Werkzeuge, Anwendungen und Einsatzgebiete 9

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.

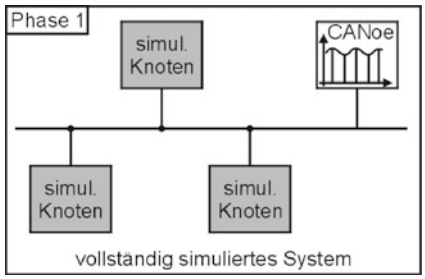
9.1 Entwurf und Test der On-Board-Kommunikation

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.

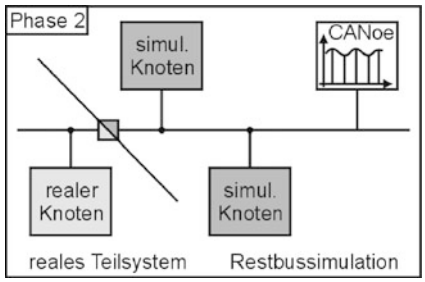
9.1.1 Entwicklungsprozess mit *CANoe* von Vector Informatik

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:

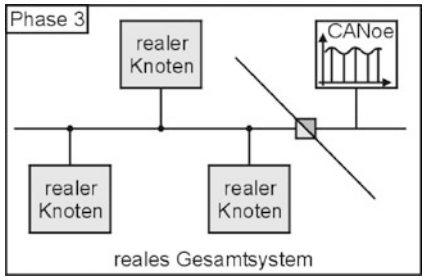
W. Zimmermann und R. Schmidgall, *Bussysteme in der Fahrzeugtechnik*, ATZ/MTZ-Fachbuch, 415
DOI 10.1007/978-3-658-02419-2_9, © Springer Fachmedien Wiesbaden 2014



Phase 1: Netzwerkdesign und Simulation
In der ersten Phase der Entwicklung legt der OEM die komplette Kommunikation zwischen den einzelnen Netzknoten fest. Durch eine Simulation in *CANoe* kann verifiziert werden, ob der Entwurf vollständig ist und Latenzzeiten 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 Netzknoten. 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 Fahrzeugherstellern 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

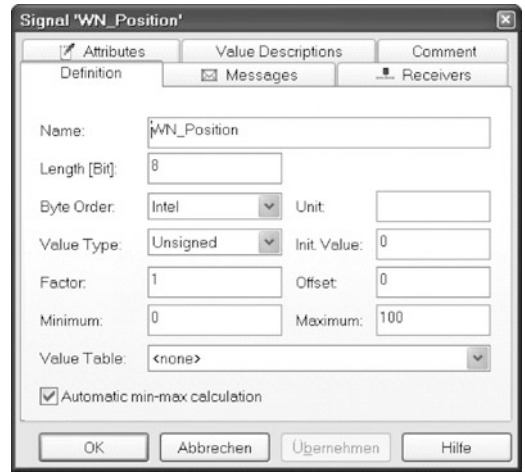


Abb. 9.1 Definition eines Signals

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 Netzknoten und die einzelnen Signale definiert. Signale sind diejenigen Informationen, die die Funktionen in den verschiedenen Netzknoten 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 Bussystem.

Anschließend folgt die Abbildung (*Mapping*) der Signale auf die eigentlichen Busbotschaften (Abb. 9.2) sowie deren Verteilung auf die Zeitschlitze (*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.

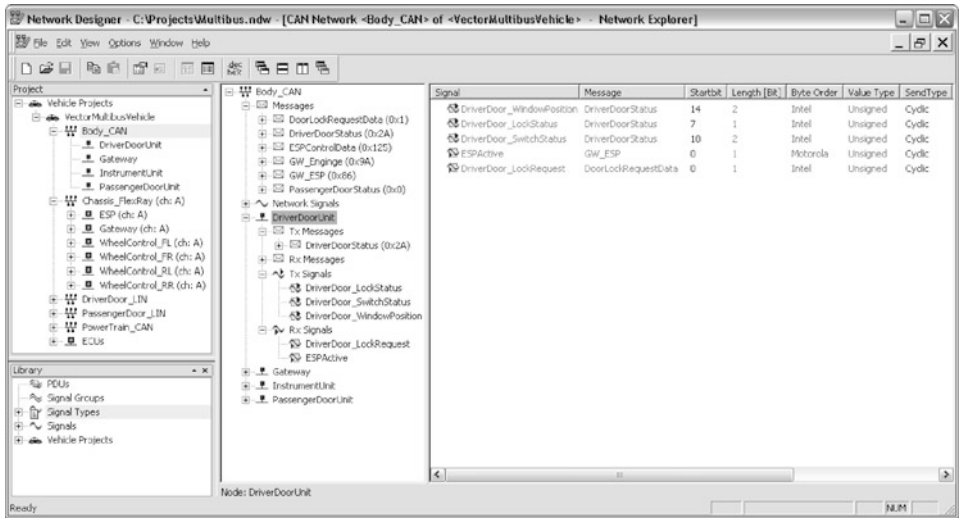


Abb. 9.2 Mapping von Signalen für einen Knoten im *Network Designer*

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-

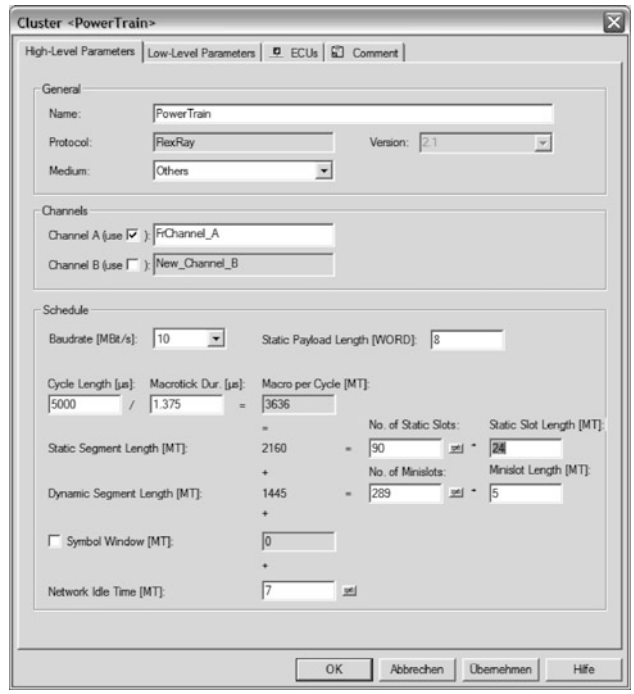


Abb. 9.3 Grundparameter für ein FlexRay-Cluster

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.

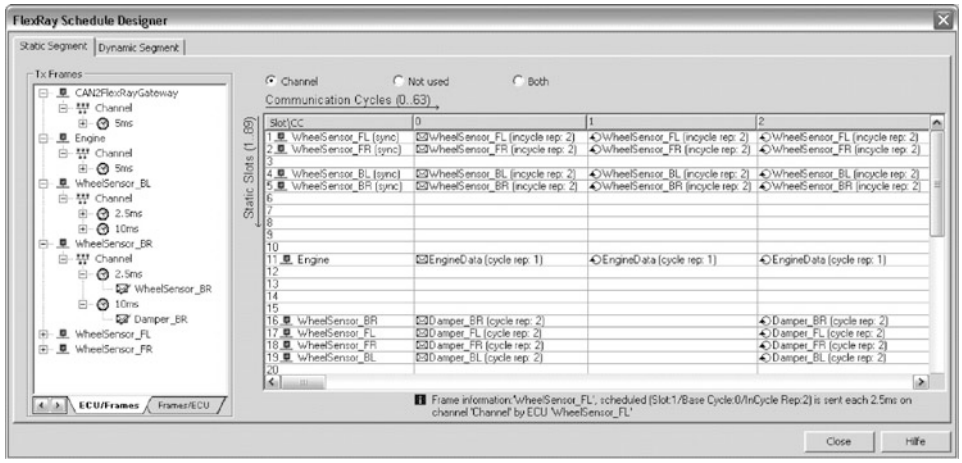


Abb. 9.4 Schedule eines FlexRay-Systems

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) verwenden 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

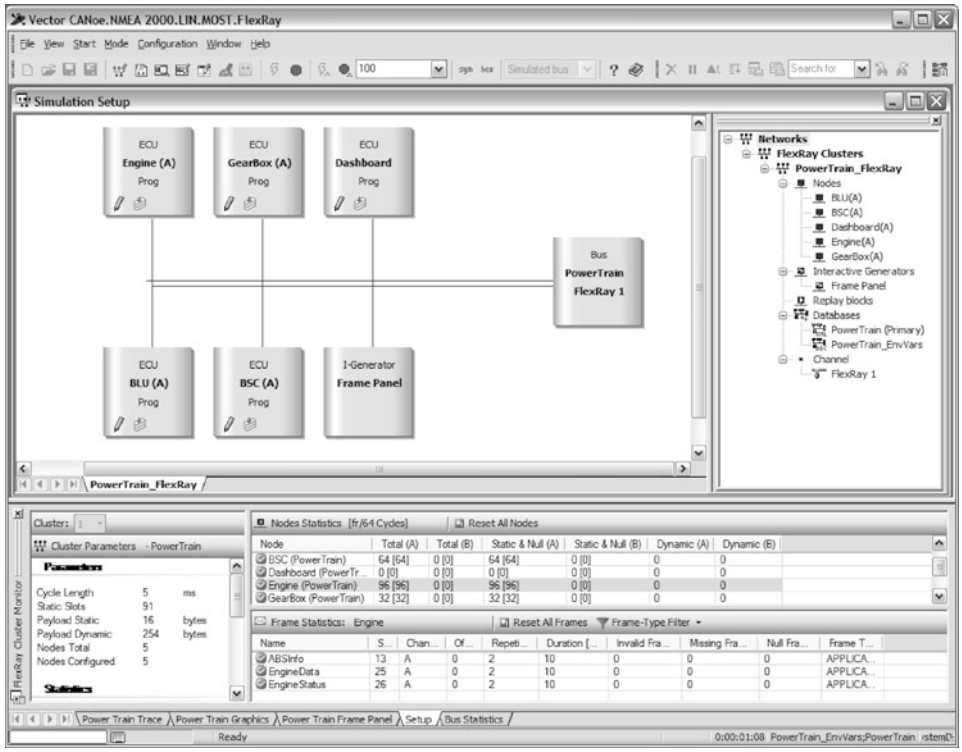


Abb. 9.5 FlexRay-Simulation in *CANoe*

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).

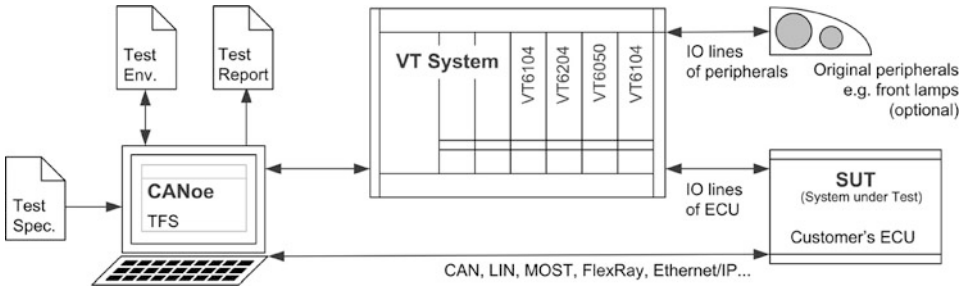


Abb. 9.6 Komplette *CANoe*-Testumgebung für die Steuergeräteentwicklung

Die Simulation darf mehrere Netze mit unterschiedlichen Bussystemen enthalten. Im simulierten Model 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 Abschlusstest 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* unterstützen 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. Nach dem 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

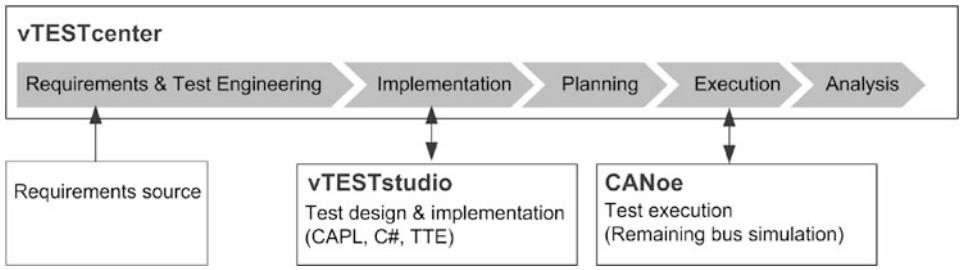


Abb. 9.7 Testplanung und Testdurchführung

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 Restbussimulation 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 Protokollschichten 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

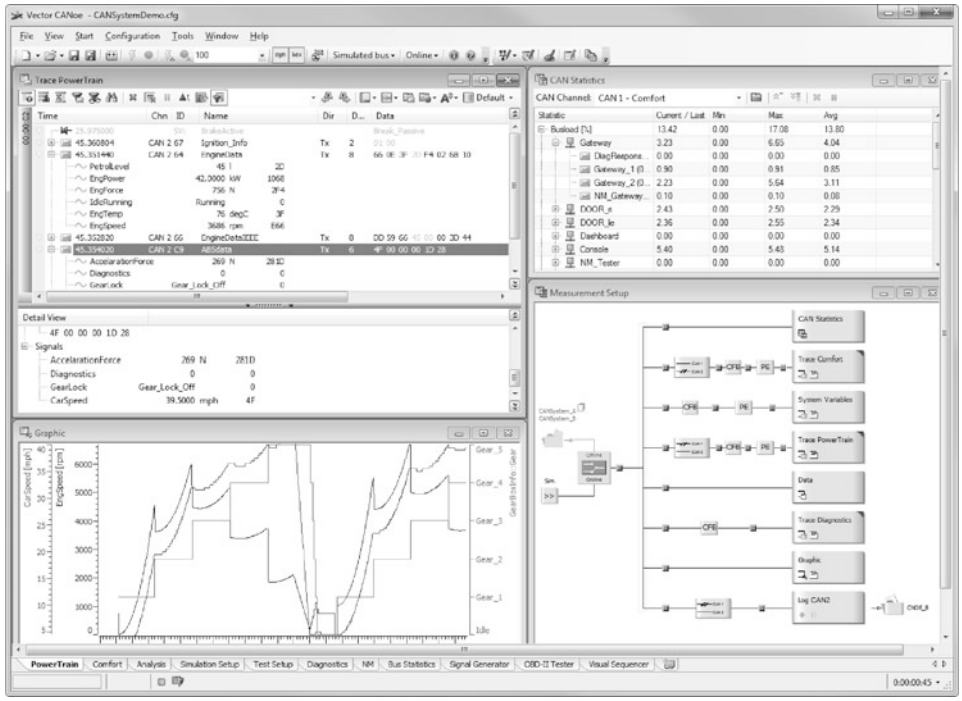


Abb. 9.8 Analyse der Buskommunikation mit *CANoe*

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.

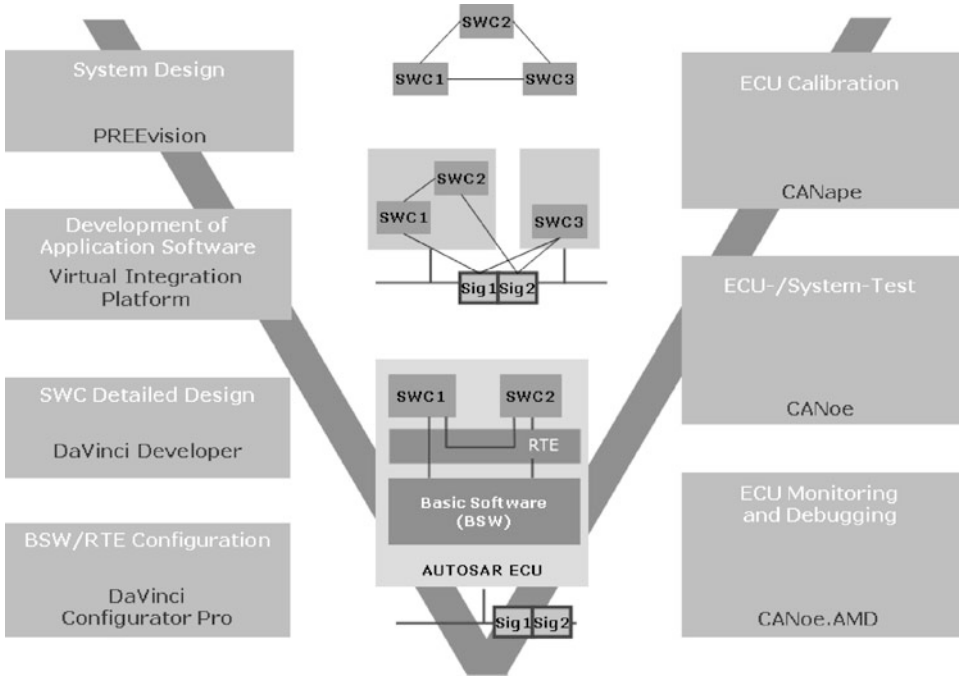


Abb. 9.9 Werkzeugkette für den AUTOSAR-Entwicklungsprozess

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).

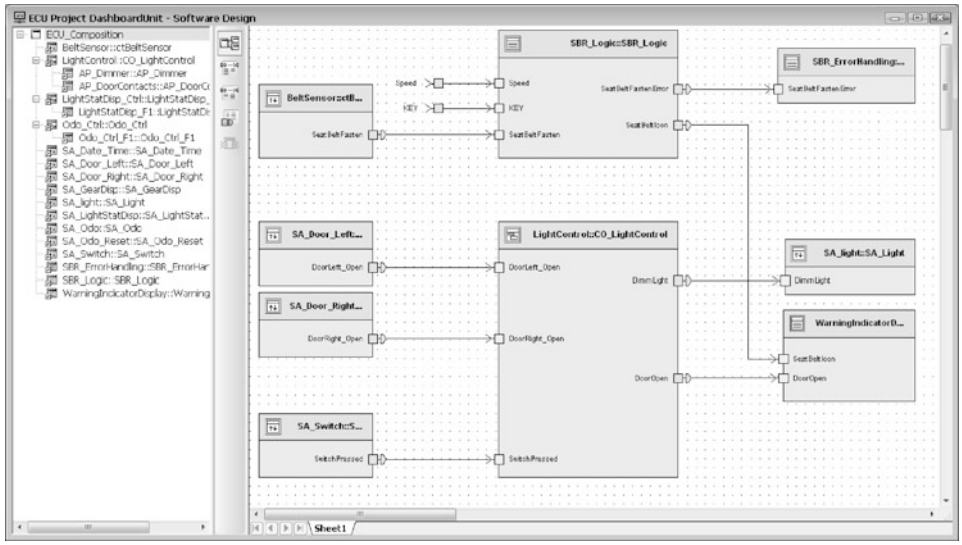


Abb. 9.10 Verknüpfung von Softwarekomponenten im *DaVinci Developer*

- 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.

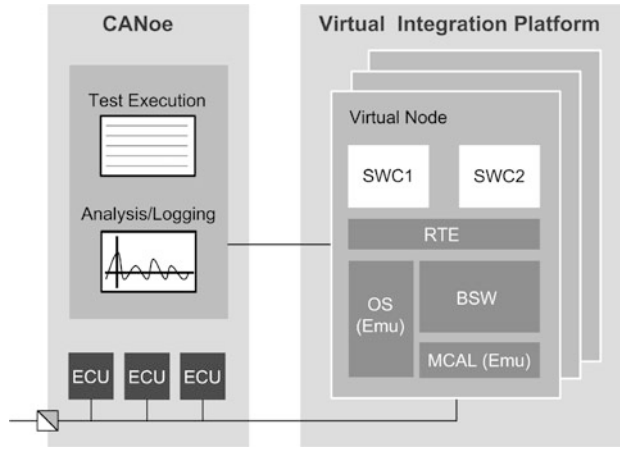


Abb. 9.11 Testumgebung für die AUTOSAR-Softwarekomponenten

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 Zahlendarstellung 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 Kalibrieraufgaben 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-

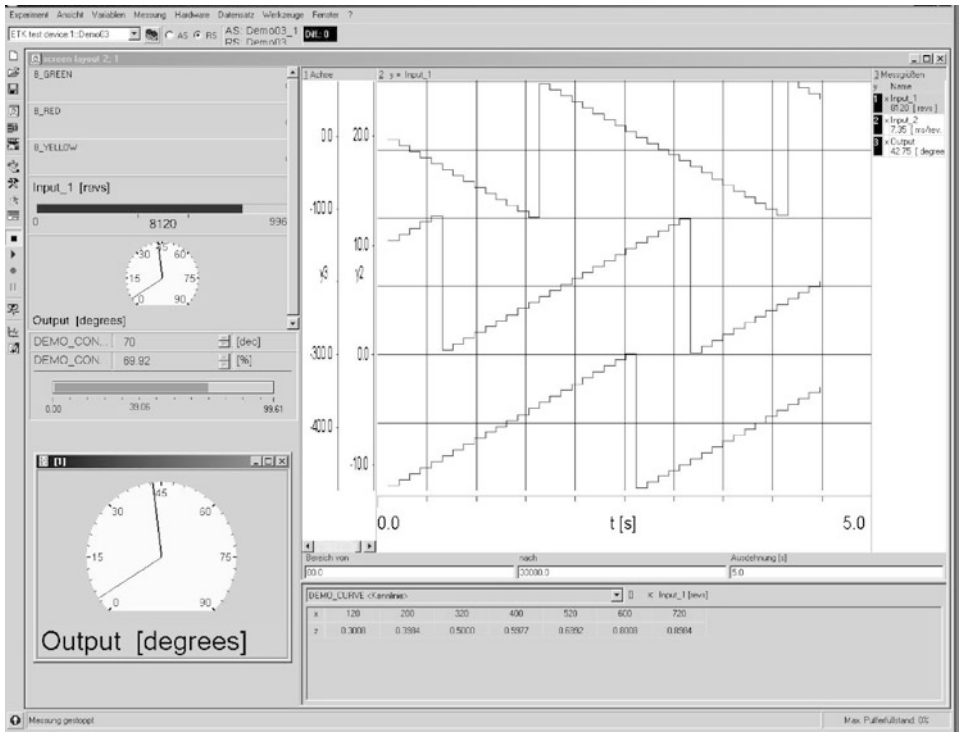


Abb. 9.12 Typisches Applikationswerkzeug zur Erfassung von Messwerten und Verstellung von steuergeräteinternen Parametern (ETAS *INCA*)

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 komplette 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:

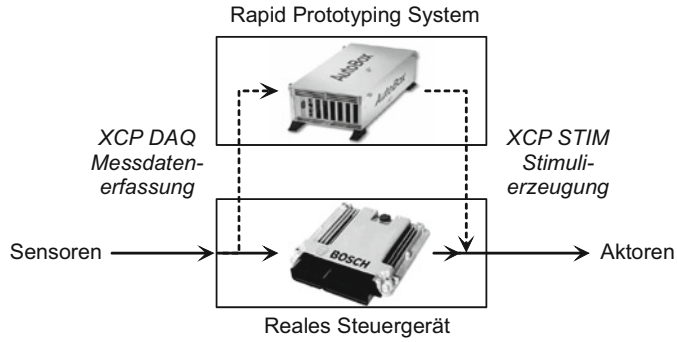


Abb. 9.13 Steuergerät mit Rapid-Prototyping-System in *Bypass*-Struktur

- 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

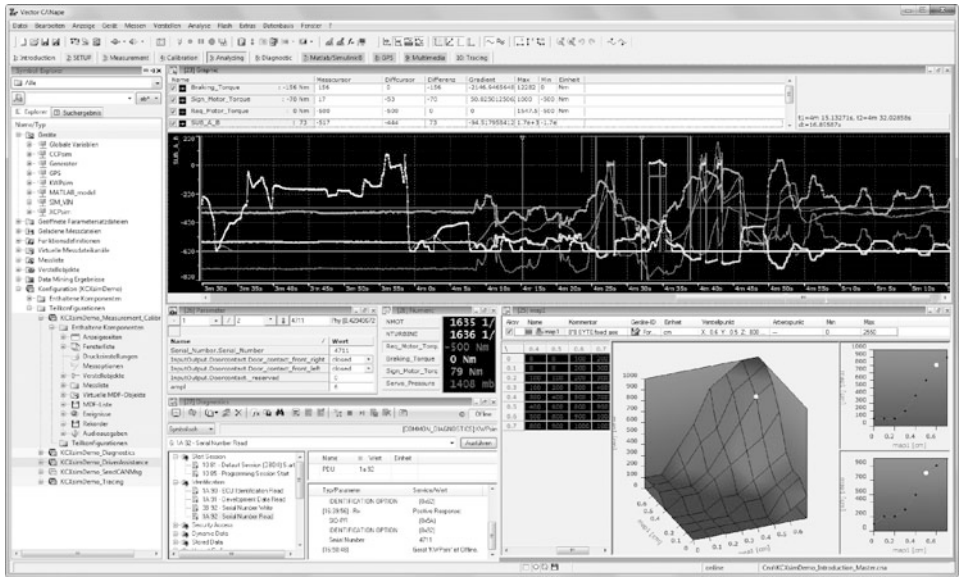


Abb. 9.14 Applikationswerkzeug *CANape* von Vector Informatik

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 Verstelloperationen 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. *M*it Hilfe des *Simulink Model Explorers* visualisiert *CANape* die *Simulink*- und *Stateflow*-Modelle. Durch Kopplung zwischen Modell und ASAP2/A2 L-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. Betriebsstundenzähler 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 Fehlerspeichereinträgen verwendet. Bei aktuellen Mikrocontrollern werden EEPROMS häufig durch ein Flash-ROM emuliert. Dadurch kann das zusätzliche EEPROM eingespart werden.

Der Lösch- und anschließende Programmiervorgang 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 Fahrzeugher-

Tab. 9.3 Typische Stationen der Flash-Programmierung

Wo wird programmiert?	Wer ist verantwortlich?	Was wird programmiert?	Erläuterung
Programmier-station	Steuergeräte-Hersteller	Anwendungs-software inklusive Betriebssystem und Diagnose-protokoll	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 Applikationsda-tensätze (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-Konfigurations-daten	Freischaltung von Funktionen für das spezifische Fahrzeug und Feinjustierung
Werkstatt (Service)	Fahrzeug- und/oder Steuergeräte-Hersteller	Neue Anwen-dungssoftware, Fahrzeugkonfigu-ration	Fehlerbeseitigung, Kompensation von Alterung und Verschleiß, Nachrüstung neuer Funktionen

steller für den Programmiervorgang 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 programmierter 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
Programmier-station	JTAG-Schnittstelle mit bis zu 10 MBit/s Datenrate	Body:	256 KB	30 sec … 5 min
		Powertrain:	1 … 4 MB	
		Multimedia:	5 … 20 MB	
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 Anwendungs-software wie Programmierstation Neue Fahrzeugkonfiguration wie EOL		5 … 10 min je Gerät, komplettes Fahrzeug im Stundenbereich Ziel: max. 15 min für Fahrzeug-Update

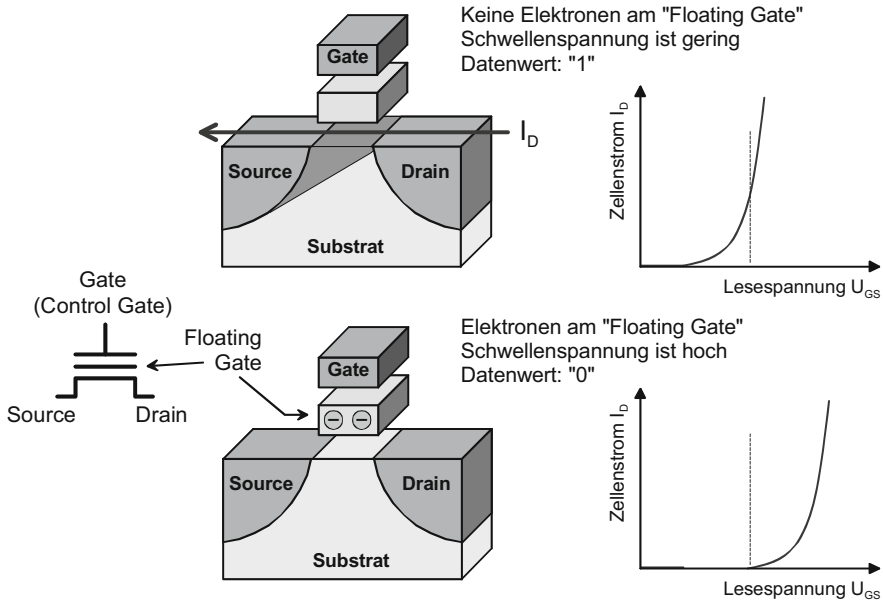


Abb. 9.15 Aufbau einer Flash-Speicherzelle

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 eingelagert, 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 *Routine Control - Check Programming Dependencies*.
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 *Hintertüren*, z. B. Sperren des Zugriffs über JTAG, nach erfolgreichem Programmiervorgang beim Gerätehersteller. Verwendung von Signaturen und Zertifikaten zur Prüfung, ob der Programmierdatensatz manipuliert wurde.
Robustheit gegen Fehlprogrammierung	Der *Flash-Lader* muss auch dann ansprechbar sein, wenn während des Programmiervorgangs ein Fehler auftritt und der Programmiervorgang abgebrochen wird. Dazu muss der *Flash-Lader* selbst gegen unbeabsichtigtes Überschreiben gesichert werden, die Konsistenz der programmierten Daten bei jedem Einschalten des Geräts überprüft werden. Nur bei *korrekten* Daten wird der normale Betrieb (*Fahrfunktion*) des Gerätes aufgenommen, ansonsten verbleibt die Software im *Flash-Lader*.

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 Programmiervorgangs so mit Elektronen geladen und dadurch die Schwellspannung soweit erhöht ist, dass die Transistoren beim Lesen nicht lei-

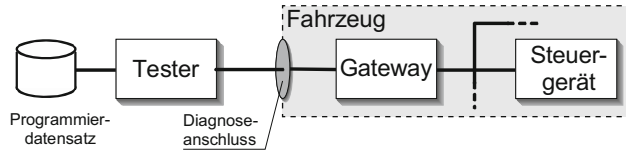


Abb. 9.16 Hauptkomponenten des Flash-Programmierprozesses

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 Programmiervorgänge 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 Programmiervorgangs 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 Programmiervorgang 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 µs.

Der Lösch- und Programmiervorgang 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 Programmiervorgangs 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 Programmiervorgang 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-Programmiervorgang 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 Steuergeräteprogramm 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 Programmiervorgang 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 Routinen zum Löschen und Programmieren des Flash-Speichers in das RAM des Steuergerätes kopiert

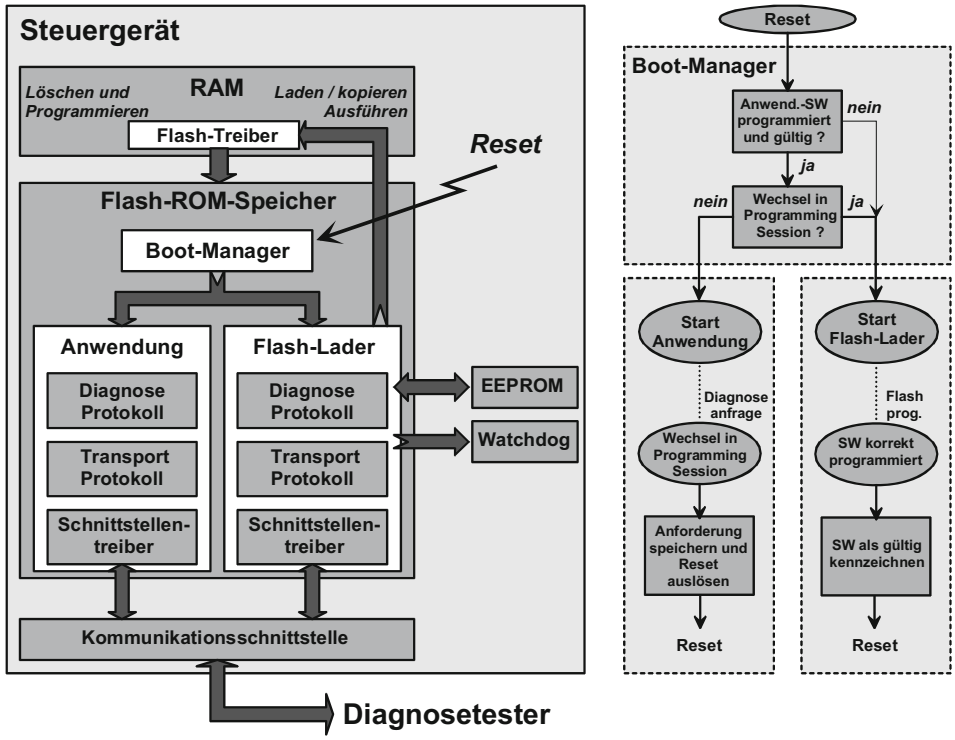


Abb. 9.17 Grundstruktur der Steuergerätesoftware für die Flash-Programmierung

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 Fahrzeughersteller 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 Fahrzeugherstellers 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)

	Diagnosesitzung starten	Beginn der Diagnosesitzung zur Vorbereitung des Programmiervorgangs. UDS: *Diagnostic Session Control – Start Extended Session*
*	**Steuergeräte-Identifikation auslesen**	Auslesen der Steuergeräteidentifikation zur Erkennung der verbauten Hardware und des alten Softwarestands. UDS: *Read Data by Identifier*
*	**Vorbedingungen prüfen**	Prüfung, ob alle gerätespezifischen Vorbedingungen für eine Programmierung erfüllt sind, z. B. Motor im Stillstand. UDS: *Routine Control – Check ProgPreCondition*
	Fehlerspeicher in anderen Geräten abschalten	Abschalten der Fehlerspeicherung in denjenigen Steuergeräten am Bus, die nicht programmiert werden, um z. B. Timeout-Fehler durch fehlende Busbotschaften zu verhindern. UDS: *Control DTC Settings – DTC Settings = off*
	Kommunikationsstopp für andere Geräte	Alle am Bus angeschlossenen Steuergeräte werden aufgefordert, die normale Kommunikation einzustellen. Somit steht für den Programmiervorgang eine höhere Busbandbreite zur Verfügung. UDS: *Communication Control – Disable non-diagnostic communication*
	Programmiermodus starten	Wechsel in Programmiermodus. UDS: *Diagnostic Session Control – Start Programming Session*
	Authentifizierung durchführen	Authentifizierung des Testers gegenüber dem Steuergerät. Bei falscher Authentifizierung wird der Zugang verweigert. UDS: *Security Access – Get Seed, Send Key*
	Historienspeicher aktualisieren	Für jeden erfolgreichen oder fehlgeschlagenen Versuch wird dokumentiert, wo und wann das Steuergerät programmiert wurde. UDS: *Write Data by ID – Write Fingerprint*
*	**Flash-Treiber Teil 1 ins RAM laden**	Falls der *Flash-Treiber* fester Bestandteil des *Flash-Laders* ist, wird er von diesem zu gegebener Zeit ins RAM kopiert. Andernfalls muss er in diesem Schritt vom Diagnosetester in das Steuergeräte-RAM geladen werden. UDS: *Request Download, Transfer Data, Request Transfer Exit*
	Flash-Speicher löschen	Der Diagnosetester startet das physikalische Löschen des Flash-Speichers. UDS: *Routine Control – Start Routine by Identifier*

Tab. 9.6 (Fortsetzung)

*	**Flash-Treiber Teil 2 ins RAM laden**	Bei Steuergeräten mit kleinem RAM ist es möglich, dass nicht der gesamte Flash-Treiber auf einmal geladen werden kann. Der Treiber wird dann in einen ersten Teil für das Löschen und einen zweiten Teil für das Programmieren aufgeteilt, der erst jetzt geladen wird.
	Neue Anwendungssoftware programmieren	Der Tester überträgt die Daten blockweise zum Steuergerät. Der *Flash-Lader* programmiert diese Daten physikalisch in den Flash-Speicher. UDS: *Request Download*, *Transfer Data, Request Transfer Exit, Start Routine by Identifier*
	Verifikation der Programmierung	Nach Abschluss der Programmierung wird die neu programmierte Anwendungssoftware verifiziert. Korrekte Programmierung und Legitimation (*Signatur*) prüfen. UDS: *Routine Control – Start Check Routine*
	Prüfung der Abhängigkeiten	Ist die Anwendungssoftware in mehrere Blöcke untergliedert, so muss die Kompatibilität der einzelnen Teile geprüft werden. UDS: *Routine Control – Check Programming Dependencies*
*	**Parametrierung**	Variantenkodierung oder ähnliche Parametrierung des Gerätes. UDS*: Write Data By Identifier*
	Reset des Steuergerätes	Ein Reset beendet den Programmiervorgang. Nach dem Neustart startet der *Boot-Manager* die neue Anwendungssoftware. UDS: *ECU Reset*
	Freigabe der Kommunikation anderer Geräte	Bei allen am Diagnosebus angeschlossenen Steuergeräten wird die normale Kommunikation wieder freigegeben. UDS: *Communication Control – Enable non-diagnostic communication*
	Fehlerspeicher in anderen Geräten einschalten	Aktivierung der Fehlerspeicherung in den anderen Steuergeräten am Bus. UDS*: Control DTC Settings – DTC Settings = on*
	Diagnose beenden	Diagnosesitzung beenden. UDS*: Session Control – Start default session*

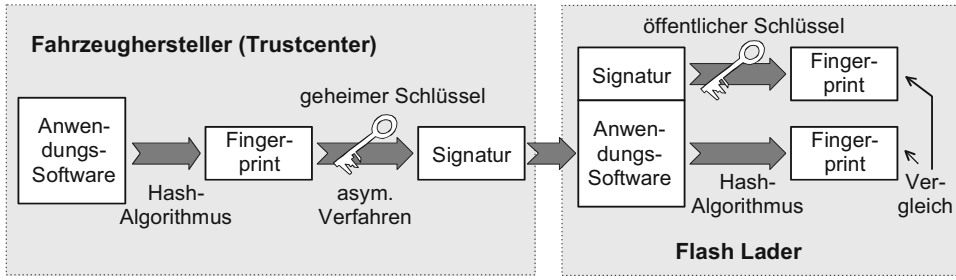


Abb. 9.18 Validierung einer Software

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 *öffentlich bekannten 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.

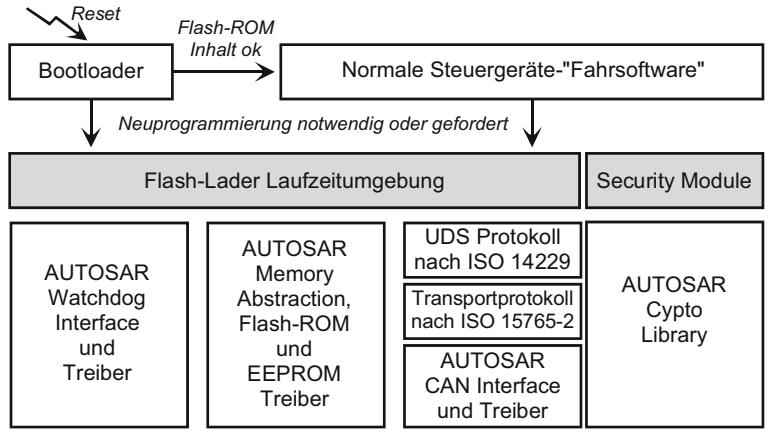


Abb. 9.19 Aufbau des HIS-Flash-Laders auf Basis von AUTOSAR-Komponenten

- 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.

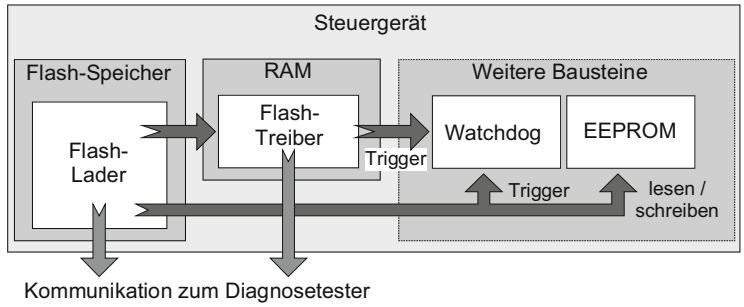


Abb. 9.20 Steuergeräte-Komponenten für die Flash-Programmierung

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 Programmiervorgang notwendigen Teile der Diagnose-Protokollsoftware in einem während des Programmiervorgangs 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 Programmiervorgang 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 *Interrup*t-Service-Routinen verwenden, da diese häufig im zu löschenden 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 Hardwaretreiber 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 *Flashen* 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 Fahrzeugherstellern 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, Fahrzeughersteller 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 Fahrzeughersteller 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.

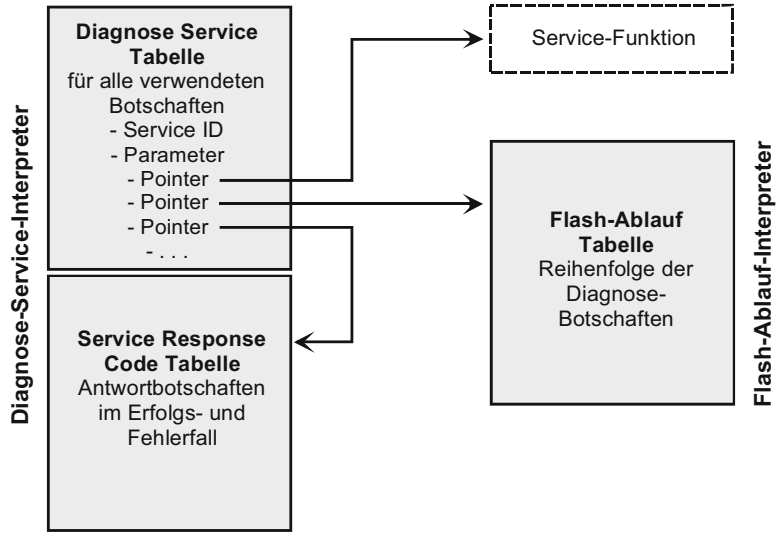


Abb. 9.21 Diagnoseservice-Beschreibung und Flash-Ablauf

- 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 *ADLATUS* 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, escrypt 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 Diagnosetesters, 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-

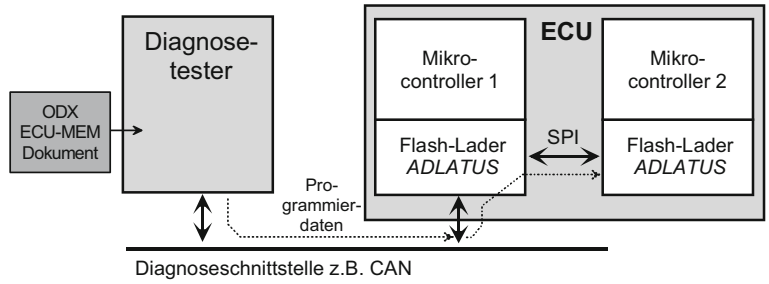


Abb. 9.22 Flash-Lader mit Gateway-Funktion

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.

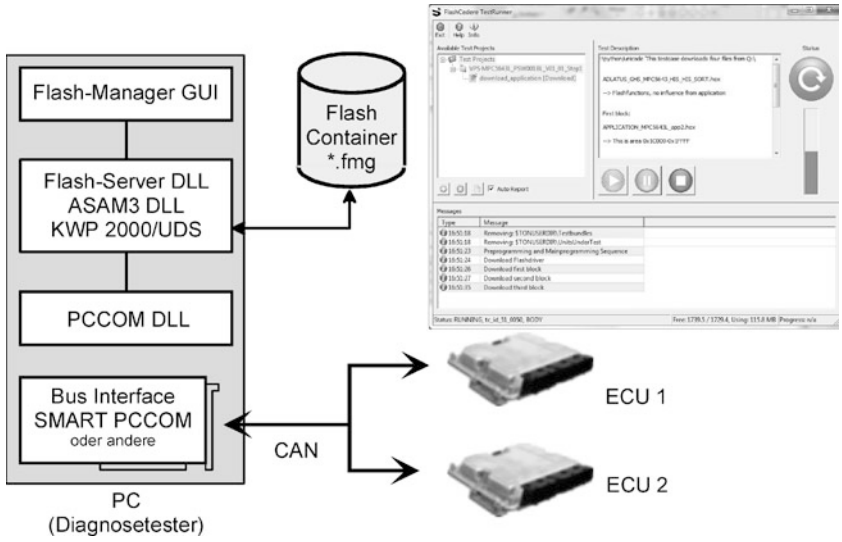


Abb. 9.23 *FlashCedere* als Diagnosetester für Entwicklung und Produktion

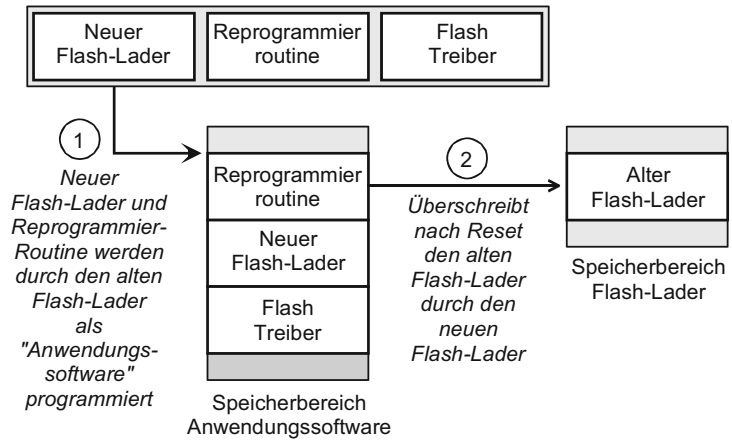


Abb. 9.24 Automatisches Ersetzen des *ADLATUS* Flash-Laders

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.

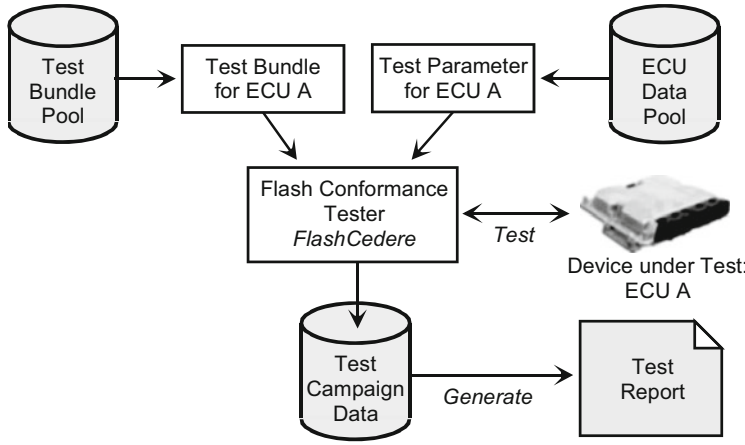


Abb. 9.25 Testen mit SMART *FlashCedere*

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 variantenspezifische Parameter, z. B. die CAN-Bitrate oder die Steuergeräte-Diagnoseadresse, nicht unmittelbar im Testfall codiert sind, sondern von außen parametriert werden können.
- 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.

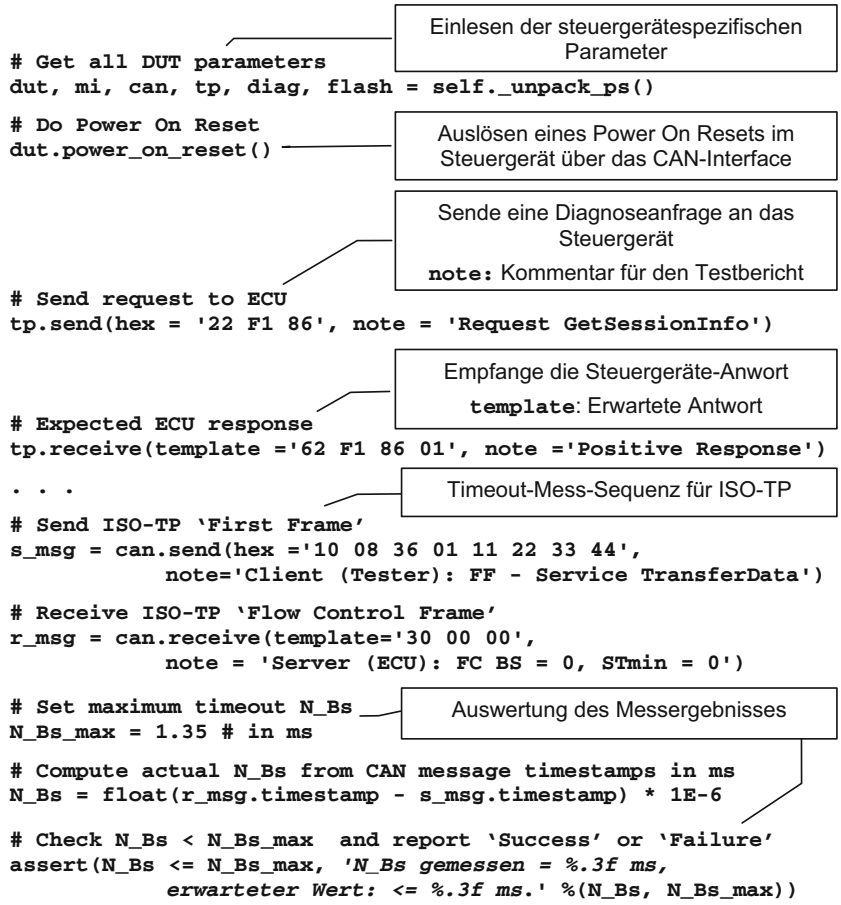


Abb. 9.26 Python-Testscript für den Testfall *Abfrage der aktiven Diagnosesitzung*

- *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.

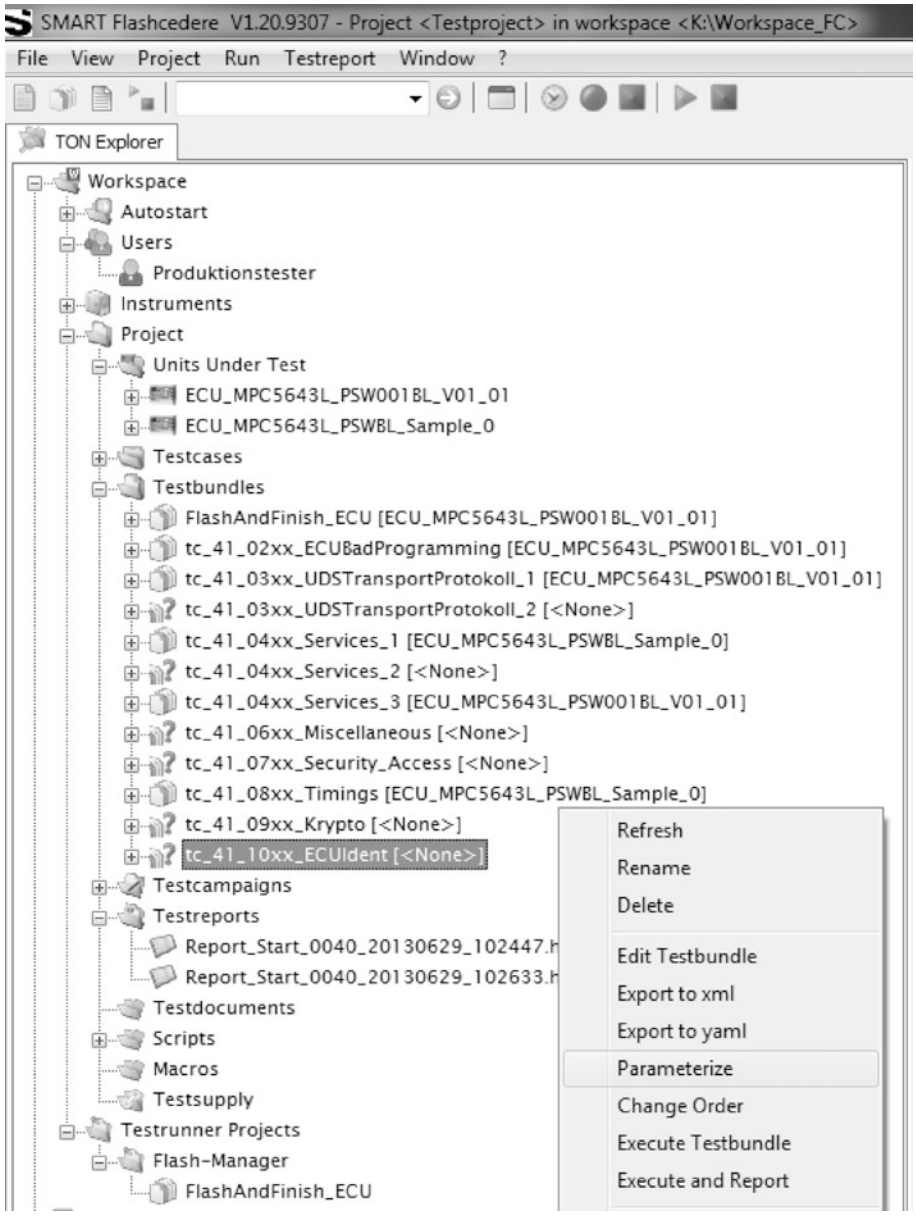


Abb. 9.27 Projektübersicht mit generischen Testfällen in *FlashCedere*

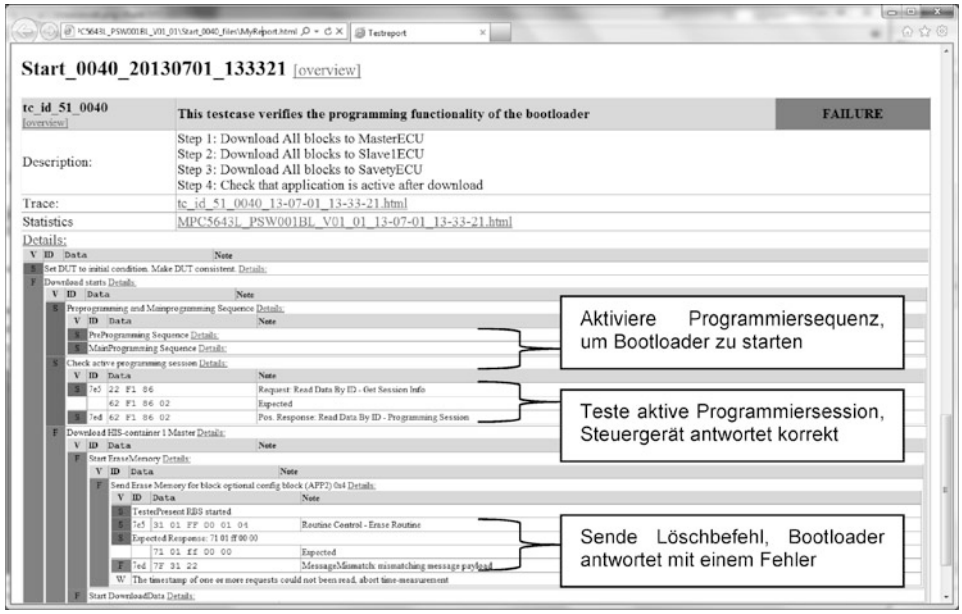


Abb. 9.28 Testbericht im HTML-Format, über *Cascaded Style Sheets* formatiert

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 Fahrzeughersteller 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

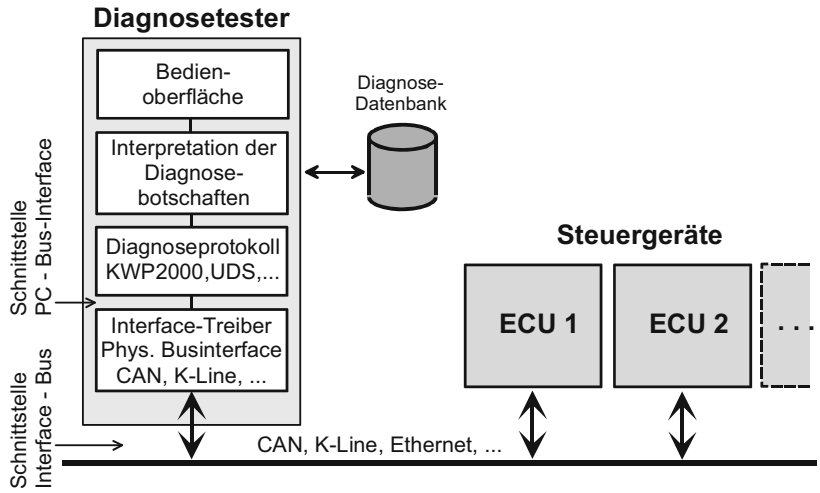


Abb. 9.29 Typische Struktur eines Diagnosetesters

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 Werkstatttester bietet für die Entwicklung in der Regel zu wenige Freiheiten und ist in dieser Phase meist auch noch gar nicht verfügbar.

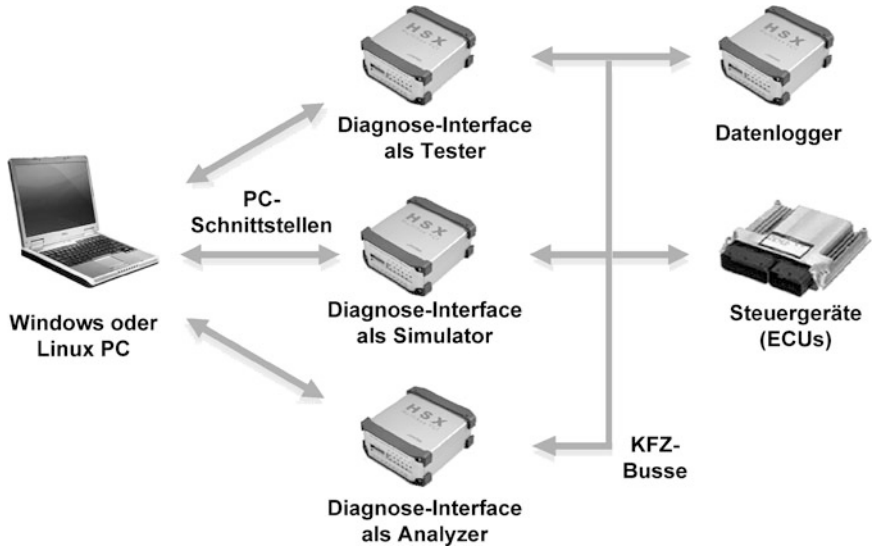


Abb. 9.30 Funktionen der Samtec Diagnose-Interfaces

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 stattet 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 Fertigungslinie 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.

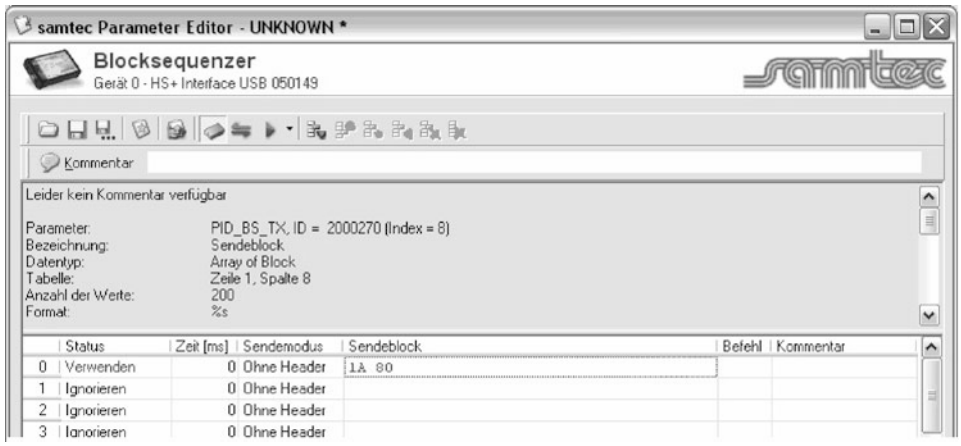


Abb. 9.31 Blocksequenzer (Samtec *samDia*)

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 Fehlerspeichers, das Auslesen der einzelnen Fehlerinformationen und das anschließende Löschen der Fehlereinträge 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-

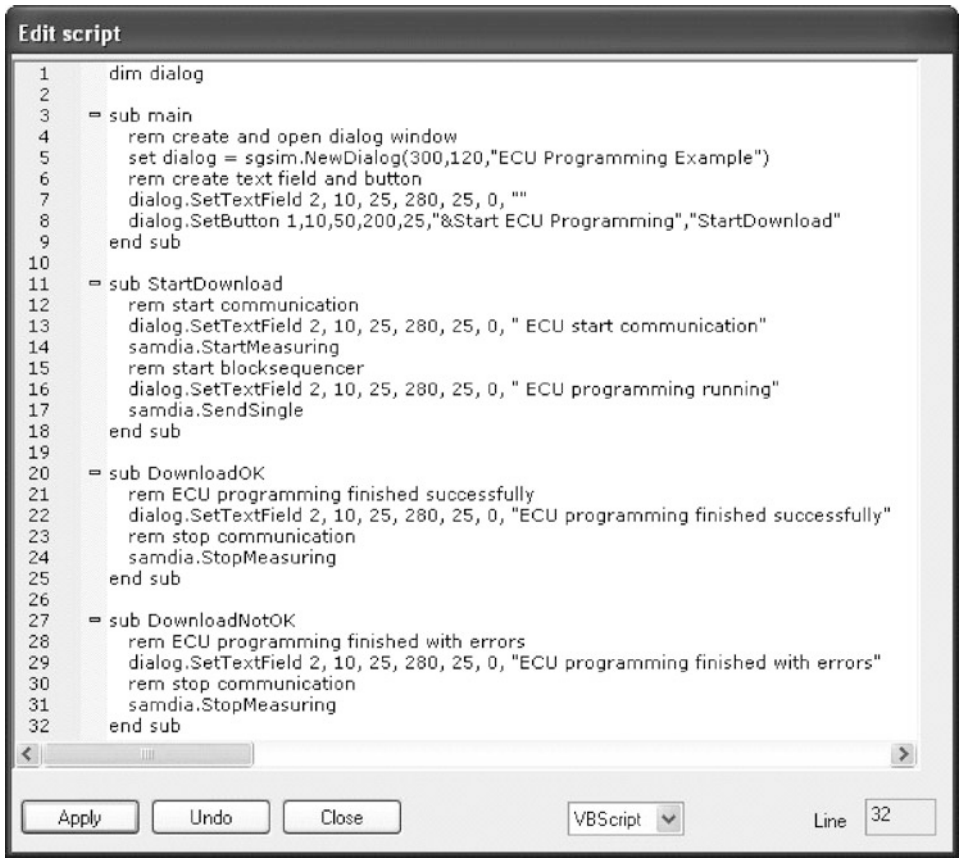


Abb. 9.32 Skript zur automatischen Steuergeräte-Programmierung über KWP 2000

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 Diagnosetester 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 Programmiervorgang erhält der Anwender eine Rückmeldung über die Funktion `DownloadOK()`.

Die vom Diagnosetester 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 Programmiervorgang 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 Diagnosetester 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 Diagnosetester 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 Diagnosetester-Entwicklung Bei der Entwicklung der Diagnosetester-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:

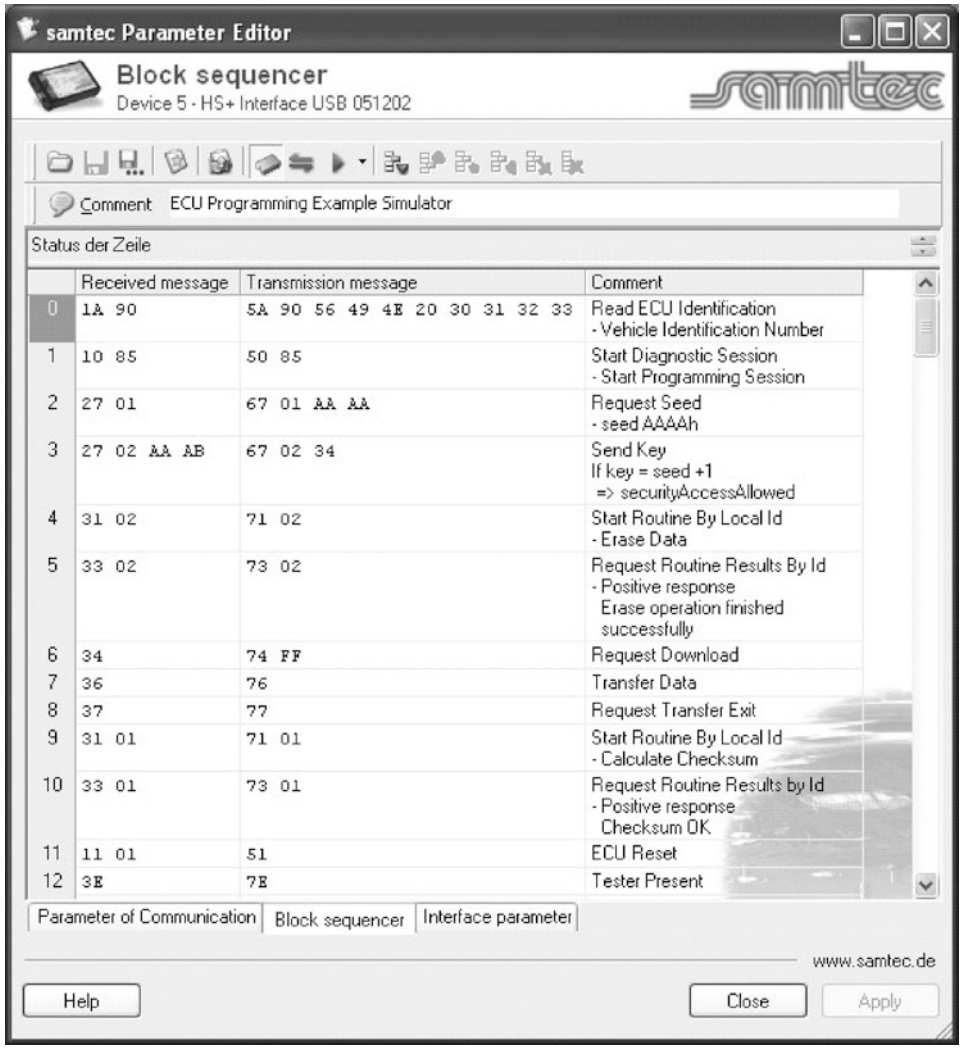


Abb. 9.33 Simuliertes Steuergerät

- 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.

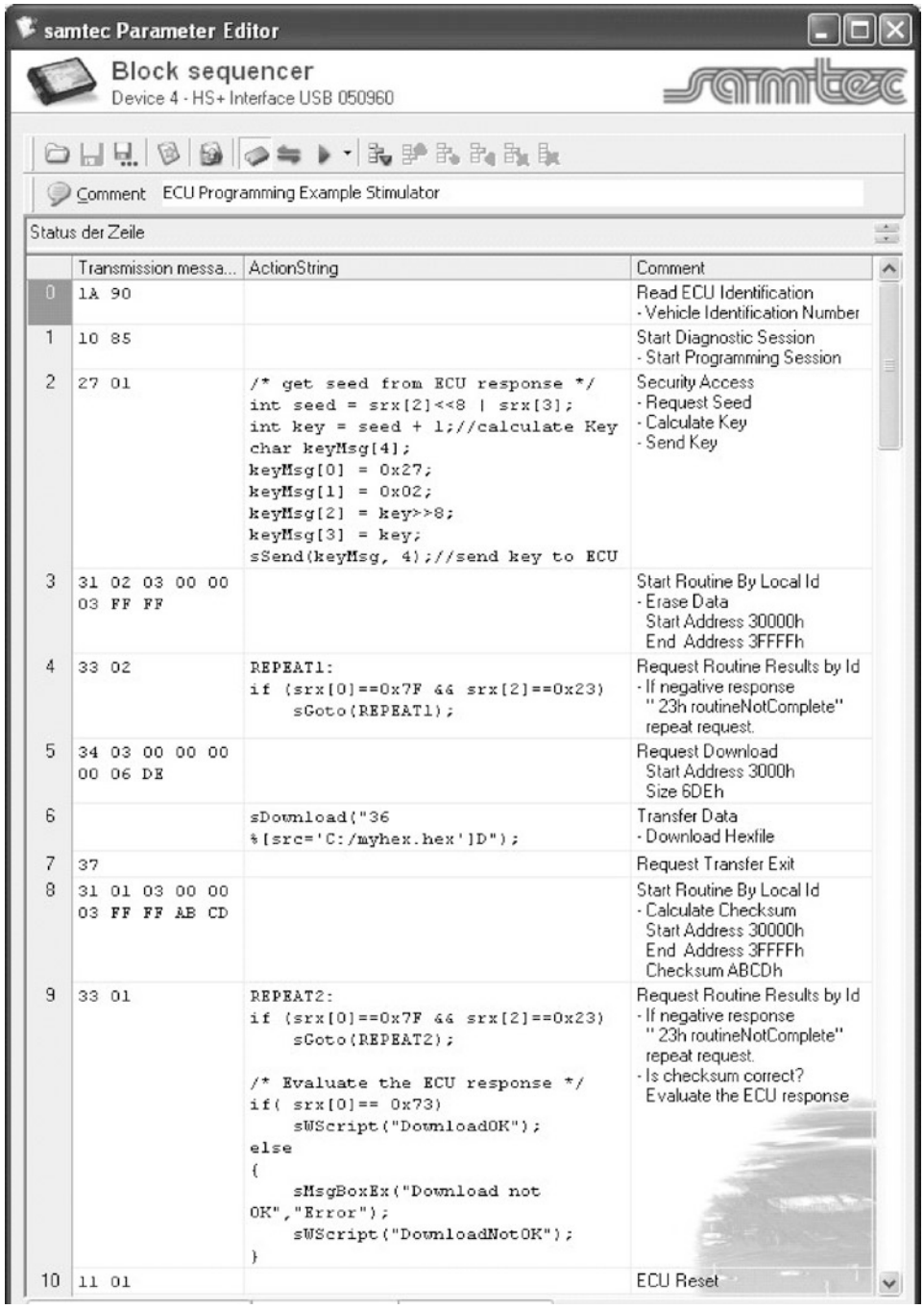


Abb. 9.34 Blocksequenz für den Diagnosetester (*Stimulator*)

```
> 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 Programmiervorgangs 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.

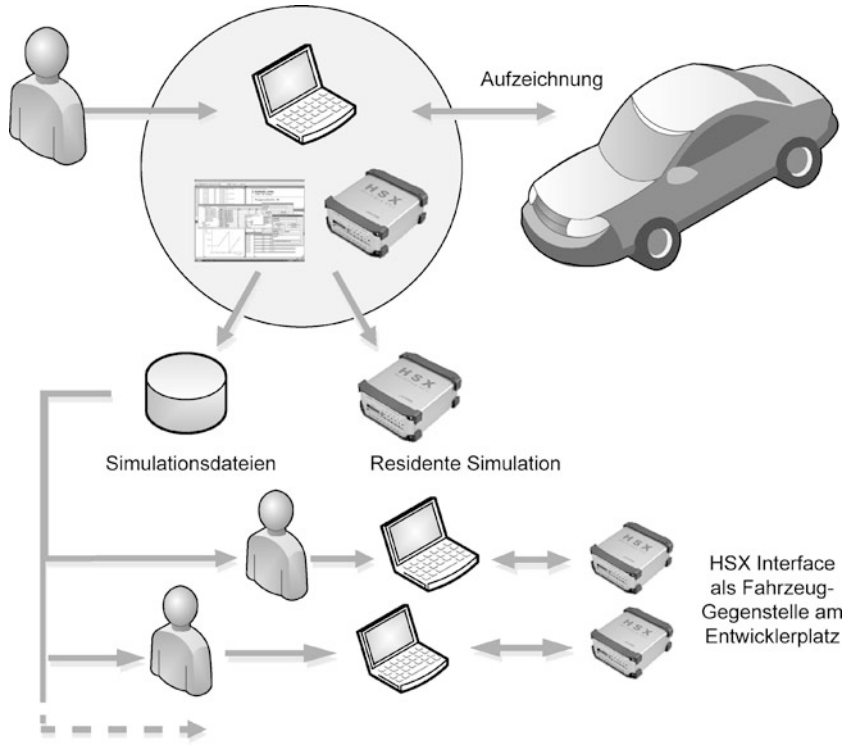


Abb. 9.36 Simulation des Komplettfahrzeuges mit *samDia* und *HSX Interface*

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.

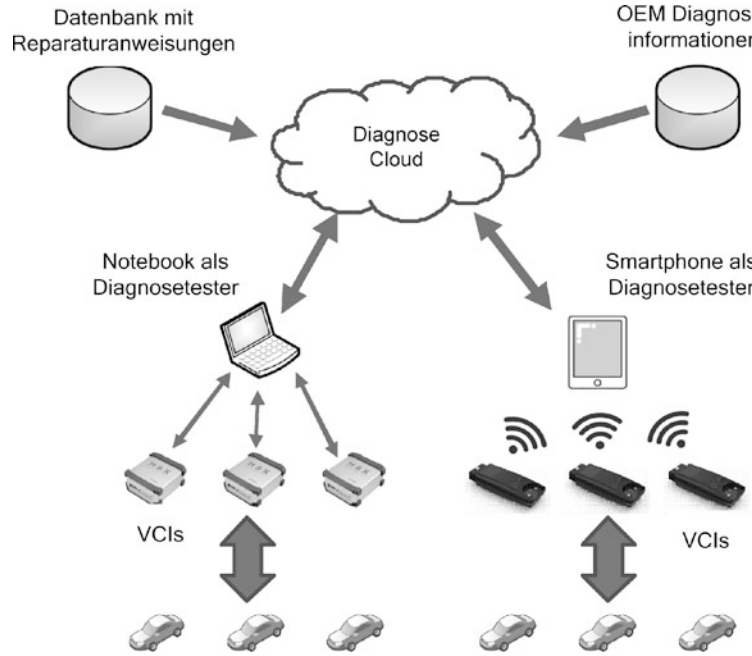


Abb. 9.37 Diagnoseapplikationen

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 Speicherkarte im VCI durchgeführt werden. Statt die Speicherkarten 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 VCI 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

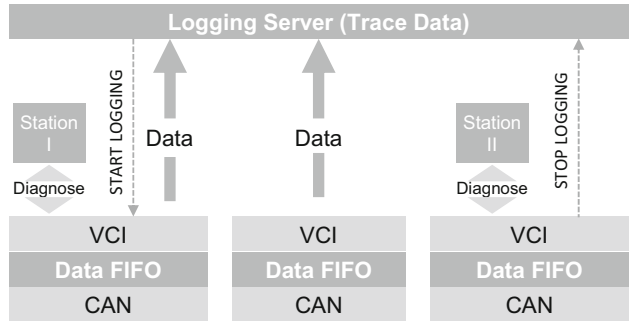


Abb. 9.38 Zentrale Protokollierung von Diagnosesitzungen in der Fertigung

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)

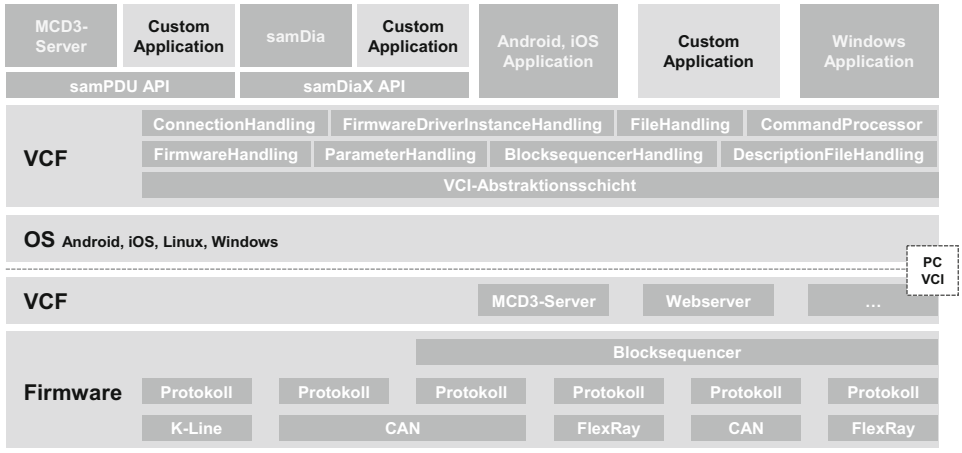


Abb. 9.39 Architektur des *Samtec VCI Communication Framework VCF*

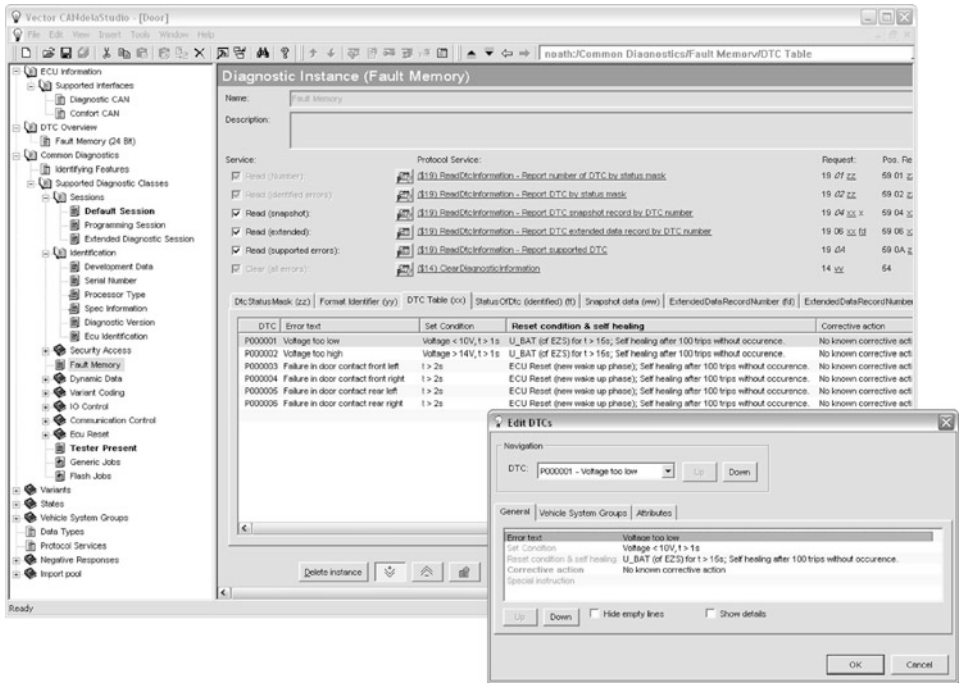


Abb. 9.40 Erstellung von Diagnosedaten (Vector Informatik *CANdelaStudio*)

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 seid 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 Editiervorlagen) und Bereitstellung komfortabler Editier- und Suchfunktionen.

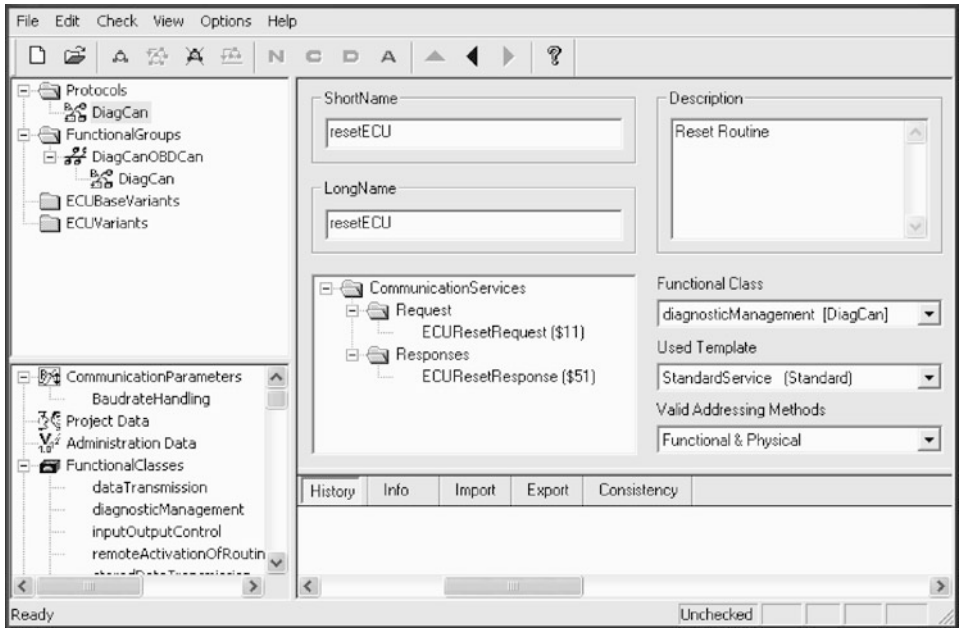


Abb. 9.41 Autorenwerkzeug für ODX-Diagnosedaten (Softing *DTS Venice*)

- 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-

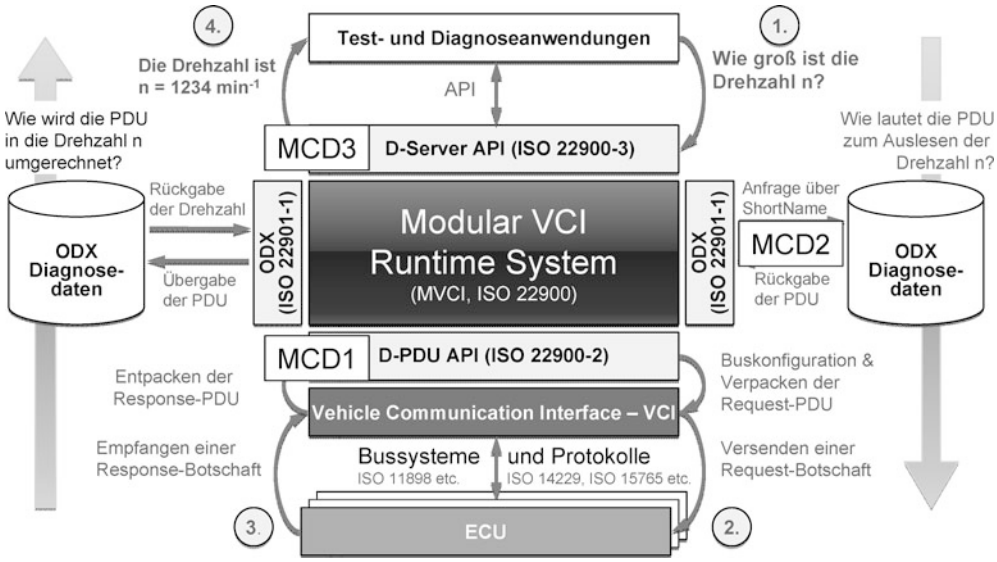


Abb. 9.42 Aufbau und Wirkungsweise eines standardisierten Diagnosesystems

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 Fehlersuchanleitungen und Ersatzteildatenbanken des Fahrzeugherstellers. Sämtliche Diagnoseschritte und die Ergebnisdaten müssen protokolliert und verwaltet werden. Nur im Zusammenwirken mehrerer Diagnosedienste 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

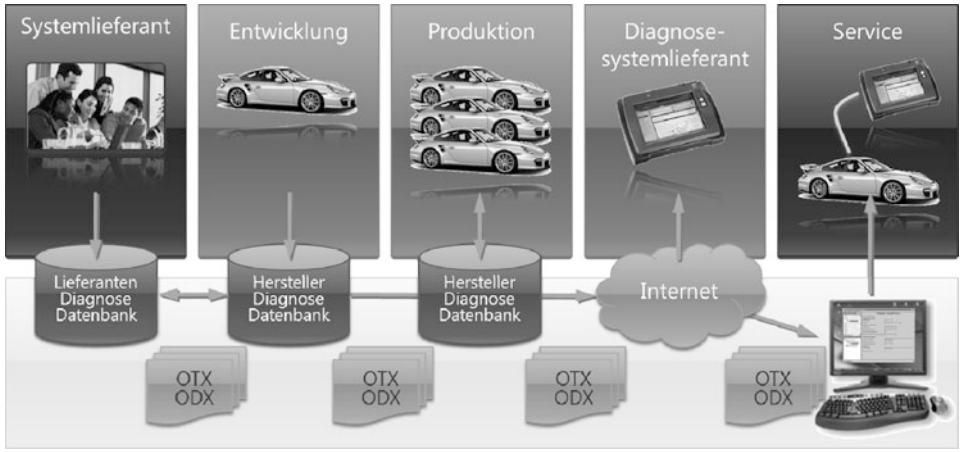


Abb. 9.43 Durchgängiger Datenaustausch in der Diagnoseprozesskette

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.

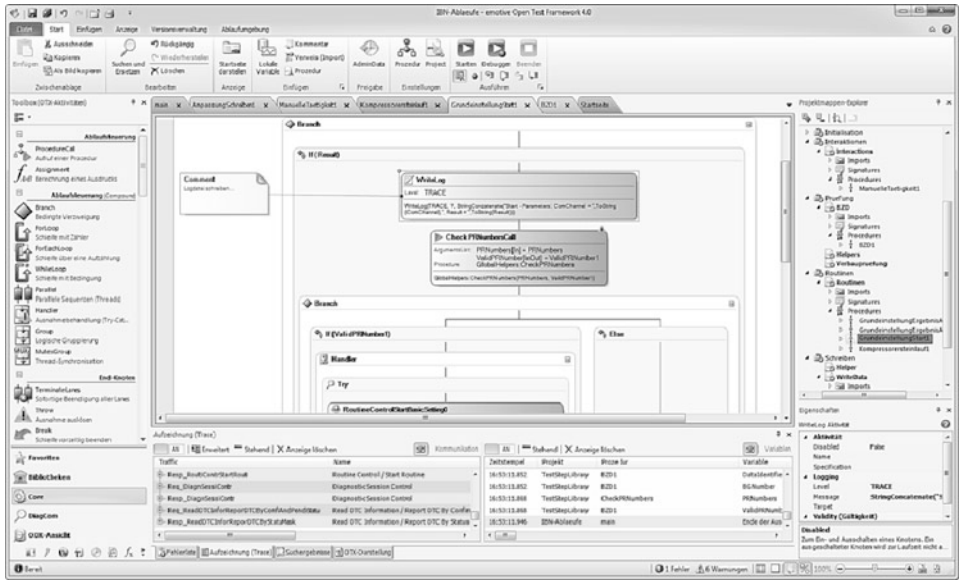


Abb. 9.44 OTX Entwicklungsumgebung *Open Test Framework*

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 Gesamtablauf 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.

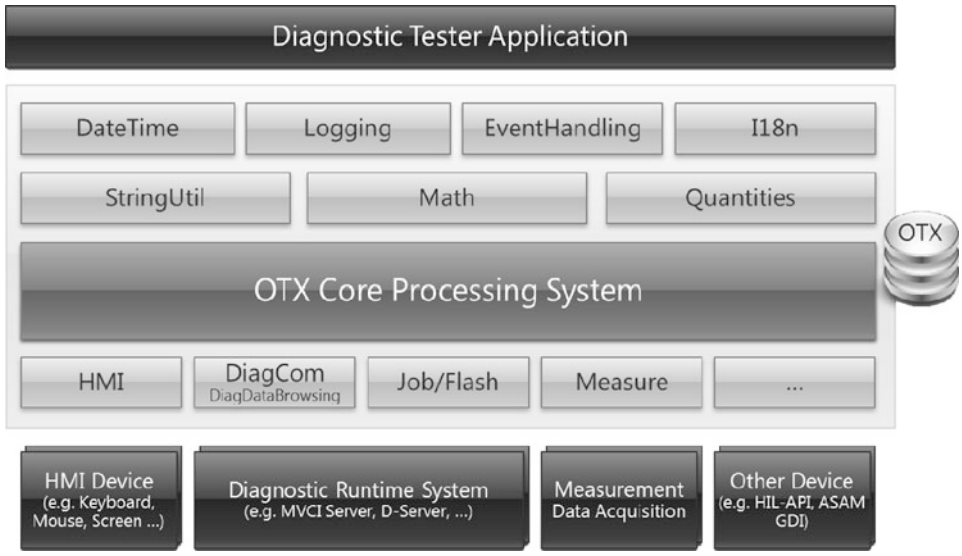


Abb. 9.45 Elemente eines OTX-Systems

- 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.

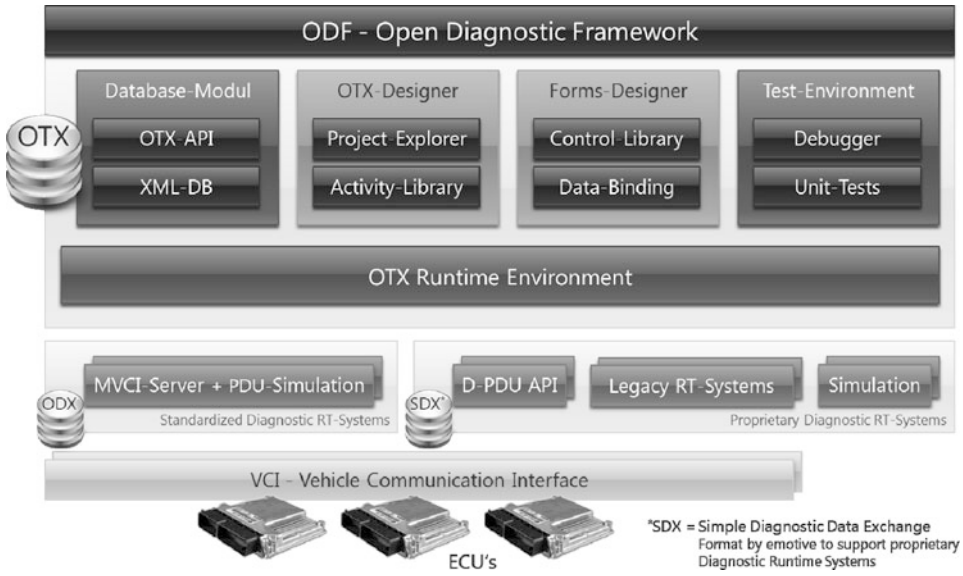


Abb. 9.46 Aufbau des *Open Test Frameworks*

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 offengelegt 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 Diagnoselaufzeitsystemen zusammenarbeitet (Abb. 9.46).

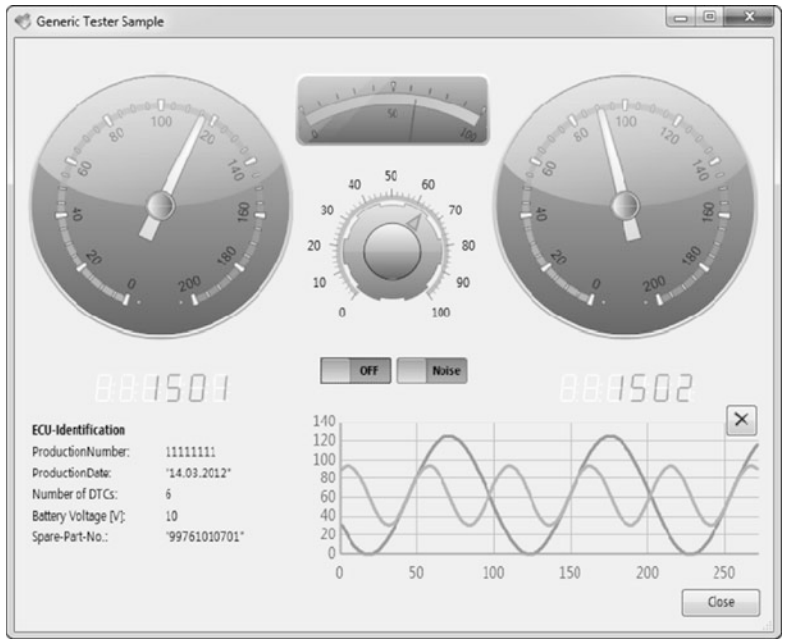


Abb. 9.47 Mit dem Forms-Designer erstellte Bedienoberfläche

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

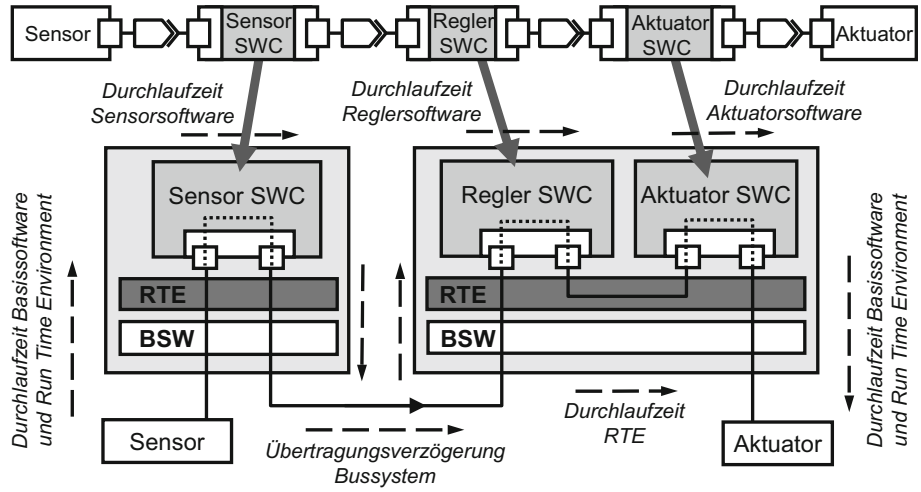


Abb. 9.49 Durchlaufverzögerungen bei einem verteilten AUTOSAR-System

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 Strukturbaum 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 *CANoe* aufzeichnet wurden, in den *Symtavision TraceAnalyzer* eingelesen und mit den Entwurfsdaten verglichen. In der Praxis zeigt sich nämlich, dass Zeitanforderungen bei der

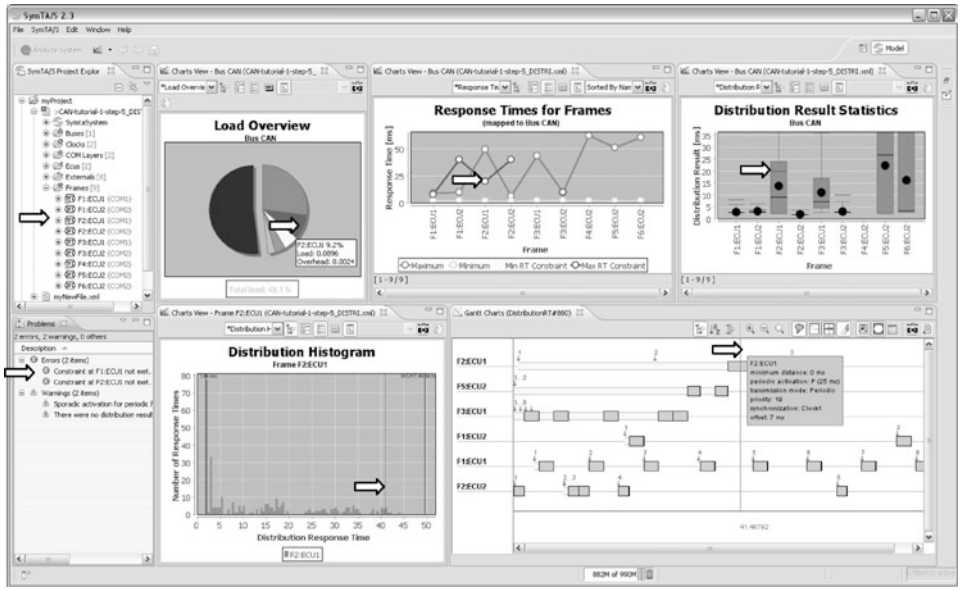


Abb. 9.50 Schedulinganalyse-Werkzeug *SymTA/S* von Symtavision

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