüssels*. Das *Schlüsselpaar* basiert in der Regel auf großen Primzahlen und muss immer paarweise verwendet werden. Mit Schlüsselkombinationen aus unterschiedlichen Schlüsselpaaren ist eine Entschlüsselung unmöglich. Die Schlüsselpaare werden gemeinsam von einer autorisierten Stelle, einem so genannten *Trustcenter*, erzeugt, verwaltet und den Anwendern zur Verfügung gestellt. Der zur Verschlüsselung verwendete geheime Schlüssel (*Private Key*) ist nur der Stelle bekannt, welche die Anwendungssoftware verschlüsselt. Der öffentliche Schlüssel (*Public Key*), der zur Entschlüsselung verwendet wird, kann beliebig verteilt werden.

Moderne Validierungsverfahren für E-Mail, z. B. Pretty Good Privacy PGP, arbeiten nach derselben Methode. Der Ablauf einer Validierung der Software für ein Steuergerät ist wie folgt:

- Von der Anwendungssoftware wird vom Fahrzeughersteller zunächst ein digitaler Fingerabdruck (*Fingerprint*) erstellt. Dies geschieht durch die Verwendung eines *Hash-Verfahrens*, durch das eine eindeutige, schwer zu fälschende Prüfsumme für die Anwendungssoftware berechnet wird. Der bekannteste Hash-Algorithmus ist SHA-1 (*Secure Hash Algorithm*), der aus einem Datensatz beliebiger Länge eine Prüfsumme fester Länge erzeugt, die sich bei kleinsten Modifikationen im Datensatz ändert, so dass Manipulationen an den Daten sofort erkannt werden können.
- Diese Prüfsumme wird vom Fahrzeughersteller mit seinem geheimen Schlüssel (*Private Key*) verschlüsselt und dann als *Signatur* der Software bezeichnet. Für die Verschlüsselung der Prüfsumme wird in der Regel der nach seinen Erfindern Rivest, Shamir und Adleman benannte *RSA*-Algorithmus mit Schlüssellängen von z. B. 1024 bit oder der auf kleinen Mikrocontrollern besonders effizient zu implementierende *ECC/ECDSA*-Algorithmus eingesetzt, der auf sogenannten elliptischen Kurven beruht.

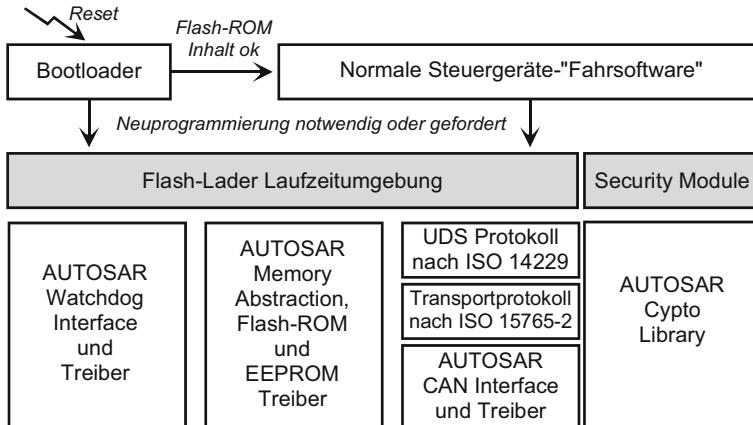


Abb. 9.19 Aufbau des HIS-Flash-Laders auf Basis von AUTOSAR-Komponenten

- Die Signatur wird zusammen mit der tatsächlichen Anwendungssoftware in das Steuergerät programmiert. Der *Flash-Lader* kennt außerdem den öffentlichen Schlüssel (*Public Key*) des Fahrzeugherstellers. Zur Validierung der Anwendungssoftware entschlüsselt der *Flash-Lader* die Signatur mit dem *Public Key* und erhält den *Fingerprint*. Danach berechnet er zum Vergleich mit demselben Hash-Algorithmus, den der Fahrzeughersteller verwendet hat, den *Fingerprint* direkt aus der eigentlichen Anwendungssoftware. Sind die beiden *Fingerprints* nicht identisch, d. h. schlägt die Validierung fehl, so wird diese Information im Steuergerät dauerhaft gespeichert und der Start der Anwendungssoftware verweigert.

Will man nicht nur sicherstellen, dass der Programmierdatensatz nicht manipuliert wurde und tatsächlich vom Fahrzeughersteller stammt, sondern das Auslesen des Datensatzes selbst durch Unbefugte verhindern, so kann man den Datensatz ebenfalls verschlüsseln und gegebenenfalls für die Übertragung zusätzlich komprimieren, um die Übertragungszeit zu verkürzen. Ein für die Verschlüsselung geeigneter Algorithmus ist etwa *AES*, der *Advanced Encryption Standard*.

Bei *Flash-Ladern* nach HIS-Spezifikation wird die oben beschriebene Funktionalität im sogenannten *Security Module* (Abb. 9.19) implementiert. Die Spezifikation dieses Moduls beschreibt die Funktionsaufrufe und Übergabeparameter. Steuergeräte werden dabei in verschiedene Sicherheitsklassen (*Security Classes*) eingeteilt. In der niedrigsten Stufe D, in der lediglich fehlerhafte Datensätze erkannt werden sollen, genügt eine 32 bit *Cyclic Redundancy Check* Prüfsumme. Für mittlere Sicherheitsklassen, in denen zusätzlich die Integrität und Authentizität geprüft werden soll, werden *HMAC* und *RSA 1024* vorgeschlagen, die deutlich aufwendiger sind. In der höchsten Stufe, in der zusätzlich ein Kopierschutz und die Vertraulichkeit der Daten erreicht werden soll, wird der komplexe *AES* Algorithmus gefordert.

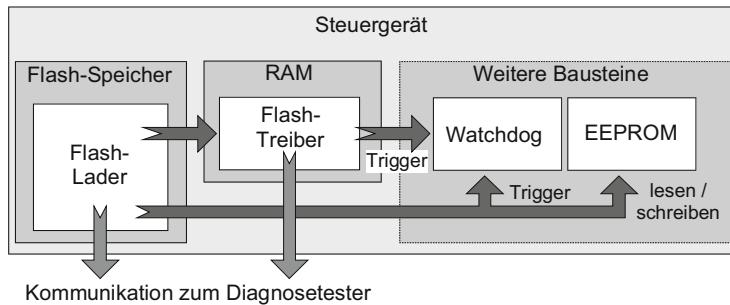


Abb. 9.20 Steuergerätekomponenten für die Flash-Programmierung

Flash-Treiber nach HIS und AUTOSAR Wie oben bereits beschrieben, ist es grundsätzlich möglich und inzwischen auch gängige Praxis, dass der *Flash-Programmieralgorithmus* erst zur Laufzeit vom Tester aus in das Steuergerät geladen wird. Daher ist es notwendig, dass die Kompatibilität des *Flash-Algorithmus* mit dem Steuergerät überprüft wird. So muss beispielsweise verhindert werden, dass der Treiber einer falschen Hardwarevariante geladen wird. Dabei wird zusätzlich davon ausgegangen, dass der *Flash-Algorithmus* nicht unbedingt vom Entwickler des *Flash-Laders* mitgeliefert wird, sondern auch vom Hersteller des jeweiligen Mikrocontrollers oder Flash-ROM-Bausteins bereitgestellt werden kann. HIS hat den Aufbau eines derartigen *Flash-Treibers* bereits 2002 spezifiziert, während der Vorschlag für den *Flash-Lader* selbst erst 2006 publiziert wurde. Der weiterentwickelte *HIS Flash-Lader-Standard* (Abb. 9.19) schreibt die Verwendung eines AUTOSAR-kompatiblen *Flash-Treibers* statt des proprietären *HIS-Flash-Treibers* vor und sieht für den Zugriff auf Hardwarekomponenten wie den *Watchdog* oder den Kommunikationscontroller AUTOSAR-Treiberschnittstellen vor (siehe Abschn. 7.6).

Der *Flash-Lader* lädt bzw. kopiert den Programmieralgorithmus (*Flash-Treiber*) ins RAM (Abb. 9.20). Wie *Flash-Lader* und *Flash-Treiber* müssen auch die *Watchdog*-Routine und die für den Lösch- und Programmervorgang notwendigen Teile der Diagnose-Protokollsoftware in einem während des Programmervorgangs zugänglichen Speicherbereich stehen, gegebenenfalls also ebenfalls ins RAM kopiert werden. Die Größe des zur Verfügung stehenden RAMs ist dabei das begrenzende Element. Die HIS-Spezifikation sieht daher vor, den *Flash-Treiber* in zwei Schritten zu laden, wobei der erste Teilschritt das Löschen des Speichers übernimmt und im zweiten Teilschritt die Neuprogrammierung durchgeführt wird.

Der *Flash-Treiber* selbst muss den internen Aufbau des Flash-Speichers (Topologie) kennen. Dazu gehören Anzahl, Größe und Anordnung der einzelnen Sektoren und Speicherbänke. Außerdem muss die Befehlssequenz zur Freischaltung des Speichers für den Lösch- und Programmervorgang einschließlich der zeitlichen Abläufe bekannt sein. Während des Löschens des gesamten Flash-Speichers oder einzelner Sektoren, das bei manchen Bausteinen mehrere Sekunden dauern kann, muss die Kommunikation des Steuergeräts mit dem

Tester im Hintergrund aufrecht erhalten werden (z. B. UDS bzw. KWP 2000 Diagnosedienst *Tester Present*). Die Löschroutine wird daher von Zeit zu Zeit verlassen und wechselt vom *Flash-Treiber* in den *Flash-Lader* zurück, um die Kommunikation zum Tester zu bedienen. *Flash-Treiber*, die aus dem RAM heraus ausgeführt werden, können meist weder das Multitasking-Betriebssystem des Steuergeräts noch eigene *Interrupt-Service-Routinen* verwenden, da diese häufig im zu löschen Speicher-Sektor liegen und bei vielen Mikrocontrollern nicht verlagert werden können. Die Implementierung des Flash-Laders wird dadurch erschwert.

Die Identifikationsdaten des Steuergerätes werden meist in einem EEPROM abgelegt. Der *Flash-Lader* legt dort Informationen über den Programmierstatus (z. B. Programmier- und Validierungshistorie) ab. Der *Flash-Lader* muss daher auch einen Treiber für das Lesen und Schreiben des EEPROMs enthalten.

Die HIS-Forderung, AUTOSAR-Schnittstellen für die Hardwaredreiber zu verwenden, bedeutet nicht, dass die AUTOSAR-Treiber der „Fahrsoftware“ auch im Flash-Lader verwendet werden können. Da die „Fahrsoftware“ durch den Flash-Lader komplett austauschbar sein soll, muss der Flash-Lader völlig unabhängig von der Fahrsoftware sein. Außerdem wird man die AUTOSAR-Komponenten für den Einsatz im Flash-Lader mit Rücksicht auf den Speicherplatzbedarf so konfigurieren, dass nur die für das *Flaschen* unbedingt notwendige Funktionalität aktiviert wird.

9.4.4 Beispiel eines Flash-Laders: ADLATUS von SMART IN OVATION

Ähnlich wie Diagnose-Protokollstapel werden Flash-Lader zunehmend zu Standardkomponenten, die von verschiedenen Herstellern angeboten werden. Ein Beispiel hierfür ist der Flash-Lader *ADLATUS* der SMART IN OVATION GmbH.

Neben den vorstehend beschriebenen technischen Eigenschaften muss ein Flash-Lader eine Reihe weiterer Fähigkeiten aufweisen, um von Steuergeräte- und Fahrzeugherrstellern als Standardkomponente akzeptiert zu werden, die in möglichst vielen Steuergeräten über mehrere Modelle und Generationen hinweg zum Einsatz kommt:

- Der Flash-Lader inklusive der eingesetzten Diagnoseprotokoll-Software muss für alle Mikrocontroller-Plattformen und Bussysteme verfügbar sein, die bei dem Hersteller verwendet werden, die aktuellen Flash-ROM-Bausteine und Kommunikationscontroller unterstützen und leicht auf neue Umgebungen portierbar sein
- Der Flash-Ablauf muss für unterschiedliche Anforderungen konfigurierbar sein und verschiedene Abläufe für Gerätehersteller, Fahrzeugherrsteller und Werkstatt im selben Gerät unterstützen.
- Unterschiedliche Sicherheitsstandards und herstellerspezifische Sicherheitsmechanismen, über deren Implementierung der Hersteller unter Umständen kein Know-how außer Haus geben will, müssen leicht integrierbar sein.

- Der Speicherplatzbedarf des Flash-Laders sollte minimal sein und die Nutzdatenrate des Bussystems bestmöglich ausnutzen, um kurze Programmierzeiten zu gewährleisten.
- Neben der Flash-Lader-Software für die bei Steuergeräten gängigen Compiler-Linker-Umgebungen sollte eine Werkzeugunterstützung für die Konfiguration des Flash-Ablaufs und für das Testen geliefert werden (siehe Abschn. 9.4.5).

Diese Anforderungen lassen sich am besten erfüllen, wenn der Flash-Lader aus einem Kern mit klaren Schnittstellen zu den anwendungsabhängig zu konfigurierenden Teilen besteht:

- Flash-Ablaufbeschreibung,
- Flash-Treiber,
- Diagnose-Protokollstapel (KWP 2000, UDS, ...) mit den verschiedenen Transportprotokollen (ISO TP, VW TP, ...) und Bussystemen (CAN, K-Line, LIN, ...),
- Sonstige projektspezifische Konfiguration des Steuergerätes, z. B. Mikrocontroller-Typ und Taktfrequenz, Ablage der Flash-Programmierhistorie im EEPROM usw.

Da es möglich sein muss, den gesamten Flash-Speicherinhalt einschließlich des Betriebssystems neu zu programmieren, bringt der Flash-Lader ein einfaches eigenes Betriebssystem sowie die notwendigen Hardware-Treiber für diejenigen Peripheriebaugruppen mit, die von ihm verwendet werden. *ADLATUS* bringt einen eigenen Kommunikations-Protokollstapel mit. Als Protokolle lassen sich u. a. UDS (ISO 14229) oder KWP 2000 mit CAN (ISO/DIS 15765), K-Line (ISO 14230) oder LIN verwenden, wobei als Transportschicht sowohl ISO-TP als auch TP 2.0 oder 1.6 einsetzbar sind.

Der *Diagnose-Service-Interpreter* (Abb. 9.21) prüft die vom Diagnosetester empfangenen Botschaften inhaltlich (Service ID, Datenlänge usw.), reicht entsprechende Programmieraufforderungen an den *Flash-Ablauf-Interpreter* weiter und erzeugt anschließend die zugehörigen positiven oder negativen Antwortbotschaften.

Der *Flash-Ablauf-Interpreter* prüft, ob die vom Hersteller definierte Reihenfolge der Botschaften für die Flash-Anforderung korrekt eingehalten wird und führt diese dann aus. Dabei müssen im selben Steuergerät unterschiedliche Abläufe für den Gerätehersteller, den Fahrzeugherrsteller und den Werkstattbetrieb konfiguriert werden können. Innerhalb dieser Abläufe muss es außerdem möglich sein, den Zugriff auf definierte Speicherbereiche (Speicher-Sektoren) mit unterschiedlichen Sicherheitsmechanismen zu erlauben, damit die Software verschiedener Hersteller integriert werden kann. Die für ein spezifisches Projekt geltenden Sicherheitsmechanismen werden dem Flash-Lader an der Sicherheitsmodul-Schnittstelle zur Verfügung gestellt. Dafür stehen je nach Sicherheitsanforderungen verschiedene zertifizierte Kryptografieverfahren zur Verfügung (Tab. 9.7).

Die Konfiguration des Flash-Laders erfolgt in drei Stufen:

- In der *Projektkonfiguration* werden der Kommunikationskanal, die Bitrate, die Geräteadressen, die Funktionen für die Hardwareinitialisierung des Steuergerätes sowie die Ansteuerung des Watchdogs definiert.

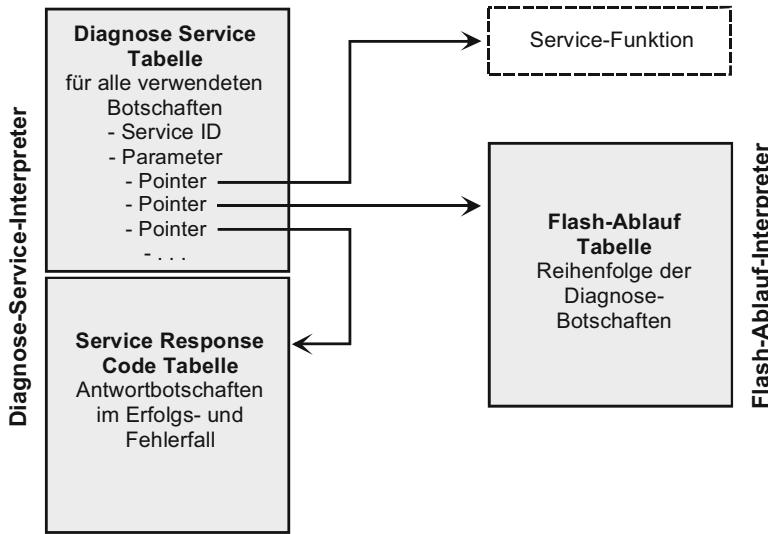


Abb. 9.21 Diagnoseservice-Beschreibung und Flash-Ablauf

- Die Definition des Flash-Ablaufs erfolgt mit Hilfe von drei Tabellen (Abb. 9.21). In der *Diagnose Service Tabelle* werden alle vom Flash-Lader zu bearbeitenden Diagnosebotschaften, deren Parameter und die zugehörigen Bearbeitungsfunktionen eingetragen. In der *Service Response Code Tabelle* werden die zugehörigen Antwortbotschaften im Erfolgs- und Fehlerfall aufgelistet. In der *Flash-Ablauf-Tabelle* schließlich wird der eigentliche Ablauf, d. h. die geforderte Reihenfolge der Botschaften festgelegt. Bei Projekten, bei denen die Diagnosefunktionalität in einer ODX-Datenbank definiert ist, wird die zugehörige Konfiguration aus der ODX-Beschreibung des Steuergeräts extrahiert. Die Konfiguration des Flash-Ablaufs muss in der Regel nur einmal erfolgen und kann für alle weiteren Geräte eines Herstellers übernommen werden.
- Falls der Flash-Lader auf einem neuen Mikrocontroller eingesetzt wird, muss lediglich die *Hardware-Treiberschicht* angepasst werden.

Nachdem die Konfiguration erstellt wurde, kann der Flash-Lader durch einen üblichen Compiler/Linker-Lauf für die jeweilige Zielhardware erzeugt werden. Tab. 9.8 zeigt die Daten des Flash-Laders in einer typischen Konfiguration.

Bei Mehrprozessorsystemen mit einer gemeinsamen Diagnoseschnittstelle kann *AD-LATUS* als lokales Gateway arbeiten (Abb. 9.22). *ADLATUS* leitet die ankommende Diagnosekommunikation dann von und zu den internen Slave-Prozessoren. Dies kann über unterschiedliche Schnittstellen erfolgen, z. B. SPI. Um die Programm- und Datenstände im System konsistent zu halten, sind übergeordnete Maßnahmen nötig.

Das Gegenstück zum Flash-Lader im Steuergerät ist der Diagnosetester, der jedoch in der Entwicklungsphase häufig noch nicht in der endgültigen Form vorliegt. Der Hersteller

Tab. 9.7 Aufwand verschiedener Verschlüsselungs-/Zugriffsschutzmechanismen (nach Angaben von A. Weimerskirch, encrypt embedded security)

Schutzmechanismus	Speicherbedarf		Laufzeit/Durchsatz	
	ROM (Byte)	RAM (Byte)	16bit CPU Freescale HCS12 20 MHz	32bit CPU Infineon TriCore TC1766 40 MHz
SHA-1 Hash (kryptographische Prüfsumme) erzeugen	1 K	100	150 KB/s	350 KB/s
RSA 1024 Signatur Verifikation (kurzer Exponent 3)	3 K	400	150 ms	20 ms
AES 128 Entschlüsselung (ECB Mode)	1,5 K	40	50 KB/s	100 KB/s
ECC 160 (ECDSA) Signatur ohne Hash erzeugen	4 K	200	2200 ms	500 ms
ECC 160 (ECDSA) Signatur ohne Hash verifizieren	4 K	200	2200 ms	1000 ms
Signatur verifizieren (RSA 1024 mit kurzem Exponent 3 inkl. SHA-1 Hash über 128 KB)	4 K	400	1000 ms	400 ms

Tab. 9.8 Typischer Ressourcenbedarf des Flash-Laders *ADLATUS*

Projektkonfiguration	
Mikrocontroller	Freescale MPC 5517
Flash-Baustein	Internes Flash-ROM 1,5 MB
Bussystem	CAN 500 Kbit/s
Diagnoseprotokoll	UDS
Transportschicht	ISO 15765-2
Flash-Ablaufkonfigurationen	1
Sicherheitsmechanismus	Session-Login, Security Access wahlweise mit Signatur und Verschlüsselung
Speicherplatzbedarf (mit Kommunikationsprotokoll und HIS-Flash-Treiber)	
ROM	32 KB, mit Signatur und Verschlüsselung 48 KB
RAM	9 KB
Programmierdauer für 100 KB (inkl. CAN-Übertragungsdauer)	13 s

des Flash-Laders muss daher für die Entwicklungsphase eine Simulation des Diagnose-testers, in Form eines PC-Programms und einer für den PC geeigneten Schnittstelle zum Kfz-Bussystem bereitstellen (Abb. 9.23). Diese Aufgabe übernimmt das *SMART* Werkzeug *FlashCedere*. Es ist in der Lage, mehrere Steuergeräte innerhalb eines Kommunikations-

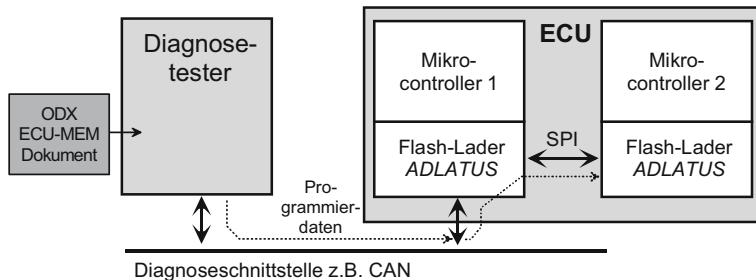


Abb. 9.22 Flash-Lader mit Gateway-Funktion

netzes parallel zu programmieren und zu testen. Dadurch werden die Programmierzeiten verkürzt, soweit die Busbandbreite dies zulässt.

Die Datenhaltung der zu programmierenden Daten erfolgt in einem sogenannten Flash-Container, der neben dem eigentlichen Flash-Speicherinhalt eine Reihe von Verwaltungsinformationen enthält. Während diese Flash-Container früher bei jedem Hersteller anders aufgebaut waren, setzt sich allmählich das von ASAM mit ODX definierte ECU-MEM-Format durch, das eine einheitliche XML-Beschreibung erlaubt (siehe Abschn. 6.6).

Zu einer Flash-Lader Lösung gehört ergänzend immer auch ein Konzept, mit dem nicht nur Betriebssystem und Anwendungssoftware eines Steuergerätes, sondern der Flash-Lader selbst über die Diagnoseschnittstelle neu programmiert werden kann. Dies wird notwendig, wenn ein Hersteller seinen Flash-Ablauf modifizieren oder neue Sicherheitsmechanismen integrieren will.

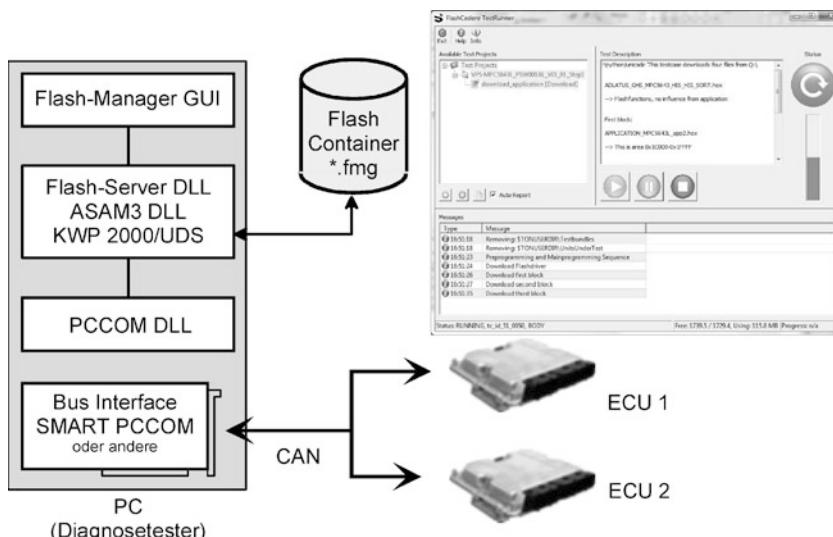


Abb. 9.23 FlashCedere als Diagnosetest für Entwicklung und Produktion

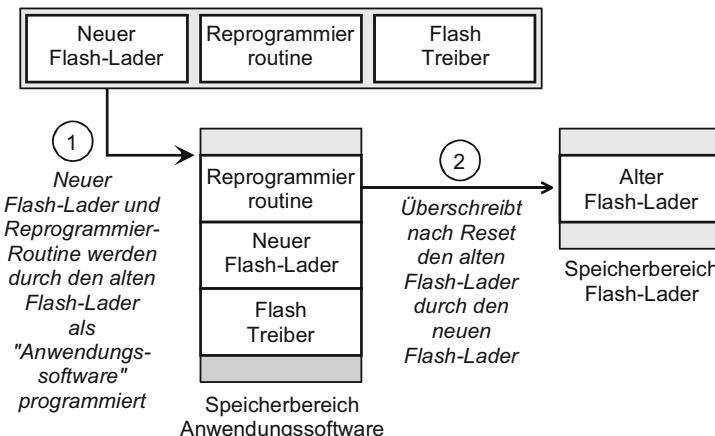


Abb. 9.24 Automatisches Ersetzen des ADLATUS Flash-Laders

Im laufenden Betrieb darf sich der Flash-Lader nicht einfach selbst überschreiben. Aus diesem Grund liegt er meist in einem gegen versehentliches Überschreiben geschützten, speziellen Bereich des Flash-Speichers. Daher ist für die Reprogrammierung des Flash-Laders eine spezielle Vorgehensweise notwendig (Abb. 9.24). Der neue Flash-Lader wird zunächst zusammen mit einer speziellen Reprogrammierroutine als normale Anwendungssoftware in den Flash-Speicher programmiert. Beim nächsten Steuergerätestart wird diese Anwendungssoftware dann gestartet. Die Reprogrammierroutine löscht den alten Flash-Lader und kopiert den neuen in den Speicherbereich des alten Flash-Laders. Anschließend markiert sich die Anwendungssoftware als ungültig. Da das Steuergerät beim nächsten Start keine gültige Anwendungssoftware enthält, wird nun der neue Flash-Lader gestartet, mit dem wiederum eine *echte* Anwendungssoftware programmiert werden kann.

Die Anpassung der Schnittstellen gemäß den neuesten AUTOSAR- bzw. HIS-Spezifikationen erfolgt beim Flash-Lader wie bei anderen Standard-Softwarekomponenten bewusst mit einer gewissen Verzögerung, da die Spezifikationen zunächst nur vorläufigen Charakter haben. Dies betrifft insbesondere die Anpassung der Konfigurationswerkzeuge, die mittelfristig auf die AUTOSAR-typische XML-basierte Konfiguration umgestellt und in Form eines Eclipse-Plugins in die AUTOSAR-Entwicklungslandschaft integriert werden sollen.

9.4.5 Softwaretest von Flash-Ladern und Busprotokollen

Der breite Einsatz von flashbaren Steuergeräten im Automobil stellt hohe Anforderungen an die Qualitätssicherung bezüglich Testtiefe und Testeffizienz. Für den Softwaretest und die Freigabe von Flash-Ladern und Kommunikationsprotokollen werden daher zunehmend automatisierte Tests eingesetzt. Als Beispiel für die Automatisierung solcher Tests soll der Konformitätstester *FlashCedere* von SMART vorgestellt werden.

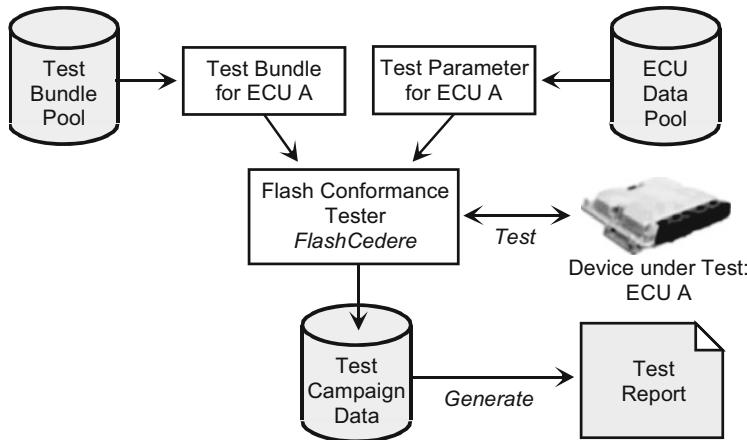


Abb. 9.25 Testen mit SMART *FlashCedere*

Der automatisierte Test erfolgt über die Kommunikationsschnittstelle des Steuergerätes. Die Testsoftware, hier *FlashCedere*, läuft auf einem gewöhnlichen PC mit entsprechendem Businterface, sendet Diagnoseanfragen an das zu testende Gerät und wertet dessen Antwortbotschaften bezüglich Inhalt, Reihenfolge und zeitlichem Verhalten aus. Das Programm unterstützt FlexRay, CAN, LIN, K-Line und Ethernet.

Die Entwicklung und Durchführung des automatisierten Tests erfolgen im Hinblick auf die Wiederverwendbarkeit in anderen Projekten in mehreren Stufen (Abb. 9.25):

- Erstellen generischer *Testfälle* in einer *Testfall-Bibliothek*: Ein Testfall besteht in der Regel aus einer Anforderungsbotschaft und der zugehörigen Antwort, in Ausnahmefällen auch mehreren Botschaften, und prüft eine eng begrenzte Teilfunktion, z. B. die Abfrage der aktiven Diagnosesitzung. Bei *FlashCedere* werden die Testfälle in der Skriptsprache Python erstellt (Abb. 9.26). Generische Testfälle zeichnen sich dadurch aus, dass variablenspezifische Parameter, z. B. die CAN-Bitrate oder die Steuergeräte-Diagnoseadresse, nicht unmittelbar im Testfall codiert sind, sondern von außen parametrierbar sind.
- Für die professionelle Testdurchführung gibt es für *FlashCedere* parametrierbare Testbibliotheken, z. B. für den Test des ISO-Transportprotokolls oder für die gängigen Flash-Lastenhefte deutscher Fahrzeughersteller. In den Testfällen sind die Flash-Ablaufdefinitionen und zugehörige Parameter strikt getrennt, um so die Tests einfach an ein Steuergerät anpassbar zu machen.
- Erstellen eines *Testbündels*: Eine Auswahl an Testfällen aus der generischen Bibliothek kann zu einem Testbündel zusammengefasst werden (Abb. 9.27). Ein Testbündel fasst den jeweils gewünschten Testumfang zusammen, z. B. die vollständige Prüfung einer Flash-Programmiersequenz. Mit *FlashCedere* können vorkonfigurierte Tests einschließlich der Erstellung des Testberichts automatisiert durchgeführt werden.

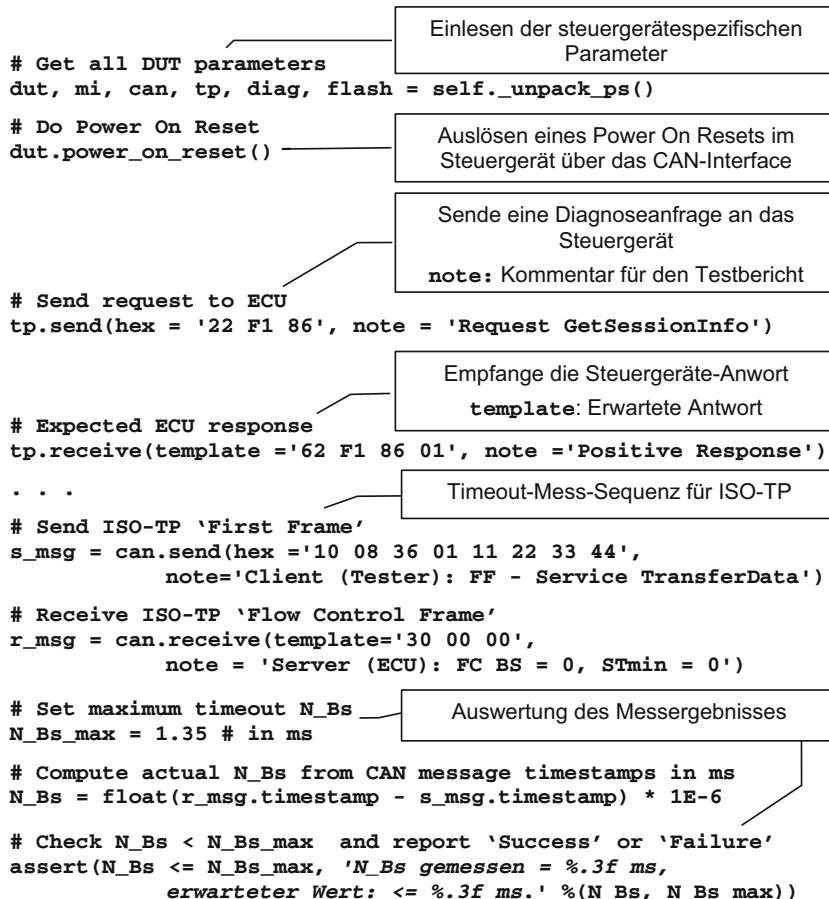


Abb. 9.26 Python-Testscript für den Testfall *Abfrage der aktiven Diagnosesitzung*

- *Testbedatung*: Die steuergerätespezifischen Parameter werden aus dem ODX-Container gelesen oder manuell über die Benutzeroberfläche eingegeben.
- *Test Campaign*: Die bei der Durchführung der Tests erfassten Daten mit Zeitstempeln werden als Datensatz archiviert.
- *Test Reports*: Nach dem Testlauf wird die *Test Campaign* ausgewertet und das Ergebnis in einem Testbericht übersichtlich zusammengefasst (Abb. 9.28).

Mit diesem Konzept lassen sich die höheren Schichten des Kommunikationsstacks eines Steuergerätes prüfen, bei einer CAN-basierten Busschnittstelle also beispielsweise die Spezifikationen bzw. Protokolle HIS Flash Lader, UDS (ISO 14229) und ISO-Transportprotokoll (ISO 15765-2). Indirekt werden sogar der Data Link Layer und der Physical Layer (ISO 11898) mit geprüft.

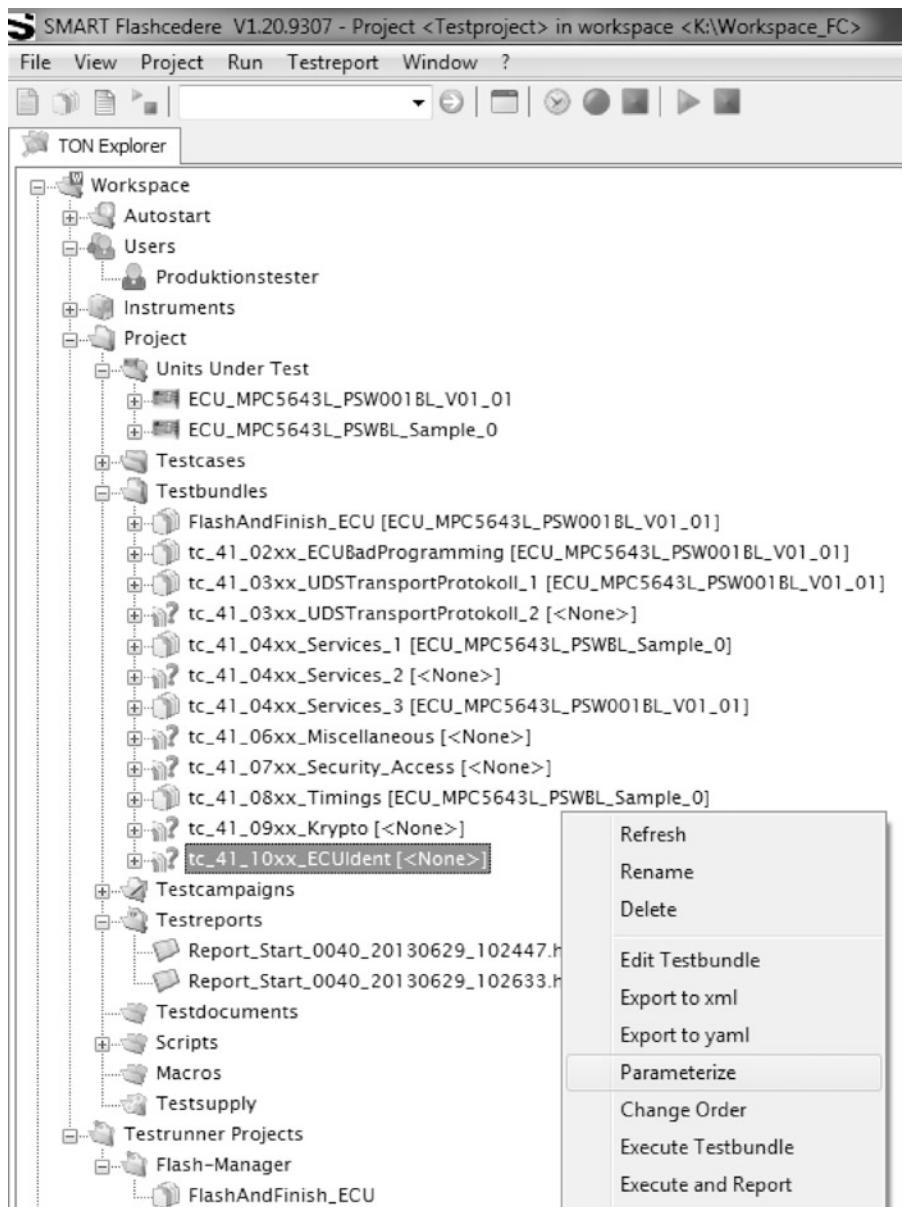


Abb. 9.27 Projektübersicht mit generischen Testfällen in *FlashCedere*

Abb. 9.28 Testbericht im HTML-Format, über *Cascaded Style Sheets* formatiert

9.5 Diagnosewerkzeuge in Entwicklung und Fertigung

Aufgrund der komplex vernetzten Strukturen in modernen Fahrzeugen spielt die Diagnose nicht nur in den Werkstätten, sondern auch in der Entwicklungs- und Applikationsphase sowie in der Fertigung beim Geräte- und beim Fahrzeugherrsteller eine wichtige Rolle. Die Anforderungen in den drei Einsatzbereichen unterscheiden sich dabei allerdings deutlich. In der Werkstatt steht die Unterstützung des Werkstattmechanikers bei der gezielten Fehlersuche im Fahrzeug im Vordergrund. Wesentlich dafür ist eine leicht bedienbare Benutzeroberfläche mit integrierten Fehlersuch- und Reparaturanleitungen und Anbindung an Ersatzteilkataloge oder gar Abrechnungssysteme. Interne Details des Diagnose- und des Busprotokolls müssen gegenüber dem Benutzer bestmöglich gekapselt werden, um ihm das Arbeiten zu vereinfachen. Im Gegensatz dazu steht beim Einsatz in der Entwicklungs- und Applikationsphase der Zugriff auf sämtliche Details bis zum Zeitverhalten der Kommunikation auf den Busleitungen im Vordergrund. Mit dem Diagnosetester werden dabei nicht nur echte Diagnoseaufgaben wie das Auslesen des Fehlerspeichers durchgeführt, sondern der Diagnosetester soll oft auch als Messwerterfassungs- oder gar Parameterverstellwerkzeug bei Software-, Aggregate- oder Fahrtests dienen. Im Extremfall wird er sogar dafür eingesetzt, die Kommunikation mit noch nicht vorhandenen anderen Steuergeräten zu simulieren (Restbus-Simulation). Der Einsatz in der Fertigung bildet eine Kombination der beiden Extreme. Die Prüfung muss wesentlich tiefer gehend erfolgen können als in der

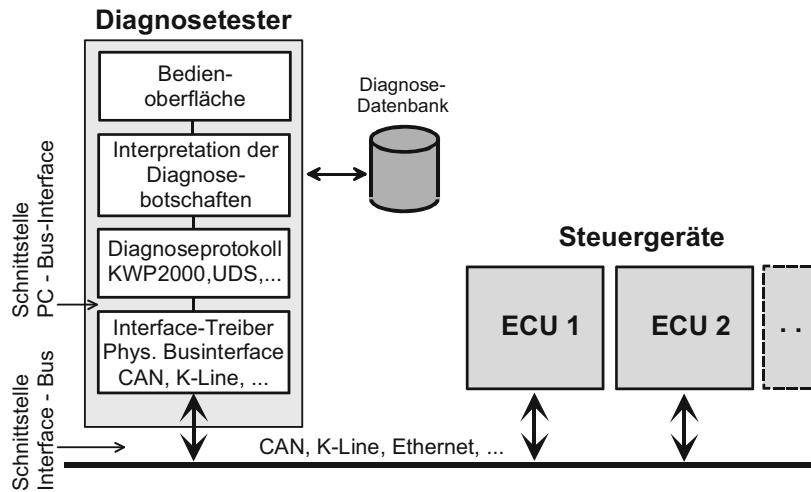


Abb. 9.29 Typische Struktur eines Diagnosetesters

Werkstatt. Die Komplexität der Buskommunikation soll zwar gekapselt sein, falls notwendig muss aber auch der Zugriff auf Details möglich sein. Im Gegensatz zu Werkstatt und Entwicklung, wo das Diagnosewerkzeug vor allem der Unterstützung eines menschlichen Bedieners dient, muss der Prüfablauf in der Fertigung weitgehend automatisiert erfolgen. Die erreichbare Prüfgeschwindigkeit ist dort extrem wichtig.

Wegen der verschiedenartigen Anforderungen in den drei Einsatzbereichen Entwicklung, Fertigung und Werkstatt sind die Ansprüche an ein gemeinsames Werkzeug sehr hoch. Bei den Fahrzeugherstellern finden sich daher für jeden Bereich oft noch unterschiedliche Diagnosesysteme. Das Ziel, durch durchgängige Werkzeuge ohne Technologiebrüche mit weniger Lieferanten Kosten zu senken, zwingt aber auch bei Diagnosekonzepten zu Gesamtlösungen, die von der Entwicklung über die Produktion bis hin zur Werkstatt eingesetzt werden können. Der typische Diagnosetester besteht heute aus einem konventionellen Windows-PC bzw. Notebook, gegebenenfalls *werkstattgerecht* verpackt (Abb. 9.29). Die in Werkstätten zum Einsatz kommende Bedienoberfläche wird in der Regel vom Fahrzeughersteller oder von großen Werkstattausrüstern wie Bosch selbst entwickelt, da hierzu umfangreiches Werkstatt-Know-How erforderlich ist. Die darunter liegenden Softwareschichten, die der Handhabung des Diagnoseprotokolls und der Anbindung an das Kfz-Bussystem dienen, sind häufig Komponenten eines Zulieferers, die bereits in der Entwicklungs- und Applikationsphase zum Einsatz kommen. Diese können auch in der Fertigung verwendet werden, sofern sie die entsprechende Automatisierung erlauben. Damit dieselben Komponenten auch in der Entwicklungs- und Applikationsphase eingesetzt werden können, muss der Zulieferer auch eine dafür geeignete Bedienoberfläche anbieten. Die spätere Benutzeroberfläche des Werkstatttesters bietet für die Entwicklung in der Regel zu wenige Freiheiten und ist in dieser Phase meist auch noch gar nicht verfügbar.

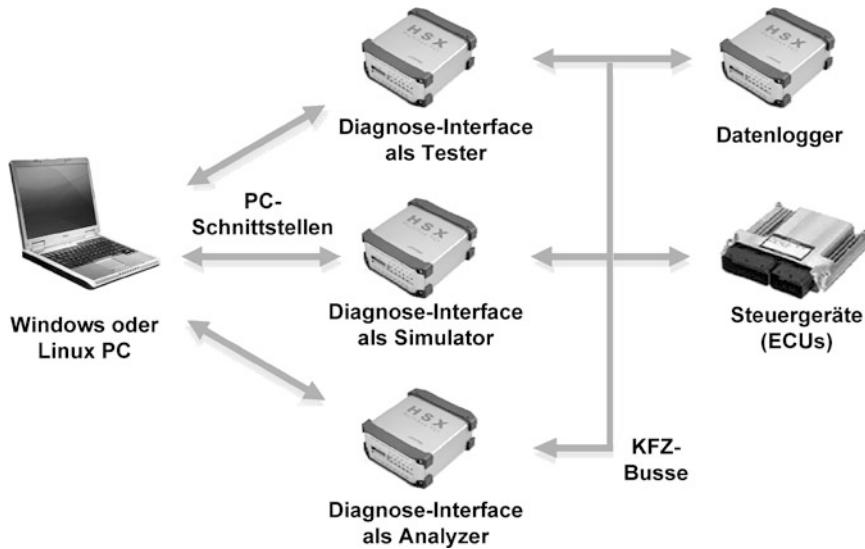


Abb. 9.30 Funktionen der Samtec Diagnose-Interfaces

9.5.1 Beispiel für Diagnosewerkzeuge: samDia von Samtec Automotive

Auf dem Markt der Diagnosewerkzeuge hat sich eine Reihe von Spezialisten etabliert, die in der Regel eng mit den Geräte- und Fahrzeugherstellern zusammenarbeiten. Mit *samDia* der Samtec Automotive Software & Electronics GmbH soll im Folgenden ein typisches, weit verbreitetes Diagnosewerkzeug vorgestellt werden, das alle gängigen Bussysteme und Protokolle unterstützt.

PC-Interface für Kfz-Bussysteme Da die typischen PC-seitigen Schnittstellen wie Ethernet oder USB für die direkte Kopplung mit den Kfz-typischen Bussystemen nicht geeignet sind, ist ein Interface einschließlich eines PC-seitigen Softwaretreibers notwendig (*Vehicle Communication Interface VCI*). Die Abwicklung des Busprotokolls, insbesondere die Einhaltung der Zeitbedingungen stellt Anforderungen, die mit den konventionellen PC-Betriebssystemen wie Windows allein nicht zuverlässig erfüllt werden können. Daher statet man das Bus-Interface in der Regel mit eigener *Intelligenz* aus, d. h. einem Mikrocontroller mit ausreichend großem Datenspeicher, der den zeitkritischen Teil des Busprotokolls autark erledigen kann und den geforderten Datendurchsatz gewährleistet. Insbesondere für die Restbus-Simulation und Datenanalyse in Echtzeit ist dies unabdingbar.

Das Bus-Interface kann sowohl als PC-Einsteckkarte oder als separates Modul mit USB- oder Ethernet-Schnittstelle zum PC ausgeführt werden. Selbst Anbindungen per Funk über WLAN, GPRS/UMTS/LTE oder Bluetooth werden angeboten und je nach Anwendungsfall eingesetzt. Ethernet z. B. in Programmierstationen, wenn es auf hohen Datendurchsatz ankommt. WLAN etwa in der Fahrzeugfertigung, wenn das Interface-Modul im Fahrzeug

mitgeführt wird und mit dem in der Fertigungsline fest montierten Prüfrechner ohne ständiges Ein- und Ausstecken eines Kabels kommunizieren soll. Selbst bei Versuchsfahrten in der Flottenerprobung und auf Teststrecken kann WLAN mit Reichweiten von bis zu 500 m heute eingesetzt werden. USB ist gut für Notebook geeignet, insbesondere wenn an einem Gerät mehrere Busse gleichzeitig angeschlossen werden müssen. Bluetooth erlaubt den mobilen Betrieb für einfache Werkstattanwendungen etwa mit einem Pad-Computer oder Smartphone.

Die vier wesentlichen Funktionen aus Anwendersicht sind (Abb. 9.30):

- **Tester/Stimulator:** Senden von Diagnosebotschaften an ein oder mehrere Steuergeräte und Empfang der Steuergeräte-Antworten, z. B. beim Auslesen des Fehlerspeichers. Dies entspricht dem typischen Betrieb eines Diagnosetesters.
- **Simulator:** Das Kommunikationsverhalten eines Steuergerätes wird simuliert. Diese Betriebsart wird während der Entwicklung eingesetzt, wenn ein Gerät noch nicht vorhanden ist oder wenn es im Labor wegen fehlender Sensor- und Aktorsignale nicht sinnvoll betrieben werden kann. Die Funktion findet ihren Einsatz aber auch in Fertigungsprüfständen, wenn ein Steuergerät einzeln getestet werden soll, das für seine Funktion auf die Kommunikation mit anderen Geräten angewiesen ist.
- **Analyzer:** Die laufende Kommunikation auf dem Bussystem wird mitgehört, aufgezeichnet und interpretiert, ohne die Kommunikation selbst zu beeinflussen, d. h. das Interface verhält sich passiv. In dieser Betriebsart sollen in der Regel nicht nur die Daten erfasst, sondern oft auch das Zeitverhalten, z. B. Inter-Block- und Inter-Byte-Zeiten mit Genauigkeit im Mikrosekundenbereich gemessen werden.
- **Datenlogger:** Der Datenlogger-Betrieb ist eine Sonderform des Analyzer-Betriebs, bei dem das Bus-Interface die Aufzeichnung der Datenkommunikation selbstständig auch ohne angeschlossenen PC bzw. Notebook durchführt. Dies kann z. B. bei Testfahrten sinnvoll sein, wenn das Mitführen der vollständigen Ausrüstung aus Platz- oder Kostengründen nicht sinnvoll möglich ist. Die vorherige Konfiguration des Datenloggers ermöglicht die Definition von Triggerbedingungen und des jeweiligen Vor- und Nachlaufes der Aufzeichnung. Die nachfolgende detaillierte Auswertung erfolgt wie im Analyzer-Betrieb. In der Prototypenphase können die Fehlerbilder sehr komplex sein, so dass die Kombination einer Vielzahl von digitalen und analogen Messdaten notwendig wird, während in der Vorserie oft die einfache Auswertung der Fehlerspeichereinträge genügt.

samDia Software Die oben aufgeführten Funktionen des Bus-Interface-Moduls werden mit der Bedienoberfläche der Software gesteuert. Nach der Grundeinstellung des Busprotokolls, der zu verwendenden Transportschicht sowie der Protokollparameter wie Geräteadressen, Bitrate, Inter-Block- oder Inter-Byte-Zeiten wird die eigentliche Funktion konfiguriert. Alle Einstellungen und die aufgezeichneten Daten können abgespeichert und jederzeit wieder aufgerufen werden.

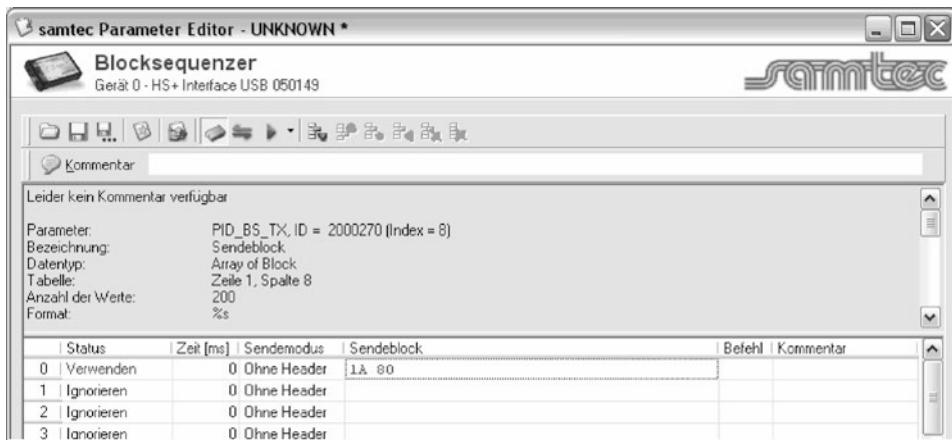


Abb. 9.31 Blocksequenzer (Samtec *samDia*)

Die Kernkomponente von *samDia* ist der sogenannte Blocksequenzer. Mit diesem können nahezu beliebige Kommunikationsszenarien sequenziell und zyklisch abgebildet werden, wobei mehrere Instanzen des Blocksequenzers gleichzeitig ablauffähig sind. Durch seine Skriptfähigkeit kann auf Busereignisse und ankommende Botschaften sowie deren Dateninhalt aktiv reagiert werden. Der Blocksequenzer inklusive der Skriptsteuerung kann autark im Interface ohne PC ablaufen. Damit lassen sich geringere Reaktionszeiten bei Prüfabläufen, bei der Restbussimulation oder bei der Nachbildung von Gateway-Funktionen erreichen als bei rein PC-gestützten Konzepten.

Für die *Stimulator*-Funktion werden die zu sendenden Botschaften mit ihren Parametern in den Blocksequenzer eingetragen (Abb. 9.31). Zusätzlich kann ein *Klartext*-Kommentar angegeben werden, der im Protokoll erscheint, sobald die Botschaft gesendet wird. Auf diese Weise können komplette Botschaftsfolgen z. B. für die Abfrage des Fehler-speichers, das Auslesen der einzelnen Fehlerinformationen und das anschließende Löschen der Fehlerinträge definiert werden. Um vorkonfigurierte Abläufe leicht modifizieren oder fehlerhafte Abläufe testen zu können, ist es möglich, Botschaften aus- oder einzublenden (ignorieren/ausblenden).

Während *samDia* im *Stimulator*-Betrieb Botschaften selbstständig sendet, wartet die *Simulator*-Funktion auf ankommende Botschaften und erzeugt dann eine entsprechende Antwort. Bei der Konfiguration im Blocksequenzer wird festgelegt, auf welche Botschaft bzw. welchen Botschaftsinhalt (Trigger-Bedingungen) reagiert werden und welche Antwortdaten zurückgesendet werden sollen.

Im *Analyzer*-Modus werden die Daten aufgezeichnet und dargestellt. Der Anwender kann Teile der Botschaft, z. B. den Header einer Botschaft mit den Geräteadressen oder den Trailer mit der Prüfsumme ausblenden, um die Anzeige übersichtlicher zu gestalten. Die Anzeige kann im *Klartext* erfolgen, wobei die Umsetzung zwischen den Hexadezimalwer-

```

1  dim dialog
2
3  = sub main
4      rem create and open dialog window
5      set dialog = sgsim.NewDialog(300,120,"ECU Programming Example")
6      rem create text field and button
7      dialog.SetTextField 2, 10, 25, 280, 25, 0, ""
8      dialog.SetButton 1,10,50,200,25,"&Start ECU Programming","StartDownload"
9  end sub
10
11 = sub StartDownload
12     rem start communication
13     dialog.SetTextField 2, 10, 25, 280, 25, 0, " ECU start communication"
14     samdia.StartMeasuring
15     rem start blocksequencer
16     dialog.SetTextField 2, 10, 25, 280, 25, 0, " ECU programming running"
17     samdia.SendSingle
18 end sub
19
20 = sub DownloadOK
21     rem ECU programming finished successfully
22     dialog.SetTextField 2, 10, 25, 280, 25, 0, "ECU programming finished successfully"
23     rem stop communication
24     samdia.StopMeasuring
25 end sub
26
27 = sub DownloadNotOK
28     rem ECU programming finished with errors
29     dialog.SetTextField 2, 10, 25, 280, 25, 0, "ECU programming finished with errors"
30     rem stop communication
31     samdia.StopMeasuring
32 end sub

```

Abb. 9.32 Skript zur automatischen Steuergeräte-Programmierung über KWP 2000

ten und dem Klartext bei den standardisierten Botschaften vordefiniert, für herstellerspezifische Botschaften vom Anwender aber auch jederzeit konfiguriert werden kann. Neben der Bedeutung eines Wertes, wie *Drehzahl* oder *Temperatur* kann die Einheit, z. B. *1/min*, eine Umrechnungsformel zwischen dem Hexadezimalwert und dem physikalischen Wert und die Art der Darstellung angegeben werden.

Neben der Definition von Botschaften auf der Ebene des Diagnoseprotokolls (*Off-Board*-Kommunikation), die dann über das darunter liegende Transportprotokoll versendet werden, lassen sich Botschaften auch direkt auf der Ebene von einzelnen CAN-Botschaften definieren (*Bus Direkt* Funktion für die *On-Board*-Kommunikation). Auf diese Weise kann man das Verhalten bei Protokollfehlern überprüfen oder Steuergeräte testen, die proprietäre Protokolle verwenden. Durch den Import der verbreiteten CANdb-Beschreibungsdateien (DBC-Format) lassen sich solche Botschaften komfortabel vorkonfigurieren.

Um komplexe Abläufe festzulegen und wiederholte Abläufe automatisieren zu können, kann *samDia* über eine der weit verbreiteten Skriptsprachen wie Visual Basic oder Java Script gesteuert werden. Als Beispiel soll die automatisierte Bandende-Programmierung eines Steuergerätes mit Hilfe von *KWP 2000 über CAN* betrachtet werden.

Der Diagnosetestler wird durch das in Abb. 9.32 dargestellte Skript automatisiert. Die Funktion `main()` des Skripts definiert eine Dialogbox, die dem Anwender den Start des Vorgangs erlaubt. Die eigentliche Programmierung findet in der Funktion `StartDownload()` statt, die den Blocksequenzer aufruft. Nach erfolgreichem Programmierungsvorgang erhält der Anwender eine Rückmeldung über die Funktion `DownloadOK()`.

Die vom Diagnosetestler zu sendenden KWP 2000-Botschaften werden im Blocksequenzer festgelegt (Abb. 9.33). Nach Lesen der Steuergerätekennung (*Read ECU Identification*), Öffnen der Diagnosesitzung (*Start Diagnostic Session*) und Login-Prozedur (*Security Access – Request Seed – Calculate Key – Send Key*) wird der Flash-Speicher zunächst gelöscht (*Start Routine by Local ID – Erase Data*). Dabei wird gewartet, bis das Löschen erfolgreich abgeschlossen ist (*Request Routine Results by ID*), bevor der Programmierungsvorgang eingeleitet (*Request Download*) und die Daten übertragen werden (*Transfer Data*). Im Beispiel ist der Name der Datei mit den Programmierdaten (`c:\myhex.hex`) fest vorgegeben, er hätte aber genauso gut im Automatisierungsskript vom Anwender über eine Dialogbox abgefragt werden können. Abschließend wird die Prüfsumme berechnet, der Erfolg der Programmierung abgefragt und der Anwender durch Aufruf von `DownloadOK/NotOK()` informiert.

Wollte man dagegen einen vorhandenen Diagnosetestler überprüfen und das zu programmierende Steuergerät simulieren, so würde die Blocksequenz wie in Abb. 9.34 aussehen. Probeweise kann man sogar den in *samDia* im Stimulator-Betrieb nachgebildeten Diagnosetestler das im Simulator-Betrieb nachgebildete Steuergerät „programmieren“ lassen, indem man zwei *samDia*-Bus-Interface-Module direkt miteinander verbindet. Abbildung 9.35 zeigt den zugehörigen Auszug aus der *samDia*-Protokolldatei.

Simulation des Gesamtfahrzeugs für die Diagnosetestler-Entwicklung Bei der Entwicklung der Diagnosetestler-Software stellt sich häufig das Problem, dass das eigentliche Fahrzeug nicht oder nur zeitweise zur Verfügung steht, weil sehr viele Entwickler damit arbeiten wollen. Auch wenn stattdessen oft Steuergeräte-Brettaufbauten verwendet werden, verhalten sich die Geräte ohne die Sensorik und Aktorik anders als in der realen Einbausituation im Fahrzeug. Dazu kommt, dass die Anzahl der verbauten Varianten in den heutigen Fahrzeugbaureihen so groß ist, dass auch mit Fahrzeug nicht alle Kombinationen getestet werden können.

Um diese Probleme zu vermeiden, benötigt man eine realitätsnahe Simulation der Diagnosefunktionen des gesamten Steuergeräteverbundes. Dazu kann man die Funktionen *Analyzer*, *Simulator* sowie *Stimulator* von *samDia* sowie die hohe Performanz des *samtec HSX-Interfaces* nutzen (Abb. 9.36). Die Erstellung der Fahrzeugsimulation erfolgt in drei Schritten:

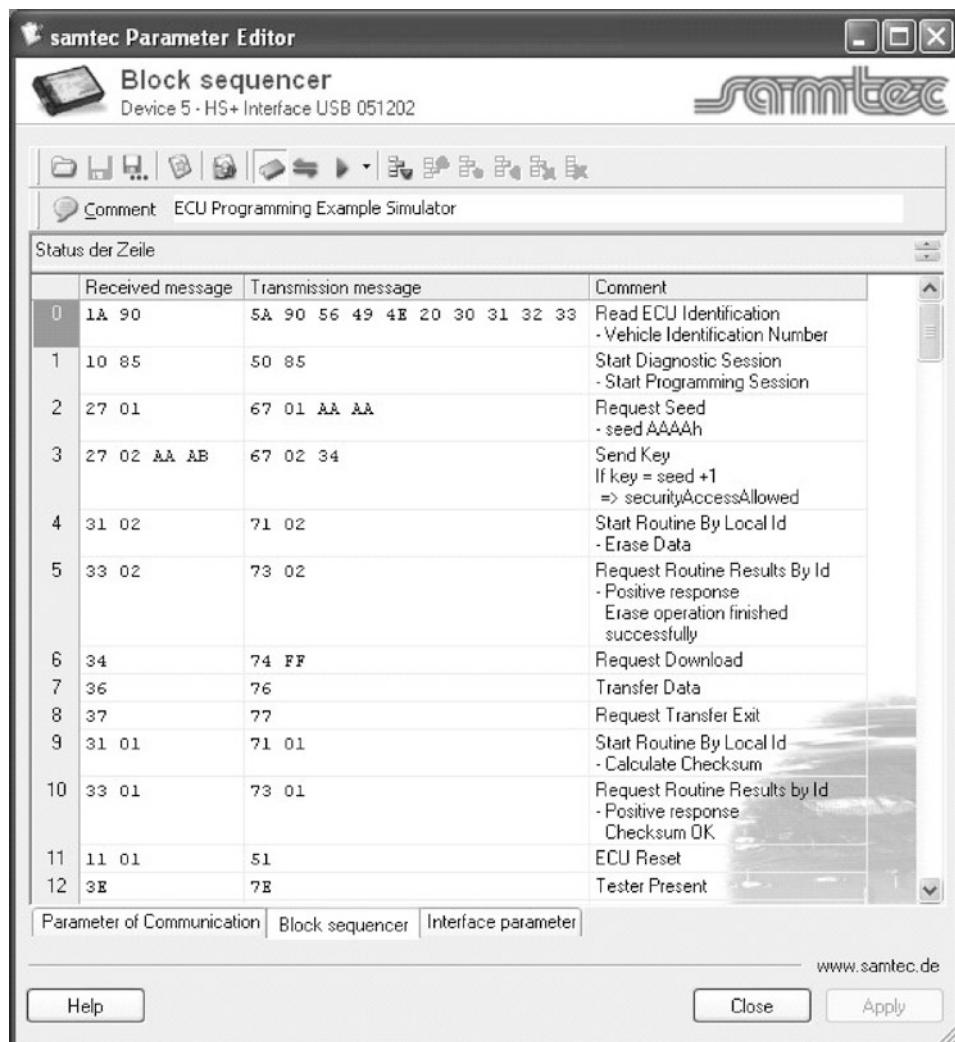


Abb. 9.33 Simuliertes Steuergerät

- Aufzeichnung der Diagnosebotschaften am realen Fahrzeug.
- Aufbereitung der gewonnenen Datenströme mit dem *samDia Simulator-Wizard*. Dieser filtert die Datenströme und gruppiert sie nach Steuergeräten. Die dynamischen Werte in den Botschaften, die sich während der Aufzeichnung verändert haben, werden erkannt und später originalgetreu wiedergegeben. Weiterhin kann der Anwender mit dem *Blocksequenzer* eigene dynamische Botschaften oder Fehlerspeichereinträge konfigurieren. Nach Vortests und Optimierung der erzeugten Daten wird die Simulationsvariante als Datei gesichert und versioniert.

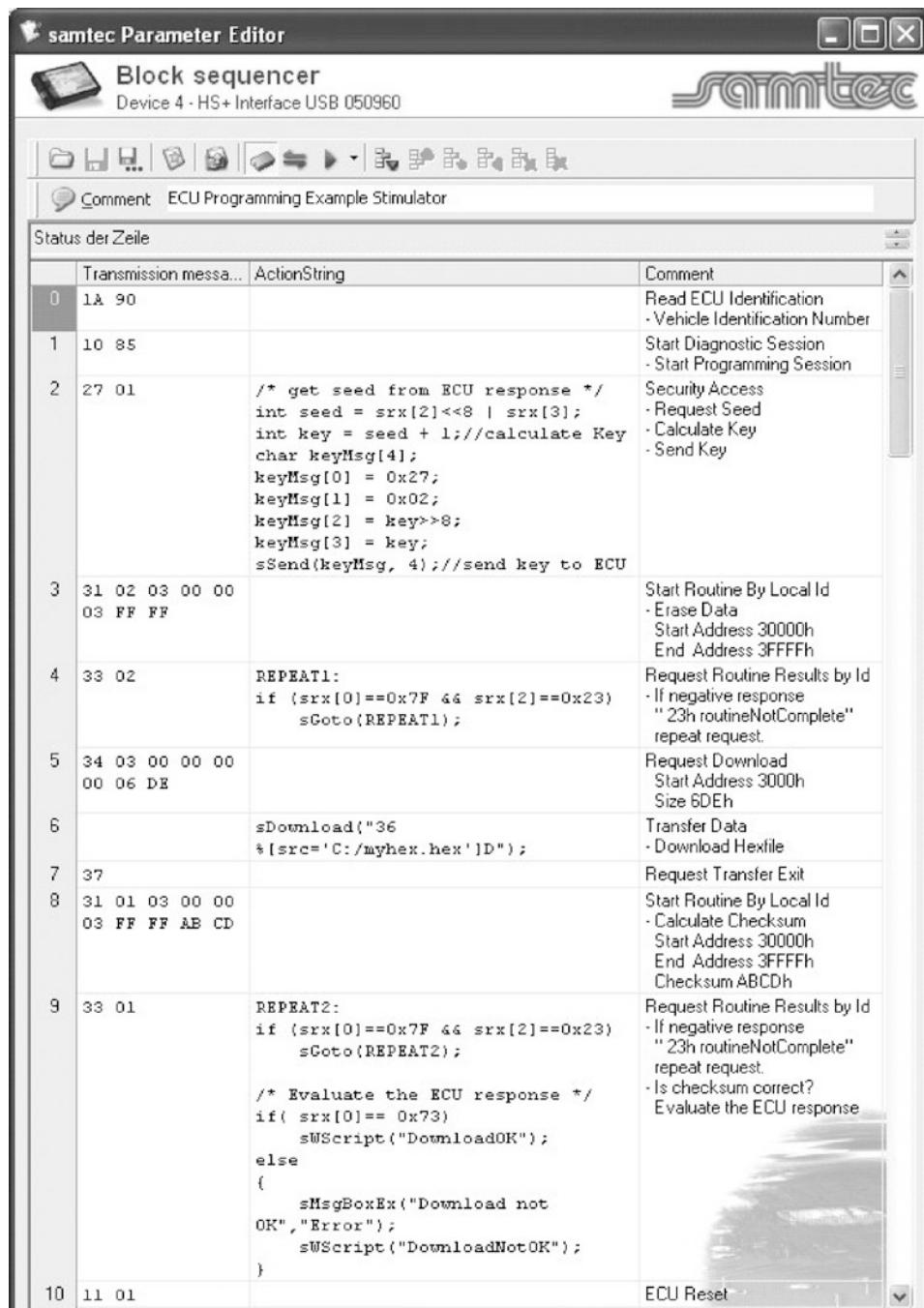


Abb. 9.34 Blocksequenz für den Diagnosetest (Stimulator)

```

> 1a 90      Read ECU Identification - Vehicle Identification Number
< 5a 90 56 49 4e 20 30 31 32 33

> 10 85      Start Diagnostic Session - Start Programming Session
< 50 85

> 27 01      Security Access- Request Seed - Calculate Key - Send Key
< 67 01 aa aa

> 27 02 aa ab Read ECU Identification - Vehicle Identification Number
< 67 02 34

> 31 02 03 00 00 03 ff ff      Start Routine By Local Id - Erase Data
                                Start Address 30000h End Address 3FFFFh
< 71 02

> 33 02      Request Routine Results by Id - If negative repeat request
< 73 02

> 34 03 00 00 00 00 06 de      Request Download
                                Start Address 3000h Size 6DEh
< 74 ff

> 36 20 20 31 48 45 . . .      Transfer Data - Download Hexfile
. . .

> 36 32 30 31 48 45 . . .      Transfer Data - Download Hexfile
. . .

> 37          Request Transfer Exit
< 77

> 31 01 03 00 00 03 ff ff ab cd      Start Routine By Local Id -
                                Calculate Checksum
< 71 01

> 33 01      Request Routine Results by Id . . . - Is checksum correct?
                                Evaluate the ECU response
< 73 01

```

Abb. 9.35 Auszug aus der Protokolldatei des „Diagnosetesters“, die während des Programmierzugs nach Abb. 9.33 und 9.34 aufgezeichnet wurde, > „Diagnosetester“, < „Steuergerät“

- Freigabe der Simulationsdaten. Die getesteten Simulationsdaten können den Entwicklern beispielsweise über einen zentralen Server zur Verfügung gestellt werden. Durch Laden der entsprechenden *Fahrzeugvariante* auf das HSX-Interface des Entwicklers kann die Diagnosekommunikation aller Geräte eines Fahrzeugs getestet werden, ohne dass ein Fahrzeug oder ein Steuergeräteaufbau benötigt wird.

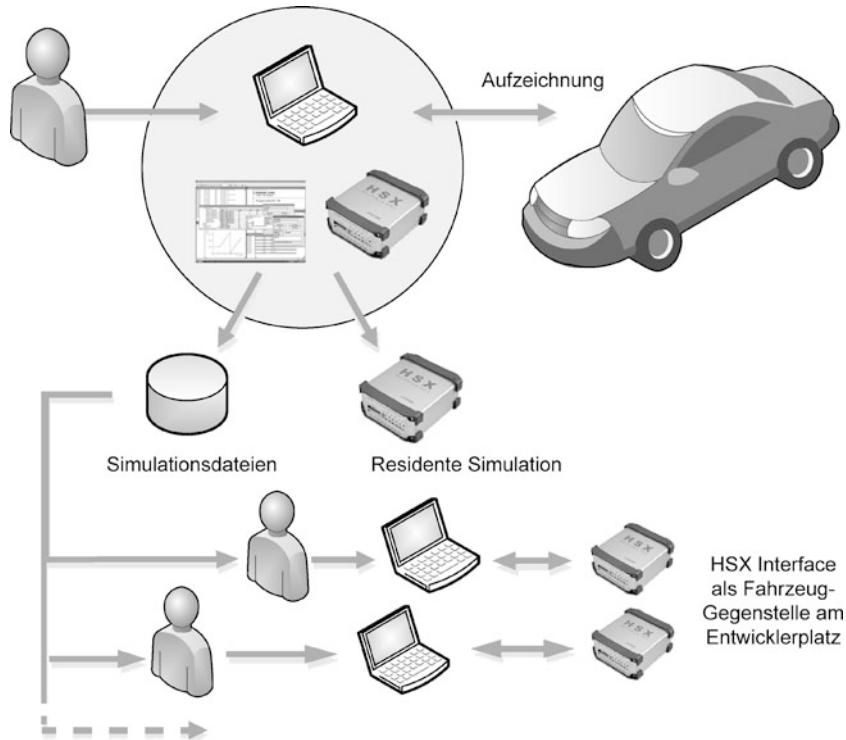


Abb. 9.36 Simulation des Komplettfahrzeuges mit *samDia* und *HSX Interface*

Diagnoseanwendungen In Entwicklung, Fertigung sowie Werkstatt bestehen unterschiedliche Anforderungen an die Bedieneroberflächen und Abläufe. Da aber alle drei nahezu dieselbe Datenbasis verwenden, kann man einen *Cloud*-Ansatz verwenden. Die Verbindung zwischen Diagnoseapplikation und *Cloud* ist bidirektional. Von der Diagnoseapplikation werden die aus der *Cloud* abgerufenen Informationen und Reparaturanweisungen passend für den Anwendungsfall (Abb. 9.37) bereitgestellt. In der Gegenrichtung ist es möglich, Daten über das diagnostizierte Fahrzeug in die *Cloud* einzuspeisen, um hieraus beispielsweise Wahrscheinlichkeiten für die geführte Fehlersuche zu berechnen.

In der Vergangenheit waren Diagnoseapplikationen meist auf eine einzelne Plattform beschränkt. Im Gegensatz dazu erlaubt das *Samtec VCI Communication Framework VCF* eine schnelle und flexible Entwicklung von plattformunabhängigen Diagnoseapplikationen. Im Folgenden werden drei typische Szenarien für den Einsatz vorgestellt:

- In der Entwicklung oder Fertigung muss häufig ein einzelnes Steuergerät isoliert vom Fahrzeug getestet werden. Die fehlenden Kommunikationspartner können durch das *Vehicle Communication Interface VCI*, z. B. ein Samtec HSX-Interface, simuliert werden.

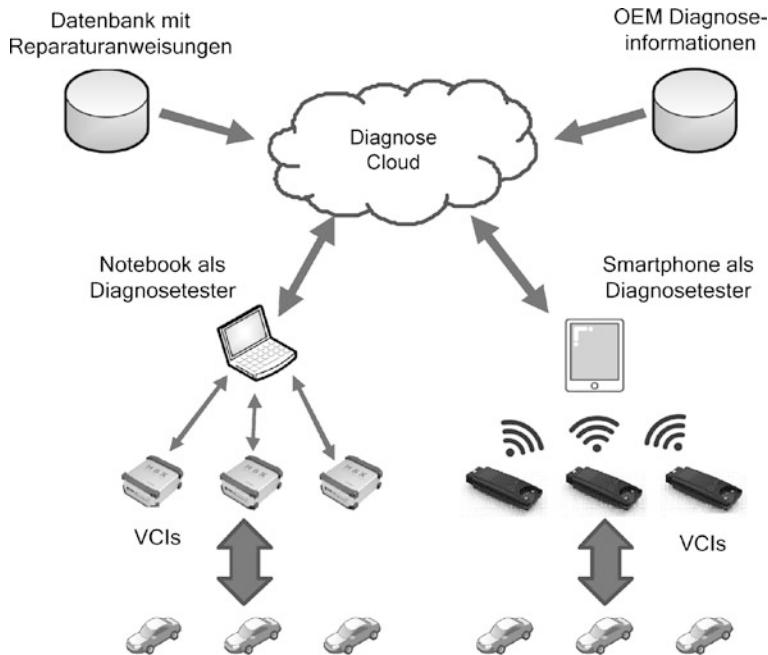


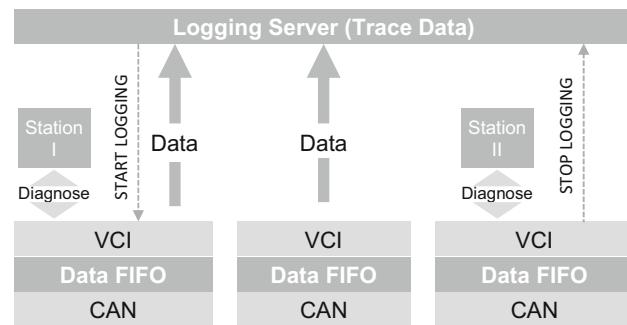
Abb. 9.37 Diagnoseapplikationen

Da das VCI parallel zur Restbussimulation auch die normalen Diagnosefunktionen zulässt, ist ein zweites VCI für die Diagnose nicht nötig.

- In der Fertigung kann es für gezielte Fehleranalysen nötig sein, Diagnosesitzungen über alle Prüfstände hinweg zentral zu protokollieren. Diese Protokollierung kann auf einer Speicherplatte im VCI durchgeführt werden. Statt die Speicherplatten für den Datenaustausch auszubauen, kann man in das VCI einen Web-Server integrieren, so dass der zentrale Fertigungsrechner die Protokolle ohne Betriebsunterbrechung abrufen und weiterverarbeiten kann (Abb. 9.38).
- VCIs existieren in verschiedenen Ausbaustufen mit unterschiedlichen Busschnittstellen. Reicht ein VCI nicht aus, so können mehrere VCIs zu einem virtuellen VCI gebündelt werden. Dadurch ist ein flexibler, kostensparender Aufbau möglich, ohne dass die Diagnoseanwendung angepasst werden müssen.

Mittels des VCF ist es möglich, Diagnoseapplikationen auf allen gängigen Plattformen wie Windows, Linux, Android oder iOS zu entwickeln. Kleinere Applikationen wie eine OBD-Diagnose können einschließlich Bedienoberfläche direkt auf dem Web-Server des VCIs oder als verteilte Anwendung mit Client-seitigen Funktionen realisiert werden. Herzstück des VCF ist das *Firmware Driver Instance Handling* (Abb. 9.39), das die Diagnoseprotokolle aus Sicht der Anwendungen kapselt und die zugehörigen Protokollinstanzen

Abb. 9.38 Zentrale Protokollierung von Diagnosesitzungen in der Fertigung



verwaltet. Dazu gehört unter anderem die Regelung von Lese- und Schreibzugriffen auf die Diagnosebotschaften und der Zugriff auf den Blocksequenzer.

9.6 Autorenwerkzeuge für Diagnosedaten

Die Erstellung und Pflege der Diagnosedaten von Steuergeräten nimmt nach wie vor erhebliche Zeit in Anspruch. Durch den Einsatz des MCD-2D-Datenformates ODX (vgl. Abschn. 6.6) ist zwar das Erscheinungsbild der Daten durch ein standardisiertes Austauschformat gesichert, allerdings ist die Erzeugung und Pflege eines solch komplexen Datensatzes nur durch EDV-Unterstützung mit Hilfe von sogenannten *Autorenwerkzeugen* möglich.

Diese Werkzeuge werden von verschiedenen Herstellern angeboten, etwa *CANdela Studio* (Abb. 9.40) und *ODXStudio* von Vector Informatik, *DTS Venice* von Softing (Abb. 9.41)

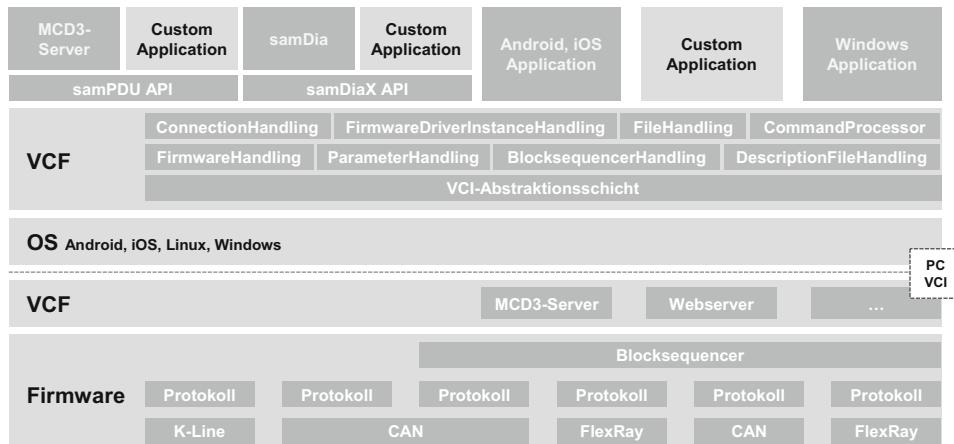


Abb. 9.39 Architektur des *Samtec VCI Communication Framework VCF*

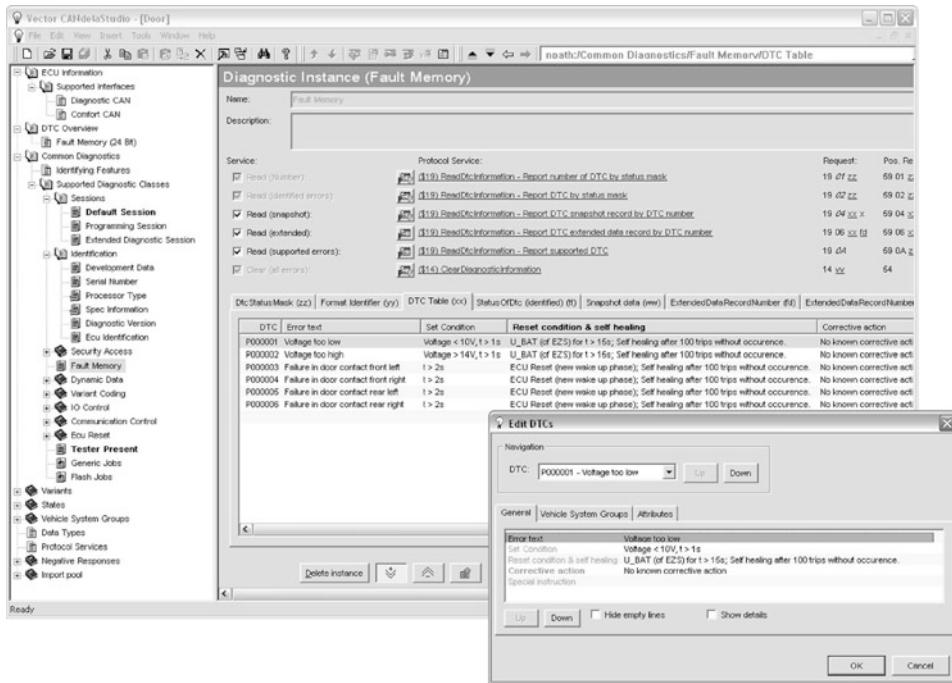


Abb. 9.40 Erstellung von Diagnosedaten (Vector Informatik CANdelaStudio)

oder *VisualODX* von In2Soft. Mit diesen Werkzeugen werden die Daten in komfortablen Bedienoberflächen erstellt, auf Konsistenz geprüft und in einer Datenbank abgelegt.

Es gibt eine Reihe von Herstellern, die Autorenwerkzeuge auf Basis von herstellerspezifischen Datenformaten schon seit längerer Zeit zur Verfügung stellen. Da in diesen Fällen bereits eine bestimmte Kundenbasis besteht, können diese Datenformate nicht einfach auf den noch recht neuen ODX-Standard umgestellt werden, weil hierdurch unter Umständen die ganze Produktinfrastruktur, d. h. alle Tools, die auf der Bedatung basieren, betroffen sind. ODX-Datensätze werden in diesem Fall über Import- und Export-Konverter erzeugt, intern wird aber weiterhin mit dem proprietären Datenformat gearbeitet. Neuere Produkte haben dieses Problem nicht. Sie können ODX als Datenbasis direkt auslesen und abspeichern, ohne dass hierfür Konvertiermechanismen mit ihren unvermeidlichen Kompatibilitätsproblemen notwendig sind.

Ziel der Arbeit mit einem ODX Autorenwerkzeug ist die komfortable, einfache und fehlerfreie Erzeugung der Diagnosebedatung. Auf Basis dieser hohen Ansprüche lassen sich folgende Hauptanforderungen an ein Bedatungswerkzeug ableiten:

- Komfortable und intuitive Bedienoberfläche mit der Möglichkeit, diese an die Bedürfnisse des Benutzers und dessen Wissensstand anzupassen (Bildschirmlayout und Editvorlagen) und Bereitstellung komfortabler Edit- und Suchfunktionen.

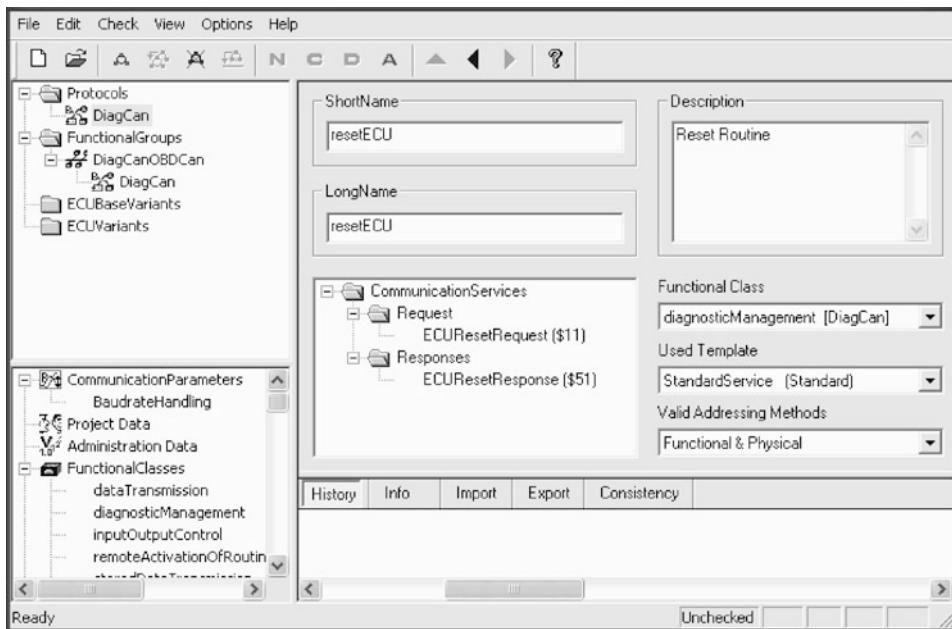


Abb. 9.41 Autorenwerkzeug für ODX-Diagnosedaten (Softing DTS Venice)

- Graphische Darstellung komplexer Bedatungszusammenhänge wie Vererbungshierarchien und Referenzen.
- Überprüfung der editierten Daten und Unterstützung des Benutzers bei der Fehlerbehebung möglichst unmittelbar bei der Dateneingabe, indem die Konsistenz mit den zugehörigen XML-Schemata, ASAM-Prüfregeln und gegebenenfalls auch benutzerdefinierbaren Regeln geprüft wird.
- Import älterer Datenformate, z. B. A2 L, um ältere Daten weiter zu verwenden.
- Unterstützung der Textdaten und Kommentare in verschiedenen Landessprachen, um die Mehrsprachigkeit von Diagnoseanwendungen sicherzustellen.
- Dokumentation der ODX-Daten in *menschenlesbaren* Dokumentenformaten.

9.7 Diagnose-Laufzeitsysteme und OTX Diagnose-Sequenzen

Mit ODX liegt ein vereinheitlichtes, in ISO 22901 (ASAM MCD 2D, siehe Abschn. 6.6) standardisiertes Beschreibungsformat für Diagnosedaten vor. ODX beschreibt den Aufbau von Diagnosediensten mit den jeweiligen Parametern, Umrechnungsmethoden, Einheiten, Varianten usw. Auf Grundlage von ISO 22900 (ASAM MCD 3D, siehe Abschn. 6.7), dem standardisierten Diagnose-Laufzeitsystem (*MVCI-Server* oder *D-Server*) kann ein Diagnosetester aufgebaut werden, der durch diesen ODX-Datensatz konfiguriert wird. Der An-

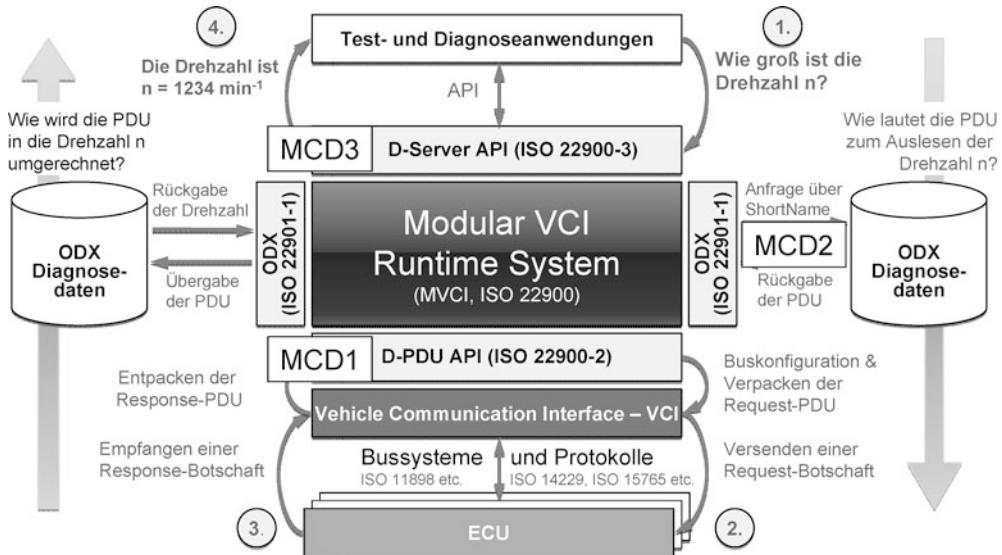


Abb. 9.42 Aufbau und Wirkungsweise eines standardisierten Diagnosesystems

wender dieses Systems muss sich dadurch um den internen Aufbau der Diagnosekommunikation wie die verwendeten Transport- und Diagnoseprotokolle nicht mehr kümmern (Abb. 9.42).

Derartige Diagnose-Laufzeitsysteme werden von verschiedenen Herstellern angeboten, etwa DTS-COS von Softing, PRODIS.MCD von DSA oder die Eigenentwicklungen einiger Fahrzeughersteller. Mit ODX lassen sich jedoch nur die Diagnosedaten beschreiben, die zur Kommunikation eines Diagnosetesters mit einem oder mehreren Steuergeräten notwendig sind.

In der Praxis hat ein Diagnosetester in Entwicklung, Produktion und Service jedoch deutlich mehr Aufgaben. Die Fehlersuche in der Werkstatt erfordert die Verknüpfung der Diagnosedaten mit den Fehlerschanleitungen und Ersatzteildatenbanken des Fahrzeugherstellers. Sämtliche Diagnoseschritte und die Ergebnisdaten müssen protokolliert und verwaltet werden. Nur im Zusammenwirken mehrerer Diagnosendienste werden vollständige Funktionstests möglich, können Fahrzeugkomponenten nach einem Austausch angelernt oder Softwareupdates der Fahrzeugsteuergeräte durchgeführt werden. Im Laufzeitsystem können derartige Diagnoseabläufe zwar als *Jobs* programmiert, aber in ODX nur als *Black Box* beschrieben werden.

Diagnoseabläufe wurden in der Vergangenheit meist innerhalb des Steuergeräte-Lastenhefts in Prosa oder durch Diagramme definiert und dann für verschiedene Zielsysteme in der Entwicklung, der Fertigung und den Werkstätten manuell implementiert. Dies ist weder effizient noch prozesssicher. Erst mit dem *Open Test sequence eXchange* OTX Standard nach ISO 13209, dessen Konzept in Abschn. 6.9 beschrieben wurde, steht

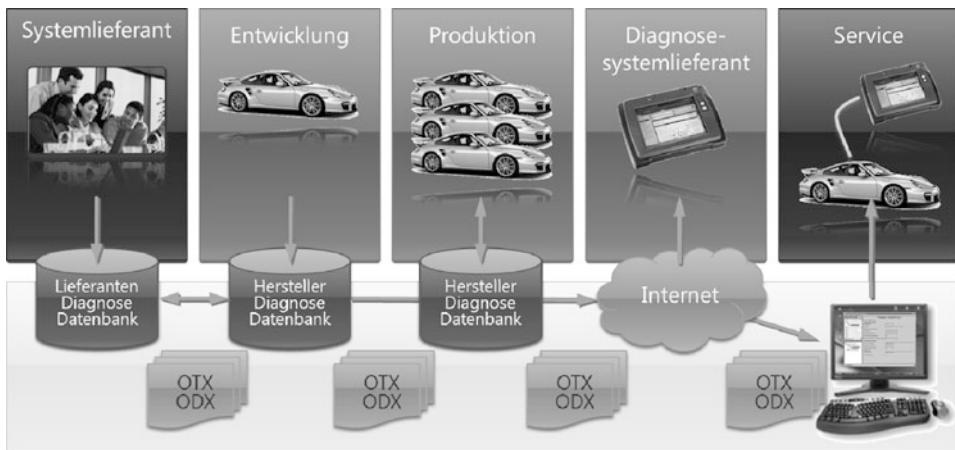


Abb. 9.43 Durchgängiger Datenaustausch in der Diagnoseprozesskette

ein Datenformat zur Verfügung, das die Einzelheiten derartiger Diagnose-Sequenzen standardisiert abbilden kann. Ziel von OTX ist es, Diagnoseabläufe durch einfache Konfiguration des Diagnosetesters statt durch mühsame Programmierung in C/C++ oder Java umzusetzen. OTX kann durchgängig von der Entwicklung, über die Produktion bis zum Service eingesetzt werden (Abb. 9.43). Außerdem kann OTX auch in der Testautomatisierung oder der HIL-Simulation verwendet werden.

9.7.1 Open Test Framework von emotive als OTX Werkzeug

Wie ASAP2, FIBEX oder andere Beschreibungsformate lässt sich auch OTX nur mit Werkzeugunterstützung sinnvoll einsetzen. Eines der ersten hierfür auf dem Markt verfügbaren Werkzeuge ist das *Open Test Framework ODF* der emotive GmbH (Abb. 9.44). ODF ist eine Umgebung für sämtliche Schritte bei der Entwicklung OTX-basierter Diagnoseanwendungen:

- Spezifikation,
- Realisierung, Test und Debugging,
- Dokumentation,
- Ausführen von Diagnoseabläufen.

Wegen der in der Praxis oft heterogenen Werkzeuglandschaft bei vielen Herstellern wurde OTF als offenes, anpass- und erweiterbares System ausgelegt. Anpassungen können vom Anwender bei Bedarf selbst vorgenommen werden.

Open Test Framework bildet die gesamte OTX Architektur (Abb. 9.45) mit entsprechenden Teilwerkzeugen ab.

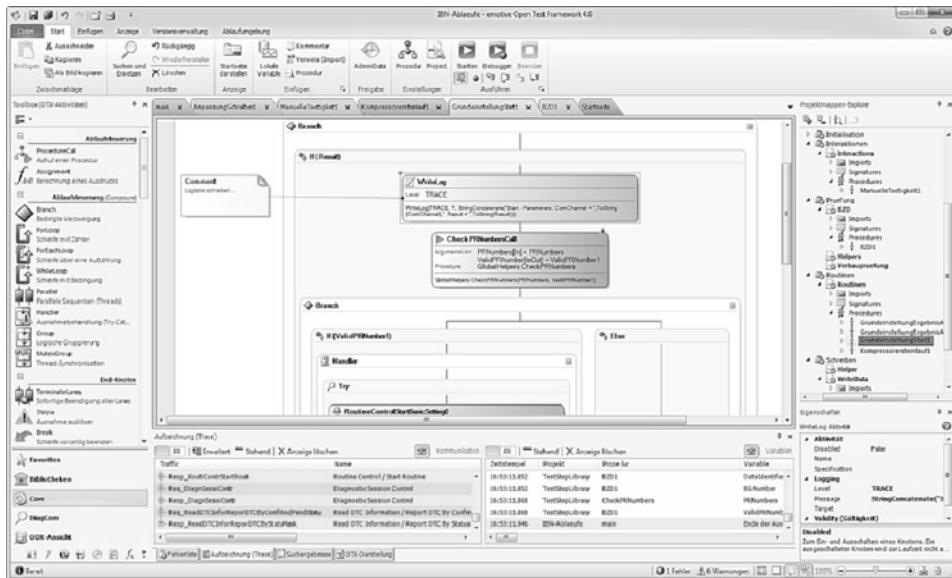


Abb. 9.44 OTX Entwicklungsumgebung *Open Test Framework*

ODF besteht im Wesentlichen aus den in Abb. 9.46 dargestellten funktionalen Elementen, die im Folgenden beschrieben werden.

OTX-Designer Der OTX-Designer ist das zentrale Eingabeelement. Er gibt dem Autor eine graphische Sicht auf die OTX-Daten in Form eines generischen Fluss-Diagramms. Die graphische Darstellung kann unterschiedlich eingestellt werden, so dass sowohl ein Überblick über den Gesamtlauf als auch ein Zoomen in die Details möglich ist. Der OTX-Designer besteht aus Komponenten, die durch den Anwender mit Hilfe des OTX-Designer SDK auch in eigene Anwendungen integriert werden können:

- Workflow-Designer

Im Workflow-Designer werden die Testabläufe graphisch editiert. Ein Ablauf besteht aus verschiedenen Aktivitäten (OTX Nodes), die als Flussdiagramm dargestellt werden. Der Autor erstellt damit die Testlogik in graphischer Form und speichert sie als OTX-Datensatz ab. Da die graphische Darstellung automatisch aus den OTX-Daten erzeugt wird, kann sich der Autor auf die Inhalte konzentrieren und muss sich nicht mit den Layoutdetails der graphischen Darstellung befassen.

- Solution-Explorer

Der Solution-Explorer stellt das gesamte OTX-Projekt in einem Baum dar. Es werden alle Elemente vom *Package* bis hin zur *ActionRealisation* hierarchisch abgebildet. An jedem Knoten können kontextsensitiv Elemente kopiert, ausgeschnitten, eingefügt und gelöscht werden.

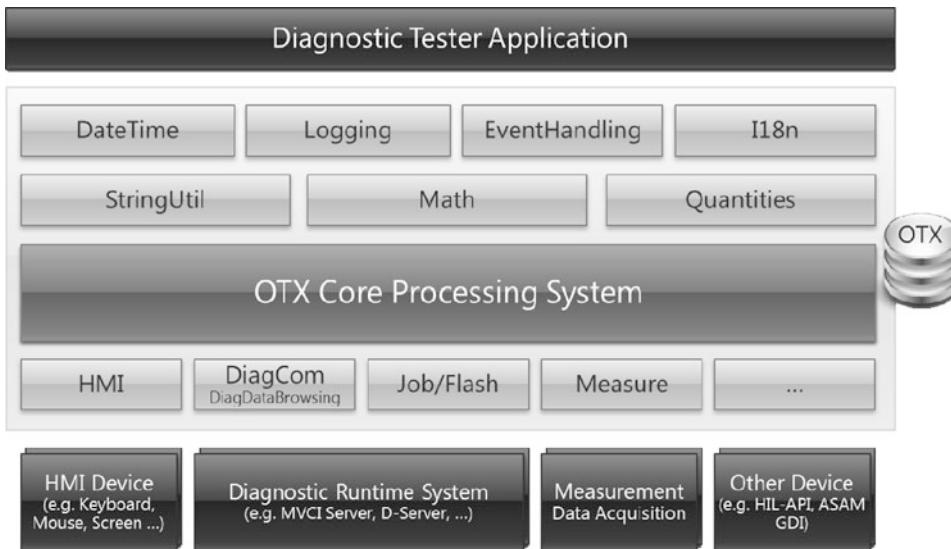


Abb. 9.45 Elemente eines OTX-Systems

- **Toolbox**
Die Toolbox beinhaltet alle Aktivitäten gruppiert nach OTX-Bibliotheken. Der Inhalt der Toolbox (Aktivitäten und Gruppierungen) wird aus einer Datei generiert, die durch den Anwender angepasst werden kann.
- **Ausgabe**
Innerhalb der Ablaufumgebung stellen die Ausgabefenster die Diagnosekommunikation und die Variablenveränderungen dar. Die Trace-Fenster speichern zu jedem Eintrag einen Zeitstempel sowie detaillierte Informationen über das Element. Es gibt einen stehenden und einen rollierenden Modus. Die Daten können in einer Textdatei gespeichert werden.

Forms-Designer Mit dem Forms-Designer lassen sich Bedien- und Ausgabeoberflächen für den Tester erstellen (Abb. 9.47).

OTX-Datenbankmodul Das Datenbankmodul ist für den Zugriff auf die OTX -Daten verantwortlich. Es validiert die Daten, sucht bei Bedarf nach OTX-Objekten und verwaltet Referenzen auf OTX-Objekte. Das Datenbankmodul besteht aus der OTX-API und einer XML-Datenbank. Die OTX-API ist die Schnittstelle für den Anwender.

OTX-Datenbanken können mit über 10000 Abläufen und 1 GB Speicherbedarf sehr groß werden und einen hohen Vernetzungsgrad aufweisen. Um auf diese Daten performant zuzugreifen, wird eine Embedded XML-Datenbank verwendet. Die Kommunikation zwischen OTX-API und XML-Datenbank erfolgt über XQuery-Abfragen (Abb. 9.48). Die Datenbank ist Bestandteil des in ODF enthaltenen OTX-API SDK.

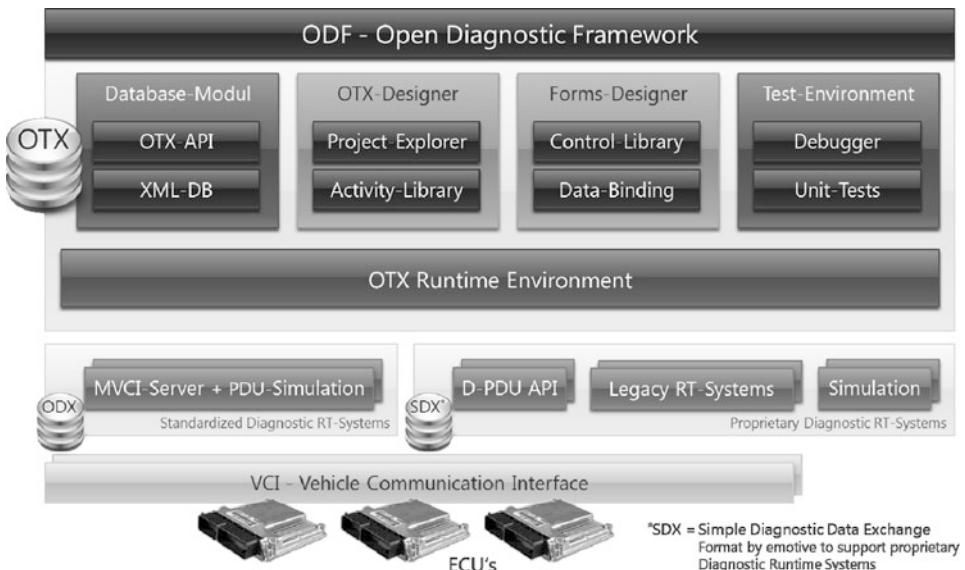


Abb. 9.46 Aufbau des *Open Test Frameworks*

OTX-Testumgebung Die Abläufe können zu Test- und Analysezwecken mit einem Debugger ausgeführt werden. Innerhalb des Debuggers können Haltepunkte (*Breakpoints*) gesetzt, Variablen angezeigt und verändert und der Ablauf schrittweise ausgeführt werden. Diagnosedaten auf Ebene der D-PDU-API können ebenso simuliert werden wie OTX-Aktivitäten, für die es noch keine Implementierung gibt. Neben der reinen Simulation kann ist natürlich auch ein Test gegen echte Hardware möglich.

OTX-Ablaufumgebung In der OTX-Ablaufumgebung können die erzeugten Abläufe sowohl in der Entwicklungsphase als auch als Laufzeitumgebung für den Werkstattbetrieb oder die End-of-Line-Programmierung ausgeführt werden. Für die Abarbeitung wird C#-Code generiert und übersetzt (Abb. 9.48). Der native C#-Code basiert auf der schlanken OTX-Runtime Bibliothek und dem .NET Framework. Im Vergleich zu OTX-Interpreten ist die Ablauflogik dadurch erheblich performanter und platzsparender. Außerdem können fertige Testabläufe als kompilierter Binär-Code sicher an Dritte weitergegeben werden, ohne dass der OTX-Datensatz selbst offen gelegt werden muss.

Die OTX-Laufzeitumgebung kann gängige MVCI-Server integrieren, direkt auf die D-PDU-API zugreifen oder so angepasst werden, dass sie mit proprietären Diagnoselaufzeit-systemen zusammenarbeitet (Abb. 9.46).

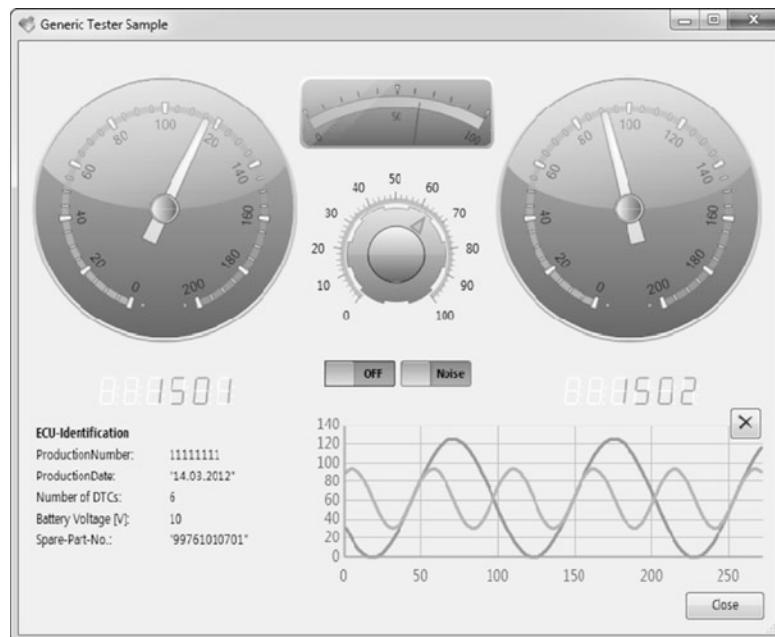


Abb. 9.47 Mit dem Forms-Designer erstellte Bedienoberfläche



Abb. 9.48 Aufbau der OTX-Ablaufumgebung

9.8 Echtzeitverhalten der Steuergeräte-Kommunikation

Unter dem Echtzeitverhalten eines Systems versteht man diejenigen zeitlichen Eigenschaften, die sich aus der Nutzung von Ressourcen in einer realen Systemumgebung ergeben. Ressourcen sind dabei insbesondere Prozessoren und Datenbusse. Ein System ist echtzeitfähig, wenn für jede Funktion ausreichend Rechenzeit und Kommunikations-Bandbreite vorhanden ist, so dass die Steuer- und Regelsignale jeweils zum richtigen Zeitpunkt bzw. innerhalb einer vorgegebenen maximalen Verzögerung zur Verfügung stehen.

Früher hat man das Zeitverhalten mit einfachen Daumenregeln ausreichend gut planen können. Die Auslegung von CAN-Bussystemen beispielsweise galt als akzeptabel, wenn die Buslast der zyklischen Nachrichten nach Gl. 3.7 bei Serienanlauf unterhalb von 30 bis 40 % lag. Ähnlich galten für die CPU-Auslastung nach Gl. 7.1 Werte bis ca. 50 % als unkritisch. Probleme mit dem Echtzeitverhalten konnten mit vertretbarem Aufwand im Rahmen von Integrationstests behoben werden. Durch die wachsende Komplexität und Dynamik der Systeme sind Aufwand, Risiko und Kosten dieses Vorgehens heute nicht mehr vertretbar.

9.8.1 Kennwerte für das Echtzeitverhalten

Das Echtzeitverhalten muss heute frühzeitig im Detail spezifiziert und optimiert werden. Die Verteilung von Funktionen, Budgetierung der Rechenzeit, und die Konfiguration von Betriebssystem- und Busparametern werden vorab durch Worst-Case-Analysen und Simulationen abgesichert. Dazu muss die Dynamik der Systeme berücksichtigt werden, die sich aus der Integration und Vernetzung einer Vielzahl von Software-Funktionen und der realen Fahrzeugumgebung ergibt [1].

Der einfache Bus- bzw. CPU-Last-Ansatz erlaubt heute kaum noch eine zuverlässige Abschätzung der Echtzeitfähigkeit. Steigende Kommunikations-Anforderungen machen es erforderlich, CAN-Busse auch mit 50 % Buslast oder mehr zu betreiben. Zudem steigt mit der Anzahl der *zeitverbrauchenden* Funktionen und Nachrichten auch die gegenseitige Wechselwirkung und führt zu einem deutlich erhöhten Risiko von Echtzeitverletzungen. Dabei sind die Echtzeiteigenschaften der einzelnen Busbotschaften und Softwarekomponenten zu betrachten (Abb. 9.49), nicht nur die Gesamtlast. Folgende Kennzahlen sind dabei relevant:

- Minimale und maximale Antwortzeiten (*Response Times*) von Bus-Nachrichten sowie deren statistische Verteilung.

Die Antwortzeit ist diejenige Zeit, die zwischen dem Erzeugen einer Nachricht im Sende-Steuergerät, z. B. im AUTOSAR COM-Layer, bis zum vollständigen Empfang im Botschaftspuffer der Empfänger verstreicht. Die Antwortzeit enthält neben der Botschaftsdauer die *Arbitrierungszeit*, die sich z. B. bei CAN aus den Botschaftsprioritäten durch die *CAN IDs* sowie den Zykluszeiten ergibt. An dieser Stelle sind die *Scheduling*-Effekte des Busses enthalten (vergl. Abschn. 3.1.7, 3.2.8 und 3.3.6). Die Antwortzeit ist

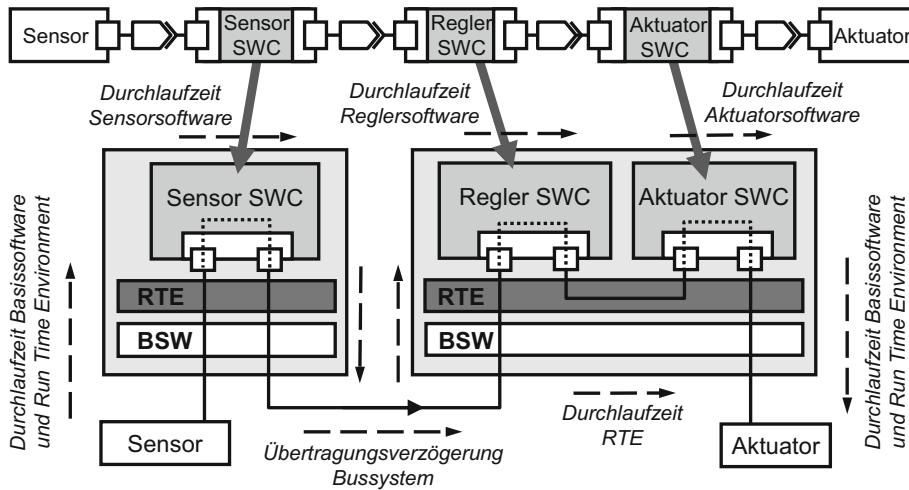


Abb. 9.49 Durchlaufverzögerungen bei einem verteilten AUTOSAR-System

die wichtigste Echtzeit-Kenngröße, da sie unmittelbar mit den Latenz-Anforderungen einer Anwendung (*Deadline*) verglichen werden kann. Beispielsweise könnte gefordert sein, dass die Antwortzeit maximal 50 % der Zykluszeit betragen darf.

- Minimale und maximale effektive Zykluszeiten von Nachrichten (mit Verteilung).
Als effektive Zykluszeiten werden die tatsächlich beobachteten zeitlichen Abstände der periodischen Botschaften auf dem Bus bezeichnet. Als Folge schwankender Arbitrierungs- und Antwortzeiten können die eigentlich konstanten Zykluszeiten ebenfalls schwanken (*Jitter*). Ähnlich wie bei den Antwortzeiten gibt es auch hier oft Vorgaben aus der Funktionsentwicklung oder allgemeine Entwurfsregeln, z. B. Jitter maximal 20 % der Zykluszeit.
- Latenzen von Ende-zu-Ende-Wirkketten (Minimum, Maximum und Verteilung).
Die Ende-zu-Ende-Wirkketten-Latenzen sind den Antwortzeiten sehr ähnlich, enthalten jedoch weitere Verzögerungen in der komplexen Kommunikations-Kette, beispielsweise Softwareanteile des AUTOSAR RTE- bzw. BSW-Layers beim Bündeln und Extrahieren von Signalen in Busbotschaften sowie Verzögerungen durch Gateways. Da die Steuergerät-Software *Multitasking* verwendet, sind hierbei auch die *Scheduling*-Effekte des Betriebssystems zu berücksichtigen (vergl. Abschn. 7.2.5). Auch Diagnose-Sequenzen unterliegen oftmals harten Echtzeitanforderungen, z. B. bei der Abgas-OBD, wo zwischen Diagnose-*Request* und *Response* maximal 50 ms liegen dürfen. Durch die Ende-zu-Ende-Latenzen werden also Verzögerungen auf Anwendungsebene erfasst.

Da sich Gesamtverzögerungen oft aus vielen Anteilen zusammensetzen, entstehen unübersichtliche Toleranzketten. Manuelle Abschätzungen können dabei nur grobe, extrem

pessimistische Anhaltswerte liefern, die zu einer unnötigen Überdimensionierung und einer Fehleinschätzung der Sicherheitsreserven des Systems führen würden. Die praktische Analysearbeit kann deshalb nur mit Werkzeugunterstützung wie dem nachfolgend beschriebenen *SymTA/S* sinnvoll durchgeführt werden.

9.8.2 Echtzeitanalyse mit *SymTA/S* von Symtavision

Das Werkzeug *SymTA/S* von Symtavision bietet für detaillierte Echtzeitanalysen eine geeignete Unterstützung für CAN, LIN, FlexRay und Ethernet-Systeme. Im ersten Schritt einer Analyse kann *SymTA/S Network* eine bereits vorhandene Bus-Konfiguration in verschiedenen Formaten (DBC, FIBEX, AUTOSAR XML) importieren und automatisch untersuchen. Die Ergebnisse umfassen die relevanten Echtzeit-Kenngrößen bis hin zum detaillierten Zeitverhalten einzelner Botschaften:

- Buslast gesamt und Anteile der einzelnen Botschaften,
- Minimal und maximal mögliche Latenzen der Botschaften sowie deren statistische Verteilung unter Berücksichtigung der Arbitrierung durch das CAN-Protokoll,
- Automatische Überprüfung der Zykluszeiten und weiteren Randbedingungen (*Timing Constraints* wie *Deadlines*),
- Visualisieren der kritischen Arbitrierungssequenzen durch *Gantt Charts*.

Abbildung 9.50 zeigt beispielhaft ein Analyseergebnis von *SymTA/S*. Man sieht neben dem Strukturbau des Systems zunächst die Lastverteilung, die minimalen und maximalen Latenzen (*Response Times*) mit Markierung von Deadline-Verletzungen und daneben die statistische Verteilung der Latenzen. Darunter erfolgt die Detaillierung für einen einzelnen Frame: links die Wahrscheinlichkeitsverteilung der Latenzen, rechts die zur links markierten Verteilungsklasse gehörende spezifische Arbitrierungssequenz. Diese Auswertungen können als PDF-Report oder als Excel-Tabelle exportiert werden.

Über die Analyse hinaus bietet *SymTA/S* umfassende Möglichkeiten zur Systemoptimierung. Das Umverteilen (*Mappen*) von Signalen auf andere Frames, Veränderung von CAN IDs oder die Optimierung der Sende-Offsets periodischer Frames zwecks Entzerrung der Buslast sind manuell und automatisch möglich. Im letzteren Fall werden Freiheitsgrade und Optimierungsziele vorgegeben. Das Werkzeug findet automatisch die interessantesten Konfigurationen und präsentiert diese dem Entwickler als Entscheidungsvorlage. Analog können Änderungswünsche (*Change Requests*) nach Serienanlauf analysiert werden oder das Potential eines Entwurfs für zukünftige Erweiterungen mit neuen Signalen und Botschaften abgeschätzt werden.

SymTA/S hilft aber nicht nur beim Systementwurf sondern auch bei der Überprüfung der Implementierung. Dazu werden Messdaten aus den Integrationstests, die z. B. mit *CA-Noe* aufgezeichnet wurden, in den *Symtavision TraceAnalyzer* eingelesen und mit den Entwurfsdaten verglichen. In der Praxis zeigt sich nämlich, dass Zeitanforderungen bei der

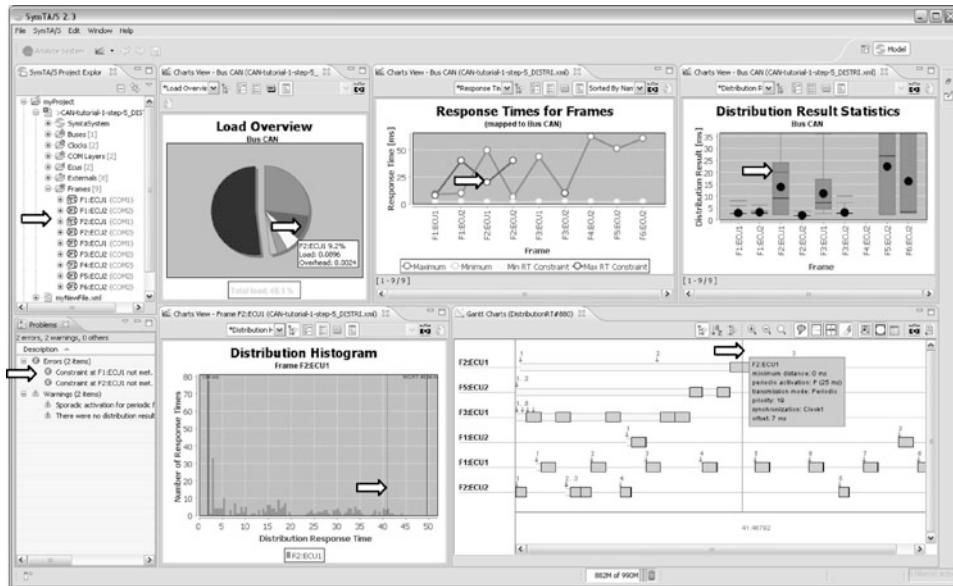


Abb. 9.50 Schedulinganalyse-Werkzeug *SymTA/S* von Symtavision

Softwareimplementierung oft nur schwer zu berücksichtigen sind und sich daher Abweichungen gegenüber dem Entwurf ergeben. Solche Abweichungen können durch Integration von *SymTA/S* in die Testumgebung automatisiert erkannt und geeignete Anpassungsmaßnahmen in der Implementierung bzw. im Timing-Modell abgeleitet werden.

Ergänzend zur Netzanalyse bietet *SymTA/S ECU* vergleichbare Analysen für die Softwareintegration auf Steuergeräten inklusive Multi-Core ECUs. Dabei wird die Steuergerätekonfiguration mit Hilfe der bekannten OSEK OIL oder AUTOSAR XML-Datensätze in das Analysewerkzeug importiert. Informationen über die Laufzeiten der Softwaremodule können aus Mess- und Debugging-Werkzeugen oder statischen Code-Analyse-Tools gewonnen werden.

SymTA/S ECU und *SymTA/S Network* ergänzen sich zu einer durchgängigen Werkzeugkette für die Systemanalyse und Optimierung. Durch die modellbasierte Analyse kann *SymTA/S* Echtzeitfehler schon früh beim System- und Netzentwurf und nicht erst nach der Implementierung bei den Integrationstests am Ende der Prototypenentwicklung aufdecken und vermeiden helfen. *SymTA/S* liefert Antworten auf folgende Kernfragen im Design verteilter Steuergerätesysteme:

- Welche Botschaften können aufgrund von Ressourcen-Knappheit verloren gehen bzw. ihre Zykluszeit oder *Deadline* verpassen? Wie oft kann dies passieren? In welchen Situationen treten die Fehler auf?

- Welche Botschaften erfüllen die Anforderungen zwar noch, nutzen den zulässigen Grenzwerte aber nahezu aus und können bei Ausnahmesituationen oder späteren Änderungen zu Echtzeitproblemen führen?
 - Wie kann das System optimiert werden, um die Reserven und Grenzen des Systems auszudehnen?
-

Literatur

[1] M. Traub, V. Lauer, T. Weber, M. Jersak, K. Richter, J. Becker: Timing-Analysen für die Untersuchung von Vernetzungsarchitekturen. ATZelektronik, Heft 3/2009, S. 36 ff

Spätestens seit automatische Mautsysteme drahtlose Funkverbindungen nutzen, wird diskutiert, wie mit Hilfe von direkten Datenverbindungen zwischen Fahrzeugen (*Car to Car Communication C2C* bzw. *Vehicle to Vehicle V2V*) sowie zwischen Fahrzeugen und Verkehrsleiteinrichtungen am Straßenrand (*Car to Infrastructure C2I* bzw. *Vehicle to Infrastructure V2I*) Verkehrssicherheit und Verkehrsfluss verbessert werden können. Solche Telematiksysteme gelten als Zukunftstechnik [1, 2].

10.1 Mautsysteme

Beim deutschen Mautsystem *Toll Collect* (Abb. 10.1) bestimmt ein fahrzeugseitiges Steuergerät, die *On-Board Unit OBU* mit Hilfe des Satellitennavigationssystems *GPS*, eines Kreiselsystems (*Gyro*) und des Tachometersignals die aktuelle Position des Fahrzeugs und übermittelt bei mautpflichtigen Straßen Informationen zur gefahrenen Strecke über eine konventionelle GSM-Mobilfunkverbindung an die Abrechnungszentrale. Zur Überwachung befinden sich auf den Straßen in größeren Abständen sogenannte Kontrollbrücken (*Road Side Unit RSU*), die die Fahrzeuge erfassen und über eine Infrarot-Verbindung nach ISO TC 204 (*Dedicated Short Range Communication DSRC*) mit dem Fahrzeug kommunizieren. An Stellen, an denen mautpflichtige und mautfreie Straßen so nah beieinander verlaufen, dass GPS keine sichere Identifikation der Fahrstrecke gewährleistet, befinden sich als weitere RSU-Einheiten *Stützbaken* am Straßenrand, die über die genannte DSRC-Infrarot-Verbindung zusätzliche Positionsinformationen übermitteln. In anderen europäischen Ländern verzichtet man zum Teil auf die GSM-Verbindung bzw. die GPS-Positionsbestimmung und verbaut an den mautpflichtigen Strecken in kurzen Abständen straßenseitige RSU-Einheiten zur Positionsbestimmung und Abrechnung. Statt einer Infrarot-Datenverbindung wird dort häufig eine Kurzstreckenfunkverbindung nach CEN TC 278 verwendet.

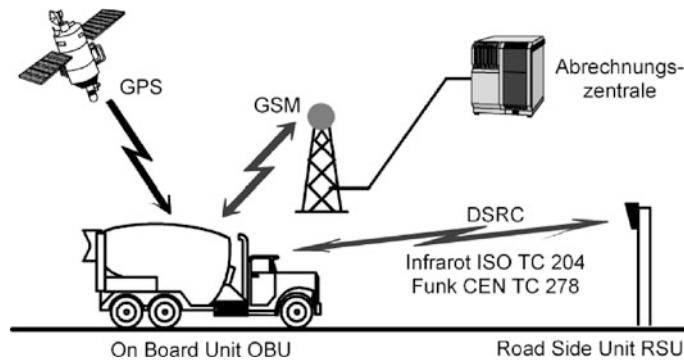


Abb. 10.1 Kommunikationsverbindungen bei Mautsystemen wie *Toll Collect*

10.2 Car2Car-Konsortium und Vehicle2X-Kommunikation

Seit einigen Jahren haben sich verschiedene Fahrzeughersteller und Zulieferer im *Car2Car-Konsortium* zusammengeschlossen, um Spezifikationen für die Kommunikation zwischen Fahrzeugen zu erarbeiten. *Car2Car-Konsortium* ist das europäische Pendant zu ähnlich gelagerten amerikanischen Aktivitäten, die dort unter dem Schlagwort *Vehicle Safety Communications VSC* von amerikanischen und japanischen Herstellern aufgenommen wurden. In beiden Fällen soll sich ein Fahrzeug mit anderen Fahrzeugen in Funkreichweite ad hoc zu einem lokalen Netzwerk zusammen schließen (Abb. 10.2) und mit diesen sicherheitsrelevante Daten austauschen. Mögliche Funktionen sind:

- Kollisionswarnung bei hinter- und nebeneinander fahrenden Fahrzeugen sowie an Einmündungen, Kreuzungen und beim Überholen,
- Aufprallwarnung bei Unfällen noch vor den üblichen Crash-Sensoren,

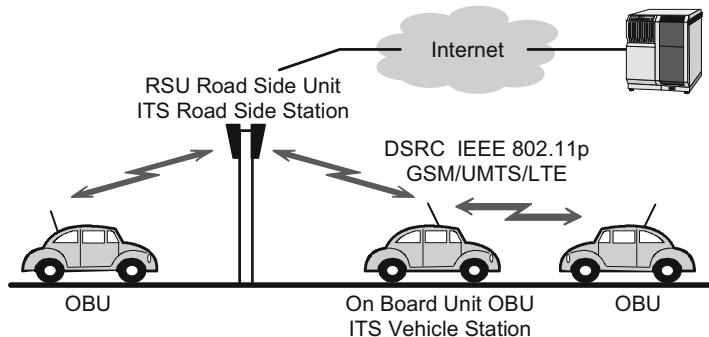


Abb. 10.2 Vorschlag des Car2Car-Konsortiums

- Warnung vor kritische Situationen, z. B. Stauende in einer unübersichtlichen Kurve, Bauarbeiten, Ölspuren usw.,
- Verbesserte Routenwahl und Navigation durch aktuellere Verkehrsinformationen und Abstimmung der Routenwahl über viele Fahrzeuge hinweg,
- Unterstützung von „Grünen Wellen“,
- Einfädelhilfe bei Autobahneinfahrten oder Engstellen,
- Ferndiagnose im Pannenfall.

Die straßenseitigen Einheiten können zur Verlängerung der Reichweite zwischen den Fahrzeugen sowie zur Anbindung an Navigations- und Verkehrsleitrechner eingesetzt werden. Außerdem sollen die Fahrzeuge darüber Kontakt zum Internet aufnehmen können, über das sowohl Verkehrsleitinformationen als auch gewöhnliche Internet-Anwendungen (*Infotainment*) abgewickelt werden sollen. Die Integration gewöhnlicher Internet-Anbindungen in das Konzept und der Einsatz einer WLAN-artigen Funkschnittstelle gelten als Schlüsselemente für eine schnelle Marktdurchdringung, ohne die solche Systeme nicht funktionsfähig sind.

Alle derartigen Aktivitäten befinden sich noch im Forschungs- und Vorentwicklungsstadium. Spezifikationen und Normentwürfe sind vielfach noch im Fluss. Daher soll nur auf die Problemstellung, aber nicht auf technische Details eingegangen werden.

Am weitesten gediehen ist die Festlegung der Funkschnittstelle. Die Funkübertragung (*Dedicated Short Range Communication*) soll im Bereich um 5,9 GHz stattfinden, wobei ein Teil des Frequenzbandes exklusiv für sicherheitskritische Übertragungen reserviert wird. Die Zuteilung des Frequenzbandes durch die ETSI ist in EN 302 571 geregelt. Die Funkschnittstelle baut auf der konventionellen WLAN-Technologie nach IEEE 802.11 auf, wie sie etwa in Notebooks üblich ist. Aufgrund der besonderen Problematik muss die Schnittstelle allerdings modifiziert werden (IEEE 802.11p und IEEE 1609). Gegenüber einem quasi-stationären WLAN ergeben sich folgende zusätzlichen Anforderungen:

- Die Fahrzeuge bewegen sich sehr schnell gegeneinander. Das System soll für Fahrgeschwindigkeiten von bis zu 250 km/h ausgelegt werden, d. h. Relativgeschwindigkeiten im Begegnungsverkehr bis zu 500 km/h.
- Da sich die Fahrzeuge schnell annähern und wieder entfernen, sind keine langwierigen Authentifizierungs-, Login- und Logout-Prozeduren bei der Verbindungsaufnahme und beim Verbindungsabbau möglich. Das System muss lokal mit einer sich schnell verändernden Zahl von Teilnehmern ohne zentrale Verwaltungsinstanz zureckkommen (*Vehicle Ad hoc-Netzwerk VANET*).
- Trotzdem müssen sich die Teilnehmer in geeigneter Form identifizieren, damit das System nicht so manipuliert werden kann, dass ein „Störsender“ böswillig andere Fahrzeuge zum Beschleunigen oder Bremsen auffordern kann.
- In Stausituationen kann die Fahrzeugdichte sehr hoch sein, während zu anderen Zeiten die Fahrzeuganzahl gering und der Fahrzeugabstand groß sein werden. Da die Funkleistung der Fahrzeuge auf 2 W limitiert werden soll, woraus sich auch bei freier Sicht nur

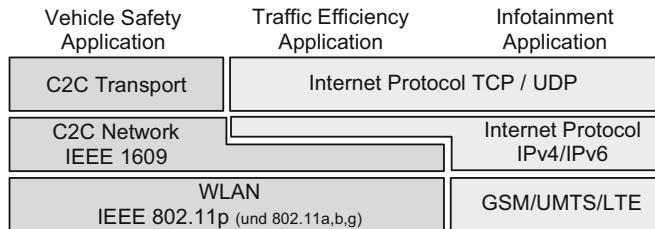


Abb. 10.3 Schichtenmodell des Car2Car-Konsortiums

eine Reichweite von einigen Hundert Metern ergibt, muss jeder Teilnehmer gleichzeitig auch Routing-Aufgaben zwischen den verschiedenen Fahrzeugen und zu den statio-nären Funkbaken (*Road Side Unit*) übernehmen.

Car2Car schlägt einen Dreifach-Protokollstapel vor (Abb. 10.3). Klassische Internet-Anwendungen (*Infotainment*) sollen auch im Fahrzeug die üblichen Internet-Protokolle TCP/IP, UDP/IP usw. einsetzen und im Regelfall die bereits vorhandenen GSM, LTE und UMTS-Funkdienste verwenden. Falls freie Kapazität und genügend viele RSUs vorhanden sind, können auch die IEEE 802.11p-Verbindungen genutzt werden. Ob und wie ein Roaming zwischen LTE/UMTS/GSM und WLAN möglich sein wird, ist offen. Aufgaben der Verkehrsleitung (*Traffic Efficiency*) werden voraussichtlich auch mit TCP/IP arbeiten, kommunizieren aber über das WLAN-Netz. Für die Fahrsicherheitsfunktionen (*Vehicle Safety*) scheint TCP/IP nicht geeignet. Hierfür soll ein C2C spezifischer Protokollstapel entwickelt werden. Der neue *Transport Layer* soll das Multiplexen von Daten sowie eine gesicherte, verbindungsorientierte Übertragung ermöglichen. Der *Network Layer* wird die Aufgabe übernehmen, Botschaften über mehrere Fahrzeuge hinweg weiterzuleiten (*Multi Hop Connections*) und soll einen Adressierungsmechanismus bieten, der auf der Position der jeweiligen Teilnehmer basiert (*Geographische Adressierung*). Die Parallelität zwischen sicherheitskritischen Botschaften, die mit hoher Dringlichkeit übertragen werden müssen, und solchen, die weniger zeitkritisch sind, schlägt bis zum Funksystem durch. Dort sieht IEEE 1609 vor, parallel zum normalen Datenkanal einen Kontrollkanal einzurichten, auf dem die Übertragung der sicherheitskritischen Daten garantiert werden kann. Ob dies durch reinen Zeitmultiplex-Betrieb darstellbar ist oder ob ein teurer doppelkanaliger Funkempfänger eingebaut werden muss, muss die Praxis zeigen.

Die Standardisierungsarbeiten werden in Europa vom *European Telecommunications Standards Institut ETSI* koordiniert. EN 302 665 beschreibt die Gesamtarchitektur (Abb. 10.4) eines *Intelligent Transport Systems ITS* und EN 102 637 und EN 102 638 typische Anwendungsszenarien wie Maßnahmen zur Verbesserung der Verkehrssicherheit (*Active Road Safety*) oder zur Verbesserung des Verkehrsflusses (*Traffic Efficiency*). Zu den bisher definierten Konzepten gehört, dass alle Teilnehmer (*ITS-Stations*), d. h. sowohl Fahrzeuge als auch die straßenseitigen Stationen, periodisch alle 0,1 ... 1 sec eine *Coopera-*

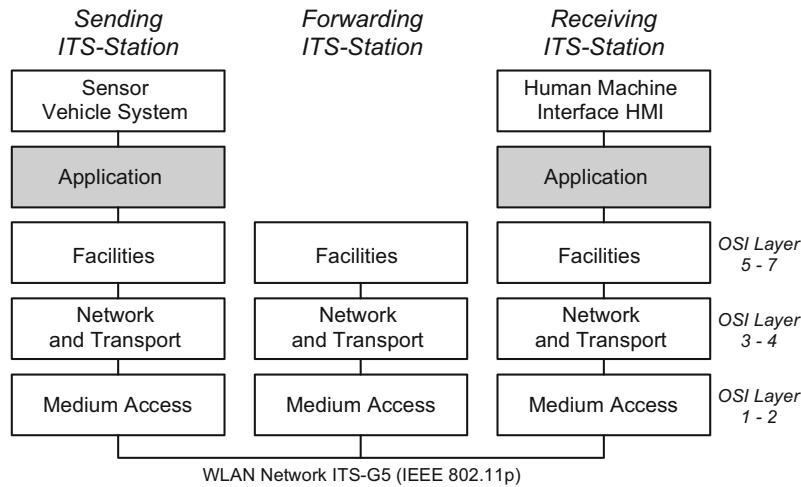


Abb. 10.4 ITS Kommunikation nach EN 302 665

tive Awareness Message CAM aussenden, so dass alle Teilnehmer wissen, welche Stationen im direkten Umfeld vorhanden sind. CAM Botschaften enthalten folgende Inhalte:

- *ITS PDU Header* mit Protokollversion, Botschaftskennung (*Message ID*) und Zeitstempel
- Eindeutige Stationskennung, geographische Position des Fahrzeugs (GPS Daten) und Fahrzeugart. Dabei wird zwischen normalen Fahrzeugen, Sonderfahrzeugen wie Polizei, Feuerwehr und Krankenwagen, öffentlichem Nahverkehr, z. B. Omnibussen, sowie straßenseitigen Stationen unterschieden.
- Fahrzeuge teilen ihre Fahrgeschwindigkeit, Beschleunigung und Fahrtrichtung sowie ihre Abmessungen mit. Sonderfahrzeuge zusätzlich, ob sie im Notfalleinsatz sind. Bei Messdaten, die größere Ungenauigkeiten aufweisen können, wird eine Genauigkeitsschätzung (*Confidence Interval*) mitgeliefert.

Im Fall einer kritischen Verkehrssituation (*Road Hazard Warning*), z. B. bei einer Notbremsung, einem Liegenbleiber oder einem Geisterfahrer, senden Fahrzeug, die die Situation erkennen, eine *Decentralized Environmental Notification Message DENM*. Diese wird periodisch wiederholt, solange die Situation andauert. Die DENM Botschaft enthält folgende Informationen:

- *ITS PDU Header* wie bei CAM Botschaften. Zusätzlich wird ein Update-Zähler übertragen, der eine eindeutige Kennung für das kritische Ereignis enthält und anzeigt, wie oft die Botschaft schon wiederholt wurde. Außerdem wird die voraussichtliche zukünftige Wiederholfrequenz angekündigt und ein Ablaufzeitpunkt mitgeteilt, zu dem die Botschaft ungültig werden soll, falls sie nicht wiederholt wird. Der Sender kann die Verläs-

lichkeit (*Reliability*) seiner Information bewerten, falls die auslösende Verkehrssituation nicht eindeutig erkannt werden kann, oder mitteilen, dass das auslösende Ereignis aus seiner Sicht nicht mehr existiert.

- Im *Situation Container* wird die auslösende Ursache der Warnmeldung beschrieben. Vorgesehen sind u. a. Notbremsungen, auf der Fahrbahn stehende oder außergewöhnlich langsam fahrende Fahrzeuge, Kollisionswarnungen, vereiste Straßen oder schlechte Sichtverhältnisse. Weiterhin wird eingeschätzt, wie kritisch die Verkehrssituation ist (*Severity*) und gegebenenfalls auf andere DENM Botschaften verwiesen, falls ein Fahrzeug die Warnung weiterer Fahrzeuge bestätigen oder ergänzen will.
- Im *Location Container* wird die geographische Position der kritischen Verkehrssituation sowie gegebenenfalls des eigenen Fahrzeugs mitgeteilt. Positionsangaben können dabei neben GPS-Koordinaten auch aus der Angabe einer kritischen Zone, z. B. einer rechteckigen oder elliptischen Fläche bestehen.

10.3 Normen und Standards zu Kapitel 10

Toll-Collect	Toll-Collect Basiswissen. Pressemappe, www.toll-collect.de ISO Technical Committee TC 204 – Intelligent Transport Systems. www.iso.org CEN Technical Committee TC 278 – Road Transport and Traffic Telematics. www.cen.eu European Telecommunications Standards Institute ETSI Technical Committee for Intelligent Transport Systems TC ITS, www.etsi.org
--------------	--

Car2X	Car 2 Car Communication Consortium Manifesto: Overview of the C2 C-CC system V1.1, 2007, www.car-to-car.org
Vehicle2X	IEEE 802.11 Wireless LAN (WLAN) Specification, 2007
	IEEE 802.11p Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, Amendment 6: Wireless Access in Vehicular Environments (WAVE), 2010
	IEEE 1609 Standards for Wireless Access in Vehicular Environments (WAVE), 2010, www.ieee.org
	SAE J2735 Dedicated Short Range Communications (DSRC), 2009, www.sae.org
	ISO 21217 Intelligent Transport Systems – Communications access for land mobiles (CALM), 2010, www.iso.org
	European Telecommunications Standards Institute ETSI Technical Committee for Intelligent Transport Systems TC ITS: Technical Characteristics for Pan-European Harmonized Communications Equipment for Road Safety and Traffic Management (Draft) – Part 1 Road-Safety Applications, 2005. Part 2 Non Safety Related Applications, 2006. www.etsi.org , inzwischen umgesetzt in
	EN 302 571 Intelligent Transport Systems ITS: Radio Communications Equipment in the 5.9 MHz Frequency Band. 2013, www.etsi.org
	EN 302 665 Intelligent Transport Systems ITS: Communications Architecture, 2010, www.etsi.org
	TS 102 638 Intelligent Transport Systems ITS: Basic Set of Applications, Definitions, 2009, www.etsi.org
	TS 102 637 Intelligent Transport Systems ITS: Basic Set of Applications, 2010, www.etsi.org
	TS 102 636 Intelligent Transport Systems ITS: Geo Networking, 2010, www.etsi.org

Literatur

- [1] J. Eberspächer: Das vernetzte Automobil. Hüthig Verlag, 1. Auflage, 2008
- [2] Europäische Kommission: Mobility and Transport – Intelligent Transport Systems. ec.europa.eu/transport/themes/its

Web-Adressen

Webseite zu diesem Buch: www.hs-esslingen.de/~zimmerma/automotive

Bussysteme

CAN, TTCAN	www.can.bosch.com www.can-cia.de www.odva.org www.kvaser.se www.honeywell.com	Bosch CAN in Automation Device Net CAN Kingdom CAN SDS
DSI	www.dsiconsortium.org	Distributed Systems Interface
FlexRay	www.flexray.com	FlexRay Consortium
LIN	www.lin-subbus.org	LIN Consortium
MOST	www.mostcooperation.com www.smsc-ais.com	MOST Cooperation SMSC (früher OASIS)
PSI5	www.psi5.org	Peripheral Sensor Interface
TTP	www.ttagroup.org www.tttech.com	Time Triggered Architecture Group TTTech
Ethernet	www.ieee.org www.opensig.org www.avnu.org	IEEE Standards Association OPEN Alliance Special Interest Group AVnu Alliance

Standardisierungsgremien und Arbeitskreise

ANSI	www.ansi.org	American National Standards Institute
ASAM/ASAP	www.asam.net	Association for Standardization of Automation and Measuring Systems
AUTOSAR	www.autosar.org	Automotive Open Systems Architecture
Car2Car	www.car-to-car.org	Car 2 Car Communication Consortium
CEN	www.cen.eu	Comité Européen de Normalisation
CE4 A	www.ce4a.org	Consumer Electronics for Automotive
ETSI	www.etsi.org	European Telecommunications Standards Institute

Standardisierungsgremien und Arbeitskreise

HIS	www.automotive-his.de	Herstellerinitiative Software
IEC	www.iec.ch	International Electrotechnical Commission
IETF	www.ietf.org	Internet Engineering Task Force
ISO	www.iso.org	International Organisation for Standardization
JASPAR	www.jaspar.jp/english	Japanese Automotive Software Platform Architecture (Informationen zum Teil nur in Japanisch verfügbar)
MSR	www.msr-wg.de	MSR-Konsortium (mittlerweile als ASAM AAS in ASAM aufgegangen)
OSEK/VDX	www.osek-vdx.org	Offene Systeme für die Elektronik im Kraftfahrzeug/Vehicle Distributed Executive
SAE	www.sae.org	Society of Automotive Engineers
UML	www.uml.org	Unified Modeling Language
VDA FAKRA	www.vda.de	Verband der Kraftfahrzeugindustrie, Normenausschuss Kfz (FAKRA)

Zulieferer für Steuergerätesoftware: Entwicklung und Tools (Auswahl)

AbsInt	www.absint.de	AbsInt Angewandte Informatik GmbH
AETA RICE	www.aeta-rice.com	Automotive Engineering Tool Alliance
Bosch BEG	www.bosch-engineering.de	Bosch Engineering GmbH
Bose	www.bose.de	Bose Automotive GmbH
Broadcom	www.broadcom.com	Broadcom Corporation
CarMediaLab	www.carmedialab.de	Car Medialab GmbH
Carmeq	www.carmeq.com	Carmeq GmbH
dSpace	www.dspace.com	dSpace GmbH
Elektrobit	www.elektrobit.com	Elektrobit Automotive GmbH (früher: 3Soft GmbH und Decomsys)
Emotive	www.emotive.de	emotive GmbH & Co. KG
ETAS	www.etas.de	ETAS GmbH
GIGATRONIK	www.gigatronik.com	GIGATRONIK GmbH
Göpel	www.goepel.com	Göpel electronic GmbH
Harman Becker	www.harmanbecker.com	Harman/Becker Automotive Systems GmbH
Eberspächer	www.eberspaecher.com	Eberspächer Electronics GmbH
I+ME Actia	www.ime-actia.de	I+ME Actia GmbH
IAV	www.iav.com	IAV GmbH Ing.ges. Auto und Verkehr
In2Soft	www.in2soft.de	In2Soft GmbH
Intrepid	www.intrepidcs.com	Intrepid Control Systems Inc.
IXXAT	www.ixxat.de	IXXAT Automation GmbH

Zulieferer für Steuergerätesoftware: Entwicklung und Tools (Auswahl)

KOPF	www.kopfweb.de	KOPF GmbH
MBtech	www.mbtech-group.com	MBtech group
Lipowsky	www.canspy.com	Lipowsky Industrie-Elektronik GmbH
Mentor (Volcano)	www.mentor.com/solutions/automotive	Mentor Graphics (früher Volcano Communications)
mm-lab	www.mmlab.de	mm-lab GmbH
Ontorix	www.ontorix.com	Ontorix GmbH
Peak	www.peak-system.com	PEAK-System Technik GmbH
RA Consult.	www.rac.de	RA Consulting GmbH
RELNETYX	www.relnetyx.com	RELNETYX AG
Samtec	www.samtec.de	Samtec Automotive Software & Electronics GmbH
SMART IN NOVATION	www.smart-in-ovation.de	SMART IN OVATION GmbH
SMART Testsolutions	www.smart-testsolutions.de	SMART Testsolutions GmbH
Softing	www.softing.com	Softing AG
STZ RE	www.stz-rechnereinsatz.de	Steinbeis TZ Rechnereinsatz Esslingen
Symtavision	www.symtavision.com	Symtavision GmbH
Tieto	www.tieto.de	Tieto Deutschland GmbH
Vector	www.vector-informatik.de	Vector Informatik GmbH

Zeitschriften und Portale

Hanser	www.hanser-automotive.de	Hanser automotive electronics
Vieweg	www.atzonline.de	ATZ elektronik
Weka	www.elektroniknet.de	Elektronik automotive

Abkürzungen

ACK	Acknowledged, Not Acknowledged
NAK	Positive bzw. negative Empfangsbestätigung
A2L	ASAM Beschreibungsformat für Applikationsdaten
AE	Automotive Electronics Neue Bezeichnung des ASAM Arbeitsgebiets MCD
API	Application Programming Interface Programmierschnittstelle
AUTOSAR	Automotive Open Systems Architecture Herstellerinitiative zur Standardisierung von Software im Kfz
AVB	Audio Video Broadcasting Echtzeitfähige Audio- und Videoübertragung über Ethernet und Internet
Big Endian	Reihenfolge der Bytes bei Daten, die mehrere Bytes lang sind. Beim
Little Endian	Big-Endian-Format beginnen die Daten beim höchstwertigsten Byte (Most
Endianess	Significant Byte) und enden mit dem niederwertigsten Byte (Least Significant Byte). Beim Little Endian-Format ist es umgekehrt.
BS	Block Size
BSW	Basic Software Betriebssystemkern und Dienstprogramme
C2C	Car to Car, Car to Infrastructure
C2I	Kommunikation zwischen Fahrzeugen und Verkehrsleiteinrichtungen
CAN	Controller Area Network Weit verbreitetes, ereignisgesteuertes Kfz-Bussystem
CARB	California Air Resources Board Kalifornische Umweltbehörde, legt u. a. die zulässigen Abgasemissionen von Kraftfahrzeugen fest.
CCP	CAN Calibration Protocol Verbreitetes Busprotokoll in der Applikation von Elektroniksystemen im Kraftfahrzeug, von ASAM spezifiziert
CDF	Calibration Data Format ASAM Beschreibungsformat für Applikationsdaten
CHI	Controller Host Interface Schnittstelle zwischen Kommunikationscontroller und steuerndem Mikrocontroller

COM	Communication Spezifikation für die steuergeräteinterne und externe Kommunikation im Rahmen von OSEK/VDX
CPU	Central Processing Unit, Microcontroller Unit
MCU	Mikroprozessor, Mikrocontroller, Hostprozessor
CRC	Cyclic Redundancy Check Algorithmus zur Berechnung von Prüfsummen zur Fehlererkennung bei Datenübertragungen
CSMA	Carrier Sense Multiple Access, Collision Detect, Collision Avoidance
CSMA/CD	Zugriffverfahren, bei dem ein Sender vor dem Buszugriff prüft, ob nicht bereits ein anderer Sender Daten überträgt und gesendete Daten mitliest, um Kollisionen zu erkennen bzw. zu vermeiden.
DL	Data Length, Data Length Code
DLC	Länge einer Botschaft, in der Regel Anzahl der Nutzdatenbytes
D-Server-API	Diagnosis Server Application Programming Interface Diagnosis Protocol Data Unit Application Programming Interface
D-PDU-API	Programmierschnittstellen eines ASAM-Diagnosesystems
DoCAN	Diagnostic communication over CAN
DoFR	Diagnostic communication over FlexRay
DoEth	Diagnostic communication over Ethernet
DoIP	Diagnostic communication over Internet Protocol
DoK-Line	Diagnostic communication over K-Line
DSRC	Dedicated Short Range Communication Kurzstrecken-Datenverbindung per Funk oder Infrarot
DTC	Diagnostic Trouble Code Kodierung von Fehlern im Steuergeräte-Fehlerspeicher
ECU	Electronic Control Unit Steuergerät
E/E	Elektrik/Elektronik
EOBD	European On-Board Diagnosis Sammelbegriff für europäische Vorschriften zur Diagnose abgasrelevanter Elektroniksysteme im Kraftfahrzeug
EOL	End of Line Programmierung Programmieren von Steuergeräten in der Fahrzeugfertigung
EPA	Environmental Protection Agency US-amerikanische Umweltschutzbehörde, legt u. a. die zulässigen Abgasemissionen von Kraftfahrzeugen fest.
EMC	Electromagnetic Compatibility
EMV	Elektromagnetische Verträglichkeit
FF, CF	First Frame, Continous Frame
SF, FC	Single Frame, Flow Control Frame Botschaftstypen des Transportprotokolls ISO 15765-2
FCS	Frame Check Sequence Prüfsumme bei Ethernet-Botschaften

Flash-ROM	Flash Read Only Memory
Flaschen	Löschen- und programmierbarer Speicher in Steuergeräten. Der Programmervorgang wird umgangssprachlich als <i>Flaschen</i> bezeichnet.
FIBEX	Field Bus Exchange Format Datenformat zur Beschreibung der On-Board-Kommunikation im Kfz
FIFO	First Input First Output Speicher, bei dem die Daten in derselben Reihenfolge gelesen werden müssen, in der sie geschrieben wurden.
FlexRay	Zeitsynchrones Kfz-Bussystem
FTDMA	Flexible Time Division Multiple Access Zugriffsmethode auf ein Bussystem, bei dem der Zugriff in Zeitschlitzten variabler Länge erfolgt.
GW	Gateway
GSM	Global System for Mobile Communications Weltweites digitales Mobilfonsystem
HAL	Hardware Abstraction Layer, Microcontroller Abstraction Layer
MCAL	Softwareschicht, die die darüberliegenden Softwareschichten von der Hardware entkoppelt.
hex	Hexadezimale Zahlen Zahlen zur Zahlenbasis 16, üblich sind unterschiedliche Schreibweisen, z. B. hex $7Fh = \$7 F = 0x7 F = 7 \cdot 16^1 + 15 \cdot 16^0 = 127$ dezimal.
HIL	Hardware-in-the-Loop-Simulation
HIS	Herstellerinitiative Software Herstellervereinigung zur Standardisierung von Kfz-Software
Host	Mikrocontroller, der einen Kommunikationscontroller steuert
HW	Hardware
ID	Identifier Kennziffer
IFB, IFS	Inter Frame Break, Inter Frame Space
IPG	Inter Packet Gap Pause zwischen Busbotschaften
IO	Input – Output
I/O	Sammelbegriff für Ein-Ausgangssignale bei Steuergeräten sowie deren softwareseitige Verarbeitung
JASPAR	Japanese Automotive Software Platform Architecture Herstellervereinigung zur Standardisierung von Software im Kfz
KWP	Keyword Protocol Meist als KWP 2000 verbreitetes Diagnoseprotokoll
LAN	Local Area Network Sammelbegriff für Bussysteme für Bürocomputer
Layer	Protokollsicht
LIN	Local Interconnect Network Verbreitetes Bussystem für einfache Anwendungen
LSB	Least Significant Bit Niederwertigstes Bit eines Datenwortes

LTE	Long Term Evolution Mobiltelefonstandard der 4. Generation
LWL	Lichtwellenleiter
MAC	Media Access Control Teil des Kommunikationscontrollers, der den Buszugriff steuert.
MCD	Messen, Kalibrieren, Diagnose (Measurement, Calibrate, Diagnose) Sammelbegriff für Aufgaben bei der Applikation von Elektroniksystemen im Kraftfahrzeug. Schwerpunkt von ASAM.
MDX	Meta Data Exchange Format for Software Module Exchange ASAM Beschreibungsformat für Steuergeräte-Software
MIL	Malfunction Indicator Light Fehlerwarnlampe im Armaturenbrett
mod	Modulo Operation $n = x \text{ mod } N \dots$ Rest einer ganzzahligen Division $n = x/N$
MOST	Media Oriented System Transport Bussystem für Infotainment-Anwendungen im Kraftfahrzeug
MSR	Messen, Steuern, Regeln
MSB	Most Significant Bit Höchstwertigstes Bit eines Datenwortes
MT	Makrotick
μ T	Mikrotick Zeitschritte bei FlexRay
MVCI	Modular Vehicle Communication Interface Schnittstelle zwischen Diagnose- oder Applikationssystem und Kfz-Bussystem
NM	Network Management Spezifikation für das Netzmanagement im Rahmen von OSEK/VDX
Nibble	Hälfte eines Bytes Bezeichnung für zusammengehörende Datenbits (4 bit)
NIC	Network Interface Controller Kommunikationscontroller
NRZ	Non Return to Zero Codierung für Datenbits auf den Busleitungen
OBD	On-Board Diagnosis Sammelbegriff für Vorschriften zur Diagnose abgasrelevanter Elektroniksysteme im Kraftfahrzeug, aus USA kommend, als EOBD auch in Europa angewendet.
ODU	On-Board Unit Fahrzeugseitiges Steuergerät bei Maut- und Telematik-Systemen
ODX	Open Diagnostic Data Exchange ASAM Beschreibungsformat für Diagnosedaten
OEM	Original Equipment Manufacturer Fahrzeughersteller
OIL	OSEK Implementation Language Beschreibungssprache für OSEK/VDX Systeme

OS	Operating System Betriebssystem
OSAL	Operating System Abstraction Layer Schicht zur Nachbildung einer Betriebssystem-Schnittstelle
OSEK/VDX	Offene Systeme für die Elektronik im Kraftfahrzeug/Vehicle Distributed Executive: Betriebssystem- und Kommunikationsstandard
OSI	Open System Interconnection Schichtenmodell für Datennetze und Protokolle
PCI	Protocol Control Information Steuerinformationen einer Protokollebene
PDA	Personal Digital Assistant Taschencomputer, andere Bezeichnungen Organizer, Palmtop
PDU	Protocol Data Unit Datenblock einschließlich Daten für die Übertragungssteuerung
PDX	Packed ODX Archivformat für ODX-Datensätze
PHY	Physical Bus Connect Busanschlusseinheit, Bustreiber
PLL	Phase Locked Loop Schaltungsblock für die Takterzeugung oder Takt synchronisation
POF	Plastic Optic Fiber Lichtwellenleiter
PWM	Pulse Width Modulation Pulsbreitenmodulation
Quadlet	Bezeichnung für vier zusammengehörige Datenbytes (32bit)
RAM	Random Access Memory: Schreib-Lese-Speicher
ROM	Read Only Memory: Nur-lesbarer Speicher
PROM	Programmable ROM: Programmierbares ROM
EPROM	Erasable PROM: Löschbares PROM (nur komplett löschbar)
EEPROM	Electrical Erasable PROM: Löschbares PROM (einzelne Zellen)
Flash-ROM	Blockweise löschbares PROM
RFC	Request for Comment Spezifikation der Internetprotokolle bei www.itef.org
RSU	Road Side Unit Strassenseitiges Steuergerät bei Maut- und Telematik-Systemen
RX	Receiver Data
RXD	Eingang für empfangene Daten
SW	Software
SCI	Serial Communication Interface, andere Bezeichnung für UART
SDU	Service Data Unit Nutzdaten einer Datenbotschaft
SPI	Serial Peripheral Interface Serielle Schnittstelle zwischen Mikrocontrollern und Peripherie-ICs
STP	Shielded Twisted Pair Abgeschirmte, verdrillte Zwei-Draht-Leitung

TCP/IP	Transmission Control Protocol/Internet Protocol Im Internet und in LANs verwendetes Netzwerkprotokoll
TDMA	Time Division Multiple Access Zugriffsmethode auf ein Bussystem, bei dem der Zugriff in vorgegebenen Zeitschlitzten fester Länge und Lage erfolgt.
FTDMA	Variante mit Zeitschlitzten variabler Länge
TTCAN	Time Triggered CAN Zeitsynchrone Variante des CAN-Busses
TPP	Time Triggered Protocol Zeitsynchron arbeitendes Bussystem
TP	Transport Protocol Verfahren zur Aufteilung von Datenblöcken auf mehrere Bus-Botschaften
TX	Transmitter Data
TXD	Ausgang für gesendete Daten
UART	Universal (Serial) Asynchronous Receiver and Transmitter
USART	Einfache Standardschnittstelle für die serielle Datenübertragung
UDS	Unified Diagnostic Services Diagnoseprotokoll nach ISO 14229
UML	Unified Modeling Language Standard für die grafische Spezifikation von Software
UMTS	Universal Mobile Telecommunications System Mobiltelefonstandard der 3. Generation
URL	Unified Resource Locator Web-Link, standardisierte Adressangabe für Dokumente im Internet
UTP	Unshielded Twisted Pair Ungeschirmte, verdrillte Zwei-Draht-Leitung
UUDT	Unacknowledged Unsegmented Data Transfer
USDT	Unacknowledged Segmented Data Transfer
AUDT	Acknowledged Unsegmented Data Transfer
ASDT	Acknowledged Segmented Data Transfer Datenübertragung ohne/mit Bestätigung durch den Empfänger sowie ohne/mit Segmentierung, d. h. Aufteilung einer Nachricht auf mehrere Botschaften.
VANET	Vehicle Ad Hoc Network Funk-Netzwerk mit kurzer Reichweite zwischen Fahrzeugen
V2V	Vehicle to Vehicle, Vehicle to Infrastructure, Vehicle Safety Comm.
V2I	Kommunikation zwischen Fahrzeugen und Verkehrsleiteinrichtungen, siehe auch C2C
VSC	C2C
WLAN	Wireless Local Area Network Funk-Netzwerk mit kurzer Reichweite
WWH-OBD	World-Wide Harmonized OBD Überarbeitete Version der Abgasdiagnose OBD
XCP	Universal Measurement and Calibration Protocol Neueres Busprotokoll in der Applikation von Elektroniksystemen im Kraftfahrzeug, von ASAM spezifiziert.
XML	Extended Markup Language Textbasiertes Format für die strukturierte Beschreibung von Informationen

Sachverzeichnis

A

Acknowledge ACK, [22](#), [27](#)
Adressierung, [20](#), [21](#), [197](#)
Akzeptanzfilterung, [65](#)
Antwortzeit, [358](#)
Applikationsdatensätze, [272](#), [275](#)
Applikationswerkzeug, [429](#), *siehe auch* CCP,
 XCP
Arbitrierung, [28](#)
ARP, [146](#)
ASAM, [10](#), [229](#)
 A2L, [255](#), [272](#)
 AML, [256](#)
 ASAP2, [272](#)
 Calibration Data Format CDF, [256](#), [275](#)
 CCP, [234](#)
 D-PDU API, [316](#)
 FIBEX, [261](#)
 MCD, [229](#)
 MCD1, [232](#)
 MCD2, MCD3, [270](#)
 Measure, Calibrate, Diagnose, [232](#)
 Meta Data Exchange MDX, [275](#)
 MVCI, [316](#)
 ODX, [277](#)
 XCP, [241](#)
Asynchron, [24](#)
Audio-Video-Bridging AVB, [142](#)
Authentifizierung, [442](#)
Autorenwerkzeug, [469](#)
AUTOSAR, [10](#), [331](#)
 AUTOSAR COM, [385](#), [389](#)
 AUTOSAR NM, [397](#)
 AUTOSAR OS, [381](#)
 AUTOSAR RTE, [403](#)
 Basissoftware, [369](#)

BSW Scheduler, [384](#)
CAN, [391](#)
Communication Services, [385](#)
Diagnostic Communication Manager, [387](#)
Diagnostic Log and Trace DLT, [379](#)
ECU State Manager, [376](#)
End-to-End Communication Protection, [380](#)
FlexRay, [393](#)
Funktionale Sicherheit, [380](#)
Hardware Abstraction, [369](#)
Hardwaretreiber, [372](#)
LIN, [395](#)
Memory Services, [374](#)
PDU Router, [390](#)
Runtime Environment, [403](#)
Softwarekomponenten, [403](#)
SPAL, MCAL, [370](#)
System Services, [376](#)
Transportprotokolle, [390](#)
Virtual Functional Bus VFB, [404](#)
Watchdog Manager, [380](#)
Werkzeugkette, [370](#)
Zeitverhalten, [358](#), [404](#)

B

Babbling Idiot, [105](#)
Backbone, [31](#)
Betriebssystem, [334](#)
Bitrate, [15](#)
 CAN, [79](#)
 FlexRay, [118](#)
 K-Line, [41](#)
 LIN, [95](#)
 MOST, [138](#)
 SAE J1850, [42](#)

- Bit-Stuffing, 27, 62
Blockierend, 24
Body, 31
Botschaft, 18
Broadcast, 21, 22
BroadR-Reach, 142
Bus, 16
 Linienbus, 16
 Topologie, 17
Bus Guardian, 105
Busabschlußwiderstand, 16
Busanalyse, 424
Bus-Interface, 459
 Media Access Control MAC, 23
 Physical Interface PHY, 23
Bussystem
 Class A, B, C, 5
 High Speed, 2
 Infotainment, Multimedia, 4
 Low Speed, 2
Buszugriffsverfahren, 27
 CSMA/CA, 28
 CSMA/CD, 28
 Master-Slave, 28
 TDMA, 29
Bypass, 430
- C**
CAN, 8, 57
 Basic, 65
 Botschaftspriorität, 67
 CAN in Automation (CiA), 60
 Controller, 64
 Full, 66
 High Speed, 59
 Jitter, 69
 Latenzzeit, 69
 Low Speed, 59
 Matrix, 63
 Partial Networking, 76
 Scheduling, 67
 Time triggered, 72
 Zeitverhalten, 67
CANdb, 64
Car to Car C2C, 483
Car to Infrastructure C2I, 483
CCP
 CAN Calibration Protocol, 234
Flashen, 238
 Kalibrieren, 238
 Messdatenerfassung, 239
 Speicherzugriff, 237
 Verbindungssteuerung, 236
Chassis, 31
Client-Server, 23
Cluster, 81
Codierung
 Non-Return-to-Zero (NRZ), 15
Confirmation, 24
Cyclic Redundancy Check CRC, 27
- D**
D2B
 Domestic Data Bus, 119
Datenrate
 FlexRay, 102
 K-Line, 42
 LIN, 95
 MOST, 138
 Nutzdatenrate, 25
Deadline Monotonic, 71, 360
Dedicated Short Range Communication DSRC, 485
Desegmentierung, 23
DHCP, 146
Diagnosedaten
 Autorenwerkzeug, 469
 ODX, 277
Diagnosedienste
 Diagnosesitzung, 205
 Diagnostic Service, 189
 KWP 2000, 199
 OBD, 213
 UDS, 205
Diagnosesteckdose, 36
Diagnosetester, 316, 458
Diagnosewerkzeug, 459, 469
Differenzsignal, 14
Distributed Systems Interface DSI, 51
DNS, 146
DoIP, 183
Domain Controller, 31
Domäne, 31
dominant, 17
D-PDU API, 316
Duplex
 Halb-Duplex, 14
 Voll-Duplex, 14

E

E/E-Architektur, 30
End-of-Line-Programmierung, 436
Ethernet, 138, 183
 BroadR-Reach, 142
Extended Calibration Protocol, *siehe* XCP
Extended Markup Language XML, 257

F

Fehlerspeicher, 220
 Entprellung, 220
 Heilung, 220
 KWP 2000, 200
 OBD, 214
 OBD Fehlercode, 218
 UDS, 209

Field Bus Exchange Format

 FIBEX, 261

Fingerprint, 445

Flashen
 Container, 452
 Flash-Lader, 448
 Flash-Programmierung, 433
 Flash-Treiber, 447
 KWP 2000, 202
 ODX, 306
 Pass-Through Programmierung, 221
 Programmiersequenz, 441
 Speichertypen, 434
 UDS, 211

FlexRay, 8, 96

 Bus Guardian, 105
 Cliquenbildung, 105
 Cycle counter, 99
 dynamisches Segment, 100
 Jitter, 107
 Keyslot, 104
 Latenzzeit, 107
 Makrotick, 99
 Mikrotick, 104
 Minislot, 100
 Network Idle Time, 99
 Network Management Vector, 105
 Scheduling, 107
 Single Slot Mode, 105
 Slot counter, 99
 Slot Multiplexing, 108, 116
 Startup Frame, 104
 statisches Segment, 99

Symbol Window, 99

Sync Frame, 104
TT-D, TT-L, TT-E, 117
Zeitverhalten, 107

Flusssteuerung, 24, 156

Frame, 20
 Consecutive, 155
 Error, 62
 First, 154

Flow Control, 156

 Remote, 63
 Single, 154

Funktionale Sicherheit, 380

G

Gateway, 19, 31

H

Hardware-in-the-Loop (HIL), 430
Hash, 445
Header, 18
HIS, 10, 331
 CAN Treiber, 363
 Flash-Lader, 363, 440
 Flash-Programmierung, 441
 Flash-Treiber, 447
 Herstellerinitiative Software, 361
 IO Driver, 361
 IO Library, 361

I

ICMP, 147
IDB 1394
 Firewire, 120
Identifier, 22
 CAN, 61
 Eröffnungs-, 170, 176
 FlexRay Frame ID, 101
 Kanal-ID, 170, 177
 LIN, 81
 Message-, 61
IEEE
 802.1 Q/AVB, 142
 802.3 Ethernet, 138
Indication, 24
Infotainment, 4, 31, 119
Intellectual Property, 426
Internet Protokoll, 183
IP, 140, 145, 396

-
- ISO**
- 9141, 8, 32
 - 10681, 162, 386
 - 11783 (ISOBUS), 8
 - 11898, 8, 57
 - 11992, 8, 60
 - 13209, 319
 - 13400, 183, 386
 - 14229, 9, 190, 203
 - 14230, 9, 32, 189, 192
 - 15031, 9, 191, 212
 - 15765, 9, 154, 190–192, 386
 - 17356, 10, 335
 - 17458, 96
 - 22898, 49
 - 22900, 232, 278, 316
 - 22901, 232, 278, 316
 - 23248, 221
 - 26262, 380
 - 27145, 212, 225
- J**
- JASPAR, 331
 - Jitter, 29, 69, 94, 107
 - JTAG, 437
- K**
- Kanal, 169, 176
 - Keyword, 33
 - K-Line, 32
 - Kommunikation
 - Off-Board, 4
 - On-Board, 2
 - Kommunikationsmatrix, 416
 - Kommunikationsmodell, 20
 - KWP 2000, 33, 38, 189, 192
 - Adressierung, 197
 - Botschaftsaufbau, 193
 - Diagnosesitzung, 194
 - Fehlerspeicher, 200
 - Flashen, 202, 441
 - Input/Output Control, 201
 - Kommunikationsmodell, 192
 - KWP 2000 on CAN, 190, 192
 - KWP 2000 on K-Line, 192
 - Remote Routines, 203
 - Response Codes, 194
 - Service Identifier, 193
- L**
- Latenz, 29, 69, 94, 107, 358
 - Laufzeit, 358
 - Leitung
 - Ein-Draht-Leitung, 14
 - Zwei-Draht-Leitung, 14
 - Lichtwellenleiter, 121
 - LIN, 8, 79
 - API, 92
 - Jitter, 94
 - Konfiguration, 87
 - Latenzzeit, 94
 - Scheduling, 94
 - Zeitverhalten, 94
 - L-Line, 34
- M**
- Malfunction Indicator Lamp MIL, 214
 - Mapping, 417
 - Master, 81
 - Mautsystem, 484
 - Medium Access Control MAC, 140
 - Message, 20
 - Message Identifier, 61
 - Modular Vehicle Communication Interface
 - MVCI, 316
 - MOST, 8, 119
 - Block, 123
 - Customer Convenience Port, 120
 - Frame, 123
 - Network Master, 133
 - Network Services, 129
 - Netzmanagement, 133
 - Ringbruch-Diagnose, 134
 - Steuerbotschaften, 126
 - Steuerdaten, 125
 - Synchrone Daten, 124
 - Timing master, 121
 - Multicast, 21
 - Multimedia, 119
- N**
- Nachrichtenorientiert, 180
 - NDP, 146
 - Netzdesign, 416
 - Normen, 7, 224, 227, 328, 413
 - Not Acknowledged NACK, 22
 - Notification, 24
 - Nutzdaten, 18

O

- OBD, 57, 191, 212
Botschaftsaufbau, 213
Fehlercodes, 218
Fehlerspeicher, 214, 215
Messwerte lesen, 215
Pass-Through Programmierung, 221, 222
Scan Tool, 197, 212
Testfunktionen, 215
- ODX, *siehe auch* ASAM
Autorenwerkzeuge, 469
Comparam-Spec, 278
Comparam-Subset, 278
Complex DOP, 299
Data Object Property DOP, 292
Diag-Layer, 280
Diag-Layer-Container, 279
Diagnoserdienste, 288
Diagnosesitzungen, 304
Diagnosevariable, 291
Diagnostic Trouble Code Object DTC-DOP, 299
ECU-Config, 280
Flash, 279, 306
Function-Dictionary, 280
Hierarchie, 280
Job, 302, 303
Open Data Exchange, 277
Packed ODX, 309
Schichtenmodell, 280
Special Data Group, 302
Vehicle-Info-Spec, 278, 283
- Off-Board-Kommunikation, 4
On Event, 23
On-Board Diagnostics, 212
On-Board-Kommunikation, 2
- OSEK/VDX, 10, 334
Betriebssystem OS, 336
Deadline, 358
FTCOM, 355
Implementation Language OIL, 341
Kommunikation COM, 346
Latzenz, 358
Netzmanagement NM, 350
Offene Systeme für die Elektronik im Kraftfahrzeug, 334
OSEK Time, 355
Protected OS, 355
Scheduling, 358

Überblick, 334

Zeitverhalten, 358

OSI, 6**P**

- Parameter Group Number, 179
Partial Networking, 75, 76, 402
Pass-Through Programmierung, 221, 316
Payload, 18
PDU Format, 179
Plastic Optic Fiber, 121
Port, 147
Powertrain, 31
Pretended Networking, 402
Private Key, 445
Protocol Control Information PCI, 19
Protocol Data Unit PDU, 19
Protokollstapel, 20
Protokolltest, 453
Public Key, 446
Publisher-Subscriber, 23
Punkt-zu-Punkt-Verbindung, 13
PWM, 43

R

- Rapid Prototyping, 430
Rate Monotonic, 71, 360
Reizung, 36
Repeater, 19
Request-Response, 23
Restbus-Simulation, 460
Restbussimulation, 422
rezessiv, 17
RFC
 768 UDP, 147
 791 IP, 146
 792 ICMP, 147
 793 TCP, 147
 826 ARP, 146
 2131 DHCP, 146
 2460 IP, 146
 3550 RTP, 143
Ring, 18, 122

S

- SAE
 J1587, 178
 J1708, 8, 178
 J1850, 8, 42

J1939, 8, 57, 178, 386	Timing
J1979, 191, 212	AUTOSAR OS, 358
J2012, 219, 226	CAN, 67
J2178, 44	CAN ISO TP, 159
J2190, 191	Ethernet, 142
J2284, 8, 57	FlexRay, 107
J2411, 60	FlexRay TP, 167
J2534, 221, 316	LIN, 94
J2602, 80	OSEK OS, 358
J2716, 45	Toll Collect, 484
Safe by Wire, 49	Trailer, 18
Schedule, 417	Transceiver, 19
Schichtenmodell	Transportprotokoll, 9, 153
ISO/OSI, 6	CAN ISO TP, 154, 390
Segmentierung, 20, 23, 154	FlexRay TP, 162, 390
Sensor-Aktor-Bus	ISO 13400, 183
ASRB, 49	LIN, 85
DSI, 51	SAE J1939, 178
PSI5, 46	TP 1.6, 175
SENT, 45	TP 2.0, 169
Service Data Unit SDU, 19	TTCAN, 72
Service Identifier	TTP, 8
SID, 39	
Signal, 19, 417	U
Binär, 15	Übertragung
Bipolar, 15	Bitstrom-orientiert, 25
Laufzeit, 16	Gesichterte, 23
Ternär, 15	Ungesicherte, 23
Unipolar, 15	Zeichenstrom-orientiert, 25
Signallaufzeit, 16	UDP, 145, 396
Signatur, 445	UDS, 190, 203
Simulation, 420	Botschaftsaufbau, 204
Slave, 81	Diagnosesitzung, 205
Sleep, 29	Fehlerspeicher, 209
Socket, 396	Flashen, 211, 441
Stand By, 29	Input/Output Control, 210
Standards, 7, 224, 227, 328, 413	Remote Routines, 211
Stern, 18, 98	Response on Event, 212
Steuergerätetest, 423	UDS on CAN, 191, 203
Stuff-Bit, 62	Unified Diagnostic Service, 203
Switch, 139	UML
Synchron, 24	Unified Modeling Language, 257, 258
Systemintegration, 424, 430	Unicast, 21
	Unified Modeling Language UML, 257
T	
TCP, 145, 396	V
Teilnetzbetrieb, 76	Validierung, 442
Telematik, 483	Vehicle Ad Hoc Network VANET, 485
Testsequenzen, 319	Vehicle to Vehicle V2V, 483

-
- Verbindungsabbau, 21
 - Verbindungsauflbau, 21
 - Verbindungslos, 21
 - Verbindungsorientiert, 21, 180
 - Verschlüsselung, 445
 - VLAN, 140
 - VPWM, 43
 - W**
 - Wake Up, 29
 - Widerstand
 - Abschluss-, 16
 - Pull-Up-, 16
 - Wired-OR, 17
 - X**
 - XCP
 - Extended Calibration Protocol, 241
 - Flashen, 247, 248
 - Kalibrieren, 249
 - Z**
 - Zeitfenster, 74
 - Zeitverhalten
 - AUTOSAR OS, 358
 - CAN, 67
 - CAN ISO TP, 159
 - Ethernet, 142
 - FlexRay, 107
 - FlexRay TP, 167
 - LIN, 94
 - OSEK OS, 358
 - XML
 - Extended Markup Language, 257