



Hochschule Karlsruhe  
Technik und Wirtschaft  
UNIVERSITY OF APPLIED SCIENCES

**Näher dran.**



**EVOMOTIV**

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





**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 19. 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 19. September 2018





# Kurzfassung

## Titel der Arbeit in deutscher Sprache

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur 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, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur 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
$\mu$ C	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
DBC	Data Base for CAN
DLC	Data Length Code
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
ROS	Robot Operating System
SAE	Society of Automotive Engineers
SG	Steuergerät
UDS	Unified Diagnostic Services
UML	Unified Modeling Language



# Inhaltsverzeichnis

<b>Kurzfassung</b>	<b>I</b>
<b>Abstract</b>	<b>III</b>
<b>Abkürzungsverzeichnis</b>	<b>V</b>
<b>1. Einleitung</b>	<b>1</b>
1.1. Motivation . . . . .	1
1.2. Zielsetzung der Arbeit . . . . .	2
<b>2. Grundlagen</b>	<b>5</b>
2.1. Autonomes Fahren . . . . .	5
2.1.1. Autonomiestufen . . . . .	6
2.2. Bussysteme . . . . .	8
2.2.1. Kommunikationsmodell . . . . .	9
2.2.2. Controller Area Network CAN . . . . .	10
2.2.3. CAN-Protokoll: Physical Layer . . . . .	10
2.2.4. CAN-Protokoll: Data Link Layer . . . . .	12
<b>3. Das EVObot-Projekt</b>	<b>17</b>
3.1. Das Modellfahrzeug . . . . .	17
3.2. Eingesetzte Sensorik . . . . .	18
3.3. Electronic Control Units . . . . .	18
3.4. ROS - Robot Operating System . . . . .	20
<b>4. Diagnosesystem</b>	<b>23</b>
4.1. Konzept der Fehlerdiagnose . . . . .	23
4.1.1. Anforderungen an die Fehlerdiagnose . . . . .	25
4.1.2. Möglichkeiten zur Umsetzung einer Fehlerdiagnose . . . . .	27
4.2. Aufbau einer Datenkommunikation auf CAN-Bus . . . . .	28
4.2.1. Fahrzeugseitige Busanbindung . . . . .	28
4.2.2. Empfängerseitige Busanbindung . . . . .	32
4.3. Implementierung der Diagnosefunktion . . . . .	34
4.3.1. Einlesen der Sensor- und Signaldaten . . . . .	35
4.3.2. Wandlung der Speichervariablen . . . . .	38
4.3.3. Versenden der CAN-Botschaften . . . . .	39

## *Inhaltsverzeichnis*

4.3.4. Interpretation und Visualisierung der CAN-Botschaften . . . . .	41
4.4. Ergebnisbetrachtung . . . . .	46
4.4.1. Test und Validierung . . . . .	46
4.4.2. Mehrwert der Diagnosefunktion . . . . .	50
<b>5. Kritische Analyse der implementierten Algorithmen</b>	<b>53</b>
5.1. Zeitsynchronisation der Programmknoten . . . . .	53
5.2. Fahrspurerkennung . . . . .	53
5.3. Verkehrszeichenerkennung . . . . .	53
5.4. Längs- und Querregelung im Fahralgorithmus . . . . .	53
<b>6. Zusammenfassung und Ausblick</b>	<b>55</b>
6.1. Zusammenfassung . . . . .	55
6.2. Ausblick . . . . .	56
<b>Tabellenverzeichnis</b>	<b>59</b>
<b>Abbildungsverzeichnis</b>	<b>61</b>
<b>Literaturverzeichnis</b>	<b>66</b>
<b>A. Anhang</b>	<b>67</b>
A.1. Auszug aus dem Datenblatt SN65HVD230 . . . . .	67
A.2. Einarbeitungsleitfaden Projekt EVObot . . . . .	69



# 1. Einleitung

Zu Beginn der Arbeit sollen zunächst die Hintergründe und damit die Motivation für die Auswahl und Bearbeitung des Themas der Masterarbeit aufgezeigt werden. Darauf folgt eine kurze Beschreibung der Aufgabenstellung in Form einer Zielsetzung.

## 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. Diverse Unfallstatistiken zeigen, dass die Anzahl Getöteter und Verletzter im Straßenverkehr zwar stetig sinkt, die Zahlen jedoch seit einigen Jahren stagnieren [1]. Die Weiterentwicklung diverser Fahrerassistenzsysteme könnte die Verkehrssicherheit weiterhin erhöhen und damit dazu beitragen, die Zahl der Unfallopfer im Straßenverkehr weiter zu reduzieren. Maßgebend für den Einzug innovativer Fahrerassistenzsysteme bis hin zur Realisierung automatisierter oder autonomer Fahrfunktionen sind die Entwicklungsarbeit und der Innovationstrieb von Hersteller und Zulieferer der Automobilindustrie zu nennen. Aktuell arbeiten alle namhaften Automobilhersteller daran, die Vision vom autonomen Fahren in naher Zukunft Wirklichkeit werden zu lassen.

Die Forschung und Entwicklung im Bereich künftiger Fahrfunktionen und effizienter Mobilität stellt die gesamte Automobilindustrie jedoch vor besondere Herausforderungen. So gilt es nicht nur die Wünsche und Ansprüche der Kunden zu erfassen um Anforderungen technisch zu realisieren, sondern diese vor Markteinführung auch umfangreich zu testen und gegen potentielle Ausfälle oder unerwünschtes Fehlverhalten abzusichern. Darüber hinaus muss der Hersteller über den gesamten Produktlebenszyklus hinweg den vollen Funktionsumfang ohne technische Mängel der sicherheitskritischen Systeme gewährleisten können. Diese Aspekte und der stetig steigende Konkurrenzkampf um Marktanteile unter den Mitbewerbern setzt forschende Unternehmen unter einen enormen Innovationsdruck. Entwicklungsdienstleistungsunternehmen und Ingenieursgesellschaften wie die *Evomotiv GmbH* können mit ihrer Arbeit hier einen entscheidenden Beitrag leisten. Das Unternehmen mit Wurzeln in der Automobilindustrie unterstützt daher eine Vielzahl an Kunden über den gesamten Produktentstehungsprozess hinweg und übernimmt komplexe Technologieprojekte rund um die Fahrzeugentwicklung.

## 1. Einleitung

Da die Evomotiv GmbH selbst ein hohes Potential in der künftigen Entwicklung innovativer Fahrfunktionen sieht, hat sich das Unternehmen bereits ein breites Knowhow im Bereich der Fahrerassistenzsysteme aufgebaut. Dieses Firmenwissen soll künftig um automatisierte Fahrfunktionen erweitert werden. Durch eigenständige Systemlösungen können Alleinstellungsmerkmale und gegebenenfalls eigene Produkte entstehen. Aufgrund dessen wurde das Projekt *EVObot* ins Leben gerufen. Das erklärte Projektziel ist es, ein Fahrzeug aufzubauen, das als Demonstrator für automatisierte Fahrfunktionen dienen soll. Dazu wurde ein Modellfahrzeug im Maßstab 1:10 aufgebaut, das ausgewählte autonome Fahrfunktionen nach dem Vorbild des studentischen Hochschulwettbewerbs *Carolo-Cup* [2] bewältigen soll. Das Entwicklungsprojekt soll für das Unternehmen sowohl als Experimentierplattform für Systeme und Funktionen dienen, als auch einen Wissenstransfer und die Ausbildung von Studierenden und Absolventen in Form von Projekt- oder Abschlussarbeiten ermöglichen.

### 1.2. Zielsetzung der Arbeit

In einer vorangegangenen Abschlussarbeit [3] wurde bereits ein Modellfahrzeug im Maßstab 1:10 als Demonstrator aufgebaut und einige automatisierte Fahrfunktionen teilweise umgesetzt. Dazu wurde geeignete Hardware und Sensorik ausgewählt und hierauf vernetzte Softwarefunktionen umgesetzt. Dabei bestand die Anforderung jedoch nicht darin, ein Modellfahrzeug aufzubauen, welches vollständig autonom in seiner realen Umgebung agiert, sondern lediglich einzelne Funktionen in einem gewählten Szenario zu Demonstrationszwecken darzustellen.

Das Ziel der vorliegenden Arbeit ist es nun, dieses Projekt weiterzuführen und den Funktionsumfang neuer Systeme so umzusetzen, dass sich diese vollständig in das bestehende Projekt integrieren. Der Verlauf der vorangegangenen Abschlussarbeit hat gezeigt, dass sich bei der Entwicklung und Umsetzung eines solch komplexen Gesamtsystems durchaus eine Reihe von Herausforderungen ergeben können. So war es während dem Entwicklungsprozess nicht möglich, eine angemessene Visualisierung und Aufzeichnung der Daten zu realisieren. Dies ist jedoch besonders für die Analyse der implementierten Algorithmen und zu Applikation der Systemparameter essenziell. Ebenso bestand keine Möglichkeit, potentielle Fehler aufzudecken und das Systemverhalten während der Laufzeit angemessen auswerten zu können. Daher soll ein Konzept eines Diagnosesystems für eine mögliche Fehlerdiagnose am Demonstratorfahrzeug entworfen werden, die dem Vorbild einer realen Diagnosekommunikation in Fahrzeugsystemen entspricht. Eine strikte Einhaltung der Entwicklungsstandards rund um Diagnosesysteme im automotiven Umfeld steht dabei jedoch nicht im Fokus der Arbeit. Das Diagnosesystem soll durch den Einsatz einer geeigneten Schnittstelle aufgebaut und eine Funktion zur Fehler- und Zustandsüberwachung implementiert werden. Im Anschluss daran sollen die implementierten Teilfunktionen und Algorithmen zur Bildverarbeitung und Fahrzeugregelung hinsichtlich ihrer Tauglichkeit in diesem Projekt analysiert werden. Abschließend sollen weitere mögliche Fahrfunktionen

## *1.2. Zielsetzung der Arbeit*

und potenzielle Verbesserungsansätze des Projektes aufgezeigt werden. Den genannten Arbeiten wird eine umfassende Einarbeitung in den aktuellen Projektstand und eine Grundlagenrecherche hinsichtlich autonomer Fahrfunktionen, Bussystemen der Fahrzeugtechnik und Bildverarbeitungsprozesse vorangestellt.



## 2. Grundlagen

Die vorliegende Schrift ist grundsätzlich als Forschungs- und Entwicklungsarbeit des automatisierten Fahrens in der Automobilbranche anzusiedeln. Es werden zunächst nötige Begrifflichkeiten definiert und in aller Kürze der industrielle Entwicklungsfortschritt hin zum vollautomatisierten Fahren vorgestellt und damit ein Projektbezug geschaffen.

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 [4] des Verfassers übernommen wurde.

### 2.1. Autonomes Fahren

Fast alle aktuellen am Markt befindlichen Fahrzeuge bieten eine Vielzahl verschiedener Fahrerassistenzsysteme. Dies wurde in den vergangenen Jahren sowohl durch stetig steigenden Kundenforderungen nach mehr Sicherheit und einem erhöhten Fahrkomfort, als auch durch den Innovationsdrang der gesamten Automobilindustrie möglich. Gängige Fahrerassistenzsysteme sorgen dabei in kritischen Situationen für eine erhöhte Fahrstabilität, halten automatisch Abstand zum vorausfahrenden Fahrzeug oder unterstützen den Fahrer bei Parkmanövern. Durch den kontinuierlichen Einzug innovativer Sensorik, leistungsstarker Steuergeräte und einer stetigen Optimierung der Datenverarbeitung ist hier jedoch noch längst nicht Schluss. Verschiedenen Einschätzungen zufolge, wird das autonome Fahren in 10 bis 20 Jahren marktreif sein [5]. 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. Auf dem Weg dahin, gibt es jedoch zunächst etliche technische, rechtliche und ethische Herausforderungen auf nationaler und internationaler Ebene zu bewältigen. In den Medien stößt man immer wieder auf unterschiedliche Meinungen und Erklärungen im Zusammenhang mit dem automatischen, automatisierten oder autonomen Fahren. Daher gilt es, diese Begrifflichkeiten zunächst klar zu definieren und voneinander abzugrenzen.

## 2. Grundlagen

### 2.1.1. Autonomiestufen

Das autonome Fahren wird umgänglich mit einem Autopiloten assoziiert, der sämtliche Fahrmanöver auf Stabilisierungs-, Bahnführungs- und Navigationsebene ohne ein menschliches Eingreifen durchführen kann. Im konkreten Zusammenhang mit mobilen Robotern spricht man eher von einer Autonomie, wenn sich ein Roboter in seiner Umgebung selbstständig bewegen und agieren kann. Der Roboter agiert jedoch nur so lange autonom, wie seine Energieversorgung dies zulässt.

Um Missverständnisse zu vermeiden und einheitliche Entwicklungsziele auf internationaler Ebene setzen zu können, hat die *Society of Automotive Engineers* SAE International in der Norm *SAE J3016* [6] sechs Stufen definiert, um Begriffe für straßengebundene Kraftfahrzeuge mit Systemen zum autonomen Fahren einheitlich zu beschreiben. Die sechs Autonomiestufen der Norm sind in Tabelle 2.1 gelistet. Diese Interpretation der Klassifizierung beschreibt, welche Aufgaben das System selbst wahrnimmt und welche Anforderungen dabei an den Fahrer gestellt werden.

Ein Wechsel der Autonomiestufe tritt bedingt durch die Ausweitung der Assistenzfunktionen und deren Systemintegration auf. Es gilt zu beachten, dass nach dieser Definition bis zur Autonomiestufe vier ein Fahrzeugführer jederzeit in der Lage sein muss, die Fahraufgabe vollständig und nahtlos zu übernehmen.

Rein passive Systeme wie Spurverlassenswarner oder Totwinkelwarner sind bereits seit einigen Jahren verfügbar und bilden in Autonomiestufe 0 den Einstieg in die Entwicklung hin zu vollautomatisierten Fahrzeugsystemen. Als Vertreter der Autonomiestufe 1 sind Systeme wie adaptive Geschwindigkeitsregelung oder Spurhalteassistenten zu nennen. Diese Systeme übernehmen zeitweise eine Längs- oder Querführung und sind heute flächendeckend in der Serie zu finden. Eine Kombination aus Längs- und Querführung ist Basis für die Autonomiestufe 2. Als Beispiel hierfür ist ein Stauassistent zu nennen, der bei geringen Geschwindigkeiten die Längs- und Querführung des Fahrzeuges übernimmt. Aufgrund der aufwändigen Kombination unterschiedlicher Sensorsysteme und der daraus resultierenden Sensordatenfusion sind solche Funktionen aktuell lediglich in hochklassigen Fahrzeugen zu finden. In der Autonomiestufe 3 sind solche Systeme für höhere Geschwindigkeitsbereiche zu finden. Einige Fahrzeughersteller bieten inzwischen bereits Systeme wie Autobahnassistenten oder Staupiloten an, bei denen der Fahrer das System auch bei erhöhten Geschwindigkeiten nichtmehr dauerhaft überwachen, jedoch seine volle Aufmerksamkeit sicherstellen muss. Die vierte Autonomiestufe bedingt eine hohe Automatisierung des Systems, das bei einem Ausfall oder einer Funktionsstörung nicht auf einen eingreifenden Fahrzeugführer angewiesen ist. Beispiele für solche Systeme sind fahrerloses Parken oder hochautomatisiertes Fahren in städtischen Bedingungen. Solche Systeme befinden sich aktuell bereits in der Testphase, bedürfen jedoch äußerst robuste Systeme mit einer hohen Ausfallsicherheit, weshalb die Gesetzeslage hochautomatisierte Funktionen in Serienfahrzeugen noch nicht zulässt. Das erklärte Entwicklungsziel bildet das vollauto-

---

<sup>1</sup>Anwendungsfälle beinhalten Straßentypen, Geschwindigkeitsbereiche und Umfeldbedingungen.

Tabelle 2.1.: Sechs Stufen der Autonomie für straßengebundene Kraftfahrzeuge nach *SAE J3016* [7].

Stufe	Bezeichnung	Definition
0	keine Automation	Fahrer führt dauerhaft Längs- <i>und</i> Querführung aus. Kein eingreifendes Fahrzeugsystem aktiv.
1	Assistenzsysteme	Fahrer führt dauerhaft Längs- <i>oder</i> Querführung aus. System übernimmt die jeweils andere Funktion.
2	Teilautomatisierung	Fahrer <i>muss</i> das System dauerhaft überwachen. System übernimmt Längs- <i>und</i> Querführung in einem spezifischen Anwendungsfall <sup>1</sup> .
3	Bedingte Automatisierung	Fahrer <i>muss</i> das System <i>nicht</i> mehr <i>dauerhaft</i> überwachen. Fahrer muss potentiell in der Lage sein, zu übernehmen. System übernimmt Längs- <i>und</i> Querführung in einem spezifischen Anwendungsfall. Systemgrenzen werden alle erkannt. Ausreichende Zeitreserve zur Übernahme.
4	Hochautomatisierung	Kein Fahrer erforderlich im spezifischen Anwendungsfall. System kann im spezifischen Anwendungsfall <i>alle Situationen</i> automatisch bewältigen.
5	Vollautomatisierung	Von <i>Start</i> bis <i>Ziel</i> ist kein Fahrer erforderlich. System übernimmt Fahraufgabe <i>vollumfänglich</i> in allen Situation.

matisierte oder autonome Fahren, welches die Autonomiestufe 5 darstellt. Bereits heute existieren diverse Konzepte für solche Systeme, eine genaue Marktreife ist jedoch noch nicht absehbar [8, 7].

Die SAE-Norm ist grundlegend für straßengebundene Kraftfahrzeuge definiert und kann daher nur bedingt auf autonome mobile Roboter angewendet werden. Das Demonstratorfahrzeug, das in dieser Arbeit beschrieben wird, soll sämtliche Fahraufgaben in gewählten Szenarien ohne menschliche Eingriffe bewältigen. Daher ist dieses Projekt in den Autonomiestufen vier und fünf anzusiedeln. Jedoch sollen nicht sämtliche Fahraufgaben des Straßenverkehrs abgedeckt werden. Es ist zur Laufzeit weder ein direkter Eingriff in die Längs- und Querführung des Fahrzeuges vorgesehen, noch eignet sich diese Projekt für eine direkte Portierung in Systeme, die im Straßenverkehr eingesetzt werden. Trotzdem eignet sich ein solches Demonstrationsmodell, um einzelne Funktionen zum Erreichen der Autonomiestufen aufzuzeigen und deren Umsetzung zu demonstrieren.

### 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 Fahr-sicherheit, Motorsteuerung und Komfortsysteme erforderten zwingend einen sicheren und schnellen Informationsfluss zwischen den kommunizierenden Steuergeräten, sodass ein ver-teiltes und vernetztes Gesamtsystem entstand. Alle für eine jeweilige Funktion benötigten Daten konnten systemweit zur Verfügung gestellt werden. Aus dem zunehmenden Elek-trifizierungsgrad ergaben sich heutige moderne Elektronikarchitekturen im Kfz in Form von seriellen Bussystemen. Das Akronym Bus stammt [9] nach von *Binary Utility System*, was auf ein drahtgebundenes Übertragungsmedium mit Anschluss an alle Systemkom-ponenten 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 Informatio-nen 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 ver-knüpften 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* über-nommen. 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 [10].

Heute standardisierte und gängige Datenkommunikationssysteme sind in Tabelle 2.2 auf-geführt [10]. 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 unter-schiedlichster 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 sicherheitsrelevan-ten 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 [11]. 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 [12].

Tabelle 2.2.: Klassifikation serieller Bussysteme [10]

Bussystem	Typische Anwendung	Maximale Datenrate	Übertragungsmedium	Sicherheitsanforderung
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* [13] festgehaltene *Open System Interconnection* OSI-Schichtenmodell kann eine standardisierte und herstellerübergreifende Kommunikation im gesamten Busnetzwerk erzielt werden. Das OSI-Schichtenmodell wird in Tabelle 2.3 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.

## 2. Grundlagen

Tabelle 2.3.: Zusammenfassung des OSI-Schichtenmodells aufgeteilt in Layer, Schicht und Funktionen [13]

	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 Busan Kopplung, 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 [14, 15].

### 2.2.3. CAN-Protokoll: Physical Layer

Die unterste Schicht im OSI-Modell beschreibt die physikalische Busan Kopplung. Das Übertragungsmedium des CAN-Busses wird in den häufigsten Fällen als verdrehte Zwei-

drahtleitung, 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* ( $\mu C$ ), 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.

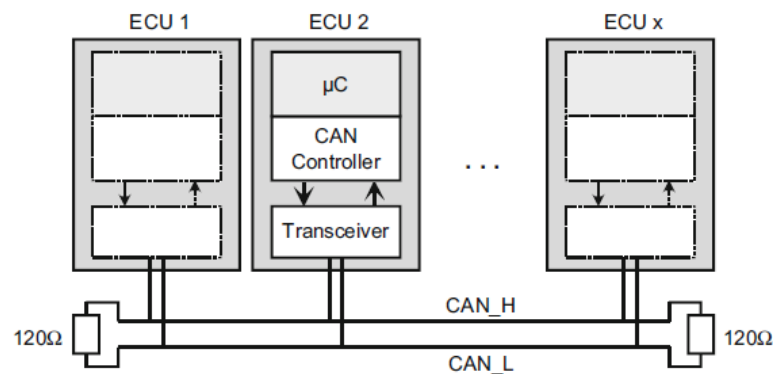


Abbildung 2.1.: CAN-Netzwerk: Ein einzelner CAN-Knoten besteht aus einem Mikrocontroller, einem CAN-Controller und einem CAN-Transceiver. Der Abschlusswiderstand unterdrückt Busreflexionen [15].

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 2.2 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 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 [15, 16].

## 2. Grundlagen

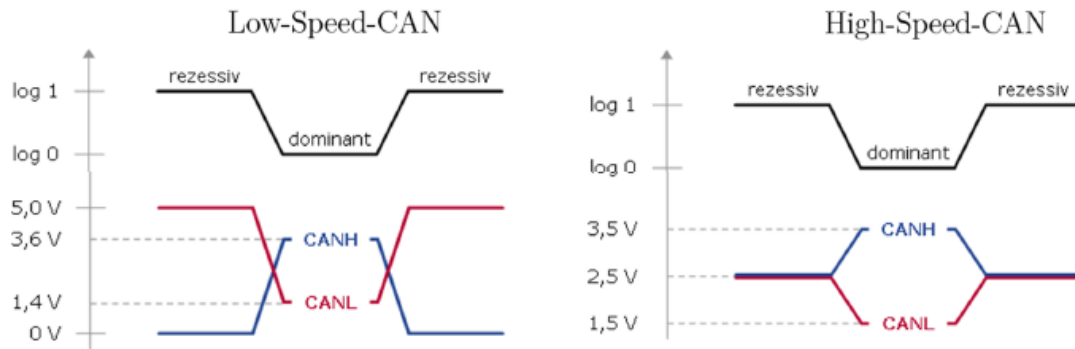


Abbildung 2.2.: Signalpegel Low-Speed-CAN (links) und High-Speed-CAN (rechts) [16].

### 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 [16]. Die einzelnen Komponenten und deren Aufgaben werden nachstehend in der zugehörigen Tabelle 2.4 näher erklärt.

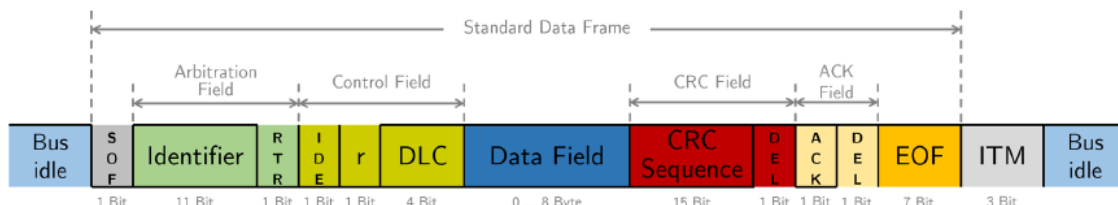


Abbildung 2.3.: Aufbau des Standard CAN Data-Frames [16].

Während der gesamten Laufzeit der Netzwerkverbindung senden die Busknoten ihre Botschaften nach dem *Broadcast*-Verfahren unaufgefordert und ereignisgesteuert auf den Bus.

Tabelle 2.4.: Funktionen der einzelnen Felder im Data-Frame [14, 15].

Feld	Name	Länge	Funktion
kein Feld	SOF	1 Bit	Der dominante <i>Start of Frame</i> 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 <i>Remote Transmission Request</i> kennzeichnet den Frametyp. Im Fall des Data Frames wird das Bit dominant übertragen.
Control Field	IDE	1 Bit	Die <i>Identifier Extension</i> 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 <i>Data Length Code</i> 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-Feld	CRC	15 Bits	Der <i>Cyclic Redundancy Check</i> bildet eine Prüfsumme der Nutzdaten und dient der Fehlererkennung bei der Botschaftsübertragung.
	DEL	1 Bit	Auf das CRC-Feld folgt ein rezessives <i>Delimiter</i> -Bit.
ACK-Feld	ACK	1 Bit	Der Sender einer Botschaft setzt das <i>Acknowledgement</i> -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 <i>Delimiter</i> -Bit begrenzt.
kein Feld	EOF	7 Bit	Der <i>End of Frame</i> markiert das Ende einer Botschaft mit sieben rezessiven Bits.

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.

## 2. Grundlagen

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 essenzielle 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* 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 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<sup>2</sup> 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

---

<sup>2</sup>im Worst Case bei acht Datenbytes

Buskommunikation ausgeschlossen werden [14, 15, 16].





## 3. Das EVObot-Projekt

Wie eingangs erläutert ist das erklärte Ziel des Firmenprojektes, ein Modellfahrzeug als Demonstrator für moderne Antriebs- und Fahrerassistenzfunktionen zu entwickeln. Hauptziel ist es, autonome Fahrfunktionen real abbilden zu können. Daneben stehen Ziele wie die Weiterbildung von Studierenden und Absolventen in der Programmierung eingebetteter und vernetzter Systeme, sowie in innovativen Datenkommunikationstechnologien.

Um einen Überblick über das Gesamtsystem zu erhalten, sollen zunächst das Fahrzeug, die Komponenten und die eingesetzten Softwareelemente in ihrer Ausgangssituation beschrieben werden.

### 3.1. Das Modellfahrzeug

Die mechanische Basis bildet das Modellfahrzeug X10E der Firma *Tamiya-Carson Modellbau GmbH & Co. KG*. Das Chassis des RC-Fahrzeuges hat den Maßstab 1:10 und bietet ein umfangreich einstellbares Fahrwerk. Auf dem Chassis ist eine Acrylglasplatte montiert, auf dem die gesamte Steuereinheit, die Elektronik und die verwendeten Sensoren verbaut sind. Im Fahrgestell sind ein Lithium-Polymer-Akkumulator, eine Motorregleinheit, ein Lenkservomotor und der Antriebsmotor verbaut. Das Antriebsstrang ist mit einem anpassbaren Übersetzungsgetriebe und einem Achsausgleichsgetriebe ausgestattet.

Als Antriebsmotor wird der originale bürstenlose Gleichstrommotor mit dem mitgelieferten Fahrregler verwendet. Der für den erhöhte Leistungsbereiche ausgelegte Antriebsmotor wird jedoch nicht über einen Drehzahlwert, sondern über den Leistungsbedarf geregelt. Abhängig vom Fahrzeuggewicht und der Haftreibung beim Anfahren kann dieser Regelwert variieren, was eine Herausforderung in der zuverlässigen Längsregelung bei geringen Fahrgeschwindigkeiten zur Folge hat. Der Antriebsmotor weist mit je 10 Spulenwicklungen einen  $kV$ -Wert<sup>3</sup> von 3600  $kV$  auf. Bei einer maximalen Spannungsversorgung von 7,4 V kann der Motor an der Ausgangswelle also eine theoretische Drehzahl von  $26\,640\text{ min}^{-1}$  erreichen.

Als Aktor zur Querregelung kommt ebenfalls ein mitgelieferter Servomotor zum Einsatz. Dieser wird mit einer Betriebsspannung von 5 V versorgt und kann einen Lenkwinkel von circa  $\pm 35^\circ$  ausgeben.

---

<sup>3</sup>Der  $kV$ -Wert beschreibt die Leerlaufdrehzahl eines Elektromotors pro 1 V angelegter Spannung. Die Bezeichnung sollte in diesem Zusammenhang nicht mit der Einheit *Kilovolt* verwechselt werden.

## 3.2. Eingesetzte Sensorik

Zur Umsetzung der autonomen Fahrfunktion und weiterer Teilfunktionen ist rund um das Modellfahrzeug verschiedene Sensorik verbaut. Zentrales Element der Sensorkomponenten bildet das Kamerasystem. Auf der Grundplatte des Fahrzeuges ist eine handelsübliche USB-Webcam der Firma *Logitech Europe S.A.* Das Kamerasystem dient zur zuverlässigen Erkennung einer Fahrspur und wird für eine Verkehrszeichenerkennung verwendet. Die Kamera wurde im Laufe der Arbeit ausgetauscht, hierzu sei auf Kapitel 5 verwiesen.

An der Front des Fahrzeuges ist ein Ultraschallsensor *HC-SR04* verbaut. Dieser wird für eine Objekterkennung und Abstandsbestimmung verwendet. Es lassen sich Entfernungen in einem Bereich zwischen 2 cm und 3 m mit einer ungefähren Auflösung von 3 mm bei einer maximalen Messfrequenz von 50 Hz bestimmen. Ein weiterer Ultraschallsensor ist auf der rechten Fahrzeugseite verbaut und dient zu einer Parklückenerkennung. Diese Funktion ist aber aktuell nicht im Fahrzeug umgesetzt.

Zentral auf der Grundplatte montiert befindet sich zusätzlich eine Inertiale Messeinheit der Firma *Bosch Sensortec GmbH*. Auf einem Breakout Board sind Beschleunigungs- und Drehratensensor zu einer Einheit zusammengefasst. Somit können lineare Beschleunigungen, Winkelgeschwindigkeiten und die magnetische Orientierung in allen Raumrichtungen erfasst werden. Dieser Sensor war gedacht, um eine umfangreiche Querregelung auf Basis der Fahrgeschwindigkeit, dem Kurvenverhalten und der Gierrate zu realisieren. Dies ist jedoch in der aktuellen Konfiguration nicht umgesetzt, weshalb die Inertiale Messeinheit keine Verwendung findet. Sie lässt sich jedoch leicht in die Systemumgebung einbinden.

## 3.3. Electronic Control Units

Auf dem EVObot befinden sich im Wesentlichen zwei Komponenten, die die gesamten Steuer- und Regelungsaufgaben des Fahrzeuges übernehmen. Zum Einen wird ein *Arduino UNO* eingesetzt. Dieses Mikrocontrollerboard dient einzig als Schnittstelle zwischen den Sensoren, Aktoren und dem Hauptrechner. Der Arduino liest lediglich die Sensorik in Form von Ultraschallsensoren ein und steuert die Aktorik der Längs- und Querregelung an. Somit findet auf dem Arduino keinerlei Datenverarbeitung im eigentlichen Sinne statt. Er kommuniziert jedoch dauerhaft über eine serielle Schnittstelle mit der Hauptrecheneinheit.

Die zentrale Entwicklungsplattform des gesamten Projektes stellt hingegen das Entwicklerkit *NVIDIA Jetson TX2* dar. Dieser kompakte aber äußerst effiziente Computer wurde speziell für Anwendungen der Künstlichen Intelligenz und Computer Vision entwickelt. Das leistungsstarke Modul verfügt über sechs Prozessorkerne und acht Gigabyte internem Arbeitsspeicher und eignet sich damit sowohl für die parallele Signalverarbeitung, als auch für rechenintensive Grafikanwendungen zur Bildverarbeitung. Auf der Adapterplatte des Computers befinden sich zahlreiche Anschlüsse für umfangreiche Erweiterungen und Systemanbindungen. Auf dem Hardwareboard selbst läuft das Betriebssystem Ubu-

tu als eine Linux-Distribution. Damit ist das Jetson TX2 als vollwertiger Computer mit einer gewohnten grafischen Benutzeroberfläche zu nutzen. Auf dem Betriebssystem des Jetson TX2 läuft das Software-Framework *Robot Operating System* ROS. Da ROS einen gesonderten Stellenwert in dieser Arbeit einnimmt, wird im folgenden Abschnitt 3.4 noch einmal ausführlich auf diese Umgebung eingegangen. Zur Verarbeitung der eingelesenen Rohdaten des Kamerasystems wird die freie Programmbibliothek *OpenCV* verwendet. Die umfangreiche Bibliothek liefert diverse Algorithmen und Programmfunktionen speziell für die Bildverarbeitung und maschinelles Sehen. Sämtliche umgesetzten Teilfunktionen und Algorithmen zur Bildvorverarbeitung, Spurerkennung, Verkehrszeichenerkennung, Hinderniserkennung und Längs- bzw. Querreglung sind in der universellen Programmiersprache *Python* mit Hilfe der integrierten Entwicklungsumgebung *PyCharm* verfasst. Zum besseren Verständnis sind in Abbildung 3.1 noch einmal alle Systemkomponenten entsprechend ihren Layern des Schichtenmodells software- und hardwareseitig als gesamte Systemarchitektur aufgezeigt. Eine Sonderstellung nimmt hierbei der Middleware Layer durch das zusätzliche Software-Framework ROS ein.

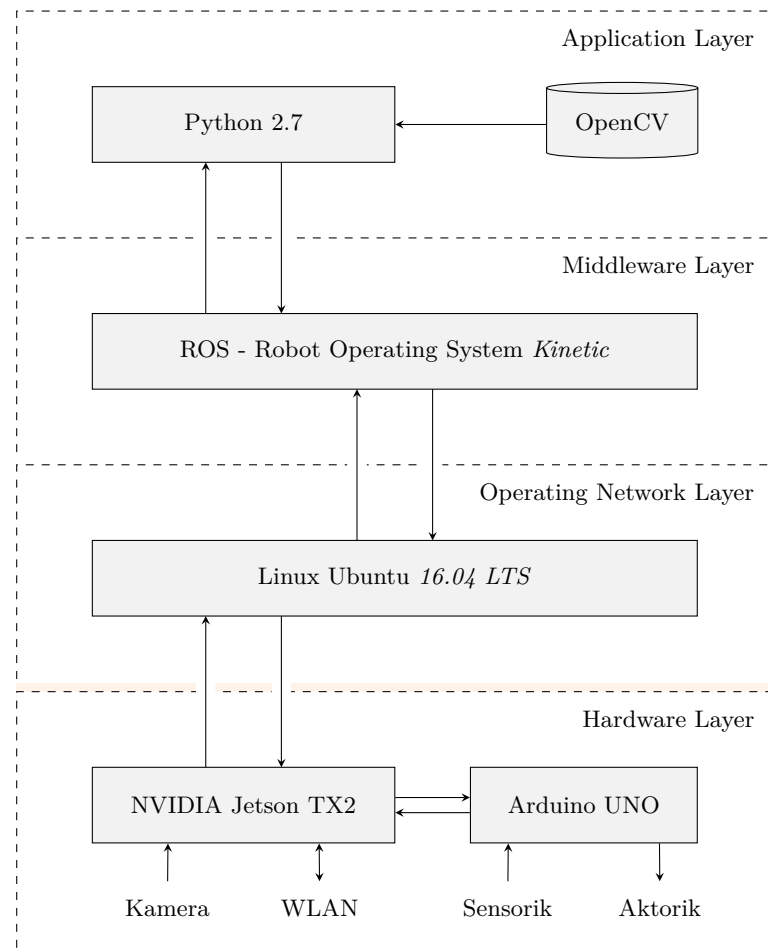


Abbildung 3.1.: Systemarchitektur der Komponenten des EVObots im angepassten Schichtenmodell

## 3.4. ROS - Robot Operating System

ROS ist eine Open Source Middleware, die speziell für Robotikanwendungen entwickelt wurde. Das umfangreiche Framework stellt zahlreiche Programmbibliotheken, Werkzeuge, Gerätetreiber, Visualisierungstools und Möglichkeiten zum Nachrichtenaustausch. Aufgrund dieser weitreichenden Funktionen wurde ROS als Meta-Betriebssystem zur Realisierung der Fahrfunktionen in diesem Projekt gewählt. Dabei ist ROS nicht als eine eigenständige Anwendung zu betrachten, sondern stellt eine komplex angelegte Middleware für eine anwendungsneutrale Prozesskommunikation mit flexiblen und simplen Datenstrukturen dar. ROS bietet Systemdienste wie die Integration einer Hardwareabstraktionsschicht oder Low-Level-Gerätesteuerungen. Da das Arbeiten mit ROS in dieser Arbeit einen essenziellen Aspekt darstellt, werden im Folgenden einige wichtige Begriffe und das fundamentale Konzept von ROS kurz erläutert [17].

- **Packages:**

Die Softwareprojekte werden in ROS in *Packages* verwaltet und organisiert. Ein Paket enthält alle Laufzeitprozesse eines Projektes, die Programmbibliothek, Datensets, Konfigurationsdateien und gegebenenfalls Softwarekomponenten von Drittanbietern. Die Verwaltung eines Projektes in Paketen ermöglicht eine einfache Modularität der Funktionalitäten.

- **Nodes:**

*Nodes* stellen in ROS ausführbare Prozesse dar. Diese Instanzen sind verteilt über die Kommunikationsinfrastruktur, können aber durch verschiedene Mechanismen zur Laufzeit miteinander kommunizieren oder Daten austauschen. Typischerweise werden die thematisch entkoppelten Funktionen eines Projektes in einzelne Knoten verteilt.

- **Topic:**

*Topics* sind Datenbussysteme, über die zur Laufzeit Nachrichten in asynchroner Form ausgetauscht werden. Dazu verwendet ROS eine *Publisher/Subscriber*-Semantik. Die Knoten publizieren dabei Informationen auf verschiedene Topics, während andere Knoten diese Topics beliebig abonnieren können, um die Informationen auf dem Datenbus einzulesen. Diese Funktionsweise ermöglicht es, einzelne Softwarekomponenten unabhängig entwickeln und in bestehende Projekte integrieren zu können.

- **Messages:**

Die Informationen auf den Datenbussen der Topics werden in Form von *Messages* in standardisierten Datenstrukturen versendet. Es existiert eine Vielzahl von nativen Datenformaten einzelner Botschaften (boolean, integer, floating point, usw.), es lassen sich aber auch eigene Messages aus Arrays der definierten Datenformate generieren.

- **Services:**

Gegenüber dem Broadcast-Prinzip der Subscriber und Publisher lassen sich mit *Ser-*

### 3.4. ROS - Robot Operating System

*vices* in ROS Informationen auch auf explizite Requests erhalten. Dabei wird von einem Knoten eine definierte Informationen durch einen Service von einem einzelnen Knoten angefordert. Der Client-Node antwortet dann auf diese Anfrage mit dem Senden der entsprechenden Message an den anderen Knoten.



## 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 umsetzen. 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* [18] als „Nichterfüllung einer Anforderung“ oder nach *DIN 55350* [19] 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 [20]. 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

#### 4. Diagnosesystem

ist. Diese Fehlerwirkung kann sich je nach Schwere des Fehlers als ein abweichender Rückgabewert einer Berechnung oder bis hin zum Totalausfall des Systems äußern [21].

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 [22] 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 Fehlerursachen diagnostiziert, kann

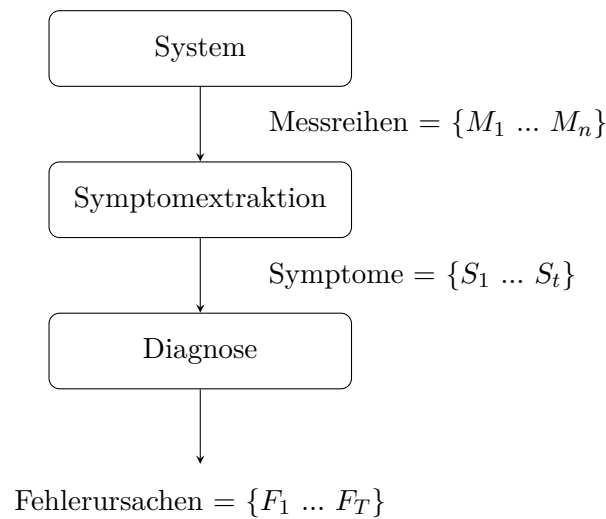


Abbildung 4.1.: Ablauf einer Diagnose im Kfz-Umfeld [23]

also formal als eine Abbildung von Fehlersymptomen  $S$  auf Fehlerursachen  $F$  angesehen werden

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

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 Analyse-



werkzeug 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 2.3) 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 zu einander 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 [15] und [24] 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.

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.
- Das Diagnosesystem muss ein echtzeitnahes Daten-Monitoring ermöglichen.
- Die Diagnosefunktion muss vom Anwender gestartet und gestoppt werden können.

#### 4. Diagnosesystem

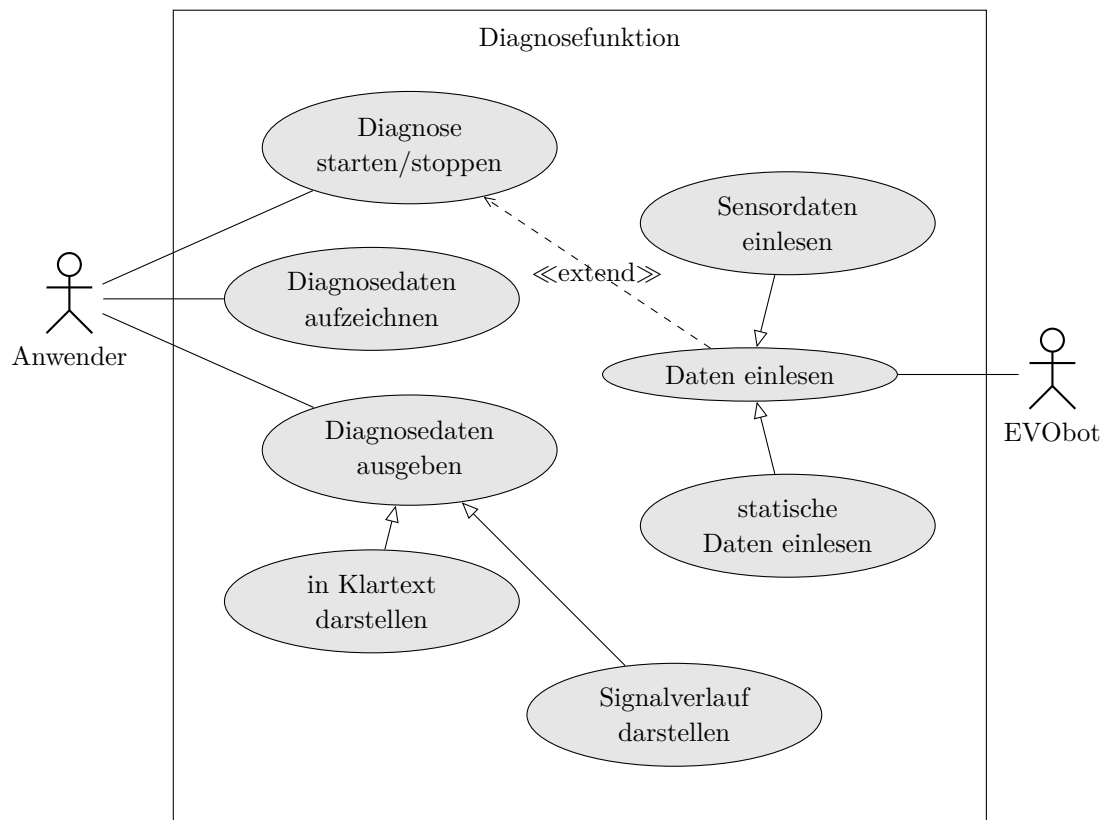


Abbildung 4.2.: Use-Case-Diagramm einer Diagnosefunktion für den Fahrdemonstrator

- 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 soll eine übersichtliche Visualisierung für den Anwender bieten.
- Die Einbindung in das bestehende System soll ohne grundlegende Modifikationen in den implementierten Programmteilen 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 Wertvariablen 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.

#### 4. Diagnosesystem

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<sup>4</sup> 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. Tabelle 2.3) 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 Hard-

---

<sup>4</sup>Als *online* wird hier nicht die aktive Verbindung zum Internet bezeichnet, sondern die intakte und betriebsbereite Datenverbindung zwischen EVObot und dem externen Ausgabegerät.

#### 4.2. Aufbau einer Datenkommunikation auf CAN-Bus

wareanbindung 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 Erweiterungsschaltung umfangreiche Programmbibliotheken, um die Verwendung einer CAN-Anbindung in der Arduino Entwicklungsumgebung zu ermöglichen. 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 NVIDIA 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 Architectureinheit 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.

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  $\mu\text{C}$  über den Controller (vgl. Abbildung 2.1). Der Transceiver setzt die empfangenen Daten in einen Spannungspegel um, sodass ein differentielles Spannungssignal 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 ver-

#### 4. Diagnosesystem

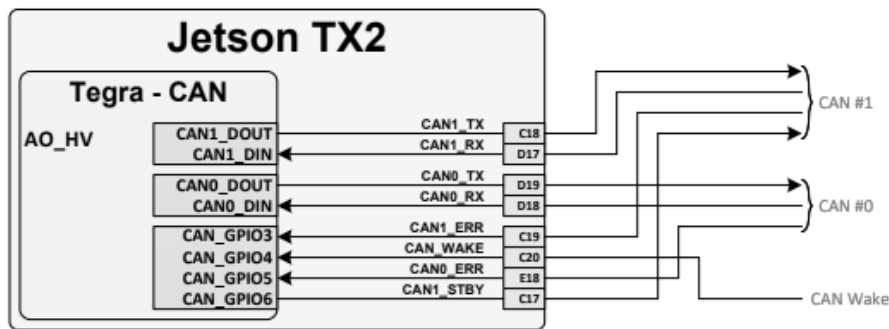


Abbildung 4.3.: Layout des CAN-Interfaces auf dem NVIDIA Jetson TX2 nach dem NVIDIA Product Design Guide [25]

fügbaren 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 3,3V 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.

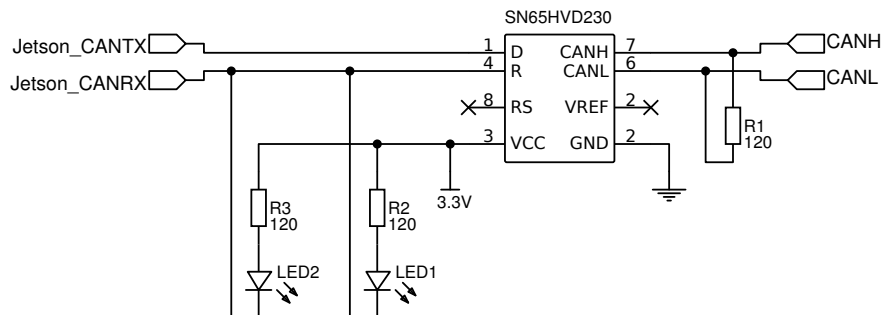


Abbildung 4.4.: Schaltplan des SN65HVD230 CAN-Transceivers

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 [26]. 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:

<b>cansend:</b>	Senden einer Single Frame Botschaft
<b>cangen:</b>	Erzeugen von zufälligem CAN-Busverkehr
<b>canplayer:</b>	Wiedergabe von CAN-Logfiles
<b>candump:</b>	Anzeigen, Filtern und Protokollieren von CAN-Botschaften
<b>cansniffer:</b>	Anzeige inhaltlicher Unterschiede zweier CAN-Botschaften
<b>canbusload:</b>	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 verdrehte Zweidrahtleitung 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
```

#### 4. Diagnosesystem

```
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*



#### 4.2. Aufbau einer Datenkommunikation auf CAN-Bus

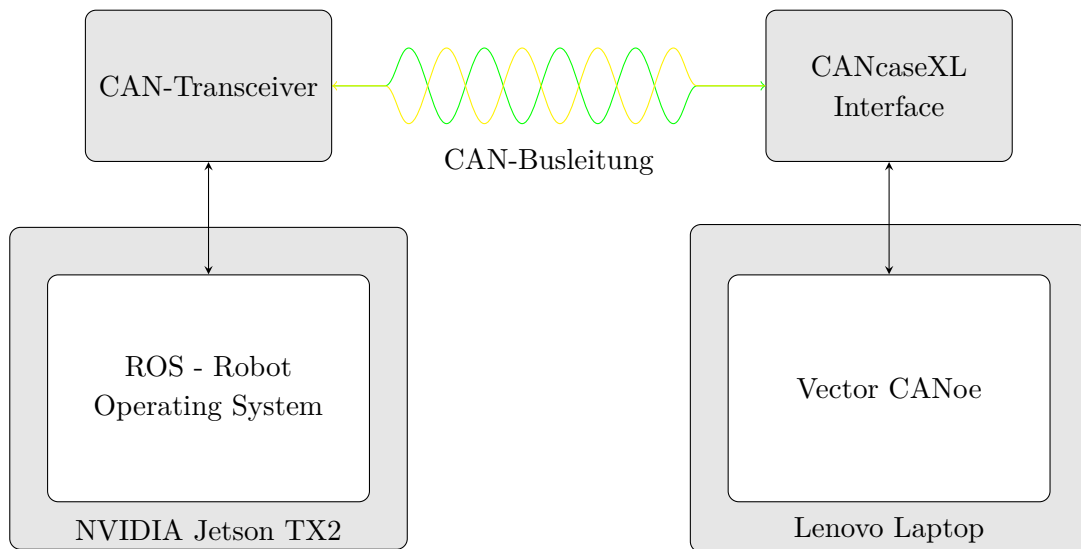


Abbildung 4.5.: Schematischer Aufbau der Buskommunikation zwischen dem EVObot und dem Diagnoserechner

*GmbH* ist ein USB-Interface, das eine physikalische Ankopplung eines 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 [27].

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 [28].

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

## 4. Diagnosesystem

Verfügung. Ebenso können eigene grafische Benutzeroberflächen erstellt werden, wodurch komplexe Simulationsumgebungen erleichtert werden [29].

### 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. Besonderes Augenmerk liegt dabei auf dem umgesetzten Datenverarbeitungsprozess. Wie bereits in Abschnitt 3.3 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<sup>5</sup> erstellt, der sich in die Knotenstruktur des ROS Entwicklungsprojektes eingliedert. Der bisherige Programmgraph wird aus Abbildung 4.6 ersichtlich.

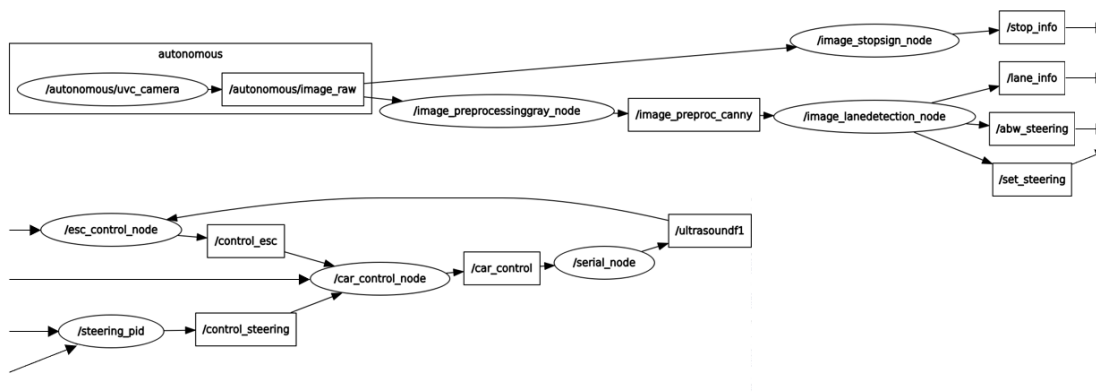


Abbildung 4.6.: Programmgraph des initialen ROS-Paketes *autonomous*, mit dem Befehl `rqt_graph` automatisch generiert

Wie zuvor in Abschnitt 3.3 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 Schaubild 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 maschinennahen Programmiersprache C++, als auch in der benutzerfreundlichen und universellen Skriptsprache Python betreiben. Python bietet

<sup>5</sup>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.

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 [17]. 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

#### 4. Diagnosesystem

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	int16 min_range	minimaler Abstand in m
		int16 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 Parameter eingelesen. Diese werden vor Start der Fahroutine im ROS-spezifischen *launch-File* parametrisiert 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.

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 Fahrspur-

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

kennung 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 äußerst 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 booleschen 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 Wertebereiche 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 [30]. 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 0x60E. 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 Variablenwert 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_pwm)
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 ver-

sendet 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 [31]. 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 [32]. 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.

#### 4. Diagnosesystem

Im Anschluss wird der Botschaftsrahmen sämtlicher CAN-Frames beschrieben. Es werden die Identifier den Botschaftsobjekten zugeordnet und alle Nutzdaten der Botschaften mit dem Inhalt 0x00 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 Identifier 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 hochprioreren Wert 0x01 und enthält die Botschaft den Bytewert 0x01, 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 Identifier 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 Bit 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 Signalwerten 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 0x01 und dem Nachrichteninhalte 0x00 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. [4] S. 18 ff.).

#### 4.3.4. Interpretation und Visualisierung der CAN-Botschaften

Die empfangenen Busbotschaften im hexadezimalen Rohformat können nun mit dem Software-Werkzeug CANoe wieder interpretiert und zur Darstellung der Signale in ihrer physikalischen Einheit gebracht werden. Hierzu ist eine vollständige Kommunikationsbeschreibung des gesamten vernetzten CAN-Bussystems notwendig. In einer solchen Kommunikationsdatenbasis werden die Eigenschaften des CAN Netzwerks, die an den Bus angeschlossenen ECU sowie die CAN-Botschaften und Signale beschrieben. Hierzu wurde zunächst die proprietäre Dateibeschreibung *Data Base CAN DBC* der Firma *Vektor Infor-*

#### 4. Diagnosesystem

*matik GmbH* entwickelt. Dieses Dateiformat hat sich inzwischen aber herstellerübergreifend als Quasi-Standard etabliert. CAN-Datenbanken definieren für jedes Signal Regeln zur Umwandlung in physikalische Einheiten. Vector bietet zum Editieren einer solchen Kommunikationsbeschreibung das Erweiterungswerkzeug *CANdb++* an. Innerhalb einer DBC-Datei werden folgende Daten eines einzelnen Signals definiert und gespeichert:

- Signalname
- Position (Start-Bit) und Größe (Anzahl der Bits) innerhalb einer Botschaft
- Byte-Reihenfolge (Motorola oder Intel)
- Skalierungsfaktor und Offset
- Wertebereich
- Initialwert
- physikalische Einheit
- Kommentar mit Informationen zum Signal in Klartext

Darüber hinaus werden diese definierten Signale den Botschaften mit eindeutigen Identifiern zugeordnet. Da in der Datenbasis die genauen Bitpositionen der Signale niedergeschrieben sind, kann eine aus bis zu 8 Byte Nutzdaten bestehende Botschaft nach Erhalt wieder in seine einzelnen Signale zerlegt und vollständig interpretiert werden. Eine Zuordnung der Signale wird in Abbildung 4.7 exemplarisch anhand der Botschaft `ultraschall_data` aufgezeigt.

Die farbliche Aufteilung macht ersichtlich, dass diese Botschaft drei Signale beinhaltet. In Byte 7 ... 4<sup>6</sup> liegt das Signal `ultraschall_front_entfernung` das ursprünglich eine Variable mit Datentyp `float32` beinhaltet, also 32 Bit in dieser Botschaft einnimmt. Es folgen in Byte 3 ... 2 die minimale und in Byte 1 ... 0 die maximale Entfernung des Ultraschallsensors als Byte Arrays gewandelt aus einem `int16` Datentypen.

An dieser Stelle gilt zu erwähnen, dass in der gesamten Umsetzung sämtliche Signale jeweils nur auf ganze Bytes gelegt wurden. Auch wenn der technisch maximal mögliche Wertebereich eines Parameters mit weniger als 8 Bit (respektive 16, 32, ... Bit) abbildbar ist, wurden immer nur ganze Bytes in den Botschaften zugewiesen. Dies ist zwar nicht speichersparend und widerspricht dem Paradigma einer niedrigen Busauslastung zur Steigerung der Effizienz, war jedoch durch die aufwendige Wandlung der Speichervariablen in Abschnitt 4.3.2 technisch nicht anders umsetzbar. Die erhöhte Speichernutzung ist allerdings aufgrund der geringen Anzahl der Busbotschaften und den nicht sicherheitskritischen Daten vertretbar.

Neben der Beschreibung der Signale wird in der Datenbasis ebenfalls die Kommunikationsmatrix, oder kurz K-Matrix, definiert. In dieser K-Matrix ist festgelegt, welche ECU welche Botschaften sendet und wiederum welche Botschaften von welchem Knoten eingelesen werden. Die einfache Kommunikationsmatrix ist auf Botschaftsebene definiert, die Signalinhalte und damit die Nutzdaten der CAN-Frames sind damit zunächst irrelevant.

---

<sup>6</sup>Das Motorola-Dateiformat (Big-Endian) wird „von hinten nach vorne“, also von Least Significant Bit nach Most Significant Bit gelesen.

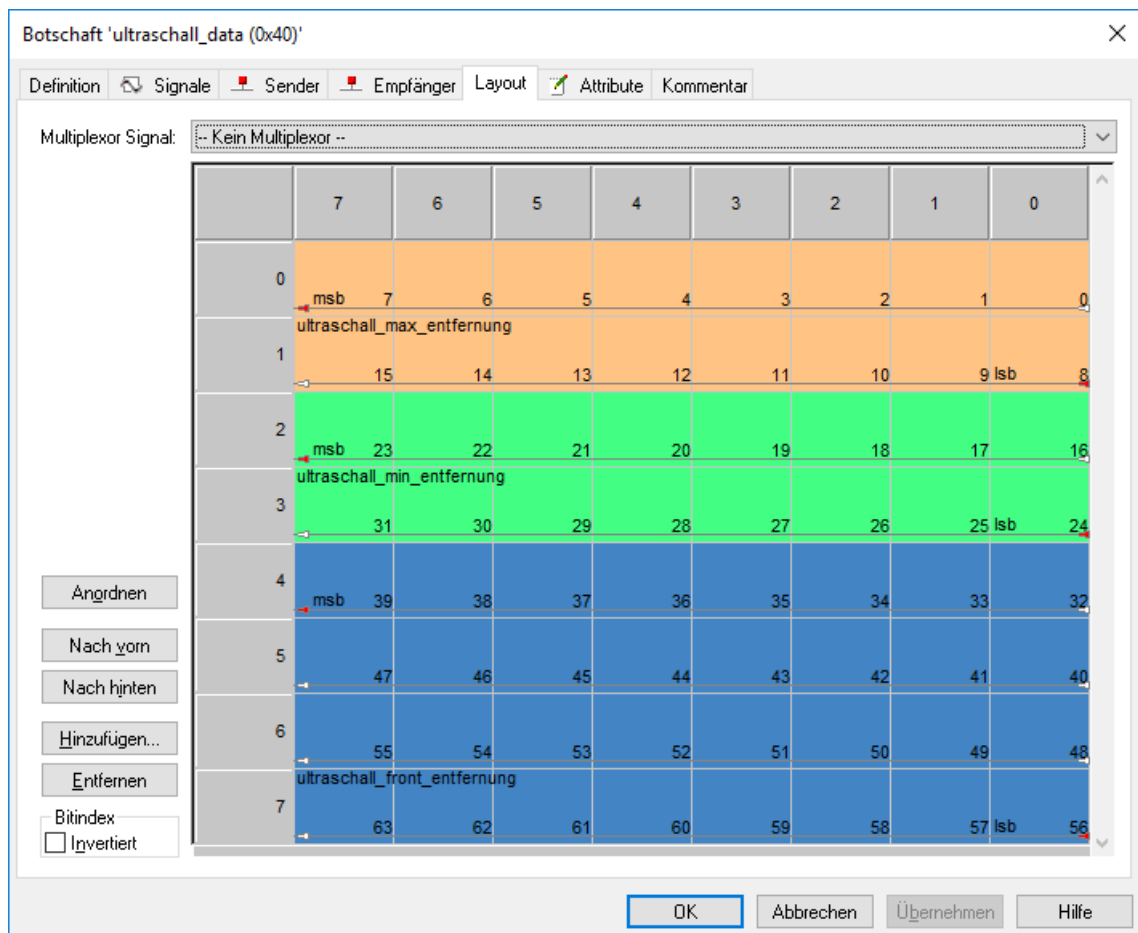


Abbildung 4.7.: Signalzuweisung innerhalb der Botschaft `ultraschall_data` mit dem Editierungstool CANdb++

Lediglich die IDs werden festgelegt, um zwischen hoch- und niederprioren Nachrichten unterscheiden zu können. Die Akzeptanzfilterung einer Botschaft wird hardwareseitig auf dem CAN-Controller durchgeführt. Sobald ein CAN-Frame von einem Knoten empfangen wird, wird zunächst die formale Korrektheit der Botschaft geprüft, anschließend wird der Identifier mit den Strukturen der eingepflegten K-Matrix verglichen. Stimmt der Identifier mit den IDs überein, die für diesen Knoten als relevant angegeben sind, werden die Nutzdaten in den Empfangspuffer übergeben. Andernfalls wird die Nachricht im Controller verworfen, da die erhaltene Botschaft keine Relevanz für die entsprechende ECU aufweist. Neben dieser Definition einer K-Matrix existiert eine erweiterte Kommunikationsmatrix, die konkrete Informationen über die Sendebedingungen, den DLC, die Zykluszeit und eine genaue Beschreibung der übermittelten Signale beinhaltet. Tabelle 4.3 zeigt in der einfachen K-Matrix den Kommunikationsablauf des CAN-Netzwerkes. Es ist ersichtlich, dass das gesamte Netzwerk aus lediglich zwei Knoten besteht. Diese sind zum einen der EVObot selbst und zum anderen der Laptop mit der aktiven CANoe-Umgebung als Ausgabegerät der empfangenen CAN-Botschaften. Alle elf Botschaften sind der Priorisierung der Identi-

#### 4. Diagnosesystem

fier nach geordnet. Es fällt auf, dass die CAN-Frames zum Busreset und Starten/Stoppen der Diagnosekommunikation die niedrigsten Identifier aufweisen und damit eine hohe Priorität besitzen. Dies soll garantieren, dass diese Botschaft die Busarbitrierung stets gewinnen und verhindert damit, dass die Nachrichten ohne Verzug durch Zwischenspeichern im Sendepuffer auf den Bus gesendet werden. Die Identifier wurden bei der Beschreibung je um den Wert 10 inkrementiert. Dies ermöglicht eine komfortable und übersichtliche Erweiterung der gesamten Datenbasis. Das gesamte Kommunikationsnetzwerk besteht damit nun aus zwei Netzknoten, elf Botschaften und 28 definierten Signalen.

Tabelle 4.3.: K-Matrix des CAN-Netzwerkes für die Buskommunikation zwischen dem EVObot und dem externen Ausgabegerät

Botschaft / Knoten	EVObot	CANoe
reset_bus_message (0x00)	Sender	Empfänger
diagnostic_status_message (0x01)	Empfänger	Sender
lane_data_1 (0x10)	Sender	Empfänger
lane_data_2 (0x11)	Sender	Empfänger
stop_data (0x20)	Sender	Empfänger
eng_data (0x30)	Sender	Empfänger
ultraschall_data (0x40)	Sender	Empfänger
ultraschall_data_gefiltert (0x41)	Sender	Empfänger
static_camera_info (0x50)	Sender	Empfänger
static_pid_info_1 (0x51)	Sender	Empfänger
static_pid_info_2 (0x52)	Sender	Empfänger

In CANoe lassen sich nun die generierten Signale ausgeben und umfangreich analysieren. Im Trace Fenster wird bei aktiver Diagnosekommunikation die gesamte Busaktivität aufgelistet. Bei Eintreffen einer neuen Botschaft wird diese mit ihrem Zeitstempel, der ID, dem Botschaftsnamen und den Nutzdaten in hexadezimaler Form ausgegeben. Es lassen sich hier ebenfalls die Signalwerte der Botschaften in Klartext und in ihrer physikalischen Einheit darstellen. Zur Analyse des Busverkehrs lassen sich Botschaftsfilter einfügen und diverse Marker setzen. Der Inhalt des Trace-Fensters lässt sich bequem aufzeichnen und exportieren, um eine Offline-Analyse durchzuführen.

In einem Grafik-Fenster lassen sich nun die Signalwerte und Umgebungsdaten grafisch als Kurve über der Zeit darstellen. Zum Messen und Auswerten des Signalverlaufs stehen in CANoe diverse Werkzeuge zur Verfügung.

Die ermittelten Signaldaten sollen dem Anwender in Echtzeit ausgegeben und in ihrer physikalischen Form dargestellt werden. Dazu bietet CANoe die Möglichkeit, eine individuelle grafische Benutzeroberfläche zu erstellen, die als Benutzerschnittstelle zur Ein- und Ausgabe von Daten dient. Im Panel wird jedem individuell anpassbarem Steuerelement ein Signal oder eine Systemvariable zugewiesen. Sobald eine Busbotschaft mit dem entspre-

### 4.3. Implementierung der Diagnosefunktion

chenden Signal eingegangen ist, wird das Steuerelement in der grafischen Benutzeroberfläche aktualisiert. Abbildung 4.8 zeigt das fertige Panel bei aktiver Netzwerkverbindung zum Modellfahrzeug.

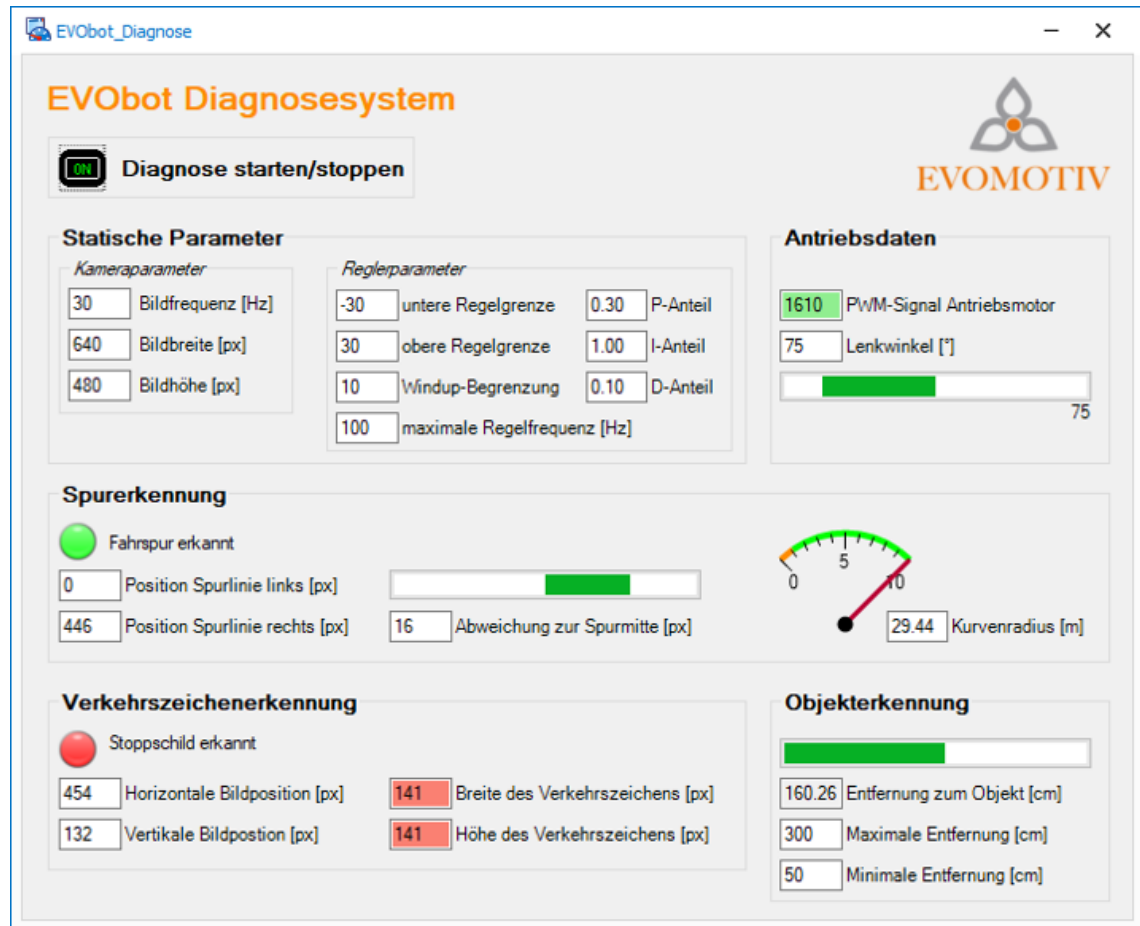


Abbildung 4.8.: Nutzerinterface der Diagnosefunktion. Es werden dem Anwender alle relevanten Informationen in Echtzeit ausgegeben.

Ist der Laptop über das CAN-Interface und die physische Busleitung mit dem Modellfahrzeug verbunden und ist die Fahrroutine in ROS aktiv, lässt sich die Diagnosefunktion und damit der Diagnoseknoten in ROS starten und stoppen. Dem Anwender werden während der Programmlaufzeit so alle Parameter in textueller und visualisierter Form zyklisch aktualisiert ausgegeben. Es ist gut ersichtlich, dass in der Momentaufnahme von Abbildung 4.8 ein Stoppschild durch die Objekterkennungsalgorithmus erkannt wird, daher wird der Antriebsmotor nicht betrieben und das Fahrzeug befindet sich in einem stehenden Zustand vor dem Stoppschild. Der Ultraschallsensor meldet einen Abstand zum nächsten Objekt, der größer als der minimale Abstand ist. Zudem wird eine Abweichung zur Fahrspurmitte nach rechts erkannt, weshalb dem Servomotor ein Lenksignal kleiner 90° und damit eine Korrektur nach links vorgegeben wird.

### 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 4.9 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 Fahroutine und sämtliche Algorithmen zur Bildverarbeitung und der Fahrzeugregelung werden durch die Diagnosefunktion nicht beeinträchtigt.

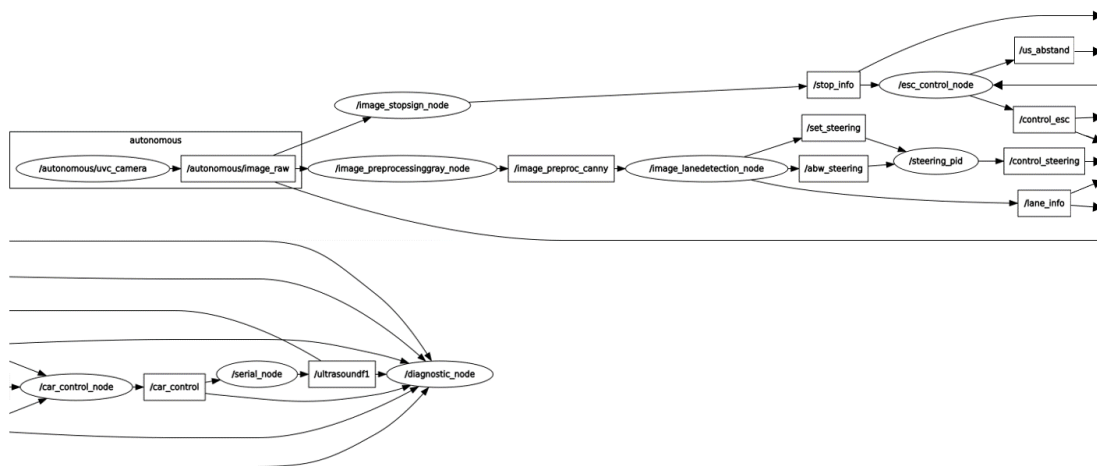


Abbildung 4.9.: Programmgraph des neuen ROS-Paketes *autonomous* inklusive dem Knoten *diagnostic\_node*, mit dem Befehl `rqt_graph` automatisch generiert

Der Anwender kann nun jederzeit mit Hilfe eines CAN-Interfaces sein Laptop über die Bananenstecker der physikalischen CAN-Leitung mit dem EVObot verbinden und die Diagnosefunktion über das Ausgabepanel in CANoe starten und steuern.

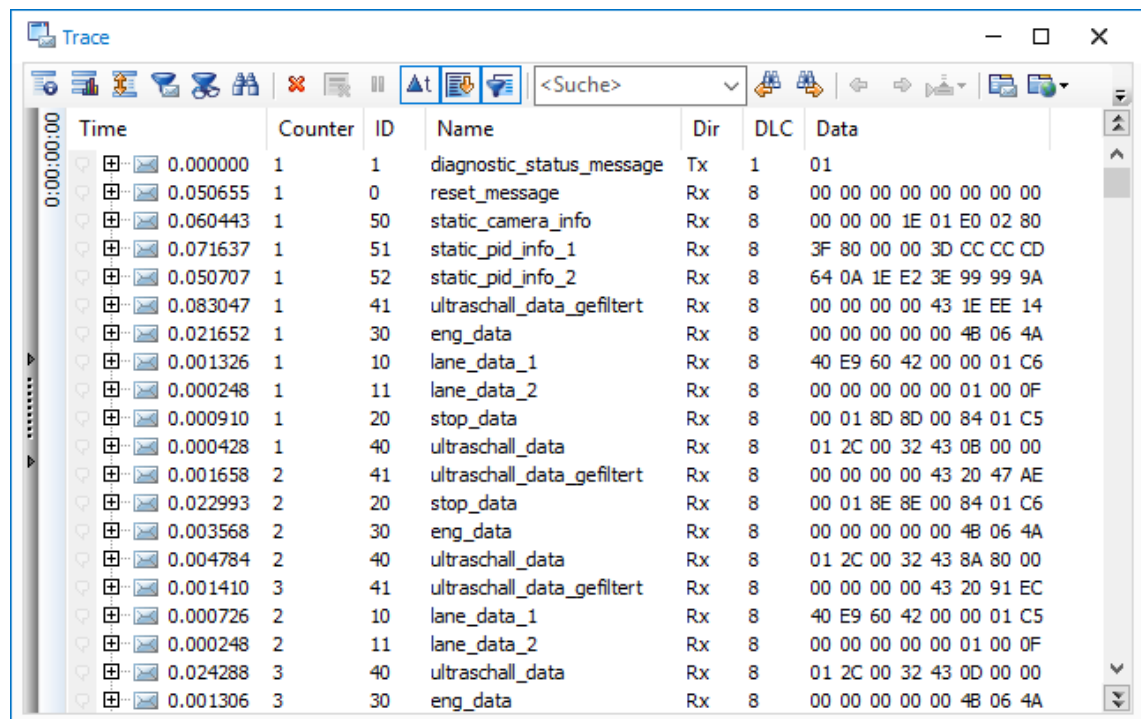
#### 4.4.1. Test und Validierung

Nachdem die Datenübertragung und Ausgabe der Signale umgesetzt wurde, soll das angewandte Verfahren getestet und auf seine Tauglichkeit geprüft werden.

Dazu wurden mehrere Testfahrten mit dem automatisiert fahrenden Roboter unter variierenden Bedingungen durchgeführt. Dabei konnte die korrekte und vollständige Funktionsweise des Diagnosesystems sichergestellt werden. Die Diagnosefunktion arbeitet bei aktivem ROS-Core zuverlässig. Eine Analyse der Prozesse und der Systemlast des Jetson TX2 während der Laufzeit der Fahroutine in ROS zeigt, dass die CPU-Auslastung bei aktivem Diagnoseknoten kurzzeitig um circa 12 Prozent gegenüber deaktiviertem Diagno-

seknoten ansteigt. Dieser Lastanstieg des Prozessors bei Zuschalten des Diagnoseknotens ist in Anbetracht der hohen Leistungswerte des Jetson TX2 absolut vertretbar.

Nach der Beseitigung einiger kleiner Logikfehler in der Implementierung kann nun der `diagnostic_node` in der ROS-Umgebung über das extern ausgegebene Diagnosepanel gestartet und gestoppt werden. Dem Anwender werden bei aktivem Diagnoseknoten in jedem Betriebszustand alle relevanten Parameter durchgängig aktualisiert ausgegeben. Bei einer Unterbrechung oder einem Defekt in der Busverbindung, wird eine Fehlermeldung sowohl auf der Sender-, als auch auf der Empfängerseite ausgegeben.



Time	Counter	ID	Name	Dir	DLC	Data
0.000000	1	1	diagnostic_status_message	Tx	1	01
0.050655	1	0	reset_message	Rx	8	00 00 00 00 00 00 00 00
0.060443	1	50	static_camera_info	Rx	8	00 00 00 1E 01 E0 02 80
0.071637	1	51	static_pid_info_1	Rx	8	3F 80 00 00 3D CC CC CD
0.050707	1	52	static_pid_info_2	Rx	8	64 0A 1E E2 3E 99 99 9A
0.083047	1	41	ultraschall_data_gefiltert	Rx	8	00 00 00 00 43 1E EE 14
0.021652	1	30	eng_data	Rx	8	00 00 00 00 00 4B 06 4A
0.001326	1	10	lane_data_1	Rx	8	40 E9 60 42 00 00 01 C6
0.000248	1	11	lane_data_2	Rx	8	00 00 00 00 00 01 00 0F
0.000910	1	20	stop_data	Rx	8	00 01 8D 8D 00 84 01 C5
0.000428	1	40	ultraschall_data	Rx	8	01 2C 00 32 43 0B 00 00
0.001658	2	41	ultraschall_data_gefiltert	Rx	8	00 00 00 00 43 20 47 AE
0.022993	2	20	stop_data	Rx	8	00 01 8E 8E 00 84 01 C6
0.003568	2	30	eng_data	Rx	8	00 00 00 00 00 4B 06 4A
0.004784	2	40	ultraschall_data	Rx	8	01 2C 00 32 43 8A 80 00
0.001410	3	41	ultraschall_data_gefiltert	Rx	8	00 00 00 00 43 20 91 EC
0.000726	2	10	lane_data_1	Rx	8	40 E9 60 42 00 00 01 C5
0.000248	2	11	lane_data_2	Rx	8	00 00 00 00 00 01 00 0F
0.024288	3	40	ultraschall_data	Rx	8	01 2C 00 32 43 0D 00 00
0.001306	3	30	eng_data	Rx	8	00 00 00 00 00 4B 06 4A

Abbildung 4.10.: Ausschnitt des Trace-Fensters in CANoe bei aktiver Diagnosekommunikation mit dem EVOBot

Abbildung 4.10 zeigt exemplarisch einen Ausschnitt der in CANoe empfangenen Busbotschaften bei aktiver Diagnosefunktion. Zu sehen ist, dass in der ersten Zeile zunächst die `diagnostic_status_message` von dem Laptop an das Fahrzeug versendet wird. Daraufhin sendet der Diagnoseknoten zunächst je einmalig die Botschaften, die die statischen Parameter des Fahrzeuges beinhalten und beginnt dann, die dynamischen Botschaften wiederholt an den Laptop zu senden. Dabei folgen die Botschaften keinem geregelten zyklischen Zeitplan, wie aus der ersten Spalte den Differenzzeiten entnommen werden kann. Dies ist auf die Aktualisierungsrate der eingelesenen Messages und damit auf den differenzierten Rechenaufwand der implementierten Prozesse in den Topics zurückzuführen. Wie bereits erläutert, weist der CAN-Bus ein nicht deterministisches, ereignisgesteuertes Verhalten auf, somit wird eine Busbotschaft immer direkt mit dem Erhalt einer aktualisierten

#### 4. Diagnosesystem

Message und dem Aufruf der entsprechenden Callback-Funktion versendet. Sollten mehrere Botschaften zeitgleich übermittelt werden, greift das Arbitrierungsverfahren, sodass sich der Identifier mit der höchstwertigen Priorität durchsetzt. Niederpriorie CAN-Frames verharren im Sendepuffer und werden im Anschluss versendet. Das ereignisgesteuerte Sendeverfahren führt dazu, dass die Gesamtzahl der gesendeten Botschaften pro Identifier je nach Prozesszeit bei einer langen Messdauer auseinanderdriftet. Daher wird es empfohlen, die Prozesszeiten der einzelnen Programmknoten in ROS global anzugleichen. Dieses Erkenntnis wird in Abschnitt 5.1 noch einmal aufgegriffen.

Das Zeitverhalten und die Wiederholungsrate eines Signals setzt sich grundsätzlich aus der Verarbeitungszeit im `diagnostic_node` ab dem Zeitpunkt des Erhalts einer neuen Message, der Übertragungsdauer auf dem Bus und der Verarbeitungszeit auf dem Empfangsrechner bis zur Ausgabe in CANoe zusammen.

Die benötigte Zeit zum physischen Versenden einer CAN-Botschaft ergibt sich allgemein aus

$$t_{Sende} = n_{Frame} * t_{Bit}. \quad (4.1)$$

Dabei ist  $n_{Frame}$  die Gesamtlänge einer Botschaft in Bit. Bei einem CAN Standard Data Frame mit einem 11 Bit großen Identifier beträgt die Gesamtlänge 108 Bit. Wird nun noch im Worst Case ein maximales Bit-Stuffing hinzugenommen, erhöht sich die maximale Botschaftslänge auf bis zu 132 Bit. Hinzu kommt das 3 Bit breite Intermission-Feld, das ein Ende einer Busbotschaft signalisiert.  $t_{Bit}$  ist die Sendezeit eines Bits, womit sich eine Datenrate zu  $1/T_{Bit}$  ergibt. Die Datenrate wird in der gesamten Konfiguration mit 500 kBit/s gewählt. Damit ergibt sich die maximale Sendedauer einer Botschaft zu

$$t_{Sende} = 135 \text{ Bit} * \frac{1 \text{ s}}{500 \text{ kBit}} = 270 \mu\text{s}. \quad (4.2)$$

Betrachtet man nun lediglich die variablen Parameter, werden während der Laufzeit wiederholt insgesamt sieben Botschaften versendet. Somit ließen sich im Extremfall sämtliche variablen Sensor- und Signaldaten in einer Zeit von 1,89 Millisekunden auf dem Bus übertragen. Dies entspricht einer theoretischen Wiederholungsfrequenz von circa 530 Hz. Diese Rechnung soll jedoch nur das theoretisch maximal Mögliche aufzeigen. Die reale Wiederholungsrate liegt aufgrund der variierenden Prozesszeiten der ROS-Knoten und der Übermittlung der Messages in den Topics deutlich darunter. Zudem wird für die Umrechnung und Zuweisung der Daten in CAN-Frames ebenfalls je eine gewisse variierende Prozesszeit benötigt. Die durchschnittliche effektive Zykluszeit, also der tatsächlich beobachtete zeitliche Abstand, der periodischen Botschaften liegt bei der realen Datenkommunikation bei 33,3 Millisekunden mit einer empirischen Standardabweichung von etwas 3 Millisekunden. Dies führt bei wiederholten Messungen zu einer durchschnittlichen Buslast von 4,54 %. Geht man davon aus, dass die Summe der Botschaften pro Zeiteinheit deutlich geringer als die maximale Übertragungskapazität ausfallen sollte, liegt dieser Wert deutlich unterhalb von 50 % Buslast, die nach einer Faustregel als Maximalwert anzunehmen sind, um



eine konsistente Datenübertragung zu gewährleisten. Somit bietet das umgesetzte Kommunikationsnetzwerk erheblichen Spielraum für künftige Erweiterungen des Bussystems. In der Praxis kommt durch das nichtdeterministische Verhalten des CAN-Bus zusätzlich eine Latenz bei der Datenübertragung hinzu. Diese Latenzzeit ist Abhängig von der Anzahl der Busteilnehmer und der Busauslastung und kann daher stark schwanken. Die Latenz kann allerdings im konkreten Anwendungsfall durch die geringe Anzahl der Busbotschaften und die niedrige Buslast als sehr gering angenommen werden. Damit lässt sich nun die Güte der Echtzeitfähigkeit des Systems bewerten. Die Literatur definiert keine konkreten Grenzen, wann ein System als echtzeitfähig gilt, die Verarbeitung der Daten muss lediglich garantiert schnell genug für die jeweilige Anwendung erfolgen. Da es sich bei der Buskommunikation um die Übertragung von nicht sicherheitskritischen Daten handelt und die Ausgabe der Sensor- und Signaldaten lediglich passiv stattfindet, kann die Echtzeitfähigkeit der umgesetzten Diagnosekommunikation als durchaus geeignet bewertet werden. Nimmt man die Trägheit der mechanischen Wirkungskette des Gesamtsystems im Fahrroboters hinzu, ist das Zeitverhalten der gesamten CAN-Übertragung vollkommen vertretbar. Mit der Anforderung einer echtzeitnahen Überwachung der Systemzustände kann diese durch die Verwendung eines Bussystems als erfüllt betrachtet werden. Ein Abgleich der umgesetzten Funktionalitäten zeigt, dass sämtliche zu Beginn gestellten Anforderungen eingehalten werden, somit ist das Diagnosesystem vollständig verifiziert, entspricht also den spezifizierten Funktionseigenschaften.

Zuletzt soll zur Vollständigkeit das physikalische Übertragungsverhalten der Bitübertragungsschicht geprüft werden. Dazu wird das Differenzspannungssignal der beiden Busleitungen mit einem Oszilloskop gemessen. Abbildung 4.11 zeigt exemplarisch das Ergebnis dieser Messung.

Hier ist das Differenzspannungssignal der CAN-Botschaft `lane_data_1` dargestellt. Zu sehen sind die CANH-Leitung in blau und die CANL-Leitung in rot. Zunächst fällt das asymmetrische Spannungsverhalten der beiden Leitungen auf. Hierbei handelt es sich jedoch nicht wie vermuten lässt um einen Übertragungsfehler in der CANH-Leitung, sondern dies ist auf den CAN-Transceiver mit einer rezessiven Spannungsversorgung von 3,3 V zurückzuführen. Typischerweise liegt die Gleichtaktspannung bei 5 V Transceivern bei  $V_{cc}/2$ , somit sind die Spannungspegel der CAN-Leitungen symmetrisch. Um ein ähnlich hohes Differenzspannungssignal zu erhalten, liegt die rezessive Gleichtaktspannung bei Transceivern mit geringerer Versorgungsspannung etwas unterhalb dieses Wertes. Dadurch lässt sich eine gleiche Differenzspannung bei geringerem Spannungsniveau erzeugen. Durch diese asymmetrische Verschiebung der Busspannung lassen sich Transceiver mit unterschiedlichen Versorgungsspannungen in einem Netzwerk betreiben, ohne dass die Transceivern aufeinander abgestimmt sein müssen. Schließlich ist für das Einlesen eines CAN-Signals lediglich die tatsächliche Differenzspannung und nicht das Spannungsniveau von Bedeutung. Somit kann der Stromverbrauch eines Transceivers bei gleicher Funktionsweise gesenkt werden [33].

#### 4. Diagnosesystem

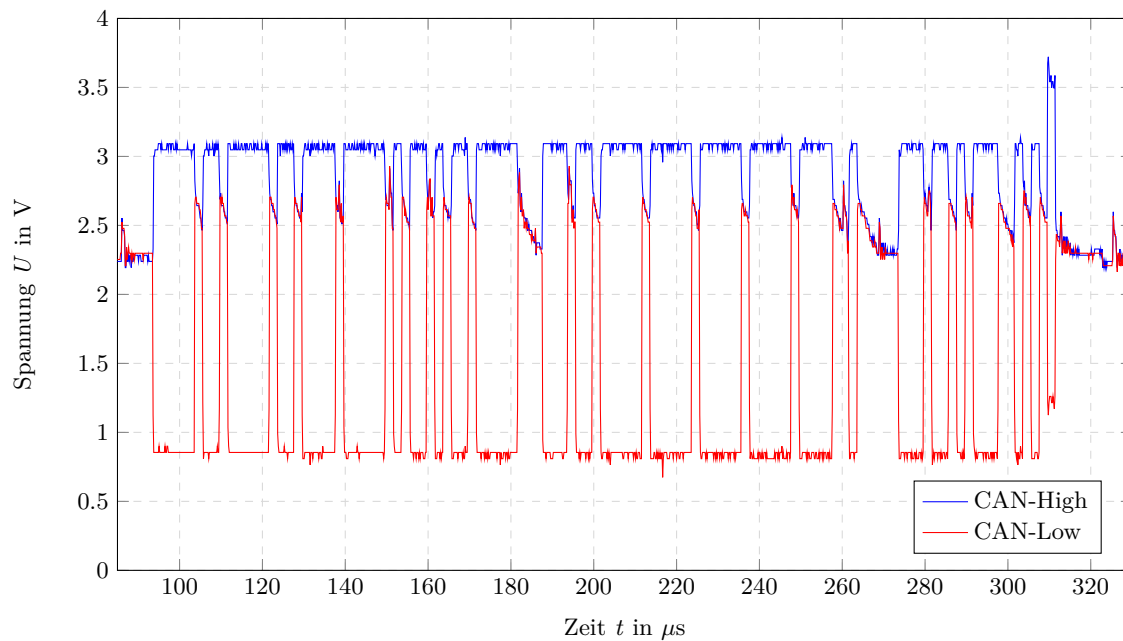


Abbildung 4.11.: Aufzeichnung des Differenzspannungssignal der CAN-Botschaft `lane_data_1` auf der physikalischen Busleitung

Eine genauere Betrachtung der übertragenen Daten zeigt, dass die Sendedauer der exemplarischen Busbotschaft  $234,8 \mu s$  beträgt. Aufgrund der Zustände der Signaldaten traten in der Botschaft 9 Stuff-Bits auf. Korrigiert man nun die Rechnung in Formel 4.1, erhält man exakt die theoretische Sendedauer von 234 Mikrosekunden. Dies zeigt, dass die Botschaft auch zeitlich korrekt und ohne Latenz übertragen wurde.

##### 4.4.2. Mehrwert der Diagnosefunktion

Bei der Umsetzung der gesamten Buskommunikation wurde mehr ein Zustandsüberwachungssystem als ein vollumfängliches Diagnosetool im klassischen Sinne entwickelt. Wie zuvor aufgezeigt, stand eine strikte Einhaltung der diversen Standards einer Diagnosekommunikation nicht im Fokus. Die vollständige Umsetzung der Diagnose nach dem standardisierten ODX Datenmodell zur Beschreibung der Diagnose- und Programmierdaten würde den Umfang der Arbeit in einem begrenzten Zeitrahmen deutlich übersteigen. Dies wäre auch für das vergleichsweise kleine Kommunikationsnetzwerk mit der stark begrenzten Anzahl an Signalen und Busbotschaften nicht verhältnismäßig. Eine Umsetzung nach dem Diagnosestandard würde jedoch auch einen Austauschprozess ermöglichen. Während der Applikation kann so von dem externen Ausgabegerät aus der Datenbestand des Fahrzeuges über die Diagnosekommunikation geändert und somit Parameter extern angepasst werden. Dies setzt allerdings grundlegende und umfangreiche Änderungen im Aufbau sämtlicher Programmteile voraus, um eine Modularität zu gewährleisten. Um Parameter nicht nur

passiv auszugeben, sondern auch aktiv in den Algorithmen ändern zu können, müssen eigens Konfigurationsdateien erstellt werden, die dann von den einzelnen Programmknöten gelesen und beschrieben werden können. Dies war jedoch von Beginn an nicht die Anforderung an das Projekt, weshalb das Diagnosesystem als eine rein passive Zustandsüberwachung und -darstellung aufgebaut wurde.

Die Diagnosefunktion bietet trotzdem einen erheblichen Mehrwert für das gesamte Projekt des automatisiert fahrenden Roboters. So hat sich gezeigt, dass die echtzeitnahe Ausgabe der Systemparameter den Applikationsprozess deutlich erleichtert hat. Zuvor war es nicht möglich, die Fahrzustände auch während der Programmlaufzeit der Fahrroutine zu überprüfen, somit war der Applikationsprozess auf eine rein heuristische und aufwändige *Trial and Error* Methode beschränkt. Inzwischen lässt sich durch die echtzeitnahe Visualisierung der Fahrdaten das Fahrverhalten auch ohne eine Probefahrt des Fahrzeuges prüfen. Darüber hinaus können die Fahrdaten bequem aufgezeichnet und im Anschluss unter Einbezug sämtlicher Signale und deren zeitlichem Verlauf umfangreich analysiert werden.

Durch die Einbindung einer Datenbasis lassen sich je nach künftigen Erweiterungen und Funktionen des Fahrzeuges später mühelos und unbedenklich weitere Signale in die Diagnosefunktion mitaufnehmen. Damit sind Flexibilität und Modularität der Diagnosekommunikation sichergestellt. Durch die Datenübertragung über die CAN-Busleitung und die Einhaltung des CAN-Protokolls wird eine Datenkonsistenz und Datensicherung über das gesamte Kommunikationsnetzwerk sichergestellt.

Es gilt zu erwähnen, dass durch die umgesetzte Diagnose nicht direkt auf eine mögliche Fehlerursache in der Implementierung der Fahralgorithmen geschlossen werden kann. Jedoch wird hierdurch nicht nur die Fehlerwirkung sichtbar, sondern auch der Fehlerzustand der einzelnen Signalparameter dargestellt. Damit kann die tatsächliche Fehlerursache leicht auf ein Minimum eingeschränkt und so in dem Programmcode gefunden werden. Nach diesem Vorgehen konnten diverse Softwarebugs und Logikfehler im Programmcode aufgedeckt und beseitigt werden. Bei künftigen Erweiterungen der Fahrfunktionen können die neu implementierten Funktionen umgehend mit dem Diagnosesystem geprüft und damit eine Fehlerfortpflanzung verhindert werden.



## **5. Kritische Analyse der implementierten Algorithmen**

Dieses Kapitel setzt sich mit den Algorithmen zur Bildverarbeitung und der Fahrzeugregelung auseinander. Es sollen die implementierten Teilfunktionen beschrieben und hinsichtlich Ihrer Tauglichkeit für dieses Projekt untersucht werden. Zudem werden die eigens umgesetzten Änderungen vorgestellt. Es sollen Verbesserungsvorschläge und mögliche künftige Erweiterungen der Fahrfunktionen aufgezeigt werden. Dabei muss klar erwähnt werden, dass ein Großteil der Fahrfunktionen nicht im Zeitraum der Masterarbeit umgesetzt wurde, sondern aus einer vorherigen studentischen Arbeit [3] hervorgingen.

### **5.1. Zeitsynchronisation der Programmknotten**

### **5.2. Fahrspurerkennung**

### **5.3. Verkehrszeichenerkennung**

### **5.4. Längs- und Querregelung im Fahralgorithmus**



## 6. Zusammenfassung und Ausblick

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

## 6. Zusammenfassung und Ausblick

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, consectetur 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, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur 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 pulvi-



nar 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, consectetur 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

2.1. Sechs Stufen der Autonomie für straßengebundene Kraftfahrzeuge nach <i>SAE J3016</i> [7]. . . . .	7
2.2. Klassifikation serieller Bussysteme [10] . . . . .	9
2.3. Zusammenfassung des OSI-Schichtenmodells aufgeteilt in Layer, Schicht und Funktionen [13] . . . . .	10
2.4. Funktionen der einzelnen Felder im Data-Frame [14, 15]. . . . .	13
4.1. Beschreibung der im Diagnoseknoten abonnierten Topics mit veränderlichen Variablen und den damit eingelesenen Variablenwerte . . . . .	36
4.2. Beschreibung der im Diagnoseknoten eingelesenen Topics mit statischen Variablenwerten . . . . .	37
4.3. K-Matrix des CAN-Netzwerkes für die Buskommunikation zwischen dem EVObot und dem externen Ausgabegerät . . . . .	44



# Abbildungsverzeichnis

2.1.	CAN-Netzwerk: Ein einzelner CAN-Knoten besteht aus einem Mikrocontroller, einem CAN-Controller und einem CAN-Transceiver. Der Abschlusswiderstand unterdrückt Busreflexionen [15]. . . . .	11
2.2.	Signalpegel Low-Speed-CAN (links) und High-Speed-CAN (rechts) [16]. . .	12
2.3.	Aufbau des Standard CAN Data-Frames [16]. . . . .	12
3.1.	Systemarchitektur der Komponenten des EVObots im angepassten Schichtenmodell . . . . .	19
4.1.	Ablauf einer Diagnose im Kfz-Umfeld [23] . . . . .	24
4.2.	Use-Case-Diagramm einer Diagnosefunktion für den Fahrdemonstrator . . .	26
4.3.	Layout des CAN-Interfaces auf dem NVIDIA Jetson TX2 nach dem NVIDIA Product Design Guide [25] . . . . .	30
4.4.	Schaltplan des SN65HVD230 CAN-Transceivers . . . . .	30
4.5.	Schematischer Aufbau der Buskommunikation zwischen dem EVObot und dem Diagnoserechner . . . . .	33
4.6.	Programmgraph des initialen ROS-Paketes <i>autonomous</i> , mit dem Befehl <code>rqt_graph</code> automatisch generiert . . . . .	34
4.7.	Signalzuweisung innerhalb der Botschaft <code>ultraschall_data</code> mit dem Editierungstool CANdb++ . . . . .	43
4.8.	Nutzerinterface der Diagnosefunktion. Es werden dem Anwender alle relevanten Informationen in Echtzeit ausgegeben. . . . .	45
4.9.	Programmgraph des neuen ROS-Paketes <i>autonomous</i> inklusive dem Knoten <i>diagnostic_node</i> , mit dem Befehl <code>rqt_graph</code> automatisch generiert . . . . .	46
4.10.	Ausschnitt des Trace-Fensters in CANoe bei aktiver Diagnosekommunikation mit dem EVObot . . . . .	47
4.11.	Aufzeichnung des Differenzspannungssignal der CAN-Botschaft <code>lane_data_1</code> auf der physikalischen Busleitung . . . . .	50



# Literatur

- [1] Deutscher Verkehrssicherheitsrat. *Unfallstatistik aktuell: Langfristige Entwicklung*. 2018. URL: <https://www.dvr.de/unfallstatistik/de/jahre/> (geprüft am 16.09.2018).
- [2] Technische Universität Braunschweig. *Carolo-Cup*. Lehrstuhl für elektronische Fahrzeugsysteme. URL: <https://wiki.ifr.ing.tu-bs.de/carolocup/carolo-cup> (geprüft am 16.09.2018).
- [3] Fabian Fitzer. “Entwicklung eines Demonstrators für Fahrerassistenzsysteme und Implementierung einer Querverführung”. Lehrstuhl Fahrzeugtechnik und –antriebe. Masterarbeit. Cottbus: Brandenburgische Technische Universität Cottbus, 31.03.2018.
- [4] Christof Kary. “Ermittlung der Rekuperationsenergie und der Leistungsaufnahme von Nebenverbrauchern in einem Elektroauto”. Institut für Energieeffiziente Mobilität. Bachelorarbeit. Karlsruhe: Hochschule Karlsruhe, 12.09.2016.
- [5] Hermann Winner u. a., Hrsg. *Handbuch Fahrerassistenzsysteme. Grundlagen, Komponenten und Systeme für aktive Sicherheit und Komfort*. 3., überarbeitete und ergänzte Auflage. ATZ / MTZ-Fachbuch. Wiesbaden: Springer Vieweg, 2015.
- [6] SAE International, Hrsg. *J3016 - Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*. SAE. Version 201806. United States: SAE International, 15. Juni 2018. URL: [https://saemobilus.sae.org/content/J3016\\_201806/](https://saemobilus.sae.org/content/J3016_201806/) (geprüft am 19.09.2018).
- [7] Verband der Automobilindustrie e.V. *Automatisierung. Von Fahrerassistenzsystemen zum automatisierten Fahren*. 2015. URL: <https://www.vda.de/de/services/Publikationen/automatisierung.html> (geprüft am 19.09.2018).
- [8] ADAC e.V. *Autonomes Fahren: Die 5 Stufen*. 2018. URL: <https://www.adac.de/rund-ums-fahrzeug/autonomes-fahren/autonomes-fahren-5-stufen/> (geprüft am 19.09.2018).
- [9] Niels Klußmann. *Lexikon der Kommunikations- und Informationstechnik. Telekommunikation, Datenkommunikation, Multimedia, Internet*. 2., erw. und aktualisierte Aufl. Heidelberg: Hüthig, 2000.

## Literatur

- [10] Vector Informatik GmbH. *Einführung in die Seriellen Bussysteme im Kfz*. URL: [https://elearning.vector.com/vl\\_sbs\\_introduction\\_de.html](https://elearning.vector.com/vl_sbs_introduction_de.html) (geprüft am 08.06.2018).
- [11] Vector Informatik GmbH. *Einführung in FlexRay*. URL: [https://elearning.vector.com/vl\\_flexray\\_introduction\\_de.html](https://elearning.vector.com/vl_flexray_introduction_de.html) (geprüft am 08.06.2018).
- [12] Daniel Schüller. “Bussysteme im Automobil. Ausarbeitung zum Seminarvortrag”. Koblenz: Universität Koblenz-Landau, 2005-01-20. (Geprüft am 08.06.2018).
- [13] 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](http://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip) (geprüft am 08.06.2018).
- [14] Henning Wallentowitz und Konrad Reif, Hrsg. *Handbuch Kraftfahrzeugelektronik. Grundlagen - Komponenten - Systeme - Anwendungen*. 2., verb. und aktualisierte Aufl. ATZ/MTZ-Fachbuch. Wiesbaden: Vieweg + Teubner, 2011.
- [15] 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. URL: <http://dx.doi.org/10.1007/978-3-658-02419-2>.
- [16] Vector Informatik GmbH. *Einführung in CAN*. URL: [https://elearning.vector.com/vl\\_can\\_introduction\\_de.html](https://elearning.vector.com/vl_can_introduction_de.html) (geprüft am 08.06.2018).
- [17] 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>.
- [18] 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).
- [19] 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).
- [20] Josef Börcsök. *Elektronische Sicherheitssysteme. Hardwarekonzepte, Modelle und Berechnung*. 2., überarbeitete Auflage. Praxis. Heidelberg: Hüthig GmbH & Co. KG, 2007. URL: [http://deposit.d-nb.de/cgi-bin/dokserv?id=2995265&prov=M&dok\\_var=1&dok\\_ext=htm](http://deposit.d-nb.de/cgi-bin/dokserv?id=2995265&prov=M&dok_var=1&dok_ext=htm).



- [21] 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).
- [22] Konrad Reif. *Automobilelektronik. Eine Einführung für Ingenieure*. 5., überarb. Aufl. ATZ-MTZ-Fachbuch. Wiesbaden: Springer Vieweg, 2014.
- [23] 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).
- [24] Florian Schäffer. *OBD - Fahrzeugdiagnose in der Praxis*. ger. Elektronik. Haar bei München: Franzis, 2012. 240 S.
- [25] 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).
- [26] 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).
- [27] Vector Informatik GmbH. *Handbuch CANcaseXL*. 2015. URL: [https://vector.com/portal/medien/cmc/manuals/CANcaseXL\\_Manual\\_DE.pdf](https://vector.com/portal/medien/cmc/manuals/CANcaseXL_Manual_DE.pdf) (geprüft am 08.06.2018).
- [28] Vector Informatik GmbH. *CANoe. Produktinformation*. 2018. URL: [https://vector.com/portal/medien/cmc/info/CANoe\\_ProductInformation\\_DE.pdf](https://vector.com/portal/medien/cmc/info/CANoe_ProductInformation_DE.pdf) (geprüft am 08.06.2018).
- [29] 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).
- [30] 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).
- [31] 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).

## Literatur

- [32] 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).
- [33] Jason Blackman und Scott Monroe. *Overview of 3.3V CAN (Controller Area Network) Transceivers*. Hrsg. von Texas Instruments Incorporated. 2013. URL: <http://www.ti.com/lit/an/s11a337/s11a337.pdf> (geprüft am 15.09.2018).

# A. Anhang

## A.1. Auszug aus dem Datenblatt SN65HVD230



SN65HVD230, SN65HVD231, SN65HVD232

SLOS346Q – MARCH 2001 – REVISED APRIL 2018

### 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  $\pm 16$  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  $\mu$ A Typical
- SN65HVD231: Ultra Low Current Sleep Mode
  - 40 nA Typical
- Designed for Data Rates<sup>(1)</sup> up to 1 Mbps
- Thermal Shutdown Protection
- Open Circuit Fail-Safe Design
- Glitch Free Power Up and Power Down Protection for Hot Plugging Applications

<sup>(1)</sup> 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

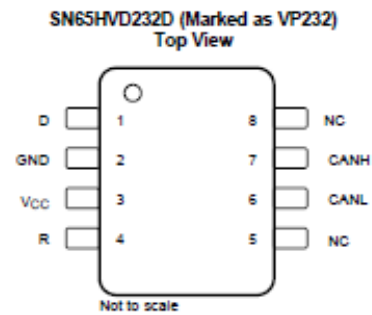
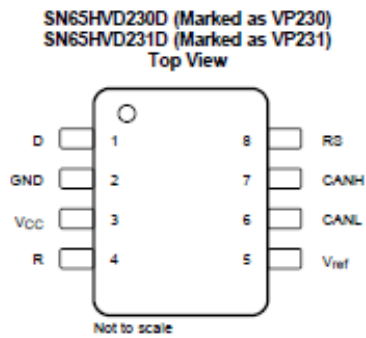
The SN65HVD230, SN65HVD231, and SN65HVD232 controller area network (CAN) transceivers are compatible to the specifications of the ISO 11898-2 High Speed CAN Physical Layer standard (transceiver). These devices are designed for data rates up to 1 megabit per second (Mbps), and include many protection features providing device and CAN network robustness. The SN65HVD23x transceivers are designed for use with the Texas Instruments 3.3 V  $\mu$ Ps, 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<sup>(1)</sup>

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

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

## 7 Pin Configuration and Functions



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 <sub>CC</sub>	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 <sub>ref</sub>	5	O	SN65HVD230 and SN65HVD231: V <sub>CC</sub> / 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 <sub>S</sub>	8	I	SN65HVD230 and SN65HVD231: Mode select pin: strong pull down to GND = high speed mode, strong pull up to V <sub>CC</sub> = 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)<sup>(1)(2)</sup>

	MIN	MAX	UNIT
Supply voltage, V <sub>CC</sub>	-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 <sub>I</sub> (D or R)	-0.5	V <sub>CC</sub> + 0.5	V
Receiver output current, I <sub>O</sub>	-11	11	mA
Continuous total power dissipation	See Thermal Information		
Storage temperature, T <sub>stg</sub>	-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](http://elinux.org) zusammengefasst. Umfangreiche benutzerfreundliche Beispiele zum Arbeiten mit dem Jetson-Board finden sich auf [jetsonhacks.com](http://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 noch 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 Grundlagen-erklärung* verwiesen. Generell eignet sich das *Wiki* der Homepage [ubuntuusers.de](http://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 lautet 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 Jetson-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 Korrekten 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 Organisation 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 Fahroutine 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 eingeleseene Kamerabild, die detektierten Fahrspuren und die Verkehrszeichenerkennung als Image Stream ausgegeben.



## 4 OpenCV

OpenCV ist eine freie Programmbibliothek mit diversen Algorithmen speziell für die Bildverarbeitung und maschinelles Sehen. Die Bibliothek kann in 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](http://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.