HOCHSCHULE MITTWEIDA
UNIVERSITY OF APPLIED SCIENCES

BACHELORARBEIT

Frau
Sina Vanessa Lipke

Entwicklung eines konfigurierbaren grafischen Benutzerinterfaces für die CAN-Analysesoftware BUSMASTER

2019

Fakultät **Angewandte Computer- und Biowissenschaften**

BACHELORARBEIT

Entwicklung eines konfigurierbaren grafischen Benutzerinterfaces für die CAN-Analysesoftware BUSMASTER

Autorin:
Sina Vanessa Lipke

Studiengang:
Allgemeine und Digitale Forensik

Seminargruppe:
FO16w5-B

Erstprüfer:
Prof. Dr. rer. nat. Dirk Labudde

Zweitprüfer:
Dipl.-Ing. (FH) Heiko Polster

Mittweida, September 2019

Faculty of **Applied Computer Sciences and Biosciences**

BACHELOR THESIS

Development of a configurable graphical user interface for the CAN analysis software BUSMASTER

Author:
Sina Vanessa Lipke

Course of Study:
General and Digital Forensics

Seminar Group:
FO16w5-B

First Examiner:
Prof. Dr. rer. nat. Dirk Labudde

Second Examiner:
Dipl.-Ing. (FH) Heiko Polster

Mittweida, September 2019

Bibliografische Angaben

Lipke, Sina Vanessa: Entwicklung eines konfigurierbaren grafischen Benutzerinterfaces für die CAN-Analysesoftware BUSMASTER, 69 Seiten, 28 Abbildungen, Hochschule Mittweida, University of Applied Sciences, Fakultät Angewandte Computer- und Biowissenschaften

Bachelorarbeit, 2019

Referat

In der vorliegenden Bachelorarbeit wird ein konfigurierbares grafisches Benutzerinterface für die CAN-Analysesoftware BUSMASTER der Firma ETAS GmbH konzeptioniert, entwickelt und an einem CAN-Demonstrator getestet. Gemäß der in dieser Arbeit vermittelten Grundlagen zu aktuellen Bussystemen der Automobilindustrie und ihrer Datenübertragung werden entsprechende CAN-Nachrichten implementiert und auf den CAN-Bus gesandt. Eine Auswertung der über den Bus laufenden Botschaften ermöglicht eine Visualisierung dieser innerhalb der GUI. Die Funktionsfähigkeit des entwickelten Userinterfaces kann durch die erfolgreiche Steuerung des CAN-Demonstrators belegt werden. Dieser wird zudem im Rahmen der Softwareentwicklung und Inbetriebnahme weiterentwickelt.

Abstract

In this bachelor thesis, a configurable graphical user interface for the CAN analysis software BUSMASTER by ETAS GmbH will be conceptualized, developed and tested with a CAN demonstrator. In accordance with the basics of current bus systems of the automotive industry and their data transmission, the corresponding CAN messages are implemented and sent to the CAN bus. An evaluation of the messages sent on the bus enables a visualization of these within the GUI. The functionality can be proven by the successful control of the CAN demonstrator. The CAN demonstrator will be further developed during software engineering and commissioning phase.

I. Inhaltsverzeichnis

Inhaltsverzeichnis I

Abbildungsverzeichnis II

Tabellenverzeichnis III

Abkürzungsverzeichnis IV

1 Einleitung 1
1.1 Motivation 1
1.2 Zielstellung 1

2 Grundlagen 3
2.1 Netzwerktopologien 3
2.1.1 Linientopologie 3
2.1.2 Ringtopologie 4
2.1.3 Sterntopologie 5
2.2 Datenübertragung 6
2.3 Bussysteme in Automobilen 7
2.3.1 Controller Area Network 7
2.3.2 Weitere Bussysteme 12
2.4 BUSMASTER 15
2.5 CAN-Demonstrator 16

3 Methoden 19
3.1 Präzisierung der Aufgabenstellung 19
3.2 Grundkonzeptionierung 21
3.3 Auswahl der Entwicklungsumgebung 24

4 Softwareentwicklung und Implementierung 29
4.1 Kommunikation mit dem BUSMASTER 29
4.1.1 Senden von CAN-Nachrichten 30
4.1.2 Empfang von CAN-Nachrichten 32
4.2 Visualisierung mit Visual Studio 33
4.2.1 Integrierung der CAN-Nachrichten in Visual Studio 33
4.2.2 Auswertung der CAN-Nachrichten in Visual Studio 34
4.2.3 Grafische Bedienoberfläche 38

5 Systemtest und Inbetriebnahme 45
5.1 Voraussetzungen 45
5.2 Inbetriebnahme der Kommunikation mit dem BUSMASTER 45
5.2.1 Versuchsaufbau 46
5.2.2 Versuchsdurchführung 46
5.2.3 Fazit 49

5.3 Inbetriebnahme der Kommunikation zw. BUSMASTER und CAN-Demonstrator . 49
5.3.1 Versuchsaufbau 49
5.3.2 Versuchsdurchführung 50
5.3.3 Fazit 51
5.4 Optimierungen 52

6 Zusammenfassung **59**

7 Ausblick **61**

A Finale Version des grafischen Userinterface **63**

B Messwerte ADC **65**

Literaturverzeichnis **67**

II. Abbildungsverzeichnis

2.1 Linientopologie 3
2.2 Ringtopologie 4
2.3 Sterntopologie 5
2.4 Serielle und parallele Datenübertragung 6
2.5 CAN-Datentelegramm Standard Frame 10
2.6 Benutzeroberfläche des BUSMASTERs 15
2.7 CAN-Demonstrator 16
2.8 Schematischer Aufbau der CAN-Verbindung 17
3.1 Schematischer Aufbau der CAN-Analyse 20
3.2 Blockbild der GUI 21
3.3 Skizze über den Ablauf des Programms für die An/Aus-Funktion 22
3.4 Skizze über den Ablauf des Programms für die Geschwindigkeits-Funktion 23
4.1 Relevante Teile der CAN-Nachricht 30
4.2 Statusnachricht des CAN-Demonstrators 34
4.3 Kontrollausgabe mit CAN-Nachrichten 37
4.4 Grafische Bedienoberfläche des Interfaces 38
4.5 Hinweisfenster in der GUI 39
4.6 Skizze über den Ablauf des Programms des ESP für die Hupe 42
5.1 Blockschaltbild Versuchsaufbau der Kommunikation mit dem BUSMASTER 46
5.2 BUSMASTER-Software 47
5.3 Blockschaltbild Versuchsaufbau der Kommunikation mit dem CAN-Demonstrator . . 49
5.4 Skizze über den Ablauf des Programms des ESP für die Akkustandsüberwachung . . 53
5.5 Spannungsteiler 54
5.6 Skizze des Schaltkreises des Spannungsteilers 54
5.7 Akkustandsanzeige in der GUI 56
5.8 Skizze des Schaltkreises der Spannungsmessung am CAN-Demonstrator 56
5.9 Skizze des Schaltkreises der Spannungsmessung am ADC 57
5.10 Verhältnis ADC-Wert zu Referenzwert 58

III. Tabellenverzeichnis

2.1 Aufbau eines Standard CAN-Frames 11
3.1 Bewertung der Eclipse IDE 25
3.2 Bewertung der NetBeans IDE 26
3.3 Bewertung der Visual Studio IDE 27
3.4 Bewertung der SharpDevelop IDE 27
3.5 Vergleich der Softwarelösungen 28
4.1 Implementierte CAN-Nachrichten 33
4.2 Aufbau der Datenbytes der Statusnachricht des CAN-Demonstrators 35
5.1 CAN-Nachrichten des Versuchsaufbaus 48
5.2 In Betrieb genommene Komponenten 50

IV. Abkürzungsverzeichnis

ACK .... Acknowledge
ADC .... Analog-to-Digital Converter
bps .... Bit per second
CAN .... Controller Area Network
CRC .... Cyclic Redundancy Check
CTR .... Control Field
DL .... Data Length (dt. Datenlänge)
ECU .... Electronic Control Unit
EoF .... End-of-Frame
ESP .... Espressif
EVB .... Evaluationboard
GND .... Ground
GUI .... Graphical User Interface
IDE .... Integrated Development Environment
IFS .... Interframe Space
ISO .... Internationale Organisation für Standardisierung
ISR .... Interrupt Service Routine
LIN .... Local Interconnect Network
MOST .... Media Oriented Systems Transport
RTR .... Remote Transmission Request
SoF .... Start-of-Frame
VS .... Visual Studio
WLAN .... Wireless Local Area Network

1 Einleitung

Die Vision vom autonomen Fahren offeriert wesentliches Innovationspotenzial. Entsprechend hat sich die Steuerung im Fahrzeug in den letzten Jahren zukunftsweisend verändert und unzählige Verkabelungen durch ein komplexes System untereinander vernetzter Steuergeräte, auch Electronic Control Units (ECUs) genannt, ersetzt. Der CAN-Bus (Controller Area Network) als serielles Bussystem ist in nahezu allen modernen Automobilen verbaut und dient der Kommunikation der ECUs innerhalb des Kfz. Das Gateway bildet hierbei als Verbindungsstück die zentrale Einheit der einzelnen Bussysteme und stellt einen systemübergreifenden Austausch der Daten sicher. Durch diesen enormen Fortschritt der Digitalisierung wird jedoch gleichzeitig eine Zunahme der IT-Kriminalität verzeichnet.
Mit eben dieser Ausbreitung der Autonomisierung sowie der Vernetzung vergrößert sich die Möglichkeit für Angriffe stetig. Durch das Senden manipulierter Nachrichten auf das Controller Area Network kann das System und damit das gesamte Automobil gezielt attackiert werden. Denkbar wären Kontrollverluste über grundlegende Motorfunktionen wie Gas und Bremse, die zum Ausfall der Sicherheit im Auto führen und demzufolge Menschenleben gefährden können.

1.1 Motivation

Um ebendiesen Übergriffen entgegenzuwirken, ist eine Analyse des CANs von signifikanter Bedeutung. Mit Hilfe spezieller Tools kann eine Überwachung der Nachrichten, die über den Bus gesendet werden, erfolgen. Eine solche Softwarelösung stellt dabei der BUSMASTER der ETAS GmbH dar. Dieser bietet die Möglichkeit der Signalüberwachung und des Loggens der über den CAN-Bus gesandten Daten, verfügt jedoch über keine grafische Bedienoberfläche mit Steuergeräten, von der aus der BUSMASTER gesteuert und die Nachrichten visualisiert werden können. Eine solche ist jedoch zur Bedienung und insbesondere im Bereich der Lehre zur anschaulichen Darstellung wichtig.

1.2 Zielstellung

Ziel dieser Arbeit stellt die Entwicklung eines grafischen Benutzerinterface für die CAN-Analysesoftware BUSMASTER der ETAS GmbH dar. Dieses soll die Möglichkeit der beliebigen Konfiguration bieten und zur Steuerung, Visualisierung sowie Simulation der über den CAN-Bus gesandten Nachrichten dienen. Durch die genannte Software werden die Nachrichten generiert, auf den CAN-Bus gesandt und mitgeschnitten. Zur Umsetzung des Projektes soll eine entsprechende Entwicklungsumgebung dienen. Ein abschließender Systemtest soll nach Inbetriebnahme durch den CAN-Demonstrator erfolgen.

2 Grundlagen

Dieses Kapitel setzt sich mit den theoretischen Grundlagen aus dem Bereich der Automobilindustrie auseinander, welche die Basis dieser Arbeit darstellen. Des Weiteren werden die vorgegebenen Hardware- und Softwarekomponenten kurz vorgestellt.

2.1 Netzwerktopologien

Die Topologie eines Netzwerkes beschreibt die Verbindungsstruktur zwischen den einzelnen Steuergeräten eines Netzwerkes, wobei der Fokus auf einem möglichst effizienten Austausch der Nachrichten liegt. Für ein solches Datennetz gibt es dabei verschiedene Formen des Aufbaus, sogenannte Topologien. Innerhalb jeder Topologie existiert mindestens ein Master-Steuergerät (rot) und weitere Teilnehmer, sogenannte Slaves (grau).

[ZIS14, Kap. 2]

2.1.1 Linientopologie

Das Merkmal der Linientopologie ist die elektrisch passive Ankopplung aller Teilnehmer an eine gemeinsame Leitung (Vgl. Abbildung 2.1). Die von einer ECU gesandten Nachrichten erhalten dabei alle Teilnehmer. Die Linientopologie ist die mit Abstand am häufigsten verwendete Topologie.

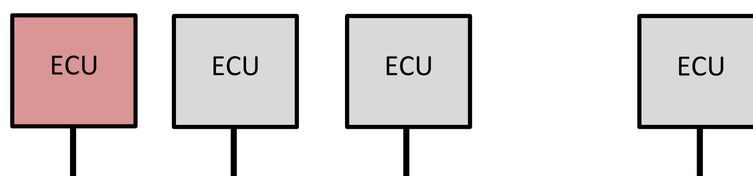


Abbildung 2.1: Linientopologie. Quelle: modifiziert nach [ZIS14, S. 17]

Vorteile

Der Verkabelungsaufwand wird hierbei gering gehalten und neue Teilnehmer können ohne eine Unterbrechung des Betriebs hinzugefügt werden. Bei Ausfall oder gezieltem Abschalten eines Teilnehmers hat dies keine Auswirkungen auf die restlichen Steuergeräte im Netzwerk.

Nachteile

Die Buslänge sowie die Anzahl der Teilnehmer sind in dieser Topologie begrenzt. Ab einer gewissen Länge bzw. Anzahl ist daher eine Signalverstärkung vonnöten. Um Kollisionen zu vermeiden ist eine Buszugriffsregelung erforderlich. Zudem müssen die Enden der Leitung durch einen Wellenwiderstand terminiert werden.

2.1.2 Ringtopologie

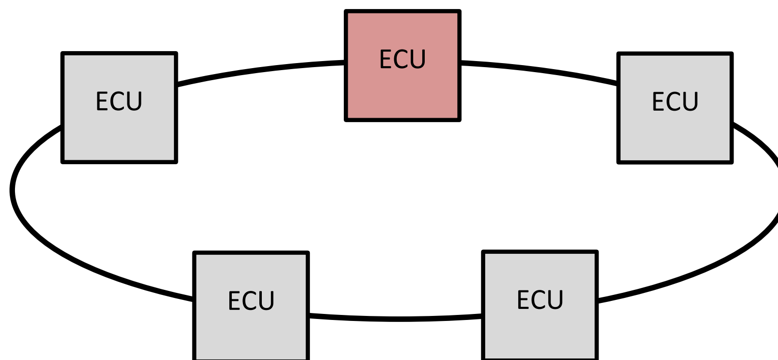


Abbildung 2.2: Ringtopologie. Quelle: modifiziert nach [ZIS14, S. 18]

Die Ringtopologie (siehe Abbildung 2.2) zeichnet sich durch eine geschlossene Kette von Punkt-zu-Punkt-Verbindungen, einem sogenannten Teilstreckennetz, aus. Eine ausgesandte Nachricht durchläuft dabei so lange den Ring, bis sie den eigentlichen Empfänger erreicht hat.

Vorteile

Von Vorteil bei dieser Topologie ist, dass jeder Teilnehmer eine Signalregeneration bewirkt und Netzwerke daher sehr groß sein dürfen. Jeder Teilnehmer kann dabei über seine geografische Position identifiziert werden. Auf Grund der Punkt-zu-Punkt-Übertragung ist diese Topologie zudem für den Einsatz von Lichtwellenleitern geeignet.

Nachteile

Problematisch ist, dass bei einem Ausfall eines einzelnen Teilnehmers das gesamte System ausfällt. Es bestehen zwar Möglichkeiten zur Absicherung durch Überbrückung eines ausgefallenen Teilnehmers oder über einen redundanten Ring, diese erhöhen jedoch den Verkabelungsaufwand. Soll ein neuer Teilnehmer eingebunden werden, muss hierzu der Bus unterbrochen werden.

2.1.3 Sterntopologie

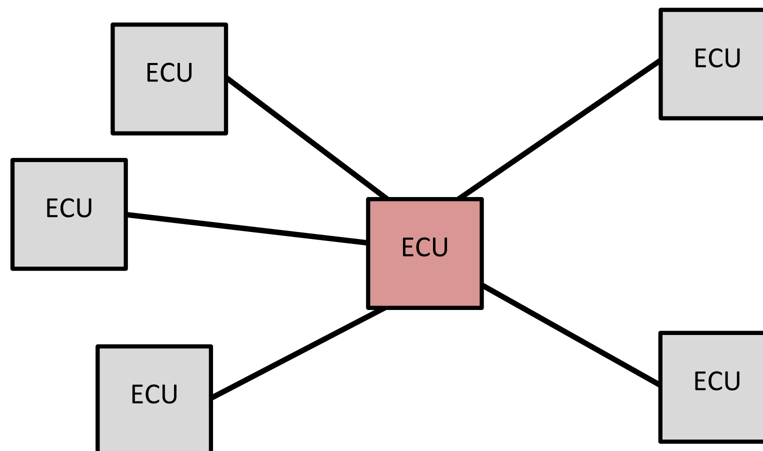


Abbildung 2.3: Sterntopologie. Quelle: modifiziert nach [ZIS14, S. 18]

Abbildung 2.3 zeigt das Netzwerk einer Sterntopologie, welche alle Steuergeräte mittels einer Punkt-zu-Punkt-Verbindung mit einem zentralen Teilnehmer verbindet.

Vorteile

Im Netzwerk selbst hat jeder Teilnehmer seine eigene Verbindung, weswegen sich eine Einbindung weiterer ECUs einfach realisieren lässt. Die Topologie basiert auf einem simplen Protokoll, welches keine Zugriffsrechte erfordert.

Nachteile

Grundsätzlich ergibt sich eine große Gesamtlänge aller Verbindungen, da ein zentraler Teilnehmer so viele Schnittstellen benötigt wie sich Steuergeräte im Netzwerk befinden. Eine Kommunikation zwischen den Teilnehmern ist dabei nur über das zentrale Steuergerät realisierbar. Kommt es zum Ausfall des zentralen Teilnehmers, ist keine Kommunikation im System mehr möglich.

[ETS00, Kap. 1.6]

2.2 Datenübertragung

Zur Datenübertragung in einem Netzwerk gibt es drei mögliche Verbindungsformen: Die Punkt-zu-Punkt-Verbindung bei der ein Steuergerät mit einem anderen kommuniziert, Punkt-zu-Gruppe (Multicast), bei der ein ECU Daten an mehrere Teilnehmer im Netzwerk sendet und Punkt-zu-allen (Broadcast), wobei das Steuergerät an alle Teilnehmer sendet. Des Weiteren gibt es drei Arten der Datenübertragung. Ein uni-direktionaler Nachrichtenaustausch in eine Richtung über eine Datenverbindung (Simplex), bidirektional im Wechselverkehr mit gemeinsamer Datenverbindung (Halb-Duplex) oder bidirektional mit getrenntem Datenweg für jede Richtung (Voll-Duplex).
Die Verbindungsausprägung kann dabei seriell, parallel oder drahtlos vorliegen. Werden die Zeichen nacheinander übertragen, ist von einer seriellen Übertragung die Rede. Bei einer parallelen Übertragung werden die Zeichen gleichzeitig auf verschiedenen Leitungen übermittelt. Die drahtlose Übertragungsart ist grundsätzlich seriell, da es sich um eine bit-serielle Übertragung des Datenstroms handelt. Eine schematische Abbildung der seriellen und parallelen Datenübertragung ist Abbildung 2.4 zu entnehmen.

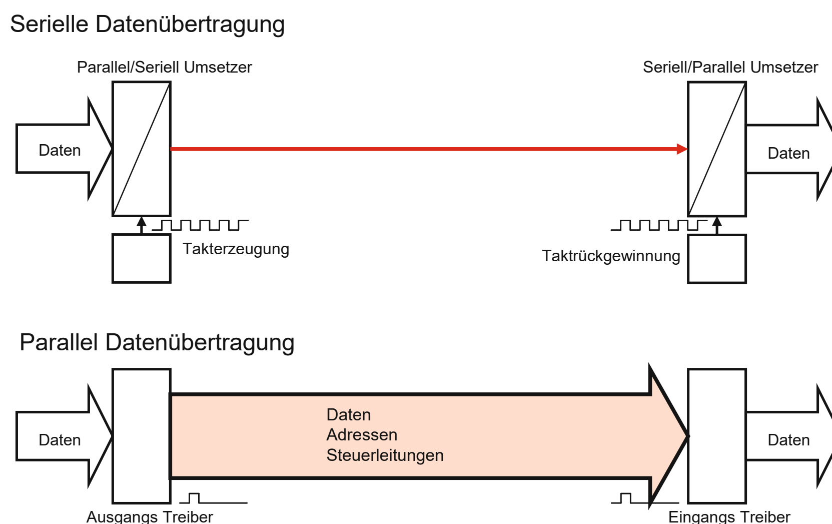


Abbildung 2.4: Serielle und parallele Datenübertragung. Quelle: [HBG14, S. 685]

Bei einer bit-seriellen Übertragung müssen Sender und Empfänger synchronisiert werden, der Takt muss dabei aus den Synchronisierungsimpulsen vor den eigentlichen Daten aus dem Datenstrom abgeleitet werden. Nach erfolgreicher Synchronisation werden die Daten als Bitstrom bis zur Endsequenz übertragen. Zur Sicherung vor fehlerhafter Übertragung können Parity-Bits oder redundante Checksummen übertragen werden.

Die parallele Datenübertragung findet insbesondere beim Austausch von sehr großen Datenmengen und hohen Geschwindigkeiten Anwendung. Der wesentliche Unterschied gegenüber der bit-seriellen Übertragung ist die Verfügung über den kompletten Datensatz, wodurch eine erheblich schnellere Abarbeitung ermöglicht werden kann. Zur Datenübertragung sind dabei die Datengruppe, Adressgruppe sowie Steuergruppe notwendig. Zum sicheren und synchronen Einlesen der Daten ist auch hier ein Taktsignal für die Synchronisation vonnöten. Da eine parallele Übertragung bei steigender Bit-Breite auf Grund der entsprechend hohen Kosten nicht mehr sinnvoll umzusetzen ist, werden Teile der Informationen gemultiplext. Dabei werden Daten und Adressen auf denselben Leitungen übertragen und an der Empfangsseite wieder getrennt.

[HBG14, Kap. 15.1]

2.3 Bussysteme in Automobilen

Um die Vernetzung der Steuergeräte innerhalb eines Automobils zu realisieren, wurden Bussysteme zur Kommunikation entwickelt, welche die bisherige Punkt-zu-Punkt-Verbindung ersetzen sollen. Unter einem Bus wird dabei eine Leitung zwischen mehreren Steuerungskomponenten zum Datenaustausch verstanden, wobei jedes ECU einen Knotenpunkt darstellt. Aufgrund der enormen Datenmenge entstand ein Netzwerk aus mehreren Systemen mit unterschiedlichen Übertragungseigenschaften. Die bekanntesten Bussysteme stellen dabei folgende dar:

- Controller Area Network (CAN)
- Local Interconnect Network (LIN)
- FlexRay
- Media Oriented Systems Transport (MOST)
- Automotive Ethernet

2.3.1 Controller Area Network

Das in den 1980er Jahren durch Bosch entwickelte Controller Area Network ist eines der am weitesten verbreiteten Bussysteme im Automobilbereich und nach der ISO 11898 international standardisiert. Mit einer Bitübertragungsrate von bis zu 1 MBit/s ermöglicht es die Datenübertragung zwischen mehreren Steuergeräten, wobei diese nach einer Linienstruktur angeordnet und gleichermaßen für das Senden und Empfangen von Nachrichten berechtigt sind. Die Anzahl der Steuergeräte ist dabei nicht durch das Protokoll begrenzt, sondern in Abhängigkeit der Leistungsfähigkeit seiner Treiberbausteine und kann durch den Einsatz von Repeatern auf 64 bis 128 Teilnehmer erweitert werden. Der signifikante Unterschied zu anderen Bussystemen liegt in seiner hohen Fehlertoleranz.

Datenübertragung im CAN

Als Übertragungsmedium kommen für das CAN Datenleitungen aus Kupfer mit verdrillten bzw. unverdrillten Adern (Twisted Pair) und einer Masseleitung zum Einsatz. An dieser Datenleitung sind die Busteilnehmer, die sog. CAN-Knoten angeknüpft. Ein CAN-Knoten ist dabei aus folgenden Bestandteilen zusammengesetzt:

- Eine Software, welche Daten zur Verfügung stellt und diese verarbeitet
- Ein Controller, der die Daten der Software in CAN-Nachrichten konvertiert und auf den Bus sendet und umgekehrt die Daten der Botschaften extrahiert und an die Software leitet
- Ein Transceiver, welcher die Signale der Datenleitung aufnimmt und Informationen des CAN-Controllers auf den Bus sendet

Die Datenübertragung im CAN findet seriell über Halb-Duplex statt und ist broadcast-orientiert. Es können dabei Bitraten bis 125 kBit/s im Low-Speed-Bereich sowie bis zu 1 Mbit/s im High-Speed-CAN erreicht werden. Zur Gewährleistung der Sicherheit und Verfügbarkeit wird auf eine Multimaster-Hierarchie gesetzt. Die Buszugriffssteuerung erfolgt dabei durch die bitweise Arbitrierung. Jede Nachricht im Netzwerk ist durch einen Identifier gekennzeichnet, anhand von welchem die einzelnen Teilnehmer im Netzwerk die Relevanz der Botschaft für sie überprüfen können. Durch diesen Identifier kann zudem die Priorität der Nachrichten bestimmt und ein bevorzugter Buszugriff erteilt werden. Dabei gilt: je niedriger die ID, desto höher die Wichtigkeit. Diese Eigenschaft garantiert in Störfällen die schnellstmögliche Übertragung wichtiger Informationen. Zudem können Kollisionen, welche durch einen Mehrfachzugriff unterschiedlicher Knoten entstehen können, durch die bitweise Arbitrierung der Informationen aufgelöst werden. Dafür vergleicht jeder Teilnehmer den Wert des Kommunikationsmediums mit dem seinen. Ist sein Wert größer, beendet er seinen Sendevorgang und bereitet sich auf den Empfang einer Botschaft vor. Hierfür wird eine dominante und rezessive Bitcodierung angewandt, wobei die logische 0 dominant, die 1 rezessiv ist. Ein globales Quittierungsfeld gibt dem Sender Auskunft über den Erhalt der Botschaft durch ein oder kein ECU. Es kann hierbei keine Information über die Anzahl der Steuergeräte, welche die Botschaft richtig empfangen haben, entnommen werden, sondern nur, ob überhaupt ein Teilnehmer diese Nachricht erhalten hat. Der Sender kann somit selbst prüfen, ob er noch eine aktive Verbindung zum Bus hat.
Um alle Steuergeräte im Netzwerke innerhalb einer Bitzeit synchronisieren zu können, ist die Buslänge begrenzt. Die Berechnung dieser erfolgt anhand folgender Formel:

$$\text{Buslänge} \leq 40...50\text{m} \cdot \frac{1\text{MBit/s}}{\text{Bitrate}}$$

Um die Synchronisation der ECUs aufrecht zu erhalten, werden sogenannte Synchronisationsflanken mittels Bitstuffing erzeugt. Hierfür überträgt der Sender nach fünf Bits gleicher Polarität (dominant oder rezessiv) einen komplementären Wert. Auf Empfängerseite wird dieses Stuff-Bit automatisch entfernt, sodass die ursprünglichen Daten wieder vorliegen.

[REI11, S.92], [ZIS14, Kap. 3.1.2], [LAB99, Kap.2.2, 4.1], [ETS00, Kap. 1.8.1, 2.2]

High-Speed- und Low-Speed-CAN

Die ISO-Norm unterscheidet im Controller Area Network zwischen High-Speed- und Low-Speed-CAN. Datenübertragungsraten im Bereich von 125 kBit/s bis 1 Mbit/s werden dabei als High-Speed-CAN in der ISO 11898-2 normiert und finden auf Grund der Geschwindigkeit für die Echtzeitanforderungen des Antriebsstranges Verwendung.
Bei Datenraten zwischen 5 und 125 kBit/s wird Low-Speed-CAN eingesetzt. Das, in der ISO 11989-3 definierte, Netzwerk findet in Komfort- und Karosseriebereichen, wie beispielsweise der Klimaanlage oder Fensterhebern, Anwendung.

[REI11, S.92]

CAN-Botschaft

Innerhalb des Controller Area Networks wird zwischen folgenden vier Nachrichtentypen unterschieden:

- **Data Frame**
  Datentelegramme werden zur Übertragung von Daten im Netzwerk genutzt.

- **Remote Frame**
  Datenanforderungstelegramme dienen der Anforderung einer bestimmten Nachricht von einem Teilnehmer im Netzwerk und sind durch eine bestimmte Codierung im Control Field (CTR) charakterisiert. In der Regel enthalten Remote Frames keine Daten.

- **Error Frame**
  Fehlertelegramme signalisieren einen, von einem Steuergerät erkannten, Fehler.

- **Overloadframe**
  Ein Überlasttelegramm führt eine Verzögerung zwischen zwei Data oder Remote Frames herbei.

Abbildung 2.5 veranschaulicht den logischen Aufbau eines Standard Datentelegramms im Controller Area Network mit den jeweiligen Feldern und zugehöriger Bitanzahl sowie dem zugehörigen Signalpegel (0 oder 1) oberhalb.

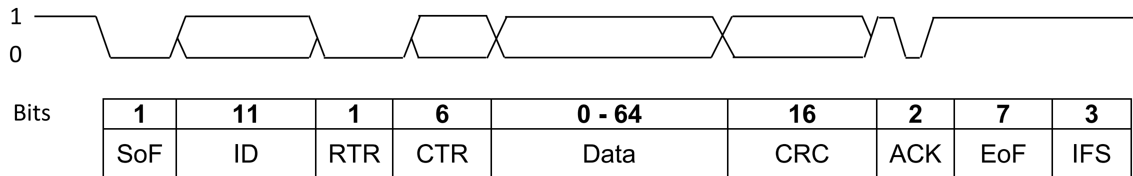


Abbildung 2.5: CAN-Datentelegramm Standard Frame. Quelle: [LI19a]

Der klassische Aufbau einer CAN-Nachricht beginnt immer mit einem Startbit, gefolgt von 11 bzw. 29 Identifier-Bits, je nachdem, ob es sich um ein Standard oder Extended Frame handelt. Die ID zusammen mit dem anschließend folgenden Remote Transmission Request (RTR) ergeben das Arbitrierungsfeld, wobei der RTR angibt, ob es sich um ein gesandtes oder angefordertes Nachrichtentelegramm handelt. Es reihen sich sechs Steuerbits an, welche die Angabe der Datenlänge der Nachricht beinhalten, die Datenbytes selbst, gefolgt von einem 16 Bit langen Cyclic Redundancy Check (CRC). Diese zyklische Redundanzprüfung dient zur Bestimmung eines Prüfwertes für die Daten, um Übertragungsfehler zu erkennen. Die maximale Datenlänge einer CAN-Nachricht ist auf acht Byte (64 Bit) begrenzt. Es folgen zwei Quittierungsbits bevor die Nachricht mit dem End-of-Frame (7 Bits) abgeschlossen wird. Die angehängten drei Interframe-Space-Bits (IFS) kennzeichnen den minimalen Abstand bis zum nächsten möglichen Botschaftsbeginn.
Die maximale Länge einer Botschaft liegt bei 117 Bits für das Standard-Frame und 136 Bits für das Extended-Frame. Aus der Anzahl der Identifier-Bits ergibt sich jeweils eine Anzahl von 2^{11} (2048) und 2^{29} (512 Millionen) verschiedenen Botschaften pro System. Eine Übersicht des Aufbaus einer CAN-Nachricht im Standard-Format ist Tabelle 2.1 zu entnehmen.

Die Begrenzung der Daten auf acht Byte ist erforderlich, um möglichst kurze Latenzzeiten für den Buszugang hochpriorer Nachrichten gewährleisten zu können. Eine kurze Nachrichtenlänge ist zudem in stark gestörten Bereichen von Vorteil, da mit zunehmender Blocklänge die Wahrscheinlichkeit eines Fehlers steigt und somit eine Übertragung der Daten unter schwierigen Umständen ermöglicht werden kann.

Tabelle 2.1: Aufbau eines Standard CAN-Frames

Feld	**Länge in Bit**	**Bedeutung**
Start-of-Frame (SoF)	1	Kennzeichnet den Beginn eines Telegramms
Identifier (ID)	11	Kennzeichnung der Nachricht, Prioritätsbestimmung
Remote Transmission Request (RTR)	1	Gibt an, ob es sich um ein Datentelegramm (0) oder ein Datenanforderungstelegramm (1) handelt
Control (CTR) Field	6	2 reservierte Bits + 4 Bits für die Angabe der Datenlänge des nachfolgenden Datenfeldes in Byte
Data Field	0 bis 64	Eigentliche Nutzdaten, entfällt bei Remote Frames
Cyclic Redundancy Check (CRC) Field	16	15 Bit Prüfsequenz + 1 Begrenzungsbit, enthält Fehlercode
Acknowledge (ACK) Field	2	1 Bit Bestätigungsfeld für den Erhalt der Nachricht, 1 Begrenzungsbit
End-of-Frame (EoF)	7	Kennzeichnet das Ende eines Telegramms
Interframe Space (IFS)	3	Zwischenraum zur Trennung der Telegramme

Fehlererkennung und Korrektur

Um die Integrität des Netzwerkes zu gewährleisten, bietet das CAN-Protokoll verschiedene Möglichkeiten, um Fehler bei der Übertragung zu erkennen, entsprechend zu behandeln und einzugrenzen. Durch das Errormanagement können vier verschiedene Fehlertypen erkannt werden:

- Bitfehler
- Stuffingfehler
- Formfehler
- Acknowledgement-Fehler

Nach der Erkennung eines Fehlers wird jeder Busteilnehmer durch ein Error Frame informiert. Diese CAN-Fehlerbotschaft besteht aus einer einzigartigen Form von sechs aufeinanderfolgenden dominanten Nullen, sieben Rahmenende Bits und drei Interframe-Space-Bits. Das Error Frame überschreibt alle anderen Botschaften und wird von allen Teilnehmern erkannt. Um einen Ausfall des gesamten Busses bei anhaltender Störung zu vermeiden, werden im Ausnahmefall Maßnahmen ergriffen, um das Einwirken des störenden CAN-Knotens einzuschränken bzw. diesen ganz zu isolieren.
Ein Teilnehmer kann generell drei Zustände annehmen. Den Normalzustand (Error Active) in dem ein uneingeschränktes Senden und Empfangen von Botschaften gewährleistet ist und in dem bei Eintritt eines Fehlers ein Error Flag gesetzt wird. Wurden mehrere Fehler auf dem Bus identifiziert, tritt der Error Passive Status ein. Es kann zwar weiter kommuniziert werden, im Fehlerfall wird jedoch nur noch ein Error Flag ausgesandt, sodass ein Teilnehmer in diesem Zustand den übrigen Busverkehr nicht weiter behindern kann. Bei der vollständigen Abkopplung vom Bus (Bus Off) ist das Senden und Empfangen von CAN-Nachrichten nicht mehr möglich. Der Knoten kann sich selbst als fehlerhaft erkennen und aus dem Netzwerk zurückziehen, um den Zusammenbruch des Systems zu vermeiden.

[LAB99, Kap. 2.2, 4.1], [ETS00, Kap. 1.8.1, 2.2]

2.3.2 Weitere Bussysteme

Local Interconnect Network

Das Local Interconnect Network (LIN) ist ein in den 1990er Jahren vom LIN-Konsortium, einem Zusammenschluss verschiedener Kfz-Hersteller mit Motorola (heute Freescale), entwickeltes Bussystem, welches als kostengünstigere Alternative zum Low-Speed-CAN dienen soll. Mit einer Bitübertragungsrate von 1 bis 20 kBit/s findet es im Bereich der einfachen Elektronik von Tür-, Spiegel- und Sitzeinstellung weit verbreitete Anwendung und ist in der ISO 17987 standardisiert. Der Master stellt dabei das Gateway zu anderen Bussystemen dar und steuert durch das Senden von Botschaftsheadern mit LIN-Identifier den gesamten Kommunikationsablauf. Diese Botschaften werden an alle Slave-Steuergeräte ausgesandt. Es kann jedoch nur ein Slave mit einer Daten-Botschaft antworten, da jedem Identifier genau ein Slave zugeordnet ist, was wiederum ausschließt, dass mehrere Steuergeräte versuchen Botschaften auf den Bus zu senden. Eine Fehlererkennung sowie entsprechende -behandlung ist für LIN nur ansatzweise definiert. Zudem bietet dieses Bussystem keinerlei Authentifizierungs- und Verschlüsselungsmöglichkeiten.

[ZIS14, Kap. 3.2]

FlexRay

FlexRay ist ein im Jahr 2000 von mehreren deutschen Fahrzeugherstellern und Zulieferern entworfenes Bussystem, welches durch höhere Bit- und Datenraten das CAN in zeitkritischen Bereichen ersetzen soll und in der ISO 17458 standardisiert ist. Das bitstrom-orientierte Übertragungsprotokoll realisiert die Kommunikation zwischen mehreren ECUs zum Austausch von Mess- und Reglersignalen sowie die Leitung elektronischer Steuersignale für beispielsweise Bremse oder Steuerung, insbesondere bei X-by-Wire-Anwendungen, im Echtzeitbetrieb und mit hoher Fehlersicherheit. FlexRay kann sowohl in einer Linien- als auch einer Sterntopologie vorliegen und erlaubt ein- und zweikanalige Systeme. Bei einer maximalen Rate von 10 MBit/s, können die Steuergeräte beim Linienbus oder in der Sternstruktur mit passivem Sternpunkt bis zu 24 Meter auseinanderliegen. Um Kollisionen auszuschließen, wird innerhalb eines Sendefensters das Senderecht genau einem Steuergerät zugeteilt. Im Gegensatz zum CAN gibt es im Fehlerfall keine automatische Botschaftswiederholung, welche den deterministischen Betrieb auf dem Bus in jedem Fall gewährleisten würde. Letztendlich ergänzt das FlexRay das Controller Area Network vielmehr als es zu ersetzen.

[ZIS14, Kap. 3.3]

Media Oriented Systems Transport

Im Gegensatz zu den bisherigen Bussystemen, die für Steuer- und Regelaufgaben entwickelt wurden, ist der Media Oriented Systems Transport (MOST) ausschließlich für die Multimedia-Anwendungen im Automobil bestimmt. Der, Ende der Neunziger, von der Firma Oasis (heute Microchip) entwickelte Infotainmentbus realisiert dabei die Vernetzung der Bordelektronik in den Modellen der oberen und mittleren Fahrzeugklassen. Höhere Übertragungsbandbreiten von bis zu 150 MBit/s bilden den wesentlichen Unterschied zu anderen Bussystemen im Kfz-Bereich. Grundsätzlich sind unter MOST unterschiedliche Bus-Topologien möglich, wobei die logische Ring-Struktur am häufigsten Anwendung findet. Das bitstrom-orientierte Übertragungsprotokoll mit Kunststoff-Lichtwellenleitern kann dabei bis zu 64 Steuergeräte über eine unidirektionale Punkt-zu-Punkt-Verbindung miteinander verknüpfen, wobei jede Botschaft stets genau einmal den gesamten Ring durchläuft. Zusätzlich existieren Steuerbotschaften für die Netzverwaltung mit automatischer Fehlerbehandlung, jedoch ohne garantierte Botschaftslatenz.

[ZIS14, Kap. 3.4]

Automotive Ethernet

Mit zunehmendem Entwicklungsfortschritt können auch moderne Bussysteme neben dem Controller Area Network den Anforderungen nicht mehr gerecht werden. Auf Grund der hohen Kosten und dem ständigen Entwicklungsaufwand der bisher verwendeten Bussysteme hat sich das, bereits in vielen Bereichen angewandte, Ethernet, spezifiziert nach IEEE 802.3, als Alternative etabliert. Ethernet als Bussystem ist kostengünstig und vereinfacht die Integration moderner Technik, wie beispielsweise Smartphones, erheblich. Jedes Steuergerät hängt dabei innerhalb einer Sterntopologie per voll-duplex als Punkt-zu-Punkt-Verbindung an einem zentralen Kopplungspunkt (Switch), welcher die ankommenden Botschaften an genau ein Steuergerät weiterleitet. Das empfangende ECU wird dabei aus der, in der Botschaft enthaltenen, Ethernet-Adresse ermittelt. Der Switch ist in der Lage gleichzeitig Nachrichten von verschiedenen Absendern an mehrere Empfänger zu übertragen. Daten können dabei mit circa 100 MBit/s übermittelt werden. Verzögerungen in der Übertragung können lediglich durch das Ankommen mehrerer Botschaften für denselben Empfänger am Switch entstehen, ansonsten arbeitet das System kollisionsfrei. Ein Verlust von Nachrichten ist nur im Falle einer Unterdimensionierung des Pufferspeichers im Switch möglich.

[ZIS14, Kap. 3.5]

2.4 BUSMASTER

Der BUMASTER [ETA17] ist eine Open-Source CAN-Analysesoftware der Firma ETAS, welche 2011 entwickelt wurde und zur Analyse des CAN-Busses sowie der Simulation einzelner Steuergeräte bis hin zu einem ganzen System dient. Das Tool unterstützt dabei eine Vielzahl an CAN-Interfaces diverser Hersteller, welche die Verbindung zwischen diesem und dem On-Board-Diagnose-Port am Automobil herstellen. Dadurch lassen sich CAN-Botschaften auf den Bus senden und es wird die Möglichkeit geboten, durch eine Signalüberwachung und das Loggen der über den Bus gesandten Daten, den Nachrichtenverkehr im gesamten Netzwerk vollständig mitzuschneiden. Die übertragenen Nachrichten können farblich hinterlegt und in Form von Diagrammen grafisch dargestellt werden. Eine Nachrichtenfilterung kann sowohl auf Anwendungsebene durch einen Softwarefilter, als auch auf CAN-Controller-Ebene mittels eines Hardwarefilters geschehen. Eine Datenbank organisiert dabei die Nachrichten. Die im BUSMASTER verwendete Programmiersprache ist C++, das Tool verfügt jedoch über einen integrierten Converter, der es ermöglicht in bzw. von anderen CAN-Programmiersprachen, wie z.B. die Communication Access Programming Language zu übersetzen.
Die BUSMASTER-Software wurde bereits in vorangegangener Arbeit eingehend evaluiert [LI19b]. Abbildung 2.6 zeigt die Benutzeroberfläche der Software.

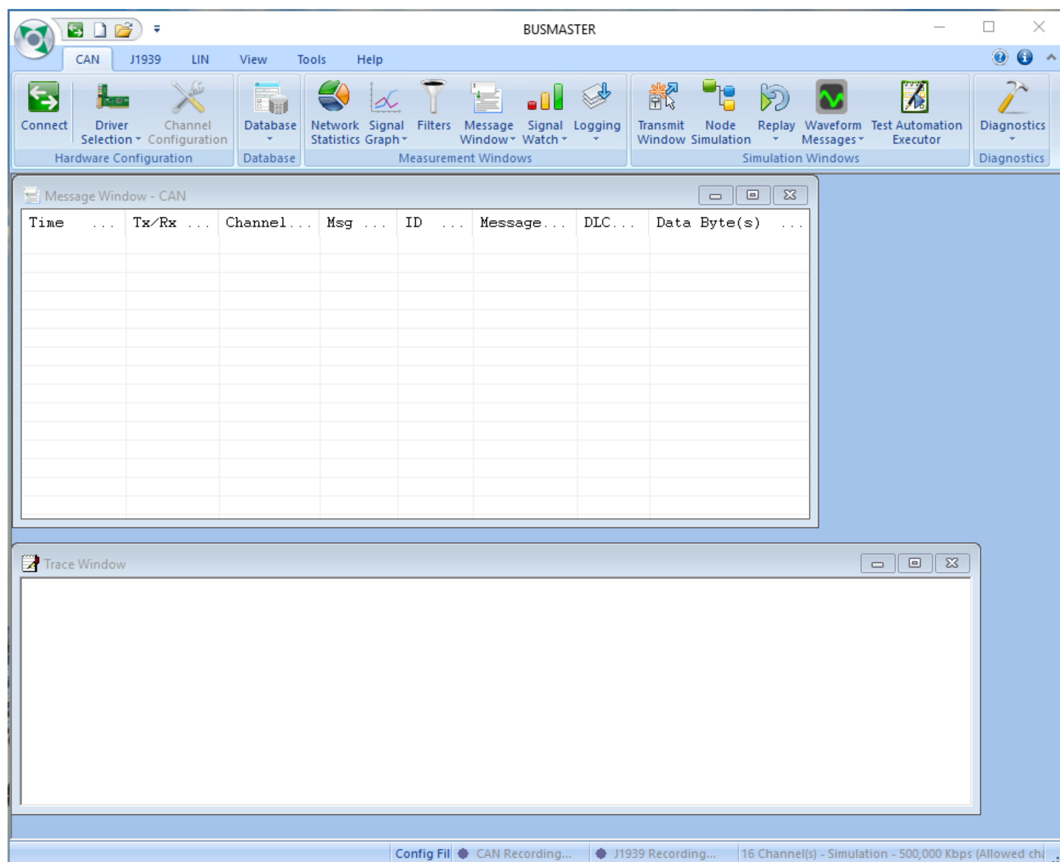


Abbildung 2.6: Benutzeroberfläche des BUSMASTERs. Quelle: [LI19a]

2.5 CAN-Demonstrator

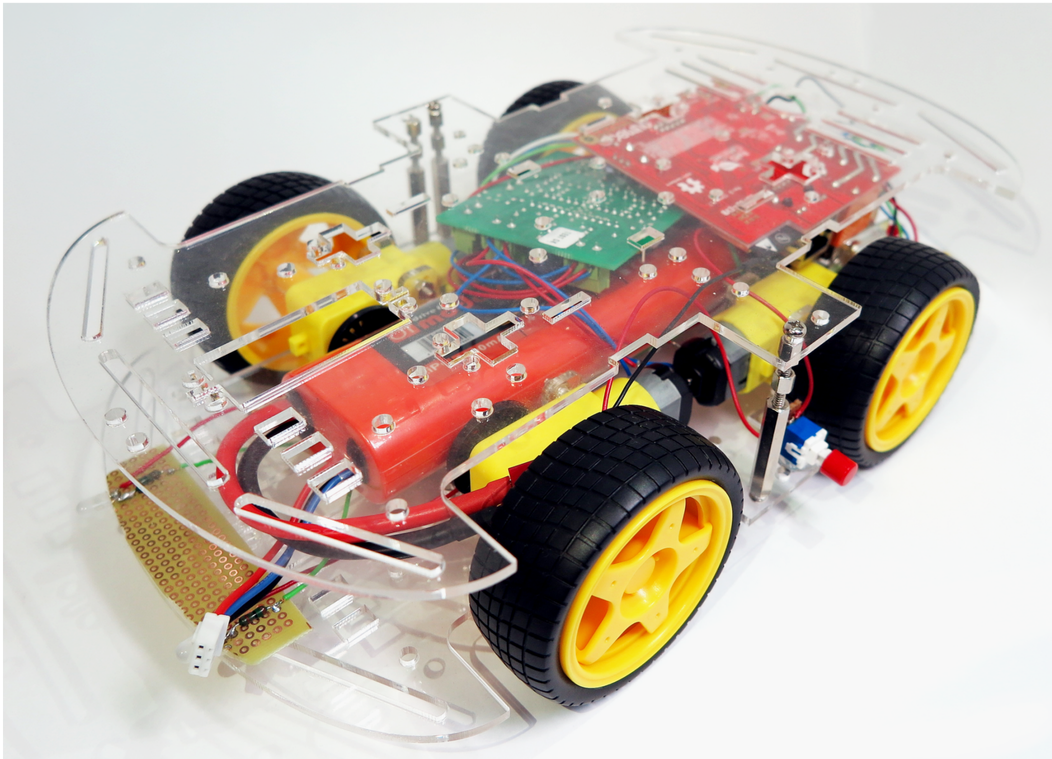


Abbildung 2.7: CAN-Demonstrator. Quelle: [HAN18, S.43]

Der CAN-Demonstrator [HAN18] (siehe Abbildung 2.7) ist ein Modellauto, welches über einen CAN-Bus verfügt und für forensische Zwecke eingesetzt wird. Die Simulation grundlegender Fahrfunktionen eines Automobils wie beispielsweise der Motorsteuerung oder diverser Beleuchtungen können über das Senden von CAN-Botschaften gesteuert werden. Der Demonstrator verfügt über einen, auf dem Entwicklungsboard (EVB) verbauten, Mikrocontroller AT90CAN128 der Firma Atmel, welcher zur Steuerung der Motoren verwendet wird. Das AVR-CAN-Board empfängt dabei als CAN-Knoten die Nachrichten auf dem Bus. Zur Realisierung der drahtlosen Kommunikation dienen zwei Mikrocontroller der Firma Espressif (ESP32), welche als Gateways fungieren und über das Wireless Local Area Network (WLAN) miteinander kommunizieren. Abbildung 2.8 stellt den schematischen Aufbau dieser CAN-Verbindung dar.

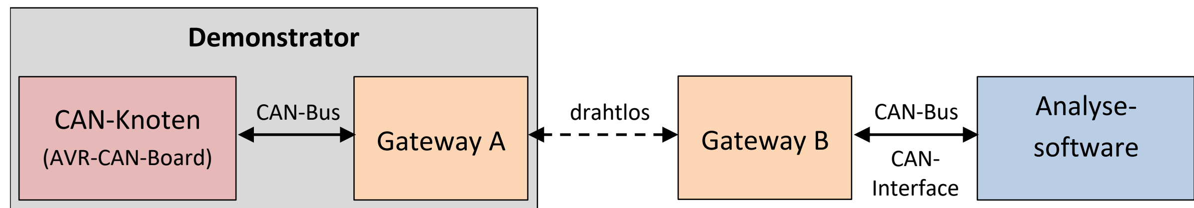


Abbildung 2.8: Schematischer Aufbau der CAN-Verbindung. Modifiziert nach: [HAN18, S. 22]

Das Gateway A im Demonstrator dient als Server und stellt als sogenannter Access Point das WLAN zur Verfügung, mit welchem sich der Client (Gateway B) verbinden kann. Wird eine CAN-Nachricht durch die Analysesoftware ausgesandt, leitet das CAN-Interface diese an den CAN-Transceiver des Boards weiter. Die Programmierung des Demonstrators kann über JTAG- und USB-Schnittstellen vorgenommen und jederzeit angepasst werden. Die Stromversorgung erfolgt über mobile Akkumulatoren mit einer Kapazität von 3000 mAh. Das Modellauto verfügt zudem über ein Intrusion Detection System zur Erkennung von Angriffen für forensische Analysen.

3 Methoden

In diesem Kapitel wird der Prozess der Konzeptionierung und die Auswahl der Softwarekomponente zur Umsetzung des grafischen Benutzerinterfaces beschrieben.

3.1 Präzisierung der Aufgabenstellung

Es soll ein grafisches Benutzerinterface entwickelt werden, welches über eine Bedienoberfläche mit Steuerelementen verfügt. Angelehnt an das Kombiinstrument in einem Automobil soll diese dem Anwender die Steuerung der CAN-Analysesoftware BUSMASTER ermöglichen und die, über den CAN-Bus laufenden, Nachrichten visualisieren. Um das Senden, Empfangen und entsprechendes Auswerten der Daten zu ermöglichen, ist eine bidirektionale Kommunikation vonnöten. Zur Visualisierung sollen die Steuerelemente des Graphical User Interface (GUI) entsprechend der ausgewerteten Statusinformationen der CAN-Nachrichten agieren. Für die individuelle Anpassung soll das Nutzerinterface zudem beliebig konfigurierbar sein.

Aus der Aufgabenstellung lassen sich folgende Ziele und Teilziele für die Entwicklung ableiten:

- Auswahl einer entsprechenden Entwicklungsumgebung für die GUI
- Entwurf der grafischen Bedienoberfläche
- Herstellung einer Verbindung zwischen GUI und BUSMASTER über eine virtuelle Schnittstelle
- Realisierung der bidirektionalen Kommunikation
    - Senden von CAN-Nachrichten
    - Empfangen von CAN-Nachrichten
- Visualisierung der CAN-Nachrichten
    - Integrierung der CAN-Nachrichten
    - Auswertung der CAN-Nachrichten

Abbildung 3.1 zeigt den schematischen Aufbau einer CAN-Analyse von der Hardware bis zum Anwender.
Das Steuergerät ist über ein CAN-Interface mit einem Computer und der darauf installierten CAN-Analysesoftware verbunden. Diese Software wiederum ist über eine virtuelle Schnittstelle mit dem, ebenfalls auf dem PC ausführbaren, grafischen Benutzerinterface verknüpft, in welches der Anwender durch die Bedienung der Steuerelemente Eingaben tätigen und die entsprechenden Reaktionen anhand der Auswertung der CAN-Nachrichten als Ausgabe wahrnehmen kann.

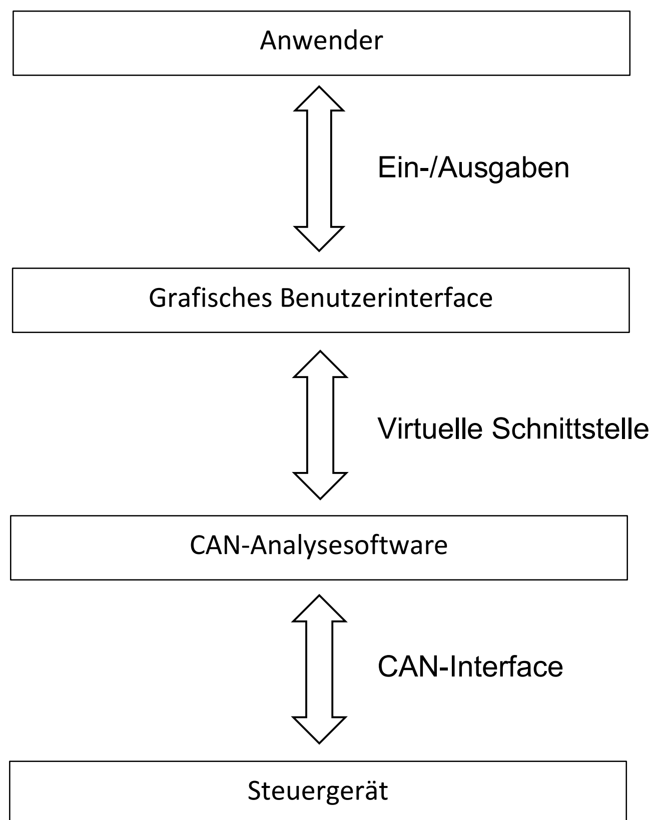


Abbildung 3.1: Schematischer Aufbau der CAN-Analyse. Quelle: [LI19a]

3.2 Grundkonzeptionierung

Zur Realisierung dieser Arbeit wurden die im Folgenden gelisteten Hardware- und Softwarekomponenten vorgegeben:

- Die Analysesoftware BUSMASTER, der ETAS GmbH, für welche das grafische Benutzerinterface entwickelt werden soll.
- Zur Übertragung der Nachrichten von einem Steuergerät an den Computer wird das CAN-Interface VN1610 von Vector eingesetzt.
- Für Systemtests wird der CAN-Demonstrator verwendet.

Gemäß der Funktionen im Automobil bzw. dem CAN-Demonstrator sollen Beleuchtungselemente wie Licht, Blinker und Warnblinker sowie jene Komponenten der Motorsteuerung implementiert werden. Diese sollen die Möglichkeit zur Geschwindigkeitsregulierung über Gas und Bremse sowie die Auswahl der Fahrtrichtung über eine Checkbox für den Rückwärtsgang beinhalten. Eine Handbremsfunktion soll automatisch kontinuierlich die Geschwindigkeit verringern. Eine Anzeige soll die Änderungen der Geschwindigkeit darstellen. Zudem sollen eine Hupe, eine Stopp-Funktion zum sofortigen Anhalten und ein Button zur Verbindungsherstellung bzw. -trennung realisiert werden. Zusätzlich sollen die übertragenen CAN-Botschaften visualisiert werden.

Das Blockbild in Abbildung 3.2 soll das grobe Konzept der grafischen Oberfläche zeigen, wobei blau hinterlegte Elemente reine Anzeigeelemente darstellen sollen und rote jene zur Bedienung, beispielsweise Buttons oder Checkboxen, sind.

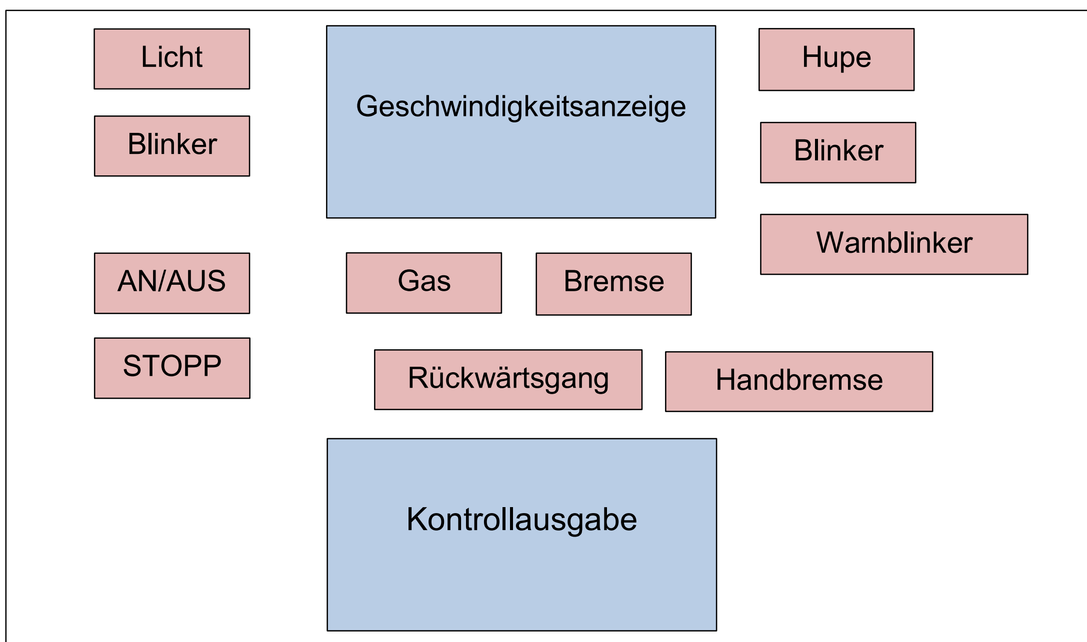


Abbildung 3.2: Blockbild der GUI. Quelle: [LI19a]

Für die Kommunikation zwischen GUI und BUSMASTER muss eine virtuelle Schnittstelle eingerichtet werden, welche die beiden Programme miteinander verbindet und ein Austauschen der CAN-Nachrichten zwischen diesen beiden realisiert.
Zur entsprechenden Visualisierung der CAN-Nachrichten in der GUI sollen die Statusnachrichten des Demonstrators ausgewertet und entsprechende Reaktionen programmiert werden. Eine Automatisierung dieser Reaktionen, insbesondere bei der Beleuchtung, soll über Timer-Funktionen umgesetzt werden.
Über den An/Aus-Button soll die Verbindung zum BUSMASTER aufgebaut werden. Abbildung 3.3 zeigt den geplanten Programmablauf für diese Funktion. So soll bei Betätigung des Buttons zuerst geprüft werden, wie der aktuelle Status ist. Besteht noch keine Verbindung (Zustand = Aus), soll diese hergestellt werden. Zeitgleich soll die Variable, die diesen Zustand charakterisiert, geändert werden sowie die Anzeige auf dem Button von „An“ zu „Aus“, um die jeweils aktuelle Funktion von diesem anzuzeigen. Besteht bereits eine Verbindung (Zustand = An), soll diese getrennt und die Variable für den Zustand sowie die Anzeige konträr geändert werden.

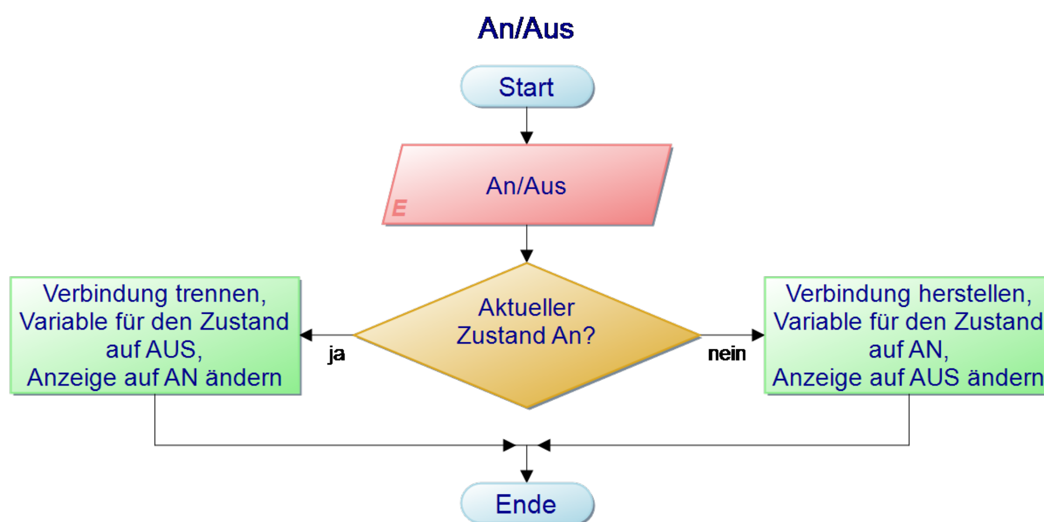


Abbildung 3.3: Skizze über den Ablauf des Programms für die An/Aus-Funktion. Quelle: [LI19a]

Für die Blinkfunktion soll geregelt sein, dass kein beidseitiges Blinken möglich ist. Diese muss folglich so programmiert werden, dass bei Betätigung des einen Blinkers, der andere deaktiviert wird. Dies ist besonders bei einem Blinkerwechsel von Bedeutung. Eine zusätzliche Warnblinkfunktion soll ein beidseitiges Blinken realisieren.

Über die zwei Buttons Gas und Bremse soll es möglich sein, die Geschwindigkeit zu regulieren. Abbildung 3.4 zeigt den geplanten Programmablauf, wobei bei eingehender CAN-Nachricht zur Änderung der Geschwindigkeit zuerst geprüft werden soll, ob es sich um eine Verringerung oder Beschleunigung handelt. Ist letzteres der Fall, soll die aktuelle Geschwindigkeit geprüft werden. Ist diese kleiner als 250, soll sie um den Wert 5 erhöht werden. Ist diese größer oder gleich 250, soll sie auf den Wert 255 gesetzt werden. Äquivalentes Vorgehen für den Fall, dass die Geschwindigkeit verringert werden soll. Ist diese aktuell größer als 5, soll sie um den Wert 5 verringert werden. Ist die Geschwindigkeit kleiner oder gleich 5, soll sie auf den Wert 0 festgesetzt werden. Im Anschluss soll immer die aktuelle Geschwindigkeit in der Anzeige ausgegeben werden.

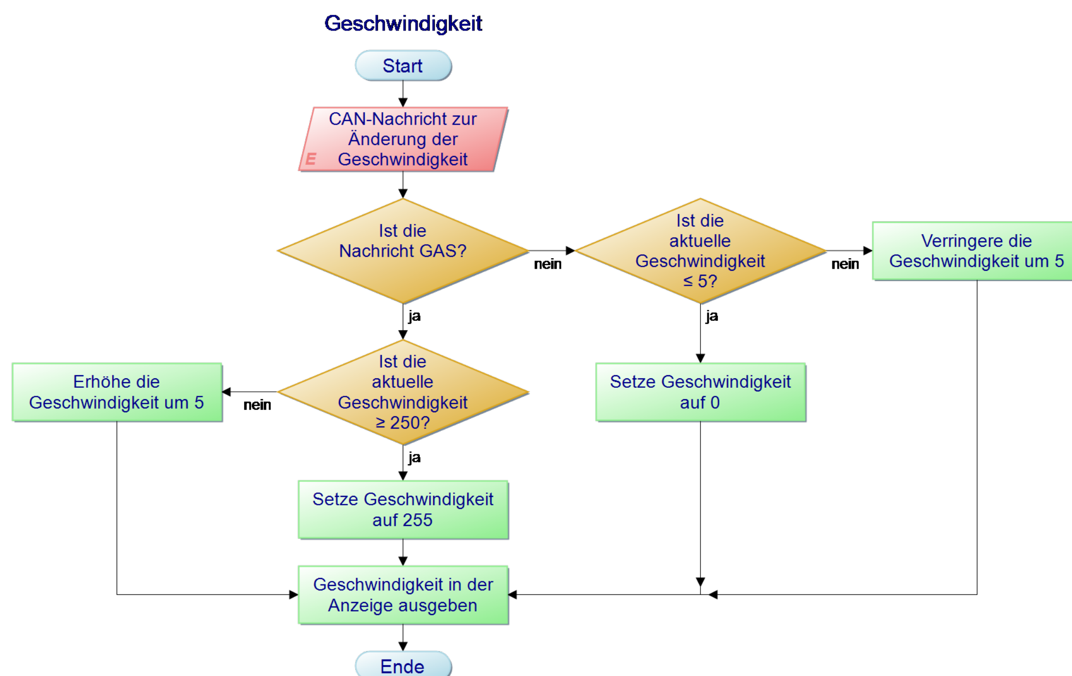


Abbildung 3.4: Skizze über den Ablauf des Programms für die Geschwindigkeits-Funktion. Quelle: [LI19a]

Es soll zudem geregelt sein, dass bei angezogener Handbremse, kein Gas gegeben werden kann. Hierfür muss der Checkbox eine Variable mit einem booleschen Wert zugewiesen werden, welche geprüft werden kann.

Während des Bremsvorganges soll neben der Verringerung der Geschwindigkeit auch die Bremsbeleuchtung aktiviert werden, diese soll jedoch erlöschen, sobald der Mauszeiger den Button Bremse verlässt, similär dem Fuß, der das Gaspedal nicht mehr betätigt.

Um der Funktionalität in einem realen Kfz möglichst nahe zu kommen, soll die Umsetzung des Rückwärtsganges über eine Checkbox realisiert werden. Zudem soll für diese Fahrtrichtung eine Geschwindigkeitsbegrenzung programmiert werden. Die Beschränkung soll den Wert 40 annehmen, wobei dieser nicht mit der Geschwindigkeit in Kilometern pro Stunde gleichzusetzen ist. Die Umsetzung soll hierbei wie bei der Handbremse mit einer booleschen Variable gelöst werden.

Mit dem Zweck den CAN-Demonstrator im Ausnahmefall abrupt zum Stehen zu bringen, soll die grafische Bedienoberfläche einen Stopp-Button vorsehen, bei dessen Betätigung eine CAN-Nachricht zum Stoppen aller Motorfunktionen ausgesandt wird.

Des Weiteren soll innerhalb der GUI eine Kontrollausgabe in Form einer Textbox implementiert werden. Diese gibt alle über den CAN-Bus laufenden Nachrichten aus. Dabei werden einzelnen Datenbytes farblich markiert, um eine entsprechend manuelle Auswertung dieser zu ermöglichen.

Das Programm soll zudem die Möglichkeit der variablen Gestaltung der Konfiguration im BUSMASTER beinhalten. Dafür sollen Konfigurations- und Datenbankdateien hinzugefügt werden können sowie die Speicherung der Einstellungen innerhalb dieser Dateien umgesetzt werden.

3.3 Auswahl der Entwicklungsumgebung

Wie bereits in Kapitel 1 erläutert, verfügt die CAN-Analysesoftware BUSMASTER über keine grafische Bedienoberfläche zur Steuerung. Aus diesem Grund wird in der vorliegenden Bachelorarbeit ein eigenes, konfigurierbares, grafisches Benutzerinterface entwickelt, welches eine entsprechende Steuerung ermöglicht und die über den CAN-Bus laufenden Nachrichten visualisiert.

Zur Umsetzung der Ziele wird eine Entwicklungsumgebung benötigt, welche folgende Anforderungen erfüllt:

- Die Software soll kostengünstig sein.
- Die Software soll die BUSMASTER Library unterstützen.
- Die Software soll online gut dokumentiert sein.
- Das Zielkriterium der Software soll eine Windowsoberfläche sein.
- Die Software soll die Programmierung einer GUI bestmöglich unterstützen.

Die Hauptanforderungen an die Entwicklungsumgebung sind die Möglichkeit der Programmierung eines grafischen Benutzerinterfaces und möglichst niedrige Kosten für diese. Auf der Suche nach derartigen Entwicklungsumgebungen werden im Folgenden untenstehende Tools näher untersucht und gegeneinander abgegrenzt:

- Eclipse IDE
- NetBeans IDE
- Visual Studio (VS)
- SharpDevelop

Eclipse IDE

Version: 2019-03
Die Eclipse IDE verfügt über diverse Plug-Ins, unter anderem den WindowBuilder, welcher die Erstellung von GUI-Anwendungen per Drag and Drop-Funktion einfach realisieren lässt. Die Entwicklungsumgebung ist kostenfrei und unterstützt hauptsächlich die Programmiersprache Java, lässt sich aber durch zusätzliche Plug-Ins auf C, C++, Perl sowie PHP und andere erweitern. Die Eclipse Foundation selbst hat auf ihrer Website eine Benutzerdokumentation veröffentlicht. Zudem lassen sich zahlreiche Foren im Internet finden, die unterschiedliche Herangehensweisen für diverse Projekte beleuchten. Das Einbinden einer externen Bibliothek ist möglich. Eclipse ist plattformunabhängig und erfüllt somit das Zielkriterium der Windowsoberfläche, die zugehörige Website ist jedoch teilweise veraltet. Eine Übersicht der Eigenschaften ist Tabelle 3.1 zu entnehmen.

Tabelle 3.1: Bewertung der Eclipse IDE

Eclipse	**Vorteile**	**Nachteile**
Kosten	Keine Lizenzkosten	-
Dokumentation	Gut	-
Handhabung	Einfach per Drag & Drop, durch Plug-Ins beliebig erweiterbar, Autovervollständigung	Website teilweise veraltet
Library	Unterstützt BUSMASTER-Bibliothek	-

[EC19a], [EC19b], [HEG17]

NetBeans IDE

Version: 11.0
Eine weitere Möglichkeit stellt die NetBeans IDE dar, welche ebenfalls über einen GUI-Builder verfügt und kostenfrei downloadbar ist. NetBeans unterstützt die Betriebssysteme OS X, Linux sowie Windows und supportet eine Vielzahl an Programmiersprachen wie beispielsweise Java, C, C++, Python und PHP. Die Dokumentation ist dagegen nur sehr gering. Grundlegendes wird zwar vom Hersteller selbst veröffentlicht, jedoch gibt es keine ausführlicheren Projektanleitungen. Zum Einbinden einer externen Library steht eine Importfunktion zur Verfügung. NetBeans verfügt über eine automatische Codevervollständigung, ist jedoch in seiner Reaktionszeit langsam und verbraucht zudem viel Speicher. Die Erweiterung über Plug-Ins ist begrenzt. Eine Übersicht der Eigenschaften ist Tabelle 3.2 zu entnehmen.

Tabelle 3.2: Bewertung der NetBeans IDE

NetBeans	Vorteile	Nachteile
Kosten	Keine Lizenzkosten	-
Dokumentation	-	Wenig, grundlegendes vom Hersteller
Handhabung	Autovervollständigung	Langsam in der Reaktionszeit, verbraucht viel Speicher, begrenzte Plug-In Erweiterung
Library	Unterstützt BUSMASTER-Library	-

[PCW10], [HEG17]

Visual Studio

Version: Community Edition 2019
Microsoft selbst bietet mit Visual Studio Community 2019 eine kostenlose Entwicklungsumgebung zur Erstellung von Anwendungen für Windows, Android und iOS sowie Webanwendungen und Cloud-Diensten. VS unterstützt eine Vielzahl an Programmiersprachen wie beispielsweise C#, Visual Basic, C++, HTML und Python. Eine ausführliche Dokumentation zur Anwendung und unzählige Programmierbeispiele sind direkt auf der Entwicklerseite zu finden. Zudem gibt es viele Foreneinträge zur Unterstützung der Programmierung in VS. Durch Drag-and-Drop sowie eine automatische Codevervollständigung wird die Entwicklung einer GUI begünstigt. Zudem können Packages eingebunden werden, die eine Erweiterung der Toolbox für die GUI ermöglichen. Eigenes Testen bestätigt die Unterstützung der entsprechenden BUSMASTER-Library für die virtuelle Schnittstelle. Da sich die IDE zur Erstellung von Windowsanwendungen eignet, ist das Zielsystem demzufolge auch Windows. Visual Studio benötigt viel Arbeitsspeicher, was bei leistungsstarken Rechnern jedoch kein Problem darstellt. Eine Übersicht der Eigenschaften ist Tabelle 3.3 zu entnehmen.

Tabelle 3.3: Bewertung der Visual Studio IDE

Visual Studio	**Vorteile**	**Nachteile**
Kosten	Keine Lizenzkosten	-
Dokumentation	Sehr gut	-
Handhabung	Einfach per Drag & Drop, Unterstützung durch Codevervollständigung, durch Packages erweiterbar	Benötigt viel Arbeitsspeicher
Library	Unterstützung der BUSMASTER-Library	-

[MS19a], [MS19b]

SharpDevelop

Version: 5
SharpDevelop, auch bekannt als #develop, ist eine freie integrierte IDE, welche auf das .NET Framework aufbaut. Die windows-basierte Entwicklungsumgebung verfügt über einen GUI-Builder und unterstützt seit Version 5 nur noch C# als Programmiersprache, bietet aber die Unterstützung der benötigten Library. Es gibt jedoch weder eine Dokumentation des Herstellers bzw. Entwicklers noch Foreneinträge. Zudem steht nur eine 32-Bit-Version zum Download verfügbar. SharpDevelop zeichnet sich durch einen geringen Ressourcenbedarf in der Handhabung aus und kann somit auch auf leistungsschwächeren Systemen ausgeführt werden. Eine Übersicht der Eigenschaften ist Tabelle 3.4 zu entnehmen.

Tabelle 3.4: Bewertung der SharpDevelop IDE

SharpDevelop	**Vorteile**	**Nachteile**
Kosten	Keine Lizenzkosten	-
Dokumentation	-	Keine Dokumentation
Handhabung	Geringer Ressourcenbedarf	Nur 32-Bit-Version
Library	Unterstützung der BUSMASTER-Library	-

[WIL12]

Auf Grund der einfachen und anwenderfreundlichen Handhabung sowie der ausführlichen Dokumentation sind sowohl die Eclipse IDE, als auch Visual Studio als Entwicklungsumgebung für dieses Projekt geeignet. NetBeans und SharpDevelop schneiden im Vergleich der Funktionalität und Handhabung schlechter ab und kommen daher für die Verwendung nicht in Frage. Schließlich wurde sich in dieser Arbeit für die kostenfreie Community Edition von der von Microsoft angebotenen Entwicklungsumgebung Visual Studio unter Verwendung des .NET Frameworks entschieden, da diese als leistungsstärkste unter allen Verglichenen hervorgeht. Eine sehr gute Dokumentation sowie zahlreiche Projektanleitungen im Internet bieten Unterstützung bei der Entwicklung. Durch die Drag-and-Drop-Funktion sowie Codevervollständigung innerhalb der Programmierung qualifiziert sich diese Variante als die mit der einfachsten Handhabung. Als Programmiersprache wurde die Standardsprache für Visual Studio, C#, in seiner Windows-Forms-Anwendung, gewählt. Eine Gegenüberstellung der vorgestellten Lösungen findet sich in der Tabelle 3.5.

Tabelle 3.5: Vergleich der Softwarelösungen

	Eclipse	**NetBeans**	**Visual Studio**	**SharpDevelop**
Dokumentation	Gut	Schlecht	Sehr gut	Schlecht
Handhabung	Drag & Drop, Codevervollständigung, erweiterbar, Website teilweise veraltet	Codevervollständigung, langsame Reaktionszeit, ressourcenlastig, begrenzt erweiterbar	Drag & Drop, Codevervollständigung, erweiterbar, ressourcenlastig	Geringer Ressourcenbedarf, nur 32-Bit-Version
Library-Unterstützung	Ja	Ja	Ja	Ja

4 Softwareentwicklung und Implementierung

Ziel ist es, die CAN-Analysesoftware BUSMASTER über eine grafische Bedienoberfläche anzusteuern sowie die über den CAN-Bus laufenden Nachrichten mit Steuerelementen zu visualisieren. Dabei soll die Möglichkeit der freien Konfigurierbarkeit bestehen.

In diesem Kapitel wird die Umsetzung der zuvor im Konzeptionsteil definierten Teilziele zur Entwicklung des grafischen Benutzerinterfaces erläutert. Hierbei wird auf die Besonderheiten der Programmierung eingegangen sowie Probleme in deren Umsetzung deutlich gemacht.

4.1 Kommunikation mit dem BUSMASTER

Um eine bidirektionale Kommunikation zwischen dem Benutzerinterface und der CAN-Analysesoftware zu ermöglichen, muss zuerst eine Verbindung zwischen dem Userinterface und dem BUSMASTER fundiert werden. Diese lässt sich unter Einbindung der CAN_MonitorApp-Library über eine virtuelle Schnittstelle realisieren. Um die Kommunikationsschnittstelle abzurufen muss zur Verwendung eine globale Variable deklariert und initialisiert werden.

Der BUSMASTER bietet Application Programming Interface-Funktionen, die während der Programmierung verwendet werden können, um mit der BUSMASTER-Anwendung zu interagieren. Eine Listung dieser sowie eine Anleitung zur Kommunikationsherstellung befinden sich im User Manual des BUSMASTERs, jedoch nur für die Programmiersprache C++. Diese lässt sich nicht auf C# übertragen, was zu Problemen führte.

In Folge dessen wurden untenstehende Befehle zur Realisierung der Verbindungsherstellung unter C# erarbeitet.

Einbinden der Library:
```
using CAN_MonitorApp;
```

Einrichtung der virtuellen Schnittstelle:
```
CAN_MonitorApp.BusMasterApp BUSMASTER = new CAN_MonitorApp.BusMasterApp();
CAN_MonitorApp.Comm Kommunikation = new CAN_MonitorApp.Comm();
```

Herstellung und Trennung der Verbindung des BUSMASTERs mit dem Bus:
```
BUSMASTER.Connect(X);
```
X ist entsprechend mit 1 – für Connect und 0 – für Disconnect zu ersetzen

4.1.1 Senden von CAN-Nachrichten

Zur Umsetzung der Kommunikation vom Userinterface zum BUSMASTER wurde für das Aussenden entsprechender CAN-Nachrichten an diesen zuerst die Struktur eines CAN-Frames sowie die des Datenfeldes einer CAN-Nachricht im Quellcode definiert. Der in diesem Kapitel relevante und bearbeitete Teil der gesamten CAN-Nachricht ist in Abbildung 4.1 grafisch dargestellt und soll dem Verständnis der künftigen Implementierungen dienen. Es werden hierbei lediglich die hervorgehobenen Felder bearbeitet und ausgewertet. Entsprechend wird das Feld der ID mit elf bzw. 29 Bits gefüllt. Die Länge der folgenden Daten in Byte von vier Bit wird innerhalb des Control Fields (CTR) angegeben. Zuletzt folgen die zugehörigen 64 Bit Daten.

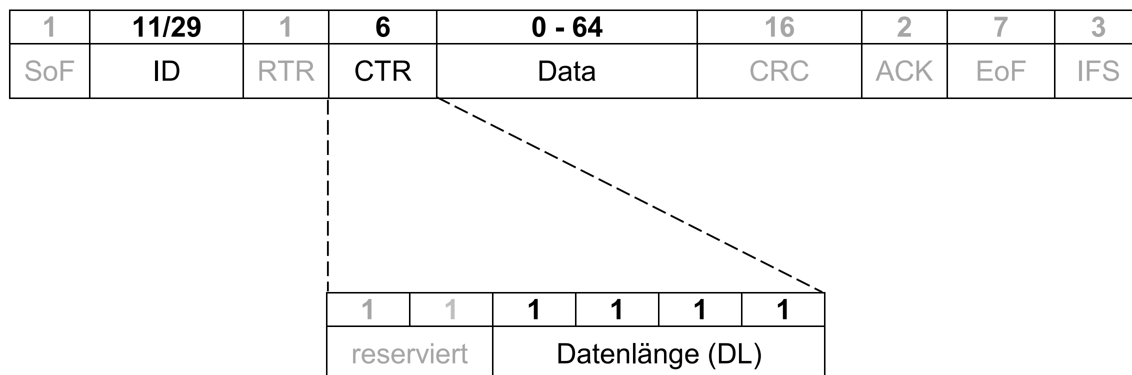


Abbildung 4.1: Relevante Teile der CAN-Nachricht. Quelle: [LI19a]

Da der BUSMASTER die Daten nicht bitweise entsprechend des Formats der CAN-Nachricht generiert, sondern jeweils volle Bytes verwendet, muss dies entsprechend bei der Implementierung beachtet und angepasst werden.
Folgender in Listing 4.1 dargestellter Code-Snippet zeigt die genannte Struktur eines CAN-Frames von 16 Byte.

```
struct CAN_FRAME
{
    public UInt16 id;
    public byte byte2;
    public byte byte3;
    public byte byte4;
    public byte byte5;
    public byte dataLength;
    public CAN_DATA data;
    public byte channel;
}
```

Listing 4.1: Struktur eines CAN-Frames in C#

Im ersten Datenbyte wird die ID der CAN-Nachricht als unsigned Integer von 16 Bit übergeben. Da eine CAN-ID je nach Art der Nachricht, Standard oder Extended, bis zu 29 Bit lang sein kann, sind Byte 0 bis 3 für diese reserviert. Diese einzigartige ID dient im realen Auto zur Entscheidung der ECUs, ob eine Nachricht für sie relevant ist oder verworfen werden kann. Des Weiteren gibt sie Aufschluss über den Inhalt der Botschaft. Gewisse IDs können ganz bestimmten Funktionen zugeordnet werden. Das folgende Byte gibt Aufschluss, ob es sich bei dieser Nachricht um ein Standard oder Extended (0 oder 1) Format des CAN-Frames handelt. Es schließt sich ein vom BUSMASTER intern reserviertes Byte an. Byte 6 enthält die Länge (Data Length (DL)) der in der darauffolgenden Struktur von acht Byte enthaltenen Daten und abgeschlossen wird das Frame durch die Angabe des CAN-Kanals. Diese Information ist ebenfalls von der CAN-Analysesoftware gesetzt.

Um die definierte CAN-Nachricht an den BUSMASTER zu senden, wurde eine entsprechende Funktion (siehe Listing 4.2) erstellt, welche beim Aufruf die Parameter ID und Datenlänge übergibt. Innerhalb dieser Funktion wird zuerst eine neue Nachricht generiert und diese mit der übergeben ID versehen. Byte zwei und drei geben Aufschluss darüber, um was für einen Nachrichtentyp (Standard oder Extended) es sich handeln soll und, ob ein Remote Transmission Request (RTR) vorliegt. 0 und 1 stehen hierbei für „gesetzt“ bzw. „nicht gesetzt“. An nächster Stelle wird die in der Funktion übergebene Datenlänge (datalen) angegeben. Folgend von 8 Byte CAN-Daten (data), die in die zuvor angelegte Struktur des Datenfeldes geschrieben werden müssen, wird die Nachricht durch die Angabe des CAN-Kanals abgeschlossen. Der Befehl `Kommunikation.SendCANMsg(Nachricht)` sendet die eben erstellte CAN-Nachricht an den BUSMASTER.

```
private void Nachrichtsenden(uint ID, byte datalen)
{
    CAN_MSGS Nachricht = new CAN_MSGS();
     Nachricht.m_unMsgID = ID;
     Nachricht.m_bEXTENDED = 0;
     Nachricht.m_bRTR = 0;
     Nachricht.m_ucDataLen = datalen;
     Nachricht.m_ucData = data;
     Kommunikation.SendCANMSg(Nachricht);
}
```

Listing 4.2: Funktion zum Senden von CAN-Nachrichten

4.1.2 Empfang von CAN-Nachrichten

Zur Realisierung der bidirektionalen Kommunikation sollen die über den CAN-Bus und somit über den BUSMASTER laufenden Nachrichten empfangen und ausgewertet werden. Dieser stellt hierbei den Server der Verbindung dar, die GUI den Client. Um diese als Client zu registrieren, dient die `RegisterClientForRx`-Funktion. Für diese wurde ein eigenes Event in Visual Studio angemeldet, welches beim Empfang einer Nachricht informiert wird und eine entsprechende Pipe angelegt, in deren Speicher eingehende Nachrichten für den Client geschrieben werden können. Durch die Erweiterung des Programmcodes, um die in Listing 4.3 dargestellte Funktion wurde das Empfangen und Verarbeiten eingehender Nachrichten realisiert. Hierbei wird zuerst ein neues CAN-Frame nach der in Listing 4.1 definierten Struktur erstellt und dessen Speicherbedarf ermittelt. Im Anschluss wird ein Speicherplatz dieser Größe für den Zwischenpuffer reserviert. Der Array-Inhalt der eingehenden Nachricht wird in den Zwischenpuffer übertragen und dieser wird in die CAN-Struktur geschrieben. Abschließend wird der reservierte Speicher wieder freigegeben.

```
private void Funktion(object sender)
{
   CAN_FRAME can = new CAN_FRAME();
   int size = Marshal.SizeOf(can);
   IntPtr ptr = Marshal.AllocHGlobal(size);
   Marshal.Copy(Eingang, 0, ptr, size);
   can = (CAN_FRAME)Marshal.PtrToStructure(ptr,can.GetType());
   Marshal.FreeHGlobal(ptr);
}
```

Listing 4.3: Funktion zum Empfang eingehender CAN-Nachrichten

4.2 Visualisierung mit Visual Studio

Um die über den BUSMASTER laufenden CAN-Nachrichten in der GUI zu visualisieren, müssen diese zuerst in die Programmierung integriert und eine entsprechende Funktion zur Auswertung der einzelnen Datenbytes erstellt werden. Innerhalb dieses Kapitels dient der CAN-Demonstrator als notwendige Hardwarekomponente, welche angesteuert werden kann.

4.2.1 Integrierung der CAN-Nachrichten in Visual Studio

Gemäß der in der Datenbank des BUSMASTERs angelegten CAN-Nachrichten, wurden diese in eine Enumerationsliste in den Code des grafischen Benutzerinterfaces eingepflegt. Dabei wurde jeder Nachricht die entsprechende ID zugewiesen. Eine Übersicht der implementierten CAN-Nachrichten ist der Tabelle 4.1 zu entnehmen.

Tabelle 4.1: Implementierte CAN-Nachrichten.
Quelle: modifiziert nach [HAN18, S.30, S.47]

ID	Nachricht
0x2	Stopp
0x3	Vorwärts
0x4	Rückwärts
0x5	Blinker links
0x6	Blinker rechts
0x7	Warnblinker
0x8	Bremsleuchte
0x9	Frontscheinwerfer
0xA	Rückleuchte
0xB	Rückscheinwerfer
0xC	Hupe
0xD	Akku
0xE	Akkustand
0x10	Status
0x100	WiFi-Status

4.2.2 Auswertung der CAN-Nachrichten in Visual Studio

Um die CAN-Nachrichten auswerten und visualisieren zu können muss sich zunächst eingehend mit dem Aufbau der Statusnachricht befasst werden, welche als Rückantwort auf jede Änderung vom CAN-Demonstrator über den CAN-Bus gesandt wird und somit die Voraussetzung für die bidirektionale Kommunikation darstellt.

Die relevanten Bereiche für die in diesem Abschnitt vorgenommene Auswertung der Statusnachrichten des Demonstrators sind Abbildung 4.2 zu entnehmen. Das bearbeitete Datenfeld ist hierfür in seine einzelnen Komponenten dieser Statusnachricht aufgespalten.

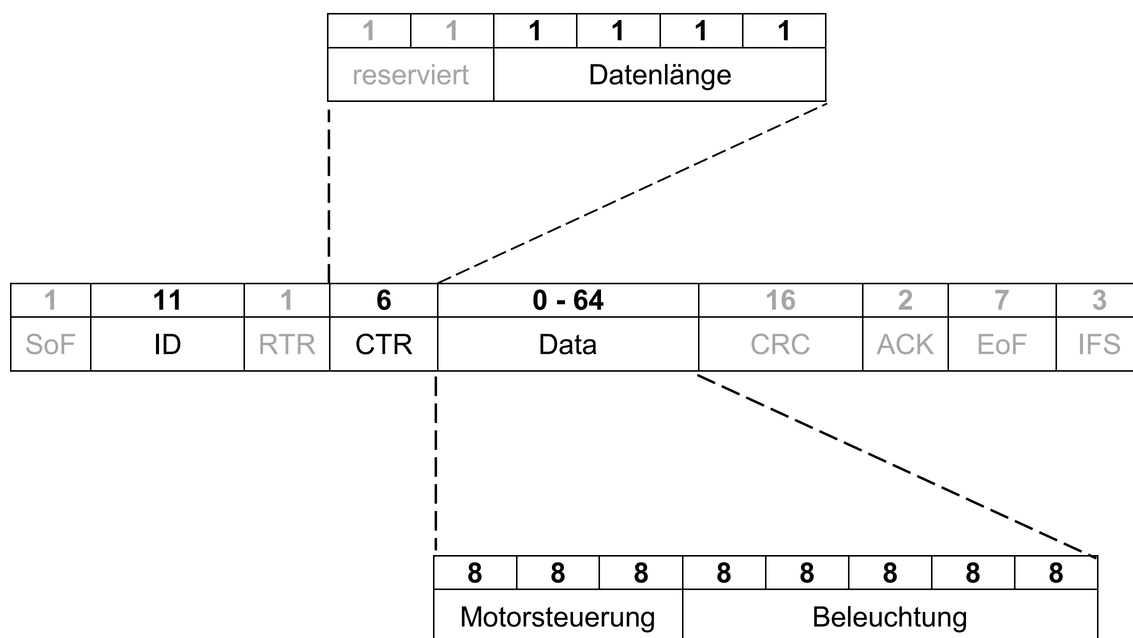


Abbildung 4.2: Statusnachricht des CAN-Demonstrators. Quelle: [LI19a]

Im Folgenden stellt Tabelle 4.2 den schematischen Aufbau des Datenfeldes einer solchen Statusnachricht mit dem jeweils möglichen Wertebereich der einzelnen Bytes dar.
Byte 0 enthält die Angabe der Fahrtrichtung des Automobils, wobei die 3 für vorwärts steht, die 4 für rückwärts. Der Wert der Geschwindigkeit in Byte 1 liegt zwischen 0 und 255. Mit einem Byte bzw. acht Bit lassen sich 256 verschiedene Zustände darstellen, weswegen sich 255 als der größtmögliche Wert für die Geschwindigkeit ergibt. Die im Folgenden übertragene Fahrtrichtung lässt sich durch 1 für rechts, 2 für links und der Zahl 0 für geradeaus differenzieren. Byte 3 bis 7 geben Aufschluss über den Status der Beleuchtungen Blinker, Warnblinker, Frontscheinwerfer sowie Rückleuchten, wobei eine 0 für Zustand „aus“ steht, eine 1 für „an“.

Tabelle 4.2: Aufbau der Datenbytes der Statusnachricht des CAN-Demonstrators.
Quelle: modifiziert nach [HAN18, S.47]

Byte	**Signal**	**Wertigkeit**
0	Fahrtrichtung (vorwärts/rückwärts)	3, 4
1	Geschwindigkeit	0-255
2	Fahrtrichtung (rechts/links/gerade)	1, 2, 0
3	Blinker links (aus/an)	0, 1
4	Blinker rechts (aus/an)	0, 1
5	Warnblinker (aus/an)	0, 1
6	Frontscheinwerfer (aus/an)	0, 1
7	Bremsleuchten (aus/an)	0, 1

Bei der Umsetzung in der Programmierung wird zuerst geprüft, ob die CAN-ID derer der Statusnachricht entspricht. Ist dem so, werden die einzelnen Datenbytes ausgelesen und entsprechend hinterlegte Reaktionen, wie das Aktivieren und Deaktivieren der Scheinwerfer, Blinker und anderer Beleuchtungen oder die Aktualisierung der Geschwindigkeitsanzeige, in der GUI ausgeführt.

Zur Lösung des Problems, dass das Programm bei eingehender Statusnachricht im Debugging-Modus abstürzt, wurde Multithreading verwendet. Um die einzelnen Elemente der GUI aus unterschiedlichen Threads bearbeiten zu können, werden bestimmte Zugriffsrechte benötigt. Zum sicheren Aufrufen eines Steuerelementes von einem anderen Thread, der dieses nicht erstellt hat, wurde die `System.Windows.Forms.Control.Invoke`-Methode genutzt. Diese ermöglicht das Aufrufen eines im Hauptthread angelegten Delegaten, der wiederum das Steuerelement aufruft. Der in Listing 4.4 dargestellte Codeausschnitt ermöglicht es die eingehenden Nachrichten in einer Textbox innerhalb der GUI auszugeben. Mit Hilfe von `InvokeRequired` wird zuerst geprüft, ob der Aufruf direkt an die GUI gehen kann oder ob hier ein Invoking vonnöten ist. Ist letzteres der Fall, muss das Steuerelement per `Invoke` geändert werden, da der Aufruf aus einem Backgroundthread kommt. Anderenfalls kann die Änderung an der GUI direkt aufgerufen werden.

```
private void Funkbox1(string output)
{
    if (textBox1.InvokeRequired)
    {
        textBox1.Invoke((MethodInvoker)delegate
        {
            textBox1.AppendText(output);
        });
    }
    else
    {
        textBox1.AppendText(output);
    }
}
```

Listing 4.4: Funktion für Multithreading

Die Anzeige der CAN-Nachrichten innerhalb der Kontrollausgabe (siehe Abb. 4.3 sowie Nr. 16 in Abb. 4.4) bietet zudem die Möglichkeit der Darstellung der über den Bus laufenden Nachrichten zur Analyse dieser und soll eine nahezu vollständige Anwendung aus der GUI heraus arrangieren. Zur anwendungsfreundlicheren Übersicht sind die Datenbytes des CAN-Frames farblich markiert. Bei jeder neuen Verbindungsherstellung wird die Kontrollausgabe geleert.

```
16 00 00 00 00 00 08 03 45 01 00 01 00 01 00 01
16 00 00 00 00 00 08 03 45 01 00 01 00 01 00 01
16 00 00 00 00 00 08 03 45 01 00 01 00 01 00 01
16 00 00 00 00 00 08 03 45 01 00 01 00 01 00 01
16 00 00 00 00 00 08 03 45 01 00 01 00 01 00 01
ID                DL        Data          Channel
```

Abbildung 4.3: Kontrollausgabe mit CAN-Nachrichten. Quelle: [LI19a]

In den ersten vier Bytes (0-3) wird der Identifier übergeben. Da der BUSMASTER die ID mit dem Least Significant Byte als erstes ausgibt, entsprechen die folgenden leeren Bytes den vorangestellten Nullen. Für diese Arbeit werden nur Identifier verwendet werden, welche maximal zwei Byte belegen. Die IDs werden innerhalb der Kontrollausgabe als Dezimalzahl dargestellt. Es folgt ein Byte für die Angabe, ob es sich um ein Standard (00) oder Extended (01) CAN-Frame handelt und ein Byte, welches intern vom BUSMASTER reserviert ist. Das sechste Byte (DL) gibt Aufschluss über die Länge der folgenden Datenbytes (Data). Diese beträgt maximal 8 Byte. Die Aufschlüsselung der Bedeutung der einzelnen Farben ist wie folgt:

- Fahrtrichtung (vorwärts/rückwärts)
- Geschwindigkeit
- Fahrtrichtung (links/rechts/gerade)
- Blinker links
- Blinker rechts
- Warnblinker
- Frontscheinwerfer
- Bremsleuchten

Das letzte Byte (Channel) gibt den aktuellen CAN-Kanal an. Eine komplette Übersicht der CAN-Nachricht mit entsprechendem Signal und zugehörigem Wertebereich ist Tabelle 4.2 zu entnehmen.

4.2.3 Grafische Bedienoberfläche

Die Umsetzung der grafischen Bedienoberfläche erfolgte durch das Einbinden verschiedener Button-Elemente und Checkboxen zur Steuerung. Für diese sind Klick- und Bewegungsereignisse erstellt worden, worüber entsprechend in der Programmierung hinterlegte Funktionen ausgeführt werden. Die Verwendung von Textboxen, Labels und weiteren Anzeigeelementen dient der Ausgabe und Darstellung der zu visualisierenden CAN-Nachrichten innerhalb der Oberfläche. Abbildung 4.4 zeigt eine Bildschirmaufnahme der grafischen Bedienoberfläche des Interfaces.

Verbindungsherstellung

Im Menü-Streifen am oberen Rand öffnet sich unter Settings (1) ein Menü, worüber die Datenbank und die für das Projekt notwendige Konfigurationsdatei dem BUSMASTER hinzugefügt werden können. Diese beiden Dateien stellen die selbst gewählte Voraussetzung für den Verbindungsaufbau durch die GUI dar. Sind diese nicht eingebunden, lässt sich keine Verbindung zum BUSMASTER herstellen. Ein hierfür programmiertes Pop-Up-Fenster informiert über das notwendige Hinzufügen der fehlenden Einstellung. Über die Diskette (2) im Menü-Punkt daneben können die Einstellungen der ausgewählten Konfiguration gespeichert werden. Bei erfolgreicher Speicherung öffnet sich ein Pop-Up-Fenster, um auf diesen Zustand hinzuweisen. Der Anwender kann durch diese Einstellungen verschiedene Konfigurationen verwenden und speichern sowie bereits erstellte Datenbanken variabel nutzen.

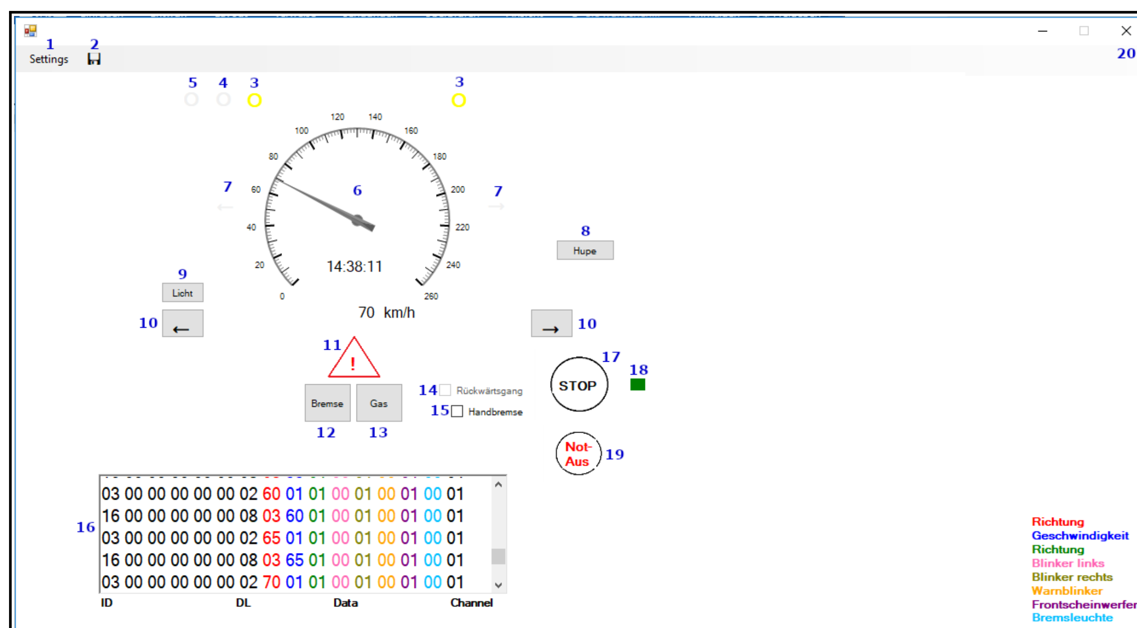


Abbildung 4.4: Grafische Bedienoberfläche des Interfaces. Quelle: [LI19a]

Die Verbindung zum BUSMASTER und der Hardware lässt sich über den Start/Stop-Knopf (17) herstellen. Ist die Verbindung bereits hergestellt, ist, um diese zu trennen, auf dem Button „STOP“ ersichtlich und umgekehrt. Beim Betätigen des Buttons wird die Verbindung hergestellt und die GUI als Client registriert. Es wird dann eine neue Pipe angelegt, aus der der Client zu lesen beginnt. Bei aktiver Verbindung zum BUSMASTER wird kontinuierlich eine CAN-Nachricht mit der ID 0x100 vom verbundenen Gateway ausgesandt, um über den WiFi-Verbindungsstatus zum Server-Gateway auf dem CAN-Demonstrator zu informieren. Wird im Datenbyte hier eine 1 übertragen ist die Verbindung zum Auto hergestellt und es können CAN-Nachrichten auf den Bus gesandt werden. Um über diesen Verbindungsstatus zu informieren, dient ein Label (18). Ist diese aufgebaut, erscheint das Label grün, ist dem nicht so, ist es rot. Wird die Verbindung über „STOP“ getrennt, wird die Pipe geschlossen. Zum Schließen des Programms dient ein Kreuz (20) in der rechten oberen Ecke, welches bei Betätigung eine Funktion auslöst, die wiederum das „Schließen-Ereignis“ induziert. Die Verbindung zum BUSMASTER wird dabei automatisch beendet, falls diese noch besteht. Es öffnet sich daraufhin ein Pop-Up-Fenster (siehe Abbildung 4.5 links), in dem gefragt wird, ob die Konfigurationseinstellungen in der Datei gespeichert werden sollen. Falls ja, wird geprüft, ob eine Konfigurationsdatei hinzugefügt wurde. Ist dem so, wird in diese gespeichert, ist dem nicht so, wird im Fenster (siehe Abbildung 4.5 rechts) darauf hingewiesen, dass keine Datei vorhanden ist und das Programm schließt sich.

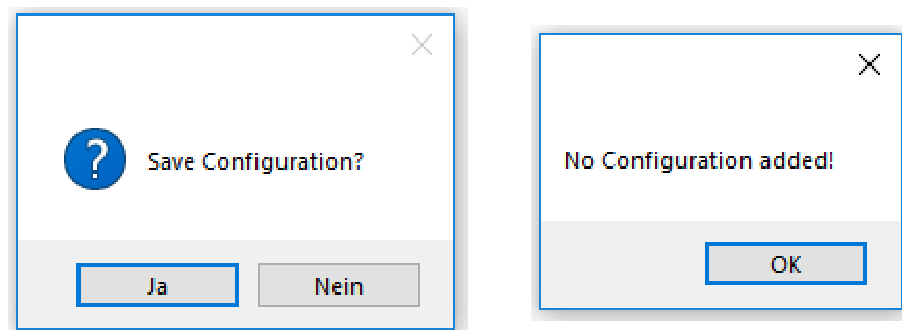


Abbildung 4.5: Hinweisfenster in der GUI. Quelle: [LI19a]

Beleuchtung

Bei der Betätigung des Buttons Licht (9) wird durch ein Klick-Event geprüft, ob der Button bereits gedrückt wurde und die Scheinwerfer aktiv sind. Trifft dies zu wird eine entsprechende CAN-Nachricht zum Deaktivieren der Front- und Rückscheinwerfer über den BUSMASTER auf den CAN-Bus gesandt. Ist dem nicht so, lautet die Information im übermittelten Datenbyte „Scheinwerfer aktivieren“. Bei der Auswertung der Rückantwort in Form einer Statusmeldung, färben sich die Scheinwerfer in der GUI (3) gelb.
Die jeweiligen Blinker lassen sich über den linken oder rechten Button (10) setzen. In der auszuwertenden Statusnachricht werden die Blinker der GUI (7) aktiviert und es ist entsprechend geregelt, dass bei einem Blinkerwechsel der jeweils andere Blinker deaktiviert wird. Ein gleichzeitiges, beidseitiges Blinken ist somit nur durch die Warnblinkerfunktion (11) möglich. Um die entsprechenden Labels in der GUI wie im Combi-Panel eines Automobils blinken zu lassen wird eine Timerfunktion eingesetzt, welche die Farbe des Labels in einem bestimmten Intervall ändert.

Beschleunigung

Unterhalb des Warnblinkers befinden sich die simulierten Pedale für Bremse (12) und Gas (13). Bei jedem Klick auf Gas wird eine CAN-Nachricht zur Erhöhung der aktuellen Geschwindigkeit um den Wert 5 ausgesandt. Da die Höchstgeschwindigkeit des Demonstrators dem maximalen Wert, der übergeben werden kann, entspricht, wird in der Programmierung geprüft, welcher Wert für die Geschwindigkeit vorliegt und bis 250 um 5 erhöht. Anschließend wird der Wert auf 255 festgesetzt. Die CAN-Nachricht zur Beschleunigung enthält im ersten Datenbyte die Geschwindigkeit und im zweiten die Fahrtrichtung. Da in der GUI selbst keine Möglichkeit zur Lenkung implementiert ist, wird hier der Status der Blinker in Betracht gezogen. So wird bei aktivem rechten Blinker als Fahrtrichtung eine 1 für rechts übergeben. Ist der linke Blinker an, wird eine 2 übertragen. Die entsprechend in der Antwort des Demonstrators übergebene Geschwindigkeit wird dem Tachometer (6) sowie der digitalen Geschwindigkeitsanzeige übermittelt. Innerhalb dieser ist zudem eine Anzeige der aktuellen System-Uhrzeit implementiert. Die Realisierung des Tachometers erfolgte über die Verwendung des AGaugeApp-Packages[1].
Analog ist die Funktion der Bremse aufgebaut. Beim Bremsvorgang wird die, in der CAN-Nachricht übermittelte, Geschwindigkeit um 5 verringert, solange diese größer 0 ist. Andernfalls wird der Wert auf 0 gesetzt. Eine weitere Nachricht zur Aktivierung der Bremsleuchte (5) wird ausgesandt. Die Anwendung eines DragLeave-Events bewirkt, dass das Bremslicht ausgeht, sobald der Mauszeiger den Bremsbutton verlässt.
Die Handbremse (15) wird durch eine Checkbox imitiert (siehe Listing 4.5). Wird diese betätigt, erfolgt zuerst die Prüfung des aktuellen Zustands. Ist diese noch nicht „gezogen“ (hbrake = true), wird unter permanenter Ausgabe der Geschwindigkeit in der Textbox und der Tachoanzeige der Gas-Button ausgegraut und ein Timer gestartet. Dieser prüft, ob die Geschwindigkeit größer 0 ist und verringert diese daraufhin um den Wert 1. Zeitgleich wird das Bremslicht (5) manuell in der GUI aktiviert, da für dieses in der Statusnachricht des Demonstrators keine Information übertragen wird. In die auszusendende CAN-Nachricht wird in das erste Nutzbyte (data[0]) eine 1 für „Bremslicht an“ geschrieben. Über den Funktionsaufruf zum Senden der CAN-Nachricht, wird die ID der Bremsleuchte sowie die Länge 1 der folgenden Daten übergeben und an den BUSMASTER geschickt. Anschließend wird die Variable für den Zustand der Handbremse geändert, sodass diese jetzt „gezogen“ ist.

```
private void checkBox_hbrake_CheckedChanged(object sender, EventArgs e)
{
    if (hbrake == true)
    {
        this.textBox_speed.Text = speed.ToString();
        this.aGauge1.Value = speed;
        this.button_gas.Enabled = false;
        timer_handbremse.Enabled = true;
        label_bremslicht.ForeColor = System.Drawing.Color.Red;
        data[0] = 1;
        Nachrichtsenden((uint)NachrichtenIDs.Bremsleuchte, 1);
        hbrake = false;
    }
```

Listing 4.5: Funktion der Handbremse (1)

[1] https://www.codeproject.com/Articles/17559/A-Fast-and-Performing-Gauge [Eingesehen am: 13.08.2019]

```
    else
    {
        this.textBox_speed.Text = speed.ToString();
        this.aGauge1.Value = speed;
        this.button_gas.Enabled = true;
        timer_handbremse.Enabled = false;
        label_bremslicht.ForeColor = System.Drawing.Color.Transparent;
        data[0] = 0;
        Nachrichtsenden((uint)NachrichtenIDs.Bremsleuchte, 1);
        hbrake = true;
    }
}
```

Listing 4.6: Funktion der Handbremse (2)

Wenn die Handbremse gelöst wird, werden analog dieselben Befehle durchlaufen, jedoch mit invertierten Werten (Vgl. Listing 4.6).
Um der Funktionsweise in einem realen Kfz möglichst nahezukommen, sieht die grafische Oberfläche eine Checkbox zum Einlegen des Rückwärtsganges (14) vor. Diese kann jedoch nur bei Stillstand betätigt werden. Beträgt die Geschwindigkeit größer null, wird diese Funktion entsprechend ausgegraut. Ist der Rückwärtsgang eingelegt, wird die Rückfahrleuchte (4) durch eine entsprechende CAN-Nachricht aktiviert. Zudem ist die maximale Geschwindigkeit in diese Richtung auf den Wert 40 beschränkt.
Die Not-Aus-Funktion (19) lässt bei Betätigung die Motorfunktion abrupt stoppen und bringt den Demonstrator zum Stehen.

Hupe

Im Rahmen des Konzeptionsteils war für das grafische Benutzerinterface eine Hupe geplant. Diese war im CAN-Demonstrator jedoch noch nicht vorgesehen und wurde daher softwareseitig implementiert und auch physisch eingebaut. Für die Umsetzung wurde zuerst der in Abbildung 4.6 gezeigte Programmablaufplan für den ESP32 erstellt. Hierbei erhält der Mikrocontroller vom BUSMASTER eine CAN-Nachricht und prüft, ob diese die für die Hupe definierte ID enthält. Ist dem nicht so, wird die nächste Nachricht ausgewertet. Entspricht die ID derer der Hupe, prüft der ESP, ob das Signal „an“ oder „aus“ lautet. Soll die Hupe ausgeschaltet werden, wird der Timer deaktiviert. Anderenfalls, wenn gehupt werden soll, wird dieser gestartet. Dieser Timer induziert, wenn er abgelaufen ist, die Interrupt Service Routine (ISR), welche den Programmablauf unterbricht und den Schaltzustand des PINs, der als Stromzufuhr dient sowie die Variable für den Zustand des Timers ändert.

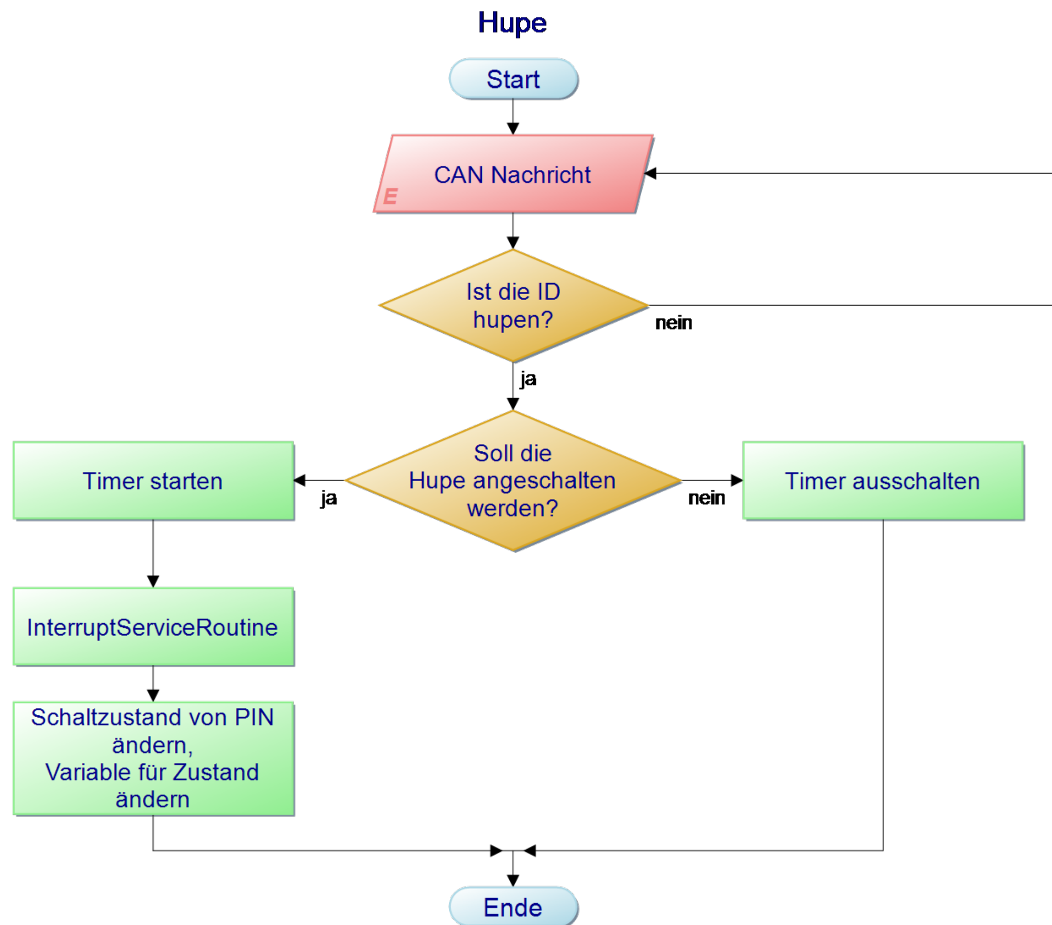


Abbildung 4.6: Skizze über den Ablauf des Programms des ESP für die Hupe. Quelle: [LI19a]

Solange gehupt werden soll, läuft der Timer in einer Schleife ab und löst jedes Mal die entsprechende ISR aus. Durch den ständigen Zustandswechsel der Stromversorgung am PIN erzeugt der Magnet im Lautsprecher eine Schwingung an der Membran, welche für das menschliche Ohr als Ton wahrnehmbar ist. Die Funktion wurde im Anschluss über die entsprechende IDE (Atom-Editor [ATO19] mit PlatformIO-Package) in den Quellcode des ESP32-EVB eingepflegt und über Micro-USB geflasht. Im nächsten Schritt wurde die Pinbelegung des ESP32-EVB aus dem Datenblatt des Herstellers [ESP19] ausgelesen und die für das Projekt relevanten PINs, die 5 V Spannungsquelle, die Masse (GND) und ein unbelegter PIN für den Datenstrom herausgelesen. Für das erste Testen wurde ein Lautsprecher auf einem Steckboard mit den entsprechenden PINs verkabelt und verlötet. Dieser hat jedoch Rückkopplungen vom BUSMASTER ausgelöst und daher nicht wie gewünscht funktioniert. Bei der finalen Hupe für den CAN-Demonstrator wurde ein 2-PIN Lautsprecher-Stecker verwendet. Hierfür wurden an den PINs GND und Datenstrom je ein Kabel mit Verbindung zum ESP32-EVB angelötet.

Im letzten Schritt wurde die Nachricht „Hupen“, mit der ID 0xC und der Länge 1 in der Datenbank im BUSMASTER definiert und nach erfolgreicher Verbindung mit dem Demonstrator-Auto an dieses gesandt. Hierbei wurde zum Aktivieren der Hupe eine 1 im Datenbyte 0 übertragen, zum Deaktivieren eine 0. Für die Bedienbarkeit wurde die grafische Oberfläche um einen Button „Hupe“ (8) erweitert und eine entsprechende Funktion programmiert.

5 Systemtest und Inbetriebnahme

In diesem Kapitel wird das grafische Benutzerinterface für den BUSMASTER in Betrieb genommen. Dabei werden die Funktionen mit Hilfe des CAN-Demonstrators getestet. Für den Zugriff auf das Controller Area Network wurde das Vector-Interface VN1610 verwendet.

5.1 Voraussetzungen

In dieser Arbeit wurde ein konfigurierbares, grafisches Benutzerinterface für die CAN-Analysesoftware BUSMASTER entwickelt. Zur erfolgreichen Anwendung der Desktop-Applikation, müssen im Vorfeld die dafür nötigen Komponenten installiert werden. Eine Voraussetzung stellt die Software BUSMASTER mit dem zugehörigen Compiler dar. Des Weiteren wird eine CAN-Schnittstelle zur Kommunikation mit der Hardware und ein entsprechendes Steuergerät, welches das CAN-Protokoll beherrscht, benötigt.

Im Folgenden sind die im Rahmen dieser Arbeit verwendeten Komponenten genannt:

- CAN-Analysesoftware
  BUSMASTER Version 3.2.2 mit Compiler MinGW
- CAN-Interface
  Vector VN1610
- Steuergerät
  CAN-Demonstrator

5.2 Inbetriebnahme der Kommunikation mit dem BUSMASTER

Zunächst soll die Verbindung zur CAN-Analysesoftware BUSMASTER hergestellt und die Kommunikation getestet werden. Hierfür müssen im BUSMASTER zuerst Voreinstellungen vorgenommen und im Anschluss eine Konfigurationsdatei sowie eine Datenbank erstellt werden. Um die Funktionalität zu testen wird die Verbindung über die GUI hergestellt und durch die Bedienung dieser CAN-Nachrichten an den BUSMASTER gesandt.
In diesem Versuch sind noch keine erkennbaren Änderungen in der GUI ersichtlich, da entsprechende Funktionen erst durch die Statusmeldung des Demonstrators ausgelöst werden.

5.2.1 Versuchsaufbau

Der theoretische Aufbau des Versuchs ist dem Blockschaltbild in Abbildung 5.1 zu entnehmen. Für diesen Versuch wurde die entwickelte grafische Bedienoberfläche mit der Analysesoftware über die virtuelle Schnittstelle verbunden.

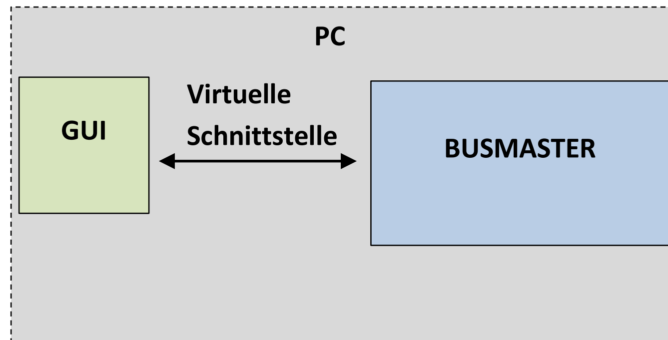


Abbildung 5.1: Blockschaltbild Versuchsaufbau der Kommunikation mit dem BUSMASTER. Quelle: [LI19a]

5.2.2 Versuchsdurchführung

Die Versuchsdurchführung besteht im Wesentlichen aus den folgenden vier Schritten:

- Konfigurationseinstellungen im BUSMASTER
- Erstellung einer Datenbank
- Verbindungsaufbau
- Senden von CAN-Nachrichten

Konfigurationseinstellungen im BUSMASTER

Wird das erstellte Programm ausgeführt, öffnet sich neben der GUI automatisch das Fenster der BUSMASTER-Software (siehe Abb. 5.2).

Zur erfolgreichen Verbindungsherstellung zwischen dem Benutzerinterface und der CAN-Analysesoftware müssen zu Beginn die im Folgenden beschriebenen Voreinstellungen im BUMASTER vorgenommen werden.
Über das rot markierte BUSMASTER-Symbol (1) muss zuerst eine neue Konfigurationsdatei angelegt und gespeichert werden. Im markierten Bereich 2, der Driver Selection, werden die im Folgenden benötigten Voreinstellungen zur Treiberauswahl festgelegt, in diesem Fall ist die Simulation auszuwählen.

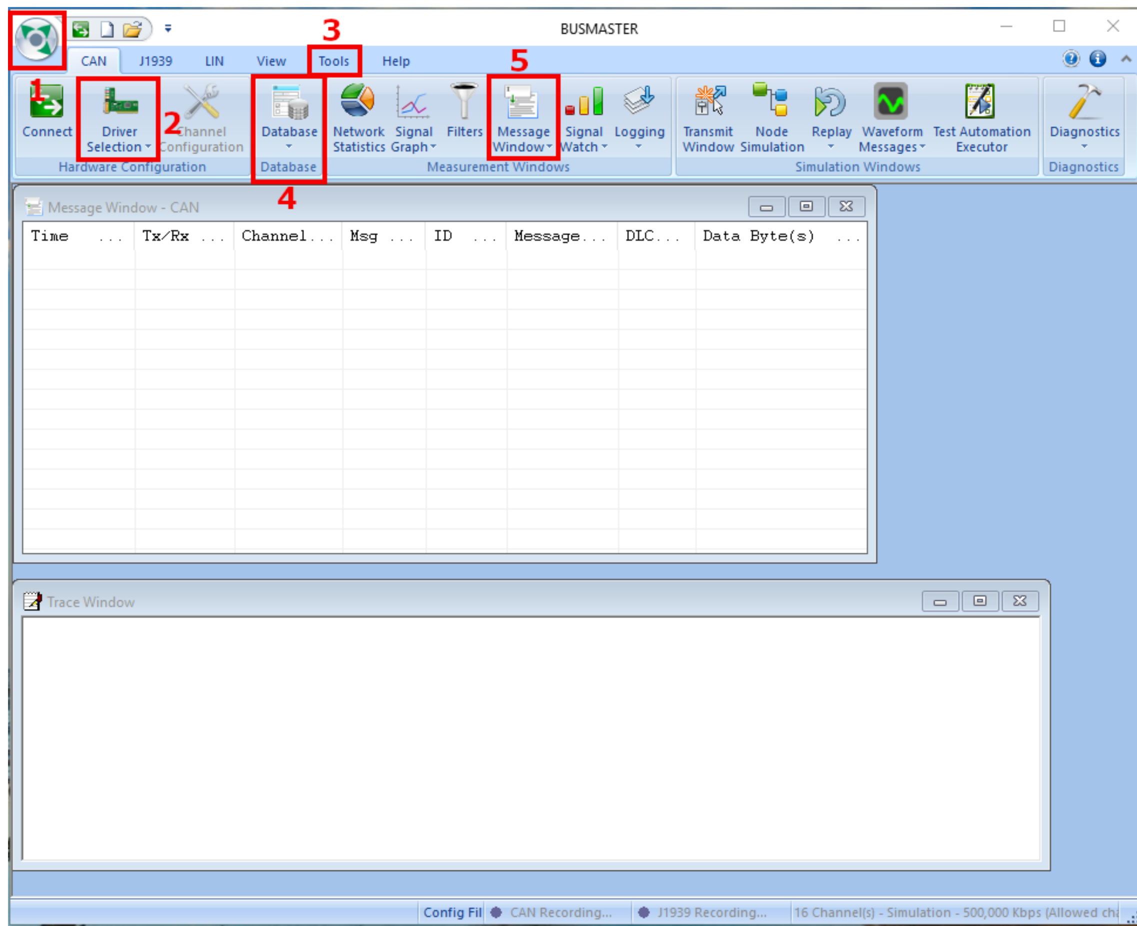


Abbildung 5.2: BUSMASTER-Software. Quelle: [LI19a]

Erstellung einer Datenbank

Anschließend muss zum Aussenden von CAN-Nachrichten eine neue Datenbank über den unter Tools (3) zur Verfügung stehenden CAN DBF Editor im BUSMASTER angelegt werden. Die Nachrichten (entsprechend Tabelle 5.1) benötigen einen Identifier und einen Namen. Über die Länge der Nachricht kann die maximale Anzahl der Bytes im Datenfeld, welche übertragen werden sollen, angegeben werden. Des Weiteren kann der Nachrichtentyp, Standard oder Extended, ausgewählt werden. Diese Nachrichten werden durch die Bedienung der GUI später mit Signalen zur Steuerung der einzelnen Funktionen befüllt, welche Aufschluss über Informationen zum Status der Beleuchtung oder Fahrtrichtung geben. Ein Signal kann dabei bis zu 8 Bit lang sein. Das Frame Format ist in diesem Versuch immer Standard.

Als nächstes muss die eben angelegte Datenbank über Database (4) dem Projekt hinzugefügt werden. Für einen zusätzlichen Überblick der über den BUSMASTER laufenden Nachrichten kann in den Einstellungen für das Nachrichtenfenster (5) die Overwrite-Funktion deaktiviert werden.

Tabelle 5.1: CAN-Nachrichten des Versuchsaufbaus.
Quelle: modifiziert nach [HAN18, S.30, S.47]

Ziel	ID	Nachricht	Länge
Motor	0x2	Stopp	0
	0x3	Vorwärts	2
	0x4	Rückwärts	2
Beleuchtung	0x5	Blinker links	1
	0x6	Blinker rechts	1
	0x7	Warnblinker	1
	0x8	Bremsleuchte	1
	0x9	Frontscheinwerfer	1
	0xA	Rückleuchte	1
	0xB	Rückscheinwerfer	1
Sonstiges	0x100	WiFi-Status	1
	0x10	Status	8
	0xC	Hupe	1

Verbindungsaufbau

In der GUI muss nun über den Menüpunkt Settings die Auswahl der angelegten Datenbank und der entsprechenden Konfigurationsdatei für das Projekt erfolgen. Anschließend kann die Verbindung über den Start-Button hergestellt werden.

Senden von CAN-Nachrichten

Sind die Voreinstellungen getroffen, kann die Kommunikation mit dem BUSMASTER beginnen. Für diesen Versuch wird der Button Licht betätigt, wodurch eine CAN-Nachricht zum Aktivieren der Front- und Rückscheinwerfer an die Analysesoftware gesandt wird. Im Message Window innerhalb des BUSMASTERs ist die übertragene Nachricht nun mit ihren Daten gelistet.

Die Trennung der Verbindung zwischen BUSMASTER und der GUI erfolgt über den Stop-Button. Über das Kreuz in der rechten oberen Ecke wird das Programm geschlossen.

5.2.3 Fazit

Nach der Anpassung der Funktionen an die Programmiersprache C# konnte eine erfolgreiche Umsetzung verzeichnet und damit der Gesamtaufwand minimiert werden. Mit dem entwickelten grafischen Benutzerinterface ist es möglich die CAN-Analysesoftware BUSMASTER über eine virtuelle Schnittstelle zu steuern und selbst generierte CAN-Frames an diese zu senden. In diesem Versuch konnte erfolgreich eine Nachricht zur Aktivierung der Scheinwerfer ausgesandt und vom BUSMASTER empfangen werden. Die Nachrichten konnten der Kontrollausgabe entnommen werden und waren in der Analysesoftware im Nachrichtenfenster ersichtlich. Außerhalb des Versuches wurden alle weiteren Funktionen in Betrieb genommen. Es konnten erfolgreich über die GUI Dateien in die BUSMASTER-Software importiert und die Verwaltung der Speicherung von Konfigurationen übernommen werden.

5.3 Inbetriebnahme der Kommunikation zwischen BUSMASTER und CAN-Demonstrator

In diesem Versuch soll die Kommunikation zwischen der CAN-Analysesoftware BUSMASTER und dem CAN-Demonstrator, durch die Steuerung in der GUI, getestet werden.

5.3.1 Versuchsaufbau

Der theoretische Aufbau des Versuchs ist dem Blockschaltbild in Abbildung 5.3 zu entnehmen. Für diesen Versuch wurde der CAN-Demonstrator als Hardware zur Ansteuerung durch CAN-Nachrichten verwendet.

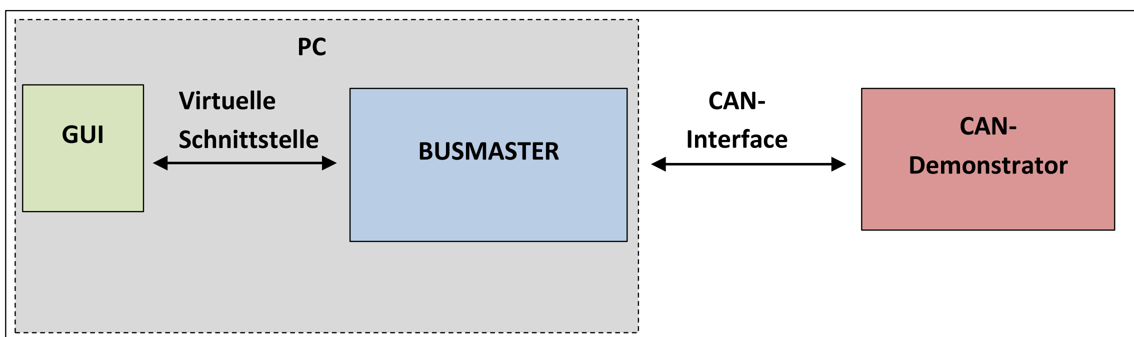


Abbildung 5.3: Blockschaltbild Versuchsaufbau der Kommunikation mit dem CAN-Demonstrator. Quelle: [LI19a]

5.3.2 Versuchsdurchführung

Für diesen Versuch kann die im vorherigen Kapitel erstellte Datenbank sowie die Konfiguration verwendet werden. Es muss jedoch nach der Auswahl der beiden Dateien folgende Einstellung für die Treiberauswahl im BUSMASTER vor der Verbindungsherstellung angepasst werden:

- Driver Selection: Vector XL
- CAN Hardware: VN1610 Channel 1
- Baudrate: 500000 bps

Nachfolgend muss das Interface zwischen dem Computer und dem Gateway des CAN-Demonstrators angeschlossen und dieser gestartet werden. Anschließend kann in der GUI die Verbindung über den „Start“-Button hergestellt werden.

Es wird nun kontinuierlich eine CAN-Nachricht mit der ID 0x100 vom Gateway ausgesandt, um über den Verbindungsstatus zum Server-Gateway auf dem CAN-Demonstrator zu informieren. Diese ist in der Kontrollausgabe einzusehen. Sobald im Datenbyte eine 1 übertragen wird, ist die Verbindung zum Auto hergestellt, die Leuchte neben dem Start/Stop-Knopf leuchtet grün und es können CAN-Nachrichten auf den Bus gesandt werden.

Eine Übersicht aller in Betrieb genommenen Komponenten ist Tabelle 5.2 zu entnehmen.
Im Folgenden wird die Funktionalität des Gas-Buttons näher beleuchtet.

Tabelle 5.2: In Betrieb genommene Komponenten

Ziel	**Funktion**
Motor	Gas
	Bremse
	Handbremse
	Rückwärtsgang
	Not-aus
Beleuchtung	Front- und Rückscheinwerfer
	Rückfahrleuchte
	Bremslicht
	Blinker
	Warnblinker
Sonstige	Menü, Speicherfunktion, Schließfunktion
	Start/Stop
	Hupe

Senden von CAN-Nachrichten

Zum Senden von Nachrichten auf den CAN-Bus des Demonstrators wird in diesem Versuch das simulierte Gaspedal zur Steuerung der Geschwindigkeit gedrückt. Die ausgesandte Nachricht enthält im Datenbyte die Informationen zur Geschwindigkeit und der Fahrtrichtung. Ob der Vorwärts- oder Rückwärtsgang eingelegt ist, wird anhand der ID erkannt. Die ausgesandte CAN-Nachricht ist im Message-Fenster des BUSMASTERs ersichtlich und die Motorsteuerung des Demonstrators beginnt die Räder vorwärts zu drehen.

Empfang von CAN-Nachrichten

Der CAN-Demonstrator sendet auf diese Änderung der Konfiguration eine Statusmeldung zur Information über den CAN-Bus. Diese Nachricht ist in der Kontrollausgabe der GUI ersichtlich. Entsprechend der Informationen in der Statusmeldung werden sowohl im Tacho als auch in der digitalen Geschwindigkeitsanzeige im Interface der Wert 5 als aktuelle Geschwindigkeit angezeigt.

5.3.3 Fazit

Die Verbindung und Kommunikation konnte zielführend mit dem CAN-Demonstrator hergestellt werden. Durch das Senden von CAN-Nachrichten auf den Bus konnten die Motor- sowie Beleuchtungsfunktionen des Autos gesteuert werden. Durch die erfolgreiche Auswertung der Statusnachricht konnten die CAN-Nachrichten in der GUI in Echtzeit visualisiert werden.
Beim Funktionstest der Blinker sind Komplikationen bei der Visualisierung im Falle des Blinkerwechsels aufgefallen. Ist eine Seite bereits aktiv und der komplementäre Blinker wird betätigt, erlöscht der bisherige nicht wie geplant.
Während weiterer Tests mit dem grafischen Benutzerinterface, ist ein Bug in der Motorsteuerung des Demonstrators festgestellt worden. Wird im Rückwärtsgang erst Gas gegeben, wieder auf 0 km/h abgebremst und in den Vorwärtsgang gewechselt, ohne zu beschleunigen, so fährt das Auto mit der Höchstgeschwindigkeit los. Der Tachometer innerhalb der GUI zeigt jedoch den erwarteten Wert für die Geschwindigkeit an.
Des Weiteren kam es bei der Ansteuerung des CAN-Demonstrators mehrfach zu unklaren Fehlschlägen beim Versuch des Verbindungsaufbaus.

Im folgenden Kapitel der Optimierung wird nach Gründen und Lösungsmöglichkeiten für die aufgetretenen Probleme gesucht.

5.4 Optimierungen

In diesem Kapitel werden die während der Inbetriebnahme und des Systemtests festgestellten Probleme analysiert und Möglichkeiten gesucht, um diese zu beseitigen.

Blinker

Zur Analyse des Problems bei der Visualisierung der Blinker bzw. eines Blinkerwechsels wurde zuerst der Quellcode des Benutzerinterfaces, der die Eingangsnachricht erzeugt, näher betrachtet und für fehlerfrei bewertet. Anschließend wurde der Programmcode der Motorsteuerung des Demonstrators in der Entwicklungsumgebung Atmel Studio [ATM18] untersucht. Dieser übergibt bei einem Blinkerwechsel keine 0 für die inaktive Blinkerseite und erstellt somit eine fehlerhafte Statusnachricht, welche ausgesendet wird. Eine entsprechende Korrektur der Programmierung wurde vorgenommen, indem bei der Aktivierung des einen Blinkers, der andere explizit deaktiviert wird. Das detektierte Problem konnte durch diesen Eingriff behoben werden.

Motorsteuerung

Bei der Problematik, dass der Demonstrator aus ungeklärten Gründen bei einem Richtungswechsel mit der Höchstgeschwindigkeit losfährt, wurde zuerst die Programmierung innerhalb des Interfaces überprüft. Hier wird jedoch korrekterweise eine CAN-Nachricht mit der Geschwindigkeit 0 übertragen. Bei der Kontrolle der Statusnachricht des Demonstrators, welche als Antwort eingeht, war ebenfalls eine richtig übertragene 0 zu verzeichnen. Beim Analysieren des Programmcodes der Motorsteuerung in Atmel Studio wurde ein Fehler bei der Bitverschiebung bei der Portansteuerung erkannt und entsprechend behoben.

Akkustandsüberwachung

Bei dem Versuch des Verbindungsaufbaus zum CAN-Demonstrator bzw. der Ansteuerung bei bestehender Verbindung waren fortlaufend Probleme zu verzeichnen. Die in der WiFi-Statusnachricht übergebene Information zur Verbindung hat in diesen Fällen immer 0 betragen oder es wurde gar keine Nachricht ausgesandt.

Bei der Untersuchung der Verbindung wurde zuerst jene über das Interface zum Gateway überprüft und festgestellt, dass das Gateway keine WiFi-Statusnachricht aussendet. Ein Reset am Client-Gateway (B) (siehe S. 17, Abb. 2.8) erzeugt einen neuen Verbindungsversuch, durch welchen die WiFi-Statusnachricht erfolgreich an den BUSMASTER gesandt wird und eine Verbindung zwischen Demonstrator und Analysesoftware hergestellt werden kann. Die Analyse der WiFi-Verbindung zwischen den beiden Gateways hat ergeben, dass diese sich aus unbekannten Gründen nicht immer miteinander verbinden, obwohl der Demonstrator eingeschalten ist. Zur weiteren Untersuchung der unklaren Verbindungsumstände wurde mit einem Spannungsmessgerät die, an der Batterie anliegende, Spannung gemessen und festgestellt, dass diese sich nahezu vollkommen entladen hat und die Stromversorgung nicht mehr für den verbauten Mikrocontroller genügt.

Zur Beseitigung des Problems, dass der CAN-Demonstrator bei zu niedrigem Akkustand abrupt nicht mehr reagiert, soll eine Überwachung von diesem implementiert werden. Um eine Möglichkeit der Akkustandsüberwachung der Hardware zu realisieren, soll der Demonstrator bei einer vordefinierten Entladung ein Signal in Form eines Huplautes von sich geben. Für die Projektplanung wurde zuerst der in Abbildung 5.4 ersichtliche Programmablaufplan erstellt. Um die anliegende Spannung über den Mikrocontroller messen zu können, wurde am ESP32-EVB nach einem PIN gesucht, welcher als Analog-zu-Digital-Converter (ADC) fungieren kann. Dieser Wandler soll kontinuierlich die Spannung messen und deren Wert ausgeben. Wird ein bestimmter Schwellwert erreicht, soll eine Meldung gesandt werden, die den Demonstrator zu hupen beginnen lässt.

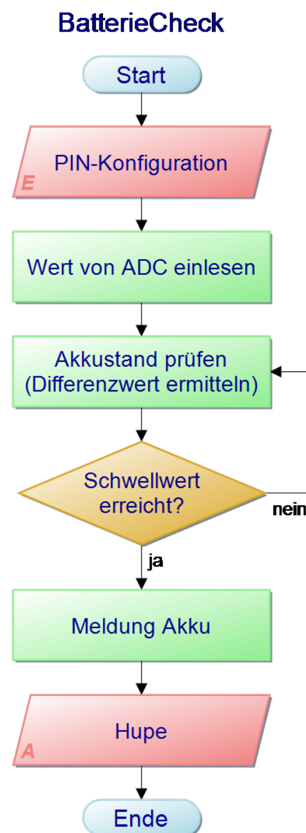


Abbildung 5.4: Skizze über den Ablauf des Programms des ESP für die Akkustandsüberwachung. Quelle: [LI19a]

Für die Auswahl eines geeigneten ADC-PINs wurde das Datenblatt des ESP32 [ESP19] sowie dessen Stromlaufplan [OLI18] verwendet. Da der Mikrocontroller über keinen freien PIN mit Analog-zu-Digital-Wandler verfügt, wurde das an PIN39 anliegende Bauteil abgelötet.

Da die maximale Spannung der Batterie 8,4 V beträgt, der ESP jedoch nur eine Betriebsspannung von 3,3 V hat, wurde ein Spannungsteiler (siehe Abbildung 5.5) gefertigt. Das Verhältnis zwischen R_1 und R_2 ist dabei so gewählt, dass über R_2 höchstens die für den ESP verarbeitbaren 3,3 V abfallen können. Der entsprechende Aufbau ist dem Schaltkreis des Spannungsteilers in Abbildung 5.6 zu entnehmen.

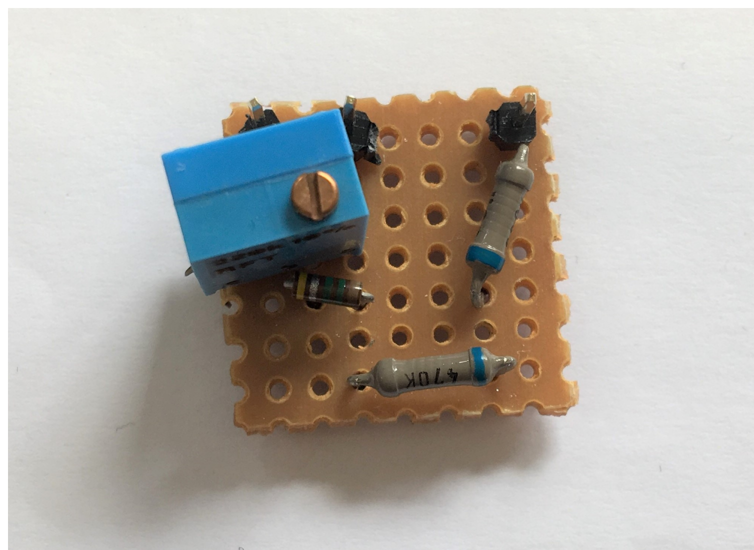


Abbildung 5.5: Spannungsteiler. Quelle: [LI19a]

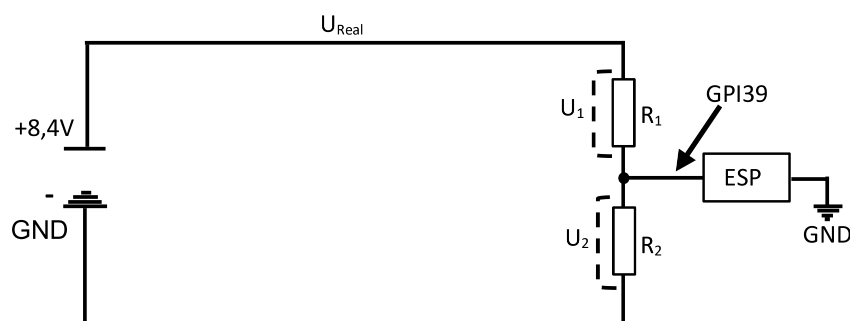


Abbildung 5.6: Skizze des Schaltkreises des Spannungsteilers. Quelle: [LI19a]

Für die Berechnung der einzelnen Komponenten wurde wie folgt vorgegangen:
Anhand der Maschengleichung (5.1) wurde zuerst die Spannung im Netzwerk berechnet. Hierfür wurde die Gleichung nach U_1 umgestellt (5.2), da sich der Spannungsabfall U_2 aus der Differenz der 3,3 V und GND ergibt und U_{Real} die bereits bekannte, an der Batterie anliegende, Spannung ist. Aus dem Datenblatt des ESP32-EVB [OLI18] wurde die maximale Stromstärke I, welche 3,3 μA beträgt, entnommen. Die Widerstände R_1, R_2 sowie R_{Ges}, über welchen die Spannungen U_1 und U_2 abfallen und den Spannungsteiler ergeben, lassen sich mit der Maschengleichung und der gegebenen Stromstärke bestimmen (5.3 - 5.5).

$$0 = U_{Real} - U_1 - U_2 \tag{5.1}$$

$$U_1 = U_{Real} - U_2 = 8,4\ V - 3,3\ V = 5,1\ V \tag{5.2}$$

$$R_1 = \frac{U_1}{I} = \frac{5,1\ V}{0,0000033\ A} = 1,55\ M\Omega \tag{5.3}$$

$$R_2 = \frac{U_2}{I} = \frac{3,3\ V}{0,0000033\ A} = 1\ M\Omega \tag{5.4}$$

$$R_{Ges} = R_1 + R_2 = 1,55\ M\Omega + 1\ M\Omega = 2,55\ M\Omega \tag{5.5}$$

Nach erfolgreicher Implementierung der Hardwarekomponente wurde über den Atom-Editor im Programmcode des ESP32-EVB-Servers eine Funktion erstellt, welche am definierten ADC-PIN 39 kontinuierlich den Sensorwert misst und diesen in Millivolt umrechnet.
Die Berechnung des Spannungswertes U erfolgte nach untenstehender Formel (5.6) durch das Produkt aus Sensorwert des ADC (S_{ADC}), Vergleichsspannung des ADC (U_2) (entspricht der Betriebsspannung des ESP in den obigen Formeln) und dem Gesamtwiderstand (R_{Ges}) geteilt durch das Produkt des höchsten digitalen Wertes für einen 12-Bit-ADC (S_{ADCmax}) und dem Teilwiderstand (R_2).

$$U = \frac{S_{ADC} \cdot U_2 \cdot R_{Ges}}{S_{ADCmax} \cdot R_2} \tag{5.6}$$

Nachfolgend wurde eine CAN-Nachricht in die Datenbank des BUSMASTERs eingepflegt, auf welche der ESP-Server mit dem aktuellen Akkustand antworten soll. Zur entsprechenden Visualisierung wurde die grafische Oberfläche um eine Akkustandsanzeige (siehe Abbildung 5.7) erweitert und das Programm so konfiguriert, dass dieses automatisch innerhalb eines bestimmten Zeitintervalls eine Nachricht zur Akkustandsabfrage über den BUSMASTER an den CAN-Demonstrator sendet. Für eine genaue Analyse ist es möglich, neben dem Spannungswert auch den reinen ADC-Wert in der Anzeige ausgeben zu lassen. Unterschreitet der Akkustand des Demonstrators einen festgelegten Schwellwert, so beginnt dieser zu hupen. Somit kann im kritischen Fall auch ohne das grafische Userinterface über die Entladung informiert werden. Die finale Version des grafischen Benutzerinterfaces ist Anhang A zu entnehmen.

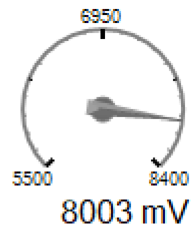


Abbildung 5.7: Akkustandsanzeige in der GUI. Quelle: [LI19a]

Bei der Durchführung von Kontrollmessungen mit einem Spannungsmessgerät wurden Ungenauigkeiten des ADC gegenüber diesem festgestellt. So wurden bei Ladeschlussspannung direkt an der Batterie 8,36 V gemessen, der ADC gibt jedoch einen Wert von etwa 3780 zurück, was umgerechnet nur 8,09 V entspricht.

Aufgrund der, auf Herstellerseite bekannten, Rauschempfindlichkeit des ESP32 ADC, kann es zu großen Abweichungen der Messwerte kommen[2]. Um dieses Rauschen zu kompensieren, sollte die Berechnung des Spannungswertes innerhalb der Programmierung des ADC angepasst werden, sodass der Ausgabewert in der GUI mit der Real-Spannung übereinstimmt.

Hierfür wurde der Demonstrator vom Akku getrennt und mit einem Labornetzteil eine fest definierte Spannung in den Demonstrator gegeben. Mit einem parallelgeschalteten Spannungsmessgerät wurde der tatsächliche Spannungswert am Auto gemessen. Für diesen Versuch wurde in Abständen von jeweils 0,1 V im Bereich von 5,5 bis 8,4 V die entsprechende Spannung am Labornetzteil eingestellt. Eine Skizze des entsprechenden Schaltkreises für die Spannungsmessung ist in Abbildung 5.8 ersichtlich.

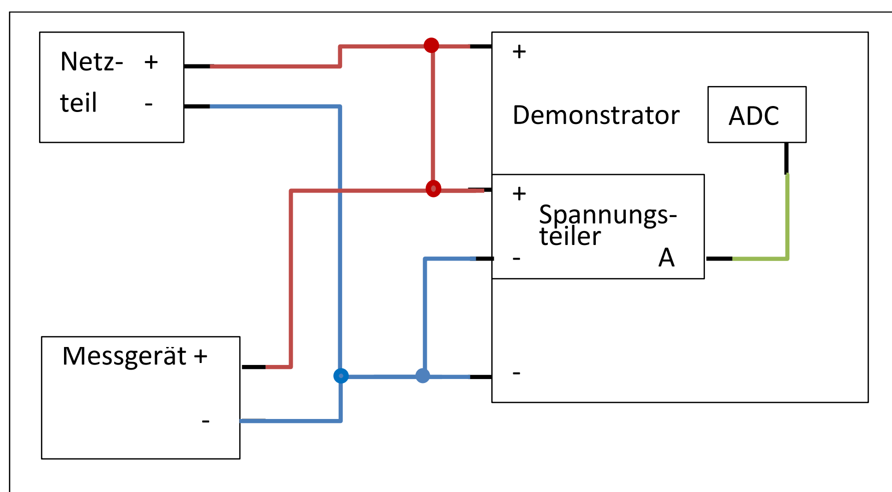


Abbildung 5.8: Skizze des Schaltkreises der Spannungsmessung am CAN-Demonstrator. Quelle: [LI19a]

[2] https://docs.espressif.com/projects/esp-idf/en/latest/api-reference/peripherals/adc.html [Eingesehen am: 17.8.2019]

Parallel wurde der ADC-Wert durch das grafische Benutzerinterface ermittelt und abgelesen. Für jeden Spannungswert wurde hierbei dreimal der ADC-Wert abgefragt und daraus ein Mittelwert gebildet. Im Anschluss wurde für jeden Spannungswert auch die tatsächliche Spannung am ADC-PIN des ESP32 mit einem Messgerät gemessen. Der entsprechende Versuchsaufbau ist der Skizze in Abbildung 5.9 zu entnehmen.

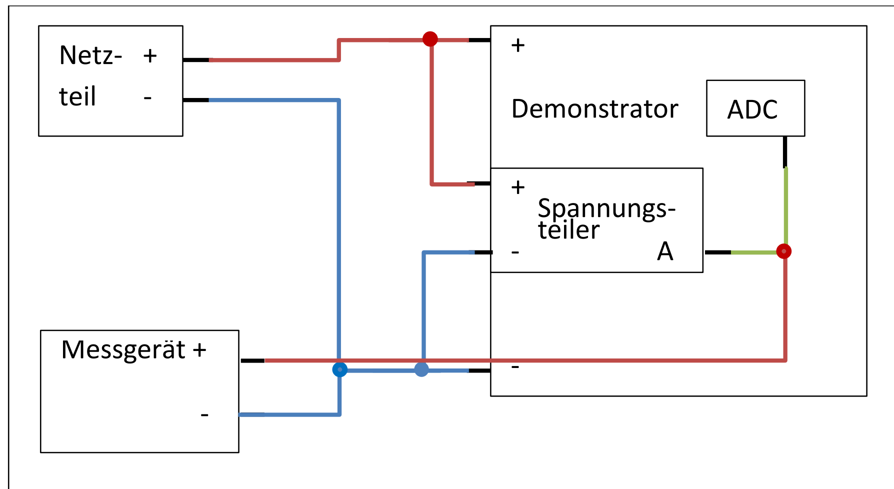


Abbildung 5.9: Skizze des Schaltkreises der Spannungsmessung am ADC. Quelle: [LI19a]

Die gemessenen Werte wurden anschließend in eine Excel-Tabelle aufgenommen, welche Anhang B zu entnehmen ist. Anhand dieser Messwerte wurde der optimale ADC-Wert für die beiden Grenzen von 6 V und 8,4 V bestimmt. Aus diesen wurde im Anschluss das in Abbildung 5.10 ersichtliche Diagramm erstellt, welches das Verhältnis zwischen den ADC-Zählwerten und dem Referenzwert des Messgerätes in Millivolt sowie die Ungenauigkeit des Wandlers darstellt. Für die Referenzwerte der Spannung von 6000 mV und 8400 mV ergaben sich folgende beiden ADC-Werte: 2473 und 3921.
Aus der Trendlinie ergab sich folgende Formel (5.7), welche in den Quellcode des ESP32 an die Stelle der Berechnung der Spannung aus dem Sensorwert des ADC eingepflegt wurde. Dieser berechnet sich nun aus dem Produkt des aktuellen Sensorwertes (U_2) und der errechneten Formel. Mit der vorgenommenen Optimierung konnte das gewünschte Ergebnis erzielt werden.

$$y = 1,6575x + 1901,1 \tag{5.7}$$

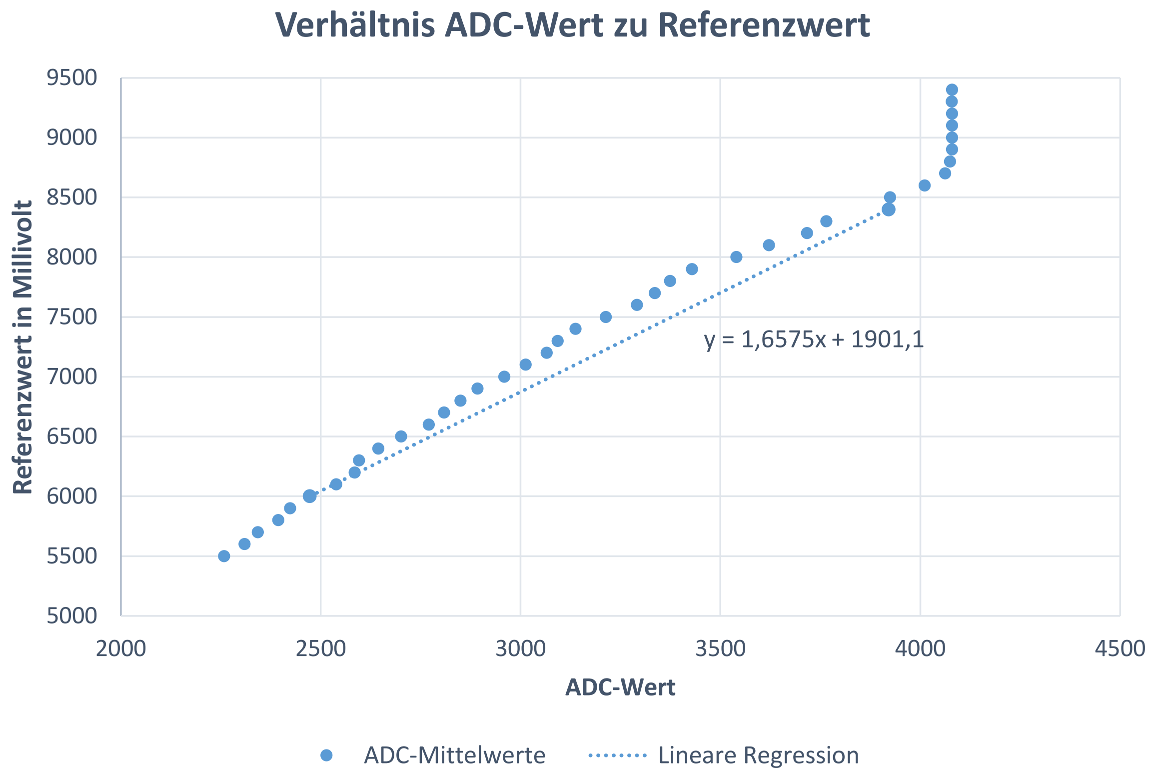


Abbildung 5.10: Verhältnis ADC-Wert zu Referenzwert. Quelle: [LI19a]

6 Zusammenfassung

Ziel der vorliegenden Bachelorarbeit war die Entwicklung eines grafischen Benutzerinterface für die CAN-Analysesoftware BUSMASTER, welches mittels entsprechender Elemente die Steuerung der Software sowie die Visualisierung der CAN-Nachrichten ermöglicht. Durch eine beliebige Konfigurierbarkeit sollte es zudem möglich sein, das Interface an die individuellen Bedürfnisse anzupassen.

Im Rahmen der Konzeptionsphase wurde sich für die Entwicklungsumgebung Visual Studio und die zugehörige Programmiersprache C# entschieden.
Um eine Kommunikation mit der Analysesoftware realisieren zu können, ist die Einrichtung einer virtuellen Schnittstelle essentiell. Die Implementierung eines CAN-Frames und entsprechender CAN-Nachrichten ermöglichen das Ansteuern der BUSMASTER-Software. Über die zugehörige Client-Funktion wird die GUI als Client registriert und das Empfangen von CAN-Botschaften vom BUSMASTER, und somit eine bidirektionale Kommunikation, realisiert. Für die Auswertung der Statusnachricht des CAN-Demonstrators wurde eine Auslösung entsprechender Reaktionen programmiert, welche die Informationen der CAN-Nachricht in der GUI visualisiert. Eine Menüleiste ermöglicht das Auswählen und Hinzufügen der Konfiguration sowie der Datenbankdatei in den BUSMASTER. Die Verbindung zu dieser lässt sich anschließend über einen Start/Stop-Button herstellen und trennen. Zur Nachrichtenanalyse dient eine Textbox innerhalb der grafischen Oberfläche.
Das Hinzufügen von Dateien und die Möglichkeit, die GUI jederzeit durch zusätzliche Implementierungen innerhalb des Quellcodes entsprechend anzupassen, sorgen für eine beliebige Konfigurierbarkeit des Interfaces.
Die Inbetriebnahme des grafischen Benutzerinterfaces sowohl mit dem BUSMASTER als auch mit dem CAN-Demonstrator waren erfolgreich. Aufgetretene Probleme bei der Visualisierung der Blinker und in der Motorsteuerung konnten behoben werden. Eine zusätzliche Implementierung einer Akkustandsüberwachung am CAN-Demonstrator verhindert künftig ein abruptes Abbrechen der Verbindung zum BUSMASTER auf Grund von zu niedrigem Akkustand.
Somit konnten alle gestellten Anforderungen, die sich aus der Zielstellung ergaben, erfüllt und sowohl die CAN-Analysesoftware BUSMASTER als auch der CAN-Demonstrator erfolgreich durch das entwickelte grafische Userinterface gesteuert werden.

7 Ausblick

Das erarbeitete Konzept stellt eine gute Basis für zukünftige Erweiterungen und Forschungen dar.

Im Hinblick auf die Weiterentwicklung könnte eine Lenkung innerhalb der grafischen Oberfläche implementiert werden, sodass die Simulation der Fahrtrichtung nicht in Abhängigkeit der Blinker steht. Denkbar wäre hier eine Art Lenkrad oder ein Joy-Stick, welcher über die Maus gesteuert werden kann und sich entsprechend der ausgewerteten Statusnachricht bewegt.

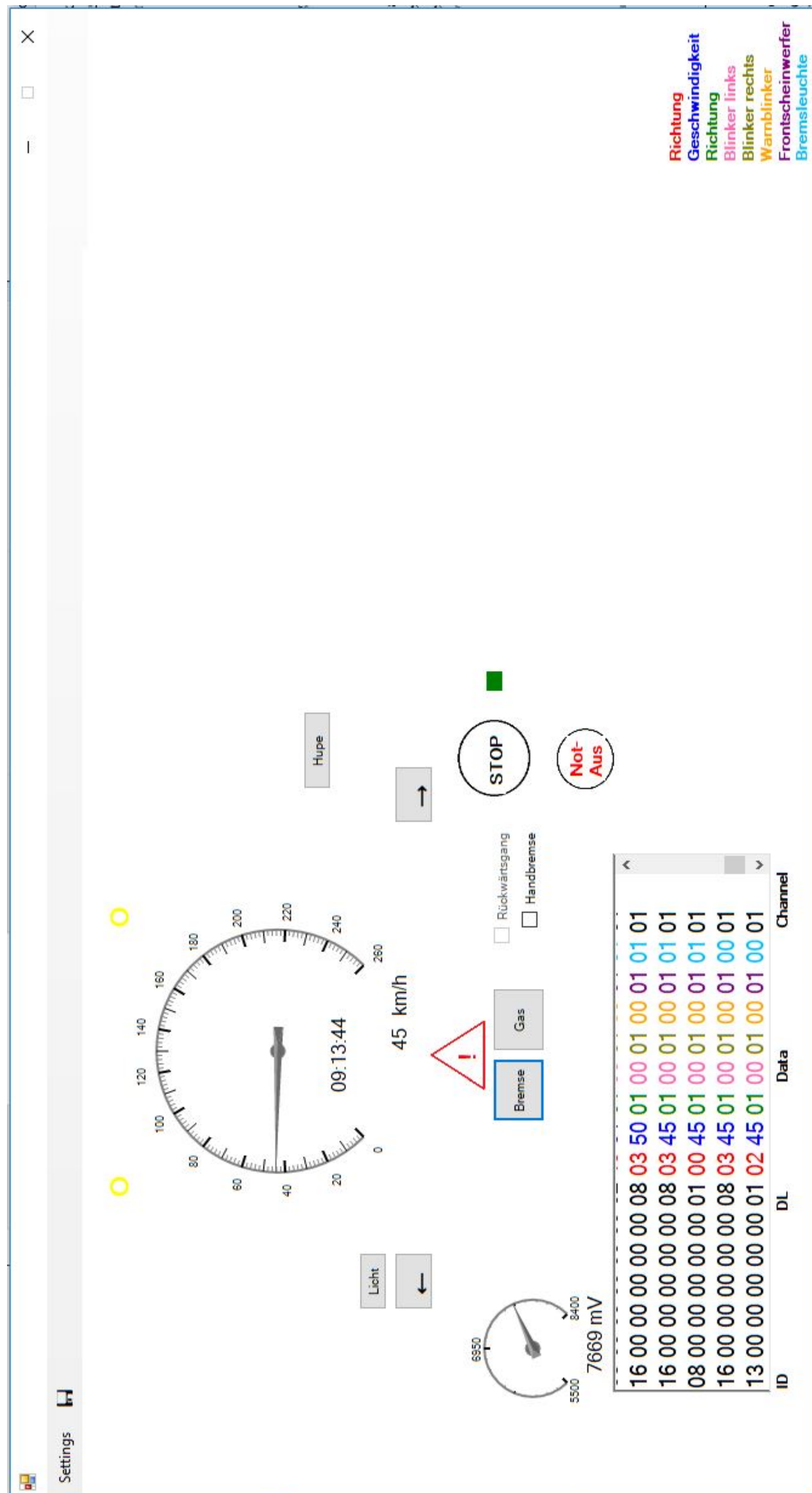
Um auch mit CAN-Nachrichten des Extended-Formats arbeiten zu können, wäre eine Erweiterung und entsprechende Anpassung im grafischen Benutzerinterface vorstellbar. Für einen besseren Überblick könnte die Anzeige der Identifier in der Kontrollausgabe in Hexadezimalwerten dargestellt werden.

Des Weiteren wäre eine Speicherfunktion der Einstellungen in der GUI möglich, um nicht nur innerhalb der hinzugefügten Dateien Änderungen beizubehalten, sondern auch, welche der Konfigurations- und Datenbankdateien zuletzt verwendet wurden.

Derzeit findet die Berechnung der anliegenden Spannung in der Programmierung des Demonstrators statt und erfolgt anhand festgelegter Referenzwerte, welche jedoch bei jeder Messung unterschiedlich sind. Für den Erhalt noch genauerer Ergebnisse bei der Akkustandsüberwachung könnte eine Kalibrierung des ADCs innerhalb des grafischen Interfaces implementiert werden. So ließe sich die Akkustandanzeige sowohl auf den Demonstrator als auch auf die Sensorwerte des Analog-zu-Digital-Wandlers anpassen und parallel die Komponente der Konfigurierbarkeit weiter ausbauen.

Aus forensischer Sicht betrachtet sind die Sicherheitslücken im Automobil schwerwiegend und bieten die Möglichkeit das System anzugreifen. Der für diese Arbeit verwendete CAN-Demonstrator warnt im Gegensatz zu einem realen Auto vor einem solchen Versuch des Angriffs. Zur weiteren Forschung auf diesem Gebiet wäre es daher denkbar, vom grafischen Benutzerinterface aus entsprechend kompromittierte Nachrichten an den CAN-Demonstrator zu senden und damit das Sicherheitssystem im Automobil weiter zu testen sowie Möglichkeiten zur Stabilisation von diesem zu entwickeln.

Anhang A: Finale Version des grafischen Userinterface



Anhang B: Messwerte ADC

Spannung	Messgerät	ADC1	ADC2	ADC3	ADC Mittelwert	ADC Spannung
9,4	9,39	4080	4080	4080	4080	3,325
9,3	9,29	4080	4076	4080	4079	3,293
9,2	9,19	4080	4080	4080	4080	3,257
9,1	9,09	4080	4080	4080	4080	3,223
9	8,98	4080	4080	4080	4080	3,185
8,9	8,89	4080	4080	4080	4080	3,151
8,8	8,78	4080	4073	4070	4074	3,114
8,7	8,69	4076	4067	4044	4062	3,08
8,6	8,59	4032	4006	3996	4011	3,042
8,5	8,49	3948	3910	3916	3925	3,008
8,4	8,38	3913	3929	3920	3921	2,971
8,3	8,29	3811	3728	3756	3765	2,938
8,2	8,18	3756	3676	3718	3717	2,9
8,1	8,09	3651	3609	3606	3622	2,867
8	7,98	3555	3545	3520	3540	2,83
7,9	7,89	3456	3449	3382	3429	2,794
7,8	7,78	3372	3347	3404	3374	2,753
7,7	7,69	3328	3353	3328	3336	2,72
7,6	7,59	3324	3264	3286	3291	2,687
7,5	7,49	3219	3200	3222	3214	2,653
7,4	7,39	3120	3129	3164	3138	2,616
7,3	7,29	3100	3104	3075	3093	2,581
7,2	7,19	3091	3024	3081	3065	2,544
7,1	7,09	3030	2992	3017	3013	2,51
7	6,99	2947	2979	2953	2960	2,473
6,9	6,89	2902	2896	2880	2893	2,439
6,8	6,79	2857	2857	2835	2850	2,402
6,7	6,69	2796	2835	2796	2809	2,368
6,6	6,59	2790	2774	2748	2771	2,331
6,5	6,49	2688	2732	2684	2701	2,296
6,4	6,39	2643	2608	2681	2644	2,26
6,3	6,29	2572	2630	2585	2596	2,225
6,2	6,19	2553	2598	2604	2585	2,194
6,1	6,09	2540	2556	2521	2539	2,159
6	5,99	2480	2470	2473	2474	2,122
5,9	5,89	2419	2448	2403	2423	2,088
5,8	5,79	2403	2390	2390	2394	2,051
5,7	5,69	2345	2339	2345	2343	2,017
5,6	5,59	2278	2316	2336	2310	1,98
5,5	5,49	2233	2275	2268	2259	1,945

Literaturverzeichnis

[ATM18] MICROCHIP TECHNOLOGY: *Atmel Studio*. Version: 7. URL https://www.microchip.com/mplab/avr-support/atmel-studio-7. Zuletzt geprüft: 03.09.2019, 13:54 Uhr.

[EC19a] ECLIPSE FOUNDATION: *Eclipse documentation - Current Release: Eclipse IDE 2019-03*. Version: 2019. URL https://help.eclipse.org/2019-03/index.jsp. Zuletzt geprüft: 10.07.2019, 12:44 Uhr.

[EC19b] ECLIPSE FOUNDATION: *Windowbuilder*. Version: 2019. URL https://www.eclipse.org/windowbuilder/. Zuletzt geprüft: 10.07.2019, 12:45 Uhr.

[ESP19] ESPRESSIF SYSTEMS: *ESP32 Series Datasheet*. Version: V3.0 URL https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf. Zuletzt geprüft: 07.06.2019.

[ETA17] ETAS GMBH: *BUSMASTER*. Version: v3.2.2. URL https://rbei-etas.github.io/busmaster/. Zuletzt geprüft: 20.06.2019, 08:08 Uhr.

[ETS00] ETSCHBERGER, Konrad (Hrsg.): *Controller-Area-Network; Grundlagen, Protokolle, Bausteine, Anwendungen*. 2., völlig überarb. Aufl. München: Hanser, 2000. – ISBN 3–446–19431–2

[ATO19] GITHUB: *Atom Editor*. Version: 1.40.1. URL https://atom.io. Zuletzt geprüft: 06.09.2019, 07:48 Uhr.

[HAN18] HANFELD, Pia: *Realisierung eines forensischen Demonstrators für den CAN-Bus*, Hochschule Mittweida, Bachelorarbeit, 2018

[HBG14] HERING, Ekbert; BRESSLER, Klaus; GUTEKUNST, Jürgen: *Elektronik für Ingenieure und Naturwissenschaftler*. 6., vollst. aktualisierte und erw. Aufl. Berlin: Springer Vieweg, 2014 (Lehrbuch). – ISBN 978–3–642–05498–3

[HEG17] HEGELER, Anna; CHIP (Hrsg.): *Eclipse vs Netbeans: Java-IDEs im Vergleich*. Version: 2017. URL https://praxistipps.chip.de/eclipse-vs-netbeans-java-ides-im-vergleich_94682. Zuletzt geprüft: 10.07.2019, 14:02 Uhr.

[LAB99] LAWRENZ, Wolfhard (Hrsg.); BAGSCHIK, Peter (Hrsg.): *CAN Controller area network: Grundlagen und Praxis*. 2., vollst. bearb. Aufl. Heidelberg: Hüthig, 1997. ISBN 3–7785–2575–1

[LI19a] LIPKE, Sina V.: *Eigene Arbeit/Aufnahme. Selbst erstelltes Bild/Grafik.* 2019

[LI19b] LIPKE, Sina V.: *Evaluierung einer CAN-Analysesoftware für den Einsatz in der Lehre*, Hochschule Mittweida, Praktkikumsbericht, 2019

[MS19a] MICROSOFT: *Visual Studio*. Version: 2019. URL https://www.microsoft.com/de-de/techwiese/aktionen/visual-studio-kostenlos.aspx. Zuletzt geprüft: 10.07.2019, 16:05 Uhr.

[MS19b] MICROSOFT: *Willkommen in der Visual Studio-IDE*. Version: 2019. URL https://docs.microsoft.com/de-de/visualstudio/get-started/visual-studio-ide?view=vs-2019. Zuletzt geprüft: 10.07.2019, 13:36 Uhr.

[OLI18] OLIMEX: *ESP32-EVB_Rev_D*. URL https://github.com/OLIMEX/ESP32-EVB/blob/master/HARDWARE/REV-D/ESP32-EVB_Rev_D.pdf Zuletzt geprüft: 05.08.2019, 12:49 Uhr.

[PCW10] PC WELT: *NetBeans IDE*. Version: 2010. URL https://www.pcwelt.de/ratgeber/NetBeans-IDE-Web-Entwicklung-363490.html. Zuletzt geprüft: 05.08.2019, 12:11 Uhr.

[REI11] REIF, Konrad: *Bosch Autoelektrik und Autoelektronik: Bordnetze, Sensoren und elektronische Systeme*. 6., überarbeitete und erweiterte Auflage. Wiesbaden: Vieweg+Teubner Verlag / Springer Fachmedien Wiesbaden GmbH Wiesbaden, 2011 (Bosch Fachinformation Automobil). ISBN 978–3–8348–1274–2

[WIL12] WILLE, Christoph; IC#CODE (Hrsg.): *The Open Source Development Environment for .NET*. Version: 2012. URL http://www.icsharpcode.net/OpenSource/SD/Default.aspx. Zuletzt geprüft: 06.08.2019, 9:13 Uhr.

[ZIS14] ZIMMERMANN, Werner; SCHMIDGALL, Ralf: *Bussysteme in der Fahrzeugtechnik: Protokolle, Standards und Softwarearchitektur*. 5., aktual. und erw. Aufl. Wiesbaden: Springer Vieweg, 2014 (ATZ / MTZ-Fachbuch). ISBN 978–3–658–02418–5

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe. Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Mittweida, 26. September 2019