Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES
Näher dran.

Hochschule Karlsruhe – Technik und Wirtschaft
Fakultät für Maschinenbau und Mechatronik

Weiterentwicklung eines autonom fahrenden Demonstrators für Fahrerassistenzsysteme

Thesis
zur Erlangung des Grades
Master of Science (M. Sc.)

Christof Kary
geb. am 05.04.1993
in Rastatt
Matrikel-Nr.: 58516

Betreuer der Hochschule Karlsruhe
Herr Prof. Dr.-Ing. Ferdinand Olawsky

Betreuer am Arbeitsplatz
Herr Dipl.-Ing. Arthur Kessler

Flacht, 1. April 2018 bis 30. September 2018

Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Fakultät für Maschinenbau und Mechatronik

Master-Thesis: **Christof Kary**
Effiziente Mobilität in der Fahrzeugtechnologie

Arbeitsplatz: EVOMOTIV GmbH
Im Bühl 16/1
71287 Flacht

Betreuer am Arbeitsplatz: Dipl.-Ing. Arthur Kessler

Betreuender Dozent: Prof. Dr.-Ing. Ferdinand Olawsky

Datum der Ausgabe: 01.04.2018 **Abgabetermin:** 30.09.2018

Kurzthema / Subject:

Weiterentwicklung eines autonom fahrenden Demonstrators für Fahrerassistenzsystem

Development of an Autonomously Driving Demonstrator for Driver Assistance Systems

Aufgabenstellung:

Durch den schnellen technologischen Fortschritt in der Entwicklung von Fahrerassistenzsystemen und der intelligenten Vernetzung der Fahrzeugsysteme stellt das hochautomatisierte Fahren einen wachsenden Markt dar, der viel Potenzial hinsichtlich Sicherheit und individueller Mobilität bietet. Bereits heute sind teilautomatisierte Fahrfunktionen in Serienfahrzeugen umgesetzt und die Entwicklung liefert stetig Fortschritte in Richtung des vollautonomen Fahrens.

Die EVOMOTIV GmbH möchte ihr firmeninternes Know-how in diesem Entwicklungssegment weiter ausbauen und damit zur fortschrittlichen Entwicklung hin zum vollautomatisierten Fahren beitragen. Zu diesem Zweck soll im Rahmen dieser Masterarbeit ein bereits bestehender Demonstrator in Form eines Modellfahrzeuges erweitert werden, der verschiedene Funktionen des automatisierten Fahrens abbilden kann. Unter anderem sollen die bereits implementierten Algorithmen optimiert, die autonomen Fahrfunktionen erweitert und eine umfangreiche Fehlerdiagnose implementiert werden.

Im Einzelnen sind die folgenden Punkte zu bearbeiten:

- Einarbeitung in vorherige studentische Arbeiten und in die Grundlagen zum autonomen Fahren, zur Bildverarbeitung und zur Fahrzeugdiagnose
- Konzepterstellung eines Diagnosesystems zur Fehlerdiagnose am Demonstrator
- Aufbau und Implementierung eines Diagnosesystems an eine CAN-Schnittstelle
- Kritische Analyse der implementierten Algorithmen zur Bildverarbeitung und Fahrzeugregelung
- Implementierung einer dynamischen Längs- und Querregelung auf variable Fahrgeschwindigkeiten
- Auswahl und Abwägung weiterer möglicher Fahrfunktionen
- Dokumentation und Präsentation der Ergebnisse

Vorsitzender des Prüfungsausschusses

Prof. Dr.-Ing. Norbert Skricka

Betreuender Dozent

Prof. Dr.-Ing. Ferdinand Olawsky

Eidesstattliche Erklärung

Der Verfasser erklärt, dass er die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt hat. Die aus fremden Quellen (einschließlich elektronischer Quellen) direkt oder indirekt übernommenen Gedanken sind ausnahmslos als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden.

Flacht, den 12. September 2018

Christof Kary

Vorwort

An dieser Stelle möchte ich mich bei allen Personen bedanken, die mich während der Anfertigung dieser Masterarbeit unterstützt und motiviert haben.

Zunächst gebührt mein Dank Frau Dipl.-Betriebsw. Nelly Berg und Herrn Dipl.-Ing. Arthur Kessler, die meine Arbeit im Unternehmen betreut und beaufsichtigt haben. Für die hilfreichen Anregungen und die konstruktive Kritik bei der Erstellung dieser Arbeit möchte ich mich herzlich bedanken. Ebenso für die spontanen Hilfestellungen und den offenen Umgang.

Ebenfalls möchte ich mich bei Herrn Prof. Dr.-Ing. Ferdinand Olawsky bedanken, der als Professor der Hochschule Karlsruhe die Betreuung der Arbeit von offizieller Seite übernommen hat. Vielen Dank für Ihr Interesse an der Betreuung und die interessanten Ideen und Anregungen, die maßgeblich dazu beigetragen haben, dass die Bachelorarbeit in dieser Form vorliegt.

Ein weiterer Dank gilt stellvertretend allen Mitarbeitern der Firma Evomotiv GmbH, die stets für spontane Unterstützungen und Fragen zur Verfügung standen.

Abschließend möchte ich mich bei meiner Familie bedanken, die mich das gesamte Studium über unterstützt und stets ein offenes Ohr für meine Sorgen hatte. Größte Dankbarkeit gebührt meinen Eltern, die mir das Studium ermöglicht und mich mit allen denkbaren Mitteln unterstützt haben. Ihr habt mir das Vertrauen und die Freiheit gegeben, die Dinge so zu tun, wie ich sie für richtig hielt.

Christof Kary,

Karlsruhe, den 12. September 2018

Kurzfassung

Titel der Arbeit in deutscher Sprache

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Abstract

Titel der Arbeit in englischer Sprache

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Abkürzungsverzeichnis

Kurzform	Bedeutung
μC	Mikrocontroller
ACK	Acknowledgement
API	Application Programming Interface
Bus	Binary Utility System
CAN	Controller Area Network
CAPL	Communication Access Programming Language
CRC	Cyclic Redundancy Check
CSMA/CA	Carrier Sense Multiple Access with Collision Avoidance
ECU	Electronic Control Unit
ID	Identifier
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Organization for Standardization
KWP	Keyword Protokoll
LIN	Local Interconnect Network
MOST	Media Oriented Systems Transport
OBD	On-Board-Diagnose
ODX	Open Diagnostic Data Exchange
OSI	Open System Interconnection
SG	Steuergerät
UDS	Unified Diagnostic Services
UML	Unified Modeling Language

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation

Bereits seit einigen Jahren zeichnet sich durch den Einzug moderner Fahrerassistenzsysteme ein Umbruch im individuellen und gesellschaftlichen Umgang mit dem Automobil ab. Als treibende Aspekte für den technologischen Umbruch in der Automobilentwicklung sind die steigende Verkehrssicherheit, der erhöhte Fahrkomfort und eine energieoptimale Fahrzeugführung zu nennen. Alleine im Jahr 2017 kamen 3.177 Menschen bei Verkehrsunfällen auf deutschen Straßen ums Leben [Quelle]. FAS bieten Potential um Unfälle zu senken.

Autonomes Fahren könnte einen Umbruch im individuellen und gesellschaftlichen Umgang mit dem Automobil nach sich ziehen und damit auch Einfluss nehmen auf Verkehr, Mobilität oder Raumstrukturen.

1.2. Zielsetzung der Arbeit

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a

nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

2. Grundlagen

Die vorliegende Schrift ist grundsätzlich als Forschung- und Entwicklungsarbeit des automatisierten Fahrens in der Automobilbranche anzusiedeln. Es werden zunächst einige Schlüsseltechnologien der Fahrerassistenzsysteme vorgestellt und Ihre Notwendigkeit für den Entwicklungsfortschritt hin zum vollautomatisierten Fahren begründet.
Der zentrale Teil der Arbeit liegt in dem Aufbau einer Datenbuskommunikation mit dem Demonstratorfahrzeug zur strukturierten Auswertung der Fahrzeugzustandsdaten. Daher werden zudem einige wichtige Informationen zu Bussystemen und der Datenkommunikation im Fahrzeug vorgestellt. Es gilt zu erwähnen, dass dieser Teilabschnitt aufgrund der thematischen Schnittmenge teilweise aus der vorangegangenen Bachelorarbeit [1] des Verfassers übernommen wurde.

2.1. Autonomes Fahren

2.1.1. Überblick Fahrerassistenzsysteme

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

2.1.2. Autonomiestufen

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero,

nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

2.2. Bussysteme

Die rasante Zunahme an elektronischen Systemen und *Steuergeräten* SG (englisch *Electronic Control Unit* ECU) in den letzten Jahrzehnten machten einen geregelten Datenaustausch in der Fahrzeugtechnik unerlässlich. Stetig steigende Anforderungen an Fahrsicherheit, Motorsteuerung und Komfortsysteme erforderten zwingend einen sicheren und schnellen Informationsfluss zwischen den kommunizierenden Steuergeräten, sodass ein verteiltes und vernetztes Gesamtsystem entstand. Alle für eine jeweilige Funktion benötigten Daten konnten systemweit zur Verfügung gestellt werden. Aus dem zunehmenden Elektrifizierungsgrad ergaben sich heutige moderne Elektronikarchitekturen im Kfz in Form von seriellen Bussystemen. Das Akronym Bus stammt [2] nach von *Binary Utility System*, was auf ein drahtgebundenes Übertragungsmedium mit Anschluss an alle Systemkomponenten hinweist. Es werden komplexe Datenmengen über einzelne Leitungen bitweise übertragen, wodurch sich eine Vielzahl an Vorteilen ergibt: Der Verkabelungsaufwand sämtlicher elektrischer Leitungen wird minimiert, wodurch sich die Kosten, das Gewicht und die Fehleranfälligkeit reduzieren. Zudem wird eine Mehrfachnutzung von Informationen möglich, was die Anzahl der verbauten Sensoren senkt. Eine Diagnosefunktion wird umsetzbar und das Gesamtsystem ist flexibel für Änderungen und Erweiterungen. Die Kommunikation innerhalb eines Gesamtsystems, welches aus mehreren miteinander verknüpften Bussystemen bestehen kann, wird als *On-Board-Kommunikation* bezeichnet. Um eine übergeordnete Datenkommunikation einzelner Vernetzungsbereiche und Netzwerke zu erhalten, müssen die einzelnen Bussysteme mit unterschiedlichen Protokollen physikalisch und logisch miteinander verbunden werden. Diese Funktion wird von einem *Gateway* übernommen. Ein Gateway stellt sämtliche Daten netzwerkübergreifend zur Verfügung. Dabei kann die Funktion entweder in bereits vorhandene Steuergeräte integriert werden oder es kommen eigene zentrale oder dezentrale Gateway-Steuergeräte zum Einsatz. Bei einer *Off-Board-Kommunikation* stellt ein Gateway die Verbindung zwischen dem geschlossenen Gesamtnetz im Fahrzeug zu einem externen Gerät her [3].

Heute standardisierte und gängige Datenkommunikationssysteme sind in Tabelle 2.1 aufgeführt [3]. Auf die zur Anwendung wichtigste Form der Buskommunikation, dem *CAN-Bus*, wird an späterer Stelle näher eingegangen.
Neben dem heutzutage am häufigsten eingesetzten Bussystem (vgl. Unterabschnitt 2.2.2) haben sich aufgrund der speziellen Anwendungsfälle und der Eigenentwicklung unterschiedlichster Hersteller, vor allem aber zur Kostenreduzierung, weitere Systeme etabliert. Die kostengünstige Variante zur seriellen Datenübertragung *Local Interconnect Network* LIN weist eine vergleichsweise geringe Datenrate auf und wird daher mittlerweile lediglich in der Komfortelektronik als Kommunikationsschnittstelle zwischen Sensorik und Aktorik verbaut.Da als physikalisches Übertragungsmedium nur eine Eindrahtleitung zum Einsatz kommt, ist das Netzwerk relativ störanfällig, was eine Verwendung in sicherheitsrelevanten Bereichen ausschließt. Eine deutlich höhere Ausfallsicherheit, aber zugleich signifikant teurere Datenübertragung liefert der sog. *FlexRay*. Aufgrund der hohen Datenraten von bis zu 10 Mbit/s bietet dieses deterministische Feldbussystem ein hohes Potential für zeit- und sicherheitskritische Anwendungsfälle [4]. Für Multimediaanwendungen im Automobilbereich hat sich der *Media Oriented Systems Transport* MOST-Bus etabliert, der als Übertragungsmedium Lichtwellenleiter verwendet und damit sehr hohe Bitraten von bis zu 150 Mbit/s ermöglicht Ein MOST-Netzwerk ist in der Regel als Ringtopologie aufgebaut und liefert daher eine lediglich geringe Ausfallsicherheit [5].

Tabelle 2.1.: Klassifikation serieller Bussysteme [3]

Bussystem	Typische Anwendung	Maximale Datenrate	Übertragungs-medium	Sicherheits-anforderung
LIN	Komfort, Karosserie	20 kbit/s	Eindrahtleitung	gering
CAN (Low Speed)	Komfort, Karosserie	125 kbit/s	Verdrillte Zweidrahtleitung	hoch
CAN (High Speed)	Antrieb, Fahrwerk, Diagnose	1 Mbit/s	Verdrillte Zweidrahtleitung	hoch
FlexRay	Fahrwerk, X by Wire	10 Mbit/s	Verdrillte Zweidrahtleitung	sehr hoch
MOST	Infotainment	150 Mbit/s	Lichtwellenleiter	gering

2.2.1. Kommunikationsmodell

Um einen reibungslosen und nachvollziehbaren Datenaustausch zu gewährleisten, mussten mit dem Einzug der Bussysteme in der Automobilentwicklung auch einheitliche, herstellerübergreifende Kommunikationsstrukturen eingeführt werden. 1983 wurde der gesamte Datentransfer in einem Datennetz von der *International Organization for Standardization* ISO in sieben einzelne *Layer* (Schichten) unterteilt und die komplexe Kommunikationshierarchie beschrieben. Durch das in *DIN EN ISO/IEC 7498-1* [6] festgehaltene *Open System Interconnection* OSI-Schichtenmodell kann eine standardisierte und herstellerübergreifende Kommunikation im gesamten Busnetzwerk erzielt werden. Das OSI-Schichtenmodell wird in Tabelle 2.2 beschrieben. Für die Automobilindustrie und für Kfz-Anwendungen sind die grau hinterlegten Schichten noch nicht relevant. Wichtig sind vor allem die beiden untersten Layer *Physical* und *Data Link*. Diese Schichten werden an späterer Stelle genauer beschrieben.

Tabelle 2.2.: Zusammenfassung des OSI-Schichtenmodells aufgeteilt in Layer, Schicht und Funktionen [6]

	Layer	Schicht	Funktion
7	Application	Anwendung	Zugriff auf das Kommunikationssystem, Entkopplung Anwendung von Kommunikation
6	Presentation	Darstellung	Semantik, Datenkompression, Verschlüsselung, Übersetzer verschiedener Datenformate
5	Session	Sitzungssteuerung	Unterhalten längerer Sitzungen, Definition von Synchronisationspunkten
4	Transport	Datentransport	Verbindungsauf- und -abbau, Segmentierung, Sequenzierung, Assemblierung
3	Network	Vermittlung	Routing, Adressvergabe, Teilnehmererkennung und -überwachung
2	Data Link	Datensicherung	Botschaftsaufbau, Buszugriff, Flusskontrolle Fehlersicherung
1	Physical	Bitübertragung	Physikalische Busankopplung, Stecker, Übertragungsmedium, Leitungscodierung

2.2.2. Controller Area Network CAN

Das Bussystem *Controller Area Network* CAN wurde erstmals in den 1980er Jahren von der *Robert Bosch GmbH* präsentiert und gilt seit 1994 als offener Industriestandard. Mit der ISO-Norm *ISO 11898* wurde die CAN-Spezifikation international vereinheitlicht. Heute stellt CAN durch seine hohe Datenübertragungsrate und der geringen Fehleranfälligkeit die am weitesten verbreitete Kommunikationsspezifikation in der Automobilindustrie dar, kommt jedoch auch häufig in industriellen Anwendungen zum Einsatz. Aufgrund der daraus resultierenden hohen Stückzahlen an CAN-Controllern ergibt sich ein stetig sinkender Stückpreis für die zugehörigen Steuergeräte, was als weitere Stärke dieses Bussystems zu zählen ist. Die Steuergeräte, oder auch *Bus-Knoten* bezeichnet, sind in einem CAN-Netzwerk üblicherweise in Form einer Linientopologie nach der *Multi-Master*-Architektur miteinander verbunden. Jeder Knoten ist berechtigt, den Datentransfer auf dem Bus ereignisgesteuert anzustoßen [7, 8].

2.2.3. CAN-Protokoll: Physical Layer

Die unterste Schicht im OSI-Modell beschreibt die physikalische Busankopplung. Das Übertragungsmedium des CAN-Busses wird in den häufigsten Fällen als verdrillte Zweidrahtleitung, als sog. *Twisted-Pair-Leitung*, ausgeführt, wodurch sich die magnetischen Felder der beiden Leitungen weitestgehend gegenseitig neutralisieren. Eine hohe Datenrate und Busauslastung können zu Reflexionen im Bussystem führen. Um diesen unerwünschten Effekt zu minimieren, müssen die Enden der Busleitung mit einem Abschlusswiderstand versehen werden. Neben der physikalischen Busleitung kommen hardwareseitig weitere Bauteile wie der *Mikrocontroller* (μC), der CAN-Controller und der CAN-Transceiver zum Einsatz. Der Mikrocontroller verarbeitet die Kommunikationsdienste der höheren OSI-Schichten in der Kommunikationssoftware. Die grundlegenden Funktionen sind hingegen in den restlichen Bauteilen implementiert. Der CAN-Controller wickelt das Protokoll ab, während der Transceiver die physikalische Verbindung zum Übertragungsmedium herstellt. Der prinzipielle Aufbau eines CAN-Netzwerks wird in Abbildung 2.1 verdeutlicht.

Die physikalische Signalübertragung in einem CAN-Netzwerk basiert auf der Übertragung von Spannungsdifferenzen zwischen der *CAN-High*-Leitung (CANH) und der *CAN-Low*-Leitung (CANL). Wie in Tabelle 2-1 dargestellt, unterscheidet die ISO-Norm zwischen dem *Low-Speed-CAN* (Class B) und dem *High-Speed-CAN* (Class C). Der Low-Speed-CAN zeichnet sich durch seine auf 125 kbit/s begrenzte Datenrate aus. Dadurch findet er häufig in Komfortsystemen wie Klimasteuergeräten Anwendung. Die Signale werden über nominelle Potentiale auf dem Bus übertragen. Beim High-Speed-CAN hingegen werden differentielle Potentiale verwendet. Dieser besitzt eine maximale Datenrate

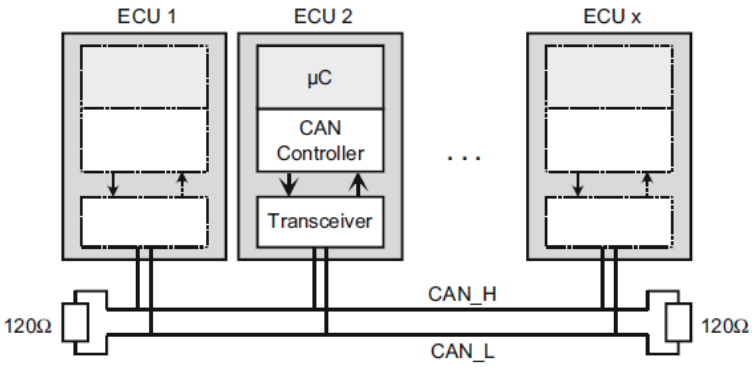


Abbildung 2.1.: CAN-Netzwerk: Ein einzelner CAN-Knoten besteht aus einem Mikrocontroller, einem CAN-Controller und einem CAN-Transceiver. Der Abschlusswiderstand unterdrückt Busreflexionen [8].

von bis zu 1 Mbit/s und eignet sich daher für zeitkritische Anwendungen wie Antriebs- und Fahrdynamikregelung. Die unterschiedlichen Signalpegel werden in Abbildung 2.2 erläutert. Aufgrund der Busankopplung ermöglicht der Low-Speed-CAN zusätzliche Mechanismen zur Fehlererkennung. Bei Ausfall einer Leitung bleibt er betriebsfähig und gilt daher gegenüber dem High-Speed-CAN als fehlertoleranter [8, 9].

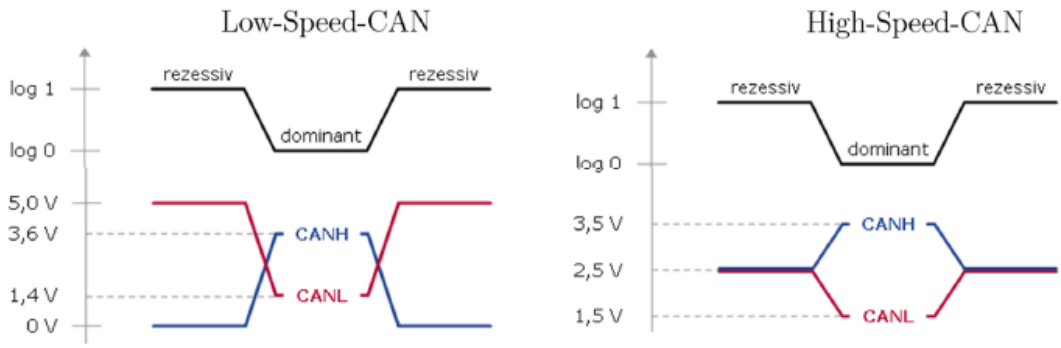


Abbildung 2.2.: Signalpegel Low-Speed-CAN (links) und High-Speed-CAN (rechts) [9].

2.2.4. CAN-Protokoll: Data Link Layer

Der *Data Link Layer* beschreibt das Zugriffsverfahren und den strukturellen Aufbau eines CAN-*Frames*. Unter einem Frame versteht man den gesamten Datenrahmen, der in einer einzelnen Botschaft über den CAN-Bus übermittelt wird. Zwischen den Begriffen Frame, Botschaft und Nachricht wird nachfolgend keine Unterscheidung getroffen.

Typischerweise gibt es mehrere Arten von CAN-Frames, die in verschiedene Kategorien unterteilt werden:

- *Data Frame* zur Übertragung von Nutzdaten.
- *Remote Frame* zur Anforderung von Nutzdaten (also Data Frames) von beliebigen CAN-Knoten. Setzt sich bis auf das fehlende Data Field wie ein Data Frame zusammen.
- *Error Frame* zur Signalisierung entdeckter Fehler während des Kommunikationsbetriebs. Mit dem Übertragen eines Error Frames geht der Abbruch der laufenden Botschaftsübertragung einher.

Das Botschaftenformat eines Data Frames ist in Abbildung 2.3 dargestellt [9]. Die einzelnen Komponenten und deren Aufgaben werden nachstehend in der zugehörigen Tabelle 2.3 näher erklärt.

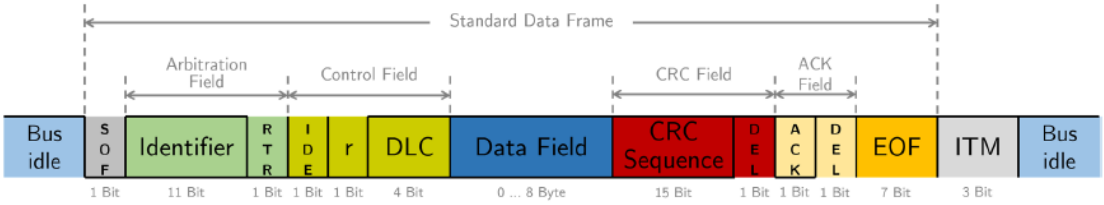


Abbildung 2.3.: Aufbau des Standard CAN Data-Frames [9].

Während der gesamten Laufzeit der Netzwerkverbindung senden die Busknoten ihre Botschaften nach dem *Broadcast*-Verfahren unaufgefordert und ereignisgesteuert auf den Bus. Jeder Empfänger einer Botschaft entscheidet, ob der Inhalt relevant ist und er damit die Nachricht auswertet oder ignoriert. Diese *Akzeptanzfilterung* findet auf Basis der inhaltsbasierten Adressierung eines Frames statt. Hierzu kennzeichnet der *Identifier* ID den spezifischen Inhalt einer Botschaft. Der Identifier ist in einem herkömmlichen CAN-Protokoll 11 Bit lang. Somit lassen sich bis zu $2^{11} = 2.048$ Botschaften eindeutig unterscheiden. Aufgrund der stetig wachsenden Komplexität moderner Busarchitekturen besteht zudem die Möglichkeit, einen Identifier im Extended-Format aufzubauen. Der ID umfasst dann 29 Bit, wodurch $2^{29} = 536.870.912$ unterschiedliche Botschaften differenziert werden können.
Durch den ereignisgesteuerten Kommunikationsaufbau kann es häufig vorkommen, dass mehrere Busteilnehmer eine Botschaftsübertragung zum gleichen Zeitpunkt beginnen möchten. Da eine Übertragung aber nicht unterbrochen werden kann und oftmals essentielle sicherheitsrelevante Nachrichten unmittelbar mit vergleichsweise marginalen Botschaften versendet werden, ist es von hoher Bedeutung, sämtliche Botschaften mit Prioritäten zu versehen. Es wird das *Carrier Sense Multiple Access with Collision Avoidance*

Tabelle 2.3.: Funktionen der einzelnen Felder im Data-Frame [7, 8].

Feld	Name	Länge	Funktion
kein Feld	SOF	1 Bit	Der dominante *Start of Frame* signalisiert den Start eines Frames. Durch den Wechsel von einem rezessiven (Buspegel im Ruhezustand) auf ein dominantes Bit findet eine netzwerkweite Synchronisation statt.
Arbitration Field	Identifier	11 Bits	Der Identifier dient der bitweisen Arbitrierung und beschreibt die logische Adressierung der Botschaft.
	RTR	1 Bit	Der *Remote Transmission Request* kennzeichnet den Frametyp. Im Fall des Data Frames wird das Bit dominant übertragen.
Control Field	IDE	1 Bit	Die *Identifier Extension* legt fest, ob der Identifier im Standard-Format (11 Bit) oder im Extended-Format (29 Bit) vorliegt.
	r	1 Bit	Dieses Bit ist für künftige Erweiterungen reserviert.
	DLC	4 Bits	Der *Data Length Code* übermittelt die Anzahl der nachfolgenden Nutzbytes.
Data Field	Data Field	0...64 Bits	Im Datenfeld werden bis zu acht Bytes an Nutzdaten übertragen. Es beinhaltet die zu übermittelnden Signalwerte und Nutzinformationen.
CRC-Field	CRC	15 Bits	Der *Cyclic Redundancy Check* bildet eine Prüfsumme der Nutzdaten und dient der Fehlererkennung bei der Botschaftsübertragung.
	DEL	1 Bit	Auf das CRC-Feld folgt ein rezessives *Delimiter*-Bit.
ACK-Field	ACK	1 Bit	Der Sender einer Botschaft setzt das *Acknowledgement*-Bit auf rezessiv, der Empfänger quittiert das korrekte Ergebnis des CRC-Feldes mit einem dominanten Bit.
	DEL	1 Bit	Das ACK-Feld wird ebenfalls durch ein *Delimiter*-Bit begrenzt.
kein Feld	EOF	7 Bit	Der *End of Frame* markiert das Ende einer Botschaft mit sieben rezessiven Bits.

CSMA/CA-Verfahren angewendet. Durch eine *bitweise Arbitrierung* wird dafür gesorgt, dass bei einer Kollision auf dem Bus die Botschaft mit der höchsten Priorität weiterhin übertragen wird. Die höchst priorisierte Botschaft weist immer den niedrigsten nominellen ID auf. Durch den Vergleich der dominanten und rezessiven Buspegel ist jeder Knoten darüber informiert, welche Botschaft gerade auf dem Bus übertragen wird. Stellt

ein Knoten der aktuell sendet fest, dass der Pegel auf dem Bus nicht seiner gesendeten Botschaft entspricht, stellt dieser umgehend den Sendevorgang ein. Die Echtzeitfähigkeit des Gesamtsystems wird somit nicht gefährdet und Botschaften mit hoher Priorität bleiben deterministisch.
Neben einer korrekten Zugriffsfolge sind besonders die Datensicherung und eine zuverlässige Fehlererkennung von großem Stellenwert. Hierzu kommen Mechanismen wie der *Cyclic Redundancy Check* CRC, *Acknowledgement* ACK oder *Bitstuffing* zum Einsatz. In der CRC-Sequenz wird aus dem Frame ein Polynom gebildet, welches durch Division mit einem Modulo-Operator eine Prüfsumme ergibt. Der Empfänger berechnet bei Erhalt einer Botschaft ebenfalls eine CRC-Prüfsumme. Stimmen die erhaltenen und berechneten Sequenzen überein, war der strukturelle Aufbau der Botschaftsübertragung korrekt. Mindestens ein Empfänger bestätigt den positiven Erhalt, indem er das ACK-Bit auf einen dominanten Pegel setzt. Liegt ein ACK-Fehler vor, wurde also entweder vom Sender ein Fehler verursacht oder es befinden sich keine weiteren Teilnehmer am Bus.
Die Bitstuffing-Methodik dient der zeitlichen Nachsynchronisation während der Übertragungslaufzeit. Die Synchronisation der Busteilnehmer erfolgt eigentlich über den Flankenwechsel zwischen unterschiedlichen Bits. Liegen jedoch mehrere homogene Bits hintereinander vor, kann diese Synchronisation nicht stattfinden. Um eine korrekte Übertragung zu gewährleisten, fügt der Sender bis zum Ende der CRC-Sequenz nach fünf homogenen Bits immer ein komplementäres Bit ein. Der Empfänger entfernt seinerseits diese Stuff-Bits, bevor das empfangene Datenpaket ausgewertet wird. Wird von einem Empfänger eine Folge von mehr als fünf homogenen Bits erkannt, liegt ein Bitstuffing-Fehler vor. Bei der Übertragung eines Data Frames mit 11 Bit Identifier kann der gesamte Botschaftsrahmen also um 29 Bit[1] auf bis zu 132 Bit anwachsen.
Wurde einer der genannten Fehler erkannt, wird ein Fehlersignal gesetzt, welches aus sechs dominanten Bits besteht und damit bewusst gegen die Bitstuffing-Regel verstößt. In diesem Fall wird ein aktives *Error Flag* in einem Error Frame von dem Knoten versendet, welcher den Sendefehler zuerst detektiert hat. Die beteiligten CAN-Knoten stellen den Sendevorgang umgehend ein. Wird ein Fehler eines Knoten mehrfach detektiert, kann diesem das Senden verweigert werden. Zur Sicherstellung der systemweiten Datenkonsistenz und zur Reduzierung der Busauslastung kann ein fehlerhafter Knoten vollständig von der Buskommunikation ausgeschlossen werden [7, 8, 9].

[1] im Worst Case bei acht Datenbytes

3. Ausgangssituation

3.1. Hardware

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

3.2. Software

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

3.3. Sensorik

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

4. Diagnosesystem

Dieser Kapitel soll detailliert die zentrale Aufgabe der Arbeit beschreiben. Es solle Verfahren zu Diagnose des EVObots entwickelt und erstellt werden, um im Fahrbetrieb die internen Zustands- und Sensordaten echtzeitnah darzustellen und somit die Applikation des Fahralgorithmus zu erleichtern. Zunächst wird die Bedeutung der Diagnose eines komplexen Fahrzeugsystems mit vernetzten Systemen aufgezeigt und ein Konzept zur Fehlerdiagnose am Modellfahrzeug aufgebaut. Nachfolgend wird das Vorgehen und der Aufbau eines Diagnosesystems ausführlich beschrieben. Abschließend wird die Diagnosefunktion validiert und die gewonnen Ergebnisse betrachtet.

4.1. Konzept der Fehlerdiagnose

Aktuelle Fahrzeugsystemen sollten selbstredend stets fehlerfrei und ohne Probleme agieren und die vom Fahrer gewünschten Funktionen vollständig umsetzten. Trotzdem kann es unter gewissen Umständen und äußeren Voraussetzung zu einem Fehlverhalten kommen. Ein Fehler kann im Betrieb eines mechatronischen Systemverbund sowohl mechanisch, elektronisch, als auch softwareseitig auftreten. Durch den stetigen Anstieg des Softwareanteils und der sicherheitskritischen Prozesse im automotiven Bereich steigt die Gefahr eines Softwarefehlers oder einer Software-Anomalie. Ein Fehler wird dabei allgemein nach *EN ISO 9000:2005* [10] als „Nichterfüllung einer Anforderung“ oder nach *DIN 55350* [11] genauer als „eine unzulässige Abweichung eines Merkmals von einer vorgegebenen Forderung“ beschrieben.
Ein Softwarefehler lässt sich allgemeingültig entweder durch physikalische beziehungsweise chemische Effekte oder durch menschliche Fehlermechanismen wie Denkfehler, Verständnisfehler, Interpretationsfehler oder simple Tippfehler beschreiben. Dabei kann man diese Beschreibung eines Fehlers grundsätzlich in zwei Kategorien einordnen: Man spricht von einem physikalischen Fehler, wenn einzelne Komponenten oder Teilsysteme ausfallen, oder von einem funktionalen Fehler, wenn ein System ausführbar ist, seine Funktion aber nicht korrekt umgesetzt wird [12]. Ein funktionaler Fehler ist offenbar in den meisten Fällen das Ergebnis von Design-Mängeln und demnach auf eine menschliche Ursache zurückzuführen. Eine Fehlhandlung in der Umsetzung einer Softwareanwendung

führt in aller Regel zu einem Fehlerzustand bei der Programmausführung. Dieser Fehlerzustand kann unter Umständen auch unerkannt bleiben. Tritt der Fehlerzustand jedoch aus Programmsicht nach Außen auf, spricht man von der Fehlerwirkung, die für den Anwender wahrnehmbar ist. Diese Fehlerwirkung kann sich je nach Schwere des Fehlers als ein abweichender Rückgabewert einer Berechnung oder bis hin zum Totalausfall des Systems äußern [13].
Im Sinne der Zuverlässigkeit gilt es also, das Auftreten eines Fehlerzustandes in einem sicherheitskritischen System zu vermeiden oder besser einen sicheren Zustand im Fehlerfall einzunehmen. Ein übergeordnetes Beispiel eines *Fail-Safe*-Zustandes kann aus der Technologie des hochautomatisierten Fahrens gegeben werden: Fällt das Kamerasystem zur Fahrspurerkennung unerwartet aus, soll ein automatisiert fahrendes Fahrzeug nicht unkontrolliert weiterfahren, sondern die Fahrgeschwindigkeit reduzieren und auf Basis der verfügbaren Streckendaten am rechten Fahrbahnrand zum Stehen kommen, um einen potentiellen Schaden zu minimieren.

Die Komplexität der vernetzten Funktionen und Systeme erfordert eine umfassende Kommunikation der Softwarekomponenten und ebenso ausgereifte Methoden zur Diagnose möglicher Fehler. Der Begriff *Diagnose* stammt eigentlich aus der Medizin, lässt sich aber [14] nach im Bezug auf Fahrzeuge wie folgt definieren: „Aus konkreten und diffusen Symptomen, die der Fahrer schildert, wird im Service unter Zuhilfenahme der Diagnosesysteme ein exaktes Fehlerbild erstellt und es werden geeignete Reparaturmaßnahmen eingeleitet.“ Der Ablauf einer klassischen Fehlerdiagnose lässt sich anhand dem Modell 4.1 veranschaulichen. Ein Diagnosesystem, welches mittels Symptome Fehlerur-

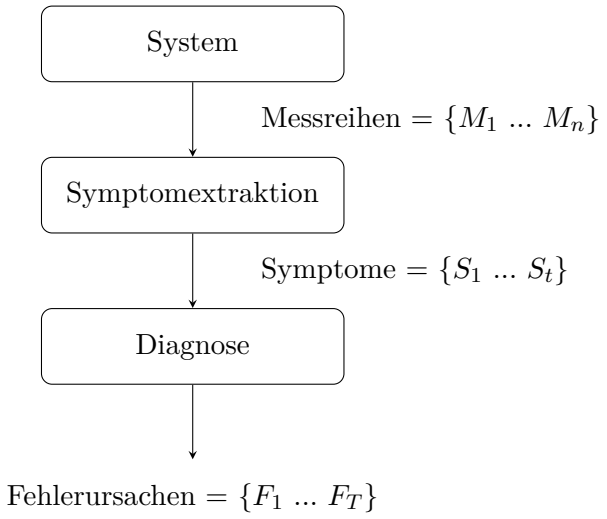


Abbildung 4.1.: Ablauf einer Diagnose im Kfz-Umfeld [15]

sachen diagnostiziert, kann also formal als eine Abbildung von Fehlersymptomen S auf Fehlerursachen F angesehen werden

$$\{S_1 \ ... \ S_t\} \rightarrow \{F_1 \ ... \ F_T\}.$$

Kommt es zu einem Fehlerfall oder Systemausfall, ist die Ursache in einem komplexen und verteilten Systemnetzwerk meist schwer auszumachen. Aufgrund dessen wurden bereits früh in der Entwicklung elektronischer Datenkommunikation Diagnosesysteme als Analysewerkzeug eingeführt. Mit einem aus Hard- und Software bestehenden Diagnosesystem kann die Datenbuskommunikation aufgezeichnet und diagnoserelevante Informationen über den Zustand der Teilkomponenten zu einem externen Testgerät übertragen und ausgewertet werden. Ein Diagnosesystem gilt damit als umfangreiches Werkzeug zur schnellen Fehlererkennung und Fehleranalyse. Während dem Entwicklungsprozess lässt sich ein Diagnosesystem auch nutzen, um über die Diagnosekommunikation die Steuergeräte-Applikation durchzuführen. Dabei ist es nicht nur nötig die reine Datenübertragung einheitlich zu standardisieren, sondern ebenfalls die Applikationsschicht der Protokolle (Vgl. Tabelle 2.2) zu normieren. Dies bietet die Möglichkeit, den Aufwand und die Pflege der Diagnoseschnittstellen und den Diagnosetestern zu begrenzen.
Um eine einheitliche Diagnosekommunikation zu schaffen, wurden seit den 1990er Jahren diverse *Diagnoseprotokolle* entwickelt, die zunächst teils proprietär und inkompatibel zueinander umgesetzt waren, inzwischen jedoch meist herstellerübergreifend genormt sind. Als heute gängige Diagnoseprotokolle sind das *Keyword 2000 Protokoll* KWP 2000, die *Unified Diagnostic Services* UDS oder die *On-Board-Diagnose* OBD zu nennen. Für weiterführende Informationen der genannten Standards sei auf [8] und [16] verwiesen. Die verschiedenen Protokolle einer Diagnosekommunikation lassen sich auf die einzelnen Schichten des bekannten ISO/OSI-Referenzmodells anwenden. Das eigentliche Diagnoseprotokoll umfasst dabei abhängig von seiner Ausprägung und dem expliziten Anwendungsfall die Schichten fünf bis sieben. Die Übertragung der Diagnosedaten erfolgt durch entsprechende Protokolle für das angewendete physikalische Kommunikationsmedium in den Schichten eins bis vier.

4.1.1. Anforderungen an die Fehlerdiagnose

Um ein Diagnosesystem für das Demonstratorfahrzeug mit angemessenen Mitteln und Werkzeugen umzusetzen, wird zunächst der gewünschte Funktionsumfang beschrieben und darauf aufbauend sämtliche Anforderungen an das System definiert. Dazu ist in Abbildung 4.2 ein Use-Case-Diagramm in der grafischen Modellierungssprache UML für den konkreten Anwendungsfall der Diagnosefunktion dargestellt. Hieraus lassen sich leicht die Spezifikationen und geforderte Funktionalitäten an die spätere Diagnose ableiten.

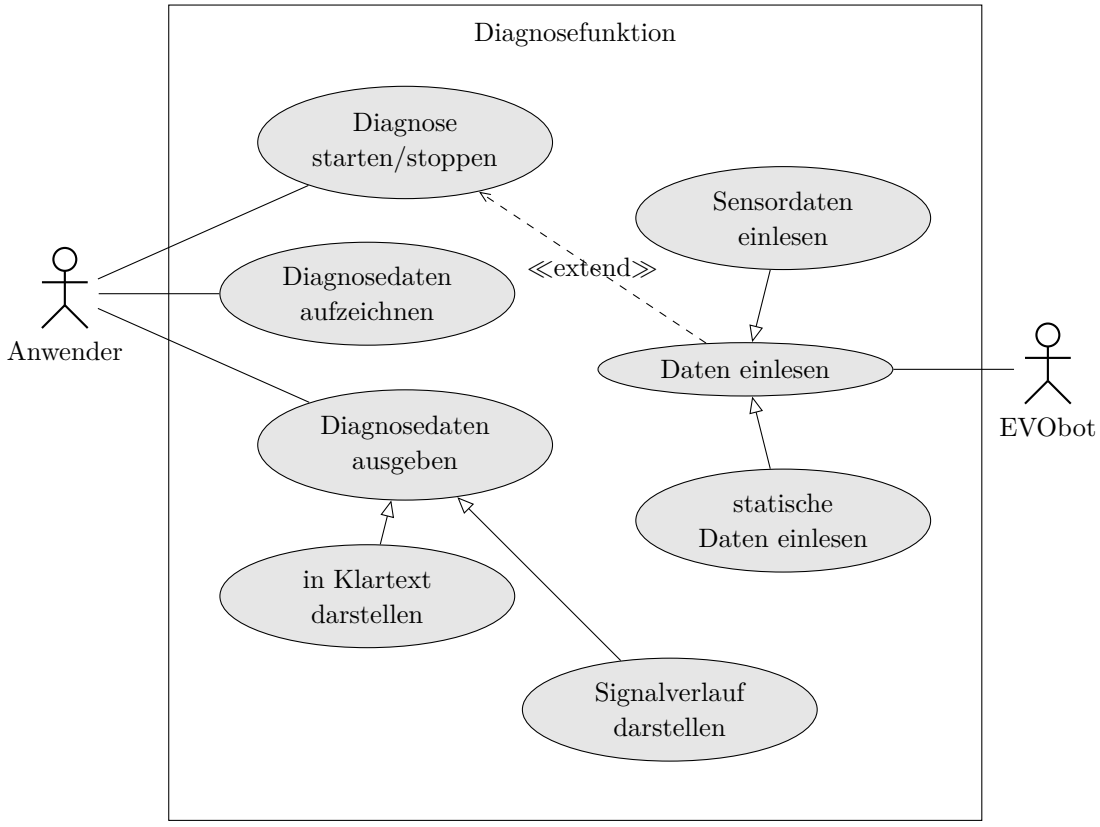


Abbildung 4.2.: Use-Case-Diagramm einer Diagnosefunktion für den Fahrdemonstrator

Das Gesamtsystem besteht aus Fahrzeug, Diagnosewerkzeug und Anwender. Aus dem Use-Case-Diagramm ergeben sich die funktionalen Anforderungen wie folgt:

- Das Diagnosesystem muss die Diagnosedaten innerhalb der Programmlaufzeit an ein Ausgabegerät übermitteln.
- Das Diagnosesystem muss die Diagnosedaten auf einem externen Ausgabegerät darstellen.
- Das Diagnosesystem muss die Daten sowohl in Klartext ausgeben, als auch die einzelnen Signalverläufe grafisch darstellen.
- Die Diagnosefunktion muss vom Anwender gestartet und gestoppt werden können.
- Das Diagnosesystem darf im inaktiven Zustand keine Daten vom Fahrzeug übertragen, um die Systemauslastung gering zu halten.

Weiterhin lassen sich folgende, nichtfunktionale Anforderungen stellen:

- Die Umsetzung der Datenkommunikation soll dem Vorbild eines realen und aktuellen Kraftfahrzeuges entsprechen.
- Die Datenkommunikation darf kabelgebunden oder kabellos stattfinden.
- Das Diagnosesystem muss ein echtzeitnahes Daten-Monitoring ermöglichen.
- Das Diagnosesystem soll eine übersichtliche Visualisierung für den Anwender bieten.
- Die Einbindung in das bestehende System soll ohne grundlegende Modifikationen in den implementierten Algorithmen möglich sein.
- Die Datenkonsistenz und Datensicherung bei der externen Kommunikation muss gewährleistet sein.
- Die Diagnosefunktion muss für spätere Ergänzungen erweiterbar sein.
- Die Programmierschnittstelle muss offen und modular sein.

4.1.2. Möglichkeiten zur Umsetzung einer Fehlerdiagnose

Die Umsetzung einer Diagnosefunktion und die Einbindung dieser Funktion in das Gesamtsystem auf dem Betriebssystem des Demonstratorfahrzeuges lässt sich grundsätzlich mit verschiedenen Mitteln und Möglichkeiten erzielen. Die bedingt voneinander entkoppelten Teilfunktionen umfassen:

- Sammeln der Sensordaten und Nutzinformationen im Gesamtsystem von ROS.
- Bereitstellung dieser Daten in einer zur Übertragung geeigneten Form.
- Echtzeitnahe Übermittlung der Diagnosedaten an ein externes Ausgabegerät.
- Analyse und Visualisierung der Daten auf dem Ausgabegerät.

Ein Ansatz zur möglichen Umsetzung besteht darin, sämtliche im bereits implementierten Quelltext des Software-Framework ROS berechneten und vorhandenen Wertevariablen und Sensordaten während der Programmlaufzeit drahtlos durch ein lokales Netzwerk zu übermitteln. Hierzu eignen sich diverse Industriestandards für Funknetzwerke nach *IEEE 802* wie Bluetooth oder WLAN. Ein großer Vorteil der Bluetooth-Technologie liegt in dem Verbindungsaufbau der Kommunikationspartner. Die Bluetooth-Knoten müssen sich lediglich in Reichweite befinden, um eine automatische Verbindung aufzubauen, ohne, dass eine Anwenderinteraktion nötig wird. Damit eignet sich Bluetooth für einen Datenaustausch mit autonomen Systemen. Um eine mögliche Bluetooth-Verbindung

mit dem *NVIDIA Jetson*-Modul aufzubauen, ist lediglich ein geeigneter Bluetooth-Adapter nötig. Die Bluetooth-Technologie eignet sich zwar für eine einfache, drahtlose Punkt-zu-Punkt-Datenkommunikation und weist eine allgemeine Reichweite von circa 10 Metern auf, die Reichweite ist jedoch stark abhängig von der Sendeleistung und der Empfindlichkeit des eingesetzten Transceivers. Zusätzlich können die Eigenschaften der Umgebung die erzielbare Sende- und Empfangsreichweite negativ beeinflussen, wodurch eine konsistente Datenverbindung je nach Einsatzumgebung des Fahrdemonstrators nicht gewährleistet ist. Kommt es bei einer Datenverbindung auf hohe Übertragungsraten und erhöhte Sicherheit an, eignet sich eher die Verbindung durch WLAN in einem drahtlosen lokalen Netzwerk. Diese Übertragung ist jedoch explizit an einen stationären Ort gebunden. Fällt das Netzwerk aus, kann also keine Datenkommunikation stattfinden. Zudem ist die Latenz bei einer WLAN-Verbindung nicht kontrollierbar und kann unter Umständen so hoch sein, dass eine Echtzeitanwendung nur mit erhöhtem Aufwand umsetzbar ist.
Um dem zentralen Anspruch gerecht zu werden, die Datenkommunikation vergleichbar zu der eines realen Fahrzeuges aufzubauen, liegt es nahe, den Datenaustausch zwischen dem Fahrzeug und einem externen Ausgabegerät durch ein in der Fahrzeugtechnik übliches serielles Bussystem zu realisieren. Hierzu eignet sich vor allem das CAN-Protokoll, das durch seine relativ hohen Datenraten, das deterministische Verhalten, eine fehlertolerante Datenübertragung und die robusten Datensicherungsmaßnahmen als die am häufigsten im Kfz eingesetzte serielle Kommunikationstechnologie gilt. Ein weiterer Aspekt, der für eine Verwendung des CAN-Protokolls spricht, ist das bereits vorhandene Vorwissen zu Bussystemen und die Verfügbarkeit sämtlicher nötiger Tools und Anwendungen im Unternehmen. Da sowohl diverse CAN-Interfaces, als auch die nötigen Softwarelizenzen unternehmensintern zur Verfügung stehen, wird die Realisierung der Diagnosefunktionalität mit einem seriellen Bussystem als besonders geeignet betrachtet. Daher wird der Ansatz verfolgt, eine drahtgebundene Off-Board-Kommunikation über eine physikalische Busleitung zwischen dem Fahrzeug und einem externen Laptop aufzubauen, um die Fahrzeugzustandsdaten online[2] im Fahrbetrieb zu analysieren und eine Datenaufzeichnung zu ermöglichen. Die Menge der zu übermittelnden Diagnosedaten hängt dabei selbstredend von der Komplexität der umgesetzten Funktionalitäten und der verwendeten Sensorik des autonomen mobilen Roboters ab. Nach einer ersten Sichtung der implementierten Algorithmik kann bereits festgehalten werden, dass eine Übertragung größerer, zusammenhängender Datenmengen nicht nötig ist. Es genügt, die internen Zustands- und Sensordaten in Form einzelner Signale gebündelt in zyklischen Busbotschaften zu übermitteln. Eine Segmentierung der Datenmengen, wie es bei einem *Transportprotokoll* in Kfz-Anwendungen auf der Transportschicht im ISO/OSI-Schichtenmodell (vgl.

[2] Als *online* wird hier nicht die aktive Verbindung zum Internet bezeichnet, sondern die intakte und betriebsbereite Datenverbindung zwischen EVObot und dem externen Ausgabegerät.

Tabelle 2.2) üblich ist, wird daher nicht nötig sein. Aufgrund dessen wird bei Implementierung der Diagnosefunktion bewusst auf die Umsetzung der Schicht vier des Referenzmodells verzichtet. Der Verzicht auf die Verwendung eines normierten Transportprotokolls impliziert eine weitere Einschränkung des Diagnosesystems: Üblicherweise werden im Fahrzeugbetrieb erkannte Fehlfunktionen und Störungen durch einen Eintrag im Fehlerspeicher des Steuergerätes dokumentiert. Diese Einträge weisen neben den nach dem Diagnoseprotokoll normierten Fehlercodes und einer Fehlerbeschreibung zusätzliche situationsabhängige Umgebungsdaten und zeitliche Informationen auf. Da die zu implementierende Diagnosefunktion lediglich ein echtzeitnahes Daten-Monitoring ermöglichen soll und eine Speicherung, respektive Auswertung der Diagnosedaten nur auf dem externen Diagnoserechner nötig ist, wird im weiteren Verlauf auf die formale Definition eines Fehlerspeichers und der Regeln zur Fehlerablage verzichtet. Demzufolge steht die strikte Einhaltung des standardisierten Datenmodells für den Austausch diagnoserelevanter Daten nach dem *Open Diagnostic Data Exchange* ODX-Standard nicht im Fokus der Arbeit.

4.2. Aufbau einer Datenkommunikation auf CAN-Bus

Neben der softwareseitigen Umsetzung einer Datenkommunikation spielt die physikalische Busanbindung auf dem EVObot entsprechend der Bitübertragungsschicht im ISO/OSI-Schichtenmodell eine zentrale Rolle.

4.2.1. Fahrzeugseitige Busanbindung

Zur Übertragung einzelner CAN-Botschaften vom Modellfahrzeug auf einen externen Diagnoserechner stehen mehrere Möglichkeiten zur Auswahl. Zum einen lässt sich die Hardwareanbindung über einen Arduino realisieren. Auf dem EVObot ist bereits ein Arduino Uno verbaut, der das Eingangssignal des Ultraschallsensors einliest und zum Ansteuern der Aktorik eingesetzt wird. Durch die Erweiterung des Arduinos um ein *CAN-Bus Shield* lässt sich ein feldbusfähiges Kommunikationssystem aufbauen. Ein CAN-Shield erweitert den herkömmlichen Mikroprozessor der Arduino-Plattform um einen CAN-Controller mit einer SPI-Schnittstelle und einem geeigneten CAN-Transceiver. Die verfügbaren CAN-Bus Shields implementieren den üblichen CAN High Speed Standard mit einer maximalen Datenrate von 1 Mbit/s, was die Anforderungen an die Datenübertragung in der späteren Verwendung in jedem Fall erfüllt. Es können Identifier sowohl im 11 Bit Standard-Format, als auch im 29 Bit Extended-Format verarbeitet werden. Zudem liefert eine solche Erweiterungsplatine umfangreiche Programmbibliotheken, um die Verwendung einer CAN-Anbindung in der Arduino Entwicklungsumgebung zu ermög-

lichen. Es lassen sich sämtliche diagnoserelevante Daten von dem Hauptrechner, dem NVIDIA Jetson TX2, über die serielle Schnittstelle zyklisch an den Arduino senden. Auf diesem werden die Informationen in das gängige CAN-Format gepackt und physikalisch auf den Bus gesendet. Da es jedoch ein explizites Projektziel ist, künftig auf die Verwendung des Arduinos zu verzichten, wäre es nicht zielführend, die Diagnosekommunikation hardwareseitig durch einen Arduino mit einem Erweiterungsboard zu realisieren.

Aufgrund dessen wird die Busanbindung direkt über das NVIDA Jetson TX2 realisiert. Dieses bietet gegenüber dem Vorgängermodell Jetson TX1 den entscheidenden Vorteil, bereits ein CAN-Interface mit zwei Controllern auf dem Mainboard verbaut zu haben. Dadurch kann die Busanbindung direkt auf dem Board umgesetzt werden, sodass kein Routing der Busbotschaften über einen Arduino nötig wird. Das interne Interface unterstützt ebenfalls übliche Datenraten bis zu 1 MBit/s und ist sowohl für Identifier im Standard-Format, als auch im Extended-Format ausgelegt. Abbildung 4.3 ist die Anschlussbelegung der CAN Architektureinheit des Tegra-Chips auf dem Jetson TX2 zu entnehmen. Es sind je eine Sendeleitung (Tx) und eine Empfangsleitung (Rx) dargestellt. Damit lässt sich ein Netzwerk mit zwei unabhängigen Busleistungen implementieren. Darüber hinaus bietet die Architektur eine Einbindung von Fehlerdetektionsleitungen zur Sicherstellung netzweiter Datenkonsistenz und weitere Signalleitungen für abstraktere Busanwendungen.

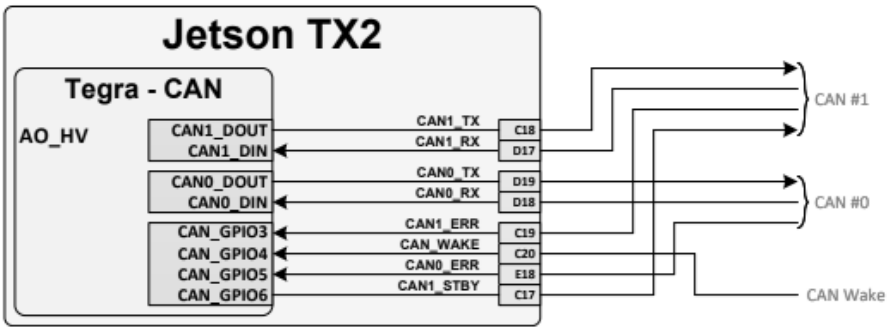


Abbildung 4.3.: Layout des CAN-Interfaces auf dem NVIDIA Jetson TX2 nach dem NVIDIA Product Design Guide [17]

Das Jetson TX2 verfügt zwar über einen integrierten Controller, um eine Busbotschaft physikalisch zu versenden oder empfangen und korrekt zu interpretieren, ist jedoch zusätzlich ein externer CAN-Transceiver nötig. Der CAN-Controller wickelt das Protokoll ab, während der Transceiver die physikalische Verbindung zu CANH und CANL darstellt. Möchte ein CAN-Knoten auf den Bus senden, erhält der Transceiver die Businformationen vom μC über den Controller (vgl. Abbildung 2.1). Der Transceiver setzt die empfangenen Daten in einen Spannungspegel um, sodass ein differentielles Spannungssi-

gnal auf die Busleitung gesendet wird. Beim Empfangen wandelt er die Spannungspegel dagegen in für den Controller verwertbare Signale um.
Nach einer Recherche zu geeigneten CAN-Transceivern fiel die Entscheidung auf das Modell *SN65HVD230* der Firma *Texas Instruments Incorporated*. Ein Auszug des frei verfügbaren Datenblattes ist Anhang A.1 zu entnehmen. Dieser Transceiver ist bereits auf einem Breakout Board der Firma *Waveshare* montiert und als Steckmodul verfügbar. Zur Spannungsversorgung des Transceivers sind 3,3 V nötig, die über einen GPIO-Port des Jetson TX2 abgegriffen werden können. Die CANH- und CANL-Leitungen können mit einer Schraubklemme mit dem Steckmodul verbunden werden. Der Transceiver wird wie in Abbildung 4.4 dargestellt mit dem Jetson-Board an der GPIO-Steckleiste J26 verbunden. Zusätzlich werden zwei LEDs angekoppelt, um das Senden und Empfangen von CAN-Botschaften zu visualisieren.

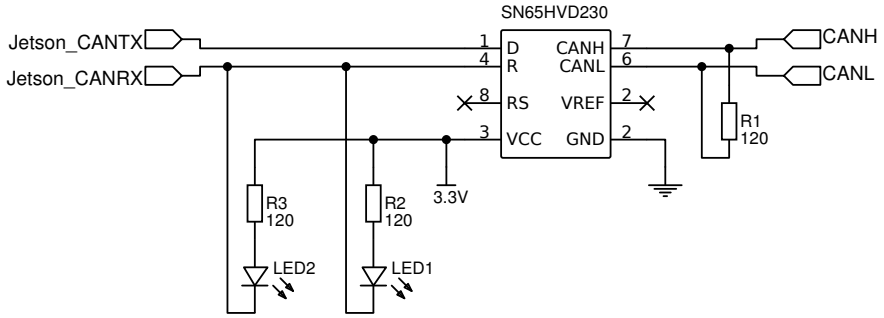


Abbildung 4.4.: Schaltplan des SN65HVD230 CAN-Transceivers

Um CAN-Botschaften über das Jetson TX2 zu versenden, sind zunächst einige grundlegende Einstellungen und Änderungen im Betriebssystem vorzunehmen. An erster Stelle muss ein geeigneter Treiber installiert und aktiviert werden, der das CAN-Interface unterstützt. Hierfür kommt das Modul *MTTCAN* der Firma *Robert Bosch GmbH* zum Einsatz. Der Treiber legt die Konfiguration der logischen Verschaltung auf dem Mikroprozessor des Jetson TX2 fest und ermöglicht damit die protokollkonforme Verarbeitung der Datenübertragung über CAN-Bus. Es lassen sich empfangene Busnachrichten filtern und nach ihrer Priorität sortieren, um dadurch die Interrupt-Last zu reduzieren. Nach einem Rebuild des Kernels und einem Hinzufügen des Kernelmoduls zur Laufzeit des Systems ist der Treiber aktiv. Bei der Überprüfung der Systemkonfiguration werden nun beide CAN-Controller als korrekt konfiguriert gelistet.
Um mit der CAN-Konfiguration auf dem Jetson TX2 arbeiten zu können, wird die generische Programmierschnittstelle *SocketCAN* eingesetzt. Diese API stellt eine Sammlung von diversen Netzwerktreibern und einer eigenen Netzwerkschicht, die ursprünglich von der Konzernforschung der *Volkswagen AG* als Open Source Projekt für die Verwendung

von CAN unter Linux entwickelt wurde. Es beinhaltet dabei Treiber zum Aufbau verschiedener Schnittstellen als Sockets und lehnt sich somit an das *TCP/IP*-Protokoll an, das zur Netzwerkprogrammierung verwendet wird. SocketCAN erzeugt ein neues virtuelles Netzwerkgerät auf dem Betriebssystem und ermöglicht damit, mehreren Anwendungen gleichzeitig auf CAN-Funktionen zugreifen zu können. Es können ebenfalls mehrere CAN-Netzwerke von einzelnen Anwendungen parallel genutzt werden. Damit bietet SocketCAN eine benutzerfreundliche Umsetzung des zyklischen Versenden von Diagnosebotschaften auf dem CAN-Bus. Zusätzlich liefert die offene Programmierschnittstelle flexible Erweiterungen zum Einsatz von Transportprotokollen um auch größere Datenmengen segmentiert auf einzelne Busbotschaften versenden zu können. Dies wird jedoch wie bereits in Abschnitt 4.1.2 erläutert aufgrund der begrenzten Anzahl der Zustands- und Sensordaten für die gewünschte Anwendung nicht nötig. Ein Mehrwert für die Diagnosefunktion bietet jedoch der Broadcast-Manager von SocketCAN. Dieser ermöglicht CAN-Botschaften zu filtern und periodisch zu verschicken [18]. Um CAN-Botschaften letztlich durch Kommandozeilenbefehle versenden und analysieren zu können und um das zunächst rudimentäre Busnetzwerk zu testen, kommt das Linux-Tool *can-utils* zum Einsatz. Einige verwendete Befehlsanwendungen, die das Dienstprogramm liefert, sind:

`cansend`:	Senden einer Single Frame Botschaft
`cangen`:	Erzeugen von zufälligem CAN-Busverkehr
`canplayer`:	Wiedergabe von CAN-Logfiles
`candump`:	Anzeigen, Filtern und Protokollieren von CAN-Botschaften
`cansniffer`:	Anzeige inhaltlicher Unterschieder zweier CAN-Botschaften
`canbusload`:	Berechnen und Anzeigen der Buslast

Um die Konfiguration des Bussystemes zu testen, werden die beiden internen CAN-Controller des Jetson TX2 als Busknoten gesetzt. Wie in Abbildung 4.4 werden zwei Transceiver an die entsprechenden GPIO-Pins für CAN_TX und CAN_RX des Jetson TX2 angeschlossen und über eine verdrillte Zweidrahtleistung miteinander verbunden. Dieser Aufbau entspricht nun einem vollständigen CAN-Netzwerk mit zwei sende- und leseberechtigten CAN-Knoten. Als letzter Schritt muss die gewünschte Datenrate konfiguriert und die Netzwerkschnittstellen aktiviert werden. Dies wird mit dem Befehl

```
nvidia@tegra-ubuntu:~$ ip link set can0 type can bitrate 500000
                       ip link set up can0
                       ip link set can1 type can bitrate 500000
                       ip link set up can1
```

durchgeführt. Die Datenübertragungsrate muss für alle CAN-Knoten im selben Netzwerk

übereinstimmen, um eine synchrone und fehlerfreie Datenübertragung zu gewährleisten. In der Fahrzeugindustrie hat sich bei CAN-Bussystemen eine Datenrate von typischerweise 500 kBit/s etabliert, weshalb auch hier diese Verwendung findet.
Werden nun durch den Befehl `cangen can0` Busbotschaften des Knoten CAN0 generiert, können diese Botschaften auf dem Interface CAN1 empfangen und ausgelesen werden:

```
nvidia@tegra-ubuntu:~$ candump can1
    can1   715   [1]   A6
    can1   4D0   [8]   43 44 AB 7B A6 62 01 1B
    can1   2B6   [2]   98 16
    can1   4C1   [8]   BE 3B 0F 36 72 40 CC 72
    can1   0DB   [4]   93 B8 5E 76
    can1   714   [5]   A8 A0 AB 78 A3
    can1   730   [1]   82
    can1   049   [8]   C7 44 D5 51 5E 07 27 75
    can1   58A   [8]   E7 C1 80 3B 5A 1A 0C 4F
    can1   509   [8]   01 7D 0D 6A BF 8F 38 5A
    can1   5CC   [8]   B3 74 A5 7D 8D FA 8A 41
    can1   405   [1]   FF
```

Dabei gilt zu beachten, dass hier lediglich zufällige Botschaften mit zufälligen IDs erstellt und versendet wurden. Der Inhalt der Botschaften hat für die weitere Umsetzung keinerlei Bedeutung und soll lediglich das korrekte Versenden und Empfangen von Busbotschaften über die physikalische Busleitung darstellen.
Es sei hier noch angemerkt, dass es sich bei den *can-utils*-Funktionen um Terminal-Anwendungen handelt, d.h. die einzelnen Befehle werden unter Linux im Terminal aufgerufen und die zugehörigen Funktionsparameter wie ID oder Botschaftsdaten als Kommandozeilenbefehl angegeben. Zur Realisierung der Diagnosefunktion wird von diesem Konzept abgewichen, um den Diagnoseumfang erweiterbar zu implementieren.

4.2.2. Empfängerseitige Busanbindung

Nachdem die Busanbindung aufgebaut und getestet wurde, muss nun der Laptop an die Busleitung gekoppelt und zum zyklischen Empfang der Diagnosedaten konfiguriert werden. Die gesamte Busverbindung wird wie in Abbildung 4.5 dargestellt aufgebaut.

Um die Buspegel auf der Empfängerseite von einer Differenzspannungssignal wieder in eine logische Signalfolge zu interpretieren, ist ein Interface nötig, welches das CAN-Protokoll auf einen universellen seriellen Bus umsetzt. Das *CANcaseXL* der Firma *Vector Informatik GmbH* ist ein USB-Interface, das eine physikalische Ankopplung eines

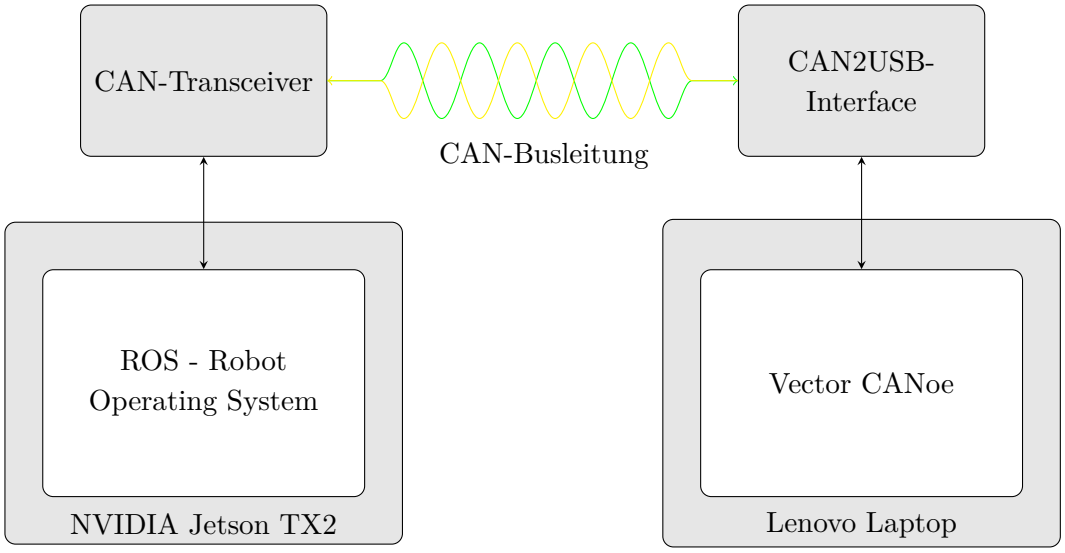


Abbildung 4.5.: Schematischer Aufbau der Buskommunikation zwischen dem EVObot und dem Diagnoserechner

Computers an ein reales Bussystem erlaubt. Mithilfe dieser Umsetzung ist es möglich, sowohl Busbotschaften zu generieren und diese auf den Bus zu senden, als auch reale Botschaften von angekoppelten Systemen auszulesen und auszuwerten. Über die beiden D-Sub DE9 Anschlüsse ist eine Verbindung zu CAN- oder LIN-Netzwerken realisierbar. Zwar wurde das CANcaseXL inzwischen von neueren Modulen abgelöst und bietet lediglich einen eingeschränkten Funktionsumfang gegenüber der weiteren Produktpalette der *Vector Informatik GmbH*, jedoch eignet es sich aufgrund der vergleichsweise kompakten Bauform besonders für die hier gewünschte Anwendung. Zudem ist der integrierte Funktionsumfang für den Einsatz als CAN-Interface völlig ausreichend [19].

Zur softwareseitigen Auswertung und zum Versenden der Busbotschaften kommt das Software-Werkzeug *CANoe* der Firma *Vector Informatik GmbH* zum Einsatz. Das vielseitige Tool bedient Anwendungsgebiete wie Analyse, Diagnose, Simulation, Stimulation und Test während dem gesamten Entwicklungsprozess von Steuergeräten und Netzwerken. CANoe unterstützt dabei sämtliche gängigen Bussysteme sowie die normierten Transport- und Diagnoseprotokolle. Neben der Überwachung des realen Busverkehrs bietet die Software die Möglichkeit, einfache Teilsysteme bis hin zu komplexen Restbussystemen zu simulieren. Dabei kann die Buskommunikation jederzeit textuell dargestellt und die Signale visuell veranschaulicht werden [20].
Das Softwaretool liefert zusätzlich die integrierte und firmeneigene Programmiersprache *Communication Access Programming Language* CAPL. Die auf *C* basierte Sprache wurde speziell für die Anforderungen zur Entwicklung von Bussystemen angepasst. CAPL

bietet als eventorientierte Skriptsprache die Möglichkeit, bequem auf Signale zuzugreifen und die Signalwerte zu verändern. Es steht eine Vielzahl an vordefinierter Funktionen zur Verfügung. Ebenso können eigene grafische Benutzeroberflächen erstellt werden, wodurch komplexe Simulationsumgebungen erleichtert werden [21].

4.3. Implementierung der Diagnosefunktion

Nachdem nun alle hardwareseitigen Voraussetzungen erfüllt und sowohl der EVObot, als auch der Diagnoselaptop für eine Buskommunikation konfiguriert sind, kann die vollständige Diagnosefunktion implementiert werden. Es soll das Vorgehen bei der softwareseitigen Umsetzung beschrieben und relevante Auszüge aus Programmcode näher betrachtet werden.
Wie bereits in Abschnitt 3.2 beschrieben, stellt das Framework Robot Operating System die zentrale Entwicklungsumgebung des Projektes dar. Daher kann auch aus den definierten Anforderungen der Anspruch entnommen werden, die Diagnose ebenfalls in ROS einzubinden. Dazu wird ein neuer Knoten[3] erstellt, der sich in die Knotenstruktur des ROS Entwicklungsprojektes eingliedert. Der bisherige Programmgraph wird aus Abbildung 4.6 ersichtlich.

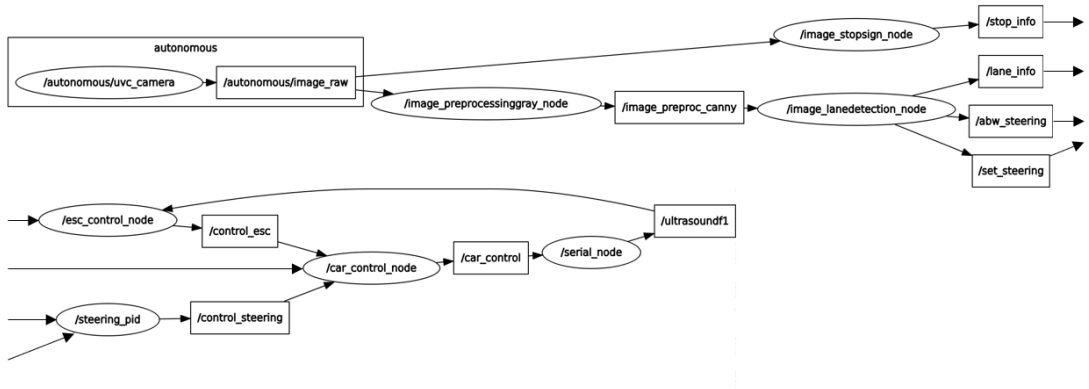


Abbildung 4.6.: Programmgraph des initialen ROS-Paketes *autonomous*, mit dem Befehl `rqt_graph` automatisch generiert

Wie zuvor in Abschnitt 3.2 erwähnt, werden Knoten in der ROS-Umgebung als ovale Felder dargestellt. Tauschen einzelne Knoten Informationen in Form von Messages aus, wird dies über die Topics genannten Datenstreams realisiert. Topics werden im Schau-

[3]Der Begriff Knoten beschreibt in ROS in den meisten Fällen entkoppelt ausführbare Skripte, in denen diverse Funktionen umgesetzt werden. Daher wird nachfolgend im direkten Bezug zur ROS-Umgebung nicht zwischen den Begrifflichkeiten Diagnosefunktion und Diagnoseknoten unterschieden.

bild als Rechtecke dargestellt. Es sei an dieser Stelle angemerkt, dass die Darstellung in Abbildung 4.6 für eine angenehme Übersicht auf zwei Zeilen aufgeteilt wurde. Auf die expliziten Funktionsumfänge der einzelnen Knoten wird an späterer Stelle eingegangen. ROS lässt sich sowohl in der sehr effiziente und maschinennahe Programmiersprache C++, als auch in der benutzerfreundlichen und universellen Skriptsprache Python betreiben. Python bietet den entscheidenden Vorteil, eine große Anzahl von Standardbibliotheken verwenden zu können, die sich zusätzlich simpel durch das Laden von Paketen erweitern lässt. Sämtliche im EVObot bereits implementierte Funktionen wurden zuvor in Python umgesetzt. Um eine generische Projektfunktionalität zu gewährleisten, wird die Umsetzung der Diagnose in der ROS-Umgebung ebenfalls in Python in der Version 2.7 realisiert.

4.3.1. Einlesen der Sensor- und Signaldaten

Zur vollständigen Integration der Diagnosefunktion in ROS eignet es sich, einen neuen Knoten mit der Bezeichnung `diagnostic_node.py` anzulegen und sämtliche, während der Programmlaufzeit durch Datenstreams ausgetauschte Informationen in diesen Knoten einzulesen. Dazu wird sich dem Prinzip der *Publisher* und *Subscriber* in ROS bedient. Um die verteilten Funktionsumfänge eines komplexen Projektes in ROS umzusetzen, publizieren die einzelnen Skripte einmalig oder zyklisch Informationen durch Topics. Diese Informationen sind beispielsweise Messdaten eines eingelesenen Sensors oder berechnete Stellwerte eines Aktors. Damit andere Knoten diese publizierten Informationen einlesen und verarbeiten können, müssen die relevanten Topics bei Initialisierung des Knoten abonniert werden, man spricht von einem Subscriber [22]. In der Diagnosefunktion werden alle global in der ROS-Umgebung publizierten Informationen über Topics abonniert, um sie anschließend über das CAN-Protokoll auf dem Datenbus versenden zu können. Der erzeugte Knoten muss also bei Initialisierung zunächst alle diagnoserelevanten Topics abonnieren, um die Daten einlesen und verarbeiten zu können:

```
NODE_NAME  = "diagnostic_node"
SUB_TOPIC1 = "/lane_info"
SUB_TOPIC2 = "/stop_info"
SUB_TOPIC3 = "/car_control"
SUB_TOPIC4 = "/ultrasoundf1"
SUB_TOPIC6 = "/control_esc"

rospy.init_node(NODE_NAME, anonymous=True)

rospy.Subscriber(SUB_TOPIC1, lane,       callback1)
```

```
rospy.Subscriber(SUB_TOPIC2, stopsign,   callback2)
rospy.Subscriber(SUB_TOPIC3, carcontrol, callback3)
rospy.Subscriber(SUB_TOPIC4, Range,      callback4)
rospy.Subscriber(SUB_TOPIC5, Image,      callback5)
rospy.Subscriber(SUB_TOPIC6, UInt16,     callback6)
```

Eine Analyse der verteilten Funktionsumfänge in ROS zeigt, dass das Abonnieren von fünf Topics nötig ist, um alle dynamischen Daten in das Diagnoseskript einzulesen. Diese Daten werden als dynamisch bezeichnet, weil sich der Inhalt der einzelnen Variablen während der Laufzeit der Fahrroutine in ROS ändert. Nachfolgend in Tabelle 4.1 werden die abonnierten Topics aufgelistet und zum besseren Verständnis die Inhalte der entsprechenden Messages erläutert.

Tabelle 4.1.: Beschreibung der im Diagnoseknoten abonnierten Topics mit veränderlichen Variablen und den damit eingelesenen Variablenwerte

Topic	Message Type	Value	Beschreibung
lane_info	lane	int16 right	Erkannte Fahrspur rechts, horizontale Position in px
		int16 left	Erkannte Fahrspur links, horizontale Position in px
		int16 abw	Abweichung zur berechneten Spurmitte in px
		bool erkennung	Status erkannte Fahrspur
		float64 radius	Berechneter Kurvenradius in cm
stop_info	stopsign	bool erkennung	Status erkanntes Stoppschild
		uint16 posx_px	horizontale Position im Bild in px
		uint16 posy_px	vertikale Position im Bild in px
		uint8 breite_px	Breite in px des erkannten Stoppschildes
		uint8 hoehe_px	Höhe in px des erkannten Stoppschildes
car_control	carcontrol	uint16 servo	Lenkwinkel als Stellwert für den Servomotor
		uint16 esc	PWM-Signal als Stellwert für den Antriebsmotor
ultrasoundf1	Range	float32 min_range	minimaler Abstand in m
		float32 max_range	maximaler Abstand in m
		float32 range	momentaner Abstand in m, ungefiltert
us_abstand	std_msgs	float32	momentaner Abstand in m, gefiltert

Gegenüber diesen dynamischen Parametern, werden zusätzlich noch statische Parame-

ter eingelesen. Diese werden vor Start der Fahrroutine im ROS-spezifischen *launch-File* parametriert und ändern sich während der gesamten Laufzeit nicht. Es liegt also auf der Hand, dass ein Abonnieren, zyklisches Einlesen und wiederholtes Versenden auf dem CAN-Bus dieser statischer Parameter nicht effizient ist, da dies die Buslast erhöht und damit die effektive Datenrate senkt. Daher werden diese statischen Parameter direkt nach Initialisierung des Diagnoseknotens in ROS einmalig durch den Befehl `rospy.get_param("/param_name")` eingelesen. Tabelle 4.2 listet die einmalig erfassten Topics auf, die die nötigen statischen Parameter enthalten.

Tabelle 4.2.: Beschreibung der im Diagnoseknoten eingelesenen Topics mit statischen Variablenwerten

Topic	Message Type	Value	Beschreibung
uvc_camera	Image	char device	Pfad zur Kamera-Konfigdatei
		uint16 fps	Eingelesene Bilder pro Sekunde
		uint16 width	Breite des Kamerabildes in px
		uint16 height	Höhe des Kamerabildes in px
steering_pid	controller	float32 Kp	Faktor der Proportionalverstärkung
		float32 Ki	Faktor der Integralverstärkung
		float32 Kd	Faktor der Differentialverstärkung
		int16 lower_limit	Unteres Limit des Regelparameters
		int16 upper_limit	Oberes Limit des Regelparameters
		int16 windup_limit	Maximale Grenze für das Fehlerintegral
		int16 max_loop_frequency	Maximale Regelfrequenz

Beim Einlesen der Sensor- und Signaldaten in den Diagnoseknoten wurde eine Inkonsistenz der verwendeten Datentypen festgestellt. Die Variablen der selbstdefinierten Messages in den Topics *lane_info*, *stop_info* und *car_control* wurden zunächst jeweils mit der maximal möglichen Speicherbreite deklariert. Beispielsweise war die Statusvariable der Fahrspurerkennung als eine 64 Bit große integer-Variable angelegt, obwohl diese Variable lediglich die logischen (booleschen) Werte *true* und *false* annehmen kann. Dies würde zu einer äußert ineffizienten Buskommunikation führen, da demzufolge nach der Signalzuweisung lediglich ein einfacher boolescher Statuswert in einer acht Byte großen Busbotschaft übermittelt wird. Die ineffiziente Speichernutzung widerspricht den Paradigmen des CAN-Protokolls.
Daher wurden zunächst sämtliche selbstdefinierten Messages geprüft und auf eine möglichst speichersparende Deklaration der Datentypen angepasst. Beim genannten Beispiel der Statusvariable der Fahrspurerkennung wurde eine triviale Änderung auf einen boole-

schen Datentypen angewandt. Die Variablen zur horizontalen Bildposition der erkannten Fahrspuren können beispielsweise einen theoretisch maximalen Wert von 640 annehmen, da die horizontale Auflösung des Kamerabildes 640 Pixel beträgt. Mit einem 64 Bit breiten integer Datentyp sind jedoch bis zu $2^{64} - 1$ Werte abbildbar. Bei einer Skalierung mit dem Faktor 1 impliziert dies einen sehr großen, nicht genutzten Speicherbereich der Variable. Es genügt ein uint16 Datentyp, um alle möglichen Werte der Bildposition in Pixel abbilden zu können, ohne einen Datenüberlauf zu riskieren. Eine Skalierung der Pixel auf den gesamten Speicherbereich der Variable ergibt hier kein Sinn, da eine höhere Auflösung der Bildwerte keinen Mehrwert erzielen würde. Nach dieser Methode wurden die möglichen Wertebreiche aller Variablen geprüft und auf einen speichersparenden Datentypen angepasst.

4.3.2. Wandlung der Speichervariablen

Bei Erhalt einer neuen Message werden die Variablenwerte der abonnierten Topics zuerst in eine geeignete Form gewandelt, um sie anschließend einer Busbotschaft zuzuweisen und auf dem CAN-Bus zu versenden. Dies geschieht in einzelnen *Callback*-Funktionen, aus denen heraus die Messages der entsprechenden Topics ebenfalls als Datenstream versendet werden. Gegenüber einer herkömmlichen Funktion in der Informatik, wird eine Callback-Funktion einer andere Funktion als Pointer übergeben und von dieser mit den definierten Argumenten aufgerufen. Eine Callback-Funktion wird für gewöhnlich zwar vom Anwender definiert, jedoch nicht durch ihn aufgerufen. In ROS wird eine Callback-Routine meist als Message Handler verwendet. Sobald ein Knoten ein Topic abonniert und eine neue Message verfügbar ist, die von diesem Knoten zyklisch publiziert wird, wird der Message Handler durch ROS aufgerufen und die neue Message durch die Rückruffunktion an eine definierte Funktion übergeben. So können zyklisches Messages ähnlich einem *Interrupt* verarbeitet werden.

Ist für den Diagnoseknoten eine neue Message der abonnierten Topics verfügbar, wird diese durch einen Callback an eine Funktion übergeben, die den Inhalt der Message in hexadezimale Bytes packt. Dazu wird die Python-Bibliothek `struct` verwendet [23]. Das Modul stellt verschiedene Funktionen bereit, mit denen strukturierte binäre Daten verarbeitet werden können. Es lassen sich Python Variablen mit sämtlichen Datentypen in einen eindimensionalen Byte Array wandeln. Dies soll anhand folgendem Beispiel verdeutlicht werden:
Der PWM-Wert zur Ansteuerung des Antriebsmotors beträgt 1550 in der Variable `eng_pwm`. Dies entspricht der hexadezimalen Form 0x 60E. Der Variablenwert wird mit dem Befehl `struct.pack("Format", "Wert")` einen Byte Array gewandelt. Als Byteorder kommt hier das Big-Endian-Format zum Einsatz. Der Variabelenwert soll durch

das Format H in den Datentypen *unsigned short* mit einer Speicherbreite von 2 Byte umgeschrieben werden.

```
eng_pwm = 1550
eng_pwm_byte = struct.pack(">H", eng_rpm)
print("Byte Array:", eng_pwm_byte)
```

Der Output des Codebeispiels ist

```
Byte Array: '\x06\x0e'
```

Demzufolge liegt das Ergebnis nun in einer Byte-Zeichenfolge vor, die hier in hexadezimaler Form dargestellt wird. Auf die einzelnen Elemente des Array kann nun wie gewohnt zugegriffen werden. Die eingelesenen Signal- und Sensordaten liegen damit nun in einer für den CAN-Bus geeigneten Form als Nutzbytes vor.

Bei der Verwendung der gezeigten Methode ist darauf zu achten, eine einheitliche Byte-Reihenfolge bei der Interpretation zu verwenden. Bedingt durch die Prozessorarchitektur werden die einzelnen Bytes einer gesamten Botschaft in unterschiedlicher Reihenfolge versendet und eingelesen. Diese Byte-Reihenfolge wird als Endianness bezeichnet. Die Endianness bestimmt, welches Byte innerhalb einer Busbotschaft das nullte und welches das höchste Byte darstellt [24]. Man unterscheidet grundlegend zwischen den beiden Formaten *Big-Endian*, also das große Ende und *Little-Endian*, das kleine Ende. Bei Big-Endian wird der signifikanteste Wert in einer Sequenz zuerst abgelegt, also in der niedrigsten Speicheradresse der Sequenz. Bei Little-Endian hingegen, wird der am wenigsten signifikante Wert zuerst gespeichert. Umgänglich wird das Big-Endian-Format auch als Motorola-Format bezeichnet. Das Little-Endian-Format ist hingegen als Intel-Format bekannt. Diese Bezeichnungen entstanden durch die grundsätzlich differenzierte Umsetzung der Prozessorarchitekturen beider Hersteller [25]. In automotiven Netzwerken wird in den häufigsten Fällen Big-Endian – das Motorola-Übertragungsformat – verwendet. CAN ist jedoch lediglich ein Kommunikationsprotokoll, die Endianness kommt mit der gewählten Prozessorarchitektur. Die Basis CAN-Spezifikation und die *ISO 11898* definieren den Informationstransport ohne eine explizite Byte-Reihenfolge.

4.3.3. Versenden der CAN-Botschaften

Wie bereits in den Grundlagen in Abschnitt 2.2.4 erläutert, können in einem Datenfeld einer CAN-Botschaft bis zu 8 Byte an Nutzdaten übertragen werden. Da die Sensor- und Signaldaten der Fahrroutine in ROS nun als Byte-Reihe vorliegen, können diese segmentiert in jeweils 8 Bytes nun in eine CAN-Botschaft gepackt werden. Hierzu wird die Bibliothek `python-can` [**python-can**] verwendet. Diese Erweiterung bietet eine CAN-Unterstützung in der Python-Umgebung, unabhängig vom Betriebssystem. Sie bietet

gängige Abstraktionen für eine Vielzahl von Hardwaregeräten, sowie eine Reihe von Dienstprogrammen zum Senden und Empfangen von Nachrichten auf einem CAN-Bus. Für gewöhnlich wird die Bibliothek in Verbindung mit einem definierten CAN-Interface verwendet. Im Anwendungsfall handelt es sich um das in Abschnitt 4.2.2 beschriebene CANcaseXL. Dieses Interface kann direkt im ausführbaren Skript spezifiziert werden, aus Konfigurationsdateien oder Umgebungsvariablen gelesen werden. Um eine Busbotschaft auf einem Python-Skript heraus über die Schnittstelle auf den physischen Bus zu versenden, wird das Interface und der Kanal direkt im Skript nach Initialisierung des Diagnoseknotens spezifiziert:

```
bus = can.interface.Bus(bustype='socketcan', channel='can0',
      bitrate=500000)
```

Als Interface wird hier die zuvor eingerichtete API SocketCAN verwendet. Der CAN-Netzwerktreiber bietet eine generische Schnittstelle für eine Vielzahl von CAN-Geräten. Zwar können in der neuesten Version der Funktionsbibliothek an dieser Stelle direkt herstellerspezifische CAN-Interfaces - auch Produkte der Firma Vektor Informatik GmbH - eingebunden werden, diese Version weist allerdings zum Zeitpunkt der Umsetzung noch diverse Mängel auf, weshalb sich mit der generischen Schnittstelle beholfen wird. Als Kanal wird einer der beiden auf dem Jetson TX2 eingerichteten Controller eingestellt. Die Datenrate wird netzwerkweit mit 500 kBit/s definiert.
Im Anschluss wird der Botschaftsrahmen sämtlicher CAN-Frames beschrieben. Es werden die Identifier den Botschaftsobjekten zugeordnet und alle Nutzdaten der Botschaften mit dem Inhalt 0x 00 initialisiert. Nach dieser Initialisierung weisen alle Botschaften eine Datenlänge von 8 Byte auf. Zudem können die Botschaftsobjekte mit einem Zeitstempel versehen und sowohl der Frametyp (Data Frame, Remote Frame, Error Frame), als auch das Identifier-Format festgelegt werden. Eine weitere Definition des Botschaftsrahmens ist nicht notwendig, das CRC-Feld und das ACK-Feld werden vom CAN-Protokoll auf der Datensicherungsebene einer Botschaft beigefügt. Der Schritt der manuellen Vergabe der Identifiern kann umgangen werden, indem unter Verwendung der Bibliothek `cantools` [**cantools**] eine definierte Datenbasis in das Skript geladen wird. Die Nutzung einer Datenbasis und einer zugehörigen Kommunikationsmatrix erleichtert das Arbeiten mit CAN-Botschaften erheblich. Die Hintergründe sollen im nächsten Abschnitt 4.3.4 erläutert werden.

Nachdem alle nötigen Komponenten initialisiert sind, wird in der Hauptroutine des Diagnoseskriptes eine Warteschleife durchlaufen und dabei der CAN-Bus eingelesen, bis eine explizite Busbotschaft empfangen wird. Diese Botschaft wird aus CANoe heraus auf den Bus gesendet (vgl. Abschnitt 4.3.4) und stellt eine Startanweisung der Diagnosefunktion dar. Entspricht der Identifier dem hochprioren Wert 0x 01 und enthält die Botschaft den Bytewert 0x 01, soll die Diagnosekommunikation gestartet werden. In diesem Fall wird

zunächst durch den Befehl

```
bus.send(reset_msg)
```

eine Reset-Botschaft auf den Bus gesendet. Dieses einmalige Senden einer Botschaft mit definiertem, aber irrelevantem Inhalt dient lediglich der Absicherung, dass keine falschen Informationen versendet werden und soll eine spätere Analyse der Buskommunikation erleichtern.
Im Anschluss werden zunächst die statischen Informationen wie zuvor aufgezeigt in den Diagnoseknoten eingelesen, die Datentypen in Byte-Arrays gewandelt und im Motorola-Format dem Nutzdatenfeld der Botschaften mit definierten Identifiern zugewiesen. Dabei muss die Breite der Nutzbytes beachtet werden. Werden mehrere Signalinformationen in einer Botschaft zusammengelegt, darf eine maximale Anzahl von 8 Bytes, respektive 64 Bits nicht überschritten werden, ansonsten kommt es zu einem Überlauf der Busbotschaft und es gehen Teile der Signalinformationen verloren. Die statischen Parameter der Kamera werden beispielsweise wie folgt an die Busbotschaft `stat_img_info_msg` übergeben:

```
for i in reversed(range(0, len(camera_width))):
      stat_img_info_msg.data[i + 6] = camera_width[i]
      stat_img_info_msg.data[i + 4] = camera_height[i]
      stat_img_info_msg.data[i + 2] = camera_fps[i]
```

Da es sich hierbei jeweils um ursprünglich 2 Byte breite integer-Werte handelt, weist die Busbotschaft eine Datenlänge von 6 Byte auf. Ist das Statusflag für den Buszustand auf aktiv, wird abschließend die entsprechende Botschaft gesendet:

```
bus.send(stat_img_info_msg)
```

Wurde die Routine der statischen Parameter durchlaufen, werden sämtliche Topics mit dynamischen Parametern abonniert und dem Anwender eine Information ausgegeben, dass der Diagnosemodus aktiv ist. Nun werden bei eintreffen einer neuen Message wie zuvor erläutert die Callback-Funktionen der Topics aufgerufen, die dynamischen Parameter nach gleichem Schema in das Datenfeld der Busbotschaften gelegt und die CAN-Nachrichten versendet. Das Eintreffen einer neuen Message ist zeitlich abhängig von dem Rechenaufwand der implementierten Prozesse in den Topics. Weitere Abhängigkeiten ergeben sich durch das Betriebssystem, die CPU-Taktrate, die Größe des Hauptspeichers und die rechnerinterne Übertragungsrate. Da der CAN-Bus ereignisgesteuert ist, wurde auf ein explizites Time-Scheduling in der Diagnosefunktion verzichtet. Die Busbotschaften werden also nicht zyklisch, sukzessiv nach einem vorgegebenen Ablauf ausgegeben, sondern werden abhängig vom Eintreffen einer neuen Message unverzüglich versendet. Zudem wird ein Senden redundanter Signalinformationen vermieden und somit die Busauslastung auf ein Minimum gesenkt.

Parallel zum dauerhaften Senden der aktualisierten Signaldaten in Busbotschaften wird der CAN-Bus auf eingehende Nachrichten gefiltert. Wie den Diagnosemodus durch eine Startanweisung zu aktivieren, kann dieser auch wieder ausgeschaltet werden. Dazu wird die Nachricht `diagnostic_status_message` mit dem Identifier 0x 01 und dem Nachrichteninhalt 0x 00 vom externen Netzwerkknoten, also dem Laptop, erwartet. Wird diese Botschaft auf dem CAN-Bus erkannt, wird das Statusflag gekippt und sämtliche Topics deabonniert. Somit wird das gesamte Diagnoseskript in einen passiven Modus gesetzt und das Senden der Busbotschaften wird unterbunden. Durch das explizite Deabonnieren wird verhindert, dass die Prozesslaufzeiten der weiterhin aktiven ROS-Knoten unnötigerweise ansteigen und damit die verfügbare Rechenleistung eingeschränkt wird. Durch nochmaliges Senden der Statusnachricht kann der Prozess der Diagnosefunktion wieder angestoßen und damit die Buskommunikation erneut gestartet werden. Somit ist ohne die Verwendung eines normierten Diagnoseprotokolls auf der Anwendungsschicht ein ähnliches Vorgehen umgesetzt: Durch das Senden einer *Diagnostic Service Request* von dem Ausgabegerät an das Demonstratorfahrzeug wird eine *Diagnostic Session* gestartet und die diagnoserelevanten Parameter an den externen CAN-Knoten versendet (vgl. [1] S. 18 ff.).

Botschaftsrahmen, Datenbytes Python Package can: Initialisierung des physischen Bus Definition der Messages "Dies entspricht der in der Datenbasis angelegten Identifiern, die im nächsten Abschnitt näher erläutert werden.SSenden der Busbotschaft innerhalb callback Deabonnieren, wenn Diagnose passiv ist

4.3.4. Interpretation und Visualisierung der CAN-Botschaften

Konfiguration der CANoe-Umgebung Erstellung Datenbasis als dbc Kommunikationsmatrix Amann Visualisierung im Panel

4.4. Ergebnisbetrachtung

Wie eingangs erläutert, war es die Intention, eine Diagnosefunktion vollständig in der ROS-Umgebung einzubinden und als einen neuen Programmknoten umzusetzen, ohne die bereits implementierten Skripte editieren zu müssen. Abbildung 4.7 zeigt, dass dies anhand dem beschriebenen Vorgehen gelungen ist. Der `diagnostic_node` reiht sich in dem initialen Programmgraphen ein und hat sämtliche Topics abonniert, die innerhalb ROS publiziert werden. Der Diagnoseknoten selbst veröffentlicht keine Informationen in der Programmumgebung und agiert damit lediglich passiv im ROS-Framework. Die

eigentliche Fahrroutine und sämtliche Algorithmen zur Bildverarbeitung und der Fahrzeugregelung werden durch die Diagnosefunktion nicht beeinträchtigt.

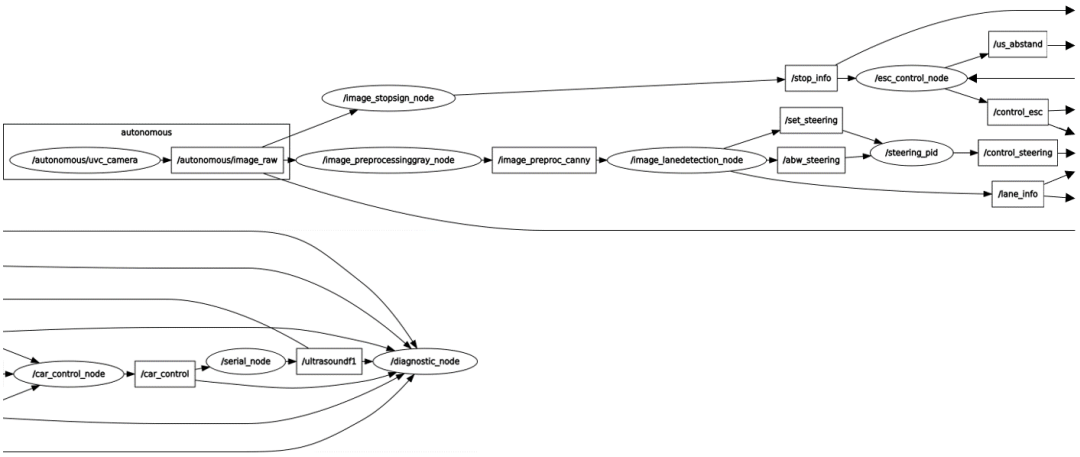


Abbildung 4.7.: Programmgraph des neuen ROS-Paketes *autonomous* inklusive dem Knoten *diagnostic_node*, mit dem Befehl `rqt_graph` automatisch generiert

4.4.1. Test und Validierung

4.4.2. Mehrwert der Diagnosefunktion

5. Dynamische Längs- und Querregelung

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

5.1. Umsetzung des Fahralgorithmus

5.2. Kritische Analyse der implementierten Algorithmen

5.3. Optimierung der Regelung

5.4. Implementierung der optimierten Spurregelung

5.5. Ergebnisbetrachtung

5.5.1. Test und Validierung

5.5.2. Mehrwert der optimierten Spurregelung

6. Zusammenfassung und Ausblick

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

6.1. Zusammenfassung

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

6.2. Ausblick

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante.

Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Tabellenverzeichnis

Abbildungsverzeichnis

Literatur

[1] Christof Kary. “Ermittlung der Rekuperationsenergie und der Leistungsaufnahme von Nebenverbrauchern in einem Elektroauto”. Institut für Energieeffiziente Mobilität. Bachelorarbeit. Karlsruhe: Hochschule Karlsruhe, 2016-09-12. 96 S. (Geprüft am 12. 09. 2018).

[2] Niels Klußmann. *Lexikon der Kommunikations- und Informationstechnik. Telekommunikation, Datenkommunikation, Multimedia, Internet*. ger. 2., erw. und aktualisierte Aufl. Heidelberg: Hüthig, 2000. 871 S.

[3] Vector Informatik GmbH. *Einführung in die Seriellen Bussysteme im Kfz*. Copyright: 2018 Vector Informatik. URL: `https://elearning.vector.com/vl_sbs_introduction_de.html` (geprüft am 08. 06. 2018).

[4] Vector Informatik GmbH. *Einführung in CAN*. Copyright: 2018 Vector Informatik. URL: `https://elearning.vector.com/vl_can_introduction_de.html` (geprüft am 08. 06. 2018).

[5] Daniel Schüller. “Bussysteme im Automobil. Ausarbeitung zum Seminarvortrag”. Koblenz: Universität Koblenz-Landau, 2005-01-20. (Geprüft am 08. 06. 2018).

[6] International Organization for Standardization, Hrsg. *ISO/IEC 7498-1: Information technology - Open Systems Interconnection - Basic Reference Model: The Basic Model*. ISO/IEC. Genf, CH: ISO/IEC Copyright Office, 15. Nov. 1994. URL: `http://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip` (geprüft am 08. 06. 2018).

[7] Henning Wallentowitz und Konrad Reif, Hrsg. *Handbuch Kraftfahrzeugelektronik. Grundlagen - Komponenten - Systeme - Anwendungen*. ger. 2., verb. und aktualisierte Aufl. ATZ/MTZ-Fachbuch. Wiesbaden: Vieweg + Teubner, 2011. 724 S.

[8] Werner Zimmermann und Ralf Schmidgall. *Bussysteme in der Fahrzeugtechnik. Protokolle, Standards und Softwarearchitektur*. 5., aktual. und erw. Aufl. ATZ / MTZ-Fachbuch. Wiesbaden: Springer Vieweg, 2014. 507 S. URL: `http://dx.doi.org/10.1007/978-3-658-02419-2`.

[9] Vector Informatik GmbH. *Einführung in FlexRay*. Copyright: 2018 Vector Informatik. URL: https://elearning.vector.com/vl_flexray_introduction_de.html (geprüft am 08. 06. 2018).

[10] DIN Deutsches Institut für Normung e.V., Hrsg. *Qualitätsmanagementsysteme*. DIN EN ISO. Version 2005-12. Berlin: DIN Deutsches Institut für Normung e.V., 1. Nov. 2015. URL: http://perinorm-fr.redi-bw.de/volltexte/CD21DE07/2325650/2325650.pdf? (geprüft am 13. 07. 2018).

[11] DIN Deutsches Institut für Normung e.V., Hrsg. *Begriffe zum Qualitätsmanagement*. DIN. Version 11. Berlin: DIN Deutsches Institut für Normung e.V., 1. Mai 2008. URL: http://perinorm-fr.redi-bw.de/volltexte/CD21DE03/1415055/1415055.pdf? (geprüft am 13. 07. 2018).

[12] Josef Börcsök. *Elektronische Sicherheitssysteme. Hardwarekonzepte, Modelle und Berechnung*. 2., überarbeitete Auflage. Praxis. Heidelberg: Hüthig GmbH & Co. KG, 2007. 608 S. URL: http://deposit.d-nb.de/cgi-bin/dokserv?id=2995265&prov=M&dok_var=1&dok_ext=htm.

[13] ISTQB AISBL, German Testing Board e.V. *ISTQB GTB Standardglossar der Testbegriffe. Version vom 21.05.2017*. 2017. URL: http://glossar.german-testing-board.info/ (geprüft am 20. 07. 2018).

[14] Konrad Reif. *Automobilelektronik. Eine Einführung für Ingenieure*. 5., überarb. Aufl. ATZ-MTZ-Fachbuch. Wiesbaden: Springer Vieweg, 2014.

[15] S. Leonhardt und M. Ayoubi. *Methods of fault diagnosis*. Hrsg. von Darmstadt University of Technology. Institute of Automatic Control. 1997. URL: https://www.sciencedirect.com/science/article/pii/S0967066197000506 (geprüft am 11. 09. 2018).

[16] Florian Schäffer. *OBD - Fahrzeugdiagnose in der Praxis*. ger. Elektronik. Haar bei München: Franzis, 2012. 240 S.

[17] NVIDIA Corporation. *NVIDIA Jetson TX2. OEM Product Design Guide*. Hrsg. von NVIDIA Corporation. 2017. URL: https://devtalk.nvidia.com/default/topic/998129/jetson-tx2/links-to-jetson-tx2-resources/ (geprüft am 06. 09. 2018).

[18] Oliver Hartkopp u. a. *Readme file for the Controller Area Network Protocol Family (aka SocketCAN)*. 2017. URL: https://www.kernel.org/doc/Documentation/networking/can.txt (geprüft am 09. 09. 2018).

[19] Vector Informatik GmbH. *Handbuch CANcaseXL*. 2015. URL: https://vector.com/portal/medien/cmc/manuals/CANcaseXL_Manual_DE.pdf (geprüft am 08. 06. 2018).

[20] Vector Informatik GmbH. *CANoe. Produktinformation*. 2018. URL: https://vector.com/portal/medien/cmc/info/CANoe_ProductInformation_DE.pdf (geprüft am 08. 06. 2018).

[21] Vector Informatik GmbH. *CAPL Documentation. Vector KnowledgeBase*. Vector Informatik GmbH. 7.08.2017. URL: https://kb.vector.com/entry/48/ (geprüft am 08. 06. 2018).

[22] Morgan Quigley, Brian Gerkey und Bill Smart. *Programming robots with ROS. A practical introduction to the Robot Operating System*. First edition. Safari Tech Books Online. Beijing u. a.: O´Reilly, 2015. URL: http://proquest.safaribooksonline.com/9781449325480.

[23] Python Software Foundation. *7.3. struct — Interpret strings as packed binary data — Python 2.7.15 documentation. The Python Standard Library*. Hrsg. von Python Software Foundation. 2018. URL: https://docs.python.org/2/library/struct.html (geprüft am 11. 09. 2018).

[24] Ford Motor Company. *Bit Numbering and Byte Order — OpenXC Vehicle Interface Firmware 7.2.1-dev documentation. OpenXC Vehicle Interface Firmware*. Hrsg. von Ford Motor Company. 2017. URL: http://vi-firmware.openxcplatform.com/en/master/config/bit-numbering.html (geprüft am 11. 09. 2018).

[25] Margaret Rouse. *Was ist Big-Endian und Little-Endian?* Hrsg. von Inc TechTarget. 2014. URL: https://www.searchnetworking.de/definition/Big-Endian-und-Little-Endian (geprüft am 11. 09. 2018).

A. Anhang

A.1. Auszug aus dem Datenblatt SN65HVD230

SN65HVD23x 3.3-V CAN Bus Transceivers

1 Features

- Operates with a single 3.3 V Supply
- Compatible With ISO 11898-2 Standard
- Low Power Replacement for the PCA82C250 Footprint
- Bus Pin ESD Protection Exceeds ±16 kV HBM
- High Input Impedance Allows for Up to 120 Nodes on a Bus
- Adjustable Driver Transition Times for Improved Emissions Performance
  - SN65HVD230 and SN65HVD231
- SN65HVD230: Low Current Standby Mode
  - 370 µA Typical
- SN65HVD231: Ultra Low Current Sleep Mode
  - 40 nA Typical
- Designed for Data Rates(1) up to 1 Mbps
- Thermal Shutdown Protection
- Open Circuit Fail-Safe Design
- Glitch Free Power Up and Power Down Protection for Hot Plugging Applications

(1) The signaling rate of a line is the number of voltage transitions that are made per second expressed in the units bps (bits per second).

2 Applications

- Industrial Automation, Control, Sensors and Drive Systems
- Motor and Robotic Control
- Building and Climate Control (HVAC)
- Telecom and Basestation Control and Status
- CAN Bus Standards Such as CANopen, DeviceNet, and CAN Kingdom

3 Description

The SN65HVD230, SN65HVD231, and SN65HVD232 controller area network (CAN) transceivers are compatible to the specifications of the ISO 11898-2 High Speed CAN Physical Layer standard (transceiver). These devices are designed for data rates up to 1 megabit per second (Mbps), and include many protection features providing device and CAN network robustness. The SN65HVD23x transceivers are designed for use with the Texas Instruments 3.3 V µPs, MCUs and DSPs with CAN controllers, or with equivalent protocol controller devices. The devices are intended for use in applications employing the CAN serial communication physical layer in accordance with the ISO 11898 standard.

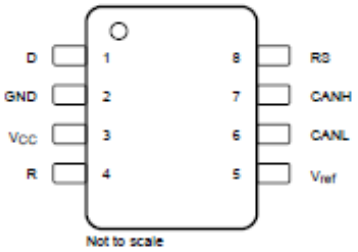
Device Information(1)

PART NUMBER	PACKAGE	BODY SIZE (NOM)
SN65HVD230	SOIC (8)	4.90 mm × 3.91 mm
SN65HVD231		
SN65HVD232		

(1) For all available packages, see the orderable addendum at the end of the datasheet.

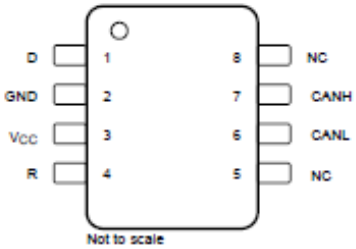
7 Pin Configuration and Functions

SN65HVD230D (Marked as VP230)
SN65HVD231D (Marked as VP231)
Top View



Not to scale

SN65HVD232D (Marked as VP232)
Top View



Not to scale

Pin Functions

PIN		TYPE	DESCRIPTION
NAME	NO.		
D	1	I	CAN transmit data input (LOW for dominant and HIGH for recessive bus states), also called TXD, driver input
GND	2	GND	Ground connection
V_{CC}	3	Supply	Transceiver 3.3V supply voltage
R	4	O	CAN receive data output (LOW for dominant and HIGH for recessive bus states), also called RXD, receiver output
V_{ref}	5	O	SN65HVD230 and SN65HVD231: V_{CC} / 2 reference output pin
NC		NC	SN65HVD232: No Connect
CANL	6	I/O	Low level CAN bus line
CANH	7	I/O	High level CAN bus line
R_S	8	I	SN65HVD230 and SN65HVD231: Mode select pin: strong pull down to GND = high speed mode, strong pull up to V_{CC} = low power mode, 10kΩ to 100kΩ pull down to GND = slope control mode
NC		I	SN65HVD232: No Connect

8 Specifications

8.1 Absolute Maximum Ratings

over operating free-air temperature range (unless otherwise noted)(1)(2)

	MIN	MAX	UNIT
Supply voltage, V_{CC}	–0.3	6	V
Voltage at any bus terminal (CANH or CANL)	–4	16	V
Voltage input, transient pulse, CANH and CANL, through 100 Ω (see Figure 24)	–25	25	V
Digital Input and Output voltage, V_I (D or R)	–0.5	V_{CC} + 0.5	V
Receiver output current, I_O	–11	11	mA
Continuous total power dissipation	See *Thermal Information*		
Storage temperature, T_{stg}	–40	85	°C

A.2. Einarbeitungsleitfaden Projekt EVObot

Ersteller: Christof Kary

Datum: 06.09.2018

Version: 1.1

Dieses Dokument soll als Leitfaden zum Einstieg in das Projekt EVObot dienen. Es soll die Einarbeitung unabhängig von der gesamten Projektdokumentation erleichtern und damit einen Wissenstransfer zwischen den Projektmitglieder sicherstellen. Sollten die in diesem Dokument genannten Begrifflichkeiten und Funktionen nicht bereits bekannt sein, wird dazu geraten, zu den jeweiligen Punkten in den genannten Links oder darüber hinaus nachzuschlagen.

1 NVIDIA Jetson TX2

Die zentrale Entwicklungsplattform des Projektes stellt das Entwicklerkit *NVIDIA Jetson TX2* dar. Der Computer wurde speziell für KI-Anwendungen, Computer Vision und rechenintensive Grafikanwendungen zur Bildverarbeitung entwickelt. Das Board verfügt über 6 CPU-Kerne und 8 GB internem Arbeitsspeicher und ist damit für hohe Rechenleistungen ausgelegt. Das zentrale Rechenmodul ist auf einer Adapterplatine mit zahlreichen Anschlüssen für umfangreiche Erweiterungen verbaut. Zur hardware- und schnittstellenspezifischen Einarbeitung sei auf den *User Guide* verwiesen. Ausführliche Beschreibungen und Tutorials sind auf der Homepage `elinux.org` zusammengefasst. Umfangreiche benutzerfreundliche Beispiele zum Arbeiten mit dem Jetson-Board finden sich auf `jetsonhacks.com`.

Um das Energie- und Leistungsmanagement zu verwalten, liefert das Jetson-Board das Kommandozeilen-Tool *nvpmodel*. Durch den Befehl

```
sudo nvpmodel -m [mode]
```

lässt sich zwischen 5 verschiedenen Leistungsmodi wählen. Es gilt zu erwähnen, dass der EVObot in der aktuellen Umsetzung dauerhaft im maximalen Leistungsmodus betrieben wird, um alle 6 CPU-Kerne bei maximaler Taktfrequenz von 2 GHz zu nutzen. Alternativ können durch Ausführen des Shell Skriptes

```
sudo ./jetson_clocks.sh
```

sämtliche Energiesparoptionen deaktiviert werden.

2 Linux Ubuntu

Auf der verwendeten Hardware läuft Ubuntu als eine Linux-Distribution in der Version *16.04.4 LTS*. Wurde zuvor nocht nicht mit Linux-Systemen gearbeitet, ist eine Einarbeitung in die Ubuntu-Umgebung dringend zu empfehlen. Es sollten sich zunächst wichtige Befehle angeeignet und der allgemeine Umgang mit der Benutzeroberfläche und dem Arbeiten über kommandozeilenbasierte Terminaleingaben angeeignet werden. Für Einsteiger in das Ubuntu-System sei auf *diese Grundlagenerklärung* verwiesen. Generell eignet sich das *Wiki* der Homepage `ubuntuusers.de` als Nachschlagewerk rund um Linux. Generell ist es von Vorteil, stets eine *Befehlsübersicht* zur Hand zu haben. Ein Auszug wichtiger, projektbezogener Befehle die häufig Verwendung finden, sind:

`cd`:	Wechsel des Arbeitsverzeichnisses
`ls`:	Auflistung von Dateien
`sudo`:	Führt nachfolgenden Befehl mit Root-Rechten aus
`mkdir`:	Erzeugung von Verzeichnissen
`cmake`:	Erstellt und kompiliert Projekte oder Makefiles
`apt-get`:	Zum Installieren, Updaten, Löschen, ... von Paketen

Der primär verwendete Benutzer auf dem *Jetson TX2* ist `nvidia`. Das Passwort für diesen Benutzer lauter ebenfalls `nvidia`. Zwar wurde die Passwortabfrage beim Systemboot deaktiviert, jedoch muss das Passwort zum Ausführen eines *sudo*-Befehls und zum Aktivieren eines *SSH*-Zugriffes verwendet werden.
Ein SSH-Zugriff ist nötig, um einen kabellosen, netzwerkbasierten Terminalzugriff auf dem EVObot zu ermöglichen. Auf Ubuntu ist bereits ein SSH-Zugang vorinstalliert. Möchte man von einem Windows-PC auf das Jetson-Board zugreifen, ist hierzu die Software *PuTTY* oder *KiTTY* nötig. Im Netzwerk *evo-flacht* ist der EVObot mit der IP 10.30.60.171 zu finden. Die IP-Adresse kann durch den Befehl

```
ifconfig
```

abgerufen werden.
Wird auf den EVObot lediglich über eine SSH-Verbindung zugegriffen, wird empfohlen, die grafische Benutzeroberfläche über den Befehl

```
sudo service lightdm stop
```

zu deaktivieren. Hiernach wird die Berechnung der GUI verhindert, um sämtliche Leistungsressourcen für die automatisierte Fahrroutine zur Verfügung zu stellen. Durch Verwendung der Option `start` im oben genannten Befehl oder durch einen Reboot des Systems lässt sich die grafische Benutzeroberfläche wieder aktivieren.

Alternativ zu einem SSH-Zugriff kann eine *VNC*-Verbindung zum Jetson-Board aufgebaut werden. Diese wird dazu verwendet, um eine Remote-Desktop Anwendung zu ermöglichen. Auf dem Jeston-Board selbst ist hierzu bereits eine VNC-Applikation aktiviert. Auf dem fernsteuernden Rechner muss zunächst die Software *VNC Viewer* installiert und eine neue Verbindung mit der Korrekte IP hergestellt werden. Sobald die VNC-Verbindung aufgebaut wurde, kann mit dem Terminal-Befehl

```
sudo xrandr --fb "Breite"x"Höhe"
```

die gewünschte Desktopgröße eingestellt werden.
Es wird dringend davon abgeraten, die Fahrroutine des EVObots über eine aktive VNC-Verbindung zu starten, da die Übertragung des Desktops einen erheblichen Anteil der verfügbaren CPU-Leistung beansprucht. Dabei kann es zu einer hohen Latenz in der Algorithmenberechnung kommen, die sich negativ auf das Regelverhalten auswirkt.

3 ROS - Robot Operating System

Auf dem Ubuntu-Betriebssystem arbeitet ROS in der Distributionsversion *Kinetic*. Dies ist eine Open Source Middleware speziell für Robotikanwendungen. Das Framework stellt umfangreiche Programmbibliotheken, Werkzeuge, Gerätetreiber, Visualisierungstools und Möglichkeiten zum Nachrichtenaustausch. Die gesamte Entwicklungsumgebung auf dem EVObot wurde mithilfe von ROS erstellt und liefert die gesamte Architektur der einzelnen Softwarekomponenten zur automatisierten Fahrfunktion. Daher ist eine ausführliche Einarbeitung in ROS essentiell. Einsteigern in das Arbeiten mit ROS wird es empfohlen, sich das *Wiki der offiziellen Homepage* anzueignen und die *Tutorials* inklusive der angegebenen Beispiele

aufmerksam durchzuarbeiten. Um eine noch umfangreichere Einarbeitung zu ermöglichen, kann das Buch *Programming robots with ROS* von Morgan Quigley, Brian Gerkey und William D. Smart herangezogen werden.

Es sollte nachvollzogen werden, dass ROS als Gesamtsystem anders agiert als unabhängig voneinander ausführbare Programme und Funktionen. Es findet innerhalb der ROS-Umgebung eine effiziente Kommunikation der Teilfunktionen durch eine flexible und simple Datenstruktur statt. Betrachtet man die Prozessarchitektur, befindet sich ROS auf der selben Ebene wie die ausführbaren Applikationen in einer Rechnerstruktur und stellt das Interface zwischen Hardware und der IPC (Inter-Process Communication) dar. Besonders die fundamentalen Konzepte von ROS und die Origanisation in *Nodes, Messages, Topics* und *Services* sollte verstanden werden. Ein weiterer wichtiger Aspekt ist das Prinzip des Informationsaustausches zwischen einzelnen Knoten (Nodes) über einen *publisher-* und *subscriber-*Mechanismus und dem damit verbundenen Aufruf von *callback-Funktionen*.
Eine Schnellübersicht der ROS-spezifischen Tools und Befehle bietet dieses *Cheat Sheet*. Darüber hinaus liefert das Framework *rqt* eine komfortable Benutzeroberfläche um Signalwerte während der Programmlaufzeit grafisch darzustellen oder um das Prozessverhalten zu analysieren. Das Paket *rostopic* liefert Debugging-Informationen einzelner Topics.

Der Workspace und sämtliche projektbezogene Dateien befinden sich im Ubuntu-System in dem Verzeichnis `/home/nvidia/EVObot/`. Das Package, das die entsprechenden launch-Files, message-Beschreibungen und ROS-Knoten als ausführbare Skripte enthält, liegt in dem Ordner `autonomous`. Die gesamte Fahrroutine wird über den Kommandozeilenbefehl `roslaunch` mit Angabe des Pakets und dem gewünschten launch-File gestartet. Wird der EVObot über SSH-Befehl ohne eine grafische Benutzeroberfläche bedient, sollte der Befehl

```
roslaunch autonomous lanefollowergray.launch
```

ausgeführt werden. Ist jedoch der Desktop aktiv, kann der Befehl

```
roslaunch autonomous lanefollowergray_visual.launch
```

aufgerufen werden. Hierbei wird dem Benutzer zusätzlich das eingelesene Kamerabild, die detektierten Fahrspuren und die Verkehrszeichenerkennung als Image Stream ausgegeben.

4 OpenCV

OpenCV ist eine freie Programmbibliothek mit diversen Algorithmen spezielle für die Bildverarbeitung und maschinelles Sehen. Die Bibliothek kann ich gängige Programmiersprachen wie C, C++, Python oder Java eingebunden werden. Durch die zahlreichen Funktionen lassen sich unter Anderem auf eine übersichtliche Art Algorithmen zur Gesichts- und Objekterkennung, zur umfangreichen Filterung und Klassifizierung oder zum Maschinellen Lernen umsetzen. Gängige Algorithmen wie beispielsweise eine *Canny Edge Detection* können zur Einarbeitung ebenfalls auf `jetsonhacks.com` nachvollzogen und entsprechende Beispiele umgesetzt werden. ALternativ bietet OpenCV eine sehr umfangreiche *Dokumentation* rund um die Funktionalitäten der Bibliothek.

5 Python

Sämtliche, auf dem EVObot umgesetzten Funktionen wurden in der universellen Programmiersprache Python mit der Version 2.7 verfasst. Als Entwicklungsumgebung kommt *PyCharm* in der quelloffenen Community Version zum Einsatz. Damit ist eine übersichtliche Versionskontrolle und eine automatische Codevervollständigung im Python-Stil möglich.