Prof. Dipl.-Ing. Dr. Wolfgang PREE

Softwareentwicklung für verteilte Systeme im Automobil

Inhaltsverzeichnis

Zielformulierung

Das Ziel dieser Lektion ist es, Ihnen ein Grundverständnis über die verteilten Systeme im Automobil zu vermitteln, das Ihnen hilft, technisch fundierte Entscheidungen bei Software-Entwicklungsprojekten und bei der Auswahl von Methoden und Werkzeugen zu treffen. Das Kapitel führt zunächst in die Charakteristika verteilter Softwaresysteme ein und zeigt deren Merkmale anhand der typischerweise im Automobil vorhandenen verteilten Systeme auf. Ein Fokus dieses Kapitels liegt auf dem FlexRay-Kommunikationsstandard, der in Zukunft vermehrt im Automobil eingesetzt werden dürfte, da sich damit Nachteile und Einschränkungen bestehender Kommunikationssysteme, wie zum Beispiel CAN, im Automobil vermeiden lassen und Engpässe beim Datentransfer überwunden werden können. Das Ziel dabei ist nicht, Ihnen die zahlreichen Details des FlexRay-Standards zu vermitteln, sondern vielmehr die relevanten Eigenschaften herauszuarbeiten, um solide zu verstehen, warum und wie sich Software-Entwicklung für FlexRay von bisherigen Ansätzen konzeptionell unterscheidet. Wir gehen auf zwei unterschiedliche Methoden zur Entwicklung von FlexRay-basierter Software und die darauf aufbauenden Werkzeuge ein: Die eine Methode beruht auf dem konventionellen Paradigma, zuerst die Plattform festzulegen, und dann die Software darauf maßzuschneidern. Im Gegensatz dazu erlaubt eine zukunftsweisende Methode, zuerst die Software zu modellieren, also das Verhalten zu definieren, und dann erst die Software mittels automatischer Code-Generatoren auf eine spezifische Plattform abzubilden.

Im Text werden folgende Symbole zum Hervorheben diverser Aspekte verwendet:

Merksätze, Definitionen und wichtige Konzepte

Querverweise zu anderen Kapiteln und Abschnitten

Beispiele oder Fallstudien

Literaturempfehlungen

Quintessenz am Ende jedes Hauptkapitels

Web-Link

1 Charakteristika verteilter Systeme im Automobil

Ein verteiltes System „besteht aus mehreren autonomen Prozessor-Speicher-Systemen, die mittels Nachrichtenaustausch kooperieren“ [Mühlhäuser, 2006]. Die autonomen Prozessor-Speicher-Systeme sind die **Knoten** des verteilten Systems. Mit dem Begriff Prozessor-Speicher-System ist typischerweise ein vollständiger Computer gemeint. In der Automobilbranche wird dafür häufig der Begriff ***Electronic Control Unit*** **(ECU)** verwendet.

Ein verteiltes System besteht aus mehreren Knoten (= ECUs), die mittels Nachrichtenaustausch kooperieren.

Ein Knoten kann auch aus mehreren Prozessoren („*Multi-Core*“) bestehen. Hingegen ist laut dieser Definition ein Computer, der nur über mehrere Prozessoren verfügt, die über einen gemeinsamen Speicher kommunizieren, alleine kein verteiltes System, da für die Kommunikation zwischen den Prozessoren kein Nachrichtenaustausch nötig ist und die Prozessoren alleine keine autonomen Prozessor-Speicher-Systeme darstellen.
Damit ein Knoten mit anderen Knoten durch den Austausch von Nachrichten kommunizieren kann, müssen die Knoten durch eine **Kommunikationsinfrastruktur** miteinander verbunden sein. Abbildung 1 zeigt exemplarisch ein verteiltes System, das aus fünf Knoten besteht, die ringförmig angeordnet sind. Die Verbindungslinie zwischen jeweils zwei Knoten stellt schematisch die Kommunikationsinfrastruktur dar. Die Knoten zusammen mit der Kommunikationsinfrastruktur werden oft auch als **Computernetz(werk)** bezeichnet.
Aufgrund der Verbindungen in unserem Beispiel sehen Sie, dass zum Beispiel Knoten1 nur mit Knoten2 und mit Knoten5 direkt kommunizieren kann. Um eine Nachricht von Knoten1 an Knoten3 oder an Knoten4 zu senden, muss die Nachricht von zumindest einem der anderen Knoten weitergeleitet werden. Diese Überlegungen führen uns zum Begriff der Topologie eines verteilten Systems.

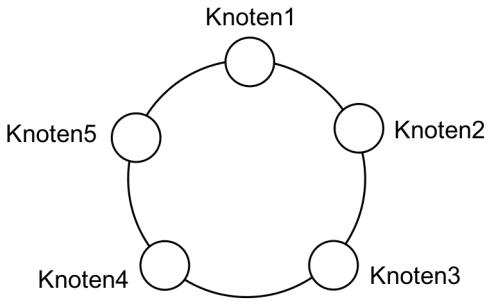

Abbildung 1. Beispiel für ein verteiltes System mit 5 Knoten.

Unter **Topologie** versteht man bei verteilten Systemen die Art der Verbindungen zwischen den Knoten. Die Topologie des verteilten Systems in Abbildung 1 wird naheliegend als Ring bezeichnet. Die Vorteile der **Ring-Topologie** sind, dass Vorgänger und Nachfolger eines Knoten definiert sind und ein Ausfall eines

Knoten das Kommunikationssystem nicht lahmlegt. Allerdings ist die Übertragung von Nachrichten zu entfernten Knoten aufwändig.
Abbildung 2 zeigt wie fünf Knoten in einer **Stern-Topologie** mit einem zusätzlichen zentralen Koordinator-Knoten angeordnet werden können. Die Topologie ist wiederum entscheidend, welcher Knoten mit welchem anderen Knoten direkt oder indirekt kommunizieren kann. In diesem Fall kann nur der zentrale Koordinator-Knoten mit jedem der anderen Knoten direkt kommunizieren. Vorteile dieser Topologie sind eine gute Erweiterbarkeit und eine einfache Fehlersuche. Der Nachteil dieser Topologie ist, dass der Ausfall des Koordinator-Knotens die Kommunikation zwischen den anderen Knoten unmöglich macht.

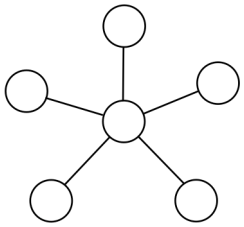

Abbildung 2. Beispiel für eine Stern-Topologie.

Abbildung 3 zeigt die Anordnung von 5 Knoten in einer sogenannten **Bus-Topologie**. Alle Knoten sind an einem Hauptkabel, dem Bus, angeschlossen. Damit steht die von einem Knoten übermittelte Nachricht, also die auf den Bus gelegte Nachricht, allen anderen Knoten am Bus zur Verfügung. Es hängt vom sogenannten Protokoll ab, ob Kollisionen auftreten können, also ob mehrere Knoten gleichzeitig versuchen können, Nachrichten über den Bus zu senden -oder nicht. Ein Vorteil der Bus-Topologie ist die Ausfallsicherheit bezogen auf die Knoten: es können ein oder mehrere Knoten ausfallen, ohne die Kommunikation zu beeinträchtigen. Ein Bussystem kann auch leicht um Knoten erweitert werden, zumindest bis zu einer bestimmten Obergrenze. Weiters ist die Verkabelung einfach. Das reduziert die Herstellungskosten und bewirkt eine Gewichtsreduzierung der Fahrzeuge. Aus der einfachen Verkabelung resultiert jedoch der Nachteil der geringen Ausfallsicherheit bei Problemen mit dem Hauptkabel. Wenn das Hauptkabel unterbrochen oder defekt ist, kann die Kommunikation unmöglich werden.

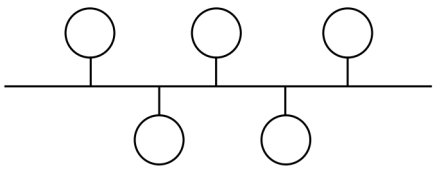

Abbildung 3. Beispiel für eine Bus-Topologie.

Die Bus-Topologie wird häufig in Automobilen verwendet, sowohl für den Nachrichtenaustausch („Daten-Bus“) als auch als paralleler zusätzlicher Bus zur Stromversorgung. Die ECU-Knoten werden über den Datenbus verbunden, die von den ECU-Knoten gesteuerten Komponenten werden an das Strom-Hauptkabel angeschlossen. Nachfolgend meinen wir mit Bussystem ausschließlich den Daten-Bus.

Auf weitere Topologien gehen wir nicht näher ein, da sie im Automobil kaum eine Rolle spielen. Ein Beispiel für eine weitere Topologie ist jene, bei der jeder Knoten mit allen anderen Knoten verbunden ist.
Was hingegen typischerweise in Automobilen vorkommt, ist die Verbindung zwischen zwei oder mehreren verteilten Systemen. Das kann zum Beispiel über ausgewiesene Knoten erfolgen, die Knoten in zwei oder mehreren verteilten Systemen sind. Solche Knoten werden oft als **Gateway**(-Knoten) bezeichnet, wenn sich die Bussysteme so unterscheiden, dass quasi eine Übersetzung nötig wird, um Nachrichten auszutauschen. Abbildung 4 zeigt die Verbindung zweier Bussysteme Bus1 und Bus2 über einen gemeinsamen, in der Abbildung grau schraffierten Knoten.

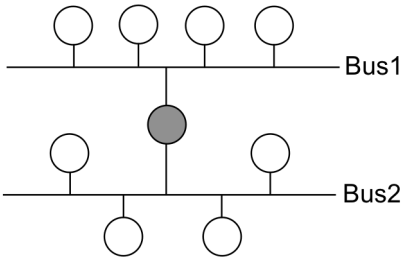

Abbildung 4. Verbindung zweier Bussysteme über einen gemeinsamen Knoten.

Weitere Charakteristika verteilter Systeme
Die folgenden zusätzlichen Charakteristika von verteilten Systemen sind insbesondere für deren Einsatz im Automobil relevant:

- die Übertragungskapazität, auch oft als „Bandbreite“ bezeichnet, also wieviele Bits pro Zeiteinheit übertragen werden können
- die Anzahl der adressierbaren Knoten (oben erwähnt als Obergrenze, bis zu der Knoten in einer Bus-Topologie leicht hinzugefügt werden können)
- die Echtzeitfähigkeit, also ob Nachrichten mit garantiertem Zeitverhalten übertragen werden können (siehe nachfolgender Abschnitt 1.1)
- die Störsicherheit, bezogen auf elektromagnetische Verträglichkeit (EMV) und die Toleranz von Versorgungsspannungsschwankungen
- die Abhörsicherheit, die bisher allerdings bei Automobilen kein Thema ist

Bezogen auf die angeführten Kriterien finden Sie in Abschnitt 1.3 einen Vergleich folgender Standards zur Realisierung verteilter Systeme: CAN, byteflight, LIN, FlexRay und MOST.

1.1 Ereignissteuerung versus Zeitsteuerung

Damit Sie den Unterschied zwischen ereignisgesteuerten (= *event triggered*, Ereignis-getrieben) und zeitgesteuerten (= *time triggered*, Zeit-getrieben) verteilten Systemen verstehen, definieren wir in diesem Abschnitt die grundlegenden Begriffe und veranschaulichen diese anhand von Beispielen.
Ein ereignisgesteuertes System reagiert auf ein Ereignis, das zu einem unvorhersehbaren Zeitpunkt auftritt, so rasch als möglich, jedoch ohne Garantie, wie lange die Reaktion maximal dauert. Als Beispiel betrachten wir die Betätigung eines Schalters, mit dem der Rückspiegel eingestellt wird. In Abbildung 5 wird der

unvorhersehbare Zeitpunkt, zu dem dieses Ereignis eintritt, mit einem Punkt auf der Zeitachse markiert.

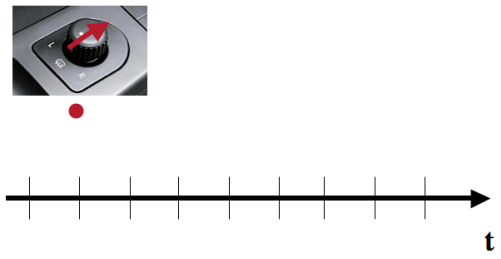

Abbildung 5. Betätigung eines Schalters als Beispiel für ein Ereignis.

Abbildung 6 zeigt schematisch die Dauer, bezeichnet mit dem Buchstaben d, der Reaktion auf das Ereignis. In diesem Beispiel dauert die Reaktion drei Zeiteinheiten, wobei die Dauer einer Zeiteinheit hier nicht näher spezifiziert ist.

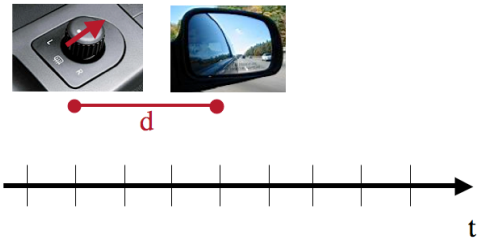

Abbildung 6. Betätigung eines Schalters als Beispiel für ein Ereignis.

Wenn mehrere Ereignisse gleichzeitig oder in knapper Abfolge auftreten, reicht möglicherweise die Rechenkapazität beziehungsweise die Übertragungskapazität auf dem Kommunikationssystem nicht aus, um rasch genug auf die jeweiligen Ereignisse zu reagieren. Abbildung 7 illustriert dies anhand unseres Beispiels: Wenn durch eine Bewegung des Lenkrads das Kurvenfahrlicht adjustiert werden muss, kann die Einstellung des Rückspiegels länger dauern, als wenn kein zusätzliches Ereignis auftritt.

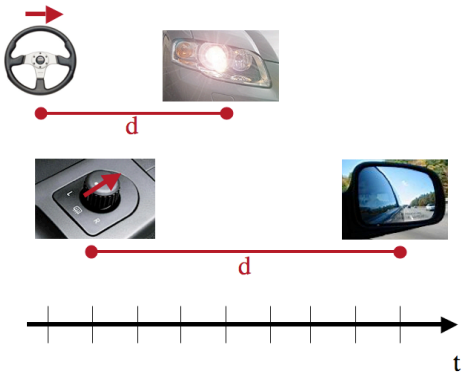

Abbildung 7. Reaktion auf zwei Ereignisse.

Ereignisgesteuerte Systeme können deswegen nicht garantieren, wie lange eine Bearbeitung eines Ereignisses dauert, da das Auftreten von Ereignissen unvorhersehbar ist.

Im Gegensatz dazu löst bei einem zeitgesteuerten System das Fortschreiten der Zeit periodisch die Ereignisse aus, auf die reagiert werden muss. Da das Fortschreiten der Zeit vorhersehbar ist, sind diese Ereignisse vorhersehbar. Wenn zum Beispiel ein Ereignis mit einer Zeitperiode von einer Millisekunde vorgesehen ist, kann von einem definierten Startzeitpunkt an das Ereignis jede Millisekunde eingeplant werden. Das macht eine exakte Planung der Rechen- und Kommunikations-Ressourcen möglich, sodass bei funktionstüchtiger Hardware die Reaktionsdauer auf Ereignisse exakt vordefiniert werden kann. Abbildung 8 veranschaulicht ein zeitgesteuertes System, in dem jede Zeiteinheit ein Ereignis vorgesehen ist, auf das zum Beispiel durch Lesen des Wertes eines Sensors und seine Verarbeitung reagiert wird.

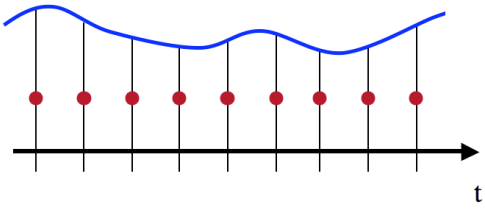

Abbildung 8. Beispiel für Ereignisse, die durch Fortschreiten der Zeit ausgelöst werden.

Ein verteiltes System ist zeitgesteuert (= zeitgetrieben), wenn das Fortschreiten der Zeit der Auslöser von Reaktionen ist. Das setzt eine globale Zeitbasis voraus. Mit anderen Worten formuliert bedeutet das, dass die Uhren der einzelnen ECU-Knoten synchronisiert sein müssen.

Wann ist welches Paradigma adäquat?

Nun stellt sich die Frage, für welche Art von Software-Komponenten welches Paradigma, die Ereignissteuerung oder die Zeitsteuerung, besser geeignet ist. Allgemein lässt sich diese Frage nicht beantworten. Grundsätzlich gilt, dass Anwendungen, die nicht sehr zeitkritisch sind und unregelmäßig auftreten, ereignisgesteuert konzipiert werden sollten. Dazu zählen zum Beispiel das Einstellen der Rückspiegel, das Betätigen der Fensterheber oder das Betätigen der Zentralverriegelung eines Fahrzeuges. Zeitkritische Anwendungen, die aber sehr selten oder am besten nie ausgeführt werden müssen, eignen sich auch gut für das ereignisgesteuerte Paradigma. Dazu zählen zum Beispiel das Auslösen der Air-Bags oder das Straffen der Gurte bei einem Aufprall. Damit auf solche Ereignisse innerhalb der von der Physik vorgegeben Zeitschranken reagiert werden kann, können **Prioritäten für Ereignisse** definiert werden. Solche Ereignisse müssen höchste Priorität haben und vor anderen zu diesem Zeitpunkt eventuell auftretenden Ereignissen behandelt werden.

Das zeitgesteuerte Paradigma ist notwendig, wenn zum Beispiel die Stabilität eines Regelkreises eine bestimmte Zeitperiode verlangt, die möglichst nicht variieren

soll. Ein Beispiel dafür sind die in Zukunft geplanten X-by-Wire-Systeme, deren Regler das zeitgesteuerte Lesen von Sensoren, das zeitgesteuerte Verarbeiten der Eingaben und das zeitgesteuerte Schreiben der Aktuatoren benötigen.

Probleme mit Prioritäten
Auf den ersten Blick könnte man vermuten, dass Software für ereignisgesteuerte verteilte Systeme im Automobil leicht durch folgende Vorgehensweise korrekt implementierbar wäre: Man teile einfach die diversen Funktionen (Fensterhebersteuerung, Air-Bag-Steuerung, etc.) in Kategorien ein, denen man eine bestimmte Priorität zuweist. Wenn dann das System im Automobil ausgeführt wird, werden die wichtigeren Ereignisse vor den unwichtigeren Ereignissen behandelt. Dieser Ansatz wurde und wird im Prinzip auch von den Automobil-Herstellern und -Zulieferern verfolgt.

Die Erfahrung hat jedoch gezeigt, dass diese Vorgehensweise in der Praxis leider nicht ausreicht, um Software-Probleme zu vermeiden. Es treten „ab und zu“ und leider nicht vorhersehbar und auch nicht reproduzierbar Konstellationen auf, die zu Pannen oder kritischen Situationen führen. Ein möglicher Grund dafür ist das bekannte **Phänomen der Prioritäten-Invertierung**:

Nehmen wir an, wir hätten drei Software-Komponenten A, B und C, deren Prioritäten mit A > B > C festgelegt sind und die auf einem ECU-Knoten ausgeführt werden. Das bedeutet, dass zu einem Zeitpunkt nur jeweils eine Komponente den Prozessor benutzen kann. A und C teilen sich außerdem eine kritische Ressource, die zu einem Zeitpunkt nur von einer Komponente benutzt werden darf. Das könnte zum Beispiel ein Stellmotor (Aktuator) einer Bremse sein. Nun nehmen wir an, dass C gerade die kritische Ressource benutzt. Dann wird durch ein Ereignis, dessen Zeitpunkt nicht vorhersehbar ist, A gestartet und A unterbricht aufgrund der höheren Priorität die Ausführung von C. A will nun auf die kritische Ressource zugreifen. Deswegen muss C wieder ausgeführt werden, um die kritische Ressource freizugeben. A wartet inzwischen auf C. Jetzt tritt ein weiteres Ereignis auf, das die Ausführung von B bewirkt. Da B eine höhere Priorität als C hat, wird die Ausführung von C (also die Freigabe der kritischen Ressource) unterbrochen und damit die Ausführung von A verzögert. Es kann also eine Komponente mit mittlerer Priorität (B) die Ausführung einer Komponente mit höherer Priorität (A) beliebig lange verzögern, wenn Ereignisse zu „ungünstigen“ Zeitpunkten auftreten. In diesem Fall wird A solange blockiert, bis B fertig ist, also den Prozessor nicht mehr benötigt. Dann erst kann C die kritische Ressource für A freigeben und A die Verarbeitung fortsetzen. Da bei einer solchen Konstellation die Prioritäten von A und B vertauscht werden, nennt man dieses Phänomen Prioritäten-Invertierung (*Priority Inversion*). Es könnte vielleicht theoretisch mittels Wahrscheinlichkeitsrechnung bestimmt werden, wie häufig statistisch solche Fehlerfälle auftreten, also zum Beispiel alle 100.000 Betriebsstunden. Aufgrund der hohen ausgelieferten Stückzahlen und damit Betriebsstunden treten solche auf den ersten Blick sehr unwahrscheinlichen Konstellationen dann in der Praxis doch auf und führen zu den bekannten, kaum reproduzierbaren Qualitäts-problemen.

Ereignissteuerung und die damit verbundene Notwendigkeit, Prioritäten zu definieren, kann zu nicht reproduzierbaren Software-Problemen wie zum Beispiel der Invertierung von Prioritäten führen.

Neben der Prioritäten-Invertierung gibt es weitere bekannte Probleme, die beim gängigen Stand der Technik der Software-Entwicklung für Automobile zu Qualitätsproblemen führen können. Ein Beispiel sind sogenannte ***Deadlocks***, also Systemverklemmungen, bei denen sich Software-Komponenten gegenseitig so blockieren, dass überhaupt keine Verarbeitung mehr erfolgt. Ein Beispiel für einen *Deadlock* im Alltag ist das Verhalten von Autofahrern bei einer Kreuzung zweier Strassen mit vier Stopp-Tafeln. Solchen Kreuzungen sind in den U.S.A. häufig vorzufinden. Wenn sich zwei oder mehr Fahrzeuge den Stopp-Tafeln nähern, hat jenes Fahrzeug Vorrang, das zuerst bei einer der Stopp-Tafeln stehen bleibt. Das funktioniert im Prinzip recht gut, bis auf den Fall, wenn aus jeder der vier Richtungen ein Fahrzeug auf die Stopp-Tafel zufährt und alle vier Fahrzeuge gleichzeitig stehenbleiben. Dann ist ein *Deadlock* eingetreten. Für die Lenker der Fahrzeuge ist das aber auch meist kein Problem, da man dann über Gesten, wie zum Beispiel dem Zuwinken mit der Hand, vereinbart, wer als erster wegfährt, wer als nächster, und so weiter, bis der *Deadlock* aufgelöst ist. Wären nun die Fahrer beziehungsweise Fahrzeuge Software-Komponenten, ist das Auflösen nicht trivial. Selbst das Erkennen eines *Deadlocks* ist in Software-Systemen schwierig oder sogar unmöglich.

Das Beispiel *Deadlock* zeigt ein weiteres Problem bei der Software-Entwicklung von Echtzeitsystemen auf. Wenn das Zeitverhalten von der Hardware abhängt, was heute typischerweise der Fall ist, kann eine schnellere Hardware zu langsamerer Software führen. Wenn bei einem bestimmten Prozessor kein *Deadlock* aufgetreten ist, kann ein schnellerer Prozessor dazu führen, dass „Fahrzeuge gleichzeitig bei der Kreuzung ankommen“, sich Software-Komponenten also verklemmen und ein *Deadlock* auftritt. In diesem Fall, der vermutlich statistisch auch selten auftritt, führt schnellere Hardware zu unendlich langsamer Software, die klemmt und überhaupt keine Verarbeitung mehr durchführt.

Die beiden exemplarisch herausgegriffenen Phänomene, die inhärent mit ereignis-gesteuerter, prioritäten-behafteter Software-Entwicklung verbunden sind, führen uns dazu, zu definieren, welche Eigenschaften Software im Automobil erfüllen sollte, um eine ähnlich hohe Qualität wie die anderen Komponenten eines Automobils aufzuweisen.

1.2 Software-Eigenschaften, die Automobil-Software erfüllen sollte

Das korrekte Funktionieren von Software bezieht sich immer auf die Erfüllung der möglichst exakt spezifizierten Anforderungen. Unabhängig davon sind Eigenschaften wie Determinismus und Kompositionalität gefordert, die wir hier definieren wollen.
Determinismus bedeutet, dass gleiche Eingaben immer zu korrespondierenden gleichen Ausgaben führen. Diese Definition bezieht sich auf die Eingabe- und Ausgabe-Werte. Damit kann präziser von Wert-Determinismus gesprochen

werden. Da bei Software im Automobil auch oft das Zeitverhalten ein wesentlicher Faktor ist, ob die Software korrekt funktioniert, muss die Definition entsprechend erweitert werden.

Echtzeitsoftware ist deterministisch, wenn gleiche Eingaben zu gleichen Zeitpunkten immer zu korrespondierenden gleichen Ausgaben zu gleichen Zeitpunkten führen.

Wenn eine derartige eindeutige Abbildung vorliegt, ist die Software Wert- und Zeit-deterministisch. Da bei der heutigen Entwicklung von Echtzeit-Software der Ansatz verfolgt wird, dass die Software dann korrekt ist, wenn Reaktionen auf Ereignisse innerhalb einer bestimmten Zeit (*Deadline*-basierte Programmierung) erfolgen, ist die Software weder Wert- noch Zeit-deterministisch. Sie können sich den Grund dafür anhand eines einfachen Beispiels veranschaulichen. Abbildung 9 zeigt die konkrete Dauer der Reaktion auf ein Ereignis und die Deadline, die dafür definiert wurde. Die Reaktion dauert in diesem Beispiel drei Zeiteinheiten (d1). Maximal darf die Reaktion sechs Zeiteinheiten dauern.

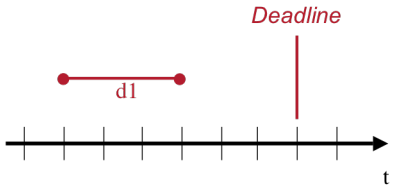

Abbildung 9. Deadline für die Reaktion auf ein Ereignis

Das bedeutet, dass eine Reaktionsdauer von zum Beispiel fünf Zeiteinheiten ebenso korrekt ist, weil die Reaktion auf das Ereignis vor der *Deadline* erfolgt (siehe Abbildung 10).

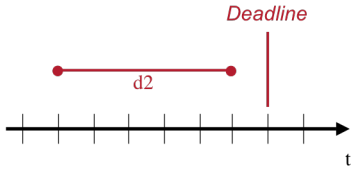

Abbildung 10. Reaktionsdauer, die auch die *Deadline* einhält.

Wenn nun eine zweite Berechnung das Ergebnis der Reaktion auf dieses Ereignis verwendet, hängt der Wert, der für die Berechnung verwendet wird, davon ab, wie schnell die Reaktion tatsächlich ist. Abbildung 11 zeigt, dass bei kürzerer Reaktionsdauer d1 der Wert für die andere Berechnung bereits zur Verfügung steht, bei längerer Reaktionsdauer d2 aber nicht und zum Beispiel ein Wert aus einer früheren Reaktion herangezogen wird. Das System ist daher nicht Wert-deterministisch. Da die Ergebnisse von Berechnungen an andere Berechnungen weitergeben werden, sobald sie vorliegen, ist das System auch nicht Zeit-deterministisch.

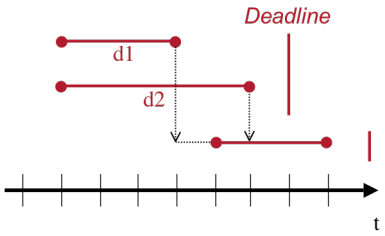

Abbildung 11. Nicht-deterministisches Verhalten.

Kapitel 4 zeigt, wie mit Hilfe einer sogenannten logischen Ausführungszeit das Verhalten von Software sowohl Zeit- als auch Wert-deterministisch wird.

Kompositionalität bedeutet, dass Software-Komponenten zu einem System hinzugefügt werden können, ohne dass sich das (Zeit- und Wert-)Verhalten der anderen Komponenten ändert. Wenn andere Komponenten nicht von einer Komponente K abhängen, muss ebenso gelten, dass K von einem System entfernt werden kann, ohne dass sich das (Zeit- und Wert-)Verhalten der anderen Komponenten ändert.

1.3 Eigenschaften von Kommunikationssystemen im Automobil

Dieser Abschnitt gibt einen Überblick über die heute typischerweise in Automobilen zum Einsatz kommenden Kommunikationssysteme: das Controller Area Network (CAN), byteflight, LIN, MOST und FlexRay. Die genannten Systeme werden insbesondere dahingehend verglichen, ob sie Ereignissteuerung oder Zeitsteuerung unterstützen, was die wesentlichen Merkmale ihrer sogenannten „Protokolle“ (siehe unten) sind, welche Topologien definiert werden können, welche Übertragungskapazität möglich ist und wofür sie heute typischerweise verwendet werden.
Ein Netzwerkprotokoll, oder kurz Protokoll, definiert, nach welchen Regeln Nachrichten zwischen den Knoten im verteilten System ausgetauscht werden. Zum besseren Verständnis, was ein Protokoll ist, denken wir an übliche Regeln, die beim Telefonieren angewendet werden:

Wenn Sie angerufen werden, melden Sie sich typischerweise mit Ihrem Namen. Damit hört der anrufende Gesprächspartner, dass die Verbindung aufgebaut ist und beginnt mit dem Gespräch. Wenn Leitungsstörungen auftreten, wird zum Beispiel nachgefragt, ob man verstanden wurde oder der Gesprächspartner ersucht, etwas zu wiederholen. Weiters hält man sich am besten an die Regel, dass zu einem Zeitpunkt nur ein Gesprächspartner spricht. Wenn dann beide versehentlich gleichzeitig zu sprechen beginnen, einigt man sich, wer das Wort bekommt. Das sind alles Beispiele für „Meta-Kommunikationen“, die es ermöglichen, Informationen zwischen zwei oder mehreren Gesprächspartnern möglichst reibungslos auszutauschen.

Was ein Mensch mehr oder weniger intuitiv meistert, wird in einem Netzwerkprotokoll in Form von exakt definierten Regeln formalisiert. Im folgenden werden im Abschnitt über das Protokoll die relevanten Aspekte der beim jeweiligen Kommunikationssystem angewandten Regeln erläutert, um die Möglichkeiten und Grenzen des Kommunikationssystems verstehen zu können.

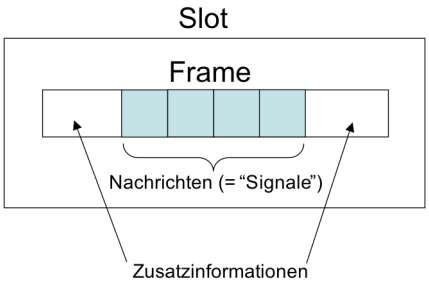

Abbildung 12. Grundbegriffe bei automobilen Kommunikationssystemen.

Weiters sind die Begriff Nachricht, Signal, *Frame* und *Slot* zu definieren (siehe Abbildung 12). Die Informationseinheit, die über ein Kommunikationssystem übertragen wird, bezeichnet man bei den betrachteten Systemen im Automobil meist als *Frame*. Ein *Frame* kann eine oder mehrere Nachrichten enthalten, also die eigentliche Information, die ausgetauscht werden soll. Manchmal wird auch der Begriff *Payload* synonym für die in einem *Frame* übertragenen Nachrichten verwendet. Im Automobilbereich verwendet man zudem oft den Begriff Signal synonym zu Nachricht. Ein *Frame* wird in einem sogenannten *Slot* des Kommunikationssystems übertragen.

1.3.1 CAN (Controller Area Network)

CAN wurde in den 1980er Jahren von der Robert Bosch GmbH als Basis zur Vernetzung von ECU-Knoten im Automobil entwickelt. CAN kommt heute nicht nur im Automobil sondern auch in anderen Bereichen, wie der Industrieautomatisierung zum Einsatz.

Protokoll. Das CAN zugrundeliegende Protokoll ist für ereignisgesteuerte, nichtdeterministische Software konzipiert worden. Jeder Knoten kann uneingeschränkt zu jeder Zeit senden. Wenn zwei oder mehrere Knoten gleichzeitig senden, werden die *Frames* gemäß ihrer Priorität versendet. Das Protokoll wird im Fachjargon CSMA/CA (*Carrier Sense Multiple Access/Collision Avoidance*) genannt. *Carrier Sense* bedeutet, dass von Knoten geprüft werden kann, ob Nachrichten übertragen werden. Bei CAN werden Kollisionen durch die sogenannte Arbitrierung vermieden: Jeder *Frame* hat eine Identifikation (ID). Dazu können entweder 11 Bits (CAN 2.0A) oder 29 Bits (CAN 2.0B) verwendet werden. Wenn nun ein Knoten die Identifikation eines Frames sendet, überwacht er den Bus. Wenn währenddessen ein weiterer Knoten zu senden beginnt, wird aufgrund der Priorität entschieden, welcher Knoten weiter überträgt: je niedriger die Identifikationsnummer ist, umso höher ist die Priorität der Nachricht. Die Nachricht mit der höchsten Priorität wird gesendet. Die Knoten, die zu diesem Zeitpunkt eine Nachricht mit niedriger Priorität senden wollten, und daher

aufgrund der Priorität unterlegen waren, wiederholen nach einer zufällig gewählten Zeitspanne den Sendevorgang.

Nachdem die Identifikation eines Frames übertragen wurde und aufgrund dessen gegebenenfalls durch Arbitrierung entschieden wurde, dass die Priorität im Vergleich zu den anderen währenddessen abgesendeten Frames am höchsten ist, wird bei CAN das Senden der restlichen Informationen eines Frames nicht unterbrochen. Das kann bewirken, dass selbst das Senden eines Frames mit höchster Priorität verzögert wird, weil gerade ein anderer Frame versendet wird. Die Konsequenz des beschriebenen Protokolls ist, dass unabhängig von der Priorität von Frames unvorhersehbare Verzögerungen beim Senden auftreten können.

Die Verzögerungen beim Senden von Nachrichten hängen von der Auslastung des Kommunikationssystems und der Priorität der Frames ab. CAN ist daher nicht deterministisch und es ist keine Kompositionalität möglich.

Zur Fehlererkennung bei der Übertragung wird die sogenannte zyklische Redundanzprüfung (*Cyclical Redundancy Check*, CRC) verwendet: Vor der Übertragung eines Frames wird der CRC-Wert aufgrund des Frame-Inhalts berechnet. Nach der Übertragung wird der CRC-Wert wiederun berechnet und mit dem im Frame mit übertragenen Wert verglichen.

Zum besseren Verständnis, welche Zusatzinformationen in einem Frame übertragen werden, zeigt Tabelle 1 den Aufbau eines CAN-Frames, bei dem 11 Bits für die Identifikation verwendet werden. In diesem Fall werden 1 + 11 + 3 + 4 + 15 + 1 + 2 + 7 = 44 Bits für die Zusatzinformationen benötigt. Wenn 8 Bytes übertragen werden, ist das Verhältnis von *Payload* zu Zusatzinformationen 64 : 44. Es werden also knapp 60% (64/108) der Übertragungskapazität zum Übertragen der Nachrichten verwendet.

Feldbezeichnung	**Länge (in Bits)**	**Anmerkung**
Frame-Beginn	1	
Identifikation	11	eindeutige ID, die der Priorität des Frames entspricht
div. Bits	3	
Datenlänge	4	Anzahl der Daten-Bytes (0 – 8)
Datenfeld	0 – 64 (= 0 – 8 Bytes)	*Payload*
Cyclic Redundancy Check	15	
CRC-Begrenzung	1	
ACK (acknowledge) Bereich	2	
End-of-frame (EOF)	7	

Tabelle 1. Aufbau eines CAN-Frames mit einer 11 Bit-Identifikation.

Topologien. Typischerweise sind die ECU-Knoten im Automobil in einer Bus-Topologie angeordnet. Eine Ring-Topologie wird für Unterhaltungsanwendungen („Infotainment“) verwendet. Die Knoten können auch in einer Stern-Topologie angeordnet werden, was zum Beispiel zur Steuerung der Knoten einer Zentralverriegelung sinnvoll sein kann.

Übertragungsraten. Mit CAN kann maximal 1 MBit/Sekunde übertragen werden, wenn das Netzwerk unter ca. 40 m lang ist. Das entspricht ca. 75.000 Bytes pro Sekunde, wenn man annimmt, dass ein Frame maximal ausgenutzt wird, also 8 Bytes Daten übertragen werden, und für die Identifikation 11 Bits verwendet werden.

Je länger das Netzwerk ist, umso länger dauert es, bis ein elektrisches Signal von einem Ende bis zum anderen Ende übertragen wird. Daher muss ein Signal mindestens solange „am Bus anliegen“, bis es eine definierte maximale Distanz zurückgelegt hat. Ist das Netzwerk länger als 40 m reduziert sich die Übertragungs-kapazität. Bis zu einer Länge von 100 m sind maximal 500 kBit/s möglich, bis zu 500 m sind maximal 125 kBit/s möglich. Dabei können aufgrund der nötigen Zusatzinformationen pro Nachricht (Start, Stopp, Identifikation, etc.) effektiv nur ca. ein Drittel für die eigentlichen Nachrichten verwendet werden: Eine Nachricht kann bis zu 8 Bytes lange sein. Dazu werden, wie zuvor erläutert, Zusatzinformationen in der Länge von 4 – 6 Bytes benötigt.

Verkabelung. Das Netzwerkkabel ist bei CAN meist als zweifaches, verdrilltes (*twisted pair*) und isoliertes Kupferkabel ausgeführt. Durch das Verdrillen wird eine ausreichende Abschirmung vor magnetischen Störungen gewährleistet. Wenn ein Kupferkabel ausfällt, kann das Netzwerk mit einem Kabel weiter betrieben werden. Alternativ kann bei langsameren CAN-Netzwerken, die zum Beispiel Eingaben von Benutzern übertragen, statt des verdrillten zweifachen Kabels auch von vorne herein nur ein Kabel vorgesehen werden. Statt Kupfer kann auch ein Glasfaserkabel verwendet werden.

CAN ist ein weltweiter Standard geworden, der in diversen ISO-Dokumenten definiert ist, wie zum Beispiel: ISO 11898-1:2003, ISO 11898-2:2003, und ISO 11898-3:2005. Ein Problem mit CAN ist, dass die Übertragungskapazität kaum mehr die heutigen und in Zukunft erwarteten Anforderungen der Automobil-industrie erfüllen kann. Die Tabelle 2 fasst die wichtigsten Eigenschaften von CAN zusammen.

Protokoll	CSMA/CA nicht deterministisch Ereignissteuerung CRC-Fehlererkennung
unterstützte Topologien	Bus, Ring, Stern
max. Übertragungskapazität	1 MBit/s bei 40 m Länge
Übertragungseffizienz	0 – 8 Daten-Bytes 4 – 6 Bytes Zusatzinformationen
Verkabelung	zweifach verdrillt (*twisted pair*) oder einfach

Tabelle 2. Zusammenfassung der CAN-Eigenschaften.

Konrad Etschberger: CAN Controller Area Network - Grundlagen, Protokolle, Bausteine, Anwendungen; Fachbuchverlag Leipzig, 3. aktualisierte Auflage, 2002

Werner Zimmermann und Ralf Schmidgall: Bussysteme in der Fahrzeugtechnik – Protokolle und Standards; Vieweg, 2. Auflage, 2007

CAN-Informationen der Robert Bosch GmbH:
www.semiconductors.bosch.de/en/20/can/index.asp

1.3.2 byteflight

Das byteflight-Protokoll wurde Ende der 1990er Jahre von BMW zusammen mit den Firmen Motorola, Infineon Technologies und Elmos Semiconductor (Dortmund) als Kommunikationssystem für Anwendungen mit zeitkritischen Anforderungen entwickelt. Im Gegensatz zu CAN, das von allen Herstellern verwendet wird, ist byteflight heute nur bei BMW-Modellen im Einsatz. Wir gehen dennoch kurz auf byteflight ein, da es eine konzeptionell interessante Weiterentwicklung des CAN-Protokolls darstellt und byteflight außerdem ein Teil des FlexRay-Standards wurde.

Protokoll. Das byteflight-Protokoll stellt insoferne eine Weiterentwicklung des CAN-Protokolls dar, als die Vermeidung von Kollisionen beim Nachrichten-übertragen erreicht wird und so ein Arbitrierverfahren auf Basis der Frame-Prioritäten überflüssig wird. Die grundlegende Idee in byteflight ist, dass die Uhren der ECU-Knoten synchronisiert werden und dann jeder Knoten seine Frames nur in einem vordefinierten Zeitfenster senden darf. Das Protokoll wurde als FTDMA (*Flexible Time Division Multiple Access*) bezeichnet.

Die Uhrensynchronisation erfordert, dass ein Knoten als ein sogenannter *SYNC-Master* ausgezeichnet wird. Der SYNC-Master synchronisiert periodisch durch einen Puls die anderen Knoten, und zwar alle 250 Mikrosekunden (also vier Mal pro Tausendstelsekunde). Zwischen zwei Synchronisations-Pulsen werden Frames von den Knoten übertragen. Analog wie bei CAN wird ein Frame vom Knoten, der ihn sendet, mit einem Identifikator versehen, wobei die von den Knoten gewählten Frame-Identifikatoren für eine Periode eindeutig sein müssen.

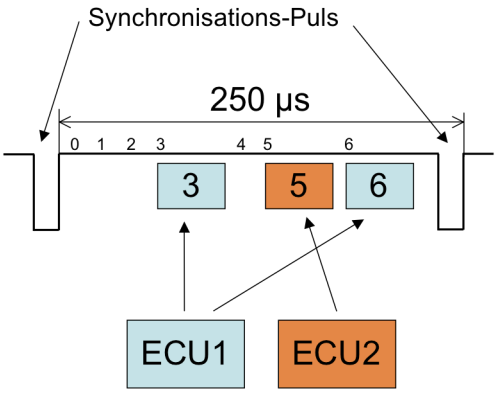

Abbildung 13. FTDMA in byteflight.

Mit dem Synchronisations-Puls, also mit dem Beginn einer 250 Mikrosekunden-Periode, zählt jeder Knoten einen *Slot*-Zähler von Null weg bis zur höchsten erlaubten Identifikator-Nummer hoch. Sobald der *Slot*-Zähler einen Wert hat, der gleich dem Identifikator eines Frames ist, der gesendet werden soll, wird der Frame übertragen und alle Knoten unterlassen das weitere Erhöhen ihrer *Slot*-Zähler bis die Übertragung abgeschlossen ist. Abbildung 13 illustriert das FTDMA-Protokoll anhand eines Beispiels. Wir nehmen an, dass zwei ECU-Knoten ECU1 und ECU2 über ein byteflight-Kommunikationssystem Nachrichten austauschen wollen. ECU1 sendet in der dargestellten 250 µs-Periode zwei Frames mit den Prioritäten 3 und 6. ECU2 sendet einen Frame mit der Priorität 5. Zu Beginn der 250 µs-Periode startet jeder ECU-Knoten damit, einen *Slot*-Zähler zu erhöhen, und zwar von 0 bis 6. Da keine ECU einen Frame mit den Prioritäten 0 bis 2 senden will, haben diese *Slots* eine geringe Zeitdauer. Erst wenn der *Slot-Zähler* bei 3 steht sendet ECU1 Frame 3. Die andere ECU2 zählt ihren *Slot*-Zähler auch nicht weiter hinauf bis die Übertragung von Frame 3 abgeschlossen ist. Analog dazu werden Frame 5 von ECU2 und Frame 6 wiederum von ECU1 übertragen.

Das byteflight-Protokoll ist ebenso wie CAN für ereignisgesteuerte Software ausgelegt. Es nutzt die Zeit lediglich zur Vermeidung von Kollisionen auf dem Kommunikationssystem. Ansonsten können die Frames, die pro Periode übertragen werden, jeweils auf das Neue von den Anwendungen, die auf den ECU-Knoten ausgeführt werden, festgelegt werden. Bei zeitgesteuerten Protokollen hingegen wird ein von vorne herein festgelegter Sendeplan (*communication schedule*) unverändert in jeder Periode ausgeführt. Das byteflight-Protokoll ist ebenso wie CAN nicht deterministisch, da es von den Prioritäten der Frames pro Periode abhängt, welche Frames übertragen werden: Pro Periode kann eine maximale Anzahl von Frames übertragen werden. Wenn Knoten Frames mit hoher Priorität übertragen müssen, bleibt keine Zeit für die Übertragung von Frames mit niedriger Priorität. Bei hoher Auslastung durch hoch priorisierte Frames besteht damit die Gefahr, dass Frames mit niedriger Priorität zu spät übertragen werden. Das einzige, das durch byteflight sichergestellt wird, ist, dass eine bestimmte Zahl von Frames mit hoher Priorität vorrangig behandelt und damit in der nächsten Periode übertragen werden. Es gibt also nicht wie bei CAN die Möglichkeit, dass ein Frame mit niedriger Priorität die Übertragung eines Frames mit hoher Priorität verzögert. Das ändert aber wie oben ausgeführt nichts an der Tatsache, dass das

byteflight-Protokoll nicht deterministisch ist und es ebenso keine Kompositionalität unterstützt.

Das byteflight-Protokoll ist wie CAN für ereignisgesteuerte Software ausgelegt. Es werden im Unterschied zu CAN aber Kollisionen vermieden und es wird sichergestellt, dass eine bestimmte Anzahl von Nachrichten mit höchster Priorität vorrangig übertragen werden.

Zur Fehlererkennung bei der Übertragung wird wie bei CAN die sogenannte zyklische Redundanzprüfung (*Cyclical Redundancy Check*, CRC) verwendet. Nur wenn der CRC-Wert eines Frames beim Empfänger validiert wurde und korrekt ist, werden Nachrichten dieses Frames an die Anwendungen weitergegeben. Wenn der SYNC-Master-Knoten ausfällt, kann ein anderer Knoten SYNC-Master werden.

Topologie. In einem byteflight-Kommunikationssystem sind die Knoten typischerweise sternförmig angeordnet. Der zentrale Knoten (*star coupler*) empfängt Nachrichten von den einzelnen Knoten und verteilt sie jeweils an alle anderen Knoten. Damit kann auch das sogenannte *Babbling-Idiot*-Problem vermieden werden. Der Begriff *Babbling Idiot* bezeichnet einen ECU-Knoten, der aufgrund eines Hardware- oder Anwendungs-Software-Fehlers Daten ständig oder zu unerwarteten Zeitpunkten sendet. Der zentrale Knoten im Stern ist in der Lage, solche fehlerhaften Knoten zu identifizieren, da jeder Knoten seine Nachrichten zuerst an den zentralen Knoten sendet. Der zentrale Knoten unterbindet dann das Weiterleiten der Nachrichten von *Babbling Idiots*.

Als Alternative zur Stern-Topologie kann bei byteflight-Kommunikationssystemen auch die Bus-Topologie gewählt werden.

Übertragungsraten und Verkabelung. Das byteflight-Kommunikationssystem ist für 10 MBit/Sekunde ausgelegt. Diese Übertragungsrate erfordert optische Polymerfasern als Verkabelung. Pro Frame kann ca. die Hälfte für die Übertragung der Nachrichten (*Payload*) genutzt werden. Die andere Hälfte wird für Protokoll-spezifische Informationen benötigt.

Die Tabelle 3 fasst die wichtigsten Eigenschaften von byteflight zusammen.

Protokoll	FTDMA nicht deterministisch Ereignissteuerung CRC-Fehlererkennung
unterstützte Topologien	Stern, Bus
max. Übertragungskapazität	10 MBit/s
Übertragungseffizienz	ca. 50%
Verkabelung	optische Polymerfasern

Tabelle 3. Zusammenfassung der byteflight-Eigenschaften.

Josef Berwanger, Martin Peller, Robert Griessbach: *byteflight—A New Protocol for Safety Critical Applications*, Seoul 2000 FISITA World Automotive Congress, 12.–15. Juni 2000, Seoul, Korea

Ausführliche Informationen zu byteflight, inklusive der aktuellen Spezifikation finden Sie unter byteflight.com

1.3.3 LIN (Local Interconnect Network)

LIN wurde Ende der 1990er Jahre als kostengünstiges Kommunikationssystem im Automobil spezifiziert. Die maximale Datenübertragungsrate von 20 kBit/Sekunde unterstreicht dieses Ziel. In einem LIN-System können bis zu 16 Knoten verbunden sein. Die Intention ist, LIN dort einzusetzen, wo CAN zu aufwändig ist. LIN-Systeme sind typischerweise ein Bestandteil eines CAN-Buses.

LIN ist als kostengünstiges Kommunikationssystem mit Datenübertragungsraten von maximal 20 kBit/Sekunde konzipiert. CAN-Systeme enthalten oft LIN-Subsysteme.

Protokoll. In LIN-Systemen soll ebenso wie in byteflight-Systemen das von CAN bekannte Problem von Kollisionen vermieden werden. Deswegen wird ein Knoten als Koordinator (*Master Node*) ausgezeichnet. Die anderen Knoten werden *Slave Nodes* genannt. Wie die Bezeichnungen schon ausdrücken, darf normalerweise nur der *Master Node* eine Kommunikation initiieren, nicht jedoch die *Slave Nodes*. Das unterscheidet LIN von CAN, bei dem jeder Knoten unabhängig ist und das Senden eines Frames starten kann.
Ein Frame besteht in LIN aus zwei Teilen, dem *Header*- und dem *Response*-Teil. Der *Header* enthält einen eindeutigen Identifikator und wird vom *Master Node* an alle *Slave Nodes* versendet. Daraufhin sendet jener Knoten, dessen Identifikator dem im *Header*-Teil gesendeten Identifikator entspricht.

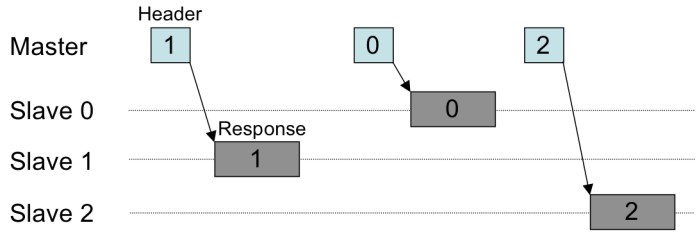

Abbildung 14. Koordination zwischen LIN-*Master* und LIN-*Slaves*.

Abbildung 14 zeigt exemplarisch das Zusammenspiel zwischen dem *Master Node* und drei *Slave Nodes*, die von 0 bis 2 nummeriert sind. Der *Master* sendet zuerst einen *Header* mit dem Identifikator 1, sodass *Slave* 1 die Antwort kommuniziert. Dann wird *Slave* 0 vom *Master* aufgefordert, die Daten zu kommunizieren und schließlich *Slave* 2.

Der *Master Node* sendet die *Header* auf Basis einer Tabelle, die periodisch abgearbeitet wird und die spezifiziert, wann welcher Knoten durch einen *Header* zum Senden seiner Daten aufgefordert werden soll. Wenn LIN so verwendet wird, ist es ein zeitgesteuertes Protokoll. Diese Tabelle wird meist als *Schedule* oder *Communication Schedule* beziehungsweise *Bus Schedule* bezeichnet. In LIN kann der *Master Node* mehrere Schedules speichern und unter diesen auswählen, also gleichsam einen „Fahrplan“ wählen, nach dem kommuniziert wird. Zu einem Zeitpunkt kann aber nur ein *Schedule* verwendet werden.

Auf weitere Details des LIN-Protokolls gehen wir nicht näher ein. Es seien an dieser Stelle aber noch die folgenden zwei wichtigen Aspekte erwähnt: Um die Menge der gesendeten Informationen zu reduzieren, gibt es die Möglichkeit, dass *Slave Nodes* ereignisgesteuert einen Frame senden. Die Menge der gesendeten Informationen wird dann deswegen reduziert, weil der *Master Node* nicht regelmäßig die *Slave Nodes* zum Senden von Frames auffordert, auch wenn bei den *Slave Nodes* keine Änderungen passiert sind. Wenn *Slave Nodes* ereignisgesteuert senden, können auch Kollisionen auftreten, die durch ein spezifisches Protokoll vom *Master Node* aufgelöst werden. Dazu schaltet der *Master Node* den *Schedule* auf einen *Collision Resolving Schedule* um. Weiters unterscheidet das LIN-Kommunikationssystem zwei Zustände: aktiv (*active mode*) und Schlafzustand (*sleep mode*). Wenn Daten übertragen werden, müssen alle Knoten aktiv sein.

Wenn das LIN-Protokoll so verwendet wird, dass ein *Communication Schedule* ausschließlich vom *Master Node* abgearbeitet wird und keine ereignisgesteuerten Frames zum Einsatz kommen, liegt ein zeitgesteuertes Protokoll vor. Gegenüber vollwertigen zeitgesteuerten Kommunikationssystemen wie FlexRay unterscheidet es sich dadurch, dass die Uhren der Knoten nicht synchronisiert werden, eine geringe Übertragungsrate geboten wird, keine Fehlertoleranz (bis auf Prüfsummen) vorgesehen ist, und der *Communication Schedule* während der Laufzeit geändert werden kann. LIN-Systeme sind außerdem typischerweise ein Teil eines ereignisgesteuerten CAN-Systems.

Da die *Slave Nodes* möglichst kostengünstig herstellbar sein sollen, erfordert das LIN-Protokoll keine genauen Uhren (Quartz oder keramische Oszillatoren). Es genügt ein Synchronisationsteil im Header, um innerhalb eines Frames eine hinreichend genaue Synchronisation zur Übertragung zu gewährleisten. Die interne Uhr eines *Slave Nodes* kann damit auch durch sogenannte RC-Oszillatoren (RC steht für *Resistor-Capacitor*, also aus Widerständen und Kondensatoren gebauten Oszillatoren) realisiert sein.

Topologie. In einem LIN-Kommunikationssystem sind die Knoten typischerweise in einer Bus-Topologie angeordnet. Ein LIN-Kommunikationssystem kann maximal 16 *Slave Nodes* enthalten. Der *Master Node* bildet meist das Gateway zu einem CAN-Kommunikationssystem.

Übertragungsraten und Verkabelung. Das LIN-Kommunikationssystem ist für maximal 20 kBit/Sekunde ausgelegt. Pro Frame können maximal 8 Bytes übertragen werden. Wenn 8 Bytes übertragen werden, wird ca. die Hälfte der

Übertragungsrate für die Übertragung der Nachrichten (*Payload*) genutzt. Die andere Hälfte wird für Protokoll-spezifische Informationen benötigt.

Die Tabelle 4 fasst die wichtigsten Eigenschaften von LIN zusammen.

Protokoll	rudimentäre Zeitsteuerung und deterministisch (wenn die Abarbeitung eines *Communication Schedules* durch den *Master Node* erfolgt) Ereignissteuerung und nicht deterministisch (wenn die Möglichkeit genutzt wird, dass *Slave Nodes* Frames bei Eintreten eines Ereignisses senden können) CRC-Fehlererkennung
unterstützte Topologien	Bus; über *Master Node* an CAN-Kommunikationssystem angebunden
max. Übertragungskapazität	20 kBit/s
Übertragungseffizienz	max. ca. 50%
Verkabelung	zweifach verdrillt (*twisted pair*) oder einfach

Tabelle 4. Zusammenfassung der LIN-Eigenschaften.

Ausführliche Informationen zu LIN, inklusive der aktuellen Spezifikation finden Sie unter
lin-subbus.org

Werner Zimmermann und Ralf Schmidgall: Bussysteme in der Fahrzeugtechnik – Protokolle und Standards; Vieweg, 2. Auflage, 2007

1.3.4 FlexRay

FlexRay ist ein Vertreter eines zeitgesteuerten Kommunikationsprotokolls. Zeitgesteuerte Kommunikationsprotokolle eignen sich insbesondere für sicherheitskritische Systeme, die streng vorgegebene Echtzeitanforderungen einhalten müssen. Die den zeitgesteuerten Kommunikationsprotokollen zugrundeliegenden Konzepte wurden in den letzten ca. 20 Jahren insbesondere von Prof. Dr. Hermann Kopetz an der Technischen Universität Wien zusammen mit Partnern der Automobil- und Flugzeugindustrie entwickelt. Daraus entstand die Time-Triggered Architecture (TTA) mit dem Time-Triggered Protocol (TTP), die als Produkte von der Firma TTTech [TTTech, 2007] vertrieben werden. Die Autoindustrie hat Ende der 1990er Jahre das FlexRay-Konsortium gegründet und den gleichnamigen Standard definiert. FlexRay wird daher in Zukunft die Palette der

Kommunikationssysteme im Automobil ergänzen. Wenn Sie an einem Vergleich von TTP/TTA und FlexRay interessiert sind, verweisen wir auf die Web-Site der Firma TTTech [TTTech, 2007].

Im nachfolgenden Teil beschreiben wir FlexRay ähnlich detailliert wie die anderen zuvor präsentierten Kommunikationssysteme. Damit können Sie die wesentlichen Eigenschaften der Kommunikationssysteme überblicksmäßig vergleichen.

Da FlexRay künftig im Automobil eine wichtige Rolle spielen wird, beschreiben die Kapitel 3 und 4, wie Software für FlexRay-Systeme entwickelt wird.

Protokoll. Das FlexRay-Protokoll sieht vor, dass die Kommunikation zwischen den Knoten periodisch wiederholt wird. Das bezeichnen wir als Kommunikationszyklus (*communication cycle*). Ein Kommunikationszyklus besteht aus einem sogenannten statischen Segment und optional einem dynamischen Segment. Abbildung 15a zeigt einen Kommunikationszyklus, der 5 Millisekunden dauert und nur aus einem statischen Segment besteht. Abbildung 15b zeigt einen Kommunikationszyklus, der 5 Millisekunden dauert und aus einem statischen Segment der Dauer von 3 Millisekunden und einem dynamischen Segment der Dauer von 2 Millisekunden besteht.

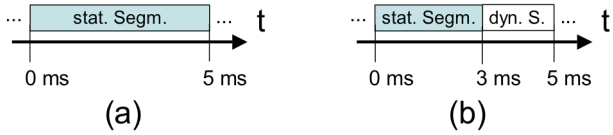

Abbildung 15. Kommunikationszyklus mit statischem Segment (a) sowie mit statischem und dynamischem Segment (b).

Das statische Segment ist der zeitgesteuerte Teil von FlexRay. Nach einem sogenannten *Time Division Multiple Access* (TDMA) Schema wird in einem *Schedule* festgelegt, wann welcher Knoten innerhalb eines Kommunikationszyklus sendet. Dieser Schedule ist statisch festgelegt und wird zur Laufzeit periodisch immer wieder abgearbeitet. Abbildung 16 zeigt exemplarisch einen Schedule eines ausschließlich aus einem statischen Segment bestehenden Kommunikationszyklus, in dem eine Nachricht von ECU1 nach 1 Millisekunde und eine Nachricht von ECU2 nach 3 Millisekunden gesendet wird.

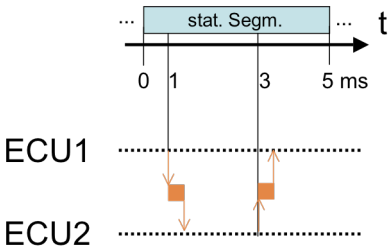

Abbildung 16. Zeitgesteuertes Senden von Nachrichten.

Das optionale dynamische Segment ist der ereignisgesteuerte Teil von FlexRay. Darin wird mittels byteflight-Protokoll kommuniziert.

Das byteflight Protokoll ist in Abschnitt 1.3.2 beschrieben.

Da FlexRay mit 10 MBit/Sekunde eine um den Faktor 10 höhere Übertragungsrate als CAN bietet, wird FlexRay oft auch als das verbesserte CAN gesehen. FlexRay löst quasi Engpässe bei der Übertragungsrate. Wenn FlexRay in diesem Sinne genutzt wird, benötigt man ausschließlich das dynamische Segment, über das dann ereignisgesteuert Werte kommuniziert werden. Der Vorteil von byteflight gegenüber CAN ist, dass Frames mit hoher Priorität garantiert vorrangig übertragen werden. Es ist durchaus realistisch, dass Serieneinsätze von FlexRay in den kommenden Jahren durch eine derartige Migration bestehender ereignisgesteuerter Anwendungen geprägt sind. Wenn tatsächlich X-by-Wire-Anwendungen oder andere sicherheitskritische Anwendungen, die ein stabiles Reglerverhalten voraussetzen, zum Einsatz kommen werden, wird verstärkt das statische Segment von FlexRay, also die Zeitsteuerung, verwendet werden.

FlexRay bietet außer der gegenüber CAN um den Faktor 10 höheren Übertragungsrate auch den Vorteil, dass mehr Nutzdaten übertragen werden können: Ein Frame kann in FlexRay bis zu 246 Bytes Nutzdaten (*Payload*) enthalten und benötigt insgesamt 8 Bytes für den Vorspann (*Header*) und den Nachspann (*Trailer*).

FlexRay wird seinem Namen gerecht, indem es sowohl Zeitsteuerung als auch Ereignissteuerung unterstützt. Die Übertragungsrate ist um den Faktor 10 höher als bei CAN.

Topologie. FlexRay unterstützt sowohl die Bus- als auch die Stern-Topologie sowie eine Kombination der beiden Topologien. FlexRay bietet als Besonderheit die Möglichkeit, Knoten mit zwei Kanälen, die mit A und B bezeichnet werden, zu verbinden. Die Idee dabei ist, dass bei wichtigen Verbindungen dadurch eine Redundanz und somit eine Ausfallssicherheit möglich ist. Die Kanäle können aber auch unabhängig voneinander zur Kommunikation verwendet werden. Abbildung 17 zeigt einen Bus, mit dem drei Knoten mit beiden Kanälen und zwei Knoten jeweils nur mit einem Kanal verbunden sind.

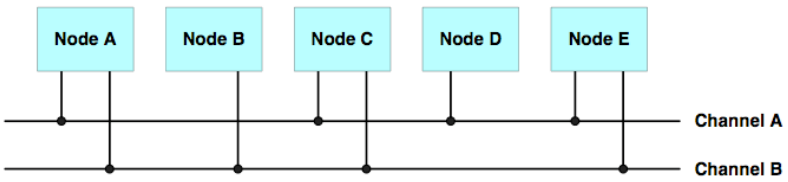

Abbildung 17. Beispiel für einen FlexRay-Bus mit fünf Knoten.

Anstatt die Knoten wie oben über einen Bus mit zwei Kanälen zu verbinden, können die Knoten auch in einer Stern-Topologie angeordnet werden (siehe Abbildung 18). Der zentrale Knoten des linken Sterns entspricht exakt dem Kanal A bei der Bus-Topologie: es sind die Knoten A, C, D und E mit dem Koordinator-Knoten verbunden.

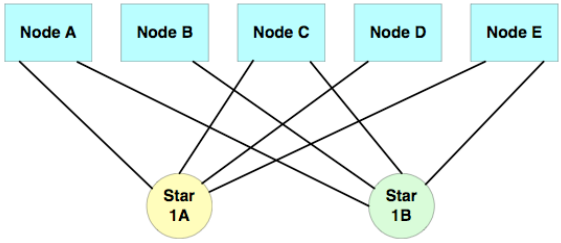

Abbildung 18. Beispiel für Stern-Topologien.

Der Unterschied zur Bus-Topologie ist, dass der Koordinator-Knoten einen Frame, den er von einem der angeschlossenen Knoten empfängt, an die anderen Knoten im Stern verteilt. Deswegen wird die Stern-Topologie auch als aktive Stern-Topologie bezeichnet. Bei der Bus-Topologie empfangen alle an den Bus angeschlossenen Knoten einen Frame, der von einem Knoten gesendet wird, ohne dass ein Koordinator-Knoten dafür sorgen muss. Deswegen wird die Bus-Topologie auch oft als passive Bus-Topologie bezeichnet.

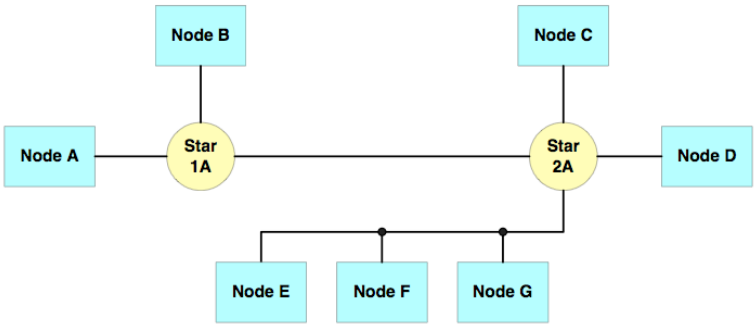

Abbildung 19. Kombination von Stern- und Bus-Topologien.

Wie erwähnt, unterstützt FlexRay die Kombination der Bus- und Stern-Topologien, also hybride Topologien. Abbildung 19 zeigt eine hybride Topologie, die aus zwei Sternen und einem Bus besteht. Dabei ist jeder Knoten nur über genau einen Kanal an den Stern-Koordinator beziehungsweise an den Bus angeschlossen. Die Möglichkeit, Redundanz über die zwei Kanäle zu erhalten wird in diesem Beispiel nicht genutzt.

Abbildung 20 zeigt eine hybride Topologie, bei der jeder Knoten einmal mit einem Bus (Kanal A) und einmal mit einem Stern-Koordinator-Knoten verbunden ist.

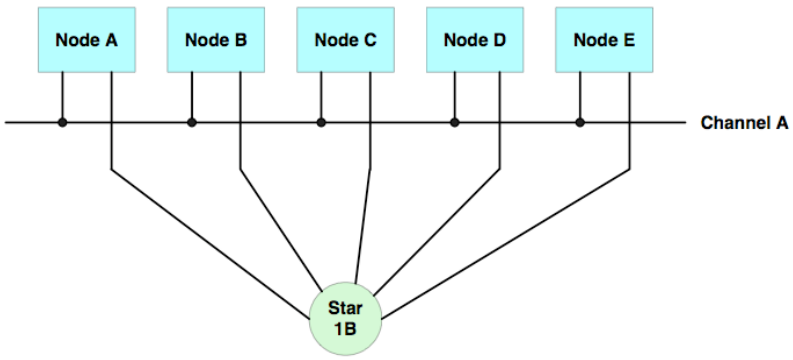

Abbildung 20. Kombination von Stern- und Bus-Topologien unter Ausnutzung beider Kanäle.

Übertragungsraten und Verkabelung. FlexRay ist für 10 MBit/Sekunde ausgelegt. Wenn beide Kanäle unabhängig genutzt werden, also die Daten nicht redundant übertragen werden, können bis zu 20 MBit/Sekunde übertragen werden. Die Verkabelung kann über verdrillte Kupferkabelpaare (*twisted-pair*) oder über optische Polymerfasern erfolgen. Meist werden die verdrillten Kupferkabelpaare bei einer Bus-Topologie und die Glasfaserkabel bei einer Stern-Topologie verwendet.

Die Tabelle 5 fasst die wichtigsten Eigenschaften von FlexRay zusammen.

Protokoll	Zeitsteuerung und deterministisch im statischen Segment Ereignissteuerung und nicht deterministisch im dynamischen Segment CRC-Fehlererkennung
unterstützte Topologien	Bus, Stern und Kombinationen davon
max. Übertragungskapazität	10 MBit/s pro Kanal
Übertragungseffizienz	max. ca. 80% (unter der Berücksichtigung der Abstände zwischen Frames)
Verkabelung	zweifach verdrillt (*twisted pair*) oder optische Polymerfasern

Tabelle 5. Zusammenfassung der FlexRay-Eigenschaften.

Ausführliche Informationen zu FlexRay, inklusive der aktuellen Spezifikation finden Sie unter
FlexRay.com

Werner Zimmermann und Ralf Schmidgall: Bussysteme in der Fahrzeugtechnik – Protokolle und Standards; Vieweg, 2. Auflage, 2007

1.3.5 MOST (Media-Oriented Systems Transport)

Das MOST-Kommunikationssystem bildet die Basis für Multimedia-Anwendungen im Automobil. Damit können Audio- und Video-Datenströme übertragen und gesteuert werden. Der Hauptkritikpunkt an MOST ist, dass bereits heute die Übertragungskapazität von 25 MBit/Sekunde (MOST25) beziehungsweise von 50MBit/Sekunde (MOST50) an ihre Grenzen stößt, da diese Übertragungsraten zum Beispiel für hochauflösende, nicht komprimierte Videos nicht ausreichen. Als Alternative zu MOST wird der von Apple Inc. (Cupertino, California) initiierte FireWire-Standard IEEE 1394 betrachtet, mit dem bis zu 400 MBit/Sekunde (1394a) beziehungsweise bis zu 800 MBit/Sekunde übertragen werden können. Sowohl MOST als auch FireWire unterstützen das Hinzufügen und Entfernen von Knoten im laufenden Betrieb (*hot plugging*).

MOST dient zur Übertragung von Audio- und Video-Datenströmen. Die maximale Übertragungskapazität von 50 MBit/Sekunde (MOST50) stößt bereits jetzt an ihre Grenzen, da das für hochauflösende, nicht komprimierte Videos nicht ausreicht.

Protokoll und Übertragungsraten. MOST bietet drei Arten von Kanälen zur Übertragung von Daten: den *Control Channel*, den *Streaming Data Channel* und den *Packet Data Channel*. Wie der Name schon suggeriert, dient der *Streaming Data Channel* zum Übertragen von kontinuierlichen Datenströmen, also Audio- und Video-Daten. Die Verbindungen, also welche Knoten an welche andere Knoten diese Datenströme übertragen, werden dynamisch mittels des *Control Channels* verwaltet. Der Control Channel dient auch zum Übertragen von Befehlen an die am Kommunikationssystem angeschlossenen Geräte, wie zum Beispiel das Einschalten des Radios. Über den Control Channel werden Frames ereignisgesteuert übertragen. Der *Packet Data Channel* ist zur Übertragung sporadisch anfallender größerer Datenmengen konzipiert, wie zum Beispiel von Bildern, Grafiken, oder Kartenausschnitten für das Navigationssystem.

Ein MOST-Kommunikationssystem besteht aus maximal 64 Knoten. Einer davon muss der *Timing Master* sein, der die anderen Knoten, die sogenannten *Slaves*, synchronisiert. Da die *Slaves* zum *Master* synchronisiert sind, ist keine Bufferung der übertragenen Daten nötig.

Ein MOST-Knoten, auch MOST-*Device* genannt, kann aus mehreren Funktionsblöcken bestehen. Das können zum Beispiel ein CD-Spieler oder ein Radio-Empfänger (*Tuner*) sein. Abbildung 21 zeigt ein Gerät, das aus drei Funktionsblöcken besteht. Der *NetBlock* ist als Funktionsblock ein Bestandteil eines jeden MOST-Gerätes und bietet Funktionen, die dem gesamten Gerät zugeordnet sind, wie zum Beispiel das Ein- und Ausschalten. Die anderen beiden Funktionsblöcke in Abbildung 21 könnten ein CD-Spieler und ein Radio-Empfänger sein. Ein Radio-Empfänger bietet unter anderem zum Beispiel Funktionen zum Festlegen der Empfangsfrequenz, des Bandes (FM/AM) und der Tonausgabe (Mono/Stereo) an. An Funktionsblöcke können Befehle gesendet werden, die die jeweiligen Funktionen aktivieren. Angenommen, der Radio-

Empfänger hat die Identifikation TUNER.1, dann würde zum Beispiel der Befehl TUNER.1.STEREO.SET(true) eine Stereo-Ausgabe bewirken. Solche Befehle werden über den *Control Channel* übertragen.

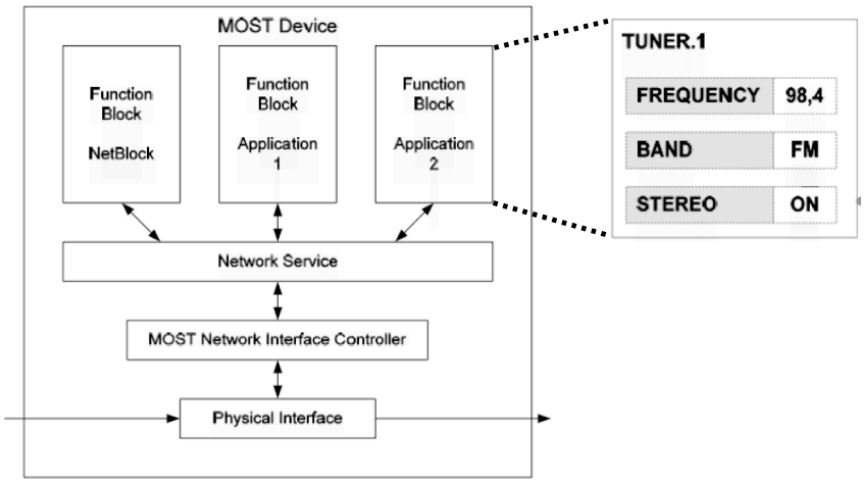

Abbildung 21. Struktur eines MOST-Gerätes.

Audio- und Video-Datenströme werden über den *Streaming Data Channel* übertragen. Dabei wird das sogenannte *Time Division Multiplexing* (TDM) Verfahren angewendet, bei dem in definierten Zeitabschnitten die Daten verschiedener Sender auf einem physischen Kanal übertragen werden. Die Übertragungskapazität der logischen Kanäle, die auf einen physischen Kanal abgebildet werden, kann je nach Anwendung und Datenvolumen definiert werden. In MOST25 stehen auf dem *Streaming Data Channel* maximal ca. 15 Audio-Kanäle in CD-Qualität zur Verfügung, in MOST50 doppelt soviele.

Der Zugriff auf den *Packet Data Channel* erfolgt auf Basis einer Token-Weitergabe: Ein Token wird von Knoten zu Knoten weitergeben, sodass jeder Knoten gleich oft den Token erhält. Das bezeichnet man auch als Token-Ring-Verfahren, da der Token quasi im Kreis weitergereicht wird. Ein Knoten, der den Token hat, darf Daten über den *Packet Data Channel* übertragen. Jeder Knoten hat dadurch einen fairen Zugriff auf diesen Kanal. Die maximale Paketgröße ist je nach gewählter Konfiguration des MOST-Kommunikationssystems entweder 48 Bytes oder 1014 Bytes. Bei MOST50 sind Pakete immer 1014 Bytes groß. Ein Paket benötigt typischerweise mehrere Frames, um übertragen zu werden.

MOST25 bietet insgesamt eine Übertragungsrate von 25 MBit/Sekunde, MOST50 von 50 MBit/Sekunde. In einem MOST-Kommunikationssystem kann die verfügbare Übertragungskapazität zwischen dem *Streaming Data Channel* und dem *Packet Data Channel* flexibel aufgeteilt werden. Das erfolgt durch das Setzen des sogenannten *Boundary Descriptors*. Abbildung 22 zeigt schematisch anhand eines MOST50-Frames, dass damit die verfügbaren Bytes innerhalb eines Frames für *Packet*- und für *Stream*-Daten festgelegt werden. Von den 128 Bytes eines MOST50-Frames werden die ersten 7 Bytes als Vorspann für administrative Zwecke (Synchronisation, Fehlerprüfung, etc.) benötigt. Die nächsten vier Bytes stehen zum Übertragen von Befehlen zur Verfügung. Die verbleibenden 117 (= 128

- 7 - 4) Bytes stehen für die Übertragung von Audio-, Video- beziehungsweise Paketdaten entsprechend der gewählten Aufteilung zur Verfügung.

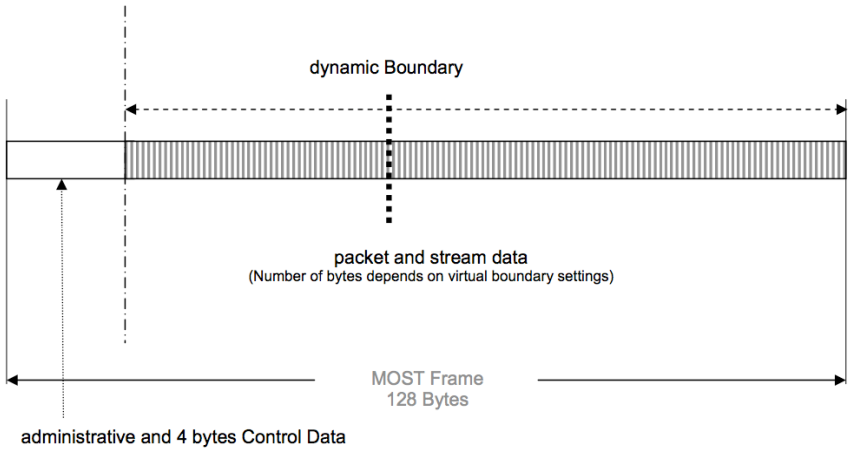

Abbildung 22. Aufbau eines MOST50-Frames.

Abbildung 22 zeigt auch, wie die drei logischen Kanäle eines MOST-Kommunikationssystems, der *Control Channel*, der *Streaming Data Channel* und der *Packet Data Channel*, realisiert werden. Innerhalb eines Frames sind eine bestimmte Anzahl von Bytes für diese drei logischen Kanäle verfügbar. Bei MOST50 sind das 4 Bytes für die Befehle (Control Data) und 117 Bytes für den Datenstrom und die Pakete. Der *Boundary Descriptor* legt die Aufteilung der 117 Bytes in diese zwei Bereiche fest.

Topologie und Verkabelung. Typischerweise sind die maximal 64 Knoten eines MOST-Kommunikationssystems in einer Ring-Topologie angeordnet und mit einem optischen Polymerfaserkabel verbunden.

Wie schon eingangs erwähnt, stoßen die Übertragungskapazitäten bei MOST-Kommunikationssystemen an die Grenzen, insbesondere wenn Video-Daten in hoher Qualität übertragen werden sollen. Ein weiterer Aspekt der Infotainment-Ausstattung von Automobilen sollte außerdem erwähnt werden, auch wenn das über das Kommunikationssystem hinausgeht. Die letzten Jahre haben gezeigt, dass sich im Infotainment-/Multimedia-Bereich Geräte etablieren, die auch nahtlos in das Infotainment-System eines Automobils integrierbar sein sollten. Beispiele sind die iPod-Produktpalette von Apple und die diversen Mobiltelefone. Die Erfahrung zeigt, dass eine nahtlose und vor allem rasche und eventuell sogar nachträgliche Integration solcher Produkte ein wichtiger Aspekt aus Sicht des Autokäufers ist. Die erwartete nahtlose Integration ist leider heute kaum gegeben. Aktuell werden zum Beispiel neue Automodelle ausgeliefert, deren iPod-Integration sich darauf beschränkt, den iPod als CD-Spieler zu behandeln. Es kann daher nicht aus den Listen der Musiktitel, die am iPod gespeichert sind, gewählt werden, sondern es müssen Zahlen (CD-Nummer, Track-Nummer) angegeben werden, bei denen sich noch dazu herausstellt, dass der Wertebereich (maximale Anzahl der CDs, maximale Anzahl der Tracks pro CD) selbst für durchschnittlich große iPod-Musiksammlungen bei weitem nicht ausreicht.

Die Tabelle 6 fasst die wichtigsten Eigenschaften von MOST zusammen.

Protokoll	synchroner Datenstrom, TDM (*Stream Data Channel*) Ereignissteuerung, nicht deterministisch, Token-Ring-Verfahren (*Paket Data Channel*) Ereignissteuerung, nicht deterministisch, CRC-Fehlererkennung (*Control Data Channel*)
unterstützte Topologien	typischerweise eine Ring-Topologie
max. Übertragungskapazität	25 MBit/s (MOST25) 50 MBit/s (MOST50)
Übertragungseffizienz	44 100 Frames/Sekunde (MOST25) 60 (Nutz-)Bytes/Frame 48 000 Frames/Sekunde (MOST50) 117 (Nutz-)Bytes/Frame
Verkabelung	optische Polymerfasern

Tabelle 6. Zusammenfassung der MOST-Eigenschaften.

Ausführliche Informationen zu MOST, inklusive der aktuellen Spezifikation finden Sie unter MOSTcooperation.com

Werner Zimmermann und Ralf Schmidgall: Bussysteme in der Fahrzeugtechnik – Protokolle und Standards; Vieweg, 2. Auflage, 2007

1.4 Auswirkung der Kommunikationsprotokolle auf die Software-Eigenschaften

In Abschnitt 1.2 wurden Determinismus und Kompositionalität als wünschenswerte Eigenschaften von Softwaresystemen im Automobil erläutert. Im Automobil ist ein Softwaresystem über mehrere Knoten verteilt. Deshalb haben die Kommunikationssysteme, über die die verteilte Software Nachrichten austauscht, einen entscheidenden Einfluss auf diese beiden Eigenschaften.

Die Definition der Software-Eigenschaften Zeit- und Wert-Determinismus sowie Kompositionalität finden Sie in Abschnitt 1.2.

Zusammenfassend ist festzustellen, dass nur die Zeitsteuerung, wie sie im statischen Segment von FlexRay unterstützt wird, eine deterministische

Kommunikation erlaubt, bei der Nachrichten zu exakt definierten Zeitpunkten ausgetauscht werden, ohne dass Kollisionen auftreten können oder hohe Prioritäten von Nachrichten das Senden von Nachrichten mit niedriger Priorität verzögern können. Damit können prinzipiell Software-Komponenten realisiert werden, die den gewünschten Zeit- und Wert-Determinismus als Eigenschaft haben. Ohne auf die Theorie dahinter einzugehen sei hier noch angemerkt, dass ein deterministisches Kommunikationssystem zwar die Voraussetzung für Zeit- und Wert-Determinismus ist, aber nicht zwingend dazu führt. Es können die Software-Komponenten so implementiert sein, dass trotz der deterministischen Kommunikation ein Softwaresystem entsteht, das nicht deterministisch ist. Um das zu vermeiden, wurde an der Universität von Kalifornien in Berkeley eine Programmierabstraktion, die sogenannte logische Ausführungszeit (*Logical Execution Time*, LET) erfunden, die garantiert, dass beide Eigenschaften, Determinismus und Kompositionalität, erfüllt sind [Henzinger et al., 2003].

Wie die LET-Abstraktion in der Praxis das Entwickeln von Software für FlexRay drastisch vereinfacht und die Qualität durch Garantie dieser beiden Eigenschaften (Determinismus und Kompositionalität) signifikant verbessert, wird in Kapitel 4 beschrieben.

Die Kommunikationssysteme CAN, byteflight und LIN (als Subsystem von CAN) sind durch das ereignisgesteuerte Paradigma geprägt. Da die Idee dabei ist, nur dann zu kommunizieren, wenn ein nicht vorhersehbares Ereignis das erfordert, können das Verhalten und die Korrektheit von Softwaresystemen, die darauf aufbauen, nur statistisch bewertet werden. Für diverse Funktionen, die nicht sicherheitskritisch sind, wie zum Beispiel das Betätigen der Fensterheber, die Betätigung der Zentralverriegelung eines Fahrzeuges oder das Verstellen der Rückspiegel, ist das zweifelsfrei angemessen und unproblematisch. Selbst sicherheitskritische Funktionen, die selten auftreten, wie das Auslösen von Airbags, können auf Basis der genannten Kommunikationssysteme implementiert werden, wenn wie bei byteflight garantiert wird, dass Nachrichten mit hoher Priorität Vorrang bekommen. Allerdings muss auch hier wiederum statistisch abgeschätzt werden, wie wahrscheinlich das Auftreten sovieler Nachrichten mit höchster Priorität ist, dass diese nicht im nächstmöglichen Frame übertragen werden können und welche Funktionen daher in solchen ungünstigen Situationen nicht mehr erfüllt werden könnten.

Das MOST-Kommunikationssystem wird typischerweise im Infotainment-Bereich eingesetzt. Obwohl die Wiedergabe von Audio- und Video-Daten zeitkritisch ist, handelt es sich hier um sogenannte „weiche“ Echtzeitanforderungen: Wenn die Zeitvorgaben nicht eingehalten werden, ist lediglich die Qualität der Wiedergabe beeinträchtigt.

Prognosen sind immer unsicher. Dennoch wagen wir hier einen Ausblick, wie sich die Kommunikationssysteme im Automobil in Zukunft konsolidieren könnten: Da FlexRay im wesentlichen Zeit- und Ereignissteuerung abdeckt, liegt die Vermutung nahe, dass Software, die auf CAN- und byteflight-Kommunikationssystemen basiert, schrittweise nach FlexRay migriert wird und so mittelfristig CAN und byteflight aus dem Automobil verschwinden werden. So würden von den heute 5

Kommunikationssystemen drei, nämlich FlexRay in Kombination mit LIN und ein weiterentwickeltes MOST oder FireWire, übrig bleiben. Mittel- bis langfristig ist es denkbar, dass Ethernet-basierte Kommunikationsysteme im Automobil verwendet werden, die aufgrund der hohen PC-Stückzahlen wesentlich kostengünstiger sind als Systeme, die nur in Automobilen und der Industrieautomatisierung eingesetzt werden. Zusätzlich zur Kommunikation im Automobil werden Szenarien angedacht, die es notwendig machen, dass Automobile mit der Aussenwelt drahtlos kommunizieren, also zum Beispiel mit anderen Automobilen, die Auskunft über die Strassen- und Verkehrssituation geben, oder wenn das Auto in der Garage steht und Musiktitel vom PC zu Hause in das Automobil übertragen werden.

Unabhängig davon, ob und welche dieser Szenarien Realität werden, ist man sich heute bei Herstellern und Zulieferern einig, dass FlexRay in den nächsten Jahren an Bedeutung gewinnen wird. Außerdem werden künftige sicherheitskritische Funktionen vermehrt Zeitsteuerung einsetzen. Deswegen gehen wir in den nächsten Kapiteln auf die Entwicklung von Software ein, die den zeitgesteuerten Aspekt von FlexRay nutzt. Wir zeigen zuerst den heute praktizierten Ansatz auf, als ersten Schritt die Plattform (ECU-Knoten, Topologie, Betriebssystem der ECU-Knoten, Kommunikationssystem) festzulegen, und dann die Software auf diese Plattform zuzuschneiden. Das hat den gravierenden Nachteil, dass Änderungen der Plattform auch Änderungen, oft sogar Neuentwicklungen der Software nach sich ziehen. Nicht zuletzt durch AUTOSAR ist die Vision salonfähig geworden, das Software-Verhalten zuerst zu definieren, also unabhängig von der spezifischen Plattform.

Wie das plattform-neutrale Modellieren von Software-Komponenten unter Einbeziehung von Echtzeitvorgaben und Zeitsteuerung machbar ist, finden Sie in Kapitel 4.

In Kapitel 1 sind wesentliche Begriffe und Konzepte definiert beziehungsweise erläutert worden: was ist ein verteiltes System, welche Topologien sind bei verteilten Systemen im Automobil typischerweise vorzufinden, was ist der Unterschied zwischen Ereignis- und Zeitsteuerung, welche Probleme verursacht Ereignissteuerung (*Priority Inversions, Deadlocks*). Dann sind überblicksmäßig die im Automobil verwendeten verteilten Systeme präsentiert und verglichen worden: CAN, das heute fast in jedem modernen Fahrzeug verwendet wird; byteflight, das den ereignisgesteuerten Teil von FlexRay darstellt; LIN als kostengünstiges Subsystem von CAN, FlexRay als künftiges Kommunikationssystem im Automobil, das zusätzlich zur Ereignissteuerung auch Zeitsteuerung ermöglicht; und schließlich MOST für Multimedia-Anwendungen.

2 Konventionelle versus modellbasierte Software-Entwicklung

Bevor wir konkret zwei Methoden und Werkzeuge zur Entwicklung verteilter FlexRay-Systeme betrachten, wollen wir definieren, was wir unter „konventioneller“ Software-Entwicklung (Kapitel 3) im Gegensatz zu einer modell-basierten Software-Entwicklung (Kapitel 4) im Zusammenhang mit FlexRay verstehen. Viele würden argumentieren, dass eine Verwendung von Matlab®/Simulink® bereits ausreicht, um die Entwicklung als „modellbasiert“ zu bezeichnen. Wir legen hier eine höhere Messlatte an, die im wesentlichen dadurch bestimmt ist, dass von der Plattform (ECU-Knoten, Topologie, Betriebssystem der ECU-Knoten, Kommunikationssystem) abstrahiert werden muss und damit das Verhalten von Software unabhängig von der Plattform, auf der die Software später ausgeführt wird, definiert werden kann.

Allgemein ist zu beobachten, dass das Schlagwort “modellbasiert” zur Zeit gerne als Attribut für Methoden und Werkzeuge gebraucht wird, um damit zu suggerieren, dass die Software-Entwicklung einfacher und damit kostengünstiger wird. Leider ist das oft nicht der Fall. Das liegt daran, dass Entwicklungskosten nur dann eingespart werden können, wenn geeignete Konzepte zur Abstraktion von den Details eines Systems oder eines Problems existieren. Eine grafische Darstellung ohne Abstraktion von Details reicht daher nicht, um die Software-Entwicklung zu vereinfachen.

Wie im vorigen Kapitel motiviert wurde, wollen wir uns auf zeitgesteuerte Echtzeitsoftware konzentrieren, also Software, die FlexRay als Kommunikations-system benutzt. Das können zum Beispiel eine aktive Hinterachslenkung oder künftige X-by-Wire-Systeme sein.

Bei Echtzeitsoftware spielt, wie der Name schon ausdrückt, das Echtzeitverhalten eine wesentliche Rolle, dass das System korrekt funktioniert. Es geht daher darum, wie das Zeitverhalten plattform-neutral beschrieben werden kann. Dazu wurden an der Universität von Kalifornien in Berkeley die *Logical Execution Time* (LET) im Giotto-Projekt [Henzinger et al., 2003] und an der Universität Salzburg die Zeitbeschreibungssprache TDL (*Timing Definition Language* [Templ, 2007]) erfunden.

Kapitel 3 geht auf die modell-basierte Entwicklung mit der *Timing Definition Language* (TDL) und den TDL-Werkzeugen näher ein.

Aus einer TDL-Beschreibung von Software-Komponenten generieren ein Compiler und ein *Communication-Schedule*-Generator alle nötigen Codes, um die Komponenten auf einem spezifischen FlexRay-System ausführen zu können. Dabei wird garantiert, dass das Verhalten sowohl in einer Simulation (zB in Matlab®/Simulink®) als auch auf der konkreten Plattform exakt der Beschreibung entspricht. Wenn die Plattform nicht genügend Rechenleistung bzw. Bandbreite bietet, wird kein Code generiert. Abbildung 23 skizziert schematisch die Vorteile der Entwicklung mit TDL, die das Attribut modellbasiert verdient. Eine Software-

Komponente (zB K1 genannt) wird nur einmal entwickelt. Die automatische Code-Generierung erlaubt die Ausführung auf einer beliebigen Plattform, die genügend Ressourcen bietet. Im Falle von FlexRay könnte K1 beispielsweise einmal auf einem FlexRay-System mit PowerPC-Knoten (ECUs) und dem Betriebssystem AES von DeComSys [DeComSys, 2007] ausgeführt werden und ein anderes mal wird der Code für ein FlexRay- System mit MicroAutoBox-Knoten von dSpace [dSpace, 2007] generiert.

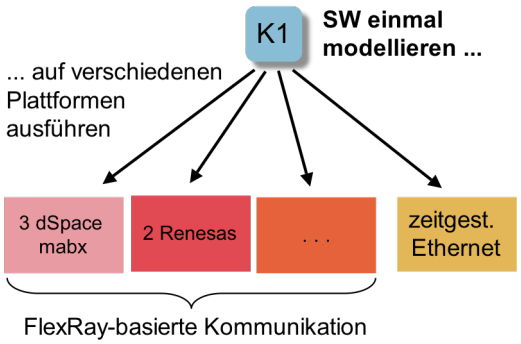

Abbildung 23. Modell-basierte Entwicklung und Code-Generierung.

modellbasierte Entwicklung: Das relevante Verhalten von Software wird plattform-neutral (für uns synonym zu plattform-unabhängig) spezifiziert. Code-Generatoren erzeugen den Code für die spezifische Plattform. Es genügt daher, die Plattform und damit die Abbildung der Komponenten auf diese Plattform (*Platform Mapping*) erst am Ende festzulegen.

Der modellbasierte Entwicklungsprozess in diesem Sinne unterscheidet sich daher fundamental von heute üblichen, „konventionellen" Ansätzen, bei denen die Plattform und die Topologie zuerst festgelegt werden müssen, bevor dann die Software auf diese Plattformgegebenheiten zugeschneidert wird. Hier nützt es auch nichts, wenn zum Beispiel das Zuschneiden der Software auf die Plattform über Matlab®/Simulink® spezifiziert wird. Eine Änderung der Plattform, aber auch schon nur die Änderung der Topologie (zB drei Knoten statt zwei Knoten) erfordert typischerweise eine Neuentwicklung. Die Komponente K1 hängt daher von der Plattform ab: K1-a wird für die Variante a entwickelt, K1-b für die Variante b, etc. (siehe Abbildung 24).

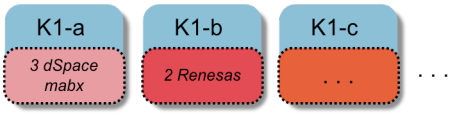

Abbildung 24. Konventionelle Entwicklung für spezifische Plattformen.

Konventionelle Entwicklung: die Plattform wird zuerst fixiert. Dann wird die Software für diese spezifische Plattform entwickelt. Änderungen der Plattform erfordern meist substantielle Änderungen der Software oder meist sogar deren Neuentwicklung.

Einsparungspotenzial mit modellbasierter Entwicklung. Wir haben gemessen, wieviel Zeit benötigt wird, um ein funktionierendes FlexRay-System mit einem aktuellen am Markt erhältlichen Werkzeug zu entwickeln, mit dem der *Communication Schedule* und die diversen FlexRay-Parameter weitgehend manuell erstellt und eingegeben werden müssen. Die Aufgabe wurde durch einen mit dem ausgewählten Werkzeug vertrauten Entwickler durchgeführt. Dieselbe Aufgabe wurde mit den TDL-Werkzeugen [7] gelöst, wiederum durch einen Entwickler, der mit TDL vertraut ist. Das Ergebnis ist, dass die Entwicklungszeit mit TDL um den Faktor 20 kürzer ist als mit dem bisher gängigen Ansatz. Dieser quantitative Vergleich wurde anhand einer einfachen Fallstudie durchgeführt, in der das FlexRay-System aus zwei ECUs besteht und im wesentlichen ein Wert von ECU1 auf ECU2 und dann wieder zurück auf ECU1 übertragen wird. In diesem Fall ist der *Communication Schedule* einfach. Wir vermuten daher, dass mit TDL komplexere Aufgaben noch rascher zu bewältigen sind als diese einfache Fallstudie.

Allgemeine Informationen über modellbasierte Entwicklung finden Sie auf der *Model-Driven Architecture* (MDA)-Site der Object Management Group:
omg.org/MDA/

Dieses im Vergleich zu den anderen Kapiteln kurze Kapitel hat erläutert, wie wichtig eine adäquate Abstraktion von Details der Plattform ist. Die Erfolgsgeschichte der Informatik ist im wesentlichen durch gute Abstraktionen geprägt. Nur so wurden höhere Programmiersprachen möglich, die dann die Maschinen- und Assembler-Programmierung ablösten, also Programmiertechniken, die Software auf eine bestimmte Hardware zugeschneidert hatten. Dieser wichtige Schritt, der die Produktivität signifikant erhöht und die Qualität sowie Portabilität von Software wesentlich verbessert, wird bei der Entwicklung von Echtzeit-Software für Automobile gerade vollzogen.

3 Konventionelle Software-Entwicklung für FlexRay mit dem DESIGNERPRO-Werkzeug

Um die konventionelle FlexRay-Entwicklung zu demonstrieren, haben wir die Werkzeuge der Firma DeComSys ausgewählt, da damit von Grund auf ein FlexRay-System, inklusive des *Communication Schedules*, entwickelt werden kann.

Kapitel 5 gibt einen kurzen Überblick über heute verfügbare Werkzeuge anderer Hersteller.

Aus Entwicklersicht wird, wenn man sich für DeComSys entscheidet, primär das Werkzeug DESIGNERPRO verwendet [DeComSys, 2007]. Wenn Matlab®/

Simulink® einbezogen werden soll, zum Beispiel, um die Software zu simulieren, stehen die SIMTOOLS von DeComSys zur Verfügung. An der Methode selbst, nämlich dass die Plattform zuerst definiert wird und dann die Software darauf zugeschnitten wird, ändert das aber nichts.

Bei der Präsentation der Fallstudie und des Werkzeuges werden diverse Details, die zur Konfiguration eines FlexRay-Systems nötig sind, nicht näher erläutert, damit der Fokus auf die Entwicklungsmethode gerichtet bleibt.

DESIGNERPRO führt im wesentlichen den Entwickler durch die einzelnen Schritte:

(1) Spezifikation der Plattform (ECUs, etc.)
(2) Konfiguration der ECU-Software, also auf welchen Knoten jeweils welche Software-Funktion ausgeführt werden soll
(3) Festlegung der Parameter für das FlexRay-Kommunikationsprotokoll, zum Beispiel die Aufteilung in das statische und dynamische Segment
(4) Definition des *Communication Schedules*
(5) Treiber-Konfiguration (COMMSTACK beziehungsweise FLEXCOM). COMMSTACK ist der Treiber, um Frame-basiert auf den FlexRay-Communication Controller zugreifen zu können. FLEXCOM ist der Treiber, um Signal-basiert auf den FlexRay-Communication Controller zugreifen zu können.

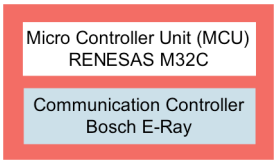

Abbildung 25. Die beiden Controller einer NODE<RENESAS> ECU.

Als Fallbeispiel verwenden wir ein einfaches Beispiel aus dem Handbuch des DESIGNERPRO-Werkzeugs. Das FlexRay-Kommunikationssystem besteht aus zwei ECU-Knoten Node1 und Node2. Ein ECU-Knoten ist ein NODE<RENESAS> wie er schematisch in Abbildung 25 dargestellt ist. Ein solcher Knoten besteht aus einem RENESAS M32C als Micro Controller Unit (MCU), und einem Bosch E-Ray als Communication Controller (CC).

Als Anwendungs-Software sollen auf Node1 zwei Tasks ausgeführt werden, ApplTask1 und ApplTask2. ApplTask1 zählt bei jedem Aufruf einen Zähler, Counter1 genannt, hoch und berechnet außerdem eine Sinus-Funktion. Beide Werte, der Zähler und der berechnete Sinus-Funktionswert werden in einem Frame an Node2 gesendet. Node2 hat nur eine Task, die auch ApplTask1 genannt wird und die lediglich die Anzahl ihrer Aufrufe in einem Zähler Counter2 hochzählt. Node2 sendet in einem Frame den Wert von Counter2 und den von Node1 empfangenen Wert von Counter1. Abbildung 26 stellt dieses Fallbeispiel schematisch dar.

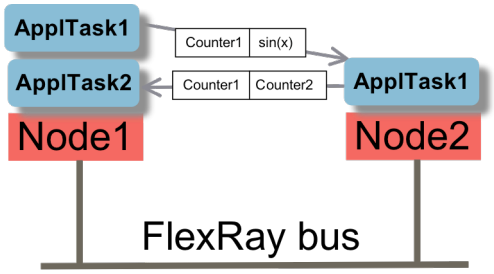

Abbildung 26. Nachrichtenaustausch im Fallbeispiel.

Mittels des DESIGNERPRO-Werkzeuges gehen wir die oben angeführten fünf Schritte durch, um die Software für dieses FlexRay-System zu erstellen.

3.1 Spezifikation der Plattform

Als Schritt (1) definieren wir in ein paar Teilschritten die Plattform. Abbildung 27 zeigt den Dialog, mit dem das „Netzwerk“ benannt wird.

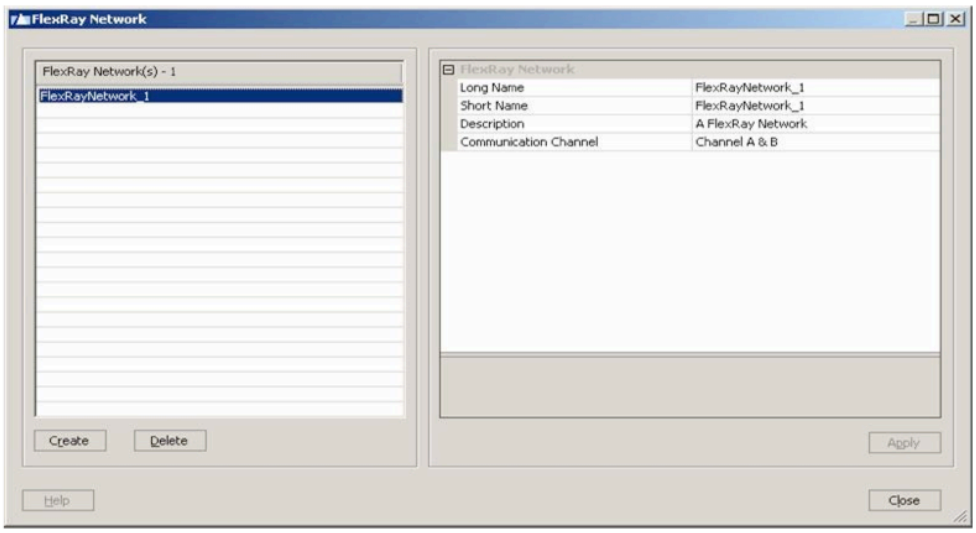

Abbildung 27. Definition des FlexRay-„Netzwerks“.

Im nächsten Teilschritt werden die ECU-Knoten definiert (siehe Abbildung 28).

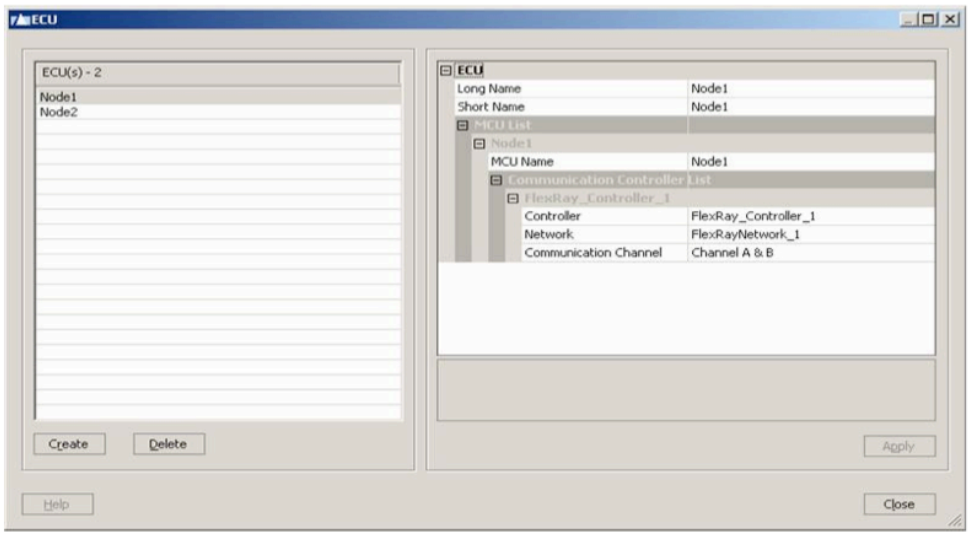

Abbildung 28. Spezifikation der ECU-Knoten.

3.2 Konfiguration der ECU-Software

In Schritt (2) erfolgt die Definition der Tasks und deren Zuordnung zu den Knoten. Abbildung 29 zeigt den entsprechenden Dialog für den ersten Teilschritt. Da zum Verständnis dieses Schrittes und der nächsten Schritte die Details der Parameter im rechten Teilfenster des Dialogs irrelevant sind, gehen wir darauf nicht ein. Ebenso ignorieren wir die beiden mit SystemTask bezeichneten Tasks. Die beiden Tasks CommTask1 und CommTask2 sind die Funktionen, die zum Kopieren der zu übertragenden Werte von der MCU zum Communication Controller benötigt werden.

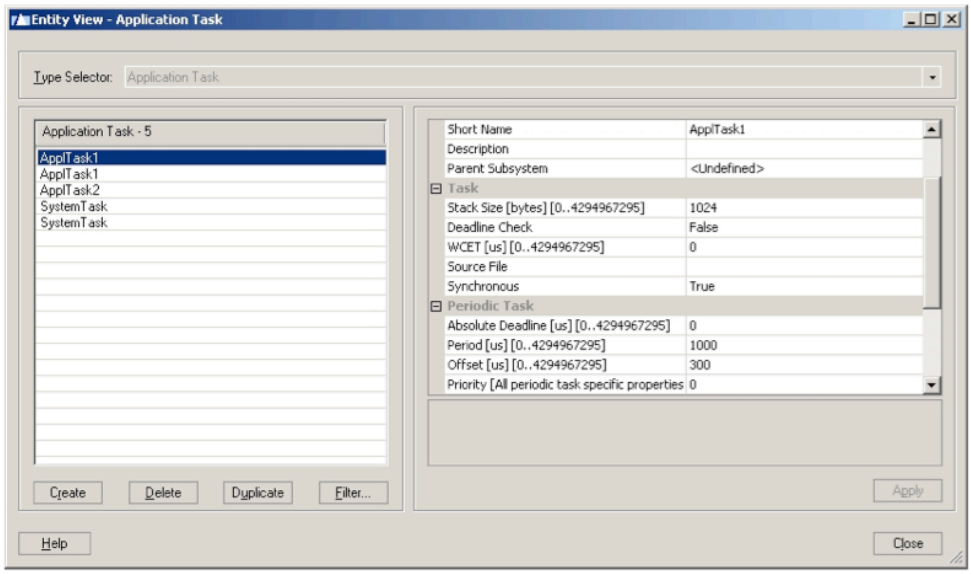

Abbildung 29. Definition der Tasks, die periodisch auszuführen sind.

Im nächsten Dialog (Abbildung 30) werden die Tasks den beiden ECU-Knoten zugeordnet.

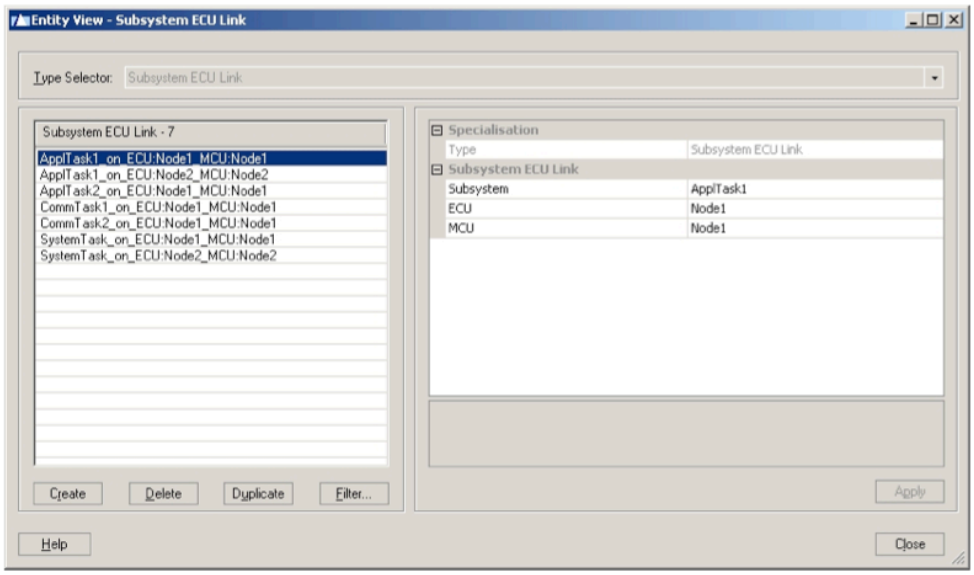

Abbildung 30. Zuordnung der Tasks zu ECU-Knoten.

3.3 Festlegung der FlexRay-Parameter

In Schritt (3) werden FlexRay-Parameter definiert. Als erster Teilschritt wird die Aufteilung in das statische und das dynamische Segment festgelegt (siehe Abbildung 31). Der Kommunikationszyklus dauert 1 ms (= 620 Mikrosekunden statisches Segment + 255 Mikrosekunden dynamisches Segment + 125 Mikrosekunden *Idle Time/Symbol Window*). DESIGNERPRO schlägt hier die angezeigte Aufteilung vor. In diesem Fallbeispiel würde das dynamische Segment überhaupt nicht benötigt werden. Auf die in der oberen und unteren Liste angeführten Parameter (gMacroPerCycle, etc.) gehen wir nicht näher ein.

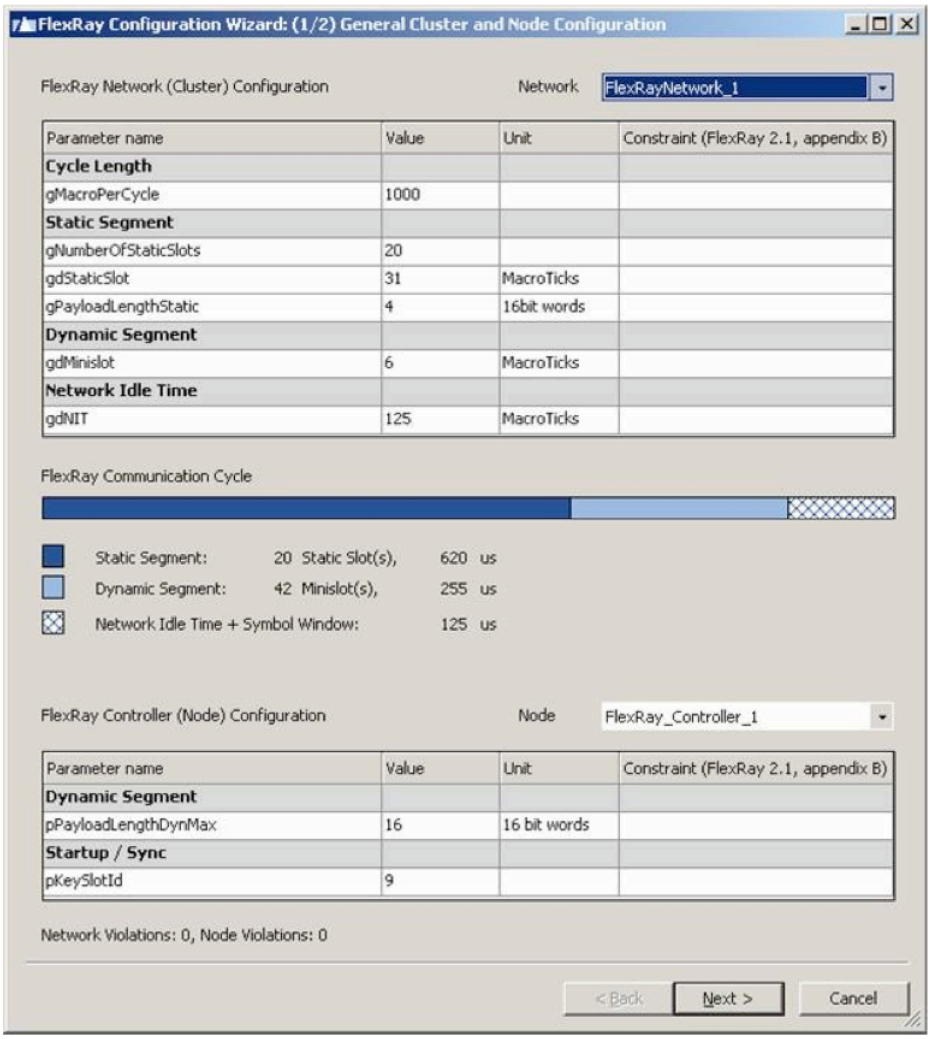

Abbildung 31. Definition des Kommunikationszyklus.

Im nächsten Teilschritt werden weitere FlexRay-Parameter eingestellt (siehe Abbildung 32), auf die auch nicht näher eingegangen wird.

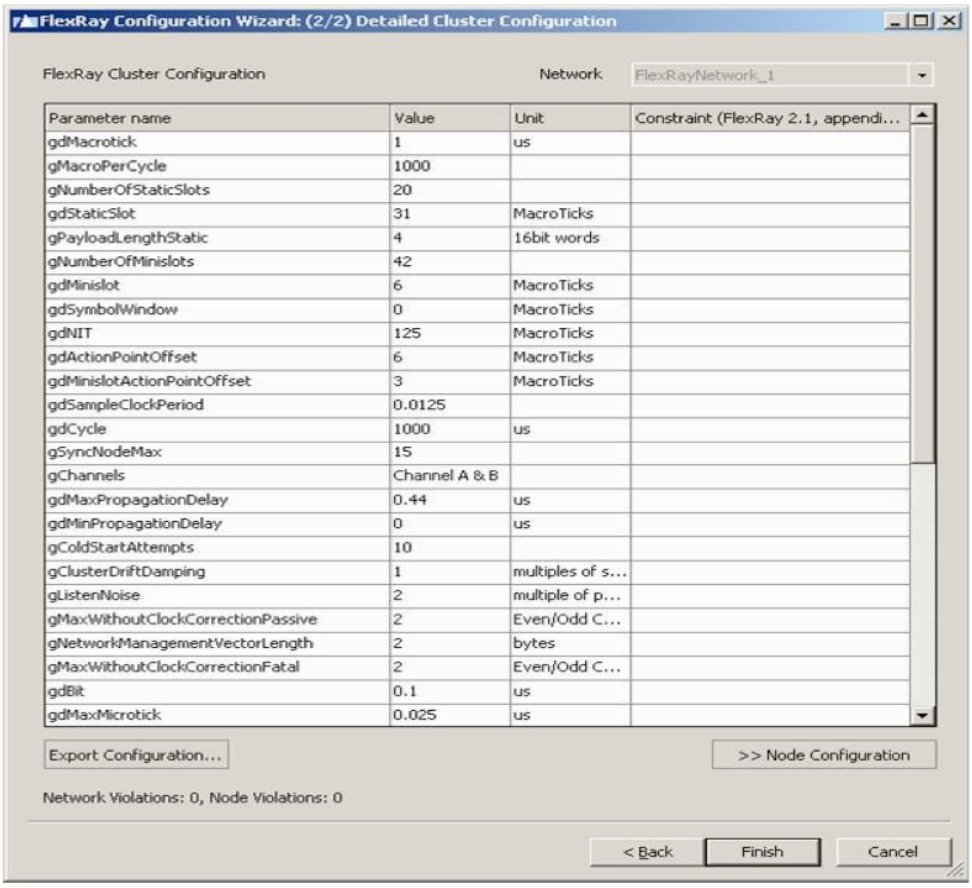

Abbildung 32. Weitere FlexRay-Parameter.

3.4 Definition des *Communication Schedules*

In Schritt (4) wird der *Communication Schedule* für die in Schritt (1) festgelegte Topologie und die in Schritt (2) definierte ECU-Software festgelegt. Als erster Teilschritt von (4) ist zu definieren, in welchem Slot – aufgrund der FlexRay-Parametrierung in Schritt (3) – welcher Frame gesendet wird (siehe Abbildung 33). Ohne auf weitere Details einzugehen, sei nur die Spalte CH (steht für *Channel*) erwähnt: darin wird festgelegt, auf welchem Kanal, also A oder B, der Slot mit dem jeweiligen Frame gesendet wird. Das erklärt, warum Frames doppelt (einmal mit dem Namens-Appendix _A und einmal mit _B) angeführt werden.

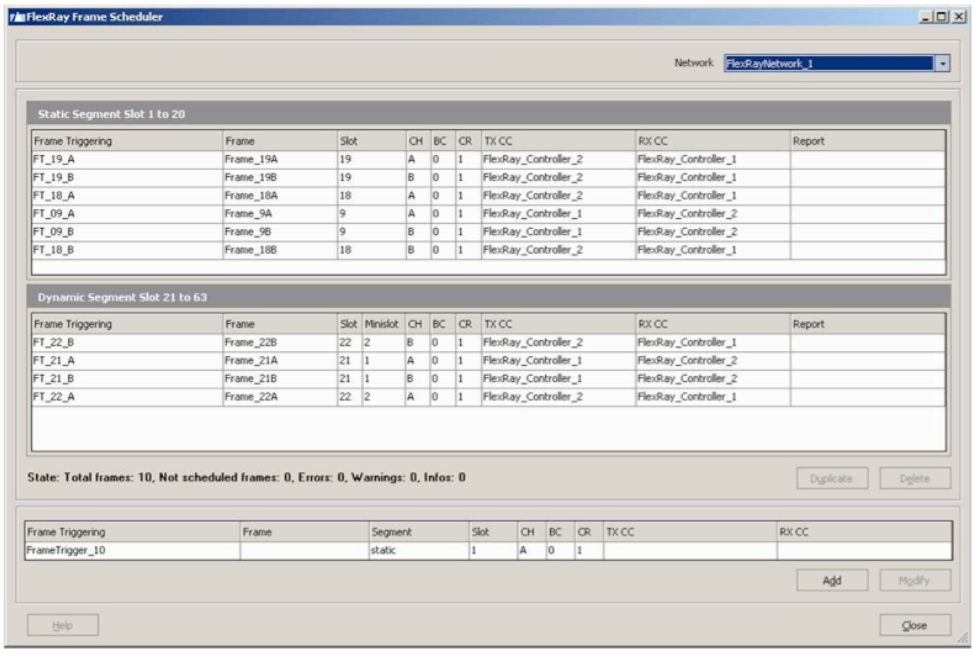

Abbildung 33. Zuordnung von Frames zu Slots.

Als nächster Teilschritt muss spezifiziert werden, welche Nachrichten (= Signale) existieren, und mit welchen Frames diese versendet werden. Die Abbildungen 34 und 35 zeigen die entsprechenden Dialoge.

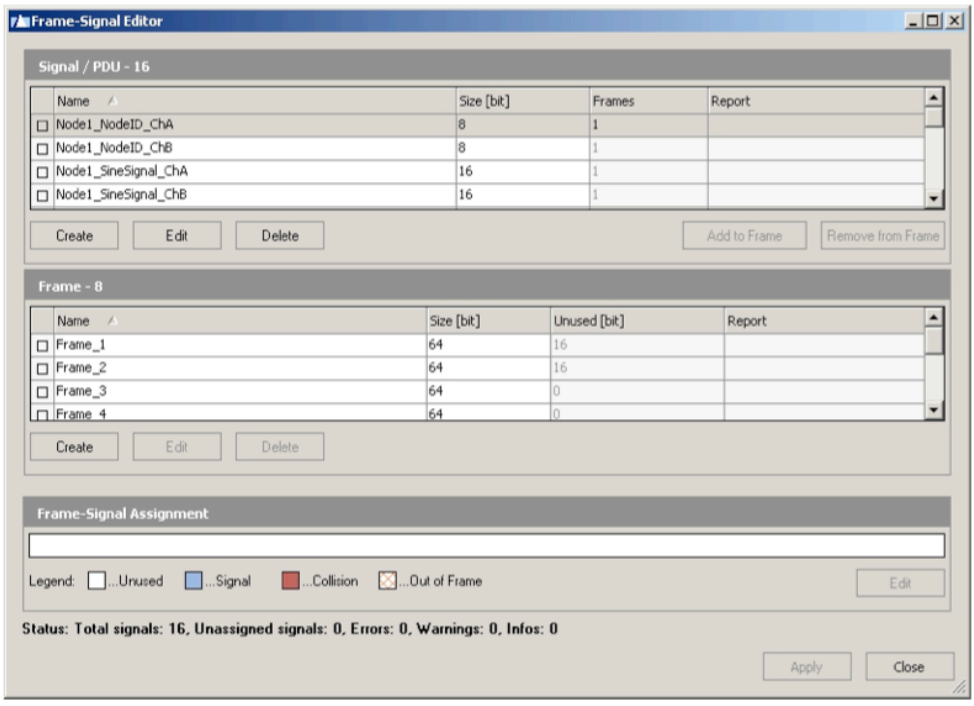

Abbildung 34. Spezifikation von Signalen ...

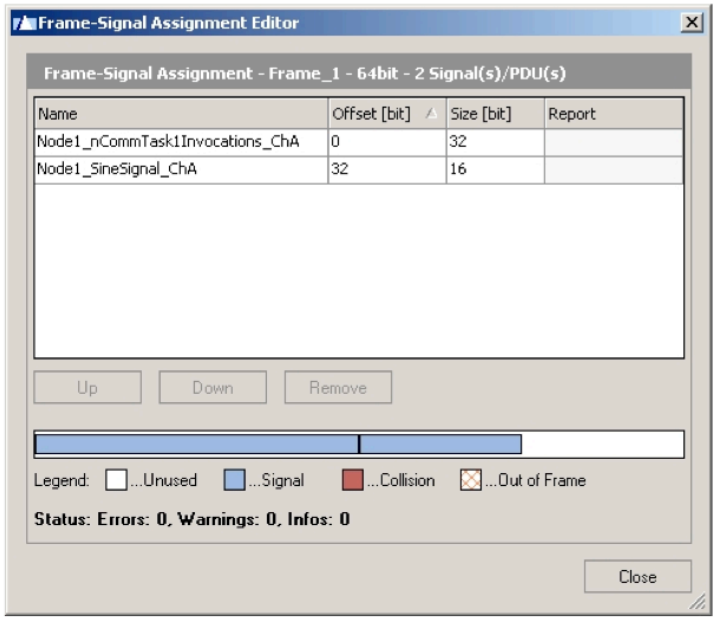

Abbildung 35. ... und deren Zuordnung zu Frames.

3.5 Treiberkonfiguration

Schließlich müssen in Schritt (5) die Treiber konfiguriert werden, und zwar für jeden *Communication Controller*. Es müssen im ersten Teilschritt die *Buffer* festgelegt werden, die Pfade für den Code-Output definiert werden und der COMMSTACK-Code generiert werden. Abbildung 36 zeigt den Dialog zur Treiberkonfiguration.

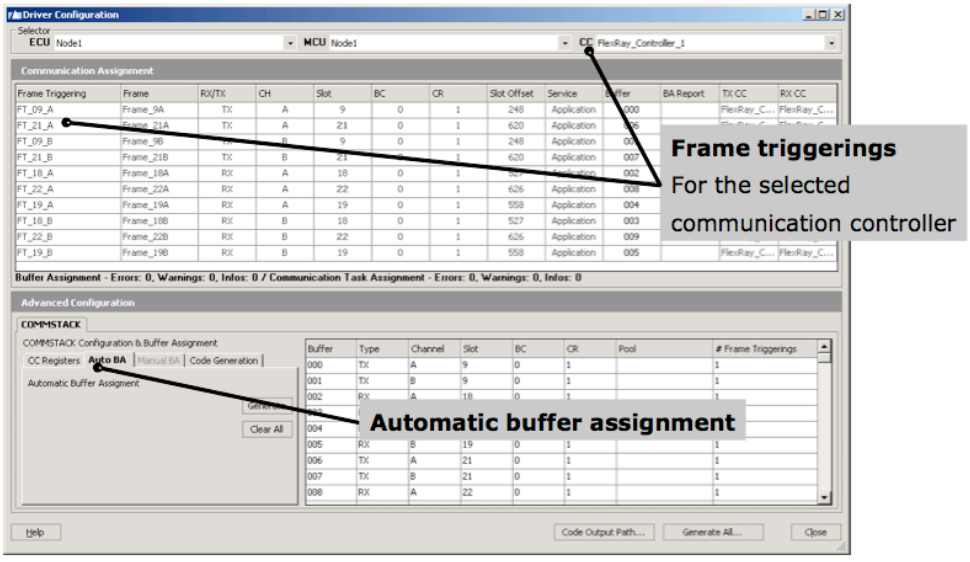

Abbildung 36. Treiberkonfiguration.

Der nächster Teilschritt ist die FLEXCOM-Konfiguration (siehe Abbildung 37). Dabei werden die Tasks zum Übertragen der Werte generiert (CommTask1, CommTask2).

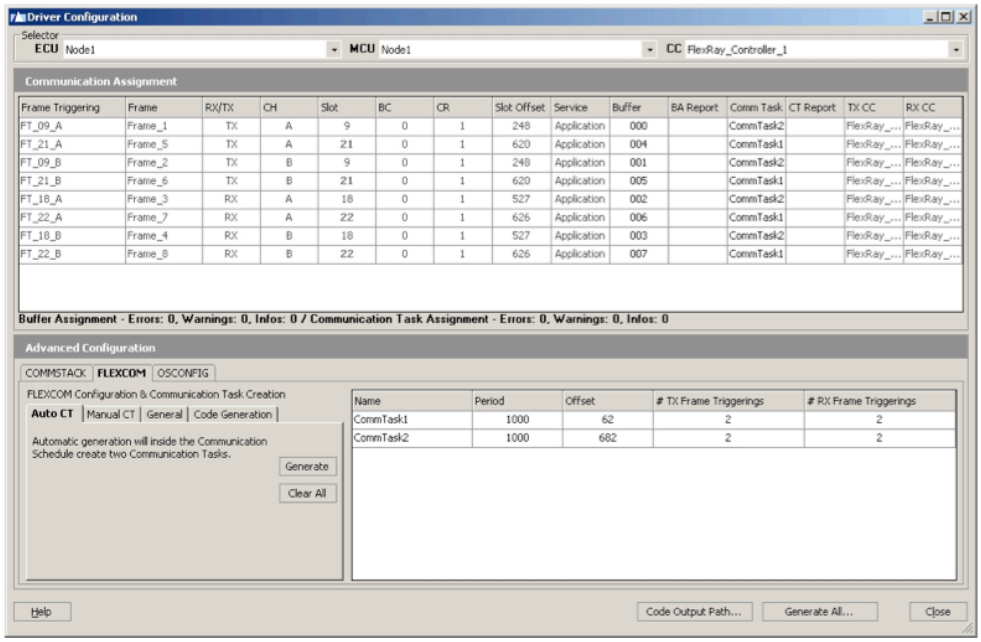

Abbildung 37. FLEXCOM Konfiguration.

Der letzte Teilschritt von (5) ist die Konfiguration des Betriebssystems, in Fall der DeComSys RENESAS Knoten das sogenannte AES (Automatic Execution System).

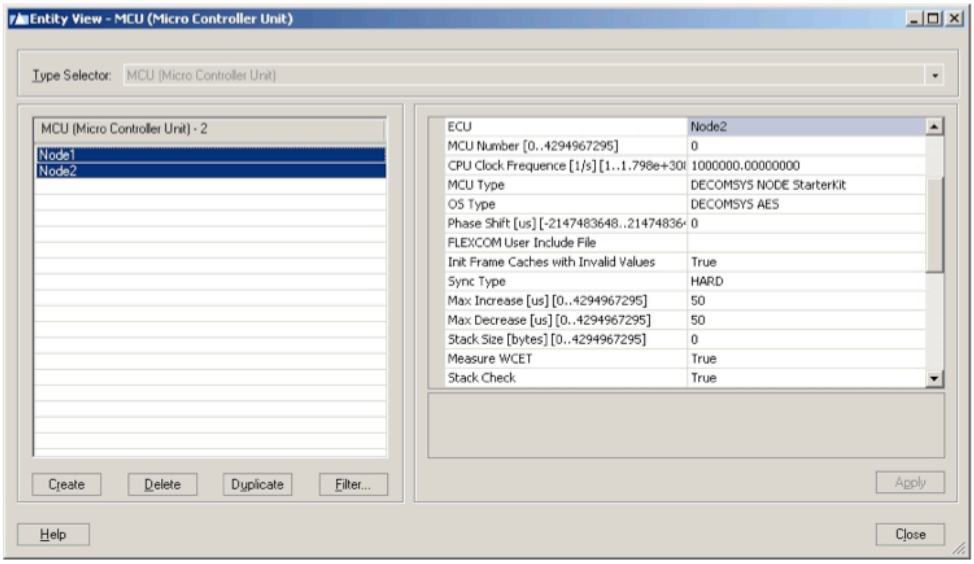

Abbildung 38. Konfiguration des Betriebssystems.

Schließlich muss sich die C-Implementierung der Tasks an bestimmte Vorgaben halten, also bestimmte Header-Dateien inkludieren, nach einem bestimmten Codier-Muster die Frames vorbereiten, etc. Dann müssen noch die periodisch auszuführenden Tasks in einem *Dispatch Table* des AES samt einer Reihe von Parametern eingetragen sein, und die Quelltexte compiliert werden. Der ausführbare Code kann dann auf die Knoten geladen und ausgeführt werden.

Nähere Informationen über die Werkzeuge von DeComSys sowie die Möglichkeit, eine Probe-Lizenz anzufordern, finden Sie unter
DeComSys.com

Die skizzierten Arbeitsschritte sollen Ihnen einen Eindruck geben, welchen beträchtlichen Aufwand es bedeutet, Software auf Basis von FlexRay auf die konventionelle Art zu entwickeln. Dabei haben wir bewusst die diversen Details der Parametrierung und *Communication Schedule* Beschreibung weggelassen, da ein detailliertes Wissen darüber im vorgegebenen Umfang einer Lektion nicht vermittelbar wäre und wir außerdem das nicht für sinnvoll erachten, wenn es darum geht, grundsätzlich zu verstehen, was es bedeutet, Software auf eine Plattform zuzuschneiden.

Man bedenke, dass ein Entwickler bei Änderungen der Topologie (zB zusätzliche Knoten) die Schritte wieder von vorne beginnen muss. Meist ist es notwendig, die Parameter und insbesondere den *Communication Schedule* völlig neu zu definieren. Der *Communication Schedule* ist in diesem Fallbeispiel einfach. Bei mehreren Knoten und Tasks, die untereinander kommunizieren, ist es bereits eine Herausforderung, überhaupt einen geeigneten *Communication Schedule* zu finden.

Das nachfolgende Kapitel zeigt, wie durch eine adäquate Abstraktion von der Plattform die Entwicklung von FlexRay-Systemen signifikant vereinfacht werden kann. Die Software-Komponenten sind transparent auf die Knoten eines FlexRay-Kommunikationssystems verteilbar. Ein Entwickler solcher Komponenten kann die Plattform, auf der diese ausgeführt werden sollen, ignorieren. Eine automatische Code-Generierung erledigt die Detailschritte, die bei der konventionellen Vorgehensweise mühsam eingestellt und definiert werden müssen. Insbesondere wird auch der *Communication Schedule* automatisch erzeugt. Wenn die Generatoren Code erzeugen, ist garantiert, dass das Verhalten der Software-Komponenten auf der Plattform exakt der Modellierung entspricht. Wenn die Plattform zuwenig Ressourcen (Rechenleistung der Knoten, Übertragungs-kapazität) bietet, wird kein Code erzeugt.

4 Modellbasierte Software-Entwicklung für FlexRay mit der *Timing Definition Language* (TDL)

An der Universität von Kalifornien in Berkeley wurde eine Abstraktion, die sogenannten *Logical Execution Time* (LET), erfunden, die für modellbasierte

FlexRay-Entwicklung adäquat ist. Andere Ansätze, wie zum Beispiel die synchronen Sprachen Esterel [Berry und Gonthier, 1992] und Lustre [Halbwachs et al., 1991], abstrahieren auf eine Art, dass eine automatische Code-Generierung für verteilte Systeme nicht oder nur suboptimal möglich ist. In diesen synchronen Sprachen geht man davon aus, dass es unendlich schnelle Computer gibt, also Task-Ausführungen keine Zeit benötigen. Eine solche Abstraktion mag für Einplatzsysteme (also einen Knoten) brauchbar sein, sie eignet sich aber nicht gut für verteilte Systeme, da eine Berechnung und die Kommunikation des berechneten Wertes eine nicht vernachlässigbare Zeit benötigen.

Die LET wurde von vorneherein so konzipiert, dass sie geschickt von verteilten Systemen abstrahiert. Es wird das Zeitverhalten von Software zunächst einmal unabhängig von der Plattform, auf der die Software ausgeführt werden soll, beschrieben. Die textuelle Sprache, die die LET-Abstraktion verwendet, ist die *Timing Definition Language* (TDL). TDL wurde an der Universität Salzburg entwickelt und ist ein offener Standard. TDL-Entwicklungswerkzeuge und Code-Generatoren werden von der Firma preeTEC [preeTEC, 2007] angeboten und von Automobil-Herstellern und -Zulieferern bereits in der Serienentwicklung eingesetzt.

Der aktuelle AUTOSAR-Standard berücksichtigt kein hartes Echtzeitverhalten. Erste Vorschläge zur Spezifikation des Zeitverhaltens greifen die Erfindung der logischen Zeitbeschreibung auf. Es wird sich erst in den nächsten Jahren zeigen, ob und wie in AUTOSAR Zeitverhalten beschrieben werden kann.

Dieses Kapitel erläutert zunächst das LET-Konzept und zeigt dann, wie man mit der TDL das Verhalten von Software plattform-neutral modelliert und wie schließlich die Software-Komponenten auf eine konkrete Plattform, wie zum Beispiel ECU-Knoten, die über ein FlexRay-Kommunikationssystem verbunden sind, abgebildet werden, also mit anderen Worten, wie auf Basis der TDL-Modellierung alles was in Kapitel 3 manuell definiert werden musste, also der Code, der *Communication Schedule* und sämtliche Parameter und Dateien, die zur Ausführung auf einem FlexRay-System nötig sind, automatisch generiert werden.

4.1 *Logical Execution Time* (LET)

Die *Logical Execution Time* (LET) besagt, dass die Eingaben (von Sensoren oder die Berechnungen einer anderen Task) einer periodisch ausgeführten, also zeit-gesteuerten Task am Beginn (*release* in Abbildung 2) der LET-Periode gelesen werden und die neu berechneten Ausgaben erst am Ende (*terminate* in Abbildung 39) der LET-Periode zur Verfügung gestellt werden. Zwischen diesen Zeitpunkten haben die Ausgänge den Wert der zuvor erfolgten Task-Ausführung.

Es wird nur angenommen, dass die Task-Ausführung auf einer spezifischen Plattform schnell genug ist, um zwischen den logischen Anfangs- und Endpunkten einer Berechnung zu erfolgen. LET bedeutet, dass das beobachtbare zeitliche Verhalten einer Task unabhängig von der tatsächlichen Ausführung auf einer Plattform ist. Zur optimalen Unterstützung von digitalen Regelungen, wie sie etwa

bei X-by-Wire-Systemen zur Anwendung kommen werden, kann die LET auch auf null gesetzt werden, wenn die Ergebnisse lokal verwendet werden, also zum Beispiel damit ein Aktuator gesetzt wird.

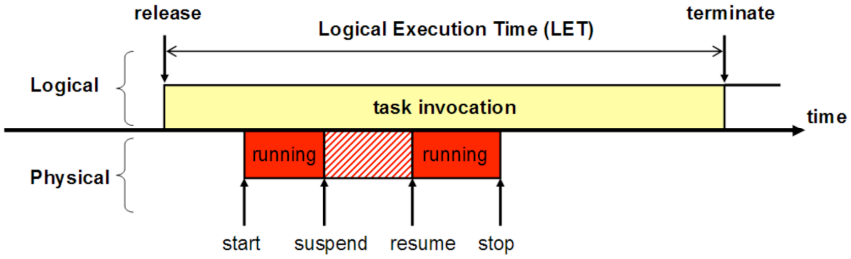

Abbildung 39. *Logical Execution Time* (LET) Abstraktion.

Die Beschreibung des Zeitverhaltens von Tasks mittels der LET bietet folgende Vorteile für die Entwicklung von Echtzeitsoftware, die bisher keine anderen am Markt erhältlichen Methoden und Werkzeuge bieten können:

1) garantierter Zeit- und Wert-Determinismus: gleiche Eingaben zu bestimmten Zeitpunkten führen immer zu denselben Ausgaben zu korrespondierenden Zeitpunkten

Die Begriffe Zeit- und Wert-Determinismus sind in Abschnitt 1.2 definiert.

2) garantiert keine Echtzeit-Anomalien, wie *Priority-Inversion* oder *Deadlocks*

Priority-Inversion und *Deadlocks* werden in Abschnitt 1.1 erläutert.

3) Plattform-neutrale Definition des Zeitverhaltens, insbesondere auch unabhängig von der Topologie eines verteilten Systems. Dadurch kann die Echtzeit-Software portiert werden, ohne dass sich das Zeitverhalten ändert (siehe oben: einmal modellieren und dann Code für eine bestimmte Plattform generieren; das könnte auch mit dem Java-Slogan formuliert werden: *develop once, deploy anywhere*).

Durch die LET kann weiters auch garantiert werden, dass das Verhalten des modellierten Systems in einer Simulation, wie zum Beispiel in einer Matlab®/Simulink®-Umgebung, immer exakt mit dem Verhalten auf einer konkreten Plattform wie zum Beispiel einem FlexRay-System übereinstimmt. Konventionelle Werkzeuge und Methoden, wie zum Beispiel die DeComSys-SIMTOOLS ermöglichen lediglich eine Simulation, die annähernd der Ausführung auf einer FlexRay-Plattform entspricht. Mit den SIMTOOLS müssen sukzessive Plattform-Details eingegeben werden, wodurch die Annäherung schrittweise besser

wird. Eine exakte Übereinstimmung zwischen Simulation und Ausführung wird bei den SIMTOOLS und analog konzipierten konventionellen Werkzeugen nicht garantiert.

4.2 TDL-Komponentenmodell

Abbildung 40 illustriert eine TDL-Komponente, die eine Einheit aus Sensoren (s1 und s2 in Abbildung 40), Aktuatoren (a1, a2 und a3) und Modi (mode1 und mode2) ist. Eine TDL-Komponente entspricht typischerweise einer Funktionseinheit im Automobil, wie zum Beispiel einer aktiven Hinterachslenkung. Eine TDL-Komponente ist zu einem bestimmten Zeitpunkt immer genau in einem Modus. Ein Modus ist eine Menge parallel ausgeführter Aktivitäten: Task-Aufrufe, Aktuator-Aktualisierungen, und die Prüfung, ob ein Wechsel in einen anderen Modus stattfinden soll. Alle diese Aktivitäten können verschiedene (logische) Ausführungszeiten haben und wenn nötig bedingt ausgeführt werden. Die Zeitangaben in eckigen Klammern hinter den Task-Namen in der TDL-Beispielkomponente in Abbildung 40 sind die jeweiligen LETs für eine Task. Die Tasks entsprechen typischerweise den Regler-Algorithmen – wir sprechen im Gegensatz zum Zeitverhalten hier von der Funktionalität. Die Funktionalität kann in einer beliebigen Sprache implementiert werden. Wenn zum Beispiel die Werkzeuge von MathWorks® verwendet werden, wird die Funktionalität in Matlab®/Simulink® modelliert und daraus C-Code generiert.

Das Lesen von Sensoren, das Setzen von Aktuatoren und das Prüfen, ob ein Wechsel zwischen zwei Modi stattfinden soll, wird in TDL in logisch null Zeit (synchron) durchgeführt. Bestimmte Teile einer TDL-Komponente, insbesondere Sensoren und Task-Output-Ports, können als *public* deklariert werden. Andere TDL-Komponenten können dann darauf zugreifen. In der Beispielkomponente sind die Sensoren s1 und s2 *public*, was in Abbildung 40 durch deren Darstellung außerhalb der Komponente ausgedrückt werden soll.

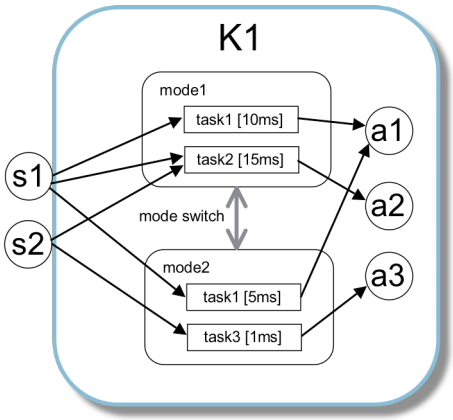

Abbildung 40. Schematische Darstellung einer TDL-Komponente K1.

Abbildung 41 zeigt das logische Zeitverhalten, wenn K1 im Modus *mode1* ist, also die Task *task1* mit einer LET von 10 Millisekunden und die Task *task2* mit einer LET von 15 Millisekunden ausgeführt werden. Der Sensorwert s1 wird jedesmal

vor Beginn der Ausführung beider Tasks als Eingabe zur Verfügung gestellt. Am Ende der LET von *task1*, also alle 10 Millisekunden, wird der von *task1* berechnete Wert auf den Aktuator a1 geschrieben. Am Ende der LET von *task2*, also alle 15 Millisekunden, wird der von *task2* berechnete Wert auf den Aktuator a2 geschrieben.

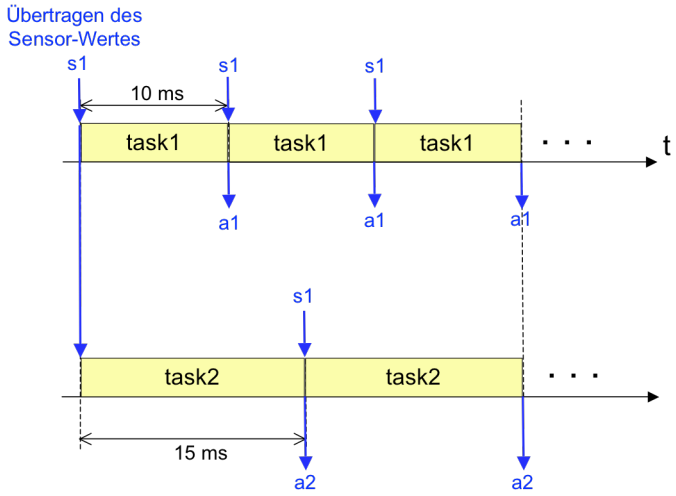

Abbildung 41. Logische Ausführungszeiten der Tasks in *mode1*.

4.3 Komponenten-Beziehungen und transparente Verteilung

Eine TDL-Komponente kann eine andere TDL-Komponente importieren und unter anderem auf deren als *public* deklarierten Sensoren oder Task-Outputs zugreifen.

Abbildung 42 zeigt ein vereinfachtes Beispiel mit zwei TDL-Komponenten, in dem die TDL-Komponente ARS (*Active Rear Steering*) die TDL-Komponente EngineCtrl (*Engine Controller*) importiert.

Der Output von *task1* in der EngineCtrl-Komponente ist *public*. Damit kann das von *task1* periodisch mit einer LET von 5 Millisekunden berechnete Ergebnis von *task2* in der TDL-Komponente ARS verwendet werden.

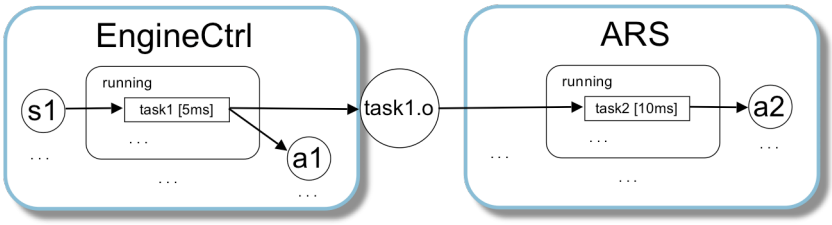

Abbildung 42. Eine TDL-Komponente verwendet den Output der Task einer anderen TDL-Komponente.

Abbildung 43 stellt die entsprechenden logischen Ausführungszeiten und die Zeitpunkte, wann Werte zwischen den beiden Tasks kopiert werden, dar. Das Einlesen der Sensor-Werte und die Ausgabe der berechneten Resultate auf Aktuatoren ist der Übersichtlichkeit halber nicht dargestellt. In der Komponente EngineCtrl wird *task1* mit einer LET von 5 Millisekunden ausgeführt. In der Komponente ARS wird *task2* mit einer LET von 10 Millisekunden ausgeführt. Deswegen muss von *task1* am Ende jeder zweiten LET-Periode das berechnete Ergebnis als Eingabe für *task2* zur Verfügung gestellt werden, und zwar laut LET-Programmiermodell synchron, also ohne Zeitverzögerung.

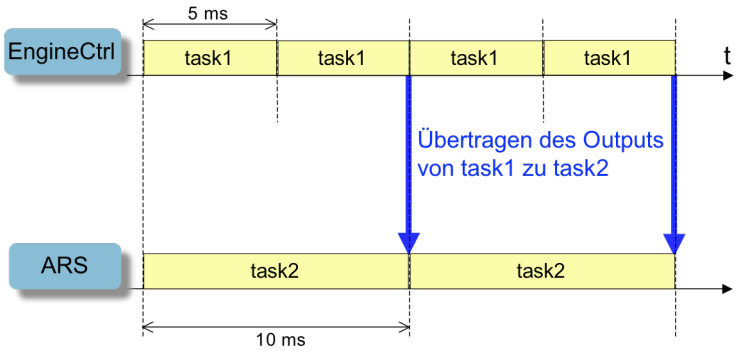

Abbildung 43. Logische Ausführungszeiten und Kommunikation zwischen Tasks.

Der Entwickler von TDL-Komponenten modelliert nur das logische Zeitverhalten und die Funktionalität der Tasks. Bitte beachten Sie, dass bei dieser Modellierung die Plattform in keiner Weise vorkommt. Die TDL-Komponenten repräsentieren also plattform-neutral das Zeitverhalten und die Funktionalität von Software-Komponenten. Da die LET-Abstraktion geschickt gewählt wurde, können diese Komponenten „transparent verteilt“ werden [Farcas et al., 2005]: Zur Laufzeit verhalten sich TDL-Komponenten exakt gleich, unabhängig davon, ob sie auf einem Einplatzsystem (*single node*) oder in einem beliebig verteilten System ausgeführt werden. Das logische Zeitverhalten ist immer gleich. Nur das plattform-spezifische Zeitverhalten, das nicht beobachtbar ist – also wann welche Tasks physisch ausgeführt werden und im Fall eines verteilten Systems wann welche Nachrichten übertragen werden – kann sich ändern. Die TDL-Code-Generatoren erzeugen den entsprechenden Code und *Communication Schedule* für die spezifische Plattform. Wenn nicht genügend Ressourcen (Rechenleistung pro Knoten beziehungsweise Übertragungskapazität im Kommunikationssystem) zur Verfügung stehen, wird kein Code generiert. Wenn Code generiert wird, ist garantiert, dass das Verhalten der Software-Komponenten auf der ausgewählten Plattform äquivalent zum mittels TDL spezifizierten Verhalten ist. Ein Entwickler muss daher nicht darauf Rücksicht nehmen, wo eine TDL-Komponente beziehungsweise die davon importierten TDL-Komponenten ausgeführt werden.

Transparente Verteilung von TDL-Komponenten: Zur Laufzeit verhalten sich TDL-Komponenten exakt gleich, unabhängig davon, ob sie auf einem Einplatzsystem (*single node*) oder in einem verteilten System wie zum Beispiel einem FlexRay-

System ausgeführt werden. Für den Entwickler bedeutet das den Vorteil, bei der Modellierung die Ausführungsplattform ignorieren zu können.

Nachfolgend sehen Sie das TDL-Programm, das die Komponente EngineCtrl entsprechend der schematischen Darstellung in Abbildung 42 modelliert. In Zeile 1 wird definiert, dass die Komponente EngineCtrl heißt. Zeile 3 definiert den Sensor s1 als Boole'schen Eingang. Das Schlüsselwort uses drückt aus, dass eine externe Funktion zum Lesen des Sensor-Wertes verwendet wird. Diese externe Funktion ist in diesem Fall getS1. Da mit TDL plattform-unabhängig modelliert wird, werden plattform-spezifische Aspekte, wie das Lesen von Sensoren oder das Schreiben von Aktuatoren von externen Funktionen durchgeführt. Diese externen Funktionen können in einer beliebigen imperativen Sprache vorliegen. Typischerweise ist das die Programmiersprache C. Bei vielen Plattformen gibt es zudem Codiermuster, wie auf welche Sensor-Eingänge beziehungsweise Aktuatorausgänge zugegriffen wird. Dann lässt sich der Code für solche externen Funktionen automatisch generieren.

```
01    component EngineCtrl {
02
03      sensor boolean s1 uses getS1;
04      ...  // weitere Sensoren
05      actuator int a1 uses setA1;
06      ... // weitere Aktuatoren
07
08      public task task1 {
09        output int o := 10;
10        uses task1Impl(s1, o);
11      }
12
13     mode running [period=5ms] {
14        task
15          [freq=1] task1();        // LET = 5ms / 1 = 5ms
16        actuator
17          [freq=1] a1 := task1.o;  // a1 alle 5ms setzen
18        ...
19     }
20     ...
21    }
```

In den Zeilen 8 bis 11 wird die Task task1 deklariert. Die Task hat lediglich einen Ausgang o, der mit dem Wert 10 initialisiert wird. Die externe Funktion, die die Task implementiert, ist task1Impl. Diese kann wiederum als C-Funktion vorliegen. Wenn TDL zusammen mit Matlab®/Simulink® verwendet wird (siehe Abschnitt 4.4), kann die Funktion auch damit modelliert und die TDL-Komponente in dieser Umgebung simuliert werden. Das Schlüsselwort public vor der Task bedeutet, dass deren Ausgänge, also in diesem Fall o, von anderen Komponenten verwendbar sind.

Schließlich wird das Zeitverhalten in der Modus-Deklaration (Zeilen 13 bis 19) festgelegt. Der Modus running hat eine Periode von 5 Millisekunden. Alle Aktivitäten innerhalb der Modus-Deklaration beziehen sich auf diese Periode. In

Zeile 15 wird festgelegt, dass die task1 in diesem Modus eine LET von 5 Millisekunden hat (Aufruf einmal pro Periode). Hier könnten mehrere Task-Aufrufe, die parallel zu erfolgen haben, definiert werden. In diesem einfachen Beispiel wird nur task1 im Modus running aufgerufen. In Zeile 17 wird definiert, dass das von task1 berechnete Ergebnis alle 5 Millisekunden auf den Aktuator a1 geschrieben wird.

Abschließend betrachten wir das TDL-Programm der Komponente ARS, die die Komponente EngineCtrl importiert (Zeile 3). Beim Aufruf der Task task2 im Modus running (Zeile 10) wird der Output der Task task1 der Komponente EngineCtrl einfach durch Hinschreiben von EngineCtrl.task1.o verwendet.

```
01    component ARS {
02
03      import EngineCtrl;
04      …
05      task task2 {
06        input int i;
07        …
08      }
09      mode running [period=10ms] {
10        task [freq=1] task2(EngineCtrl.task1.o);// 10ms LET
11        …
12      }
13      ...
14    }
```

Statt der textuellen Notation können TDL-Komponenten ebenso visuell/interaktiv modelliert werden. Abschnitt 4.4 zeigt anhand eines weiteren Fallbeispiels, wie Sie das TDL:VisualCreator-Werkzeug dabei unterstützt.

Code-Generierung und Ausführung auf einer spezifischen Plattform. In Abbildung 44 nehmen wir an, dass die ARS-Komponente auf ECU1 und die EngineCtrl-Komponente auf ECU2 eines FlexRay-Systems ausgeführt werden sollen. Die roten Rechtecke stellen analog zu Abbildung 39 die angenommene physische Ausführungszeit auf der jeweiligen ECU dar, die gelben Rechtecke stellen die logischen Ausführungszeiten dar. Die TDL-Spezifikation des Zeitverhaltens mittels LET gibt dem *Communication Schedule*-Generator vor, welche Kommunikations-Einschränkungen vorliegen. In unserem Beispiel kann die Kommunikation des Outputs von *task1* nach dem Ende der tatsächlichen Berechnung beginnen. Der Output-Wert muss bis spätestens zum Ende der LET-Periode als Input für *task2* auf ECU2 vorliegen. Dieser Abschnitt auf der Zeitachse ist in Abbildung 44 als „Zeitfenster für Kommunikation“ dargestellt. Wenn der Wert von ECU1 auf ECU2 übertragen wurde, wird er in einem lokalen Buffer gespeichert und exakt dann, wann er von *task2* benötigt wird, als Eingabe zur Verfügung gestellt.

Aufgrund der Zuordnung von TDL-Komponenten zu ECU-Knoten mit dem sogenannten TDL:VisualDistributor-Werkzeug (siehe Abschnitt 4.5) generiert dieses Werkzeug, wenn möglich, einen *Communication Schedule* sowie die diversen Parameter, um ein voll funktionsfähiges FlexRay-System zu erhalten.

Die in Kapitel 3 dargestellten Entwicklungsschritte bei der konventionellen, „manuellen“ FlexRay-Entwicklung sind somit bei Verwendung von TDL vollständig automatisiert.

Mit anderen Worten formuliert: Ein Entwickler muss lediglich die TDL-Komponenten, insbesondere deren LETs, spezifizieren. Diese Komponenten können dann transparent, zum Beispiel auf einem FlexRay-System, verteilt werden. Der entsprechende Code, der *Communication Schedule* und die FlexRay-Parameter werden automatisch generiert.

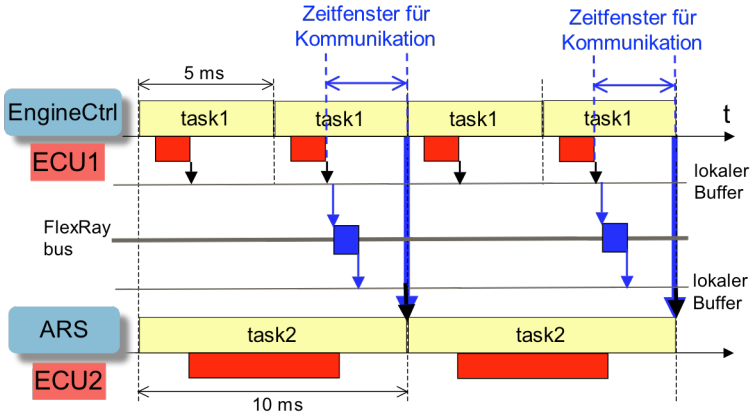

Abbildung 44. LET-Vorgaben zur *Communication Schedule*-Generierung.

4.4 Visuell/interaktive Modellierung von TDL-Komponenten

Wie oben anhand der beiden Komponenten EngineCtrl und ARS skizziert wurde, können TDL-Komponenten textuell modelliert werden. Die textuelle Syntax und Semantik von TDL sind in [Templ, 2007] definiert. Alternativ zur textuellen Beschreibung können TDL-Komponenten visuell/interaktiv mit dem TDL:VisualCreator-Werkzeug modelliert werden. Der TDL:VisualCreator ist ein Syntax-getriebener TDL-Editor, der die gleichen Sprachkonstrukte wie die textuelle Notation bietet. Der TDL:VisualCreator kann als *stand-alone* Werkzeug oder integriert in Matlab®/Simulink® verwendet werden. Eine *stand-alone*-Verwendung macht Sinn, wenn man TDL visuell/interaktiv modellieren will, die Funktionalität der Tasks jedoch als bestehender C-Code vorliegt, also eine Siumlation in Matlab®/Simulink® nicht möglich ist. Wenn die Task-Funktionen bereits als Matlab®/Simulink®-Modelle vorliegen oder damit modelliert werden sollen, empfiehlt sich die Verwendung des TDL:VisualCreators in dieser Umgebung. Da Matlab®/Simulink® ein De-Facto-Standard zur Modellierung und Simulation von Reglern geworden ist, zeigen wir in diesem Abschnitt wie TDL-Komponenten mit dem TDL:VisualCreator in Matlab®/Simulink® modelliert und simuliert werden.

Nachfolgend modellieren wir eine vereinfachte Version einer aktiven Hinterachslenkung. Die Fallstudie wurde von Magna Steyr Fahrzeugtechnik [2] zur Verfügung gestellt.

Eine detaillierte Video-Vorführung der TDL-Werkzeuge anhand dieser Fallstudie finden Sie unter

preeTEC.com Product Demo

Diese Video-Vorführung der Werkzeuge ist eine Alternative zum Durchlesen dieses Abschnittes.

Von der Web-Seite können Sie auch eine Vorführversion des TDL:VisualCreator-Werkzeuges für Matlab®/Simulink® laden.

Als erster Schritt bei der Modellierung wird eine TDL-Komponente von einer Komponentenbibliothek (dem Simulink® Library Browser, siehe Abbildung 45) in das entsprechende Simulink®-Modell gezogen.

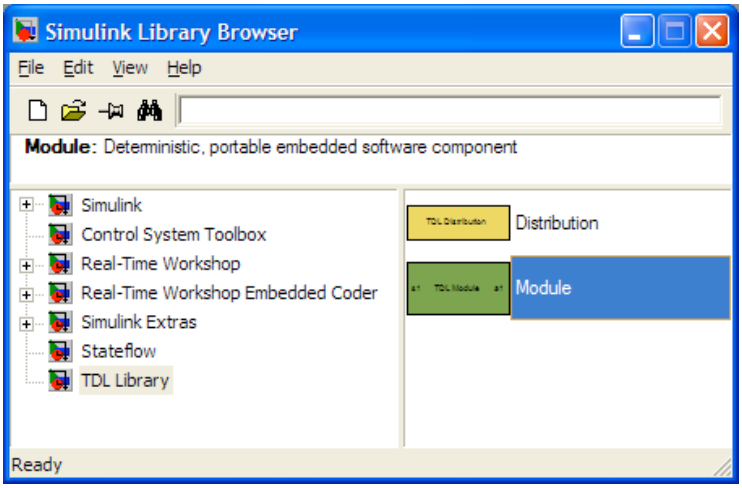

Abbildung 45. TDL-Komponente (= *Module*) in Matlab®/Simulink®

Durch einen Doppel-Klick auf diesen TDL-*Module* wird das TDL:VisualCreator-Werkzeug geöffnet, in dem die diversen Elemente einer TDL-Komponente spezifiziert werden (siehe Abbildung 46). Der Baum links listet die TDL-Konstrukte auf: die importierten Komponenten im Ordner Imports, die Konstanten, Typen[1], Sensoren, Aktuatoren, die Task-Deklarationen im Ordner Tasks, sowie die Modi einer TDL-Komponente. Abbildung 46 zeigt, dass wir drei Sensoren (delta_r_act, angular_rate, current) und einen Aktuator (voltage) der TDL-Komponente RearActuatorController definiert haben. Durch Auswahl eines

[1] Auf die Möglichkeit, in TDL Konstanten und Typen zu deklarieren, wurde bisher nicht eingegangen. Da wir diese TDL-Konstrukte auch nicht im Fallbeispiel benötigen, gehen wir darauf nicht weiter ein.

Elements können dessen Eigenschaften festgelegt werden. In Abbildung 46 werden die Eigenschaften des Sensors current unterhalb des Baums gezeigt.

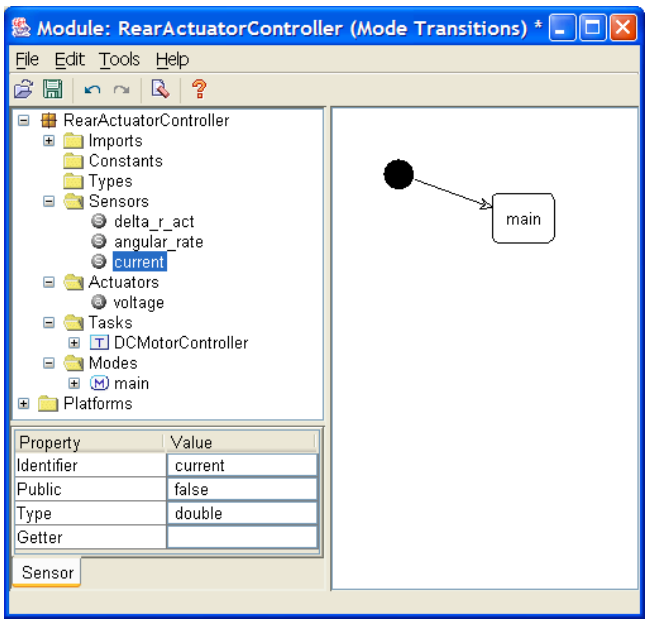

Abbildung 46. Editieren einer TDL-Komponente im TDL:VisualCreator

Die TDL-Komponente RearActuatorController hat in dieser vereinfachten Fallstudie nur eine einzige Task DCMotorController, die auch im Baum in Abbildung 46 aufgelistet ist. Der Modus main ist der einzige Modus in der TDL-Komponente RearActuatorController.

Als nächstes zeigen wir, wie die Funktionalität der Task DCMotorController und deren Zeitverhalten (also deren LET) definiert werden. Durch einen Doppel-Klick auf die Task wird ein Simulink®-Editor geöffnet. Alle verfügbaren diskreten Simulink®-Blöcke können verwendet werden, um den Regler-Algorithmus zu realisieren. Abbildung 47 zeigt das exemplarisch für die Task DCMotorController.

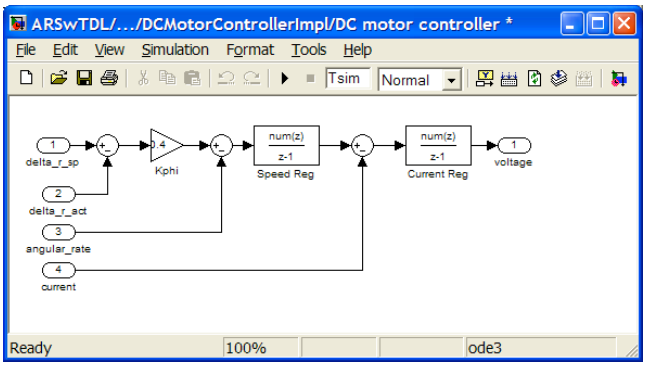

Abbildung 47. Modellierung der Funktionalität einer Task in Matlab®/Simulink®

Im nächsten Schritt legen wir die Zeitperiode fest, innerhalb der der Modus main seine Aktivitäten wiederholt. Weiters definieren wir, von wo die Task ihre

Eingaben erhält und an wen sie die Ausgaben weitergibt. Dazu wählen wir den Modus main im Baum aus (siehe Abbildung 48) und setzen dessen Periode auf eine Millisekunde (1 ms). Im Datenflusseditor (rechtes Teilfenster in Abbildung 48) legen wir die Eingaben und Ausgaben der Task fest. Beispielsweise stellt die Ausgabe der Task den Aktuator voltage ein.

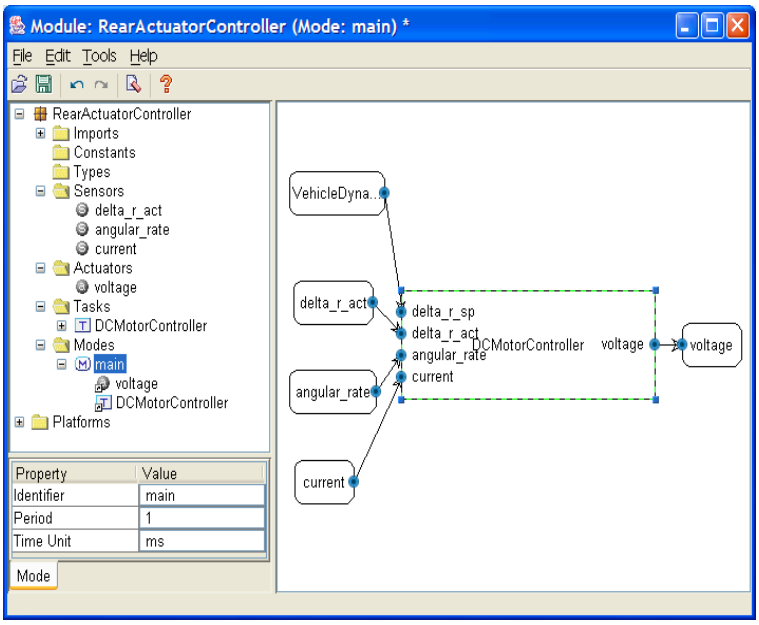

Abbildung 48. Definition des Zeitverhaltens des Modus main

Abschließend definieren wir die LET der Task DCMotorController im Modus main (siehe Abbildung 49). Man beachte, dass für jede Task verschiedene LETs in verschiedenen Modi definiert werden können. Da die Frequenz auf 1 gesetzt wird, bedeutet das, dass die LET 1 ms (Periode von Modus main) dividiert durch 1 (die Frequenz), also 1 ms beträgt.

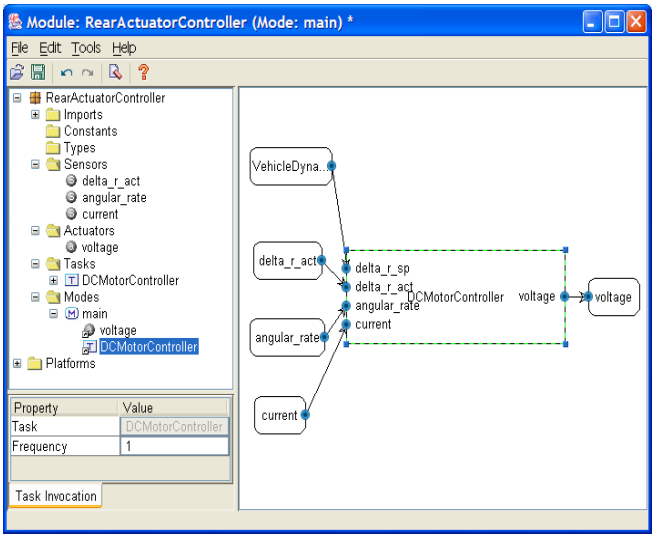

Abbildung 49. Definition der LET einer Task

Abbildung 50 zeigt das vollständige Modell der aktiven Hinterachslenkung, das aus den zwei TDL-Komponenten RearActuatorController und VehicleDynamics besteht. Das Subsystem Vehicle modelliert die relevanten Aspekte des Fahrzeugverhaltens.

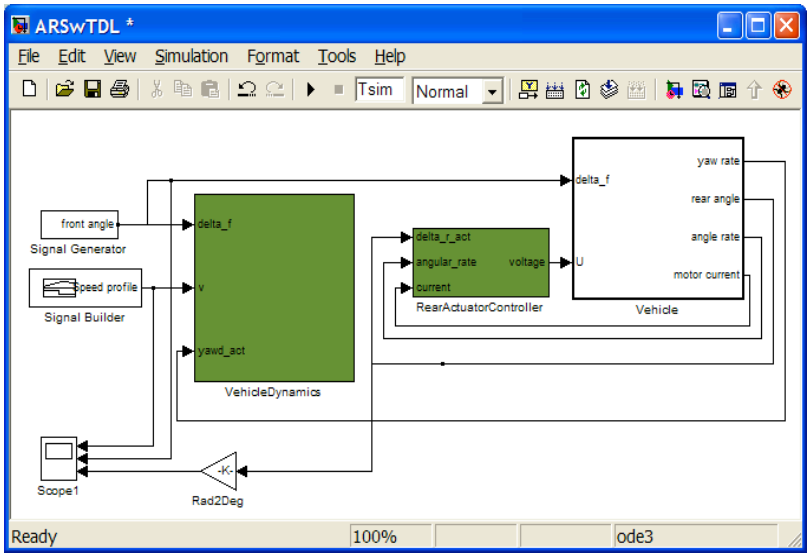

Abbildung 50. ARS-Modell mit zwei TDL-Komponenten

Der *Scope*-Block (Abbildung 51: *Speed, Front angle, Rear angle*) zeigt das Systemverhalten: Bei niedriger Geschwindigkeit werden die Hinterräder in die andere Richtung als die Vorderräder gelenkt. Bei höheren Geschwindigkeiten zeigen beide Radpaare in die gleiche Richtung.

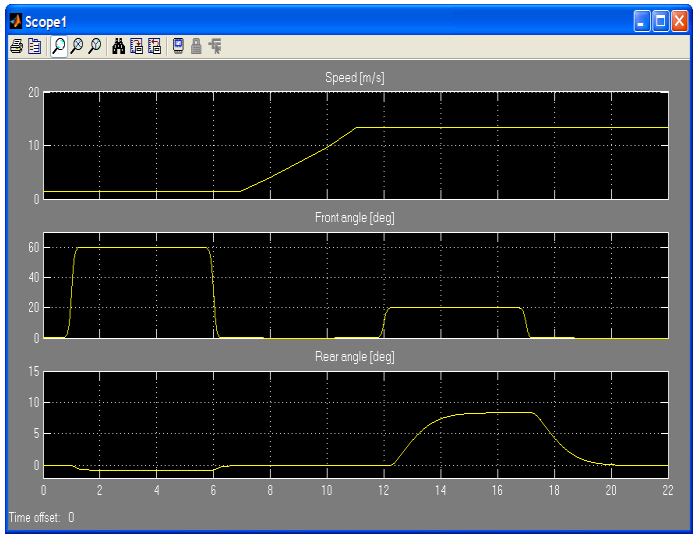

Abbildung 51. Simulation des Verhaltens der aktiven Hinterachslenkung

4.5 Ausführung von TDL-Komponenten auf einem FlexRay-System

Um TDL-Komponenten auf Knoten einer bestimmten Plattform abzubilden, zieht der Entwickler beziehungsweise System-Integrator einen sogenannten Distribution-Block vom Simulink® Library Browser in das Modell (siehe Abbildung 52). Man beachte, dass beliebig viele Zuordnungen von TDL-Komponenten zu Plattformen definiert werden können, also beliebig viele Distribution-Blöcke in einem Modell definiert werden können.

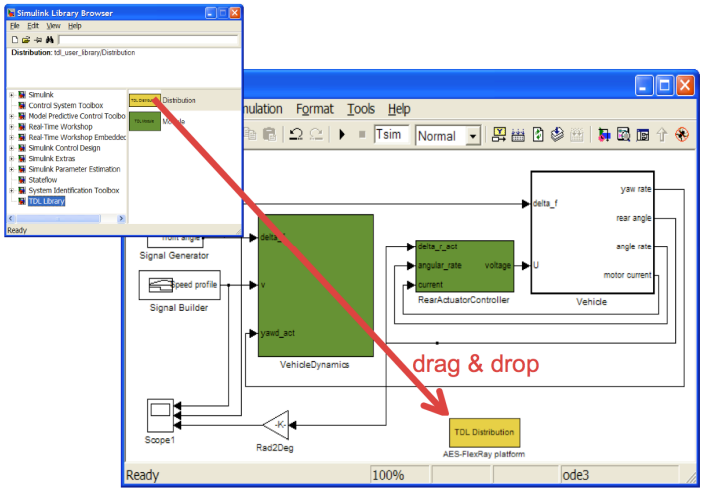

Abbildung 52. Hinzufügen eines Distribution-Blocks zu einem Modell

Ein Doppelklick auf einen Distribution-Block öffnet das TDL:VisualDistributor-Werkzeug. Damit können sowohl die Plattform und Topologie als auch die Komponenten-zu-Knoten-Zuordnung definiert werden. Wir nehmen an, dass ein FlexRay-System bestehend aus einem Renesas-Knoten und einer Micro-Auto-Box von dSpace gegeben ist und die Beschreibung dieser Plattform bereits erfolgte und gespeichert wurde. Daher kann die Plattform-Beschreibung in den TDL:VisualDistributor geladen werden. Abbildung 53 zeigt die Darstellung der Plattform als Baum im linken Teilfenster.

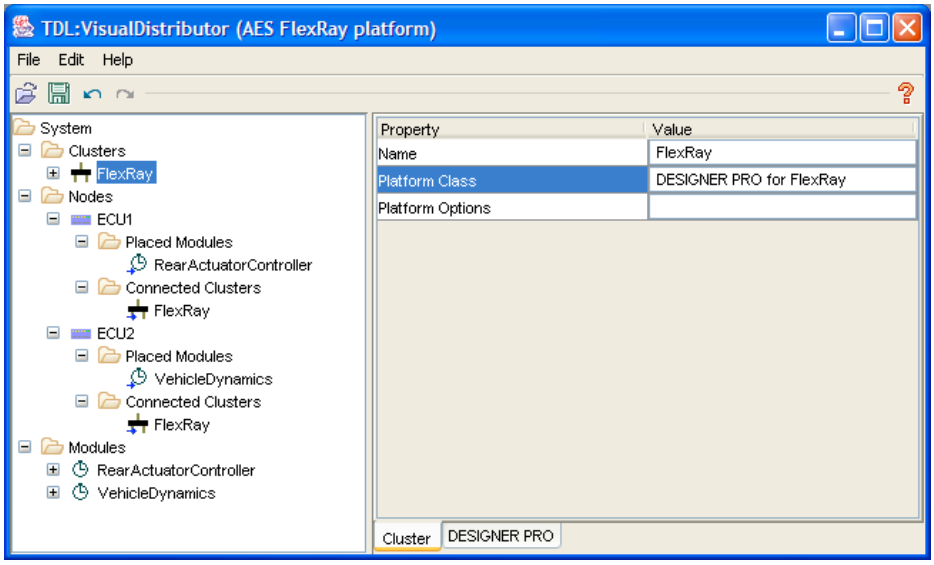

Abbildung 53. FlexRay-System mit zwei ECUs

Die Zuordnung der beiden TDL-Komponenten zu den beiden ECU-Knoten erfolgt nun einfach durch Hinziehen (*drag & drop*) der TDL-Komponenten (aus dem Ordner *Modules*) zum jeweiligen Knoten. Durch Auswählen des Menu-Eintrages *Build All* (siehe Abbildung 15) werden sämtliche benötigten Dateien generiert: der TDL-Quelltext, der C-Code (zB mit Real-Time Workshop Embedded Coder) für jede Task-Funktion, der *Communication Schedule* (auch als FIBEX[2]-Datei) sowie die FlexRay-spezifischen Parameter und Einstellungen und makefiles zur Automatisierung des Compilierens. Nach dem Compilieren der Quelltexte und dem Laden der ausführbaren Dateien auf die einzelnen ECUs verhält sich das System exakt so wie es in TDL beschrieben und in Matlab®/Simulink® simuliert wurde.

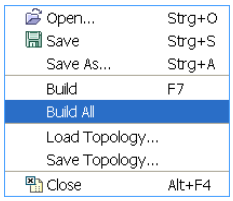

Abbildung 15. Code und *Communication-Schedule*-Generierung

Status und Ausblick

Die beiden Werkzeuge TDL:VisualCreator und TDL:VisualDistributor sind als Produkte der Firma preeTEC [preeTEC, 2007] erhältlich. Die Werkzeuge werden gerade dahingehend erweitert, dass bestehende *Communication Schedules* eingelesen und bei Bedarf inkrementell erweitert werden können. Damit ist eine Migration von bestehenden FlexRay-Anwendungen nach TDL möglich.

---

2 FIBEX steht für *Field Bus EXchange format* und wurde von der Association for Standardisation of Automation- and Measuring Systems (ASAM, www.asam.de) definiert. Mit FIBEX soll der Austausch von Informationen zwischen Werkzeugen verschiedener Anbieter ermöglicht werden.

Das Kapitel hat anhand verschiedener Beispiele und Fallstudien illustriert, wie Software plattform-neutral modelliert und auf eine spezifische verteilte Plattform wie zum Beispiel ein FlexRay-System abgebildet wird (*platform mapping*). Zur Modellierung wurde die *Timing Definition Language* (TDL) verwendet. Das Anwenden von TDL setzt das Verständnis der der TDL zugrundeliegenden Abstraktion, der *Logical Execution Time* (LET), voraus. Wichtig ist die Erkenntnis, dass durch die LET-Abstraktion von Plattform-Details abstrahiert wird und ein transparentes Verteilen der Software unter Einhaltung der LET möglich wird.

5 Anbieter von Entwicklungs-, Analyse- und Test-Werkzeugen

Ohne Anspruch auf Vollständigkeit wird nachfolgend eine Auswahl von Firmen in alfabetischer Reihenfolge angeführt, die Produkte zur Entwicklung und zum Testen von Software für verteilte Systeme im Automobil anbieten. Bei jeder Firma werden jeweils ein paar Produkte angeführt, die in diesem Zusammenhang relevant erscheinen. Nähere Informationen finden Sie auf den Web-Seiten der Firmen.

- DeComSys (DeComSys.com)
  Produkte: DESIGNERPRO & SIMTOOLS (FlexRay-Konfiguration), FlexRay-Diagnose, -Testen: BUSDOCTOR, BUSMIRROR CONFIG

- dSPACE (dSPACE.com)
  Produkte: SystemDesk (Architektur verteilter Systeme im Automobil), FlexRay Configuration Package, ControlDesk, AutomationDesk, MTest, TargetLink, Real-Time Interface (RTI), RTI-LIN-Block-Set, RTI-CAN-Blockset, C-Compiler

- ElektroBit, vormals 3Soft (elektrobit.com)
  Produkte: tresos (LIN, Autosar), GUIDE (GUI-Entwicklung)

- Esterel Technologies (Esterel-Technologies.com)
  Produkte: SCADE, Esterel Studio, zertifizierte C-Compiler

- ETAS (ETASgroup.com)
  Produkte: ASCET, INTECRIO, LABCAR, RTA-OSEK (Betriebssystem)

- GOEPEL electronic (goepel.com)
  Produkte: CANbus-, LINbus-Tools

- Hitex Development Tools (hitex.com)
  Produkte: Tanto2 (Analyse- und Aufnahme-Werkzeug für FlayRay, CAN)

- MathWorks (MathWorks.com)

Produkte: Matlab/Simulink sowie zahlreiche Werkzeuge und Bibliotheken zur Modellierung von Reglern und zur C-Code-Generierung

- preeTEC (preeTEC.com)
  Produkte: TDL:VisualCreator, TDL:VisualDistributor, TDL:Compiler, TDL:Machine (TDL-Laufzeitumgebung)

- samtec (samtec.de)
  Produkt: samDia (Diagnose-Tool-Suite für CAN, LIN etc.)

- Softing (softing.com)
  Produkt: Diagnostic Tool Set (DTS)

- SymtaVision (symtavision.com)
  Produkt: SymTA/S (Analyse des Echtzeitverhaltens von verteilten Systemen)

- TTTech (TTTech.com) und TTAutomotive (TTAutomotive.com)
  Produkte: TTP-Tools, TTP/X-Plan, TTP/X-Build, TTP-OS, LIN-Plan

- Vector Informatik (vector-informatik.de)
  Produkte: CANoe (Simulation und Test von CAN-/LIN-/MOST-/FlexRay-Systemen), CANdb++, DaVinci Tool Suite (zum Entwickeln der Task-Funktionen)

- Warwick Control Technologies (warwickcontrol.com)
  Produkte: X-Analyser (CAN, LIN), CANopen DeviceMonitor, FlexRay-Simulation

Aufgrund der Fülle von Hardware-Anbietern verzichten wir auf die Auflistung von Firmen und Produkten in diesem Bereich. Gute Quellen sind hier zum Beispiel Ausstellerverzeichnisse von Industriemessen, wo sowohl Software- als auch Hardware-Anbieter ausstellen, wie zum Beispiel bei der *embedded world* in Nürnberg (embedded-world.de) oder am *FlexRay-Product Day* (www.hanser.de/seminare).

Zusammenfassung

Wir erinnern uns an das Ziel dieser Lektion, nämlich ein Grundverständnis über die verteilten Systeme im Automobil zu vermitteln, damit Sie noch bessere, technisch fundierte Entscheidungen bei Software-Entwicklungsprojekten und bei der Auswahl von Werkzeugen treffen können. Dazu haben wir als Grundlage die heute im Automobil vorhandenen, zum Teil heute bereits standardisierten verteilten Systeme überblicksmäßig beschrieben: CAN, byteflight, LIN, FlexRay und MOST. Zum soliden Verständnis der Unterschiede war es notwendig, (a) die Unterschiede zwischen Ereignissteuerung und Zeitsteuerung zu verstehen, (b) diverse Probleme bei Echtzeit-Software wie zum Beispiel die unbeabsichtigte Invertierung von Prioritäten kennenzulernen und (c) über gewünschte künftige Eigenschaften von Software im Automobil wie Determinismus und Kompositionalität zu erfahren. CAN stellt die erste Generation von verteilten Systemen im Automobil dar und wird fast in jedem modernen Fahrzeug verwendet. CAN unterstützt nur Ereignissteuerung. Da Knoten gleichzeitig senden können, sind Kollisionen beim Nachrichtenaustausch möglich. Es ist bei CAN nicht vorhersehbar, wie lange das Übertragen einer Nachricht dauert. Das byteflight-Protokoll behebt den zuletzt genannten Nachteil von CAN für eine bestimmte Anzahl von Nachrichten mit höchster Priorität. LIN ist als kostengünstiges Subsystem für CAN definiert worden. FlexRay verbindet Zeitsteuerung und Ereignissteuerung. Für die Ereignissteuerung wurde das byteflight-Protokoll übernommen. Die Zeitsteuerung wird durch das sogenannte *Time Division Multiple Access* (TDMA) Verfahren realisiert, bei dem periodisch eine Tabelle abgearbeitet wird, in der festgeschrieben ist, wann exakt welcher Knoten sendet. MOST ist für Multimedia-Anwendungen konzipiert.

Da FlexRay künftig im Automobil verstärkt zum Einsatz kommen wird und voraussichtlich CAN schrittweise ablösen dürfte, haben wir im zweiten Teil der Lektion zwei Ansätze vorgestellt, wie Software für den zeitgesteuerten Teil von FlexRay entwickelt werden kann. Im ersten Fall, der bisher üblich ist, werden die Plattform und Topologie des FlexRay-Systems zuerst festgelegt. Die Software wird dann darauf zugeschneidert. Das bedeutet eine oft mühsame Entwicklung und Probleme bei der Wartung, wenn die Plattform oder Topologie geändert wird. Der zukunftsweisende alternative Ansatz beruht darauf, die Software Plattform- und Topologie-neutral zu modellieren und erst dann den Code für ein spezifisches Flexray-System zu generieren.

Modellbasierte Software-Entwicklung für verteilte Systeme im Automobil setzt geeignete Abstraktionen voraus. Da bei den Anwendungen oft Echtzeitanforderungen zu erfüllen sind, hat sich gezeigt, dass eine abstrakte logische Spezifikation des Zeitverhaltens zielführend ist. Das wird in der *Timing Definition Language* (TDL) durch die *Logical Execution Time* (LET) Abstraktion ermöglicht. Die Beispiele und Fallstudien zur modellbasierten Entwicklung haben daher die TDL und damit die LET-Abstraktion verwendet.

Ein kurzer Überblick über die zur Zeit angebotenen Werkzeuge zur Entwicklung von Software für verteilte Systeme im Automobil hat die Lektion abgerundet.

Literaturverzeichnis

[Berry und Gonthier, 1992]
Berry, G und Gonthier, G: *The Esterel Synchronous Programming Language: Design, Semantics, Implementation*, Science of Computer Programming, 1991

[DeComSys, 2007]
Web-Site der Firma DeComSys: DeComSys.com

[dSpace, 2007]
Web-Site der Firma dSpace: dSpace.com

[Farcas et al., 2005]
Farcas, E, Farcas, C, Pree, W, und Templ J: *Transparent Distribution of Real-Time Components Based on Logical Execution Time*, ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05), Chicago, Illinois, 15.-17. Juni, 2005

[Halbwachs et al., 1991]
Halbwachs, N, Caspi, P, Raymond, P, und Pilaud, D: *The synchronous dataflow programming language LUSTRE*, Proceedings of the IEEE, 1991

[Henzinger et al., 2003]
Henzinger, T, Kirsch, C, Sanvido, M, Pree, W: *From Control Models to Real-Time Code Using Giotto*, IEEE Control Systems Magazine 23(1), 2003

[Mühlhäuser, 2006]
Mühlhäuser, M: Verteilte Systeme, Kapitel 9 im *Informatik Handbuch* (Editoren: P. Rechenberg, G. Pomberger), Hanser Verlag, 4. Auflage, 2006

[preeTEC, 2007]
Web-Site der Firma preeTEC: preeTEC.com

[Templ, 2007]
Templ, J: *Timing Definition Language 1.2 – Specification*, Februar 2007
Den TDL-Sprach-Report finden Sie auf der Web-Site der Firma preeTEC, preeTEC.com, unter *Technology*, Unterpunkt *Further Documents* (Abfrage Mai 2007)

[TTTech, 2007]
Web-Site der Firma TTTech: TTTech.com
Einen Vergleich verschiedener Protokolle (*Protocol Comparisons*) finden Sie unter dem Punkt *Technology*, Unterpunkt *Articles* (Abfrage Mai 2007)