Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AUTOSAR	Automotive Open Systems Architecture
ASAM	Association for Standardization of Automation and Measuring Systems
CAN	Controller Area Network
CCP	CAN Calibration Protocol
CRC	Cyclic Redundancy Check
CSMA	Carrier Sense Multiple Access
DoIP	Diagnostic over Internet Protocol
ECU	Electronic Control Unit
EOL	End of Line Programmierung
FlexRay	Zeitsynchrones Kfz-Bussystem
GW	Gateway
I/O	Sammelbegriff für Ein-Ausgabesignale bei Steuergeräten sowie deren softwareseitige Verarbeitung.
KWP	Keyword Protocol
	Meist als KWP 2000 verbreitetes Diagnoseprotokoll
Layer	Protokollschicht
LIN	Local Interconnect Network
	Vorbereitetes Bussystem für einfache Anwendungen.
MOST	Media Oriented System Transport
NRZ	Non Return to Zero
	Codierung für Datenbits auf den Busleitungen
OBD	On-Board Diagnosis
OSI	Open System Interconnection
	Schichtenmodell für Datennetze und Protokolle
PDU	Protocol Data Unit

PWM	Pulse Width Modulation
	Pulsbreitenmodulation
TCP/IP	Transmission Control Protocol / Internet Protocol
TÜV	Technischer Überwachungsverein
TTCAN	Time Triggered CAN
	Zeitsynchrone Variante des CAN-Busses
TTP	Time Triggered Protocol
	Zeitsynchron arbeitendes Bussystem
TP	Transport Protocol
	Verfahren zur Aufteilung von Datenblöcken auf mehrere Botschaften.
WLAN	Wireless Local Area Network
	Funk-Netzwerk mit kurzer Reichweite.
XCP	Universal Measurement and Calibration Protocol

Begriffsverzeichnis

Domäne	Eine Domäne ist eine Ansammlung von Computer-Konten.
Powertrain	Antriebsstrang
Remote Login	Ermöglicht einem Computer-Benutzer, sich auf einem entfernten Computer über ein Netzwerk anzumelden, um diesen zu verwenden, ohne am Ort des Gerätes sein zu müssen.

Kurzfassung

Diese Arbeit beschäftigt sich mit Steuergeräten und ihren Anwendungen im Kraftfahrzeug. Dabei werden wir uns mit vielen bekannten Kfz-Bussystemen wie CAN, LIN und FlexRay befassen und diese ausführlich untersuchen. Das Hauptziel dieser Arbeit ist der Entwurf eines Gateway, ein zentrales Steuergerät, das die Kommunikation zwischen mehreren Bussystemen regelt und für die Flusssteuerung zwischen allen Sendern und Empfängern sorgt. Unser Entwurf befasst sich nicht mit der Umwandlung von Busprotokollen wie z.B. CAN in FlexRay und umgekehrt. Er soll dafür sorgen, dass die Kommunikation zwischen mehreren Bussystemen effizient geregelt wird. Das Gateway kann die verschiedenen Arten von Busprotokollen empfangen und sie an das richtige Zielbussystem weitersenden.

Abstract

This work is about control devices and their applications in vehicles. We will discuss many well-known automotive networks such as CAN, LIN and FlexRay, and examine them in detail.The main objective of this work is the design of a gateway, a central control unit that controls the communication between multiple bus systems, and provides flow control between all transmitters and receivers. The conversion of bus protocols such as CAN to FlexRay, and vice versa aren't considered in our design. It aims to ensure that communications areregulated efficiently between multiple bus systems. The gateway can receive various types ofbus protocols, and send them to the right destination bus.

1 Einführung

Elektroniksysteme im Auto sind heutzutage das Maß aller Dinge. Nicht nur, weil ungefähr ein Drittel der Produktionskosten eines Fahrzeugs auf die Elektronik entfällt, sondern auch weil sie das Autofahren angenehmer und einfacher machen, und vor allem auch für die Sicherheit während des Autofahrens sorgen.

1.1 Bussysteme Entwicklung und Geschichte

Da die Diagnose solcher Systeme immer komplexer wird, ist ihre Entwicklung eine der größten Herausforderungen im Bereich der Automobilindustrie. Laut Statistik, siehe Abb. 1.1 und Abb.1.2, erwartet man für Länder mit hohem Ausstattungsgrad in der Automobilelektronik, dass das Wachstum für das nächste Jahr zwischen 30% und 50% liegen wird, und Länder mit weniger Ausstattung teilweise über 100%. Gründe für dieses Wachstum sind Kostendruck, z. B. einsparen von Rohstoffen, sowie der Wettbewerbsdruck. Und vor allem der Wunsch, dass Autos immer besser an den Kunden angepasst werden. Deswegen sucht man immer nach besserem Komfort und neuen Sicherheiten. Dabei ist zu beachten, dass diese Entwicklung durch gesetzgeberische Randbedingungen eingeschränkt wird.

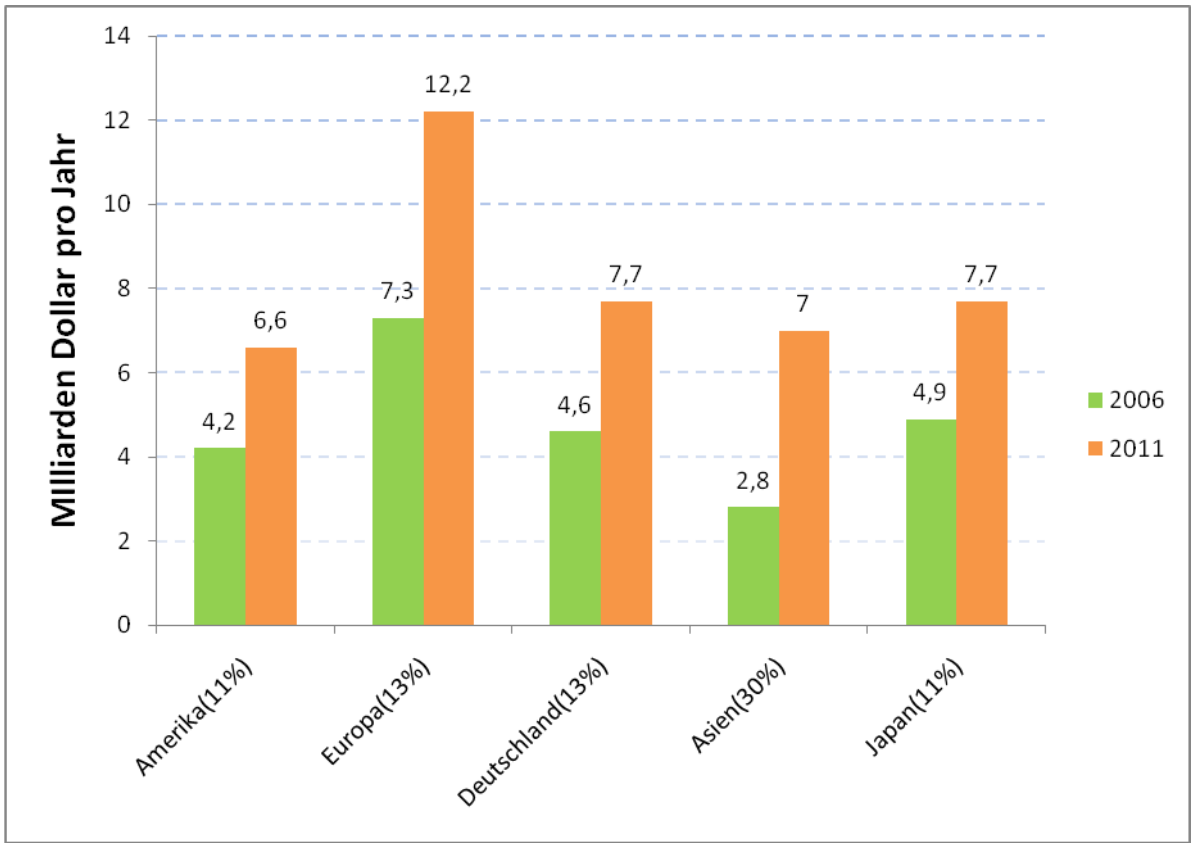


Abbildung 1.1: Marktvolumen und Wachstum in der Automobilelektronik [Emot11]

Die Vernetzung zwischen den einzelnen Steuergeräten spielt in dieser Entwicklung eine immer größere Rolle, weshalb man nach immer neuen Lösungen sucht, die den Vernetzungsgrad im Elektroniksystem eines modernen Fahrzeugs verbessern. Das wird hauptsächlich durch neue Software ermöglicht, die mehrere Funktionen über mehrere Steuergeräte verteilten kann.

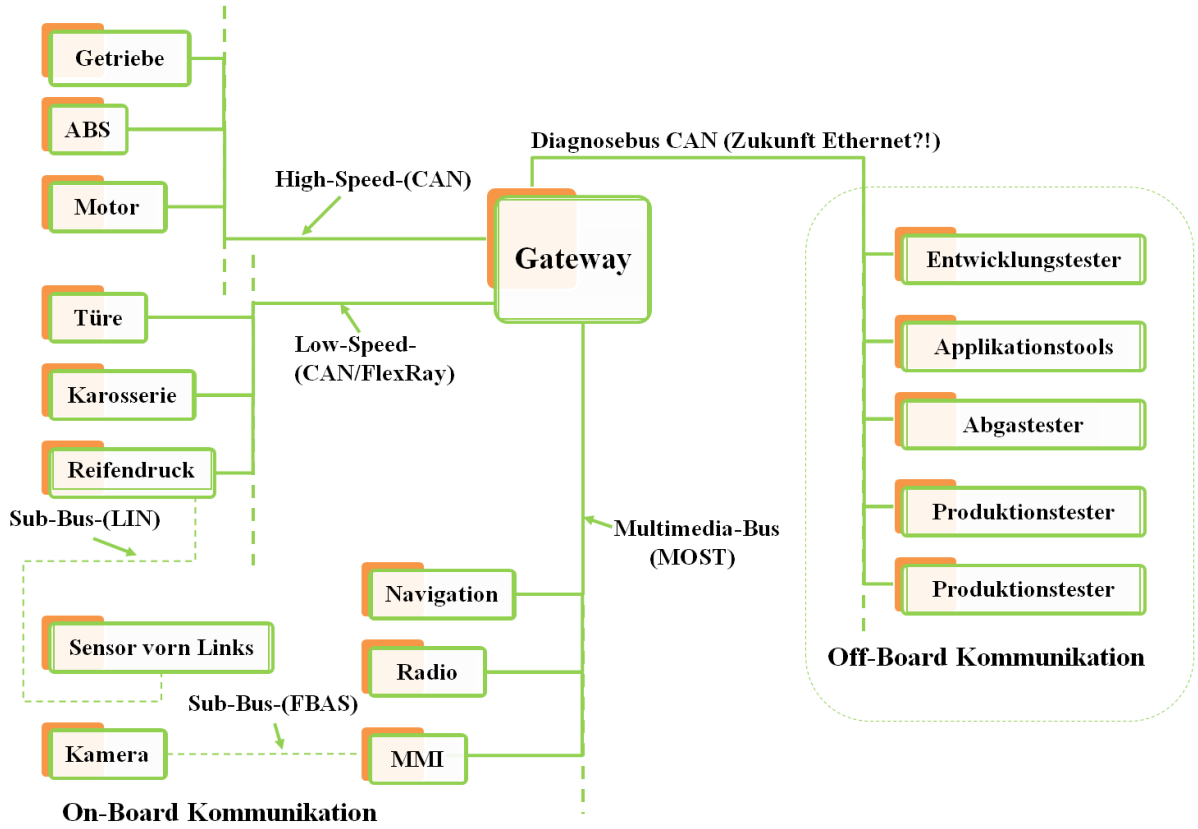


Abbildung 1.2: Kommunikation im Fahrzeug

1.1.1 On-Board Kommunikation

Ein modernes Fahrzeug hat viele Steuergeräte, die das Auto zum Fahren bringt. Solche Steuergeräte kann man in verschiedenen Kategorien unterteilen, zum einem die Grundelektronik, wie Motorsteuerung, ABS, ESP usw.

Licht-, Klima- und Türsteuerung fallen unter die Kategorie Komfort. Weiterhin hat man im modernen Fahrzeug Fahrerassistenzsysteme, z.B. Einparkhilfe, Abstandregelung und Spurwechsel. Für Radio, Navigation und Autotelefon gibt es Infotainment Systeme.

Alle diese Kategorien sind durch verschiedene Bussysteme miteinander verbunden, da verschiedene Busse für unterschiedliche Aufgaben geeignet sind. Zum einem ist es aus Sicherheitsgründen besser, wenn man die verschiedene Funktionalitäten voneinander trennt, und zum anderen ist ein einziges Bussystem für alle Kategorien viel zu teuer.

In einem modernen Fahrzeug findet man über 80 verschiedene Steuergeräte, die insgesamt mehr als 1000 verschiedene Funktionen haben, und dafür über 100 Megabyte Speicher benötigen.

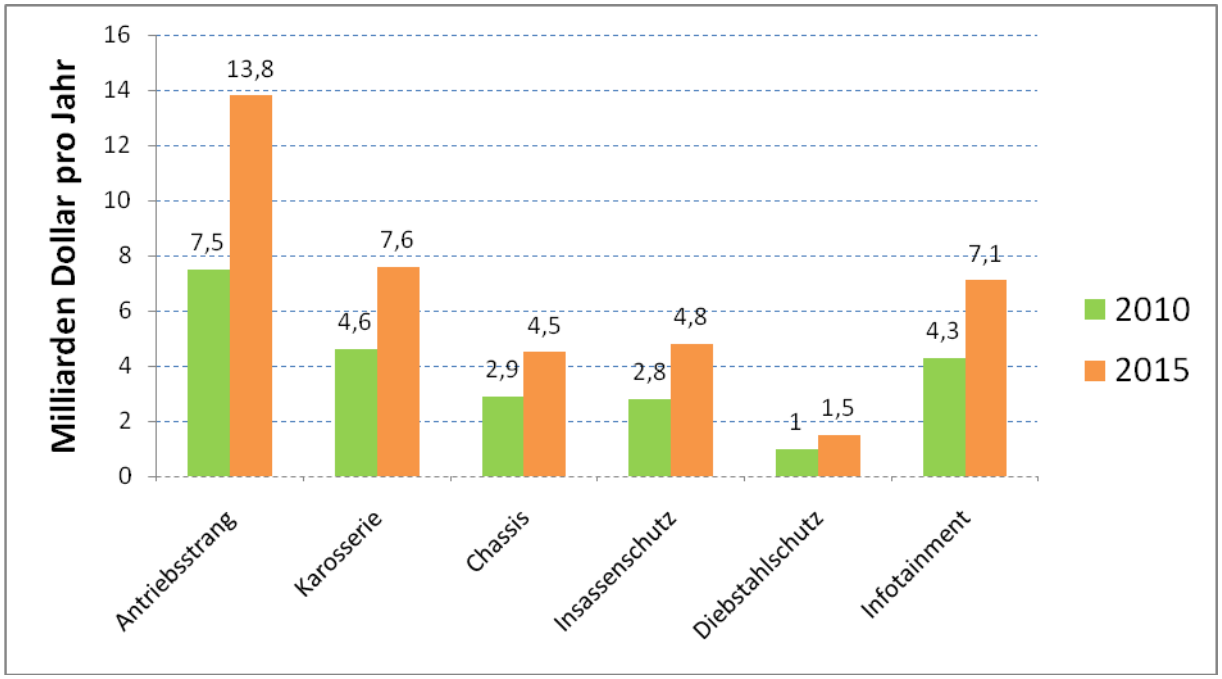


Abbildung 1.3: Mikroelektronikverbrauch für Kfz nach Anwendungssegmenten[Emot11]

Für das gesamte Netz von Steuergeräten werden ungefähr 3 km Leitungslänge, d.h. hoher Vernetzungsgrad für mehr Datenaustausch aber weniger Bussysteme benötigt. Aktuell kommen K-Line, CAN, LIN, MOST und FlexRay zur Anwendung, Tendenz sinkend. Die Anzahl der Busse und Sub-Busse im Fahrzeug liegen heutzutage ungefähr bei 20, können aber durch leistungsfähigere Bussysteme reduziert werden. Genauso wird man für die Zukunft versuchen, durch höhere Integration mit weniger Steuergeräten auszukommen. Die Busbelastung liegt und wird auch in der Zukunft im Grenzbereich liegen.

- Bis zu 80 Steuergeräte, Tendenz sinkend
- Mehr als 1000 Funktionen, Tendenz steigend
- 100 MB Speicherbedarf, 13% des Produktionswertes, Tendenz steigend
- Hoher Vernetzungsgrad zwischen den Steuergeräten
- Bis zu 3 km Leitungslänge
- Heute bis zu 5 verschiedene Bussysteme, Tendenz sinkend
- Insgesamt bis zu 20 Busse und Sub-Busse

- Busbelastung teilweise im Grenzbereich

Wie komplex solche Systeme sind, merkt man, wenn ein Fehler auftritt oder wenn etwas nicht funktioniert, siehe Abbildung 1.4. Ein Automobilhersteller berichtet beispielsweise folgendes [Emot11]: „Wenn bei einem seiner Fahrzeuge ein Pin-Anschluss am Gateway-Stecker ausfällt, dann wird dies von vielen Steuergeräten erkannt, denn alle Steuergeräte überwachen die Buskommunikation. Es werden dann für nur eine Ursache insgesamt 38 Fehlermeldungen in 5 Steuergeräten generiert.“ Eine richtige Lösung für solches Problem ist in der Werkstatt unter Umständen eine große Überforderung.

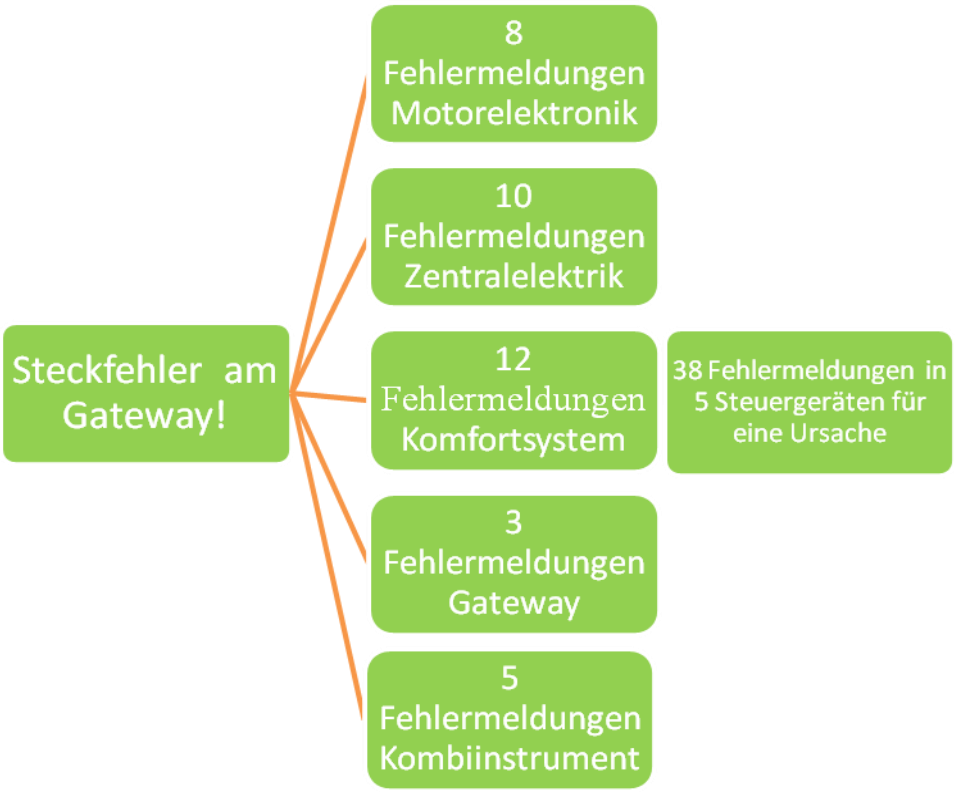


Abbildung 1.4: Fehlerbericht

1.1.2 Off-Board Kommunikation

Auf der anderen Seite bezeichnet die Off-Board Kommunikation, die Kommunikation der internen einzelnen Systeme im Fahrzeug mit einem externen Bussystem außerhalb des Fahrzeugs. Auch als Tester genannt. Solche Tester werden z.B. in der Werkstatt sowie in der Produktion, Applikation und bei der Entwicklung oder auch beim TÜV verwendet.

Die Grundstruktur, die in der Abbildung 1.2 zu sehen ist, hat sich heutzutage in den meisten Fahrzeugen durchgesetzt:
Mehrere Bussysteme im Fahrzeug sind über ein zentrales Steuergerät (Gateway) miteinander gekoppelt, das zu jeder Zeit an den Diagnosebus angeschlossen werden kann.

1.2 ISO/OSI Schichtenmodell

Abbildung 1.5: ISO/OSI Schichtenmodell

OSI ist das sogenannte Open System Interconnection (Offenes System für Kommunikationsverbindungen). Es wurde von der ISO 1978 als Grundlage für die Bildung von Kommunikationsstandards entworfen und standardisiert. Es wird oft als ISO/OSI-Schichtenmodell oder auch ISO/OSI-Referenzmodell genannt.

1.2.1 Physical-Layer und Data-Link-Layer

Mit den Schichten eins (Physical-Layer) und zwei (Data-Link-Layer) wird das eigentliche Bussystem beschrieben. Der Physical-Layer beschreibt wie die Signale kodiert sind, wie sie auf die Busleitung versendet werden und welchen Signalpegel die verschiedenen Signale haben etc. Die Zusammensetzung von Botschaften aus einzelnen Bits und die Erkennung von Übertragungsfehler usw. ist die Aufgabe des Data-Link-Layer.

1.2.2 Transport-Layer und Applikation-Layer

Wie gelangen die Botschaften vom Sender zum Empfänger?

Damit beschäftigen sich die Schichten drei bis sieben. In der Automobiltechnik spielen die Segmentierung und die Desegmentierung von Botschaften eine große Rolle. Dabei geht es um die Übertragung von größeren Datenblöcken, obwohl die meisten Bussysteme nur Botschaften mit kleinen Datenmengen übertragen können. Für diese Aufgabe ist Schicht vier (Transport-Layer) aus dem ISO/OSI-Schichtenmodell zuständig.

Die Anwendungsschicht (Schicht sieben) stellt für die eigentliche Anwendung wie beispielsweise Diagnosetester fertige Dienste Sie ermöglicht die Anwendung Zugriff auf das Netz z.B. für E-Mail, Remote Login etc.

Alle sieben Schichten sind aufeinander aufbauend, siehe Abbildung 1.5. Jede Schicht benötigt die Details der darunter liegenden Schichten. Schicht eins benötigt nur Busleitungen und Stickverbindungen, das sind die sichtbaren Teile des Bussystems. Sie werden manchmal auch als Schicht null bezeichnet.

2 Bussysteme

In diesem Abschnitt geht es um die wichtigsten Bussysteme (LIN, CAN, FlexRay und MOST), die man heutzutage im modernen Fahrzeug findet. Zuerst wollen wir aber mit allgemeinen Informationen, die die Unterschiede zwischen diesen Bussen verdeutlichen, anfangen, und danach werden wir den einzelnen Busse detailliert anschauen.

2.1 Allgemein

Wie bereits in der Einführung erwähnt wurde, definieren die Schichten 1 und 2 (Physical- und Data-Link-Layer) aus dem ISO/OSI- Schichtenmodell, also die Ebenen des allgemeinen Kommunikationsmodells, das eigentlichen Bussystem, siehe Abbildung 1.5. Dabei geht es um Bits-Kodierungen auf den Bussen und die Zusammensetzung von Bits zu Botschaften, welche Signalleitungen und Pegel verwendet werden, Fehlererkennung und Korrigieren von Fehlern. Dort beschäftigt man sich auch mit Adressierung von Botschaften und mit der Frage, wer zuerst auf den Kommunikationsbus zugreifen oder kommunizieren darf oder anders gesagt wie der Versand von Botschaften geregelt wird.

Die Elektronikarchitektur eines modernen Fahrzeugs besteht aus einem zentralen Gateway, das mit verschiedenen Domänen verbunden ist. Jeder Domäne besteht aus einem einzigen Bussystem. Alle Bussysteme sind über ein zentrales Steuergerät (Gateway) mit einander gekoppelt. Die verschiedene Bussysteme sind für bestimme Anwendungsbereiche geeignet, siehe Abbildung 1.2.

- Für Antriebsstrang und Powertrain also der Fahrwerkelektronik kommt normalerweise CAN mit einer Hochgeschwindigkeitsvariante genannt High-Speed-CAN oder FlexRay zum Einsatz.
- Für den Anwendungsbereich Komfort- und Karosserieelektronik vertraut man eine CAN-Variante, die niedrigere Busgeschwindigkeit hat. Hier kommt manchmal auch LIN zum Einsatz aber nur als Ergänzung d.h. Sub-Bussystem unter Low-Speed-CAN.
- Für den Unterhaltungsbereit (Infotainment) hat man heutzutage meist in Oberklassefahrzeugen MOST im Einsatz. Bei manchen Mittelklassefahrzeugen kommt auch bis heute noch CAN im Einsatz.
- Für Diagnose verwendet man meistens CAN aber zunehmend auch Ethernet-Anschluss (DoIP = Diagnostic over Internet Protocol), damit man das Installieren von Softwaren innerhalb der Fertigung und in der Werkstatt beschleunigen kann. Davor wurden in dem Diagnose-Bereich alte Bussysteme wie K-Line bzw. J1850 in den USA verwendet.

2.2 Entwicklung Bussysteme im Fahrzeug

Die nächste Grafik soll die Entwicklung der Bussysteme in Fahrzeugen anschaulich machen, siehe Abb. 2.1. Die horizontale Achse zeigt, in welchem Jahr das jeweilige Bussystem

erschienen ist und auf der Vertikalen sieht man, welche Übertragungsrate es hat. Je höher es steht, für umso anspruchsvollere Aufgaben kann es eingesetzt werden. Mit der Übertragungsrate steigen jedoch auch die Preise an, die durch die Größe der Kreise verdeutlicht werden sollen. Das LIN-System gehört zu den günstigeren Bussystemen, kann aber dafür auch nur für weniger komplexe Aufgaben eingesetzt werden. Wohingegen MOST zu den teuersten aber gleichzeitig auch zu den leistungsstärksten zählt.

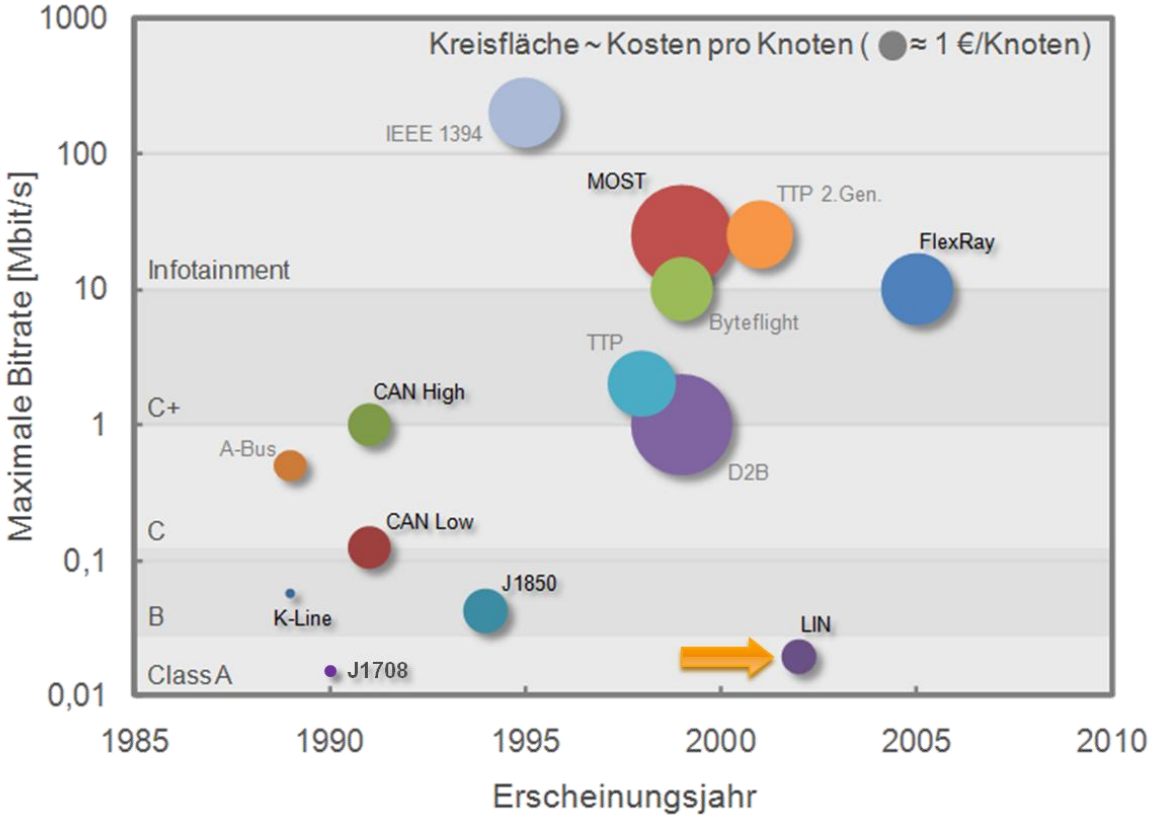


Abbildung 2.1: Entwicklung Bussysteme [Emot11]

2.3 Leistungen und Anforderungen

2.3.1 Aufgabenbereiche

Die folgende Tabelle soll uns die verschiedenen Aufgaben, Einsatzbereiche und Zwecke aller Bussysteme darstellen.

Tabelle 2.1: Aufgabenbereiche [Emot11]

Aufgabe	Kommunikation Wo?	Zweck	Bussystem, Protokoll
On-Board Kommunikation zwischen den Steuergeräten im Fahrzeug:			
Fahren	Steuergerät zu Steuergerät	Steuer- und Regelaufgaben, Hohe Echtzeit- und Sicherheitsanforderung	CAN, LIN, FlexRay
Unterhalten	Steuergerät zu Fahrgast	Sehr hohe Datenraten, keine Sicherheitsanforderung	MOST
Off-Board Kommunikation zwischen den Fahrzeug und externen Geräten:			
Testen	Werkstatt TÜV, Dekra	Fehlerspeicher lesen, Parameter ändern, Stellglieder ansteuern, Flash-Programmierung	CAN, Diagnoseprotokolle: UDS, KWP 2000
Fertigen	Fahrzeughersteller (OEM), Steuergerätehersteller		
Applizieren	Entwicklung	Ungeschützter schneller Zugriff auf alle Steuergeräte Interna	CAN, Kalibrierprotokolle: CCP, XCP

2.3.2 Klassifizierung nach Bitrate

Bussysteme in Fahrzeugen werden durch ihre Bitraten klassifiziert. Je höher die Klasse des Bussystems ist, kann diese in umso anspruchsvolleren Anwendungsgebieten eingesetzt werden. LIN gehört zu den einfachen Class-A Bussystemen, welche eine Bitrate unter 25 kbit/s aufweisen, und wird in der Karosserieelektronik eingesetzt. Für etwas komplexere Aufgaben wird Low-Speed-CAN eingesetzt, ein Vertreter der Class-B Bussysteme, die bis zu 125 kbit/s übertragen können.

Das Class-C Bussystem hingegen, zu welchem auch der High-Speed-CAN dazugehört, kann bis zu 1000 kbit/s übertragen und wird im Bereich der Fahrwerke und auch für Diagnosezwecke eingesetzt.

Für Übertragungsraten jenseits 1 Mbit/s werden FlexRay und TTP eingesetzt. Diese gehören zu den Class-C+ Bussystemen und werden für Lenkung und Bremsen eingesetzt.

Für Aufgaben im Multimediabereich wird MOST eingesetzt, das zur höchsten Klasse der Bussysteme zählt, nämlich zur Class-D, welche über 10 Mbit/s übertragen kann.

Tabelle 2.2: Klassifizierung der verschiedenen Bussysteme[ZiSc08]

Klasse	Bitrate	Bussystem	Bereich
Diagnose	< 10 kbit/s	ISO 9141(K-Line)	Werkstatt und Abgastester
A	< 25 kbit/s	LIN	Karosserieelektronik
B	25-125 kbit/s	Low-Speed-CAN	Karosserieelektronik
C	125-1000 kbit/s	High-Speed-CAN	Antriebstrang, Fahrwerk, Diagnose
C+	>1 mbit/s	FlexRay, TTP	Steer an Brake by Wire
D	>10 mbit/s	MOST	Multimedia

2.3.3 Anforderungen

Welche Anforderungen, die die verschiedenen Bussysteme erfüllen müssen bezüglich ihrer Botschaftslänge, Botschaftsrate, Sicherheit und Kosten, zeigt uns Tabelle 2.3.

Tabelle 2.3: Anforderungen [ZiSc08]

Anwendung	Botschaftslänge	Botschaftsrate	Sicherheit	Kosten	Lösung
On-Board Kommunikation- zwischen den Steuergeräten im Fahrzeug:					
Very-Low-Speed z.B. Fensterheber	Kurz	niedrig	niedrig	sehr niedrig	Class A Bus LIN (20 Kbit/s)
Low-Speed z.B. Klimaanlage	Kurz	mittel	mäßig	niedrig	Class B Bus Low-Speed-CAN (125 Kbit/s)
High-Speed z.B. Motorsteuerung	Kurz	hoch	sehr hoch	mittel	Class B Bus High-Speed-CAN (500 Kbit/s)
Very-High-Speed z.B. Fahrwerksteuerung	Kurz	sehr hoch	extrem hoch	mittel	Class C+ Bus FlexRay (10 Mbit/s)
Multimedia	Mittel	extrem hoch	niedrig	hoch	Class D Bus MOST (150 Mbit/s)
Off-Board Kommunikation- zwischen den Fahrzeug und externen Geräten:					
Applikation(MC) Entwicklung	Kurz	mittel bis hoch	gering	unwichtig	CAN

EOL Programmierung Fertigung (Flashen)	lang bis sehr lang	mittel	mäßig	unwichtig	K-Line (alt) J1850 (alt) Zukünftig Ethernet
Werkstattdiagnose After Sales (OBD)	Kurz	niedrig	gering	niedrig	

2.4 Protokollstapel eines Bussystem

Zum Senden und Empfangen von Nachrichten besitzen Steuergeräte von Fahrzeugen Protokollstapel. Die Anwendungsschicht kennzeichnen die Nachrichten, damit klar ist, ob sie z.B. Information über die Drehzahl des Motors oder Zündungsdaten enthalten. Zusätzlich können sie auch eine Prüfsumme anhängen, die zur Fehlerkorrektur eingesetzt wird. Schließlich wird die fertige Nachricht (PDU - Protocol Data Unit) dem Transport-Layer weitergeleitet. Falls sie für einen Frame zu lang ist, wird diese in mehreren kleinen Nachrichten unterteilt und jeweils mit einem Header ausstattet, der Auskunft darüber gibt, welcher Teil einer größeren Nachricht vorliegt. Damit ist ein späteres Zusammenfügen wieder möglich. Anschließend werden die Nachrichten an den Data Link Layer weitergeleitet.

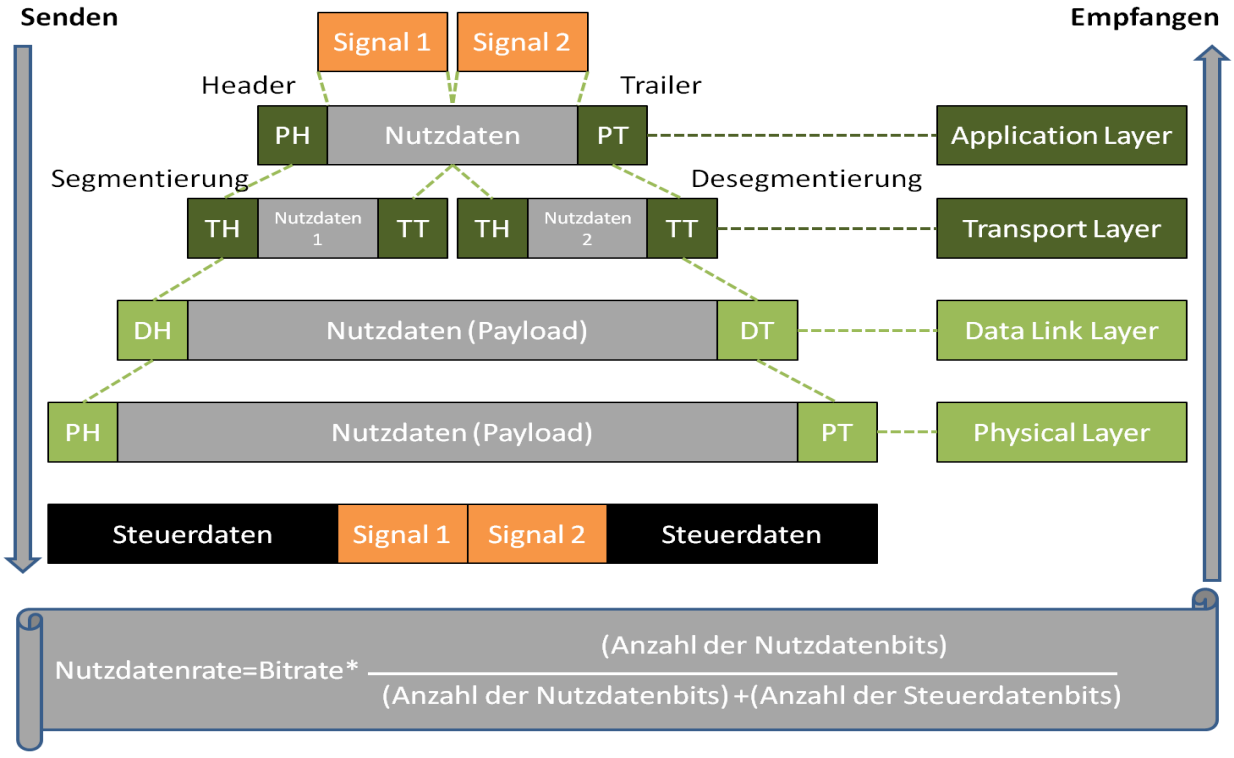


Abbildung 2.2: Protokollstapel

Der Data-Link-Layer erweitert die Nachrichten mit einem Header und Trailer, um z.B. Steuerbits und Adressinformationen an den Physical-Layer senden zu können, siehe Abb. 2.2.

Im Kommunikationscontroller werden die Nachrichten vom Data-Link-Layer serialisiert und Bits werden kodiert und eventuell noch ergänzt.

Nun können die eigentlichen Daten in Form einer Nachricht mit Steuerinformationen auf dem Bus gesendet werden. Dabei können bei CAN und FlexRay die eigentlichen Daten nur etwa 50% der verfügbaren Bitrate benutzen. Der Rest der Nachricht wird für die hinzugefügten Zusatzinformationen benutzt.

2.5 Elektrotechnische Grundlagen Bussysteme

2.5.1 Verbindungstypen und Topologien

Die Übertragung bei Kraftfahrzeug-Bussystemen erfolgt in der Regel über Halb-Duplex-Verbindungen, sodass die Kommunikation in beide Richtungen möglich ist, jedoch immer nur in eine Richtung gleichzeitig erfolgt. Wenn also A sendet, kann B nur empfangen und wenn B sendet, kann A nur empfangen. Senden und empfangen zur gleichen Zeit ist bei einer Halb-Duplex-Verbindung also nicht möglich. Hierfür wäre eine Voll-Duplex-Verbindung nötig, die aber nicht eingesetzt wird, da sie zu teuer wäre, , siehe Abb. 2.3.

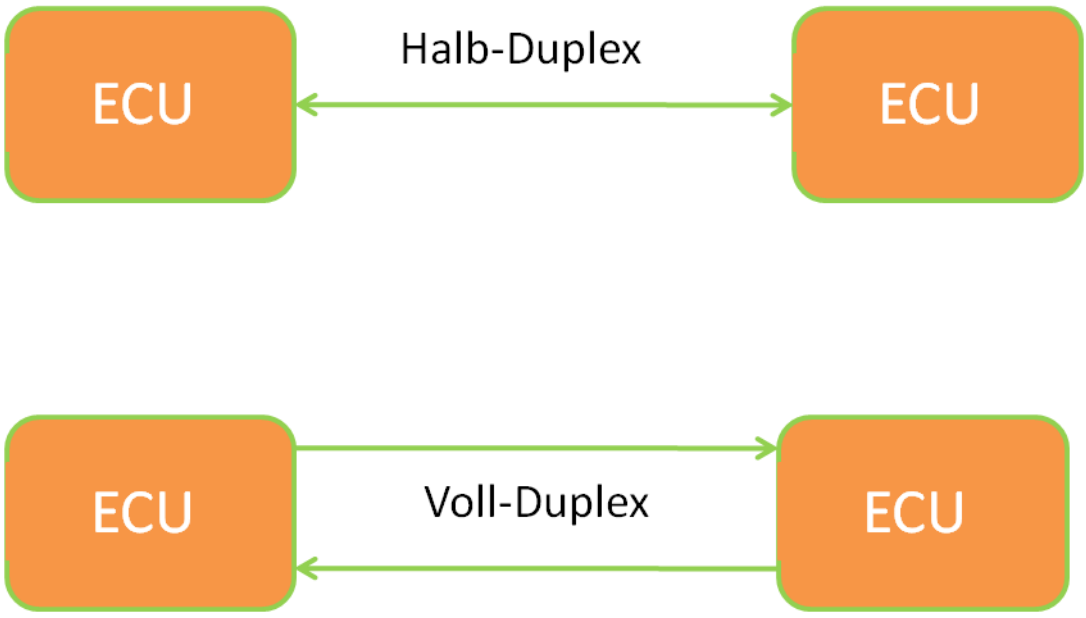


Abbildung 2.3: Halb-, Voll-Duplex

Die Leitungen bei Class-A Bussystemen bestehen meist aus Ein-Draht-Verbindungen. Bei Bussystemen mit höherer Übertragungsrate verwendet man hingegen üblicherweise Twisted-Pair-Leitungen, die die Signale über zwei miteinander verdrillte Drähte übertragen.

Daten werden vom Sender über die Busleitung gesendet, sodass diese von allen Empfängern bei Bedarf empfangen werden können. Diese Form der Übertragung wird Broadcast genannt.

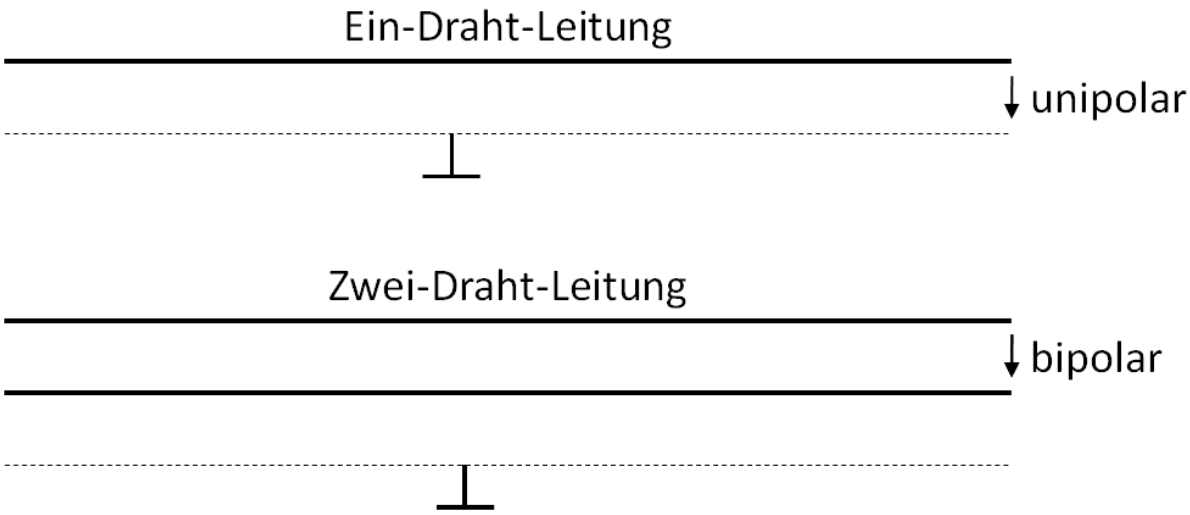


Abbildung 2.4: Busleitung

Die Gestaltung von Kabelbäumen in Fahrzeugen hängt stark von der Topologie der Bussysteme ab, siehe Abb. 2.5. Hierbei wird zwischen Linien-, Stern- und ringförmigen Bussystemen unterschieden.

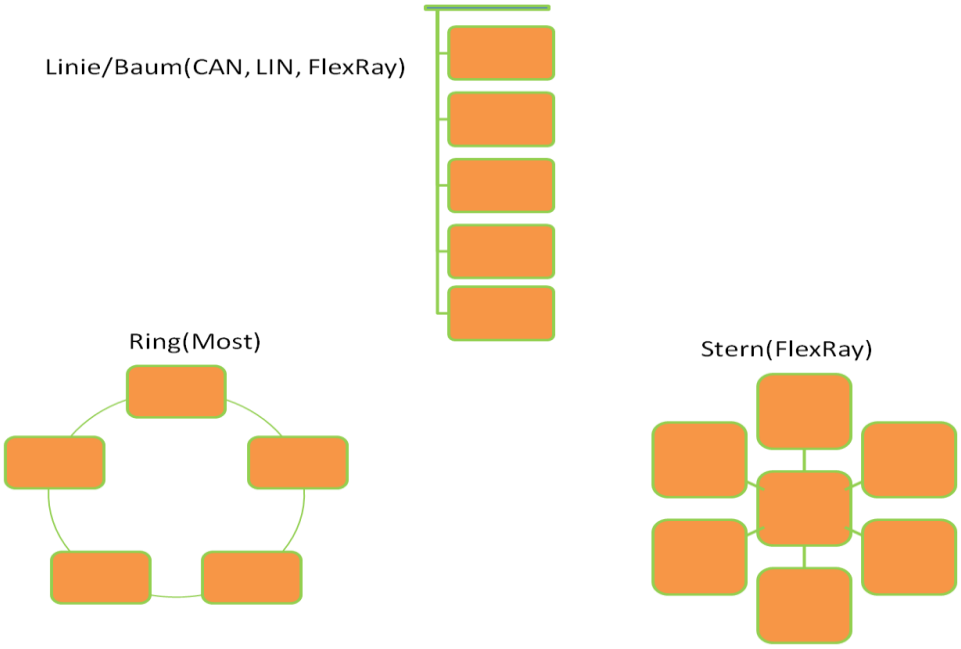


Abbildung 2.5: Verschiedene Bus-Topologie (Linie, Stern und Ring)

Bei linienförmigen sind die Steuergeräte über Stichleitungen an die Hauptleitung angeschlossen, die möglichst kurz gehalten werden, um hohe Bitraten zu erreichen. Zu diesen zählen z.B. LIN und CAN.

Sternbussysteme wie FlexRay besitzen einen zentralen Sternpunkt, der eine Verbindung zu jedem Steuergerät hat.

Bei Bussystemen mit einer Ringform sind die Steuergeräte als Teil eines geschlossenen Kreises miteinander verbunden. MOST verwendet auch dieses System, jedoch sind die Steuergeräte optisch statt wie üblich elektrisch miteinander verbunden.

2.5.2 Kopplung der Steuergeräte

Ein Bus-Controller ist für das Bus-Management zuständig. Dieser sitzt meist im Mikrocontroller von Steuergeräten und koppelt diese an das Bussystem. Um längere Signalleitungen zu ermöglichen setzt man oft Repeater in Sternpunkten ein, die das Signal verstärken.

Zur Kommunikation von verschiedenen Bussystemen, die nicht die gleichen Protokolle oder Bitraten verwenden, werden Steuergeräte eingesetzt, die die von einem Bussystem empfangenen Daten in eine für das andere Bussystem kompatible Form bringen. Solche Steuergeräte werden Gateways genannt.

2.5.3 Bitkodierung und Leitungslänge

Die NRZ-Kodierung (Non-Return-to-Zero) ist die übelicherweise verwendete Bitkodierung von Signalen.

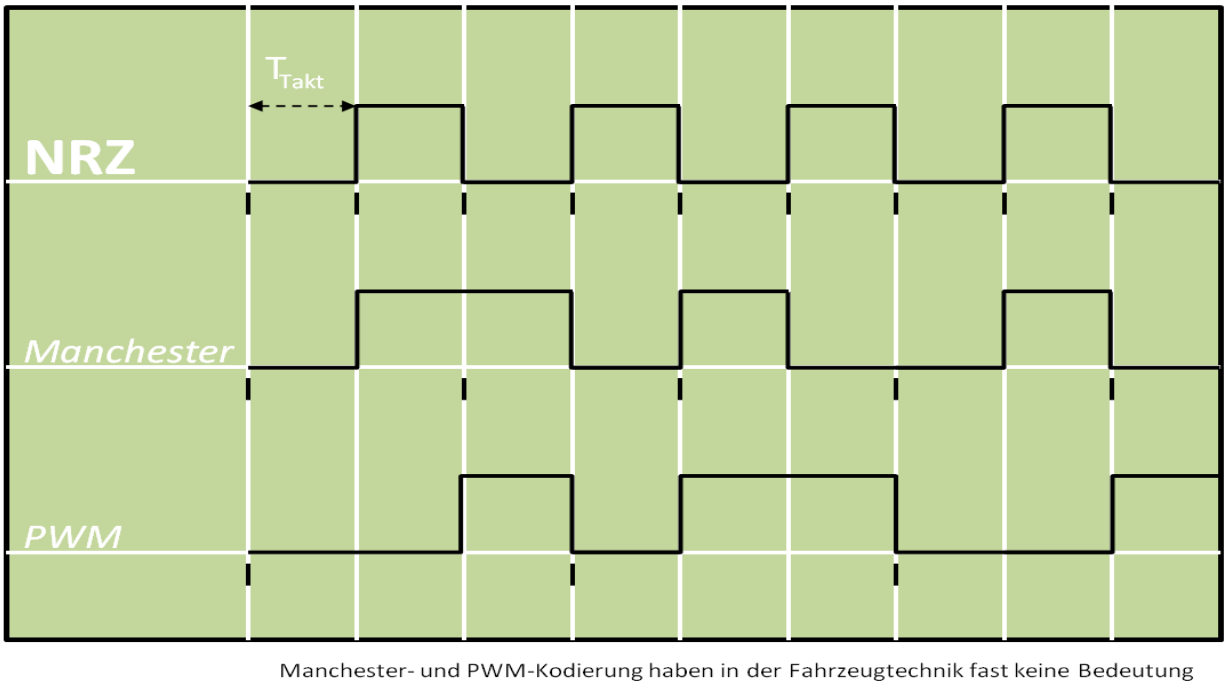


Abbildung 2.6: Bitkodierung

Hier ist der Pegel des Signals während eines Bittaktes entweder auf 0 oder auf positiver Spannung, siehe Abb. 2.6. Neben NRZ gibt es auch noch die Manchester-Codierung, die bei Fahrzeugen noch nicht benutzt wird und PWM, bei der die Pulsbreite die Kodierung festlegt.

Für die Sicherheit bei der Übertragung spielen die Wellenlänge und die Bitrate der Signale eine wichtige Rolle. Probleme treten auf, wenn die Leitungslänge etwa einem Zehntel der Signalwellenlänge entspricht.

Bei 1 Mbit/s und einer Leitung mit einer Länge größer als 30m würden Beeinträchtigungen der Übertragungsqualität entstehen. Um dem entgegenzuwirken, müssten entsprechende Widerstände an die Leitung angebracht werden.

2.5.4 Datenübertragung

Mit Zeichen- und bitstrombasierten Übertragungen werden Bits zu Nachrichten zusammengesetzt.

Die zeichenbasierte Übertragung, die auch bei K-Line und LIN eingesetzt wird, setzt ein Zeichen meist aus 8 Bits zusammen mit einem vorangehenden Startbit und einem Stoppbit am Ende zur Synchronisation zwischen Sender und Empfänger. Zwischen den Zeichen und Nachrichten gibt es Pausen, die die nutzbare Bitrate des Bussystems beeinträchtigen.

Bussysteme mit höheren Bitraten versuchen Pausen möglichst kurz zu halten, indem die einzelnen Bits einer Nachricht ohne Pausen dazwischen gesendet werden und führen die Übertragung bitstrombasiert durch. Pausen werden somit nur noch zwischen den einzelnen Nachrichten eingesetzt.

Um Nachrichten auch an die richtigen Empfänger zu leiten, werden zwei unterschiedliche Verfahren eingesetzt. Bei der gerätebasierten Adressierung, die auch im Diagnosebereich eingesetzt wird, welche im Layer 7 des OSI Modells sitzt, besitzt jedes Steuergerät eine eindeutige Adresse. Im Gegensatz dazu haben die Steuergeräte in der inhaltsbasierten Adressierung, die im Data-Link-Layer (Layer 2) sitzt, keine eindeutige Adresse. Stattdessen haben die Nachrichten eine Kennung, die auch Message-Identifier genannt wird, und Informationen über den Inhalt. Alle Steuergeräte, die sich für diese Kennung interessieren, können die Nachricht empfangen.

Beim Versand und Empfang von Daten trägt zur Laufzeit neben der Dauer und Anzahl der Übertragung auch die Bereitstellung der Nachrichten, die auf der Anwendungsebene bereitgestellt werden, eine wichtige Rolle. Diese wird erst Schritt für Schritt erzeugt und abgeschickt, sobald das Bussystem frei ist. Nach dem Empfang wird die Nachricht entpackt und umgerechnet, sodass die Inhalte auf der Anwendungsebene verfügbar sind. Je länger die Übertragung dauert, desto höher ist die Latenz, deren Schwankung auch als Jitter bezeichnet wird und sehr wichtig für Steuer und Regelaufgaben ist.

2.5.5 Buszugriffsverfahren

Bei Bussystemen können zwar mehrere Steuergeräte gleichzeitig Nachrichten empfangen, jedoch ist das Senden dieser immer nur von einem Steuergerät gleichzeitig möglich. Die Reihenfolge, welches Steuergerät jeweils senden darf, wird mit Hilfe eines Buszugriffsverfahrens festgelegt, das neben der Bitrate oft das wichtigste Entscheidungskriterium für das entsprechende Bussystem darstellt.

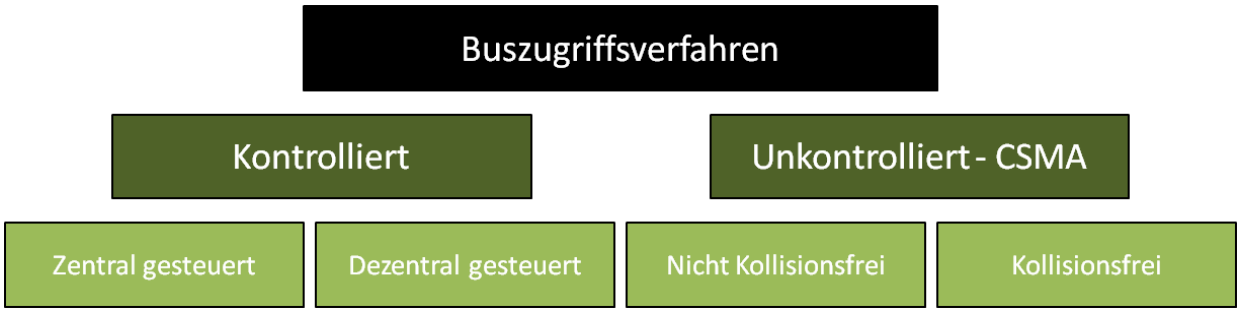


Abbildung 2.7: Buszugriffverfahren

Man unterscheidet bei den Buszugriffsverfahren zwischen den kontrollierten und unkontrollierten Verfahren. Die ersteren werden nochmals in zentral und dezentral gesteuerte Verfahren unterteilt. LIN ist ein zentralgesteuertes System, bei dem ein Master-Steuergerät den Slave-Steuergeräten Sendeberechtigungen vergibt. Ohne diese ist ein Senden nicht möglich.

FlexRay und TTCAN gehören zu den dezentral gesteuerten Systemen. Hier kommt ein Zeitschlitzverfahren zum Einsatz, bei welchem die Steuergeräte synchron auf einer Zeitbasis arbeiten. Ihnen wird ein Zeitfenster zugewiesen, in denen das Senden möglich ist.

Bei den unkontrollierten Verfahren der Buszugriffe unterscheidet man zwischen kollisionsfreien und nicht kollisionsfreien. Beide Verfahren setzten auf das Carrier-Sense-Multiple-Access Verfahren, das jedem Steuergerät das Senden erlaubt, jedoch erst wenn das Bussystem frei ist. Somit geschieht das Senden asynchron oder wird durch Ereignisse gesteuert. Jedoch können Kollisionen auftreten, wenn zwei Steuergeräte gleichzeitig mit dem Senden anfangen.

Um mit einer Kollision umzugehen, kann eine Kollisionserkennung oder zusätzlich auch eine Kollisionsauflösung eingesetzt werden. Bei der Kollisionserkennung wird die Busleitung überwacht und gesendete und empfangene Daten miteinander verglichen. Bei Kollisionen wird nach einer Pause mit zufällig gewählter Länge, welche die Chance auf eine erneute Kollision verringert, versucht erneut zu senden. Bei diesem Verfahren ist es jedoch ungewiss, ob eine Kollision ansteht oder nicht, daher setzt man zusätzlich auf eine Kollisionsauflösung, die auch von CAN eingesetzt wird. Hier setzt das Steuergerät, welches die höhere Priorität besitzt, das Senden fort, sollte es zu einer Kollision kommen. Dieses Verfahren wird auch Arbitrierung genannt.

Tabelle 2.4: Buszugriffsverfahren Vor- und Nachteile

Buszugriffsverfahren	Vorteile	Nachteile
Master-Slave	1) Einfache Realisierbarkeit. 2) Sichergestellte maximale Zeit, deterministisch	1) Maximale Latenzzeit proportional zur Anzahl der Busteilnehmer 2) Ausfall des Masters Ausfall des Gesamtsystems 3) Redundanz bei zyklischer Übertragung
Time-Division-Multiple-Access - TDMA	1) Hohe zeitliche Genauigkeit 2) Hohe Protokolleffizienz 3) Streng deterministisch	1) Zeitliche Synchronisierung der Teilnehmer notwendig 2) Begrenzte Anzahl von Teilnehmern 3) Begrenzte Anzahl von Nachrichten 4) Übertragung redundanter Daten
CSMA/CD (Collision Detect):	1) Sehr viele Teilnehmer möglich 2) Niedrige Buslast 3) Teilnehmer kann ohne Bus-Rekonfiguration hinzugefügt oder entfernt werden	1) Nicht deterministisch 2) Lange Wartezeiten bei Hochlast
CSMA/CA (Collision Avoidance):	1) Viele Teilnehmer möglich 2) Teilnehmer kann ohne Bus-Rekonfiguration hinzugefügt oder entfernt werden 3) Kaum Effizienzeinbruch bei Hochlast 4) Für hochpriore Botschaften deterministisch	1) Maximale Latenzzeiten hochpriorer Nachrichten

2.5.6 Busanschluss – Wired-OR

Im folgenden Abschnitt wird an einem Schaltungsbeispiel gezeigt, wie sich Systeme mit Kollisionserkennung und Kollisionsauflösung verhalten Quelle [Emot11].

Die Abbildung 2.8 zeigt den Busanschluss eines LIN Bussystems, welches die Steuergeräte ECU1 und ECU2 besitzt. Man sieht hier rechts oben den Busanschluss eines LIN Bussystems mit zwei Steuergeräten ECU1 und ECU2. Die Versorgungsspannung +UB ist mit den Transistoren T1 und T2 über einen Pull-Up-Widerstand verbunden.

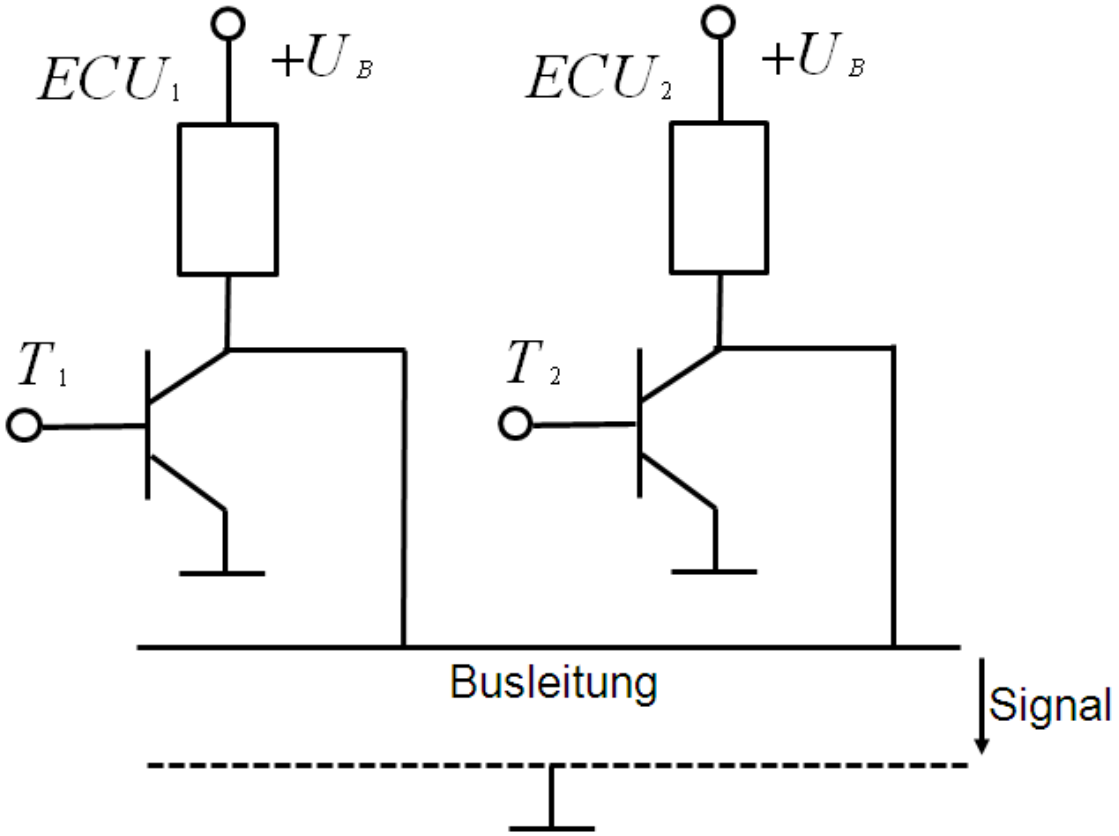


Abbildung 2.8: Wired-OR

Die Busleitung ist im Ruhezustand auf dem Pegel der Versorgungsspannung, während die Transistoren T1 und T2 ausgeschaltet sind. Wenn das Steuergerät 2 in Ruhe ist, und das Steuergerät 1 eine logische 0 sendet, wird T1, welcher niederohmiger als die Pull-Up-Wiederstände ist, von Steuergerät 1 eingeschaltet und die Busleitung wird auf Masse gezogen.

Tabelle 2.5: Zustände bei der Kollisionsauflösung

Zustand	T1	T2	Signal	Bemerkung
Ruhezustand	Aus	Aus	U_B	
Senden	Aus (High)	Aus (High)	U_B	Wie Ruhezustand
Senden	Ein(Low)	Aus (High)	0	Dominantes Signal Low (T1) gewinnt
Senden	Aus (High)	Ein(Low)	0	Dominantes Signal Low (T2) gewinnt

Sollte jedoch von Steuergerät 1 eine logische 1 gesendet werden, während Steuergerät 2 immer noch in Ruhe ist, so schaltet Steuergerät 1 T1 ab, sodass die Busleitung weiterhin auf der Versorgungsspannung liegt, also liegt ein positives Signal an.

Wenn nun aber Steuergerät 1 eine logische 1 und Steuergerät 2 eine logische 0 senden will, schaltet Steuergerät 2 T2 ein und zieht die Busleitung somit auf Masse, was einer logischen 0 gleichkommt. Das Signal wird also von Steuergerät 2 dominiert. Beim Mitlesen und Vergleichen der Signale auf der Busleitung von beiden Steuergeräten gäbe es für Steuergerät 2 keine Probleme. Steuergerät 1 jedoch würde eine Kollision feststellen, da eine 0 gelesen wird, aber eigentlich eine 1 erwartet wurde. Somit wartet Steuergerät 1 und Steuergerät 2 sendet weiter, als gäbe es keine Kollision.

2.6 CAN

Das erste Class-C und demnach das erste in Kraftfahrzeugen verwendete Hochgeschwindigkeits-Bussystem, ist das von der Firma Bosch entwickelte Controller Area Network. Eingeführt wurde es 1991 mit der Mercedes S-Klasse, die zu jener Zeit neu war. Das Bussystem wird heute standartmäßig in Kraftfahrzeugen eingesetzt. Entworfen wurde das Bussystem zunächst nur für den Antriebsstrang, inzwischen wird es nahezu allen Bereichen des Fahrzeuges eingesetzt.

Die durch die Firma Bosch entwickelte Spezifikation wurde bis heute kaum verändert und bildet die Basis für das heutige Bussystem. 1994 erfolgte ihre Übernahme in die offizielle Standardisierung als ISO 11898.

Der Data-Link-Layer kommt in zwei Ausführungen vor: der sogenannte Message-Identifier (CAN-ID), der für die Adressierung und Arbitrierung benötigt wird, existiert in 11 und 29 Bit Länge. Es besteht die Möglichkeit des parallelen Betriebs, d.h. der Koexistenz beider Varianten auf dem gleichen Bussystem.

Der Physical-Layer kommt in den drei nicht miteinander kompatiblen Ausführungen High-Speed-CAN, Low-Speed-CAN und Single-Wire-CAN vor. Die letztere wurde von General Motors betrieben, in den USA standardisiert und besitzt nur eine Datenleitung. Darüber hinaus existieren Versionen für Nutzfahrzeuge (SAE J1939) und für die Industrieautomatisierung (CANopen, DeviceNet).

CAN selbst erfährt eine Standardisierung auf der Layer-1 und Layer-2-Ebene des ISO/OSI-Schichtenmodells. Obwohl die übergeordneten Ebenen anfangs nicht standardisiert wurden, wird inzwischen der Bereich der Diagnose mit der ISO 14229 und ISO 15765 auf Anwendungen standardisiert.

CAN macht von einem bitstrombasierten Übertragungsverfahren mit einer bidirektionalen Zweidrahtleitung Gebrauch. Kurze Stichleitungen verbinden die Steuergeräte mit diesem Linienbus (siehe Abbildung 2.9), wobei die Bitrate bestimmt, wie lang eine Stichleitung sein darf. Bei 1 Mbit/s ist eine Stichleitung-Länge von bis zu 30 Zentimeter zulässig. Der High-Speed-CAN wird in der Regel nur mit 500 kbit/s betrieben, obwohl er Bitraten bis zu 1 Mbit/s

unterstützt. Ein Abschlusswiderstand sollte an den beiden Busenden angebracht werden. Bitrate und Buslänge sind invers, d.h. je geringer die Bitrate, umso länger die Buslänge, wobei die Buslänge bei 1 Mbit/s höchstens 40 bis 50 Meter entsprechen darf.

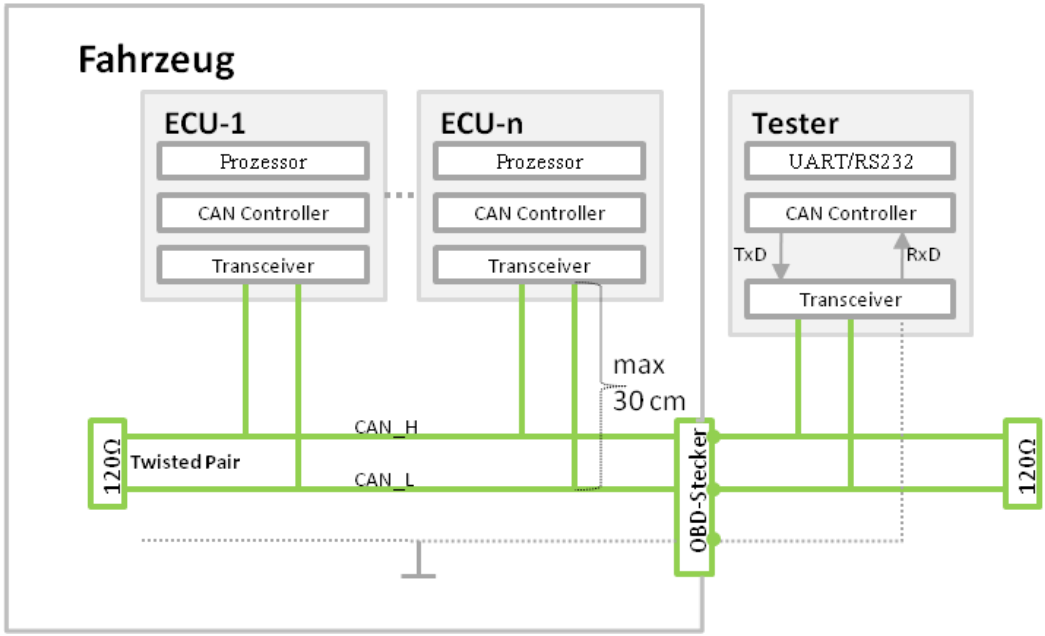


Abbildung 2.9: High-Speed-CAN-Bus

Die Einführung des Low-Speed-CAN erfolgte zum einen aufgrund seiner einfacheren Implementierung durch die fehlende Notwendigkeit an Abschlusswiderständen und die Zulassung längerer Stichleitungen, zum anderen aufgrund der Fehlertoleranz des Low-Speed-CAN in eingeschränktem Maß, z.B. werden von diesem Bussystem Unterbrechungen einer einzelnen Leitung oder auch Kurzschlüsse zugelassen.

Den Anfang eines CAN-Frame, bei welchem die Null das dominante Signal darstellt, bildet ein Null-Bit. Diesem Start-Of-Frame Bit folgt ein CAN-Identifier, der 11 oder 29 Bit lang ist. Dieser Identifier ist für die eindeutige Kennzeichnung des Inhalts der Nachricht zuständig und wird zugleich für die Arbitrierung angewandt. Das Bit mit dem höchsten Wert wird beim CAN-Identifier als erstes gesetzt. Somit gewinnt bei einer möglichen Kollision auf dem Bus immer die Botschaft mit der geringsten CAN-ID bzw. mit den meisten führenden Nullen in der CAN-ID.

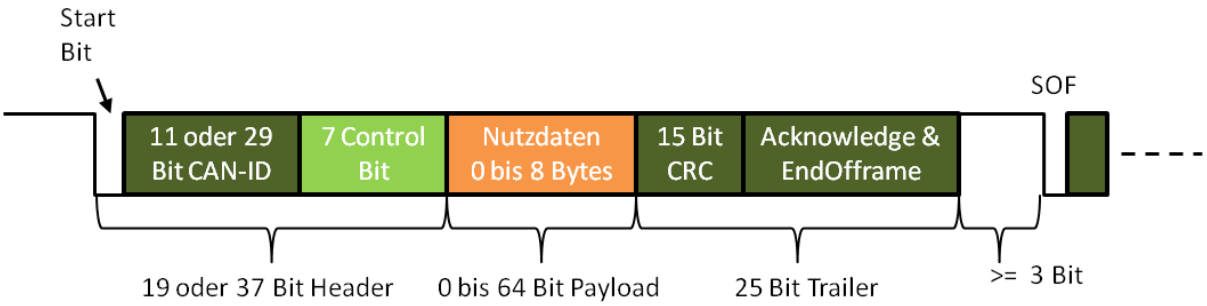


Abbildung 2.10: CAN-Botschaftsformat

In den dem ID-Identifier folgenden sieben Steuerbits ist die Datenlänge des folgenden Nutzdatenfelds enthalten. Diese Nutzdaten können eine Länge von 0 bis 8 Byte haben. An diese Nutzdaten schließt sich ein 25 Bit langer Trailer an, der eine CRC-Prüfsumme von 15 Bit Länge, einen Acknowledge-Bit, welcher den korrekten Empfang der Botschaft bestätigt, sowie einige Steuerbits enthält.

2.7 LIN

LIN steht für Local Interconnect Network. LIN Version 1.0 wurde von den Firmen Daimler-Chrysler, Audi, BMW, Motorola, Volvo, Volcano und Volkswagen 1999 vorgelegt. Inzwischen gibt es LIN in der Version 2.1

Die Anforderungen an LIN sind im Vergleich zu CAN geringer aber auch kostengünstiger. Im Einzelnen soll LIN ein einfaches Konzept erhalten, welches flexibel und kostengünstig eingesetzt werden kann. Das einfache Konzept führt zur Robustheit des Protokolls. Die geringen Kosten können durch die geringe Datenrate gewährleistet werden. LIN ist ein offener Standard mit der Interoperabilität ausgestattet, welches z.B. die Kommunikation über ein Gateway mit CAN erlaubt.

Der Einsatzbereich von LIN reicht über Sitze, Türen, Dach, Klimaanlage bis hin zu Steuereinheiten am Lenkrad und Motorsteuerung.

Hier einige Eckdaten von LIN

- Übertragungsrate: 1 - 20 kbit/s
- Teilnehmer: Richtwert: max. 16 Knoten
- Daten/Nachricht: 1-8 Byte (ab Version 2.0 oder höher)
- Leitungslänge: max. 40 Meter
- Betriebsspannung: 8-18V

- UART: Startbit, 8 Datenbits, Stoppbit
- Busmanagement: Master/Slave (Zeitslots, deterministisch)
- Busmedium: 1-Draht-Leitung

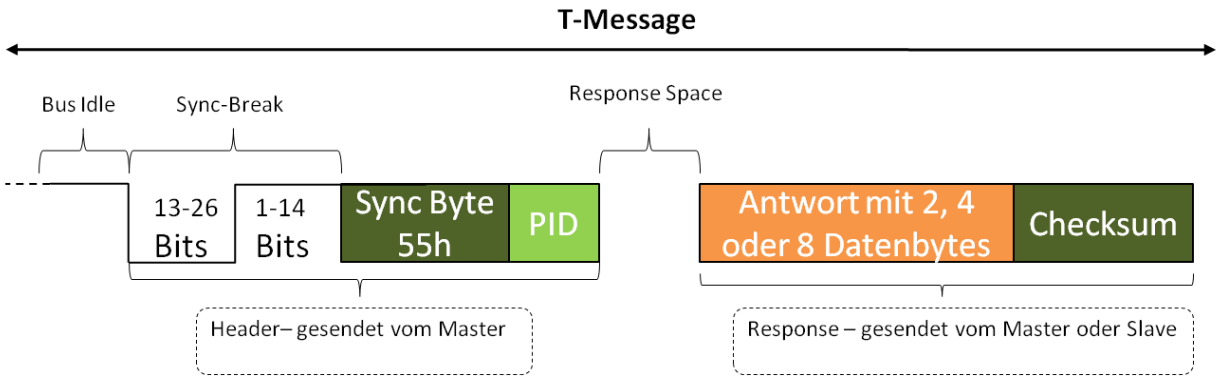


Abbildung 2.11: LIN Botschaftsformat

Die Kommunikation erfolgt streng nach dem Master-/Slave-Prinzip. Der LIN-Master sendet den Header in Form von Master Task. Die LIN-Slaves fügen eine Response hinzu, auch genannt als LIN Slave Task.

Es gibt insgesamt 6 verschiedene Frametypen:

- Standardframe

  Zu erkennen am Identifier 0 bis 59.

  Es gibt einen Sender und mehrere Empfänger.

- Ereignisgesteuertes Frame

  Master sendet über den Standardframe Anfragen an (mehrere) „sendebedürftige" Slaves.

  Kommt es zu einer Kollision, spricht der Master die Slaves vereinzelt erneut an.

- Spontanes Frame

  In der Time Schedule Table wird ein Platz reserviert. Dieser kann wahlweise für unterschiedliche Zwecke genutzt werden.

- Diagnoseframe

  Konfigurieren und Diagnose mit dem Identifier 60 und 61

- Nutzerdefiniertes Frame

  Kunden- und herstellerspezifische mit dem Identifier 62.

- Reserviertes Frame

  reserviert für zukünftige Erweiterungen

Es gibt mehrere Fehlererkennungsarten, wie Bitfehler, Checksum-Fehler oder Identifier-Parity-Fehler. Leider wird die Fehlerbehandlung nicht in der Spezifikation erwähnt, sodass die Behandlung auf Applikationsebene erfolgen muss.

Ein interessanter Aspekt von LIN ist das Power-Management. Mit dem Goto-Sleep-Kommando (Identifier 60, 8 Bytes, 1 Mal 0x00, 7 Mal 0xFF) werden alle Slaves in Schlafmodus versetzt. Mit einem Wake-Up-Signal (dominant: 250µs-5ms, rezessiv: mind.100ms) werden alle Slaves wieder aufgeweckt.

Im Entwurfsprozess von LIN Teilnehmern spielen zwei Dateien eine wichtige Rolle. Node Capacity File beschreibt die Fähigkeit des Slaves, z.B. Protokollversion und maximale Übertragungsrate. Mit der LIN Description File werden die Zeitparameter des Masters eingestellt. Darüber hinaus die Slaves, Nachrichten mit ihren Signalen und die wichtigen Scheduling-Tabellen.

2.8 FlexRay

Anlass für die Entwicklung des FlexRay waren zum einen die geringe Nutzrate des CAN, die für anspruchsvolle Anwendungen im Bereich des Antriebsstranges nicht genug war, zum anderen die künftige X-By-Wire Anwendung wie Break-By-Wire oder Stear-By-Wire, bei welchen neben einer hohen Bitrate auch neue Ansätze für sicherheitskritische Anwendungen gefordert sind. FlexRay hat einen zweikanaligen Ansatz, wodurch größere Anwendungsmöglichkeiten gegeben sind.

2006 wurde FlexRay zum ersten Mal in den BMW X5 eingesetzt. Die in 2005 entwickelte Spezifikation (Version 2.1) blieb außer einigen kleinen Entwicklungen stabil.

Die Übertragung erfolgt beim FlexRay wie beim CAN über Bitstrom, mit einem bidirektionalen Zwei-Draht-Bussystem. Abgeschlossen werden die beiden Enden der Busleitungen mit Abschlusswiderständen. Der zweite Kanal ist nicht obligatorisch, sein Einsatz kann entweder redundant oder zur Verdopplung der Bandbreite erfolgen.

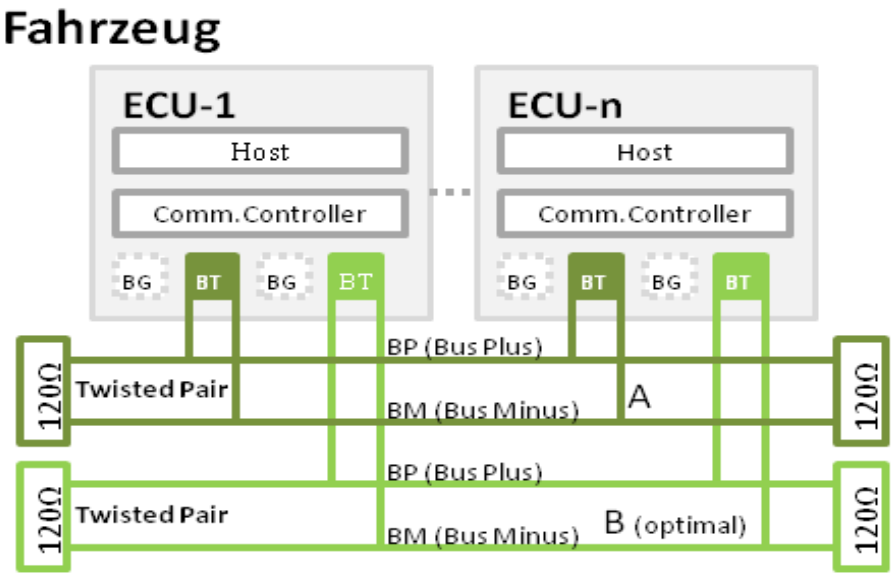


Abbildung 2.12: FlexRay Physical Layer [Wiki11d]

Der Zugriff auf dem Bus erfolgt zeitgesteuert mit einem Zeitscheibenverfahren. Die maximale Anzahl an Steuergeräten, die an das Bussystem angeschlossen werden können, beträgt 64, wobei der Anschluss in verschiedenen Topologien erfolgen kann. Im Gegensatz zum CAN mit seiner Linienstruktur wird beim FlexRay bevorzugt eine Sternstruktur mit einem aktiven Sternpunkt (Active-Star) eingesetzt, welcher als Koppelpunkt zur Regenerierung der Signale dient. Dies ermöglicht, dass der Abstand von Steuergerät und Steuerpunkt eine Länge von bis zu 24 Meter erreichen kann. Beim FlexRay ist die Standartbitrate 10 Mbit/s, wobei die Bitrate im Gegensatz zum CAN nicht inhärent beschränkt ist, d.h. zukünftige höhere Bitraten sind möglich.

In FlexRay erfolgt der zeitliche Ablauf der Übertragungen in periodisch wiederkehrenden Zyklen mit konstanter Länge, die ein statisches und dynamisches Segment enthalten.

Das statische Segment besteht aus Zeitschlitzen, deren Länge so festgelegt ist, dass sie genau eine Botschaft enthalten kann. Die Entscheidung, in welchem Zeitschlitz welches Steuergerät senden darf bzw. muss, wird in der Entwicklungsphase getroffen, wobei ein Cycle-Multiplexing möglich ist, d.h. für unterschiedliche Zyklen können für einen bestimmten Zeitschlitz unterschiedliche Steuergeräte zugeordnet werden. Da jedes Steuergerät die Zeitschlitze mitzählt, kennen alle Steuergeräte den jeweiligen Standort des Übertragungssystems.

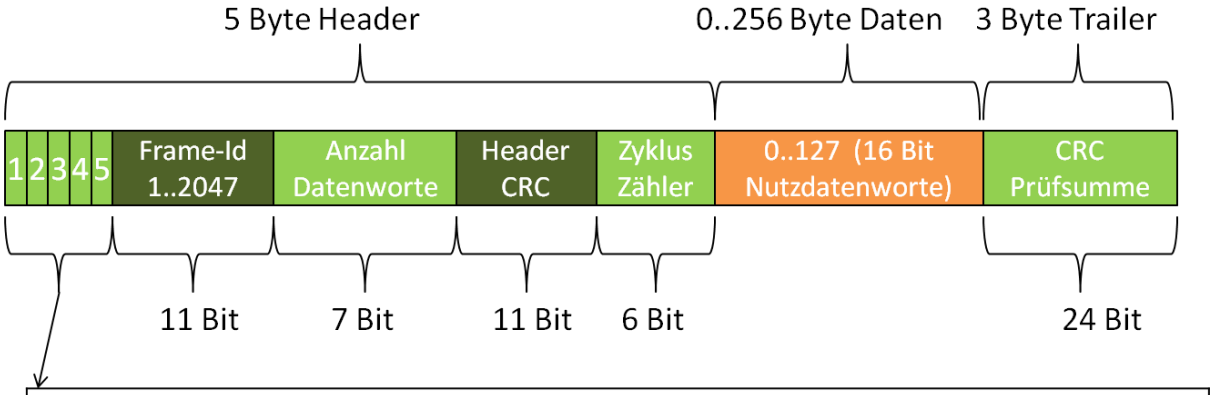
Den in dem dynamischen Segment ebenso enthaltenen Zeitschlitzen, den Minislots, sind ebenfalls je ein Steuergerät zugeordnet, sodass Zusammenstöße verhindert werden. Anders als im statischen Segment ist hier die Länge einer Botschaft variabel. Sie kann länger sein als ein Minislot, ein Steuergerät kann aber auch darauf verzichten zu senden. Im ersten Fall bleibt der Slot-Counter bis zum Abschluss der Botschaft stehen, im zweiten Fall wird der entsprechende

Minislot frei gelassen und der Slot-Counter läuft weiter zum nächsten Minislot, in welchem ein anderes Steuergerät zählen kann. Der Kommunikationscontroller darf eine Botschaft nicht versenden, wenn sie nicht innerhalb des laufenden Zyklus vollständig versendet werden kann.

Den Abschluss eines Kommunikationszyklus bilden die Network-Idle-Time zur Synchronisation zwischen Sender und Empfänger und das Symbol-Window für zukünftige Erweiterungen.

2.8.1 Botschaftsformat

Die nachfolgende Abbildung 2.13 veranschaulicht das Botschaftsformat von FlexRay. Den Anfang einer Übertragung bilden eine Reihe von Steuerbits, an welche sich eine Frame-ID anschließt. Diese Frame-ID, die 2048 verschiedene Werte enthalten kann, entspricht der Nummer des Slots bzw. des Minislots. Anders als bei CAN oder LIN folgt danach die Anzahl der Datenworte, wobei eine Botschaft Null bis 127 Worte enthalten kann, d.h. eine maximale Datengröße von 254 Byte. Die Absicherung des anschließenden Headers erfolgt durch eine Prüfsumme CRC. Die Übertragung des nachfolgenden Zykluszählers dient der Realisierung des Cycle-Multiplexing. Den Abschluss einer Übertragung bildet eine Prüfsumme, die 24 Bit groß ist und sich über die ganze Botschaft erstreckt.



Steuerbits Format:		
Bit 1	Reserved	= 0
Bit 2	Payload Preample Indicator	= Statisches Slot Daten enthalten einen Netzwerkmanagement Vector Dynamisches Slot Daten enthalten 2-Byte-Message-ID
Bit 3	Null Frame Indicator	= Zeigt an, dass die Nutzdaten keine gültigen Daten enthalten(1)oder nicht(0)
Bit 4	Sync Frame Indicator	= Zeigt an, ob der Frame zur synch. Verwendet werden (1) kann oder nicht (0)
Bit 5	Startup Frame Indicator	= Zeigt einen Startup-Frame zur synch. Bei Systemstart an (1)

Abbildung 2.13: FlexRay Format [Emot11]

Die Abbildung 2.13 zeigt das Botschaftsformat auf Ebene des Data-Link-Layers auf physikalischer Ebene kommen einige Bits hinzu. Beispielsweise fügt der

Kommunikationscontroller zu Synchronisationszwecken am Anfang die Trailing-Start-Sequence bzw. die Frame-Start-Sequence hinzu. Bei der Byte-Start-Sequence werden vor jedem Byte 2 Bits gesendet, sodass für jedes übertragene Byte effektiv 10 Bits gesendet werden. Dies führt dazu, dass der Protokoll-Overhead des Gesamtprotokolls höher ist als bei CAN. Bei 10 Mbit/s ergibt sich eine Nutzdatenrate von höchstens 500 kByte/s, wobei diese in der Praxis oft geringer ausfällt, da, um die Botschaft auf jeden Fall in den Zeitschlitz zu bekommen, am seinem Anfang und Ende immer eine Lücke frei gelassen wird.

Die eindeutige Zuordnung der Zeitschlitze an die einzelnen Steuergeräte ist ein wichtiger Faktor, und hat in der Entwicklungsphase des Systems zu erfolgen. Dies und die Schaffung von Platzhalter für künftige Erweiterungen führen zur weiteren erheblichen Senkung der effektiven Nutzdatenrate bei FlexRay, d.h. die Arbitrierung erfolgt in den Entwicklungsabteilungen und nicht im Fahrzeug.

2.9 MOST

MOST steht für Media Oriented Systems Transport und wird vor allem im Bereich Infotainment der Fahrzeugkommunikation eingesetzt. Infotainment ist ein englischer Kunstbegriff aus Information und Entertainment. Beispiele für Infotainment-Geräte sind Radio, CD- und DVD-Player sowie Navigationssystem.

Die Anforderung an MOST ist ein schneller Zugriff, hohe Datenrate und eine relativ hohe Zugriffshäufigkeit. Dagegen haben Infotainment-Geräte keine Echtzeitanforderung, außerdem muss das Protokoll keine Sicherheitsrelevanz erfüllen.

MOST gibt es in drei Ausführungen: MOST 25, MOST 50 und MOST 125. Letztere ist der neueste Standard, welcher im Jahr 2007 von der MOST Kooperation spezifiziert wurde. MOST 25 hat eine Bandbreite von 512 Bits pro Frame, MOST 50 verdoppelt diese Bandbreite und MOST 125 versechsfachte die Bandbreite auf 3072 Bits pro Frame. Namensgebend ist jedoch die Geschwindigkeit 25, 50 und 125 Mbit/s.

Anders als CAN, LIN und FlexRay benutzt MOST ein optisches Übertragungsmedium. Der Kunststoff-Lichtwellenleiter überträgt digitale Daten. Im Vergleich zu elektrischen Systeme ist MOST nicht störempfindlich, jedoch braucht man teure Steckverbindung. Ebenfalls lassen sich Lichtwellenleiter nicht besonders gut biegen. Diese Besonderheit führt zu Einschränkung bei der Kabelverlegung.

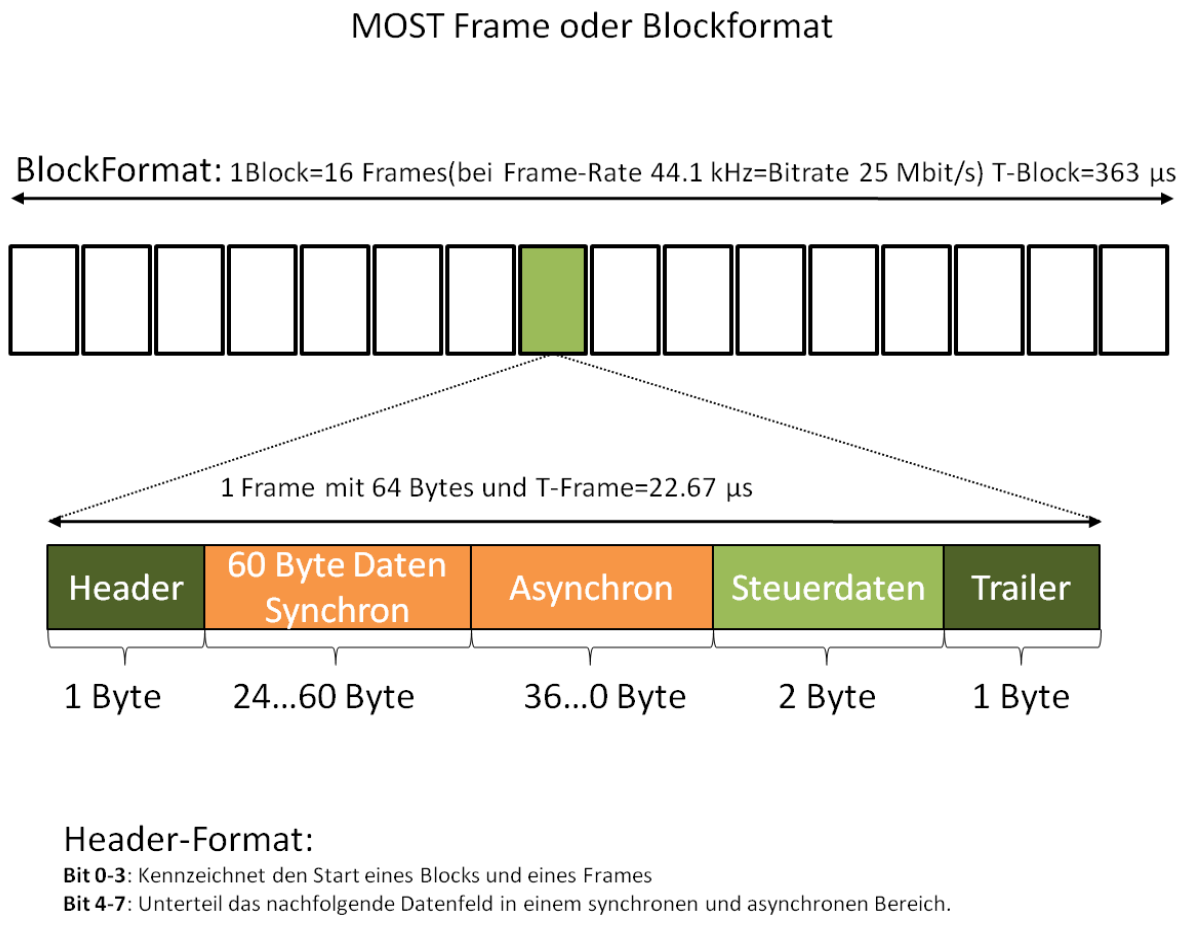


Abbildung 2.14: MOST Frame oder BlockFormat

MOST-Geräte werden in Ringtopologie verschaltet. Es gibt also Punkt-zu-Punkt-Verbindungen zwischen MOST-Geräten. Das Signal wird innerhalb eines Geräts neu erzeugt und in der Topologie weiter versendet. Die Plug & Play Fähigkeit von MOST macht es einfach neue Geräte in die Topologie einzugliedern.

In der Ringtopologie übernimmt ein Gerät die Eigenschaft als Timing-Master. Dieser gibt den Bit Takt an und generiert die Frames. Die übrigen Geräte sind Slave und synchronisieren sich auf den angegebenen Takt. Aufgrund der Master-Eigenschaft erfüllt dieses MOST-Gerät die Plug & Play Funktion nicht und ist ein Single Point of Failure in der Ringtopologie.

Die Daten werden in Frames übertragen, wobei 16 Frames nur ein Block bilden. Bei MOST 25 hat ein Frame 512 Bits oder 64 Bytes. Die 64 Bytes setzen sich aus ein Byte Header, 60 Bytes Daten, zwei Bytes Steuerdaten sowie ein Byte Trailer zusammen (siehe Abbildung 2.14). Die 60 Bytes Daten können variable benutzt werden und teilen sich wiederum in synchrone Daten und asynchrone Daten, wobei die Grenze zwischen synchrone und asynchrone Daten sich verschieben dürfen.

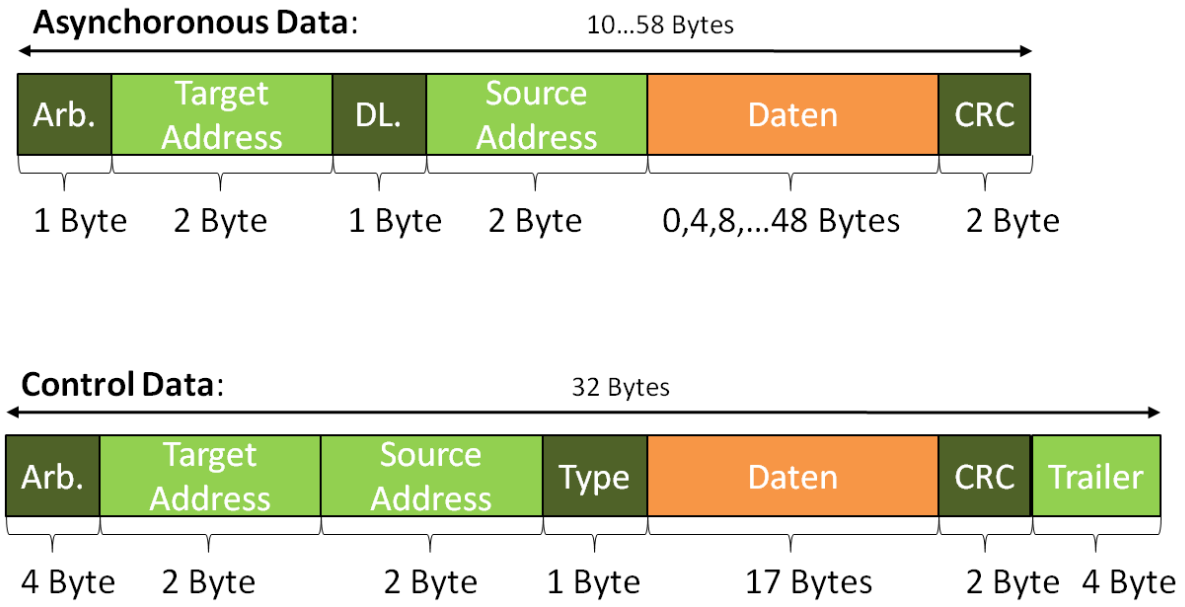


Abbildung 2.15: Asynchrones Data und Control Data

Es gibt insgesamt drei Kanäle, um Daten zu übertragen: synchroner Bereich, asynchroner Bereich und Steuerdaten. Synchrone Daten werden reserviert, diese sind unformatiert und werden benutzt um Audio-Samples zu übertragen. Der asynchroner Bereich und Steuerdaten sind formatiert. Die Größe kann in beiden Fällen variieren und wird auf einer höheren OSI-Schicht segmentiert, sodass die Daten in einem Frame passen.

2.10 Fazit

In diesem Kapitel haben wir verschiedene Arten von Bussystemen und ihren Funktionalität gründlich untersucht und beschrieben, wie z.B. die Nachrichten zwischen den verschiedenen Steuergeräten in einem Bussystem versendet und empfangen werden? Und für welche Anwendungsbereiche die einzeln Bussystemen geeignet sind. Alles ging um die Kommunikation zwischen Sender und Empfänger auf dasselbe Bussystem. Wie die Kommunikation zwischen die verschiedene Bussysteme abläuft, werden wir uns selber Gedanken machen und in dem nächsten Kapitel ausführlich beschreiben und dokumentieren.

3 Entwurf eines Gateway

In diesem Abschnitt geht es um das am Anfang beschriebenes Ziel der Arbeit. Ein Gateway soll von überall mit Daten überflutet werden, die dort empfangen und weiter bearbeitet werden. Was versteht man unter einem Gateway?

3.1 Gateway

Damit die verschiedenen Bussysteme mit ihren unterschiedlichen Protokollen miteinander verkoppelt werden können sind spezielle Netzübergänge, genannt wie Gateways sehr wichtig. Diese Netzübergangsobjekte empfangen und senden von und zu allen verfügbaren Busschnittstellen, realisieren das Konvertieren von Protokollen sowie die Fehlererkennung. Abhängig von ihrer jeweiligen Verkopplung können bestimmte Schnittstellen eines Gateways auf das Senden oder auf das Empfangen von Daten beschränkt sein. Zudem ist die Implementierung einfacher Filterregeln auf der Ebene der jeweils angeschlossenen Bussysteme möglich.

3.2 Überlegungen

Unser Entwurf wird sich nicht auf die in der Einführung behandelten Bussysteme exakt und detailliert beziehen, sondern wir werden das Ganze versuchen zu verallgemeinern. Wir sprechen nicht mehr von bekannten Bussystemen wie CAN, LIN, FlexRay und MOST, sondern interessieren uns nur für verschiedene Klassen von Bussen z.B. A, B, C und D. Das bedeutet ein Bussystem der Klasse A unterscheidet sich von die anderen Klassen B, C und D nur durch den **header**, **trailer** und **body** seines Protokolls. Der **body** unterscheidet sich nur durch seine Größe, hat aber das gleiche Format.

Abbildung 3.1: Datenprotokoll

Bussysteme der gleichen Klasse haben in diesem Fall immer das gleiche Protokoll. Also exakt gesagt: wir haben nur vier verschiedene Protokolle und ein Gateway als Regelungsstation, die alle vier Klassen verarbeiten kann. Verarbeiten bedeutet in andere Klasse umwandeln, Informationen auswerten, zurückschicken oder Informationen weiterschicken usw.

Wir definieren weiterhin eine Klasse G, die sozusagen alle Protokolle der anderen vier Klassen vereinigt. Also eine neue Überklasse für das Gateway, die die Klassen A bis D als Unterklassen auffassen kann. Damit man z.B. A in B umwandelt, braucht man zusätzlich einem Konverter von G nach B. A wird zuerst in G umwandelt und dann durch den B-

Konverter zu B. Insgesamt sind also vier Konverter nötig. Botschaften der verschiedenen Klassen sollen zuerst in einem dafür geeigneten Puffer hinterlegt werden, für vier Klassen benötigen wir für unsere Zwecke nur einen Puffer, der alle Arten von Protokollen aufnehmen kann. Konverter G wandelt die Protokolle aller Klassen in einem einzigen Format G, die dann in dem Puffer gespeichert werden können. Der einfachheitshalber gehen wir davon aus, dass in diesem Puffer die Botschaften nach Priorität sortiert sind. Botschaft der Format G ist so groß, dass sie alle anderen Botschaften der Klassen A bis D aufnehmen kann.

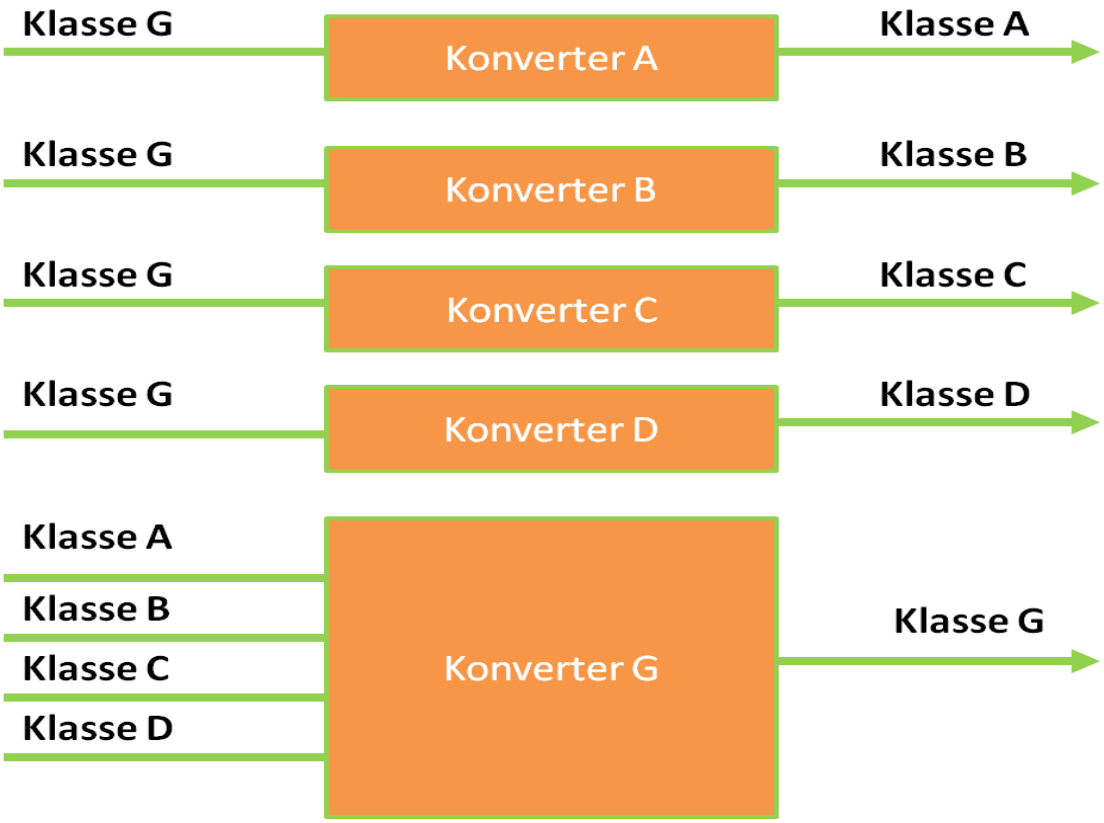


Abbildung 3.2: Konverter

Protokolle aller Klassen sind in ihrem Aufbau ähnlich. Sie bestehen nur aus den drei Komponenten Header, Body und Trailer, siehe Abbildung 3.1. Vor dem eigentlichen Datenfeld wird ein Kopf (Header) übertragen, der Adressinformation über Sender und Empfänger und Angaben über die Anzahl der zu übertragenden Daten und die Art enthält. Datenfelder der verschiedenen Klassen unterscheiden sich nur in ihrer Größe.

An den Body ist immer ein Trailer angehängt, der Informationen zur Fehlerprüfung und Fehlerkorrektur enthält.

Bevor wir mit dem Entwurf anfangen, fassen wir unsere Überlegungen noch einmal zusammen. Wie man in der Abbildung 3.3 sieht, haben wir vier verschiedene Klassen aus Bussystemen A, B, C, D. Hinter diesen Bussen steht eine Topologie von Busleitungen und mechanische Steckverbindungen, also eine Reihe von Steuergeräten, die miteinander verbunden. Es kann sich auch um Diagnosebus für Entwicklung, Applikation, Produktion und

Werkstatt handeln. Alle diese Klassen interessieren uns nur als Sender und Empfänger von Nachrichten. Eine Begrenzung für das Senden und das Empfangen haben wir nicht, das Gateway hat für alle Klassen einen Puffer, der alle Nachrichten bzw. ihre Protokolle zu jeder Zeit empfangen kann. Genauso sorgt jede Klasse auf der anderen Seite dafür, dass jede mögliche Anzahl von Nachrichten, die das Gateway sendet, gepuffert oder sofort bearbeitet werden können.

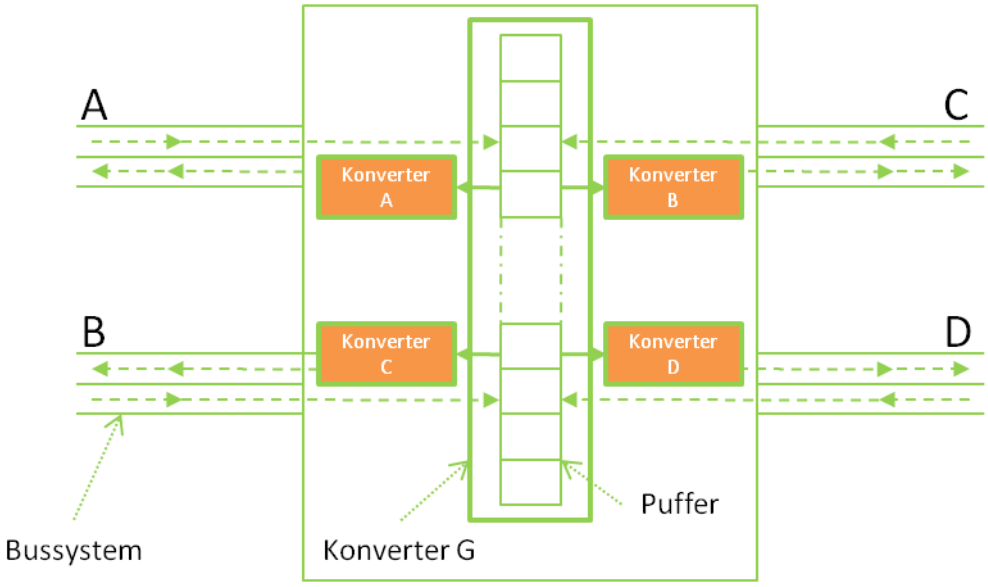


Abbildung 3.3: Gateway

Der innere Teil unseres Gateway hat einen Bereich G, der den Konverter G und andere Funktionalitäten enthält. Zum Beispiel, nachdem ein Protokoll in dem Puffer gelandet ist, muss eine andere Funktion *selectMessage* definiert werden, die diese Nachricht aus dem Puffer holt und diese weiter bearbeitet. Dabei geht es um Prioritäten von Nachrichten. Da die verschiedenen Nachrichten unterschiedliche Prioritäten haben, muss *selectMessage* nur die Priorität und den Ziel-Type der Nachricht kennen. Für die Realisierung von *selectMessage* brauchen wir einen Threadpool, der eine Anzahl von Threads verwaltet. In unseren Fall haben diese Threads immer die gleiche Aufgabe und zwar Nachrichten nach Priorität aus dem Puffer zu holen und für deren Weiterverarbeitung zu sorgen. Die Weiterverarbeitung bedeutet Ziel-Typ und Ziel-Konverter ermitteln, konvertieren und auf das richtige Bussystem weiterleiten. Jeder Thread bekommt vom Threadpool die Aufgabe *selectMessage*, sobald er frei ist, und muss nur diesen Code ausführen.

Für das Umwandeln in G-Protokoll sorgt an dieser Stelle die Funktion *convertToG*. Eine weitere Funktion *checkMessage* soll den Trailer untersuchen und im Falle eines Fehler, muss die Nachricht zurück zu seinem Ursprung geschickt werden. Nach dem Konvertieren in G soll eine weitere Funktion *targetMessege* entscheiden, wohin das konvertierte Protokoll versendet wird. Dieser wird zum Konverter A, B, C, oder D geschickt.

Die vier Konverter A, B, C und D wandeln Klasse G in ihrer Klasse wieder um, damit die Nachrichten von den jeweiligen Bussen empfangen werden können. Für diese Umwandlung

sorgen die vier Funktionen *convertToA*, *convertToB*, *convertToC* und *convertToD*. Hier kann auch der Fall eintreten, dass der Konverter aus einer Nachricht mehrere erstellen muss, falls die Ursprungsbotschaft zu groß für das Zielbussystem ist. Die Nachricht muss dann in kleinere Segmente geteilt und einzeln verschickt werden.

Das Senden geschieht durch die Funktion *sendMessage*, diese Funktion ist für alle vier Konverter die gleiche. Der Funktionsparameter hat den Typ der Protokollmenge {A, B, C, D}.

Für das Empfangen von Nachrichten sorgt der Puffer, den der Konverter G enthält. In unserer Überlegung ist ein Puffer ähnlich wie ein Stack, mit den eingeschränkten Funktionen *emptyBuffer*, *receiveMessage* und *popMessage*. *emptyBuffer* gibt *true* zurück, falls der Puffer leer ist ansonsten *false*. Die Funktion *receiveMessage* fügt Nachrichten in dem Puffer und zwar sortiert nach Priorität. *popMessage* wird von *selectMessage* aufgerufen, die die ausgewählte Nachricht aus dem Puffer entfernt und sie an den richtigen Konverter weiterleitet.

3.3 Entwurf

Als erstes sollen wir das Format eines Protokolls genauer betrachten und die Klassen *protocolA*, *protocolB*, *protocolC*, *protocolD* und *protocolG* definieren, die alle vier verschiedene Protokolle A, B, C und D beschreiben. Ein Protokoll besteht wiederum aus den Komponenten *header*, *body* und *trailer*. *body* variiert nur in der Größe. *header* und *trailer* haben in den verschiedenen Klassen unterschiedliche Formate.

Für den Entwurf benötigen wir keinen exakten Aufbau von jedem Protokolltyp. Wichtig ist nur zu wissen wie groß die einzelnen Botschaften sind und welche Quelle und Ziel diese Botschaften haben. Die Simulation soll am Ende Testfälle durchführen um die Gateway-Auslastung zu prüfen. Dafür brauchen wir nur die Informationen, die für den Ablauf eines Protokolls wichtig sind. Konvertierung bedeutet nur Typänderung und abhängig von der Größe bestimmt diese die Konvertierungsdauer.

Es gibt heute viele Transportprotokolle die das Umwandeln von einem Format zu einem anderen erledigen. Beispiele sind ISO TP für CAN, AUTOSAR TP für FlexRay, TP 2.0 für CAN, TP 1.6 für CAN oder das Transportprotokoll SAE J1939/21 für CAN. Unsere Arbeit behandelt alle diese Transportprotokolle nicht, wir weisen aber darauf hin, dass der Konverter sie oder andere ähnliche Transportprotokolle verwenden könnte.

Der folgende Abschnitt stellt trotzdem die bekanntesten Protokoll-Formate vor. Als erstes sehen wir in Abbildung 3.4 das allgemeine Aufbau-Format eines CAN-Protokolls. Dort findet man keine Ziel- und Ursprungadresse, aber stattdessen einen Identifier, der Informationen über Quelle und Ziel der Botschaft enthält. D.h. durch die Konvertierung wird dafür gesorgt, dass aus Identifier Ziel und Quelle berechnet wird, z.B. von CAN in K-Line Abbildung 3.5 oder CAN in FlexRay Abbildung 3.6.

Abbildung 3.7 zeigt uns das Aufbau-Format eines MOST-Protokolls, das in der Regel selten in ein anderes Protokoll-Format umwandelt wird, da seine Anwendung nur bei der Multimedia stattfindet, und große Mengen von Daten auf CAN-, K-Line- und FlexRay-Bussysteme nicht in Frage kommen.

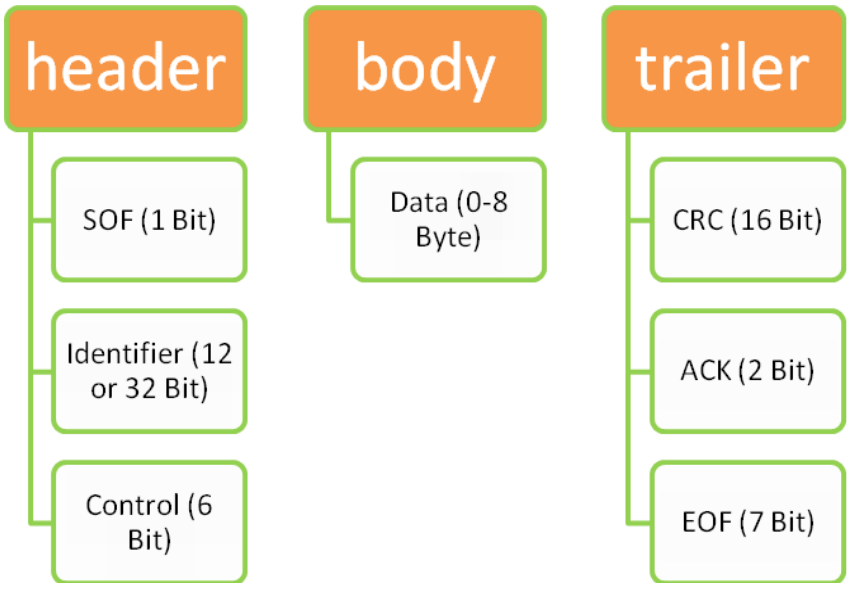


Abbildung 3.4: Der Aufbau eines CAN-Protokolls.

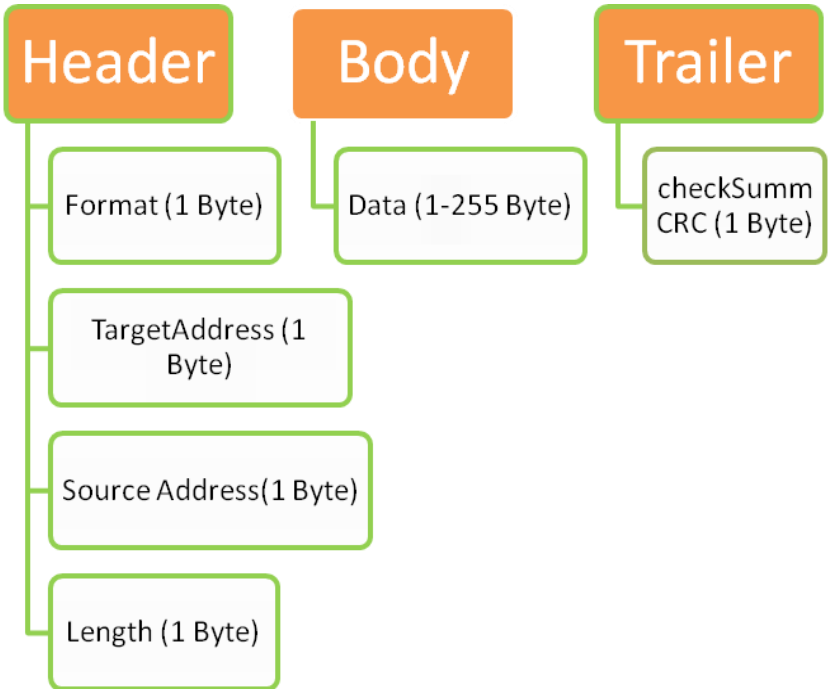


Abbildung 3.5: K-Line

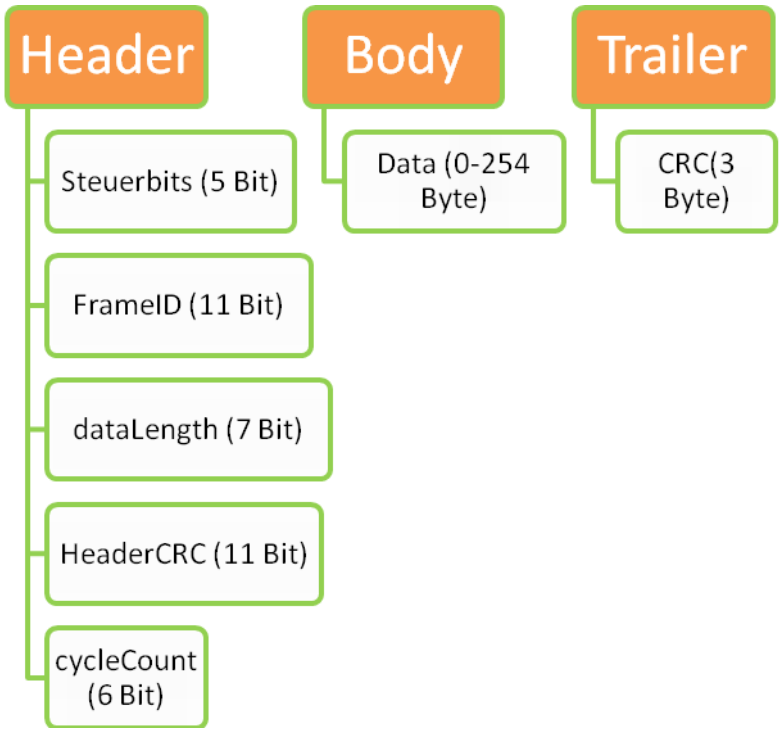


Abbildung 3.6: FlexRay- Botschaftsformat

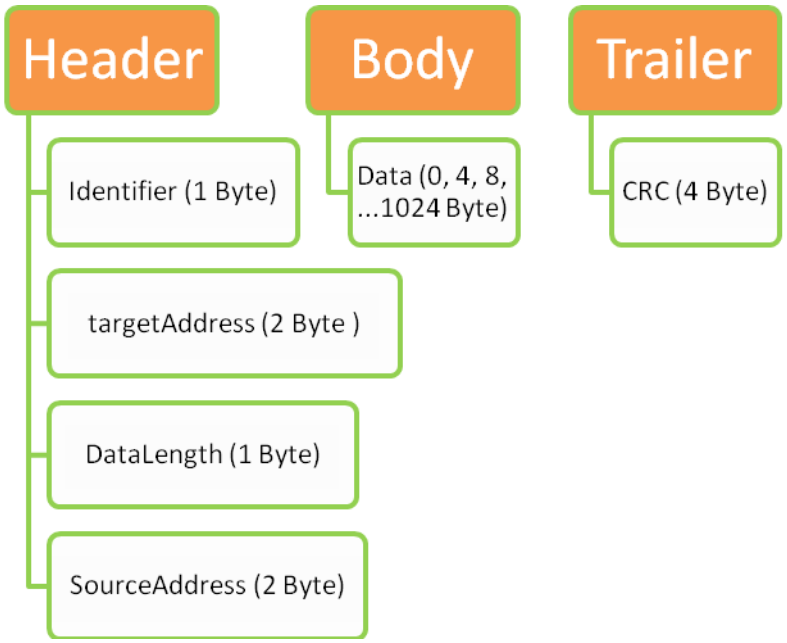


Abbildung 3.7: MOST-Datenpaket für asynchrone Daten.

3.3.1 Klasse G

Die Klasse G gilt als Überklasse für A, B, C und D. Den Puffer den der Konverter G zur Verfügung hat, kann alle anderen vier Klassen aufnehmen. Wir können z.B. an dieser Stelle annehmen, dass der G-Puffer die Ganze Botschaft in seinem *body* speichert. Er benötigt die

Zieladresse (*targetAddress*), damit die anderen Konverter als Threads das Ziel kennen. Darüber hinaus wird die Priorität von jeder Botschaft (*protocolPriority*) benötigt. Botschaften mit der höchsten Priorität werden beim Puffer vorne eingefügt und werden bevorzugt behandelt. Mit *sourceClass* kann der Konverter erkennen, woher die Botschaft kommt.

Damit der Empfänger nicht überlastet wird, steuert der Konverter auch den zeitlichen Abstand zwischen den gesendeten Datenblöcken.

Der Konverter als Threads berechnen die *header*- und *trailer*-Information, und wandelt, falls nötig ist, *targetAddress* in *identifier* oder vice versa um. Der Konverter erstellt eine neue oder mehrere neue Botschaften und leitet sie dann als Bit-Folge auf seinem Bus mit der Funktion *giveMessageToBus*. Der Empfänger dort kann diese Botschaften von seiner Seite weiter bearbeiten. Die nächste Abbildung 3.8 gibt eine grobe Vorstellung, wie die verschiedenen Protokolle in dem Puffer G aussehen.

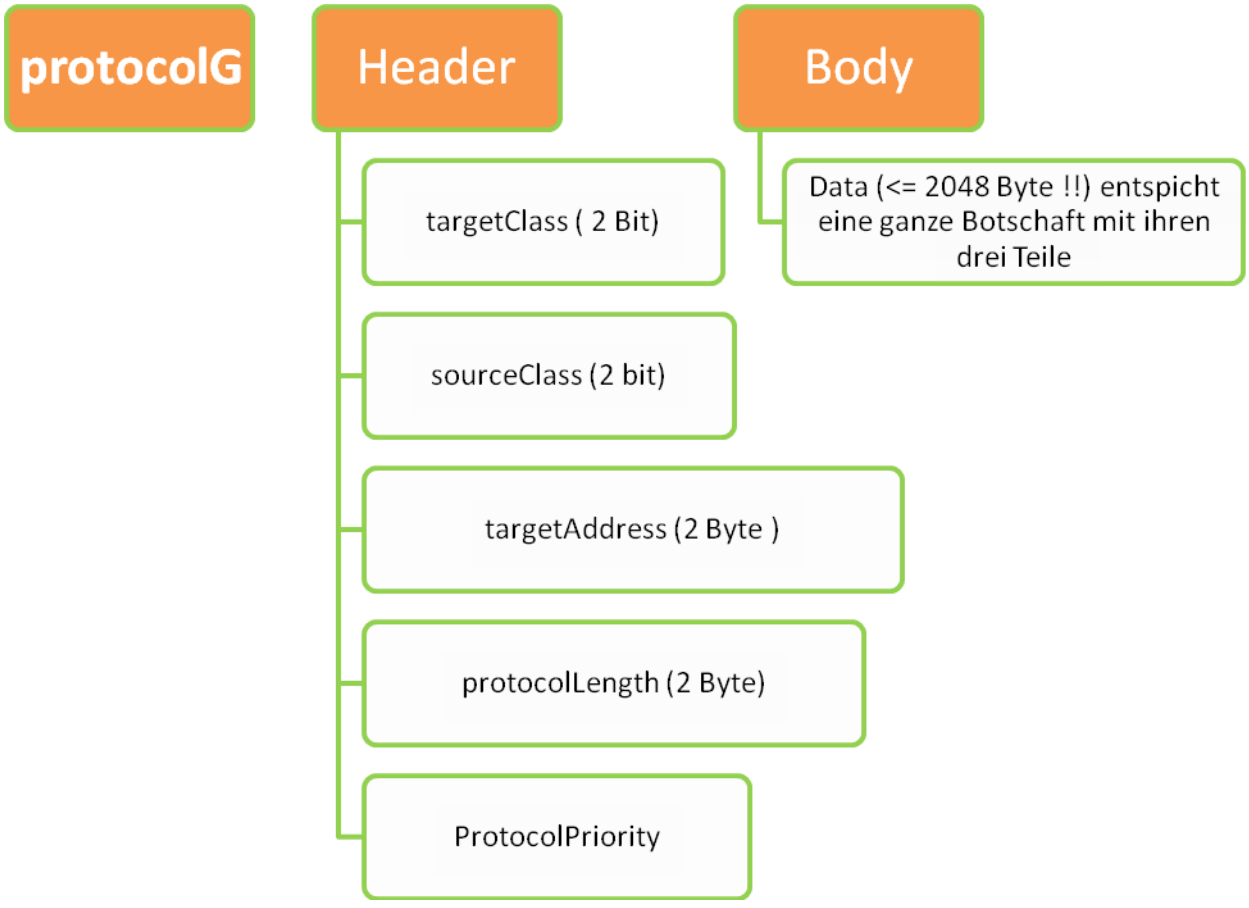


Abbildung 3.8: Format der Klasse G

3.4 Klassen und Methoden

Wir fassen an dieser Stelle alle Methoden zusammen. Zuerst fangen wir mit den Puffer Methoden an.

3.4.1 Puffer Methoden

```
Function selectMessage () return source; /* source :{ A, B, C, D} */
```

Diese Funktion sucht in dem Puffer und legt das Protokoll fest, das als nächstes bearbeitet wird. Dabei verwendet sie für die Suche targetMessage(), die an Puffer Elemente angewendet wird, um Ziele zu berechnen. Es folgt *popMessage*, die eine Nachricht aus dem Puffer entfernt und an den Konverter G weiterleitet.

```
TargetMessageA (p: protocolA) return source; /* source :{ A, B, C, D} */
```

Findet das Ziel von *protocolA* in Puffer A.

```
TargetMessageB (p: protocolB) return source; /* source :{ A, B, C, D} */
```

Findet das Ziel von *protocolB* in Puffer B.

```
TargetMessageC (p: protocolC) return source; /* source :{ A, B, C, D} */
```

Findet das Ziel von *protocolC* in Puffer C.

```
TargetMessageD (p: protocolD) return source; /* source :{ A, B, C, D} */
```

Findet das Ziel von *protocolD* in Puffer D.

```
Function popMessage () return protocol; /**/
```

Eine Nachricht aus dem Puffer entfernen**.**

```
procedure receiveMessageA( m: Message) ; /**/
```

Erhält eine Bit-Folge, wandelt sie in *protocolA* und fügt sie dann sortiert in den Puffer.

```
procedure receiveMessageB( m: Message) ; /**/
```

Erhält eine Bit-Folge, wandelt sie in *protocolB* und fügt sie dann sortiert in den Puffer**.**

```
procedure receiveMessageC( m: Message) ; /**/
```

Erhält eine Bit-Folge, wandelt sie in *protocolC* und fügt sie dann sortiert in den Puffer.

```
procedure receiveMessageD( m: Message) ; /**/
```

Erhält eine Bit-Folge, wandelt sie in *protocolD* und fügt sie dann sortiert in den Puffer.

```
Function emptyBuffer () return Boolean; /**/
```

Ist der Puffer leer?

3.4.2 Konverter Methoden

```
Function checkMessageA (p: protocolA) return Boolean; /**/
```

Prüft den *trailer* Teil auf Fehler, im Falle von *false* wird die Nachricht zu seinem Ursprung geschickt wird.

```
Function checkMessageB (p: protocolB) return Boolean; /**/
```

Prüft den *trailer* Teil auf Fehler, im Falle von *false* wird die Nachricht zu seinem Ursprung geschickt wird.

```
Function checkMessageC (p: protocolC) return Boolean; /**/
```

Prüft den *trailer* Teil auf Fehler, im Falle von *false* die Nachricht zu seinem Ursprung geschickt wird.

```
Function checkMessageD (p: protocolD) return Boolean; /**/
```

Prüft den *trailer* Teil auf Fehler, im Falle von *false* wird die Nachricht zu seiner Ursprung geschickt wird.

```
Function convertToA (p: protocolG) return protocolA; /**/
```

Konvertierung von *protocolG* in *protocolA***.**

```
Function convertToB (p: protocolG) return protocolG; /**/
```

Konvertierung von *protocolG* in *protocolB***.**

```
Function convertToC (p: protocolG) return protocolG; /**/
```

Konvertierung von *protocolG* in *protocolC***.**

```
Function convertToD (p: protocolG) return protocolG; /**/
```

Konvertierung von *protocolG* in *protocolD***.**

```
Function convertAtoG (p: protocolA) return protocolG; /**/
```

Konvertierung von *protocolA* in *protocolG***.**

```
Function convertBtoG (p: protocolB) return protocolG; /**/
```

Konvertierung von *protocolB* in *protocolG***.**

```
Function convertCtoG (p: protocolC) return protocolG; /**/
```

Konvertierung von *protocolC* in *protocolG***.**

```
Function convertDtoG (p: protocolD) return protocolG; /**/
```

Konvertierung von *protocolD* in *protocolG***.**

```
Function targetMessageG (p: protocolG) return target; /* target :{ A, B, C,
D} */
```

Findet das Ziel an den *protocolG* gesendet wird.

```
Procedure takeMessage (p: protocolG, t: target); /**/
```

Nach dem *targetMessage* das Ziel t herausfindet, vermittelt *takeMessage* Protokoll p an den Konverter. Vom Threadpool wird ein Thread zu beauftragt, der eine Nachricht aus dem Puffer holt.

```
procedure giveMessageToBusA(p: protocolA);/**/
```

Nachdem der Konverter A das Protokoll *protocolA* erstellt hat, wird *protocolA* in einer Bit-Folge umwandelt und auf dem A-Bus weitergeleitet**.**

```
procedure giveMessageToBusB(p: protocolB);/**/
```

Nachdem der Konverter-B das Protokoll *protocolB* erstellt hat, wird *protocolB* in einer Bit-Folge umwandelt und auf dem B-Bus weitergeleitet**.**

```
procedure giveMessageToBusC(p: protocolC);/**/
```

Nachdem der Konverter-C das Protokoll *protocolC* erstellt hat, wird *protocolC* in einer Bit-Folge umwandelt und auf dem C-Bus weitergeleitet**.**

```
procedure giveMessageToBusD(p: protocolD);/**/
```

Nachdem der Konverter-D das Protokoll *protocolD* erstellt hat, wird *protocolD* in einer Bit-Folge umwandelt und auf dem D-Bus weitergeleitet**.**

3.5 Statistische und visuelle Analyse

Im 3.2 und 3.3 Überlegungen und Entwurf haben wir uns nur mit der Kommunikation zwischen Puffer und Konverter oder zwischen Konverter und Bus beschäftigt. Aber wie viele Daten können in einem Puffer passen? Was geschieht, wenn der Puffer voll ist? Oder wie schnell kann ein Konverter arbeiten? Solche und weitere Fragen wurden bisher nicht berücksichtigt. Wir müssen uns daher an dieser Stelle noch Gedanken machen, wie man dafür sorgt, dass das Gateway nicht überbelastet wird, und wie die verschiedenen Busse gerecht behandelt werden. Dafür müssen noch andere Parameter und Methoden definiert werden, die alle Funktionalitäten eines Gateway überwacht, und uns letztendlich die Möglichkeit gibt Testfälle und Statistiken generieren zu können.

Damit wir wissen können, wie schnell einen Konverter ist, müssen neue Variablen definiert werden. Als erstes gibt der Parameter *speed* in Bit/s die Geschwindigkeit eines Konverters an. Für alle fünf Konverter brauchen wir dann *speedA*, *speedB*, *speedC*, *speedD* und *speedG*.

Die Abholdauer von G zu den anderen Konvertern berechnet die Funktion *speedGto<X>*. *isFreeA*, *isFreeB*, *isFreeC*, *isFreeD*, und *isFreeG* werden hauptsächlich von *selectMessage* benötigt, gibt an ob die entsprechenden Konverter Einsatz bereit sind oder nicht. Jeder

Konverter hat daher eine Variable *free*, die die Werte *true* und *false* annehmen. *free* wird auf *false* gesetzt, sobald der Konverter eine Nachricht erhält. Ist er mit der Bearbeitung der Nachricht fertig setzt er nach einem *timeOut* die Variable *free* wieder auf *true*. *timeOut* ist eine optionale Konstante und kann ignoriert werden, indem den Wert auf null setzt.

Die Bearbeitungszeit einer Nachricht bedeutet für den Konverter das Zeitintervall zwischen Nachrichtübername von G und die Nachrichtübergabe in Bit-Folge an das Bussystem. Was außerhalb des Gateway passiert wird nicht berücksichtigt. Daher wird an dieser Stelle die Annahme gemacht, dass alle Busse A bis D frei sind. Die Übergabe an den Bussen funktioniert immer und einwandfrei.

Zunächst betrachten wir die Übergabe von dem Puffer an Konverter G. Hier müssen wir uns die Funktion *selectMessage* nochmal detailliert untersuchen. Wie vorher schon erwähnt worden, sucht *selectMessage* immer einen Konverter aus, von diesem wird die nächste Nachricht herausgeholt und bearbeitet. G kann aber nicht immer frei sein. In diesen Fall muss *selectMessage* warten und das festgelegte Pufferelement beibehalten, d.h. sie geht solange in einer Schleife, bis *freeG* auf *true* gesetzt wird.

Wenn *freeG true* wird, heißt das nicht automatisch, dass der ausgewählte Nachricht an Konverter G weitergeleitet wird, denn der wirkliche Ziel-Konverter angenommen A soll in diesem Moment auch bereit sein eine Nachricht zu erhalten. Ist Konverter A in diesem Moment frei, so kann *popMessage* aufgerufen werden, die dann *speedMessageToG* Zeit braucht, um die ausgewählte Nachricht an G zu schicken.

Jetzt sollen wir aber den schwierigen Fall betrachten und zwar, wenn A auch belegt ist. D.h. *selectMessage* hat ein Pufferelement gewählt. Konverter G ist frei aber A nicht. Bei so einer Situation soll *selectMessage* die Suche neu starten und betrachten, ob ein anderer Konverter frei ist, und den, der mit der höchsten Priorität zu bedienen, falls einer oder mehrere frei sind. Ansonsten werden durch aktives Warten die Variablen *freeA*, *freeB*, *freeC* und *freeD* ständig überprüft, bis zumindest eine auf *true* gesetzt wird, damit für die sichere Kommunikation zwischen den Puffer und die fünf Konverter gesorgt wird.

Anhand der Methode *selectMessage* sieht man, wie kompliziert so ein Gateway zu implementieren ist. Weitere Überlegungen und Verbesserungsmöglichkeiten für den ganzen Entwurf werden in dem Abschnitt Skalierbarkeit und Verbesserungen noch erwähnt.

3.5.1 Methoden zur Zeitberechnung

```
function speedMessageAtoG(p: protocolA) return Time;
```

Berechnet die Zeit, die vergeht bis *protocolA* von Puffer A in G landet.

```
function speedMessageBtoG(p: protocolB) return Time; /**/
```

Berechnet die Zeit, die vergeht, bis *protocolB* von Puffer B in G landet.

```
Function speedMessageCtoG(p: protocolC) return Time; /**/
```

Berechnet die Zeit, die vergeht, bis *protocolC* von Puffer C in G landet.

```
function speedMessageDtoG(p: protocolD) return Time; /**/
```

Berechnet die Zeit, die vergeht, bis *protocolD* von Puffer D in G landet.

```
function speedMessageGtoA(p: protocolG) return Time; /**/
```

Berechnet die Zeit, die vergeht, bis *protocolG* von Konverter G in A landet.

```
function speedMessageGtoB(p: protocolG) return Time; /**/
```

Berechnet die Zeit, die vergeht, bis *protocolG* von Konverter G in B landet.

```
function speedMessageGtoC(p: protocolG) return Time; /**/
```

Berechnet die Zeit, die vergeht, bis *protocolG* von Konverter G in C landet.

```
function speedMessageGtoD(p: protocolG) return Time; /**/
```

Berechnet die Zeit, die vergeht, bis *protocolG* von Konverter G in D landet.

```
function speedConverterA(p: protocolG) return Time;/**/
```

Berechnet die Zeit für das Konvertieren von G in A + das Versenden an Bus A.

```
function speedConverterB(p: protocolG) return Time;/**/
```

Berechnet die Zeit für das Konvertieren von G in B + das Versenden an Bus B.

```
function speedConverterC(p: protocolG) return Time;/**/
```

Berechnet die Zeit für das Konvertieren von G in C + das Versenden an Bus C.

```
function speedConverterD(p: protocolG) return Time;/**/
```

Berechnet die Zeit für das Konvertieren von G in D + das Versenden an Bus D.

4 Ergebnisse

In diesem Abschnitt erläutern wir zuerst, wie die Eingabe definiert werden soll, damit wir die Überflutungen in dem Gateway kontrollieren und beeinflussen können. Danach kommen die Ausgabe-Darstellung und Beobachtungen, die die Ausgabe auswerten und verdeutlichen. Am Ende kommen wir dann zur Vorteile, Nachteile und Verbesserungsvorschläge.

4.1 Eingabe und Ausgabe

Für das Testen und die Ausführung von unserem Entwurf sollen wir noch vier weiteren Dateien erstellen, die die gesendeten Botschaften und deren Verlauf darstellen. Da wir uns auf vier Busse beschränkt haben, erhält jeder dieser Dateien immer nur Botschaften, die zu einer Klasse gehören. D.h. die Datei *A_Message.txt hat* nur Botschaften die aus dem Bussystem A kommen.

Alle vier Dateien beinhalten eine Anzahl von Botschaften, die zeilenweise untereinander gelistet sind. Die verschiedenen Informationen in einer Zeile sind durch einen Trennzeichen z.B. ein Komma getrennt. Jeder Zeile fängt mit dem Argument *StartingTime* an, das der Ausführungszeitpunkt einer Zeile entspricht, die restlichen Argumenten sind nur Daten, die eine Botschaft normalerweise enthält, siehe Abbildung 4.2.

In *A_message.txt* fangen die ersten drei Zeilen mit 0, 32 und 76 an, d.h. die erste Zeile wird direkt mit dem Start des Programm ausgeführt also zum Zeitpunkt null und die zweite Zeile nach 32 Millisekunden usw.

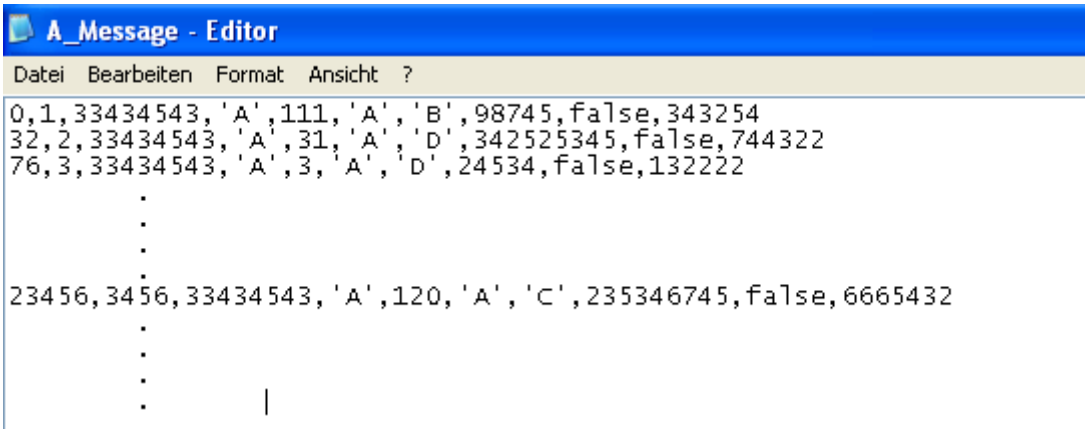


Abbildung 4.1: A_Message.txt

Ausführung Zeile zwei bedeutet, sende die zweite Botschaft aus dem Bussystem A an den Puffer G erst 32 Millisekunden nach dem Start. Jede Datei hat seinen eigenen *Task*, der die Zeilen der Reihe nach bearbeitet, wir brauchen also vier Tasks A, B, C und D für die vier verschiedenen Dateien. Da vier Tasks gleichzeitig in dem Puffer schreiben wollen, kann es zu

Verzögerungen kommen, die zufolge haben, dass Task A z.B. länger als 32 Millisekunden braucht, bis er die Zeile 2 erreicht, in diesem Fall wird die Botschaft in Zeile 2 nicht

```
public class Protokoll
{
    public int ProId { get; set; }
    public int ProTimeDuration { get; set; }
    public string ProType { get; set; }
    public int ProPriority { get; set; }
    public string ProSource { get; set; }
    public string ProTarget { get; set; }
    public int ProSize { get; set; }
    public bool ProError { get; set; }
    public int ProSpeed { get; set; }
```

Abbildung 4.2: Klasse-Protokoll

übersprungen d.h. alles, was in der Datei ist, kommt irgendwann in Puffer.

Verzögerungen werden nicht nur wegen der vier Tasks verursacht, wir haben noch zum einen ein Threadpool, der die Botschaften aus dem Puffer holt, und zum anderen einen weiteren Task, der eine ähnliche Funktionalität wie *garbage collector* hat, und ständig die Botschaften in dem Puffer nach Lebensdauer überprüft. Falls diese überschritten würde, werden sie gelöscht.

Auf dem Puffer darf nur ein Task oder ein Thread zur gleichen Zeitpunkt arbeiten. Alle anderen müssen warten, bis er mit seiner Aufgabe fertig wird.

In den Methoden für Zeitberechnungen sollen alle diese Verzögerungen beachten und in der Ausgabe-Datei mit ausgegeben.

Nachdem die Nachrichten von den Konvertern bearbeitet werden, landen sie auch zeilenweise in der Ausgabe-Datei. Dort enthält eine Zeile noch weitere Informationen wie z.B. die geplante Ankunftszeit und die wahre Ankunftszeit. Die wahre Ankunftszeit wird auch davon beeinflusst, dass in der Ausgabe-Datei nur ein Thread oder einen Task den Zugriff zu schreiben hat, alle anderen müssen auch hier warten. Botschaften, die deren Lebensdauer überschritten würde, sollen auch in der Ausgabe-Datei landen in diesem Fall muss nur der Argument *ProError* auf *true* gesetzt wird. Geplante Ankunftszeit wird durch Methoden für Zeitberechnung ermittelt und entspricht die Zeit, die keine Verzögerungen, die durch die anderen Prozesse in dem Gateway entstehen, betrachtet.

4.2 Mögliche Beobachtungen

Wie das Gateway ausgelastet wird, können wir durch die Eingabe kontrollieren und zwar durch *StartingTime* der Botschaften. Wir können die Auslastung verringern, indem wir z.B. die Ausführungszeiten so eingeben, dass der Puffer bei jedem Speichern leer ist. So haben wir sogar keine Auslastung. Dieser Fall ist für das Gateway das Beste, aber durfte nicht vorkommen, sonst wären der Puffer und der Threadpoll überflüssig. Der schlimmste Fall tritt

dagegen ein, wenn *StartingTime* überall gleich null ist, d.h. alle Nachrichten wollen beim Start sofort in dem Gateway eindrängen, aber nur einer darf hinein. In solcher Situation hat das Gateway die höchste Auslastung, denn ein Thread oder ein Task arbeitet und alle anderen warten auf ihn, obwohl sie alle genug Arbeit haben.

Da viele Botschaften lange Zeit warten müssen geht ihre Lebensdauer zu Ende, sodass diese Botschaften zu einem späteren Zeitpunkt von den Bussystem wiederholt geschickt werden müssen. Das bedeutet auch zusätzliche Arbeit für die Bussysteme.

Wir wollen auch den Fall betrachten, in dem nur der Threadpool arbeitet, d.h. alle Nachrichten sind in dem Puffer gespeichert und die vier Tasks, die die Daten von außen liefern, sind mit ihren Aufgaben fertig. Hier kann es z.B. dazu kommen, dass die ganze Auslastung auf einen Konverter fällt, denn es ist auch möglich, dass alle Pufferinhalte nur Richtung einen einzigen Bussystem wandern wollen, also alle Botschaften haben das gleiche Ziel-Bussystem. In diesen Fall sind mehrere Threads nicht nötig und ein einziger Thread kann die ganze Arbeit leisten.

4.3 Skalierbarkeit und Verbesserungen

Ungerechte Verteilung der Nachrichten auf mehreren Bussen kann den Grad der Skalierbarkeit sehr stark beeinflussen und das, obwohl wir davon ausgehen können, dass der Puffer zur Laufzeit des Programmes nie leer ist. Verbesserungen sind hier nur durch Hardwarelösungen möglich, d.h. entweder erhöhen wir der Anzahl der Konverter oder wir verwenden Hochleistungskonvertern, die mehrere Protokolle gleichzeitig bearbeiten können.

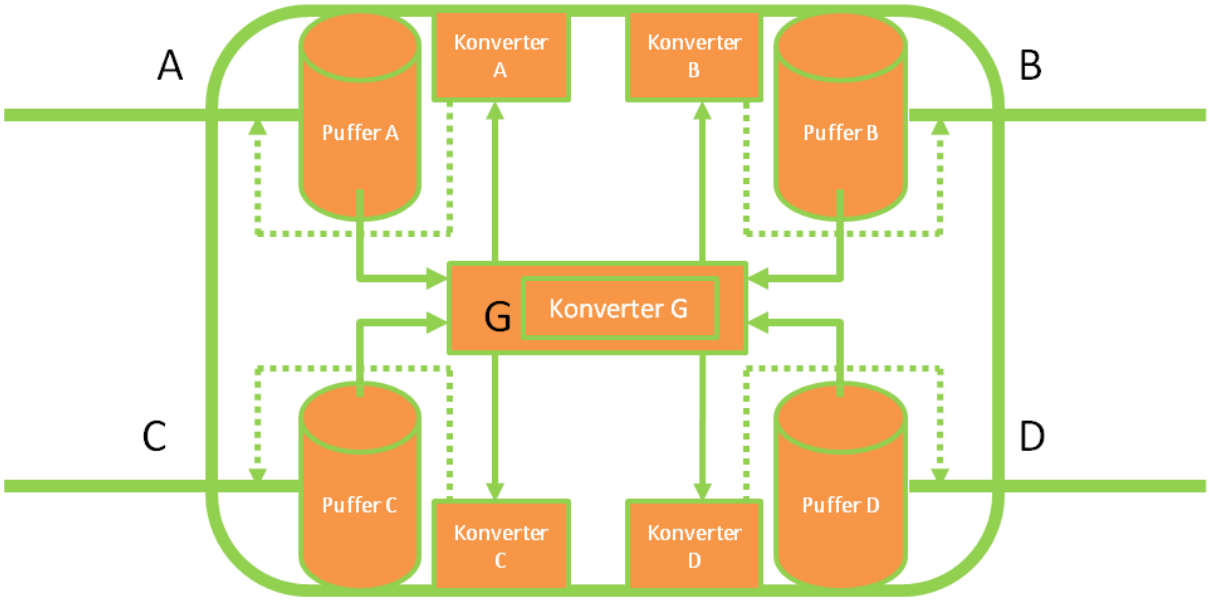


Abbildung 4.3: Gateway mit vier Puffern

Die Verwendung von nur einem einzigen Puffer für das gesamte Gateway kann für die äußeren vier Tasks, die die Daten zu dem Gateway schicken, als Nachteil angesehen werden, denn aufgrund des Verhindern von Race-Condition, dürfen mehrere Tasks nicht gleichzeitig schreiben oder den Puffer manipulieren. Das führt uns zu der Idee, mehreren Puffern zu benutzen. So hat jedes Bussystem seine eigenen Puffer, siehe Abbildung 4.3. Dort sind die vier Tasks unabhängig voneinander und können zu jeder Zeit in ihren Puffer schreiben. Dieser Vorschlag kann zwar dafür sorgen dass alle Nachrichten unverzögert in das Gateway hineinkommen, aber wie sie aus dem Puffer herausgenommen werden, ist wiederrum sehr kompliziert. Im Puffer A liegen Nachrichten mit verschiedenen Zielen. Das heißt für die Funktion *selectMessage* immer abhängig von dem freien Konverter in allen vier Puffern suchen. Die Suche beansprucht viel Zeit, da nur ein Thread die Puffer durchsuchen darf. In dieser Suchphase sind alle Puffer gelockt, um ein Race-Condition zu vermeiden.

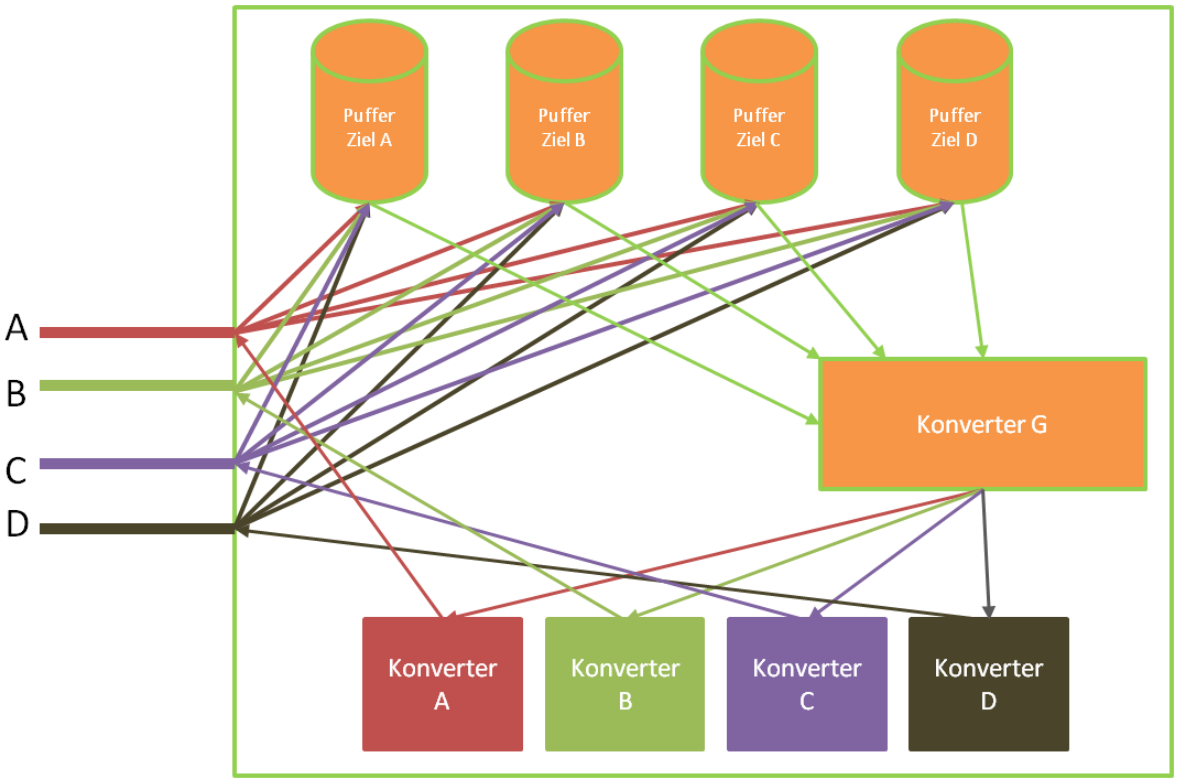


Abbildung 4.4: Gateway speichern nach Ziel

Ein besserer Vorschlag wäre wie in Abbildung 4.4 zu sehen ist, von Anfang an nach Ziel zu Speichern. Hier kann man die Suche ersparen aber beim Schreiben in den Puffer kann es dazu kommen, dass alle vier Tasks gleichzeitig in einem einzigen Puffer schreiben wollen. Und das führt wiederum zu Verzögerungen.

5 Ausblick

Im realen Fall läuft der Datenaustausch meistens nur auf einem einzigen Bussystem zwischen verschiedenen Steuergeräten, die an diesem Bussystem angeschlossen sind. Es ist nur manchmal für den Tester oder andere externe Geräte notwendig, Protokolle der anderen Busse in ihr Format umzuwandeln. Aus diesem Grund sollte man den Zeitfaktor, dem wir eine sehr wichtige Rolle zugeordnet haben, nicht akribisch untersuchen. Denn Datenüberflutung kann beim Autofahren nur innerhalb eines einzigen Bussystems entstehen und nicht zwischen verschiedenen Bussen. Ein Gateway, das alle Bussysteme miteinander verbindet, dient nur für den externen Zugriff, z.B. für Datenauswertung oder für das Installieren von neuer Software.

Innerhalb eines Bussystems können Sender X und Empfänger Y miteinander sehr leicht kommunizieren, da ihre Botschaften das gleiche Format haben. Man könnte sich aber überlegen, wie ein Datenaustausch zwischen X und Y funktionieren kann, wenn sie zu zwei verschiedenen Bussen A und B gehören und trotzdem miteinander kommunizieren wollen. Eine Idee wäre: X erstellt eine Botschaft für Y und schreibt sie in seinem Datenfeld (Payload) siehe Abbildung 5.1, in dem Header stehen die üblichen Daten, die das Bussystem A benötigt, wobei die Zieladresse nicht Y ist, sondern ein A-Steuergerät, das sich im Gateway befindet.

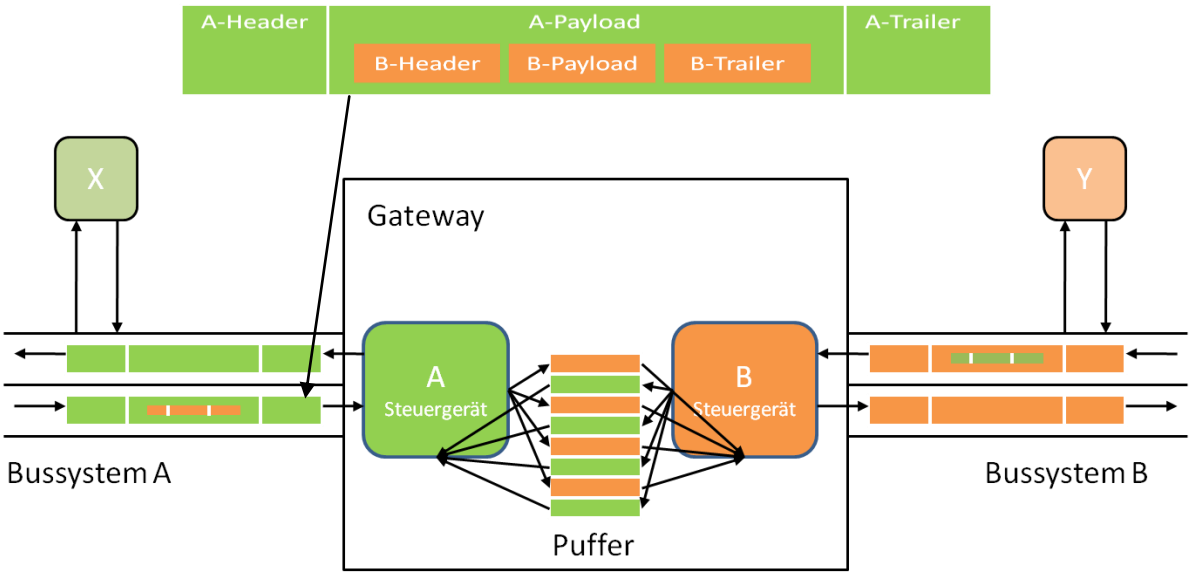


Abbildung 5.1: Gateway

Dieses A-Steuergerät holt nur das Datenfeld (Payload) einer Botschaft und schreibt es in einen externen Puffer, der sich im Gateway befindet. Ein weiteres B-Steuergerät, das auch zum Gateway gehört, kann den Puffer nach B-Botschaften durchsuchen und sie, falls es solche gibt, an das B-Bussystem weiterleiten. So könnte eine Kommunikation zwischen X und Y erfolgen.

Aber kann X überhaupt eine Botschaft für Y erzeugen? Ist das durch neue Softwares möglich? Oder benötigt man dafür leistungsfähigere Steuergeräte? Das wird sich im Endeffekt auch auf die Kosten auswirken.

Literaturverzeichnis

[ZiSc08] **W. Zimmermann, R. Schmidgall:** *Bussysteme in der Fahrzeugtechnick*, Vieweg+Teubner, ISBN 978-3-8348-0447-1, 2008

[ScWi06] **G. Schnell, B. Wiedemann:** *Bussysteme in der Automatisierungs- und Prozesstechnik***,** Vieweg, ISBN 3-8348-0045-7, 2006

[MaSu11] **C. Marscholik, P. Subke,** *Datenkommunikation im Automobil*, VDE, ISBN 978-3-8007-3275-3, 2011

[Borg08] **K. Borgeest:** *Elektronik in der Fahrzeugtechnik*, Vieweg+Teubner, ISBN 978-3-8348-0201-1, 2008

[RiSc06] **R.V. Basshuysen, F. Schäfer:** *Lexikon Motorentechnik*, Vieweg+Teubner, ISBN 978-3-528-13903-2, 2006

[Witt02] **F. Wittgruber:** *Digitale Schnittstellen und Bussysteme,* Vieweg**,** ISBN 3-528-17436-6, 2002

[BrSe07] **H.H. Braess, U. Seiffert:** *Automobildesign und Technik,* Vieweg+Teubner, ISBN 978-3-8348-0177-7, 2007

[Raif09] **K. Raif:** *Automobilelektronik*, , Vieweg+Teubner, ISBN 978-3-8348-0446-4, 2009

[Trau09] **T. Trautmann:** *Grundlagen der Fahrzeugmechatronik,* Vieweg+Teubner, ISBN 978-3-8348-0387-0, 2009

[WaKo06] **H. Wallentowitz, K. Raif:** *Handbuch Kraftfahrzeugelektronik,* Vieweg+Teubner, ISBN 978-3-528-03971-4, 2006

[WFO09] **H. Wallentowitz, A Freialdenhoven, I. Olschewski:** *Strategien in der Automobilindustrie,* Vieweg+Teubner, ISBN 978-3-8348-0725-0, 2009

[Bosc07] **Robert Bosch GmbH:** *Autoelektrik/Autoelektronik*, Vieweg+Teubner, ISBN 978-3-528-23872-8, 2007

[HoBr08] **E. Hoepke, S. Breuer:** *Nutzfahrzeugtechnik,* Vieweg+Teubner, ISBN 978-3-8348-0374-0, 2008

[Elle10] **F. Eller:** *Visual C# 2010,* Addison Wesley, ISPN 978-3-8273-2916-5, 2010

[Emot11] **Emotive GmbH:** http://www. http://emotive.de/de/doc/car-diagnostic-systems/contents, 2011

[Flex11] **FlexRay Consortium:** http://www.flexray.com , 2011

[Can-11] **CAN in Automation:** http://www.can-cia.de, 2011

[Most11] **MOST Cooperation:** http://www.mostcooperation.com, 2011

[Lin-11] **LIN Consortium:** http://www.lin-subbus.org, 2011

[Vect11] **Vector Informatik GmbH:** http://vector-informatik.de, 2011

[Wiki11a] **Wikipedia, die freie Enzyklopädie:** http://de.wikipedia.org/wiki/Bussysteme, 2011

[Wiki11b] **Wikipedia, die freie Enzyklopädie:** http://de.wikipedia.org/wiki/Gateway_(Informatik), 2011

[Wiki11c] **Wikipedia, die freie Enzyklopädie:** http://de.wikipedia.org/wiki/Remote_Login ,2011

[Wiki11d] **Wikipedia, die freie Enzyklopädie:** http://de.wikipedia.org/wiki/FlexRay,2011

[Me-s11] **ME-Meßsysteme GmbH,** http:// http://www.me-systeme.de/canbus.html, 2011