technische universität
dortmund

Bachelorarbeit

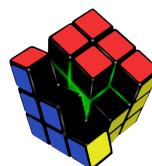
Aufbau und Dokumentation einer Experimentierplattform für automotive Softwareentwicklung

Daniel Noack

10. August 2010

Betreuer:
Prof. Dr.-Ing. Olaf Spinczyk
Dr. Michael Engel

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12
Arbeitsgruppe Eingebettete Systemsoftware
http://ess.cs.tu-dortmund.de



Zusammenfassung

Im Rahmen der am LS12 durchgeführten Projektgruppen „AutoLab“ [1] und „CoaCh - Car on a Chip“ [2, 3] sind Hard- und Software-Komponenten für automotive Steuergeräte - basierend auf Microcontroller- und FPGA-Lösungen - entstanden [4, 5]. In dieser Bachelorarbeit sollen nun die entstandenen Hardware-Komponenten zu einem Fahrzeugnetz kombiniert werden, das die Entwicklung und den Test von typischen Software-Systemen des automotiven Bereichs in einer verteilten Umgebung ermöglicht. Zudem sollen bisher nicht integrierte Komponenten, wie ein Lenksäulenmodul, analysiert und in das System integriert werden, sowie aufwendig wiederbeschaffbare Komponenten, wie das verwendete CAN-Gateway, ersetzt werden. Zwecks Reproduzierbarkeit soll der Aufbau zudem durch eine ausführliche Dokumentation beschrieben werden.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Aufgabenstellung . . . 1
1.2 Aufbau dieser Arbeit . . . 1

2 Bussysteme in heutigen Kraftfahrzeugen **3**
2.1 LIN-Bus . . . 3
2.2 CAN-Bus . . . 4
2.2.1 Allgemeine Informationen . . . 4
2.2.2 Die verschiedenen Varianten des CAN-Busses . . . 5
2.2.3 Aufbau der CAN-Nachrichten . . . 6
2.2.4 Kommunikation auf dem CAN-Bus . . . 9
2.3 FlexRay . . . 12

3 Konzept der Experimentierplattform **15**

4 Verwendete Hardware **19**
4.1 Scheinwerfer . . . 19
4.2 Lenksäulenmodul . . . 19
4.3 Dachmodul . . . 20
4.4 Gangwahlschalter . . . 20
4.5 TriBoard TC1796 . . . 21
4.6 Spartan-3E S500 . . . 22
4.7 Pedaleinheit . . . 22
4.8 DELL OPTIPLEX 755 . . . 23
4.9 USBprog & CAN-SPI-Adapter . . . 24
4.10 Netzteil . . . 24

5 Integration der Komponenten in die Experimentierplattform **25**
5.1 Spartan-3E S500 . . . 25
5.1.1 Konfiguration des System-on-Chip . . . 25
5.1.2 Anschluss externer CAN-Transceiver . . . 26
5.1.3 Aufbau des ausgeführten Programms . . . 27
5.1.4 Erzeugen einer dauerhaften Konfiguration für das FPGA . . . 28
5.2 Scheinwerfer . . . 30
5.2.1 Aufbau der Transistorkästchen . . . 31
5.2.2 Anschluss an das TC1796 . . . 32
5.2.3 Software zur Scheinwerfersteuerung . . . 32

5.3 Lenksäulenmodul 33
5.3.1 Inbetriebnahme des Lenkwinkelgebers 34
5.3.2 Inbetriebnahme des Schaltermoduls 35
5.4 Dachmodul 36
5.5 Gangwahlschalter 37
5.6 Pedaleinheit 38
5.6.1 Aufbau und Funktion der Pedaleinheit 38
5.6.2 Umbau der Pedaleinheit 40
5.6.3 Anschluss am TC1796 40
5.6.4 Software zur Auswertung der Pedaleinheit 41
5.7 USBprog & CAN-SPI-Adapter 43
5.8 AVM-Board mit VirtualMachine 44

6 Anpassung der Entwicklungsumgebung 45
6.1 Ubuntu 10.04 64-Bit 45
6.1.1 Grundlegende Installation 45
6.1.2 Zusätzlich benötigte Pakete 46
6.2 Rennsimulation TORCS 48
6.3 AutoLabIDE 48

7 Fazit 51

A Weitere Informationen 53
A.1 Kabelbelegungen 53
A.1.1 CAN-Bus 53
A.1.2 USBprog & CAN-SPI-Adapter 53
A.2 Fotos von den Elementen des Aufbaus 54
A.2.1 CAN-Gateway 54
A.2.2 Scheinwerferboard 55
A.2.3 Pedalboard 55
A.2.4 AVM-Board 56
A.2.5 Dachmodul 56
A.2.6 Gangwahlschalter 57
A.2.7 Lenksäulenmodul 57
A.3 Programmieren & Flashen der Komponenten 58
A.3.1 Steuergeräte mit CodeWarrior programmieren 58
A.3.2 Flashen eines TC1796 59
A.4 Spannungsversorgung der einzelnen Komponenten 61
A.5 CAN-Nachrichten der Experimentierplattform 62
A.5.1 Gangwahlschalter 62
A.5.2 Dachmodul 63
A.5.3 Pedaleinheit 64
A.5.4 Simulation 64
A.5.5 Scheinwerfer 66

A.5.6 Lenksäulenmodul 68
A.6 Softwarezuordnung 69

Literaturverzeichnis **72**

Abbildungsverzeichnis **74**

1 Einleitung

In diesem Kapitel wird zunächst die Aufgabenstellung dieser Bachelorarbeit genauer vorgestellt, bevor der Aufbau dieser schriftlichen Ausarbeitung beschrieben wird.

1.1 Aufgabenstellung

Die Aufgabenstellung für diese Bachelorarbeit ist der Aufbau und die Dokumentation einer Experimentierplattform für automotive Softwareentwicklung. Aus diesem Grund lässt sich die Arbeit in zwei wesentliche Teile gliedern.
Der erste Teil ist ein praktischer Teil, in dem die vorhandenen Hard- und Softwarekomponenten zu einem Fahrzeugnetz kombiniert werden sollen, in dem typische Software-Systeme des automotiven Bereichs entwickelt und getestet werden können. Für die noch fehlenden Komponenten des Fahrzeugnetzes müssen geeignete Ersatzlösungen gefunden werden.
Zu den fehlenden Komponenten zählt neben dem CAN-Gateway auch die Pedaleinheit. Eine weitere problematische Komponente dieser Bachelorarbeit ist das zur Verfügung gestellte Lenksäulenmodul aus einem Audi A8. Bei diesem Lenksäulenmodul handelt es sich um eine Komponente, die bisher in keinem Aufbau integriert war und deshalb bisher nicht dokumentiert ist. Aus diesem Grund muss das Lenksäulenmodul zunächst analysiert werden, bevor es in die Experimentierplattform integriert werden kann.
Der zweite Teil dieser Bachelorarbeit ist die hier vorliegende Dokumentation. Innerhalb dieser Dokumentation sollen alle Informationen enthalten sein, die notwendig sind, um den Aufbau der Experimentierplattform reproduzieren zu können. Es soll hierbei insbesondere auf die Details zu den bisher nicht verwendeten Komponenten eingegangen werden.
Das Ergebnis dieser Bachelorarbeit soll eine funktionsfähige Experimentierplattform sein, die vollständig dokumentiert ist.

1.2 Aufbau dieser Arbeit

Die vorliegende Bachelorarbeit ist in sieben Kapitel gegliedert. Innerhalb des zweiten Kapitels werden verschiedene Bussysteme des automotiven Bereichs vorgestellt. Der CAN-Bus wird hier im Gegensatz zu den anderen Bussen ausführlich vorgestellt, weil er innerhalb der Experimentierplattform verwendet wird.
Im dritten Kapitel wird das Konzept vorgestellt, welches dem Aufbau der Experimentierplattform zu Grunde gelegt wird. Hier wird eine Übersicht über den geplanten Aufbau

gegeben und auf seine Besonderheiten eingegangen.
Das sich anschließende Kapitel stellt zunächst die innerhalb dieser Bachelorarbeit verwendeten Komponenten vor. Hier wird ein grober Überblick über diese Komponenten gegeben und auf ein paar Eckdaten eingegangen.
Im nächsten Kapitel wird die Integration der einzelnen Komponenten in die Experimentierplattform beschrieben. Hier wird auf die jeweiligen Besonderheiten hingewiesen, die bei den einzelnen Komponenten zu beachten sind.
Das vorletzte Kapitel befasst sich mit den notwendigen Anpassungen an der Entwicklungsplattform. Hier wird beschrieben welche Probleme es innerhalb der Entwicklungsplattform gibt und wie diese gelöst werden können.
Das Fazit bildet das letzte Kapitel und fasst die Ergebnisse dieser Bachelorarbeit noch einmal zusammen.
Abgeschlossen wird die Arbeit mit dem Anhang, in dem weitergehende Informationen wie beispielsweise die Kabelbelegungen (Anhang A.1) oder die Programmieranleitungen (Anhang A.3) zu finden sind.

2 Bussysteme in heutigen Kraftfahrzeugen

Im Laufe der Zeit wurden viele verschiedene Bussysteme für den automotiven Bereich entwickelt. Da das Vorstellen jedes dieser Bussysteme innerhalb dieser Bachelorarbeit den Umfang dieser Arbeit überschreiten würde, wird nur auf die drei wichtigsten Bussysteme moderner Kraftfahrzeuge eingegangen. Neben einem kurzen Überblick über den LIN-Bus und FlexRay, wird es eine ausführliche Beschreibung des CAN-Busses geben, weil dieser ein wichtiger Bestandteil der vorliegenden Arbeit ist.
Auf reine Multimedia-Busse, wie beispielsweise MOST, wird an dieser Stelle bewusst nicht eingegangen.

2.1 LIN-Bus

Beim Local Interconnect Network (LIN) handelt es sich um ein Bussystem, das Ende der 90er Jahre des letzten Jahrhunderts vom LIN-Konsortium entwickelt wurde. Ziel dieses Konsortiums war es ein Bussystem zu entwickeln, das eine preisgünstige Vernetzung von mechatronischen Komponenten [4] erlaubt und dabei eine kostengünstige Alternative zu Low-Speed-Bussystemen bereitstellt [6].
In heutigen Kraftfahrzeugen wird der LIN-Bus überwiegend im Bereich der Tür-, Sitz- und Schiebedach-Elektronik eingesetzt, weil dort keine besonders hohen Ansprüche an die Zuverlässigkeit und Geschwindigkeit des Bussystems gestellt werden. Hier erfolgt in der Regel auch der Anschluss an andere Bussysteme, wie z. B. den CAN-Bus.
Die derzeit aktuelle LIN-Spezifikation ist die im November 2006 veröffentlichte Version 2.1 [7].

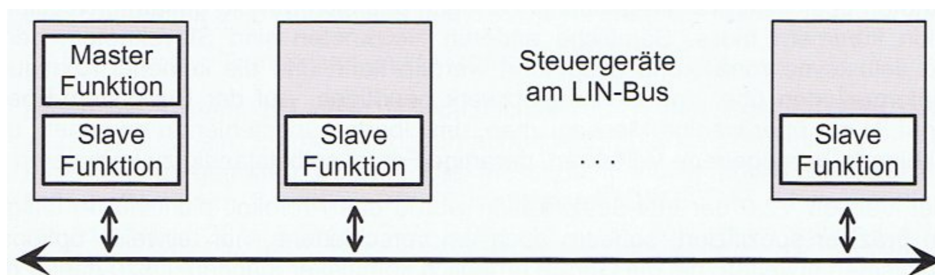


Abbildung 2.1: Struktur eines LIN-Bussystems (Quelle: [6] Seite 44)

Beim LIN-Bus handelt es sich, wie der Abbildung 2.1 zu entnehmen ist, um ein Single-Master-System, bei dem nur ein Master-Knoten im gesamten Bus existiert. Dieser Knoten steuert die Kommunikation mit den Slave-Knoten, von denen es maximal 16 Stück im Bus geben kann [8].
Die Kommunikation über den LIN-Bus erfolgt über ein UART-basiertes Übertragungsprotokoll mit einer maximalen Datenrate von 20 Kbit/s. Weil das LIN-Protokoll die Verwendung von Nachrichten-Identifiern (Nachrichten-IDs) zur Identifizierung der Busnachrichten vorsieht, handelt es sich um ein nachrichtenorientiertes Protokoll. Durch die Verwendung von Nachrichten-IDs ist eine Broad- und Multicast-Kommunikation sehr einfach zu realisieren. Die am Bus angeschlossenen Knoten empfangen alle dort gesendeten Nachrichten und werten die Nachrichten aus, die mit einer von ihnen zu bearbeitenden Nachrichten-ID versehen sind. Durch die Verwendung eines 6-Bit langen Feldes für die Nachrichten-ID können insgesamt 64 verschiedene Nachrichten unterschieden werden.
Der Zugriff auf den Bus erfolgt in einem Master-Slave-Verfahren. Hierbei sendet der Master-Knoten, abhängig von seiner internen Scheduling-Tabelle, sogenannte Header auf den Bus, die mit der Nachrichten-ID der sendeberechtigten Nachricht versehen sind. Alle Slave-Knoten empfangen diese Nachrichten und werten sie aus. Wenn ein Slave-Knoten Daten für die empfangene Nachrichten-ID zu übermitteln hat, sendet er diese Daten direkt im Anschluss an den empfangenen Header in Form einer Response. Ein kompletter Frame auf dem LIN-Bus besteht somit aus dem Header des Master-Knotens und der Response eines Slave-Knotens. Insgesamt können in der Response 1 bis 8 Byte Daten übertragen werden.
Der genaue Aufbau der Nachrichten und weitere Einzelheiten zum LIN-Bus, können der einschlägigen Literatur, wie beispielsweise [9] oder [6], entnommen werden.

2.2 CAN-Bus

In diesem Abschnitt wird der Controller Area Network Bus (CAN-Bus) vorgestellt. Dieser Bus bildet einen elementaren Bestandteil der Experimentierplattform, weshalb er an dieser Stelle ausführlicher als die anderen Bussysteme vorgestellt wird.

2.2.1 Allgemeine Informationen

Der CAN-Bus wurde in der zweiten Hälfte der 80er Jahre von der Robert Bosch GmbH entwickelt und liegt momentan in der seit 1991 gültigen CAN Spezifikation Version 2.0 [10] vor. Später wurde er von der ISO genormt und ist seit dem unter den Normen ISO11898-1 bis ISO11898-5 zu finden.
Beim CAN-Bus handelt es sich um das erste digitale Bussystem, das herstellerübergreifend in Kraftfahrzeugen eingesetzt wurde [11, 6]. Aufgrund der daraus resultierenden großen Verbreitung sind CAN-Controller sehr günstig in der Anschaffung, so dass der CAN-Bus auch in vielen industriellen Anwendungen eingesetzt wird.
Der CAN-Bus ist ein bitstrom-orientierter Linien-Bus, der eine maximale Bitrate von

1 Mbit/s ermöglicht. Die maximale Buslänge und die Länge der Stichleitungen zu den einzelnen Steuergeräten, sind dabei abhängig von der eingesetzten Bitrate. Innerhalb eines Busses wird nur eine Bitrate zugelassen, so dass alle an diesen Bus angeschlossenen Geräte mit derselben Bitrate arbeiten müssen. Sollen Geräte mit verschiedenen Bitraten eingesetzt werden, müssen diese Geräte an getrennten Bussen betrieben und die Busse über ein Gateway miteinander verbunden werden.

2.2.2 Die verschiedenen Varianten des CAN-Busses

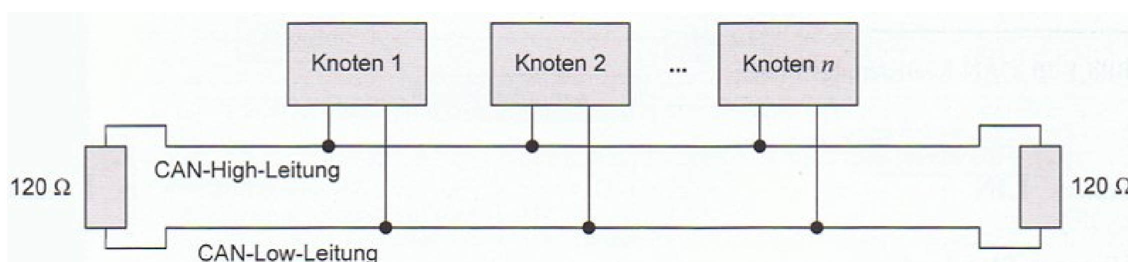
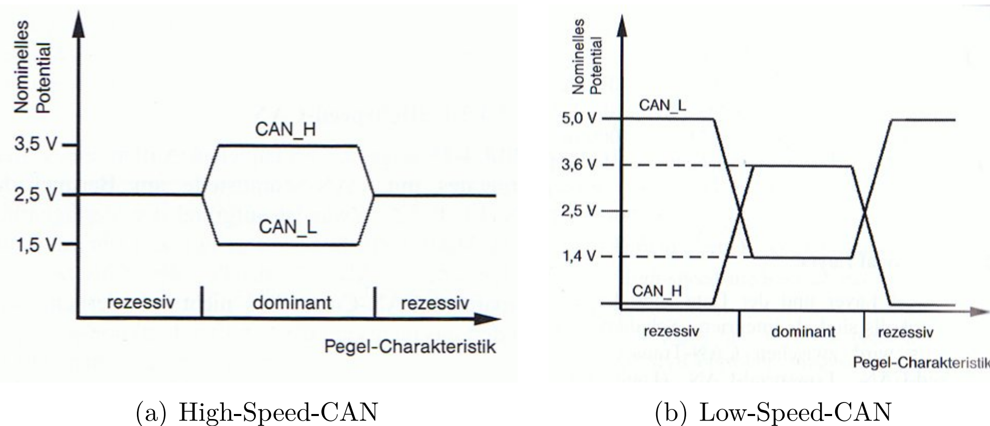


Abbildung 2.2: High-Speed-CAN-Bus (Quelle: [4] Seite 19)

Den CAN-Bus gibt es in zwei verschiedenen Varianten, die sich hinsichtlich ihrer physikalischen Eigenschaften unterscheiden. Die erste dieser Varianten ist der High-Speed-CAN, der für schnelle Übertragungen vorgesehen ist und im Kraftfahrzeug die Komponenten des Antriebsstrangs miteinander vernetzt. Typischerweise arbeitet dieser Bus mit einer Bitrate von 500 Kbit/s [6], die auch innerhalb der Experimentierplattform verwendet wird. Als Busmedium kommt eine verdrillte Zwei-Draht-Leitung zum Einsatz, auf der die versendeten Daten in Form von Differenzsignalen übertragen werden. Damit es durch die hohe Übertragungsgeschwindigkeit keine ungewollten Reflexionen an den Kabelenden gibt, werden diese an ihren Enden mit jeweils einem Abschlusswiderstand von 120Ω versehen. Der typische Aufbau eines High-Speed-CAN ist in Abbildung 2.2 zu sehen.
Die zweite Variante des CAN-Bus ist der Low-Speed-CAN, der für den Komfortbereich und die Karosserieelektronik eingesetzt wird und mit einer geringeren Bitrate arbeitet. Typischerweise liegt diese Bitrate bei 125 Kbit/s [6], wobei einige Hersteller auch mit 100 Kbit/s arbeiten. Innerhalb der Experimentierplattform wird mit 100 Kbit/s gearbeitet, weil eine der verwendeten Komponenten fest mit dieser Bitrate arbeitet. Als Medium kommt auch hier eine verdrillte Zwei-Draht-Leitung zum Einsatz, wobei die Abschlusswiderstände nicht benötigt werden. Die Datenübertragung erfolgt hier ebenfalls in Form von Differenzsignalen. Gegenüber dem High-Speed-CAN ist der Low-Speed-CAN jedoch in der Lage den Ausfall einer Leitung zu verkraften und mit nur einer Leitung zu arbeiten. Wenn dieser Fall eintritt, wird die verbliebene Leitung gegen Masse ausgewertet. So können die gesendeten Daten trotzdem ausgewertet werden.
Die Wahl der verdrillten Zwei-Draht-Leitung als Medium und des Differenzsignals als Übertragungsart gewährleisten eine gute Robustheit des Busses, weil sich Störsignale in

der Regel auf beide Adern gleich auswirken und durch die Betrachtung eines Differenzsignals kaum einen Einfluss auf die übertragenen Daten besitzen.



(a) High-Speed-CAN (b) Low-Speed-CAN

Abbildung 2.3: Signalpegel des CAN-Bus (Quelle: [8] Seite 206)

Die Daten werden auf dem Bus mit Hilfe von zwei verschiedenen Signalpegeln übertragen. Eine logische Eins entspricht dabei dem rezessiven Pegel und eine logische Null dem dominanten Pegel. Für den Bus bedeutet dies, dass eine auf ihm gesendete logische Null sich immer gegenüber einer logischen Eins durchsetzt. Weitere Informationen zu den rezessiven und dominanten Pegeln und ihrer Funktion sind im Abschnitt 2.2.4 zu finden.
Die verwendeten Signalpegel zur Datenübertragung unterscheiden sich je nach eingesetzter CAN-Variante. Wie Abbildung 2.3(a) zu entnehmen ist, führen die Leitungen beim High-Speed-CAN einen Pegel von 2,5V, wenn ein rezessives Signal anliegt. Das Differenzsignal liegt in diesem Fall bei 0V. Bei einem dominanten Signal führt die CAN-H-Leitung einen Pegel von 3,5V, wohingegen die CAN-L-Leitung einen Pegel von 1,5V führt [8]. Der Signalhub des Differenzsignals beträgt hier 2V.
Beim Low-Speed-CAN sind die Pegel etwas anders (Abbildung 2.3(b)). Im Fall eines rezessiven Signals führt die CAN-H-Leitung einen Pegel von 0V und die CAN-L-Leitung einen Pegel von 5V. Bei einem dominanten Signal führt die CAN-H-Leitung einen Pegel von 3,6V und die CAN-L-Leitung einen Pegel von 1,4V [8]. Das Differenzsignal kann hier bis zu 5V betragen.

2.2.3 Aufbau der CAN-Nachrichten

Der Austausch von Nachrichten auf dem CAN-Bus basiert, seit der CAN Spezifikation Version 2.0 [10], auf fünf verschiedenen Telegrammformaten (Frame-Formaten). Gegenüber vorherigen CAN-Spezifikationen, welche Remote-Frames, Fehler-Frames, Überlast-Frames und einen Daten-Frame kannten, gibt es in der aktuellen Version nun zwei verschiedene Formate des Daten-Frames.

Diese beiden Formate unterscheiden sich hinsichtlich der Länge ihres Nachrichten-Identifiers (Nachrichten-IDs). Beim Standard-Format wird ein 11 Bit langer Identifier verwendet, der es ermöglicht 2048 verschiedene Nachrichten zu kennzeichnen.
Durch die große Akzeptanz von CAN und dem Einsatz in vielen verschiedenen Bereichen ist es notwendig geworden, dass für einige Anwendungsbereiche eine feste Zuordnung von Nachrichten-IDs zu Funktionen erfolgt [10]. Damit dieses Vorgehen nicht zu Problemen mit der geringen Anzahl verschiedener Identifier führt, wurde das Extended-Format eingeführt, das einen 29 Bit langen Identifier verwendet.
Innerhalb der Experimentierplattform wird nur das Standard-Format eingesetzt, weil alle verwendeten Komponenten mit diesem Arbeiten und die Anzahl verschiedener Nachrichten-IDs ausreichend ist, um alle benötigten CAN-Nachrichten eindeutig zu kennzeichnen.
Im Folgenden werden die Daten-Frames genauer vorgestellt und kurz auf den Remote- und den Fehler-Frame eingegangen. Für weitere Informationen über den Überlast-Frame sei auf die einschlägige Literatur verwiesen.

2.2.3.1 Daten-Frame

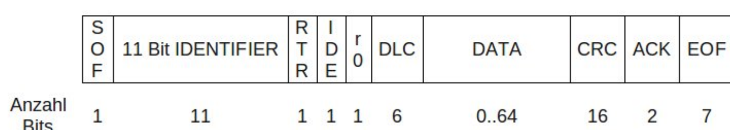


Abbildung 2.4: Daten-Frame im Standard-Format

Der Daten-Frame im Standard-Format aus Abbildung 2.4 beginnt mit einem SOF-Bit, das den Beginn des Frames (**S**tart-**o**f-**F**rame) kennzeichnet und immer einen dominanten Pegel besitzt.
An dieses Bit schließen sich die 11 Bit der Nachrichten-ID an, um den Daten-Frame eindeutig zu identifizieren.
Das nächste Bit ist das sogenannte **R**emote-**T**ransmission-**R**equest-Bit und dient dazu, eine Nachricht als Remote-Frame zu markieren. In normalen Daten-Frames ist dieses Bit grundsätzlich dominant, damit sich ein Daten-Frame bei der Arbitrierung immer gegen einen zugehörigen Remote-Frame durchsetzt. Weitere Informationen zur Arbitrierung sind dem Abschnitt 2.2.4.2 zu entnehmen.
Mit Hilfe des **Id**entifier-**E**xtension-Bits wird signalisiert, ob es sich um eine Nachricht im Standard- oder im Extended-Format handelt. Beim Einsatz des Standard-Formats hat dieses Bit immer einen dominanten Pegel.
Das r0-Feld ist ein reserviertes Bit, das für spätere Erweiterungen vorgesehen ist und einen dominanten Pegel besitzt.
Mit Hilfe des sich anschließenden DLC-Feldes (**D**ata-**L**ength-**C**ode) wird angegeben, wie viele Bytes an Daten in der Nachricht enthalten sind. Dabei sind Werte zwischen 0 Byte und 8 Byte möglich.

Nach dem DLC-Feld befindet sich das DATA-Feld, welches die eigentlichen Daten der Nachricht beinhaltet und eine Länge in Abhängigkeit der übermittelten Daten besitzt.
Im Anschluss an dieses Feld befindet sich das CRC-Feld (**C**yclic **R**edundancy **C**heck) mit der Prüfsumme der Nachricht.
Das folgende ACK-Feld (**Ack**nowledgement) hat eine besondere Bedeutung innerhalb der Nachricht. Der Sender einer Nachricht legt an dieser Position ein rezessives Bit auf den Bus. Falls nun ein Empfänger die Nachricht korrekt empfangen hat, legt er an dieser Position ein dominantes Bit auf den Bus und überschreibt somit das vom Sender gesendete Bit. Weil der Sender diese Abweichung des Signals auf dem Bus gegenüber seinem gesendetem Signal erkennt, weiß er, dass die Nachricht von mindestens einem Empfänger korrekt empfangen wurde. Ist das vom Bus gelesene Signal nicht dominant, geht der Sender von einem Fehler bei der Übertragung aus und sendet die Nachricht erneut.
Mit dem 7 Bit langem **E**nd-**o**f-**F**rame-Feld wird das Ende der übermittelten Nachricht signalisiert. Diese 7 Bit besitzen immer einen rezessiven Pegel.

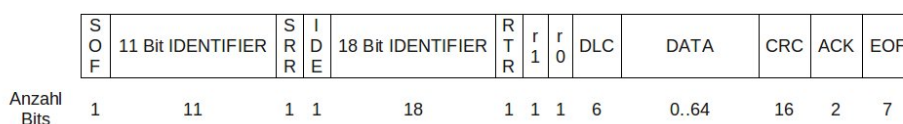


Abbildung 2.5: Daten-Frame im Extended-Format

Beim Extended-Format aus Abbildung 2.5 gibt es einige Unterschiede gegenüber dem Standard-Format. Die erste Änderung befindet sich an der Position des SRR-Bits. Hier befindet sich im Standard-Format das RTR-Bit für die Kennzeichnung als Remote-Frame. Im Extended-Format handelt es sich bei dem SRR-Bit um eine Substitution dieses RTR-Bit, das immer auf einem rezessiven Pegel liegt.
Das IDE-Bit muss im Extended-Format konsequenterweise einen rezessiven Pegel besitzen, um den Frame als Extended-Frame zu kennzeichnen.
An dieses Bit schließen sich die zusätzlichen 18 Bit des Identifiers an, damit die Nachricht über insgesamt 29 Bit für den Identifier verfügt.
Nun folgt das eigentliche RTR-Bit, das wieder die Kennzeichnung eines Frames als Remote-Frame übernimmt. Die beiden folgenden Bits r0 und r1 sind wiederum reservierte Bits für eine spätere Erweiterung.
Der Rest des Nachrichtenaufbaus entspricht dem des Standard-Formats.

2.2.3.2 Remote-Frame

Ein Remote-Frame besitzt im Wesentlichen denselben Aufbau wie ein Daten-Frame. Die einzigen beiden Unterschiede sind, dass das RTR-Bit einen rezessiven Pegel besitzt und das DATA-Feld immer die Länge 0 hat. Trotz des leeren DATA-Feldes wird im DLC-Feld eine Länge für die Daten der Nachricht angegeben. Diese Länge muss mit der Länge der Daten, der angeforderten Nachricht, übereinstimmen.

2.2.3.3 Error-Frame

Der Error-Frame besteht aus einer Sequenz von 6 Bits gleicher Polarität und verstößt somit bewusst gegen die Regel des Bit-Stuffing, auf die im Abschnitt 2.2.4 eingegangen wird. Hierdurch erhalten alle Busteilnehmer eine ungültige Sequenz von Daten beim Auslesen des Busses und melden ihrerseits einen Fehler. Mit Hilfe dieses Frames wird erreicht, dass eine Nachricht busweit verworfen wird, sobald ein Busteilnehmer sie als fehlerhaft meldet.

2.2.4 Kommunikation auf dem CAN-Bus

In diesem Abschnitt werden die Details der Kommunikation über den CAN-Bus vorgestellt. Hierzu wird zunächst das allgemeine Vorgehen bei der Kommunikation beschrieben und anschließend genauer auf die Busarbitrierung und die Fehlererkennung auf dem CAN-Bus eingegangen.

2.2.4.1 Allgemeines zur Kommunikation

Beim CAN-Bus handelt es sich um ein Multi-Master-System, bei dem alle am Bus angeschlossenen Geräte gleichberechtigt sind. Alle Busteilnehmer können unaufgefordert Nachrichten verschicken, weshalb es passieren kann, dass mehrere von ihnen zum selben Zeitpunkt eine Nachricht versenden wollen. Um dieses Problem zu lösen arbeitet der CAN-Bus mit einer Busarbitrierung, die bestimmt welches Steuergerät seine Nachricht versenden darf. Wie die Busarbitrierung im Einzelnen funktioniert ist dem Abschnitt 2.2.4.2 zu entnehmen.
Weil es beim CAN-Bus kein Steuergerät gibt, welches für sich und alle anderen als Zeitmaster dient, müssen die einzelnen Steuergeräte ihre internen Uhren anhand der auf dem Bus übermittelten Nachrichten synchronisieren. Nun arbeitet der CAN-Bus jedoch mit der **N**on-**R**eturn-to-**Z**ero-Kodierung (NRZ-Kodierung), bei der die angeschlossenen Steuergeräte in Abhängigkeit von ihren internen Uhren die Signale auf dem Bus auswerten. Damit es bei einer langen Folge von gleichen Signalen nicht zu Problemen durch unterschiedlich schnell laufende Uhren der verschiedenen Busteilnehmer kommt, werden sogenannte Stopf-Bits (Stuff-Bits) in die Nachricht integriert.
Ein Stuff-Bit wird immer dann in die Nachricht integriert, wenn fünf gleiche Bits nacheinander gesendet wurden. Das Stuff-Bit besitzt dann den invertierten Wert, damit die Busteilnehmer sich durch den Flankenwechsel des Bussignals neu synchronisieren können. Das Stuff-Bit wird auch dann gesendet, wenn ohnehin ein Wechsel des Signals stattfinden würde. Dieses Verfahren wird auch als Bit-Stuffing bezeichnet. Durch die festgelegten Regeln, wann ein Stuff-Bit eingebaut wird, sind die Busteilnehmer in der Lage die vom Sender eingebauten Stuff-Bits zu erkennen und aus der empfangenen Nachricht zu entfernen. Anzumerken sei noch, dass Stuff-Bits nur im Bereich zwischen dem Start-of-Frame-Feld und dem CRC-Feld eines Daten-Frames eingebaut werden [11].
Hat ein Busteilnehmer seine Nachricht übertragen, bestätigen ihm die anderen Busteilnehmer den korrekten Empfang der Nachricht. Diese Bestätigung erfolgt wie beschrieben innerhalb des ACK-Feldes der gesendeten Nachricht, indem zum richtigen Zeitpunkt ein

dominantes Signal auf den Bus gelegt wird. Bleibt das ACK aus, sendet der Nachrichtenabsender die Nachricht erneut. Nach dem erfolgreichen Senden einer Nachricht, warten alle Steuergeräte ein paar Takte, bevor sie den Bus als frei ansehen und das Sendeverfahren erneut beginnt.

2.2.4.2 Busarbitrierung

Mit Hilfe der Busarbitrierung entscheiden die Steuergeräte welches von ihnen seine Nachricht auf dem Bus senden darf und welche ihre Übertragung abbrechen müssen.
Die Arbitrierung erfolgt anhand des auf dem Bus befindlichem Signals. Man kann sich den Anschluss aller Steuergeräte an den CAN-Bus im Prinzip wie ein großes Wired-AND vorstellen, bei dem immer das Resultat des Wired-AND als Signal auf dem Bus vorhanden ist. Hieraus resultiert, dass eine gesendete logische Null sich immer gegenüber einer oder mehrerer gesendeter logischer Einsen durchsetzt. Deshalb wird die logische Null, wie bereits erwähnt, als dominanter Pegel angesehen und die logische Eins als rezessiver Pegel. Wenn nun ein Steuergerät keine Nachrichten zu versenden hat, legt es eine logische Eins auf den Bus, weil diese die dort gesendeten Signale nicht beeinflusst.
Weil jedes am CAN-Bus angeschlossene Steuergerät immer die Signale vom Bus ausliest, auch wenn es selber sendet, wird dieses Verhalten für die Busarbitrierung benutzt. Wenn mehrere Busteilnehmer gleichzeitig senden wollen, tun sie dies zunächst auch alle. Durch das permanente Auslesen des Busses merken die Busteilnehmer, wenn das von ihnen gesendete Signal nicht mit dem vom Bus gelesenem Signal übereinstimmt. Tritt diese Diskrepanz zwischen gesendetem und gelesenem Signal zwischen dem Start-of-Frame-Feld und dem DLC-Feld eines Daten-Frames auf, geht das Steuergerät davon aus, dass ein anderes Steuergerät eine Nachricht mit einer höheren Priorität versendet und stoppt seinen eigenen Sendeversuch.
Dieses Verhalten führt dazu, dass Nachrichten mit einer kleineren Nachrichten-ID eine höhere Priorität besitzen, als Nachrichten mit einer größeren Nachrichten-ID. Zudem werden Daten-Frames den zugehörigen Remote-Frames vorgezogen, weil ein Busteilnehmer gerade versucht die Daten zu senden, die ein anderer Busteilnehmer anfordern möchte. Der wichtigste Vorteil dieser Art von Busarbitrierung ist jedoch die verlustlose Arbitrierung, weil die Gewinnernachricht direkt vollständig übertragen wurde. Diese verlustlose Übertragung wird durch die dominanten und rezessiven Pegel ermöglicht.
In Abbildung 2.6 ist ein Beispiel für die Busarbitrierung zu sehen. In diesem Beispiel beginnen drei Busteilnehmer gleichzeitig mit dem Senden ihrer Nachricht. Die ersten sechs übertragenen Bits sind bei allen drei Teilnehmern gleich, so dass alle ihre Übertragung fortsetzen. Beim siebten Bit sendet der Knoten 1 ein rezessives Signal, liest dann jedoch ein dominantes Signal vom Bus. Dies veranlasst Knoten 1 dazu seine Übertragung einzustellen, weil es sendende Busteilnehmer gibt, die Nachrichten mit einer höheren Priorität senden.
Bis Bit 12 senden die beiden verbleibenden Knoten weiterhin dieselben Signale. Erst bei Bit 13 unterscheiden sie sich, so dass Knoten 3 das Senden seiner Nachricht aufgrund einer niedrigeren Priorität einstellt. Knoten 2 hat in diesem Beispiel also die Busarbitrierung gewonnen. Dieses Beispiel zeigt auch das Verhalten des Remote-Frames, wenn er

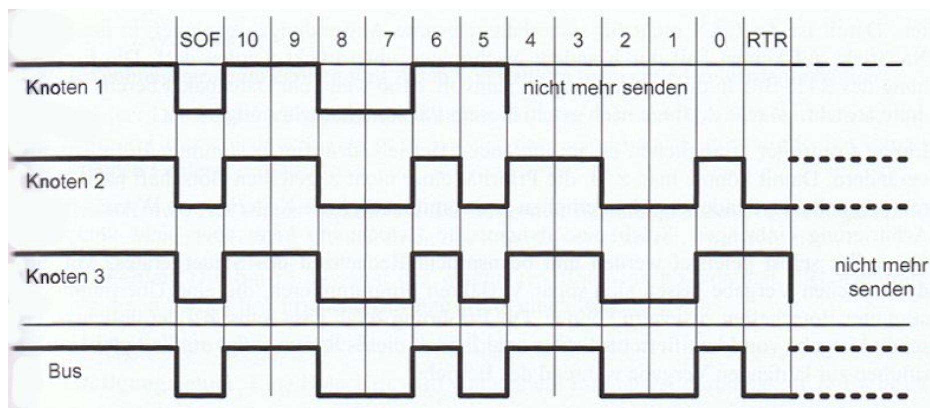


Abbildung 2.6: Arbitrierung beim CAN-Bus (Quelle: [11] Seite 93)

auf den zugehörigen Daten-Frame trifft. Knoten 3 fordert in diesem Beispiel den Daten-Frame an, den Knoten 2 gerade zu senden versucht. Beim Vergleich des Signalverlaufs auf dem Bus mit dem von Knoten 2 ist zu erkennen, dass die Nachricht von Knoten 2 trotz Arbitrierung verlustfrei übertragen wurde.

2.2.4.3 Fehlererkennung

Die am CAN-Bus angeschlossenen Busteilnehmer verfügen über eine Reihe von Fehlererkennungsverfahren. Neben den offensichtlichen Prüfungen der CRC-Prüfsumme, der Einhaltung des Frame-Formats und dem Erhalt des ACKs, gibt es noch weitere Fehlererkennungsverfahren. Eines dieser Verfahren ist die Überwachung des Bussignals. Ein sendender Busteilnehmer überwacht die ganze Zeit den Bus, um sicherzustellen, dass die von ihm gesendeten Signale auch auf dem Bus vorhanden sind. Als Letztes wird noch das Einhalten des Bit-Stuffing geprüft.
Sollte ein Busteilnehmer einen Fehler erkennen, sendet er den in Abschnitt 2.2.3.3 beschriebenen Error-Frame, der aktiv gegen das Bit-Stuffing verstößt und somit alle Busteilnehmer einen Fehler erkennen lässt.
Falls ein Fehler erkannt wird zählt das erkennende Steuergerät einen internen Fehlerzähler hoch. Hierzu gibt es in jedem CAN-Controller einen Fehlerzähler für Sende- und einen für Empfangsfehler. Überschreitet einer dieser Zähler die in der CAN-Spezifikation festgelegten Fehlerschranken, wird das Steuergerät in einen Fehlerzustand versetzt. Als erstes wechselt es in den Fehlerzustand *fehlerpassiv* (error passive), in dem keine Error-Frames beim Feststellen einen Fehlers mehr verschickt werden. Sollten die Fehlerzähler den zweiten Schwellwert überschreiten, wird in den Zustand *vom Bus abgeschaltet* (bus off) gewechselt, in dem überhaupt nicht mehr an der Kommunikation teilgenommen wird [12].
Beim Empfangen und Versenden fehlerfreier Nachrichten werden die Fehlerzähler wieder herunter gezählt. Unterschreitet ein Steuergerät dabei eine Fehlerschranke, wird es wieder in die Kommunikation integriert. Diese Integration erfolgt allerdings nur für Steu-

ergeräte im *fehlerpassiv*-Zustand. Ist das Steuergerät ganz vom Bus abgeschaltet, kann es nur durch einen Reset wieder in die Kommunikation integriert werden [12]. Dieses Verfahren erfüllt den Zweck, dass ein fehlerhaftes Steuergerät nicht die gesamte Kommunikation im CAN-Bus lahmlegen kann. Weitere Informationen zur Fehlererkennung und dem Umgang mit ihnen sind der Literatur, beispielsweise [12], zu entnehmen.

2.3 FlexRay

Bei FlexRay handelt es sich um ein noch ziemlich neues Bussystem, welches primär vor dem Hintergrund der zukünftigen X-by-Wire-Anwendungen, wie z. B. Steer-by-Wire oder Brake-by-Wire, entwickelt wurde [6]. Diese Anwendungen erfordern ein deterministisches Zeitverhalten, welches vom normalen CAN-Bus nicht für alle Nachrichten eingehalten werden kann. Aus diesem Grund wurde 2000 das FlexRay-Konsortium gegründet, in dem unter anderem die Kfz-Hersteller BMW und DaimlerChrysler aktiv waren.

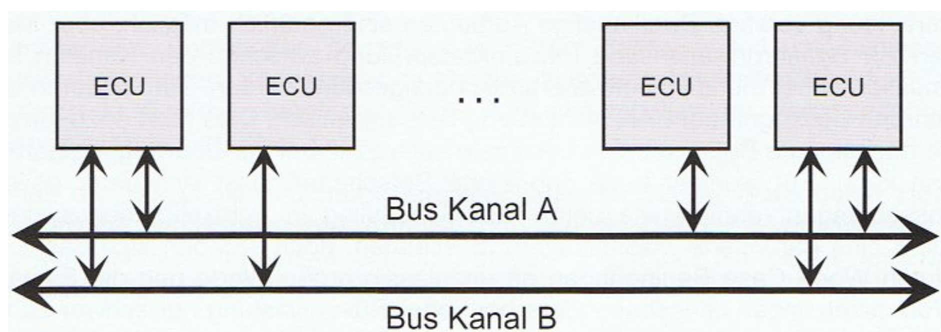


Abbildung 2.7: FlexRay-Bus in Zwei-Kanal-Linien-Struktur (Quelle: [6] Seite 58)

Bei FlexRay handelt es sich um einen zeitgesteuerten Bus, der für eine Übertragungsrate von 10 Mbit/s ausgelegt ist. Der Bus kann dabei in Form eines Ein- oder Zwei-Kanal-Busses aufgebaut werden. Wenn ein Zwei-Kanal-Bus verwendet wird, gibt es zwei verschiedene Nutzungsmöglichkeiten des zweiten Kanals. Bei der ersten Möglichkeit werden auf dem zweiten Kanal dieselben Nachrichten wie auf dem ersten Kanal gesendet und somit eine Redundanz geschaffen. Der zweite Kanal kann alternativ auch für die Übertragung weiterer Daten benutzt werden und so die mögliche Übertragungsrate auf bis zu 20 Mbit/s erhöhen. Bei der Struktur des Busses gibt es ebenfalls verschiedene Möglichkeiten. Der Bus kann in einer Linien-Struktur, wie in Abbildung 2.7 zu sehen, oder in einer Stern-Struktur (siehe Abbildung 2.8) aufgebaut werden. Es sind sogar Kombinationen aus den verschiedenen Strukturen möglich.
Damit FlexRay den Echtzeitanforderungen und dem deterministischem Zeitverhalten gerecht wird, dabei aber weiterhin flexibel bleibt um beispielsweise verschiedene Ausstattungsvarianten für Kraftfahrzeuge zu ermöglichen, wird eine zyklische Datenkommunikation eingesetzt. Ein Kommunikationszyklus unterteilt sich dabei in ein statisches und ein dynamisches Segment. Im statischen Segment gibt es eine Reihe von Zeitschlitzen,

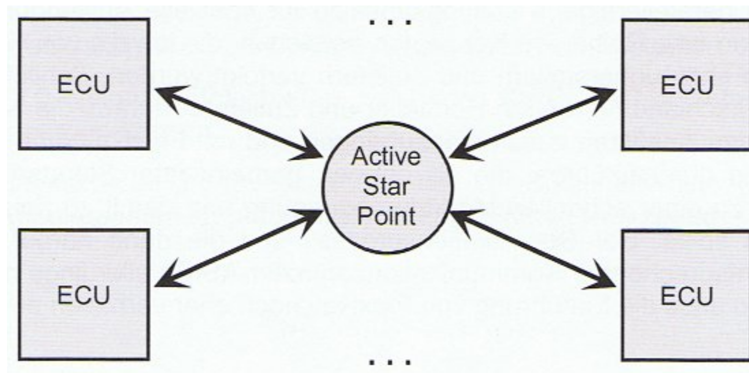


Abbildung 2.8: FlexRay-Bus in Ein-Kanal-Stern-Struktur (Quelle: [6] Seite 58)

die nach dem Prinzip des Time Division Multiple Access (TDMA) festen Nachrichten zugeordnet sind. In diesem Segment werden alle Nachrichten vorgesehen, die ein deterministisches Zeitverhalten benötigen, wie z. B. Nachrichten vom Lenkwinkelsensor. Das dynamische Segment wird ebenfalls in verschiedene Zeitschlitze unterteilt, die aber im Gegensatz zum statischen Segment nicht festen Nachrichten zugeordnet sind. In diesem Segment können Nachrichten auch mehr als einen Zeitschlitz belegen, so dass Nachrichten mit einem Inhalt von maximal 254 Datenbytes ermöglicht werden. Durch das dynamische Segment ist es auch möglich verschiedene Ausstattungsvarianten zu produzieren, da ihre zusätzlichen Nachrichten in diesem Segment verschickt werden können, auch wenn sie vorher nicht berücksichtigt wurden. Dies wäre im statischem Segment nicht möglich, weil hier alle Zeitschlitze festen Nachrichten zugeordnet sind.
Weitere Details zu FlexRay sind in der einschlägigen Literatur, wie beispielsweise [6, 8], zu finden.

3 Konzept der Experimentierplattform

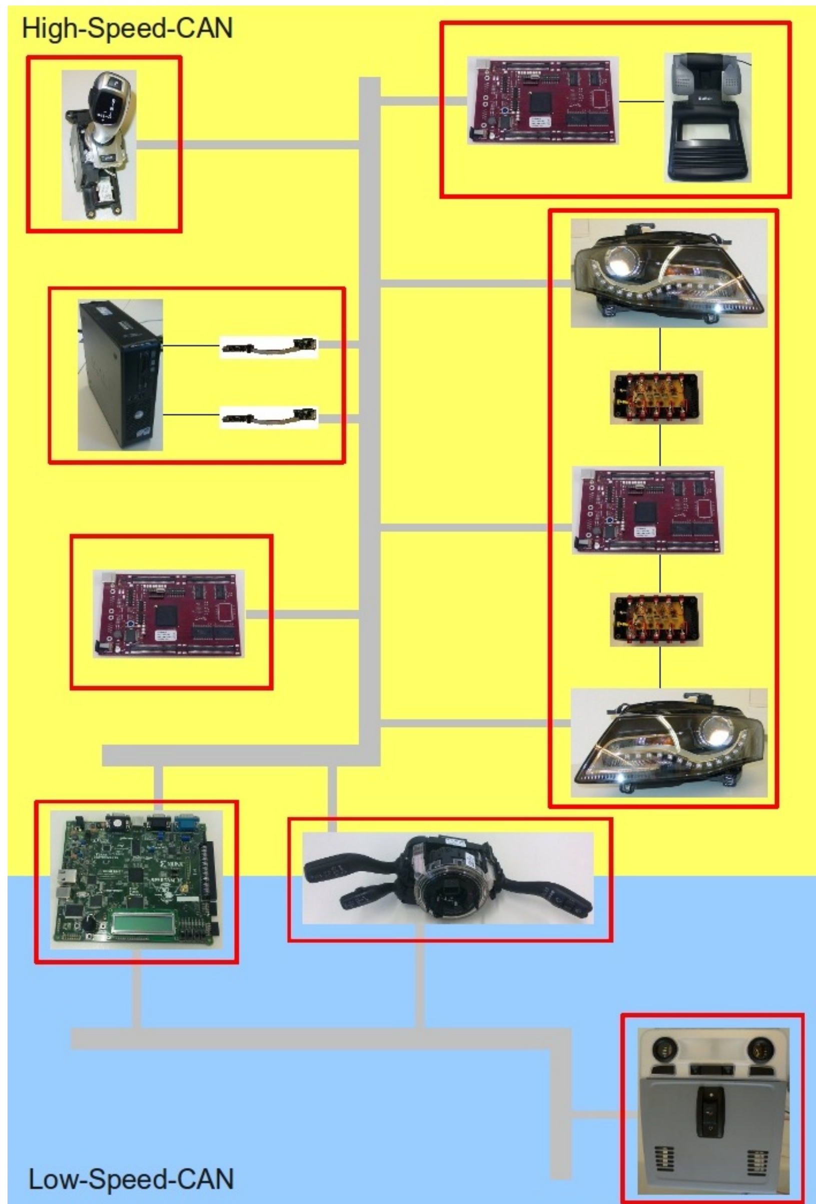


Abbildung 3.1: Aufbau der Experimentierplattform

In modernen Kraftfahrzeugen gibt es eine große Anzahl von Steuergeräten, die sich um die Bewältigung verschiedener Aufgaben kümmern. Prinzipiell lässt sich eine Unterteilung in einen kritischen und einen weniger kritischen Bereich vornehmen, aufgrund welcher die einzelnen Steuergeräte den verschiedenen Bussen zugeordnet werden. Bei Fahrzeugen, die mit einem CAN-Bus ausgestattet sind, werden die Komponenten des Antriebsstrangs an den High-Speed-CAN angeschlossen, weil dieser eine höhere Übertragungsgeschwindigkeit als der Low-Speed-CAN besitzt. Typische Kandidaten für den Anschluss an den High-Speed-CAN sind beispielsweise das Lenksäulenmodul, das ABS-Steuergerät oder die Steuerung der automatischen Leuchtweitenregulierung bei Fahrzeugen mit Xenon-Scheinwerfern. Diese Komponenten sind für den sicheren Betrieb eines Kraftfahrzeuges von enormer Bedeutung, so dass eine schnelle und zuverlässige Kommunikation gewährleistet sein muss.
Die weniger kritischen Komponenten gehören im Wesentlichen zu dem Komfort-Bereich. Bei diesen Komponenten kommt es nicht auf eine sehr hohe Übertragungsgeschwindigkeit an, weil ihre Informationen für den sicheren Betrieb eines Kraftfahrzeuges nicht so kritisch sind, wie die des Antriebsstrangs. So kann z. B. das Signal zum Einschalten der Innenraumbeleuchtung ohne große Probleme eine längere Verzögerung besitzen, wohingegen eine große Verzögerung bei den Eingriffen der Bremsassistenzsysteme einen Unfall zur Folge haben kann. Typische Komponenten des Komfort-Bereichs sind beispielsweise die Lenkstockhebel an der Lenksäule oder ein Steuergerät zur Ansteuerung eines Schiebedaches. Der Anschluss dieser Komponenten erfolgt an den Low-Speed-CAN.
Insgesamt bilden alle in einem Kraftfahrzeug vorhandenen Busse zusammen das Fahrzeugnetz. An den jeweiligen Übergängen zwischen verschiedenen Bussen sind Gateways vorhanden, die für die Weiterleitung der notwendigen Nachrichten auf die anderen Busse sorgen. In einem aktuellen Kraftfahrzeug, das mit dem CAN-Bus arbeitet, übernimmt in der Regel das Kombiinstrument die Rolle des Gateways zwischen dem High-Speed- und Low-Speed-CAN.
Über dieses Fahrzeugnetz kommunizieren die verschiedenen Steuergeräte auf verschiedene Art und Weise miteinander. So kann es Steuergeräte geben, die nur Informationen für andere bereitstellen, selber aber auf keine empfangenen Nachrichten reagieren. Es können aber auch komplexe Zusammenhänge aus den Informationen der verschiedenen Steuergeräte erkannt werden und so beispielsweise das Kurvenlicht bei steigender Geschwindigkeit einen immer kleineren Ausschlag erhalten. Prinzipiell empfängt jedes Steuergerät alle Nachrichten, die auf dem Bus gesendet werden, an dem das Steuergerät angeschlossen ist. Anhand dieser vorhandenen Nachrichten und Einflüssen von Außen, wie beispielsweise das Betätigten von Schaltern, entscheidet sich das Verhalten eines Steuergeräts.
Die in dieser Bachelorarbeit aufgebaute Experimentierplattform beinhaltet ein Fahrzeugnetz, das aus dem High-Speed-CAN und dem Low-Speed-CAN besteht. Innerhalb dieser Plattform kann eigene Software für einen Teil der Komponenten geschrieben werden und somit Einfluss auf ihr Zusammenspiel mit den anderen Komponenten genommen werden. Für die Visualisierung der Abläufe auf dem Bus gibt es insgesamt mehrere Möglichkeiten. Es können zum einen verschiedene Lichtfunktionen gesteuert werden und zum anderen die Reaktionen eines Fahrzeuges innerhalb einer Rennsimulation dargestellt werden. Mit

Hilfe dieses Aufbaus ist es möglich Software für den automotiven Bereich zu entwickeln und ihr Verhalten innerhalb eines Fahrzeugnetzes zu testen.
Für den Aufbau der Experimentierplattform stehen die in Kapitel 4 genannten Komponenten zur Verfügung. Aus ihnen wird das in Abbildung 3.1 zu sehende Fahrzeugnetz aufgebaut, dass wie in aktuellen Kraftfahrzeugen aus einem High-Speed-CAN (gelber Bereich) und einem Low-Speed-CAN (blauer Bereich) besteht. Bis auf das Dachmodul sind alle Komponenten an den High-Speed-CAN angeschlossen. Das CAN-Gateway und das Lenksäulenmodul besitzen zusätzlich einen Anschluss an den Low-Speed-CAN und kommunizieren auf beiden Bussen. Eine besondere Rolle innerhalb dieses Aufbaus nimmt der PC ein, weil dieser über zwei Schnittstellen mit dem High-Speed-CAN verbunden wird. Eine dieser Schnittstellen wird für den Betrieb der Rennsimulation benutzt und die andere für das Programmieren des AVM-Boards (siehe Abschnitt 5.8), welches die in der Entwicklungsumgebung AutoLabIDE geschriebenen Programme ausführt und so komplexe Steuerungsaufgaben innerhalb des Fahrzeugnetzes übernimmt.
Um eine möglichst hohe Flexibilität für den Betrieb des Fahrzeugnetzes zu schaffen, wurde die Experimentierplattform in mehrere Module unterteilt. Die einzelnen Module entsprechen den rot umrandeten Komponenten in Abbildung 3.1 und können zum größten Teil getrennt voneinander betrieben werden. Lediglich für den Betrieb des Lenksäulenmoduls und des Dachmoduls gibt es momentan eine Einschränkung. Weil diese Komponenten den Low-Speed-CAN benutzten und derzeit im ESS-Labor keine Möglichkeit besteht den Low-Speed-CAN direkt auszuwerten, muss für die Arbeit mit diesen Komponenten das CAN-Gateway verwendet werden, das die Nachrichten des Low-Speed-CAN an den High-Speed-CAN weiterleitet. Durch die Modularisierung ist es möglich kleinere Teile des Fahrzeugnetzes zu betreiben und somit den Einfluss fehlender Komponenten im Fahrzeugnetz zu bestimmen oder das direkte Zusammenspiel bestimmter Komponenten zu testen. Ein weiterer Vorteil ist die Möglichkeit mit mehreren Gruppen an verschiedenen Komponenten zu arbeiten und diese in Form von kleinen Fahrzeugnetzen zu betreiben, so dass sich die einzelnen Gruppen bei der Entwicklung nicht gegenseitig stören können.
Alle Module sind soweit es möglich ist in separate Kisten verbaut, die in Anhang A.2 zu sehen sind. Diese Kisten besitzen an ihren Außenseiten alle benötigten Schnittstellen für den Anschluss an den CAN-Bus und die Spannungsversorgung. Die Schnittstelle für den CAN-Bus ist dabei direkt als Y-Weiche ausgelegt und mit einer D-Sub-Buchse und einem D-Sub-Stecker versehen, die es erlauben die einzelnen Kisten untereinander mit einfachen Kabeln zu verbinden und die anfälligen Y-Kabel auf ein Minimum zu reduzieren. Die Kabelbelegungen sind im Anhang A.1 zu finden.
Zur Spannungsversorgung der Experimentierplattform wird das in Abschnitt 4.10 gezeigte Netzteil verwendet. Dieses Netzteil ist ausreichend, um alle Komponenten bei maximaler Stromaufnahme zuverlässig zu versorgen. Bei Bedarf können die einzelnen Module mit einer separaten Spannungsversorgung betrieben werden. Das Scheinwerferboard muss jedoch immer an derselben Spannungsquelle wie die Transistorkästchen und die Scheinwerfer angeschlossen werden und ein ausreichend dimensioniertes Netzteil verwendet werden. Genauere Informationen zu den benötigten Versorgungsspannungen der Komponenten sind dem Anhang A.4 zu entnehmen.

4 Verwendete Hardware

In diesem Kapitel werden die einzelnen Hardwarekomponenten vorgestellt, die im Rahmen dieser Bachelorarbeit verwendet und in die Experimentierplattform integriert wurden.

4.1 Scheinwerfer

Abbildung 4.1: Scheinwerfer aus dem Audi A4

Unter den bereitgestellten Fahrzeugkomponenten befinden sich zwei Scheinwerfer aus dem Audi A4, die wie die anderen Komponenten in die Experimentierplattform integriert werden. Wie in der Abbildung 4.1 zu sehen ist, handelt es sich bei dem Scheinwerfer um ein Modell, welches neben Xenonlicht bereits über ein aus LEDs bestehendes Tagfahrlicht verfügt. Ein weiteres Feature dieser Scheinwerfer ist das vorhandene Kurvenlicht.
Die einzelnen Lichtfunktionen werden durch das Anlegen einer Spannung an den entsprechenden Anschlüssen gesteuert, wohingegen das Kurvenlicht und die Leuchtweitenregulierung direkt über den High-Speed-CAN angesteuert werden.

4.2 Lenksäulenmodul

Das verwendete Lenksäulenmodul aus dem Audi A8 (Abbildung 4.2) verfügt, neben den immer vorhandenen Blinker- und Scheibenwischerhebeln, zusätzlich über den Lenkstockhebel zur Steuerung einer Geschwindigkeitsregelanlage.
Alle Informationen des Lenksäulenmoduls werden über die CAN-Busse zur Verfügung gestellt, wobei die Informationen zum Lenkwinkel auf dem High-Speed-CAN und die Informationen der Lenkstockhebel auf dem Low-Speed-CAN gesendet werden.

Abbildung 4.2: Lenksäulenmodul aus dem Audi A8

4.3 Dachmodul

Abbildung 4.3: Dachmodul aus dem BMW X5

Aus dem Komfort-Bereich eines modernen Kraftfahrzeuges kommt ein Dachmodul aus einem BMW X5 zum Einsatz (Abbildung 4.3). Dieses Dachmodul stellt verschiedene Beleuchtungsmöglichkeiten bereit. Es verfügt über Leseleuchten für Fahrer und Beifahrer und besitzt in der Mitte ein großes Leuchtelement für die Innenraumbeleuchtung. Zusätzlich besitzt es zwei LEDs, die normalerweise zur Beleuchtung der Mittelkonsole verwendet werden und verfügt über einen Schalter zum Bedienen eines Schiebedaches.
Alle vom Dachmodul bereitgestellten Informationen werden auf dem Low-Speed-CAN gesendet, über den auch die verschiedenen Leuchtmittel, mit speziellen Nachrichten, eingeschaltet werden können.

4.4 Gangwahlschalter

Eine weitere Komponente aus dem BMW X5 ist der Gangwahlschalter aus Abbildung 4.4. Dieser Gangwahlschalter wird für Automatikgetriebe eingesetzt und besitzt zwei Arbeitsmodi. Entweder er befindet sich in der rechten Gasse und arbeitet im Automatikmodus oder er befindet sich in der linken Gasse und erlaubt es das Getriebe mit einer TipTronic von Hand zu schalten.
Neben den verschiedenen Stellungen des Gangwahlschalters verfügt er noch über diverse

Abbildung 4.4: Gangwahlschalter aus dem BMW X5

LEDs, die dem Fahrer den aktuellen Schaltmodus und die eingelegte Fahrstufe signalisieren. Zudem gibt es drei Tasten, über die beispielsweise der Sport- oder Parkmodus aktiviert werden können.
Der Gangwahlschalter ist an den High-Speed-CAN angeschlossen, den er für das Senden seiner Informationen und das Empfangen von Steuernachrichten benutzt.

4.5 TriBoard TC1796

Abbildung 4.5: TriBoard TC1796

Für verschiedene Steuerungsaufgaben werden mehrere TriBoards vom Typ TC1796 (siehe Abbildung 4.5), aus dem Hause Infineon, eingesetzt.
Diese TriBoards arbeiten mit einer 32-Bit TriCore-CPU, die mit bis zu 150MHz betrieben werden kann. Durch den integrierten MultiCAN-Chip, der vier CAN-Knoten beinhaltet, kann das TC1796 direkt an den CAN-Bus angeschlossen werden und mit der übrigen Hardware kommunizieren.
Durch seine übrige Ausstattung, wie beispielsweise den 44 Analogeingängen, den 123 digitalen Ein-/Ausgängen und den integrierten Timern, ist das TC1796 in der Lage komplexe Steuerungsaufgaben zu übernehmen und wird deshalb in der Experimentierplattform verwendet.
Für eine genaue Auflistung der Bestandteile und Funktionen des TC1796 sei auf das zugehörige Datenblatt [13] verwiesen.

4.6 Spartan-3E S500

Abbildung 4.6: FPGA Spartan-3E S500

Innerhalb der Experimentierplattform wird als CAN-Gateway das in Abbildung 4.6 zu sehende Field-Programmable Gate Array (FPGA) verwendet. Es handelt sich bei diesem FPGA um ein Spartan-3E S500 der Firma Xilinx, welches günstig in der Anschaffung ist und frei konfiguriert werden kann.
Warum gerade ein FPGA als CAN-Gateway eingesetzt wird, liegt darin begründet, dass für diese Bachelorarbeit schnell ein funktionierendes CAN-Gateway benötigt wurde, um die anderen Komponenten zu testen. Bei dem in der Projektgruppe „AutoLab“ benutzten CAN-Gateway handelt es sich um ein Einzelstück, das bereits anderweitig eingesetzt wird und deshalb in der Experimentierplattform durch ein anderes CAN-Gateway ersetzt werden muss.
Weil im Rahmen der Projektgruppe „CoaCh - Car on a Chip“ ein CAN-Controller entwickelt wurde, der auf dem Spartan-3E S500 lauffähig ist und ein Spartan-3E S500 direkt am Lehrstuhl vorhanden war, wurde dieses als CAN-Gateway ausgewählt. Die genaue Konfiguration des FPGAs ist im Abschnitt 5.1 beschrieben.
Das Spartan-3E verfügt nicht über integrierte CAN-Transceiver, so dass für die Anbindung an den CAN-Bus zwei externe Transceiver verwendet werden. Für den High-Speed-CAN wird ein Transceiver vom Typ MAX 3051 und für den Low-Speed-CAN ein Transceiver vom Typ MAX 3055 verwendet. Weitere Informationen zu den CAN-Transceivern und deren Anschluss an das FPGA, sind dem Abschnitt 5.1.2 zu entnehmen.
Weitergehende Informationen zur Ausstattung des FPGAs sind im Spartan-3E FPGA Starter Kit Board User Guide [14] zu finden.

4.7 Pedaleinheit

Weil sich unter den bereitgestellten Komponenten keine Pedaleinheit befand, wurde auf die Pedaleinheit eines PC-Lenkrads zurückgegriffen. Es wird die Pedaleinheit aus Abbildung 4.7 benutzt, die von einem Saitek R440 Force Feedback Wheel stammt. Damit diese Pedaleinheit in der Experimentierplattform verwendet werden kann, muss sie ihre Pedalstellungen mit Hilfe von CAN-Botschaften den anderen Komponenten mitteilen. Die hierzu notwendigen Anpassungen können dem Abschnitt 5.6 entnommen werden.

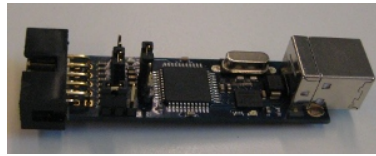
Abbildung 4.7: Pedaleinheit vom Saitek R440 Force Feedback Wheel

4.8 DELL OPTIPLEX 755

Abbildung 4.8: DELL OPTIPLEX 755

Bei dem für diese Bachelorarbeit zur Verfügung gestelltem PC handelt es sich um einen DELL OPTIPLEX 755 (siehe Abbildung 4.8). Dieser PC verfügt über einen Quadcore-Prozessor, dessen 4 Kerne jeweils mit einer Taktfrequenz von 2,83GHz arbeiten. Zudem ist der PC mit 4GB Arbeitsspeicher, einer 250GB großen Festplatte und einer zusätzlichen Grafikkarte vom Typ Radeon HD 2400XT ausgestattet.
Innerhalb der Experimentierplattform wird dieser Rechner die Entwicklungsumgebung AutoLabIDE ausführen und die Rennsimulation TORCS bereitstellen.
Damit der PC in der Lage ist mit den anderen Komponenten über den CAN-Bus zu kommunizieren, wird er mit Hilfe von zwei USBprog im Zusammenspiel mit je einem CAN-SPI-Adapter an den CAN-Bus angeschlossen.
Der Aufbau und die Konfiguration der USBprog und der zusätzlichen CAN-SPI-Adapter ist im Abschnitt 5.7 beschrieben.

4.9 USBprog & CAN-SPI-Adapter



(a) USBprog

(b) CAN-SPI-Adapter

Abbildung 4.9: Schnittstelle vom PC zum CAN-Bus

Als Schnittstelle zwischen PC und CAN-Bus, werden zwei USBprog (siehe Abbildung 4.9(a)) verwendet, die jeweils mit einem CAN-SPI-Adapter aus Abbildung 4.9(b) versehen werden. Beim USBprog handelt es sich um ein Open-Source Universalwerkzeug [15], welches hauptsächlich zum Programmieren verschiedener Mikrocontroller verwendet wird. Durch die Möglichkeit eigene Firmware aufzuspielen, kann der USBprog auch für andere Zwecke verwendet werden, wie dies in dieser Bachelorarbeit geschieht.
Der verwendete CAN-SPI-Adapter stammt aus dem Hause Elektronik-Atelier Kallen und verbindet einen High-Speed-CAN-Bus mit einem über SPI angeschlossenem Gerät. Die genauen Spezifikationen des CAN-SPI-Adapters sind dem Datenblatt [16] zu entnehmen.
Die Programmierung der USBprog und die Verbindung zum zusätzlichen CAN-SPI-Adapter ist im Abschnitt 5.7 beschrieben.

4.10 Netzteil

Abbildung 4.10: Verwendetes Netzteil

Bei dem in Abbildung 4.10 zu sehendem Netzteil, handelt es sich um die Spannungsversorgung für die Experimentierplattform. Dieses Netzteil liefert die in Kraftfahrzeugen übliche Spannung von 13,8V, bis zu einer maximalen Stromaufnahme von 50A. An diesem Netzteil können alle verwendeten Komponenten betrieben werden, die mit einer Versorgungsspannung von 13,8V arbeiten. Die Informationen zu den benötigten Versorgungsspannungen der verwendeten Komponenten sind im Anhang A.4 zu finden.

5 Integration der Komponenten in die Experimentierplattform

In diesem Kapitel wird die Integration der einzelnen Komponenten in die Experimentierplattform beschrieben und auf etwaige Probleme eingegangen.

5.1 Spartan-3E S500

In diesem Abschnitt werden die notwendigen Schritte zur Nutzung des Spartan-3E S500 als CAN-Gateway vorgestellt. Hierzu wird zunächst das für das FPGA synthetisierte **S**ystem-**o**n-**C**hip (SoC) vorgestellt und anschließend auf die innerhalb des SoCs laufende Software eingegangen. Abschließend wird erläutert wie das gebaute System dauerhaft auf dem FPGA aufgespielt werden kann.

5.1.1 Konfiguration des System-on-Chip

Damit das Spartan-3E seiner Aufgabe als CAN-Gateway gerecht werden kann muss ein passendes SoC erzeugt werden. Dieses SoC muss über zwei getrennte CAN-Controller verfügen, die für die Anbindung an den High-Speed-CAN und den Low-Speed-CAN zuständig sind.
Damit die empfangenen CAN-Nachrichten ausgewertet und bei Bedarf weitergeleitet werden können wird ein Prozessor benötigt, der das vom Benutzer geschriebene Programm ausführt.

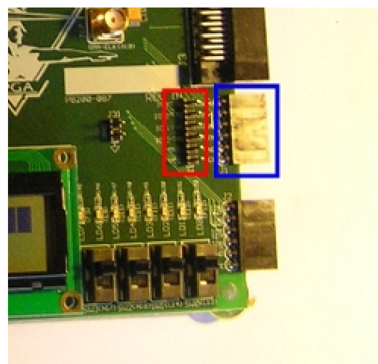


Abbildung 5.1: Zuordnung der Anschlüsse zu den CAN-Controllern

Als Prozessor wird der Plasma von OpenCores.org eingesetzt (http://opencores.org/project,plasma), der den MIPS-Befehlssatz versteht. Innerhalb des gebauten SoCs arbeitet dieser Prozessor mit einer Taktgeschwindigkeit von 25MHz, die für den Einsatz als CAN-Gateway mehr als ausreichend ist. Die Anbindung an den CAN-Bus übernehmen zwei Instanzen des CAN-Controllers, der im Rahmen der Projektgruppe „CoaCh - Car on a Chip“ entwickelt wurde. Es ist bei deren Konfiguration darauf zu achten, dass die Puffer innerhalb der CAN-Controller aktiviert werden, weil es ansonsten zu Problemen während des Betriebs kommen kann. Genauere Details zum CAN-Controller sind dem Abschlussbericht [2] der Projektgruppe zu entnehmen. Den beiden CAN-Controllern sind dabei die in Abbildung 5.1 markierten Anschlüsse zugeordnet. Für den Low-Speed-CAN ist der rot markierte und für den High-Speed-CAN der blau markierte Anschluss vorgesehen.
Damit die vom Lenksäulenmodul benötigten CAN-Nachrichten zeitgesteuert vom CAN-Gateway erzeugt und verschickt werden können, ist zusätzlich ein Timer im SoC vorhanden. Um bei auftretenden Problemen diese besser aufspüren zu können ist eine UART-Schnittstelle im SoC enthalten, mit der es ermöglicht wird Debug-Ausgaben über die Serielle-Schnittstelle auszugeben.
Dieses System-on-Chip wurde synthetisiert und das erzeugte Bitfile anschließend mit der in Abschnitt 5.1.3 beschriebenen Software versehen.

5.1.2 Anschluss externer CAN-Transceiver

Weil auf dem SoC nur CAN-Controller vorhanden sind aber keine CAN-Transceiver, muss der physikalische Anschluss an den CAN-Bus über externe CAN-Transceiver erfolgen.
Für den High-Speed-CAN wird ein CAN-Transceiver-Baustein vom Typ MAX3051 [17], der Firma Maxim Integrated Products, verwendet. Es handelt sich bei diesem Baustein um einen High-Speed-CAN-Transceiver, der mit einer Versorgungsspannung von 3,3V arbeitet und eine Übertragungsrate von maximal 1 Mbit/s erlaubt. Weil die Spannung an den Pins des Spartan-3E maximal 3,3V beträgt, kann dieser CAN-Transceiver direkt vom FPGA versorgt werden. Hierzu wird der Transceiver auf eine Adapterplatine gelötet und diese direkt ans FPGA angesteckt. Als Anschluss an den CAN-Bus ist ein 9-poliger D-Sub-Stecker auf der Adapterplatine vorhanden.
Für den Low-Speed-CAN wird ein CAN-Transceiver-Baustein vom Typ MAX3055 [18] verwendet. Es handelt sich bei diesem Baustein um einen Low-Speed-Transceiver. Der Anschluss an das FPGA gestaltet sich für diesen Baustein jedoch etwas schwieriger, weil er im Gegensatz zum MAX3051 nicht mit einer Versorgungsspannung von 3,3V auskommt. Dieser Umstand liegt in den anderen Signalpegeln des Low-Speed-CAN begründet. Der Signalhub des Differenzsignals, kann beim Low-Speed-CAN, wie in Abschnitt 2.2.2 beschrieben, bis zu 5V betragen. Aus diesem Grund benötigt der MAX3055 auch eine Versorgungsspannung von mindestens 5V, damit er in der Lage ist die Signale vom Low-Speed-CAN korrekt auszuwerten.
Glücklicherweise gibt es auf dem Spartan-3E einen unbestückten Anschluss für eine Batterie (siehe rote Markierung in Abbildung 5.2), an der die Versorgungsspannung

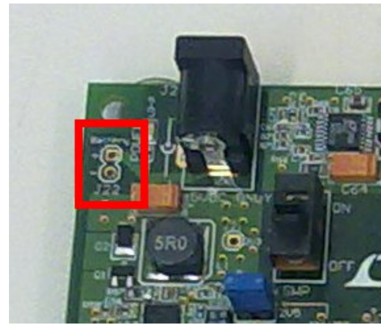


Abbildung 5.2: Batterieanschluss des Spartan-3E

des FPGAs, die 5V beträgt, abgegriffen werden kann. Die Spannungsversorgung des MAX3055 wird über diese Pins hergestellt. Die Datenpins zum Senden und Empfangen werden an dem in Abbildung 5.1 rot markierten Anschluss angeschlossen. Der Batterieanschluss hat zudem den Vorteil, dass hierduch der CAN-Transceiver und das FPGA dieselbe Masse besitzen. Bei Versuchen mit einer externen Spannungsversorgung des Low-Speed-CAN-Transceivers hat sich gezeigt, dass es bei verschiedenen Massen für FPGA und CAN-Transceiver zu Problemen bei der Nachrichtenerkennung kommen kann. Ein Bild des gesamten Aufbaus des CAN-Gateways ist im Anhang A.2.1 zu finden.

5.1.3 Aufbau des ausgeführten Programms

Das auf dem CAN-Gateway ausgeführte Programm arbeitet im Wesentlichen durch das Auswerten von Interrupts. Nach dem Aufrufen einer Initialisierungsmethode verweilt das Programm in einer Endlosschleife.
Innerhalb der Initialisierungsmethode werden alle notwendigen Vorbereitungen für den Betrieb getroffen. Hier werden die Interrupts aktiviert, damit beim Eintreffen einer CAN-Nachricht immer ein Interrupt ausgelöst wird. Die beiden CAN-Controller werden mit den korrekten Übertragungsraten und den zugehörigen Interrupt-Methoden versehen. Der High-Speed-CAN-Controller arbeitet mit 500 Kbit/s und der Low-Speed-CAN-Controller mit 100 Kbit/s. Innerhalb dieser Methode werden auch die benötigten, regelmäßig zu sendenden Nachrichten mit ihren Werten initialisiert. Schließlich wird noch der Timer initialisiert und so eingestellt, dass er alle 25ms einen Interrupt auslöst. Über diesen Interrupt werden die regelmäßig zu versendenden CAN-Nachrichten verschickt.
Die Interrupt-Methode des Timers ist schnell erklärt. Hier werden immer abwechselnd zwei verschiedene CAN-Nachrichten auf dem Low-Speed-CAN verschickt, die für den Betrieb des Lenksäulenmoduls notwendig sind.
Die Interrupt-Methode zur Auswertung des High-Speed-CAN-Controllers filtert die eintreffenden Nachrichten. Es werden hier nur die Nachrichten an den Low-Speed-CAN weitergeleitet, die für den Betrieb des Dachmoduls und des Lenksäulenmoduls notwendig sind. Alle anderen Nachrichten werden an dieser Stelle verworfen, damit der Low-Speed-CAN nicht mit Nachrichten überschwemmt wird.

In der Interrupt-Methode des Low-Speed-CAN-Controllers werden die empfangenen Nachrichten auf die Nachrichten-ID des Lenksäulenmoduls geprüft. Wenn eine eintreffende Nachricht diese ID nicht besitzt, wird sie direkt an den High-Speed-CAN weitergeleitet. Besitzt sie jedoch die Nachrichten-ID des Lenksäulenmoduls, wird die Nachricht in eine neue Nachricht verpackt. Die neue Nachricht besitzt den Aufbau und die ID der korrespondierenden Nachricht des im DLR-SchoolLab eingesetztem Lenksäulenmoduls. Dieser Schritt wurde gewählt, damit die übrigen Komponenten weitestgehend unverändert bleiben können, um im Bedarfsfall im DLR-SchoolLab als Ersatz eingesetzt zu werden. Die neue Nachricht wird schließlich auf dem High-Speed-CAN versendet.
Es ist jedoch sehr wichtig, dass man sich im Klaren darüber ist, dass nicht die Originalnachricht auf dem High-Speed-CAN vorhanden ist. Bei einer Fehlersuche oder dem Versuch, den Hebel zur Steuerung der Geschwindigkeitsregelanlage doch noch funktionsfähig zu machen (weitere Informationen siehe Abschnitt 5.3), muss dies entsprechend berücksichtigt werden. Damit die originale Nachricht auf dem High-Speed-CAN vorhanden ist, muss das Umsetzen der neuen Nachricht auf die alte Nachricht im CAN-Gateway deaktiviert werden. Geschieht dies nicht erhält man nicht alle vom neuen Lenksäulenmodul gesendeten Informationen.

5.1.4 Erzeugen einer dauerhaften Konfiguration für das FPGA

Damit sich das FPGA nach dem Einschalten direkt als CAN-Gateway konfigurieren kann, muss aus dem erzeugten Bitfile eine dauerhafte Konfiguration erzeugt werden. Um diese Konfiguration zu Erzeugen wird in iMPACT der Punkt „PROM File Formatter“ ausgewählt. Dieser Punkt befindet sich im linken Fensterbereich des zum Programmieren des FPGAs benutzten Fensters (siehe Abbildung 5.3).

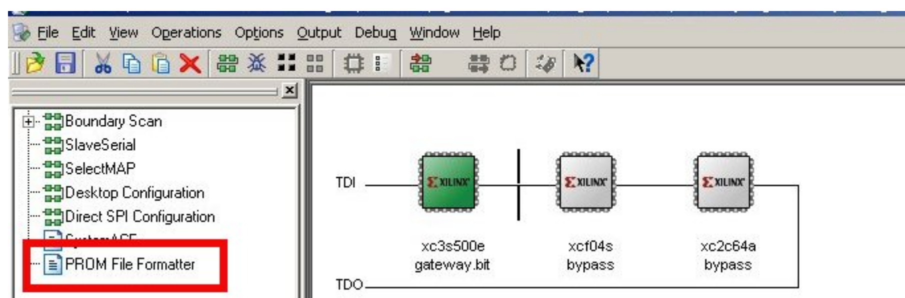
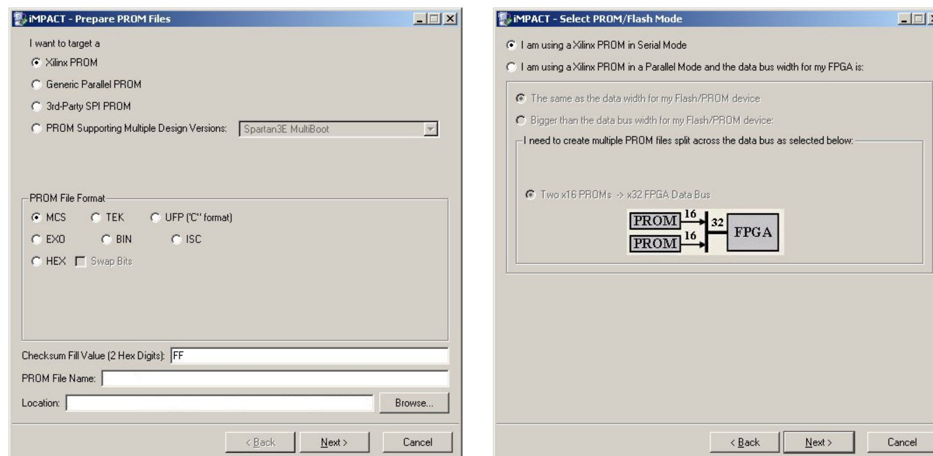
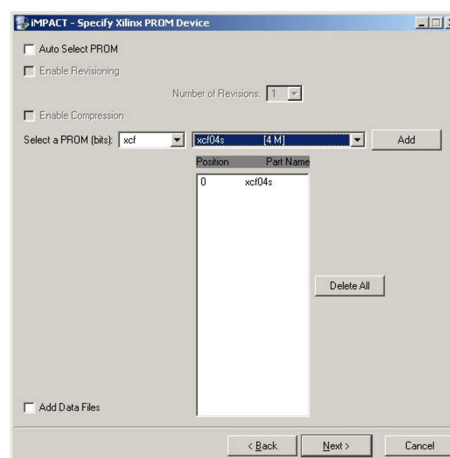


Abbildung 5.3: Menüeintrag PROM File Formatter

Nach einem Doppelklick auf diesen Menüpunkt wird der „Prepare PROM Files Dialog“ angezeigt, der wie in Abbildung 5.4(a) eingestellt werden muss. Im unteren Bereich dieses Dialogs können der Name und der Pfad für die zu erzeugende PROM-Datei eingegeben werden.
Auf der nächsten Seite des Dialogs wird die vorausgewählte Option beibehalten (siehe Abbildung 5.4(b)). In der nun folgenden Dialog-Seite, muss der Speicher des FPGAs



(a) Erste Seite (b) Zweite Seite



(c) Auswahl des Zielspeichers

Abbildung 5.4: Prepare PROM Files Dialog

ausgewählt werden, in dem die Konfiguration abgelegt werden soll. Hierzu wird im Auswahlfeld der Eintrag xcf04s ausgewählt und über einen Klick auf den Add-Button in die Liste der zu verwendenden PROM-Devices hinzugefügt (siehe Abbildung 5.4(c)). Die letzte Seite des Dialogs liefert noch einmal eine Übersicht und kann mit einem Klick auf Finish geschlossen werden. Als nächstes muss das zu verwendende Bitfile angegeben werden.

Zurück im Hauptfenster von iMPACT ergibt sich das Bild aus Abbildung 5.5, in dem zu erkennen ist, dass wir unser Bitfile in den entsprechenden PROM des FPGAs laden wollen. Nun muss über den Menüeintrag „Generate File“, im Menü Operations, die eigentliche PROM-Datei erzeugt werden. Nach dem erfolgreichem Erzeugen dieser Datei erscheint am unteren Fensterrand die Meldung „PROM File Generation Succeeded“ und es kann mit einem Klick auf „Boundary Scan“ im linken Fensterbereich wieder zur ur-

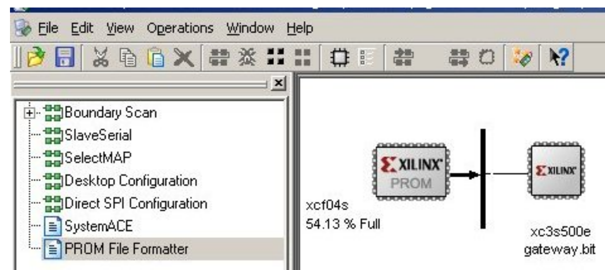


Abbildung 5.5: Erzeugte Konfiguration für die PROM-Datei

sprünglichen Ansicht gewechselt werden.

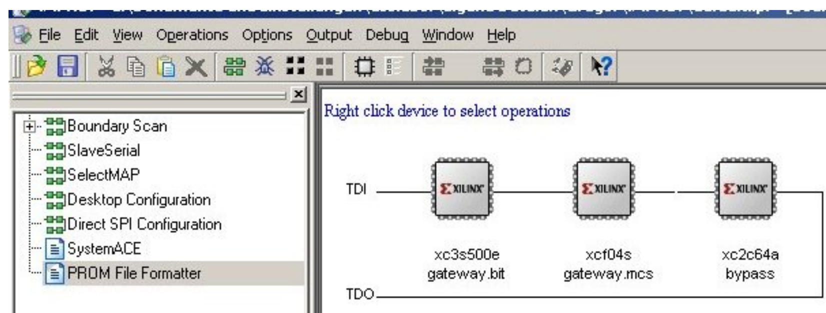


Abbildung 5.6: Fertig zum Programmieren der dauerhaften Konfiguration

In diesem Fenster kann, wie in Abbildung 5.6 zu sehen, der Komponente xcf04s, die gerade erzeugte PROM-Datei gateway.mcs zugeordnet werden und anschließend über den Kontextmenüeintrag „Program“, dieser Komponente, auf das FPGA programmiert werden.
Als Resultat wird jetzt bei jedem Einschalten des FPGAs, die im PROM abgelegte Konfiguration geladen und das FPGA somit als CAN-Gateway konfiguriert.

5.2 Scheinwerfer

Die Steuerung des Kurvenlichts und der Leuchtweitenregulierung erfolgen bei diesen Scheinwerfern direkt über CAN-Nachrichten. Um eine bestimmte Position anzufahren, muss diese an den Scheinwerfer übermittelt werden. Der Scheinwerfer stellt sich entsprechend der erhaltenen Informationen ein und sendet regelmäßig Statusnachrichten über seine aktuelle Scheinwerferstellung. Diese Statusnachrichten können von anderen Komponenten ausgewertet werden, um festzustellen in welcher Position sich die Scheinwerfer befinden.
Im Gegensatz zum Kurvenlicht und der Leuchtweitenregulierung werden die Lichtfunktionen nicht über CAN-Nachrichten, sondern direkt durch Anlegen einer Spannung geschaltet. Aus diesem Grund musste eine Möglichkeit gefunden werden, die es erlaubt die

Spannung über CAN-Nachrichten zu steuern. Diese Aufgabe übernimmt ein TriBoard vom Typ TC1796 (siehe Abschnitt 4.5), das um zwei externe Transistorkästchen erweitert wird.
In diesem Kapitel wird zunächst der Aufbau der Transistorkästchen vorgestellt und anschließend der Anschluss der Scheinwerfer an das TC1796 beschrieben.

5.2.1 Aufbau der Transistorkästchen

Um die benötigte Spannung von 13,8V, mit Hilfe eines TriBoards zu schalten, ist eine Treiberstufe notwendig. Diese muss in der Lage sein, mit den 3,3V des TC1796, die Versorgungsspannung der Scheinwerfer zu schalten. Im Rahmen der Projektgruppe „AutoLab“ wurde für diesen Zweck die in Abbildung 5.7 zu sehende Transistorschaltung entworfen.

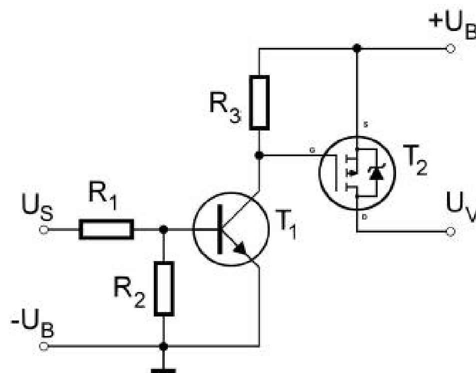


Abbildung 5.7: Transistorschaltung der Projektgruppe „AutoLab“ (Quelle: [19] - Abschnitt Transistorkästchen)

Diese Transistorschaltung besteht aus einem selbst-leitendem P-Kanal MOSFET vom Typ SPP80P06P [20] (T_2) und einem selbst-sperrenden N-Kanal Transistor vom Typ BC550C [21] (T_1). Zusätzlich werden noch zwei 12kΩ Widerstände (R_1, R_2) und ein 27kΩ Widerstand (R_3) benötigt. Wenn eine Steuerspannung U_S anliegt wird T_1 leitend und zieht die Steuerspannung von T_2 auf Masse. T_2 wird hierdurch ebenfalls leitend und die anliegende Spannung U_B wird an den Verbraucherausgang U_V geschaltet. Wenn keine Steuerspannung U_S anliegt sperrt der Transistor T_1. T_2 sperrt in diesem Fall ebenfalls, weil seine Steuerspannung nicht mehr durch T_1 auf Masse gezogen wird. Am Verbraucheranschluss U_V liegt dann keine Spannung an.
Weil jeder Scheinwerfer insgesamt über fünf Spannungsanschlüsse für seine Leuchtelemente verfügt, wird diese Schaltung für jeden Scheinwerfer fünf mal benötigt. Diese fünf Einheiten befinden sich auf einer Platine und sind mit dieser in einer kleinen Box eingebaut. Wie in Abbildung 5.8 zu sehen ist, gibt es sowohl für die Steuerseite (oben im Bild), wie auch für die Verbraucherseite (unten im Bild), eigene Anschlussbuchsen. Die Versorgungsspannung aus dem Netzteil wird an den verbleibenden Anschlüssen (links im Bild)

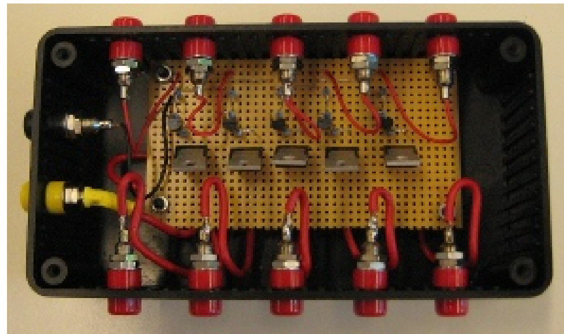


Abbildung 5.8: Fertiges Transistorkästchen

angeschlossen. Insgesamt gibt es zwei von diesen Boxen, damit für jeden Scheinwerfer eine vorhanden ist.

5.2.2 Anschluss an das TC1796

Der Anschluss der Scheinwerfer an das TC1796 gestaltet sich nicht weiter schwierig. Die Ausgänge an der gebauten Kiste für das Scheinwerferboard (siehe Anhang A.2.2), sind mit einer Reihe von Pins des TriBoards verbunden. Die genaue Zuordnung der Funktionen zu den Pins sind der Tabelle 5.1 zu entnehmen.

	Anschluss für	
Funktion	**linken Scheinwerfer**	**rechten Scheinwerfer**
Blinker	P2.8	P3.8
Fernlicht	P2.9	P3.9
Standlicht	P2.10	P3.10
Tagfahrlicht	P2.11	P3.11
Abblendlicht	P2.12	P3.12

Tabelle 5.1: Pinzuordnung für das Scheinwerferboard

An die Anschlüsse der Kiste wird ein Transistorkästchen angeschlossen, auf dessen Verbraucherseite der entsprechende Scheinwerfer angeschlossen wird. Es ist beim Anschluss der Scheinwerfer nur auf eine Sache zu achten. Damit das TriBoard und das Transistorkästchen fehlerfrei zusammenarbeiten, müssen sie über dieselbe Masse verfügen. Weil das Transistorkästchen ohnehin am großen Schaltnetzteil angeschlossen werden muss, wird das Scheinwerferboard auch an diesem angeschlossen.

5.2.3 Software zur Scheinwerfersteuerung

Als Software für die Scheinwerfersteuerung wird die bereits im DLR-SchoolLab eingesetzte Software der Projektgruppe „AutoLab“ verwendet. Es handelt sich bei dieser Software um ein Betriebssystem, welches in der Automobilindustrie Verwendung

findet. Im konkreten Fall wird ein ProOSEK (http://www.proosek.de/) eingesetzt, welches die OSEK/VDX Spezifikation für Betriebssysteme im automotiven Bereich erfüllt. Für weitere Informationen zu OSEK sei auf [22] und die Spezifikationen unter http://www.osek-vdx.org/ verwiesen.
Das eingesetzte Betriebssystem arbeitet mit verschiedenen Tasks, welche die Auswertung von CAN-Nachrichten und das Steuern der Scheinwerfer übernehmen. Durch das Auswerten einer Reihe von CAN-Nachrichten wird entschieden, welche Ausgänge des TC1796 geschaltet werden müssen. Eine Besonderheit gibt es für die Funktion des Tagfahrlichts. Dieses kann in 15 verschiedenen Helligkeitsstufen betrieben werden. Die verschiedenen Helligkeitsstufen werden über eine Pulsweitenmodulation erzeugt, die den zum Tagfahrlicht gehörenden Anschluss entsprechend kurz oder lang einschaltet, um die gewünschte Helligkeit zu erreichen.
Im Zusammenspiel mit den im Rahmen dieser Bachelorarbeit verwendeten Scheinwerfern hat sich gezeigt, dass die Initialisierungsnachrichten für das Kurvenlicht und die Leuchtweitenregulierung von den Scheinwerfern nicht korrekt ausgewertet werden. Diese Initialisierungsnachrichten werden innerhalb des InitTask verschickt und sollen erreichen, dass die Scheinwerfer einmal ihre Stellmotoren von Anschlag zu Anschlag bewegen, um sich so zu kalibrieren.
Versuche haben gezeigt, dass diese Initialisierung problemlos funktioniert, wenn das Scheinwerferboard erst nach den Scheinwerfern eingeschaltet wird. Aus diesem Grund wurde die Software dahingehend angepasst, dass im InitTask vor dem Versenden der Initialisierungsnachrichten eine Wartezeit von 3 Sekunden erfolgt. Diese Zeit reicht aus, damit die Scheinwerferelektronik, nach dem Einschalten, genügend Zeit hat, um sich selbst zu initialisieren und anschließend auf CAN-Nachrichten zu warten. Im Gegensatz zu den im DLR-SchoolLab verwendeten Scheinwerfern, scheinen die neuen Scheinwerfer eine längere Startzeit zu besitzen.
Der Rest der Software ist unverändert geblieben.

5.3 Lenksäulenmodul

Von den reinen Funktionen betrachtet, lässt sich das Lenksäulenmodul in zwei Teile unterteilen. Der erste Teil ist die High-Speed-CAN-Seite des Lenksäulenmoduls, auf der Nachrichten mit Informationen zum Lenkwinkel versendet werden. Der zweite Teil ist die Low-Speed-CAN-Seite. Hier werden die Informationen über die verschiedenen Schalterstellungen gesendet.
In diesem Kapitel werden beide Teile vorgestellt und auf die Schritte eingegangen, die zum Aktivieren des Lenksäulenmoduls notwendig waren.
Es sei an dieser Stelle angemerkt, dass alle Informationen zum Lenksäulenmodul einer NDA von Audi unterliegen und nur intern verwendet werden dürfen. Sollten Teile der CAN-Nachrichten visualisiert werden, müssen die von Audi stammenden Nachrichten unkenntlich gemacht werden. Selbiges gilt für die Informationen zu den Steckerbelegungen.
Dieses Kapitel beschreibt das Vorgehen zur Inbetriebnahme des Lenksäulenmoduls, geht

dabei aber nicht auf die genauen Steckerbelegungen und CAN-Nachrichten ein. Diese Informationen sind im SVN hinterlegt und nur für den internen Gebrauch bestimmt. Die offizielle Dokumentation zum Lenksäulenmodul, sofern sie mir vorlag, befindet sich ebenfalls im SVN. Diesen Unterlagen können weitere Details zum Aufbau des Lenksäulenmoduls und der Funktion einzelner Bestandteile entnommen werden.

5.3.1 Inbetriebnahme des Lenkwinkelgebers

Der erste Versuch den Lenkwinkelgeber des Lenksäulenmoduls, durch einfaches Anlegen der Versorgungsspannung, zum Senden von Nachrichten zu bewegen, ist ohne Erfolg verlaufen. Selbst das Verschicken verschiedener Netzwerkmanagement-Nachrichten auf dem CAN-Bus hat keinen Einfluss auf den Lenkwinkelgeber gehabt.
Anhand der offiziellen Steckerbelegung des verwendeten Lenksäulenmoduls und der Information, welcher Pin dem Zündungs-Signal entspricht, konnten weitere Versuche unternommen werden. Das Anlegen einer Spannung an den Pins für die Versorgungsspannung und das Zündungs-Signal hat den Lenkwinkelgeber in seinen Arbeitsmodus versetzt. An dem bereits vorhandenem Kabel für das Lenksäulenmodul entspricht das grüne Kabel dem Zündungs-Signal.
Während des Betriebs sendet der Lenkwinkelgeber dieselben Nachrichten, wie bereits das im DLR-SchoolLab verwendete Lenksäulenmodul. Der Aufbau und Inhalt dieser Nachrichten, hat sich gegenüber der für dieses Modul vorhandenen Dokumentation nicht verändert. Es ist auch im Rahmen dieser Bachelorarbeit nicht gelungen weitere Informationen über die schon damals unbekannte Nachricht zu erhalten, die zyklisch durch eine Reihe von Werten wechselt.
Dafür konnten weitere Informationen der Nachricht, die den Lenkwinkel enthält, entschlüsselt werden. Der genaue Sinn der übertragenen Bytes 0 bis 4 ist nach wie vor unbekannt. Die Bytes 5 und 6 geben den aktuellen Lenkwinkel wieder, wie dies bereits bei dem anderen Lenksäulenmodul der Fall war.

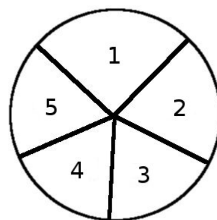


Abbildung 5.9: Unterteilung des Lenkkreises in Segmente

Neu ist die Zuordnung der Information, die das 7. Byte der Nachricht liefert. Wenn man einen Kreis in fünf Segmente, wie in Abbildung 5.9 zu sehen, unterteilt, dann signalisiert das 7. Byte der Nachricht, in welchem Segment des Kreises sich die auf dem Schleifring aufgedruckte Markierung befindet. Diese Segmente scheinen einen Einfluss auf den Startwert des Lenkwinkels zu haben, den dieser erhält, wenn die Spannungsversorgung für

das Lenksäulenmodul eingeschaltet wird.
Damit das Lenksäulenmodul beim Einschalten den Lenkwinkel 0 für seine Mittelstellung erhält, muss sich die am Schleifring aufgedruckte Markierung im Segment 1 befinden. Bei aufgestecktem Lenkrad bedeutet dies, dass das Lenkrad genau gerade steht oder tendenziell etwas nach links gedreht ist. Sollte das Lenkrad nach rechts verdreht sein, erhält der Lenkwinkel beim Einschalten einen anderen Wert und die Mittelstellung ist verschoben. Aus- und Einschalten der Spannungsversorgung, bei korrekter Stellung des Lenkrads, können dieses Problem wieder beheben.
Damit ist der High-Speed-CAN-Teil des eingesetztem Lenksäulenmoduls voll funktionsfähig und kann in der Experimentierplattform benutzt werden.

5.3.2 Inbetriebnahme des Schaltermoduls

Die ersten Versuche das Schaltermodul in Betrieb zu nehmen waren nicht erfolgreich. Der erste Versuch bestand darin die Konfiguration zu benutzen, die für den Betrieb des Lenkwinkelgebers notwendig ist. Hier hat das Schaltermodul allerdings keine Nachrichten über seine Schalterstellungen verschickt. Beim Ausprobieren aller Hebelstellungen ist dann jedoch aufgefallen, dass das Betätigen der Lichthupe das Schaltermodul dazu veranlasst, kurzzeitig Nachrichten zu verschicken. Diese Nachrichten enthalten Informationen über die betätigte bzw. nicht betätigte Lichthupe. Alle anderen Hebelstellungen haben keinen Einfluss auf den Inhalt der Nachricht und deren Versenden gehabt.
Durch das Senden der Netzwerkmanagement-Nachricht, die im DLR-SchoolLab-Aufbau das Lenksäulenmodul am Leben hält, wurde auch das Schaltermodul wach gehalten. Dieser erste Schritt hat allerdings keinen Einfluss auf den Inhalt der Nachrichten des Schaltermoduls gehabt, so dass weiterhin nur Informationen über die Lichthupe gesendet wurden.
Die Berücksichtigung der Funktionen, die in einem Audi A8 in der Regel vorhanden sind und der Erfahrung mit Kraftfahrzeugen aus dem eigenen Umfeld, haben zu der Vermutung geführt, dass es sich bei diesem Verhalten um eine Comming-Home-Funktion handelt. Diese wird bei ausgeschalteter Zündung in der Regel durch betätigen der Lichthupe aktiviert und schaltet die Scheinwerfer, zum Ausleuchten des Gehwegs, für ein paar Sekunden ein. Die Suche ging deshalb in Richtung des Signalisierens einer eingeschalteten Zündung weiter. Weil das Anlegen der Versorgungsspannung an den Pins für das Zündungs-Signal nicht ausgereicht hat um eine eingeschaltete Zündung zu simulieren, konzentrierte sich die Suche von nun an auf CAN-Nachrichten, die diese Aufgabe übernehmen können.
Im Internet waren eine Reihe von möglichen CAN-Nachrichten schnell ausgemacht, die zum Teil von verschiedenen Audi-Modellen, zum Teil aber auch von anderen Fahrzeugen des Volkswagen-Konzerns stammten. Leider war keine Aufzeichnung eines Audi A8 zu finden, so dass mit den anderen Nachrichten experimentiert wurde. Die Versuche mit den gefundenen Nachrichten brachten nicht den erhofften Erfolg.
Um das Schaltermodul dennoch in Betrieb nehmen zu können wurde eine Anfrage beim Automobilzulieferer Kostal gestellt, der dieses Modul freundlicherweise der Universität überlassen hat. Die Antwort bestätigte die Vermutung, dass ein Zündungs-Signal fehlt.

Das innerhalb dieses Lenksäulenmoduls verwendete Schaltermodul wertet die Pins für das Zündungs-Signal nicht mehr eigenständig aus, sondern erwartet diese Information über den CAN-Bus. Durch eine spezielle CAN-Nachricht (siehe SVN) kann das Zündung-Ein-Signal über den CAN-Bus signalisiert werden.
Durch diese Nachricht wird das Schaltermodul zum Leben erweckt und sendet die Informationen über die Hebelstellungen des Blinker- und Scheibenwischerhebels. Gegenüber den korrespondierenden Nachrichten im DLR-SchoolLab hat sich neben der Nachrichten-ID auch der Aufbau der Nachricht ein wenig geändert.
Die erste Vermutung warum die Nachricht dieses Moduls über einen anderen Aufbau verfügt, ging in Richtung des Steuerhebels für die Geschwindigkeitsregelanlage.
Bis zum Schluss dieser Bachelorarbeit war dieser Hebel nicht erfolgreich in Betrieb zu nehmen. Als Grund wird das Fehlen eines Steuergerätes im CAN-Bus vermutet. Weil der Steuerhebel der Geschwindigkeitsregelanlage nicht in Betrieb genommen werden konnte und dessen Nachrichten bisher innerhalb der Experimentierplattform nicht ausgewertet werden, wurde der aufgesteckte Steuerhebel vom Modul entfernt und dieses ohne ihn in Betrieb genommen.
Das Problem mit der geänderten Nachrichten-ID und dem etwas anderem Aufbau der Nachricht wurde innerhalb des CAN-Gateways gelöst (siehe Abschnitt 5.1.3). Dieses Vorgehen wurde aufgrund der Kompatibilität zu den anderen bereits entwickelten und getesteten Softwarekomponenten gewählt. Hierdurch wird erreicht, dass diese Komponenten nicht auf die neue Nachricht angepasst werden müssen und somit nur eine Softwareversion für den DLR-SchoolLab-Aufbau und die im Rahmen dieser Bachelorarbeit aufgebauten Experimentierplattform benötigt wird.
Dieses Vorgehen wird auch dadurch begünstigt, dass sich die Kodierung der Informationen für den Blinker- und Scheibenwischerhebel nicht geändert hat. Bis auf den Steuerhebel für die Geschwindigkeitsregelanlage ist somit auch das Schaltermodul voll funktionsfähig und kann in der Experimentierplattform benutzt werden.

5.4 Dachmodul

Die Integration des Dachmoduls in die Experimentierplattform benötigte keinerlei große Vorarbeiten. Durch die vorliegenden Ergebnisse der Projektgruppe „AutoLab“ [1, 19], ist die Funktion des Dachmoduls bereits bekannt. Es existiert zudem eine Software der Projektgruppe, die in der Lage ist Statusmeldungen über die Schalter zu verschicken und auf eingehende CAN-Nachrichten zu reagieren, die zur Steuerung des Dachmoduls vorgesehen sind. Diese Software wird im Rahmen dieser Bachelorarbeit weiterhin verwendet.
Eine Änderung gegenüber der Nutzung während der Projektgruppe hat sich jedoch ergeben. Im Rahmen des Baus einer Kiste zum Einbau des Dachmoduls wurde die Aufstecklösung einer kleinen Adapterplatine, welche die im Dachmodul durchtrennte Reset-Leitung für den Betrieb wieder schließt, durch einen Schalter ersetzt. Es kann nun bequem von außen, über diesen Schalter, entschieden werden, ob das Modul betrieben oder programmiert werden soll. Die gebaute Kiste mit dem integriertem Schalter ist in Anhang A.2.5

zu sehen.
Beim Programmieren der Software für das Dachmodul ist darauf zu achten, dass immer als erstes der System-Basis-Chip mit Hilfe von SPI-Nachrichten so programmiert wird, dass er den Mikrocontroller des Dachmoduls wach hält. Geschieht diese Programmierung des System-Basis-Chips nicht früh genug oder wird unter Umständen eine fehlerhafte Software eingespielt, schaltet der System-Basis-Chip den Mikrocontroller ab und ein Programmieren ist nicht mehr möglich.
Um das Dachmodul wieder in Betrieb zu nehmen, muss das Dachmodul geöffnet werden und an den NRES-Pin des System-Basis-Chip eine Spannung von 5V angelegt werden, so dass eine funktionsfähige Software eingespielt werden kann. Die genaue Position des NRES-Pins und die SPI-Nachrichten sind der unter [19] vorhandenen Dokumentation zu entnehmen.
Die Programmierung des Dachmoduls erfolgt über die BDM-Schnittstelle mit Hilfe des im Labor vorhandenem inDART-One Programmiergerätes. Als Programmiersoftware wird Metrowerks CodeWarrior eingesetzt. Wie mit Hilfe von CodeWarrior ein Steuergerät über die BDM-Schnittstelle programmiert werden kann, ist im Anhang A.3.1 beschrieben.

5.5 Gangwahlschalter

Die Integration des Gangwahlschalters gestaltete sich wie beim Dachmodul recht einfach. Diese Komponente wurde ebenfalls von der Projektgruppe „AutoLab“ verwendet und ist deshalb bereits unter [1] und [19] dokumentiert.
Auch hier wird wieder die von der Projektgruppe geschriebene Software verwendet. Konkret wird die Softwareversion verwendet, die in der Lage ist die Spulen und den Sport-Knopf anzusteuern. Innerhalb des Programmcodes sind die Spulen allerdings derzeit deaktiviert.
Bei der Programmierung muss, wie schon beim Dachmodul, darauf geachtet werden, dass als erstes der System-Basis-Chip mit Hilfe von SPI-Nachrichten, so programmiert wird, dass er den Mikrocontroller nicht abschaltet. Sollte dies hier vergessen werden oder ein Fehler bei der Programmierung auftreten, muss der Gangwahlschalter geöffnet werden und wie unter [19] beschrieben, eine Spannung von etwa 7V an den Test-Pin des System-Basis-Chip angelegt werden. Hierzu muss ggf. dieser Pin von der Platine abgehoben werden, weil er dort direkt mit Masse verbunden ist. Es sollte nach Möglichkeit vermieden werden, dass der Pin von der Platine abgehoben werden muss, weil dieser sehr schnell abbricht.
Die eigentliche Programmierung erfolgt wie schon beim Dachmodul über die BDM-Schnittstelle und mit Hilfe von Metrowerks CodeWarrior (siehe Anhang A.3.1).

5.6 Pedaleinheit

In diesem Abschnitt wird die Funktionsweise der verwendeten Pedaleinheit vorgestellt und das Vorgehen zur Integration in die Experimentierplattform beschrieben.

5.6.1 Aufbau und Funktion der Pedaleinheit

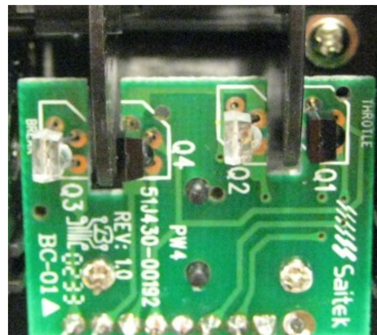


Abbildung 5.10: Platine im Inneren der Pedaleinheit

Weil zu der Pedaleinheit des Saitek R440 Force Feedback Racing Wheels keine öffentliche Dokumentation exisitert, musste deren Funktion durch Reverse Engineering ermittelt werden.
Im Inneren der Pedaleinheit befindet sich lediglich eine kleine Platine (siehe Abbildung 5.10), die neben zwei Infrarotdioden und zwei Fototransistoren keine weiteren Bauteile besitzt. Zwischen je einer Infrarotdiode und einem Fototransistor läuft eine Lochscheibe hindurch, die mit der Achse eines Pedals verbunden ist. Es handelt sich bei der hier eingesetzten Technik um einen Inkrementalgeber, der in ähnlicher Form auch zur Auswertung der Scrollräder in Computermäusen eingesetzt wird [23].
Weil die verwendeten Fototransistoren über zwei Sensorelemente verfügen, ist es möglich die Drehrichtung der Pedalachse zu bestimmen. Jedes mal wenn ein Schlitz der Lochscheibe an einem Sensor des Fototransistors vorbei läuft, wird dieser durch den Lichteinfall leitend und erzeugt so ein Signal auf der zugehörigen Ausgangsleitung.
Weil die beiden Sensorelemente im Fototransistor nebeneinander angeordnet sind, ergeben sich bei der Rotation der Lochscheibe zwei identische Signale, die durch den Abstand der Sensorelemente jedoch Phasenverschoben sind. Eine Aufzeichnung während der Betätigung der Pedale ist der Abbildung 5.11 zu entnehmen. In der Abbildung gehören das gelbe und das grüne Signal zum Gaspedal und die beiden verbleibenden Signale zum Bremspedal.
Wenn immer ein Signal eines Fototransistors beobachtet wird, ist es durch die Phasenverschiebung des zweiten Signals dieses Fototransistors möglich, die Drehrichtung der Lochscheibe zu bestimmen und somit festzustellen, ob das Pedal getreten oder losgelassen wird. Durch den begrenzten Pedalweg, bei aufgesteckten Pedalen, ergibt sich eine

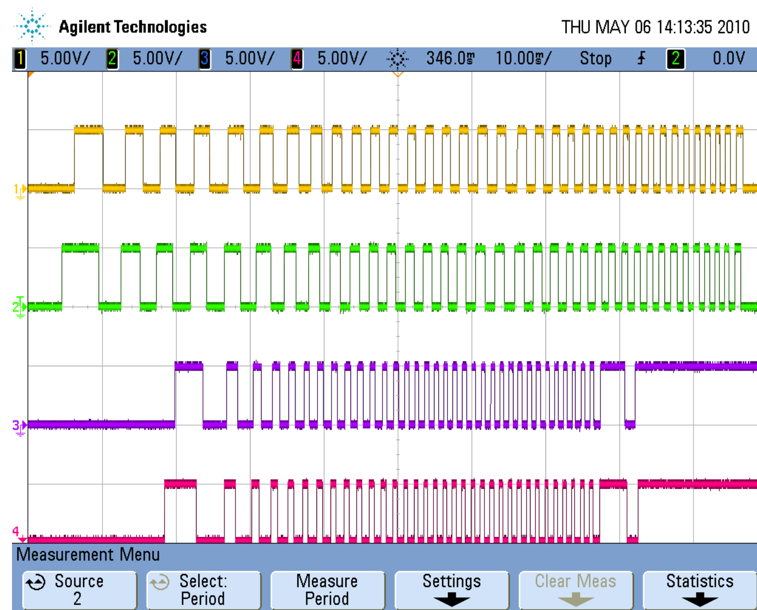


Abbildung 5.11: Signale der Pedaleinheit

Auflösung von 33 Schritten zwischen komplett getreten und komplett losgelassen.
Eine Messung während des Betriebs der Pedaleinheit hat ergeben, dass die Infrarotdioden mit einer Spannung von 1,25V versorgt werden und an den Fototransistoren 5V Spannung anliegen. Der Anschlussplatine innerhalb der Lenksäule lies sich durch Nachmessen entnehmen, dass die Pedale mit einer Versorgungsspannung von 5V betrieben werden und die Infrarotleds über einen Vorwiderstand von 300Ω an diese angeschlossen sind. Dies bedeutet, dass eine Infrarotled eine Stromaufnahme von ca. 13mA besitzt.
Der Tabelle 5.2 kann die Zuordnung der einzelnen Kabel zu den Bauteilen und deren Funktion entnommen werden.

Kabel	**Funktion**
orange	Masse für gesamte Platine
blau	von den Fototransistoren geschaltete Spannung
weiß	Versorgungsspannung der Infrarotled des Bremspedals
grün	Versorgungsspannung der Infrarotled des Gaspedals
rot	erstes Sensorsignal vom Fototransistor des Bremspedals
schwarz	zweites Sensorsignal vom Fototransistor des Bremspedals
lila	erstes Sensorsignal vom Fototransistor des Gaspedals
gelb	zweites Sensorsignal vom Fototransistor des Gaspedals

Tabelle 5.2: Funktionen der einzelnen Kabel

5.6.2 Umbau der Pedaleinheit

Damit die Pedaleinheit mit den anderen Komponenten der Experimentierplattform kommunizieren kann, muss sie an den CAN-Bus angeschlossen werden. Hierzu wird die Pedaleinheit an eines der im Abschnitt 4.5 vorgestellten TriBoards, vom Typ TC1796, angeschlossen, das die Auswertung der Pedaleinheit und die Kommunikation über den CAN-Bus übernimmt.
Weil das TC1796 nur eine Spannung von 3,3V an seinen Portpins zur Verfügung stellt, waren Anpassungen an der Pedaleinheit notwendig.
Ein Test hat ergeben, dass die reduzierte Spannung keinerlei Einfluss auf die Fototransistoren hat, so dass diese ohne weiteres mit 3,3V versorgt werden können. Für die Anpassung der Versorgungsspannung der Infrarotleds mussten geeignete Widerstände bestimmt werden. Aufgrund der gemessenen Versorgungsspannung von 1,25V und der Stromaufnahme von ca. 13mA errechnet sich ein Widerstandswert von 180Ω. Damit das TC1796 keinen Schaden nimmt, wenn die Fototransistoren schalten, wurde je ein 100Ω Widerstand in die Signalleitungen eingelötet, um den fließenden Strom zu begrenzen.
Die so angepassten Pedale können nun am TC1796 angeschlossen werden und liefern, über die Fototransistoren, Signale mit einem Signalpegel von 3,3V, die direkt vom TC1796 an seinen Pins verarbeitet werden können.

5.6.3 Anschluss am TC1796

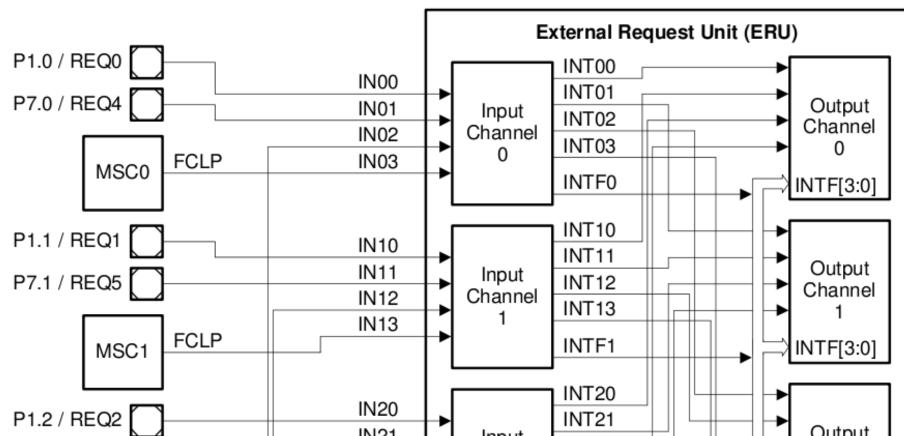


Abbildung 5.12: Ein Teil der Eingänge der External Request Unit (Quelle: [24] Seite 5-14)

Um die Pedaleinheit am TriBoard TC1796 anzuschließen und die Pedalsignale über Interrupts auszuwerten, mussten zunächst geeignete Portpins gefunden werden. Wie dem TC1796 User's Manual System and Peripheral Units [24] zu entnehmen ist, besitzt das TC1796 eine External Request Unit (ERU), die unter anderem in der Lage ist Interrupts zu erzeugen, wenn an bestimmten Eingängen die von der Software konfigurierten Signale

anliegen. Einem Teil dieser Eingänge sind Portpins von den Erweiterungsplatinen des TC1796 zugeordnet. Wie der Abbildung 5.12 zu entnehmen ist, gehören unter anderem die Portpins P1.0 und P1.1 zu diesen.
Weil auf der zugehörigen Erweiterungsplatine neben den Portpins P1.0 und P1.1 auch Portpins für Masse und 3,3V Versorgungsspannung vorhanden sind, wurden diese Portpins für den Anschluss der Pedaleinheit ausgewählt.
Damit die Pedaleinheit nicht fest mit dem TC1796 verbunden werden muss, wurde eine Steckverbindung in Form von D-Sub-Stecker und D-Sub-Buchse gewählt. Die Zuordnung der einzelnen Pins der Steckverbindung zu den Portpins des TC1796 und den Leitungen der Pedaleinheit, sind der Tabelle 5.3 zu entnehmen.

Pin an Stecker/Buchse	**Kabel der Pedaleinheit**	**Portpin am TC1796**
1	blau	3,3V
2	weiß	3,3V
3	rot	P1.3
4	lila	P1.2
5	-	-
6	orange	GND
7	grün	3,3V
8	schwarz	P1.1
9	gelb	P1.0

Tabelle 5.3: Belegung der Steckverbindung

5.6.4 Software zur Auswertung der Pedaleinheit

Die Auswertung der Pedaleinheit erfolgt in der Software durch das Auswerten der von der ERU erzeugten Interrupts. Weil durch den Einsatz eines Inkrementalgebers kein direktes Abfragen der Pedalstellung, wie beim Einsatz eines Potentiometers, möglich ist, muss gezählt werden wie viele Signale ein Pedal über seine Fototransistoren erzeugt hat. Hierzu wird jeweils ein Signal vom Gas- und Bremspedal mit einem Interrupt versehen, so dass bei Auftreten dieses Interrupts das zugehörige zweite Signal ausgewertet werden kann. In Abhängigkeit von diesem zweiten Signal kann die Drehrichtung der Pedalachse bestimmt und somit der entsprechende Wert für die Gas- bzw. Bremspedalstellung um Eins erhöht oder reduziert werden.
Die Werte für Gas- und Bremspedalstellung sind globale Variablen, damit diese direkt aus den Interruptroutinen verändert werden können und für das Versenden von CAN-Nachrichten zur Verfügung stehen.
Zur Auswertung des Bremssignals wird das schwarze Kabel mit einem Interrupt beobachtet. Dabei wird nur bei einer steigenden Flanke ein Interrupt ausgelöst. Beim Auslösen dieses Interrupts wird geprüft, ob der Pegel des roten Kabels ebenfalls High ist. Sollte dies der Fall sein, wurde das Bremspedal weiter getreten, so dass der Wert für die Bremspedalstellung um Eins erhöht werden muss. Ist der Pegel des roten Kabels jedoch

Low, wurde das Pedal losgelassen und der Wert für die Bremspedalstellung muss um Eins reduziert werden. Die Auswertung für das Gaspedal erfolgt auf dieselbe Art und Weise, nur dass hier das gelbe Kabel mit einem Interrupt verbunden ist und bei dessen Auslösung das lila Kabel ausgewertet wird.
Damit ein sicheres Verhalten zur Bestimmung der Pedalstellungen erreicht wird, wurde in der Software die Auflösung auf 25 Schritte pro Pedal reduziert, so dass bei Erreichen des Maximal- bzw. Minimalwertes diese nicht weiter erhöht bzw. reduziert werden. Diese künstliche Reduzierung der Auflösung hat allerdings den Nachteil, dass nicht mehr der komplette Pedalweg ausgenutzt wird. Es wird mit diesem Vorgehen jedoch erreicht, dass sich die Pedale beim Einschalten der Versorgungsspannung in einer beliebigen Stellung befinden können und nach einmaligen Durchtreten und Loslassen der Pedale neu kalibriert haben.
Diese automatische Kalibrierung hat noch einen weiteren Nutzen, da die Pedaleinheit am oberen Anschlag der Pedale nicht immer die genaue Endposition erreicht. Durch diese Ungenauigkeit am oberen Anschlag der Pedale kann es passieren, dass trotz losgelassenem Pedal der zugehörige Wert zur Pedalstellung nicht komplett auf Null zurückgeht. Hier hilft die automatische Kalibrierung weiter, weil es ausreicht das entsprechende Pedal einmal komplett durchzutreten und wieder los zu lassen, damit die Werte wieder korrekt auf Null zurückgehen.
Um das problematische Verhalten der Pedaleinheit weiter zu reduzieren, wird aus beiden Pedalstellungen die Differenz bestimmt und anhand dieser entschieden, wie stark Gas gegeben oder Gebremst wird. Dieses Vorgehen kommt der Umsetzung der CAN-Integration in das Rennspiel TORCS insofern entgegen, dass dort das Bremssignal höher gewichtet wird, als das Gassignal. Würden die beiden Pedalstellungen unabhängig voneinander gesendet, könnte eine „hängende“ Bremse dazu führen, dass sich das Fahrzeug nicht mehr bewegt. Die Kombination der beiden Achsen reduziert dieses Problem auf ein Minimum. Die Ursache für diese Probleme ist der mechanische Aufbau der Pedaleinheit, der sich leider nicht anpassen lässt. Würden statt der Inkrementalgeber Potentiometer verwendet, gäbe es diese Probleme nicht und die Auswertung der Pedalstellung könnte direkt durch Analog-Digital-Wandler erfolgen.
Damit die zur Auswertung benötigten Interrupts innerhalb der Software benutzt werden können, muss zunächst das Interrupt-System des TC1796 korrekt initialisiert werden. Als erstes muss eine Interrupt-Tabelle angelegt werden, die Verweise auf die zu benutzenden Interrupt-Routinen beinhaltet. Diese Tabelle wird direkt in einer Assembler-Datei angelegt und verweist auf die jeweiligen Interrupt-Routinen für Gas- und Bremspedal. Damit das TC1796 diese Interrupt-Tabelle benutzt, muss innerhalb der Software dem TC1796 die Adresse dieser Interrupt-Tabelle mitgeteilt werden.
Nachdem die Interrupt-Tabelle gesetzt wurde, können die benutzten Portpins mit den korrekten Interrupts versehen werden. Hierzu muss jedem der Portpins ein Input Channel zugeordnet werden, über den der anliegende Wert in die ERU hineingeführt wird (siehe Abbildung 5.12). Die ERU vergleicht das anliegende Signal mit einem gesetztem Muster und erzeugt bei Übereinstimmung einen Interrupt. Das Vergleichsmuster wird ebenfalls für den Portpin gesetzt und entspricht in dieser Software einer steigenden Flanke. Erkennt die ERU eine steigende Flanke auf einem Portpin, erzeugt sie im Output Channel

des zugehörigen Portpins einen Interrupt. Der verwendete Output Channel muss wieder für die verwendeten Portpins konfiguriert werden.
Damit die erzeugten Interrupts bis zur CPU gelangen können, müssen noch die gewählten Output Channel dem DMA-Controller zugeordnet werden, damit dieser die Interrupts weiterleiten kann. Zum Schluss muss schließlich das Interrupt-System global aktiviert werden, damit Interrupts ausgelöst werden können. Falls nun die Pedale betätigt werden, löst die ERU einen Interrupt aus und es werden die selbst geschriebenen Interrupt-Methoden aufgerufen. Weitere Informationen zu dem Interrupt-System des TC1796 sind unter [24], in den Kapiteln 5 und 14, zu finden.

5.7 USBprog & CAN-SPI-Adapter

Bei den verwendeten USBprog handelt es sich um die Version 3.0. Dieses Universalwerkzeug wird über den USB-Anschluss mit dem PC verbunden und kann sowohl unter Windows, wie auch Linux programmiert werden. Neben den vielen bereitgestellten Firmwareversionen, zum Einsatz als Programmiergerät für verschiedene Mikrocontroller, können auch selbst entwickelte Firmwareversionen eingespielt werden. Von dieser Möglichkeit wird Gebrauch gemacht, um eine Firmware einzuspielen, die vom PC empfangene Daten so aufbereitet, dass sie als SPI-Nachrichten an den CAN-SPI-Adapter und von diesem als CAN-Nachricht auf dem Bus verschickt werden. Die vom CAN-SPI-Adapter empfangenen Nachrichten werden ausgewertet und an den PC weitergeleitet.

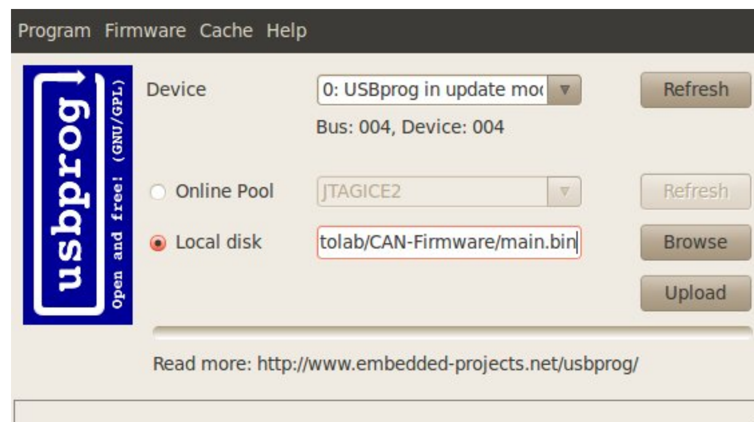


Abbildung 5.13: grafisches Programmiertool unter Ubuntu

Damit die eigene Firmware auf dem USBprog aufgespielt werden kann, muss sie zunächst aus dem Quellcode erzeugt werden. Der Quellcode befindet sich im Verzeichnis CAN-Firmware, das im DLR-SchoolLab-SVN zu finden ist. Im ersten Schritt muss im Unterverzeichnis canlib durch Aufruf von `make` die CAN-Library gebaut werden, bevor im Hauptverzeichnis mit dem erneuten Aufruf von `make` die eigentliche Firmware gebaut werden kann. Diese Firmware kann nun beispielsweise mit dem in Abbildung 5.13 zu sehenden Programmiertool auf den USBprog gespielt werden. Hierzu muss dieser per

Jumper in den Programmiermodus versetzt werden. Eine entsprechende Beschreibung ist im CAN-Firmware-Verzeichnis vorhanden.
Die Verbindung zwischen USBprog und CAN-SPI-Adapter wird durch ein 10-poliges Flachbandkabel hergestellt. Die genaue Kabelbelegung ist dem Anhang A.1.2 zu entnehmen. Über dieses Flachbandkabel kommuniziert der CAN-SPI-Adapter mit dem USBprog und stellt seine Spannungsversorgung her. Der Anschluss an den CAN-Bus erfolgt über eine D-Sub-Schnittstelle. Eine Besonderheit am CAN-SPI-Adapter ist der blaue Jumper, der im Auslieferungszustand auf der Platine steckt. Bei gestecktem Jumper, wird der Abschlusswiderstand mit dem Bus verbunden, so dass hier darauf geachtet werden muss, dass nicht zu viele Abschlusswiderstände im Bus vorhanden sind.

5.8 AVM-Board mit VirtualMachine

Eine weitere wichtige Komponente innerhalb der Experimentierplattform ist das TriBoard, das die VirtualMachine beinhaltet. Auf diesem Board wird ein bereits vorhandenes CiAO-System ausgeführt, welches eine VirtualMachine (VM) betreibt. In dieser VM werden die innerhalb der AutoLabIDE erzeugten Programme ausgeführt. Damit das System während des Betriebs mit erzeugten Programmen versehen werden kann, stellt es nach dem Starten einen Bootloader bereit. Dieser Bootloader ermöglicht eine Progammierung mit Hilfe von CAN-Nachrichten und erlaubt es, dass sich die ausgeführten Programme jederzeit durch die AutoLabIDE austauschen lassen. Im Simulationsbetrieb übernimmt dieses Board alle anfallenden Aufgaben, wie beispielsweise das Koppeln der Steuernachrichten für das Kurvenlicht an die erhaltenen Lenkwinkelinformationen oder das Erzeugen der für die Rennsimulation TORCS benötigten CAN-Nachrichten. Die Verwendung eigener Nachrichten für die Rennsimulation ermöglicht es, Einfluss auf die von den Komponenten gesendeten Daten zu nehmen und nur ausgewählte oder angepasste Daten an die Rennsimulation weiterzuleiten.
Für weitere Informationen zu CiAO sei auf [25] verwiesen.

6 Anpassung der Entwicklungsumgebung

In diesem Kapitel werden die notwendigen Anpassungen am Betriebssystem und der verwendeten Software beschrieben, die für den Einsatz der AutoLabIDE und der Rennsimulation TORCS notwendig sind.

6.1 Ubuntu 10.04 64-Bit

Als Betriebssystem wird auf dem Rechner der Experimentierplattform die Linux Distribution Ubuntu 10.04 in der 64-Bit-Version eingesetzt. Damit die verwendeten Programme unter diesem Betriebssystem laufen und die angeschlossene Hardware benutzt werden kann, müssen ein paar Anpassungen am Betriebssystem vorgenommen werden. Diese Anpassungen werden in den folgenden Abschnitten beschrieben.

6.1.1 Grundlegende Installation

Weil es in der Experimentierplattform keine besonderen Anforderungen bezüglich der Partitionierung gibt, wurde Ubuntu komplett in eine Partition installiert. Der einzige auf dem System vorhandene Benutzer ist der Benutzer „autolab“ mit dem im SVN hinterlegten Passwort, unter dem später die benötigten Programme ausgeführt werden.
Damit die im Rechner verbaute Grafikkarte mit ihrem vollen Potenzial genutzt werden kann, muss der proprietäre Treiber von ATI installiert werden. Diese Installation erfolgt unter Ubuntu über den Menüpunkt „System-Administration-Hardware Drivers“. Hier kann der notwendige Treiber einfach aktiviert werden. Nach einem Neustart wird der neue Treiber verwendet und die Grafikkarte kann ihr gesamtes Potenzial entfalten.
Eine weitere Anpassung ist in Hinblick auf die Nutzung der USBprog notwendig. Damit der Benutzer „autolab“ die USBprog benutzten kann, benötigt er die passenden Zugriffsrechte auf die Hardware. Diese Rechte lassen sich beim Einstecken eines USBprog, durch eine zusätzliche udev-Regel, korrekt setzen.
Für die udev-Regel wird die Datei `/etc/udev/rules.d/99-autolab-torcs-can.rules` angelegt und mit dem Inhalt aus Listing 6.1 versehen. Ab jetzt wird der USBprog beim Einstecken anhand seiner VendorID und ProductID erkannt und dem Benuzter „autolab“ mit den korrekten Rechten zugeordnet.

```
# USB2CAN device: owner "autolab"
SUBSYSTEM=="usb", ATTRS{idProduct}=="0c62",
ATTRS{idVendor}=="1786", OWNER="autolab",MODE="0600"
```

Listing 6.1: Inhalt der Datei 99-autolab-torcs-can.rules

6.1.2 Zusätzlich benötigte Pakete

Damit die in der Experimentierplattform benötigten Programme kompiliert und benutzt werden können, werden eine Reihe von zusätzlichen Paketen benötigt. Welche Pakete dies genau sind wird in diesem Abschnitt beschrieben.

6.1.2.1 Pakete für die Rennsimulation TORCS

Damit die Rennsimulation TORCS aus ihren Quelldateien erzeugt werden kann, wird eine komplette Entwicklungsumgebung benötigt. Die notwendigen Pakete lassen sich mit dem Befehl aus Listing 6.2 installieren.

```
sudo aptitude install build-essential dpkg-dev freeglut3 \
    freeglut3-dev g++ g++-4.4 libalut-dev libalut0 libdrm-dev \
    libgl1-mesa-dev libglu1-mesa-dev libice-dev libopenal-dev \
    libopenal1 libplib-dev libplib1 libpng12-devlibusb-dev \
    libpthread-stubs0 libpthread-stubs0-dev libsm-dev \
    libstdc++6-4.4-dev libx11-dev libxau-dev libxcb1-dev \
    libxdmcp-dev libxext-dev libxi-dev libxmu-dev \
    libxmu-headers libxrandr-dev libxrender-dev libxt-dev \
    libxxf86vm-dev mesa-common-dev x11proto-core-dev \
    x11proto-input-dev x11proto-kb-dev x11proto-randr-dev \
    x11proto-render-dev x11proto-xext-dev \
    x11proto-xf86vidmode-dev xlibmesa-gl-dev xtrans-dev \
    xz-utils zlib1g-dev
```

Listing 6.2: Befehl zum Installieren der für TORCS benötigten Pakete

6.1.2.2 Pakete für die Entwicklungsumgebung AutoLabIDE

Weil es sich bei der AutoLabIDE um ein in Python geschriebenes Programm handelt, wird eine Python-Umgebung benötigt. Diese ist unter Ubuntu bereits standardmäßig installiert, so dass nur ein paar zusätzliche Pakete installiert werden müssen, um beispielsweise von Python aus auf die USB-Schnittstelle zugreifen zu können. Der Befehl aus Listing 6.3 installiert die für den Betrieb der AutoLabIDE notwendigen Pakete.

```
sudo aptitude install libblas3gf libgfortran3 liblapack3gf \
    libmikmod2 libportmidi0 libsdl-image1.2 \
    libsdl-mixer1.2 libsdl-ttf2.0-0 libsmpeg0 python-numpy \
    python-pygame python-usb
```

Listing 6.3: Befehl zum Installieren der für AutoLabIDE benötigten Pakete

6.1.2.3 Pakete für die CAN-Firmware

Damit die Firmware für den USBprog kompiliert werden kann, muss der benötigte Compiler installiert werden. Die benötigten Pakete werden mit Hilfe des Befehls aus Listing 6.4 installiert.

```
sudo aptitude install avr-libc binutils-avr gcc-avr
```

Listing 6.4: Befehl zum Installieren der notwendigen Pakete für die CAN-Firmware

6.1.2.4 Pakete für Programmierung der USBprog

Um die am Rechner angeschlossenen USBprog programmieren zu können, wird ein Programmiertool benötigt. Die Programmierung kann direkt über die Konsole oder eine grafische Oberfläche erfolgen. Die benötigten Pakete können mit dem Befehl aus Listing 6.5 installiert werden.

```
sudo aptitude install libreadline5 libusbprog0 libwxbase2.6-0 \
    libwxgtk2.6-0 usbprog usbprog-gui
```

Listing 6.5: Befehl zum Installieren der für USBprog benötigten Pakete

6.1.2.5 Pakete für Subversion

Damit das DLR-SchoolLab-SVN ausgecheckt werden kann, wird ein SVN-Client benötigt. Dieser lässt sich mit dem Befehl aus Listing 6.6 installieren.

```
sudo aptitude install bsd-mailx exim4 exim4-base exim4-config \
    exim4-daemon-light libapr1 libaprutil1 \
    libconfig-inifiles-perl libsvn-perl libsvn1
    python-subversion subversion subversion-tools
```

Listing 6.6: Befehl zum Installieren des SVN-Clients

6.2 Rennsimulation TORCS

Wie bereits bei der Projektgruppe „AutoLab“, wird die Open-Source-Rennsimulation TORCS verwendet, um eine Demonstrationsumgebung für die in der Experimentierplattform verwendeten Eingabegeräte zu schaffen. Zum Einsatz kommt wieder die Version 1.3.0, die bereits für das Empfangen und Auswerten von CAN-Nachrichten angepasst wurde.
Weil im Gegensatz zum DLR-SchoolLab als Betriebssystem Ubuntu 10.04 in der 64-Bit-Version verwendet wird, mussten ein paar Anpassungen am Quellcode vorgenommen werden, damit dieser sich kompilieren ließ. Diese Anpassungen sind notwendig, weil die unter Ubuntu 10.04 verwendete Version des gcc das Zuweisen eines const char an einen normalen char nicht mehr ohne expliziten const-Cast erlaubt. Zudem mussten in einer Datei ein paar Includes ergänzt werden.
Die vorgenommenen Änderungen sind der Tabelle 6.1 zu entnehmen.

Datei	src/modules/graphic/ssggraph/grtexture.cpp
Anpassung	Zeile 51 ändern in s = const_cast<char*>(strrchr(tfname, ’/’));

Datei	src/tools/trackgen/objects.cpp
Anpassungen	Zeile 276 ändern in s = char *s = const_cast<char*>(strrchr (tfname, ’\\’)); Zeile 279 ändern in s = s = const_cast<char*>(strrchr (tfname, ’/’));

Datei	src/drivers/olethros/Trajectory.cpp
Anpassungen	#include <time.h> #include <stdio.h> #include <string.h> im Kopf ergänzen

Tabelle 6.1: Anpassungen am TORCS-Quellcode

Die angepasste Version ist im DLR-SchoolLab-SVN zu finden. Um die Rennsimulation TORCS aus den Paketquellen zu erzeugen, sind die Schritte in der dort hinterlegten Anleitung zu befolgen. Nach dem erfolgreichen Kompilieren kann TORCS durch den Aufruf von /home/autolab/Torcs/torcs_can/bin/torcs gestaret werden.

6.3 AutoLabIDE

Beim Testen der bereits für Ubuntu 10.04 angepassten AutoLabIDE, aus dem DLR-SchoolLab-SVN, hat sich gezeigt, dass ein Problem bei der Kodierung der Umlaute

existiert. Dieses Problem entsteht durch die Ereignisnamen des Gangwahlschalters, die Umlaute enthalten. Beim Versuch ein Programm zu speichern tritt ein UnicodeEncodeError auf, weil unter Ubuntu 10.04 UTF-8 als Kodierung eingestellt ist, XML aber standardmäßig nicht in UTF-8 speichert.
Zur Lösung des Problems muss die in Tabelle 6.2 gezeigte Anpassung vorgenommen werden. Nach dieser Anpassung sind die Umlaute kein Problem mehr. Die aktualisierte Version wurde nach Rücksprache in das DLR-SchoolLab-SVN integriert, so dass zukünftig diese Anpassung nicht mehr notwendig ist, da sie bereits integriert wurde.

Datei	`loadsave.py`
Anpassung	Zeile 104 ändern in s = `file_object.write(doc.toprettyxml(encoding="utf-8"))`

Tabelle 6.2: Anpassungen am AutoLabIDE-Quellcode

Um die AutoLabIDE in den Fenstermodus zu versetzen, muss in der `Autolab.py` die Zeile 698 auskommentiert und die Zeile 699 einkommentiert werden. In Zeile 699 kann auch die gewünschte Fenstergröße eingestellt werden.
Zum Starten der AutoLabIDE muss der Befehl `python Autolab.py -s` innerhalb des AutoLabIDE-Verzeichnisses ausgeführt werden. Der Parameter `-s` erlaubt das Speichern eigener Programme. Ohne diesen Parameter können erstellte Programme nicht gespeichert werden.

7 Fazit

Im Rahmen dieser Bachelorarbeit ist eine Experimentierplattform entstanden, die neben den Komponenten der Projektgruppen „AutoLab“ und „CoaCh - Car on a Chip“ bisher nicht verwendete Komponenten enthält. Das bisher nicht benutzte Lenksäulenmodul aus einem Audi A8 konnte erfolgreich in die Experimentierplattform integriert werden und funktioniert mit Ausnahme des Steuerhebels der Geschwindigkeitsregelanlage vollständig.
Die nicht vorhandene Pedaleinheit ist durch die Pedale eines PC-Lenkrads ersetzt worden, die so modifiziert wurden, dass sie mit Hilfe eines TriBoard TC1796 in den CAN-Bus integriert werden konnten.
Für das zu ersetzende CAN-Gateway ist eine auf einem FPGA basierende Lösung gewählt worden. Diese Lösung gewährleistet zusätzlich die Kompatibilität des neuen Lenksäulenmoduls zu den bereits aus den Projektgruppen vorhandenen Komponenten, indem die geänderten CAN-Nachrichten auf die alten CAN-Nachrichten abgebildet werden.
Durch die Modularisierung des Aufbaus der Experimentierplattform ist eine flexible Umgebung entstanden. In dieser Umgebung können sowohl einzelne Komponenten, wie auch größere Fahrzeugnetze getestet werden. Ein weiterer Vorteil der Modularisierung ist, dass mehrere Gruppen an verschiedenen Komponenten arbeiten können, ohne immer auf den gesamten Aufbau zurückgreifen zu müssen.
In der vorliegenden schriftlichen Ausarbeitung ist die gesamte Experimentierplattform dokumentiert und es wird auf besondere Probleme während des Aufbaus eingegangen. Mit Hilfe dieser Dokumentation sollte es möglich sein, die Experimentierplattform zu reproduzieren.
Ein paar Anregungen für zukünftige Versionen der Experimentierplattform sollten dennoch erwähnt werden. Die Lösung den PC über zwei USBprog mit CAN-SPI-Adaptern an den CAN-Bus anzuschließen ist nicht sehr optimal. Bei einer zu hohen Buslast kann es passieren, dass Nachrichten verloren gehen. Dies hat den Nachteil, dass Steuernachrichten die Rennsimulation TORCS möglicherweise nicht mehr erreichen. Auch der Ersatz des CAN-Gateways ist eine Lösung, die in der Zukunft überdacht werden sollte. Für die derzeitige Aufgabe ist das FPGA mehr als überdimensioniert, da seine rekonfigurierbare Logik nicht sinnvoll benutzt wird und noch erhebliche Kapazitäten auf dem FPGA frei sind. Hier sollten entweder weitere Steuergeräte auf dem FPGA integriert werden oder es sollte darüber nachgedacht werden, das FPGA durch eine auf Mikrocontrollern basierende Lösung zu ersetzen. Geeignet könnte hier beispielsweise ein Atmel AT128CAN mit einem zusätzlich über SPI angeschlossenem zweiten CAN-Controller sein.
Als zusammenfassendes Ergebnis kann festgehalten werden, dass alle gestellten Aufgaben im Rahmen dieser Bachelorarbeit erfüllt wurden.

A Weitere Informationen

A.1 Kabelbelegungen

A.1.1 CAN-Bus

Es wird CAT-5e-Patchkabel verwendet, um die CAN-Bus-Kabel zu bauen. Dabei wird an einer Seite ein D-Sub-Stecker und an der anderen Seite eine D-Sub-Buchse angebracht. Die genaue Belegung sieht wie folgt aus.

Pin an D-Sub-Stecker/Buchse	**Ader**	**CAN-Signal**
2	orange	CAN-Low
7	orange/weiß	CAN-High

Tabelle A.1: Kabelbelegung für CAN-Bus

A.1.2 USBprog & CAN-SPI-Adapter

Die USBprog werden über ein 10-poliges Flachbandkabel mit den CAN-SPI-Adaptern verbunden. Auf der Seite des USBprog kann der Anschluss über einen ganz normal aufgepressten Stecker erfolgen. Damit die Anschlüsse des USBprog korrekt an den CAN-SPI-Adapter angeschlossen werden, muss auf der Seite des CAN-SPI-Adapters eine andere Ordnung der einzelnen Adern des Flachbandkabels, vor dem Aufpressen des Steckers, hergestellt werden. Die Zuordnung der einzelnen Funktionen zu den Pins der Anschlüsse ist der folgenden Tabelle zu entnehmen.

Funktion	**USBprog**	**SPI-CAN**
MOSI	1	4
VCC	2	1
-	3	-
/CS	4	2
/RESET	5	7

Funktion	**USBprog**	**SPI-CAN**
-	6	-
SCK	7	3
/INT	8	8
MISO	9	5
GND	10	6

Tabelle A.2: Kabelbelegung für USBprog & CAN-SPI-Adapter

A.2 Fotos von den Elementen des Aufbaus

A.2.1 CAN-Gateway

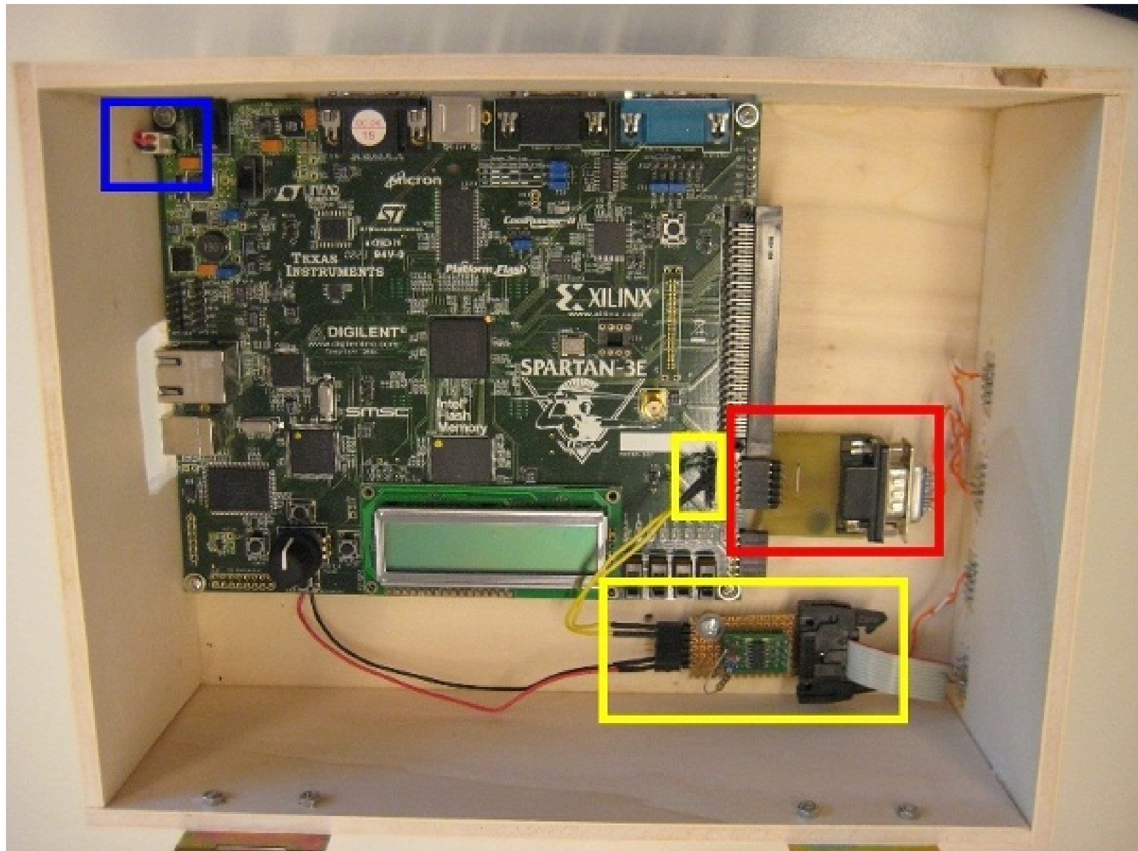


Abbildung A.1: Gesamtübersicht CAN-Gateway

rot: High-Speed-CAN mit MAX3051
gelb: Low-Speed-CAN mit MAX3055
blau: Spannungsversorgung für MAX 3055

A.2.2 Scheinwerferboard

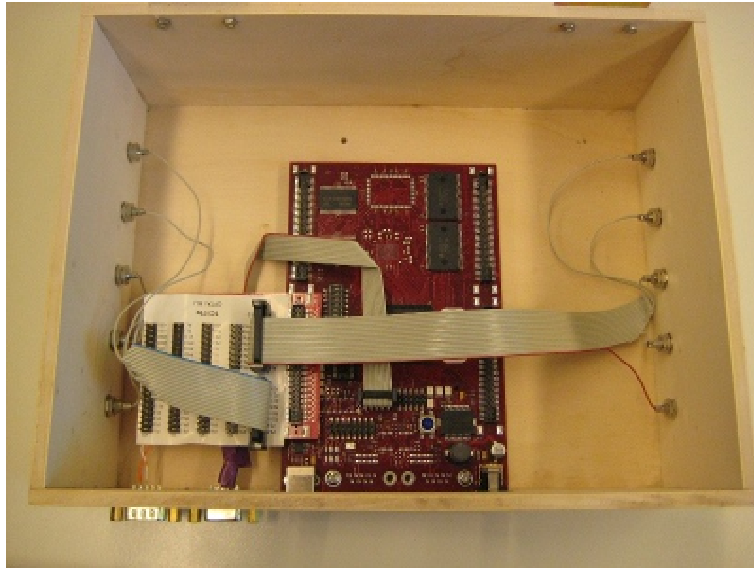


Abbildung A.2: Scheinwerferboard in seiner Kiste

A.2.3 Pedalboard

Abbildung A.3: Pedalboard in seiner Kiste mit Anschluss für Pedale

A.2.4 AVM-Board

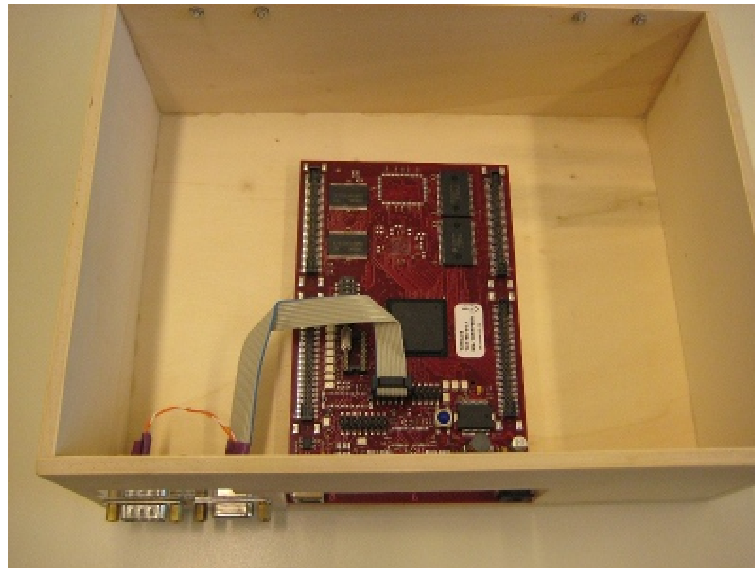


Abbildung A.4: TC1796 mit Virtual Machine in seiner Kiste

A.2.5 Dachmodul

Abbildung A.5: Dachmodul in seiner Kiste mit Schalter für Betriebsmodus

A.2.6 Gangwahlschalter

Abbildung A.6: Gangwahlschalter in seiner Kiste

A.2.7 Lenksäulenmodul

Abbildung A.7: Lenksäule in ihrer Kiste

A.3 Programmieren & Flashen der Komponenten

A.3.1 Steuergeräte mit CodeWarrior programmieren

In diesem Abschnitt wird beschrieben, wie das Dachmodul und der Gangwahlschalter mit Hilfe der Software CodeWarrior über die BDM-Schnittstelle programmiert werden können.

A.3.1.1 BDM-Schnittstelle

Bei der BDM-Schnittstelle (Background Debug Mode) handelt es sich im Wesentlichen um eine proprietäre Schnittstelle von Freescale, die es erlaubt eingebettete Systeme zu Debuggen [26]. Sie wird innerhalb der Experimentierplattform nur im Dachmodul und im Gangwahlschalter verwendet. Diese Komponenten können mit Hilfe des inDartOne-Programmiergerätes direkt über die BDM-Schnittstelle programmiert werden.

A.3.1.2 Programmierung

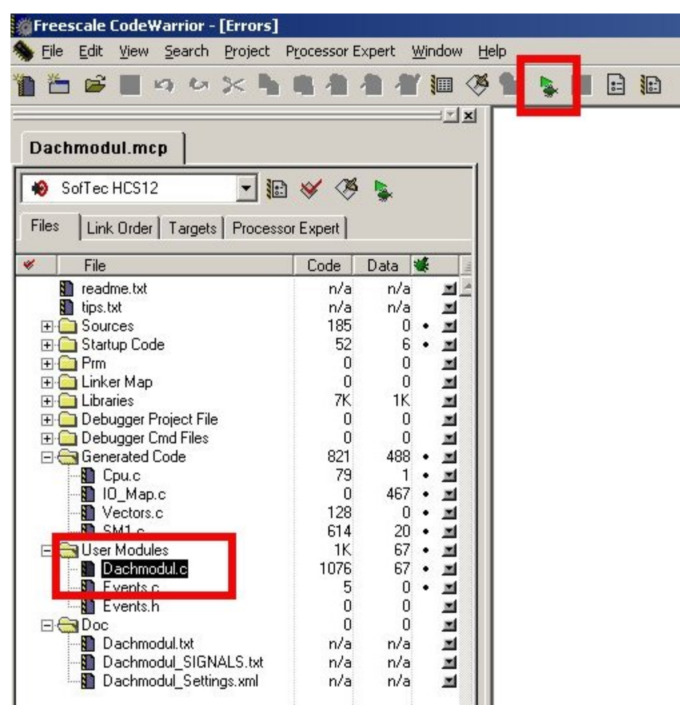


Abbildung A.8: CodeWarrior-Oberfläche zum Programmieren

Um ein Steuergerät mit CodeWarrior zu programmieren, muss das entsprechende Projekt geöffnet und die zu programmierende Datei ausgewählt werden (siehe Abbildung A.8 unterer roter Kasten). Das eigentliche Programmieren wird durch einen Klick auf den

grünen Play-Knopf in der Symbolleiste gestartet (siehe Abbildung A.8 oberer roter Kasten). Nach dem Klick auf diesen Knopf erscheint eine Abfrage, in der die Voreinstellungen beibehalten werden. Die Software versucht nun ein angeschlossenes Gerät zu finden und dieses zu Programmieren.

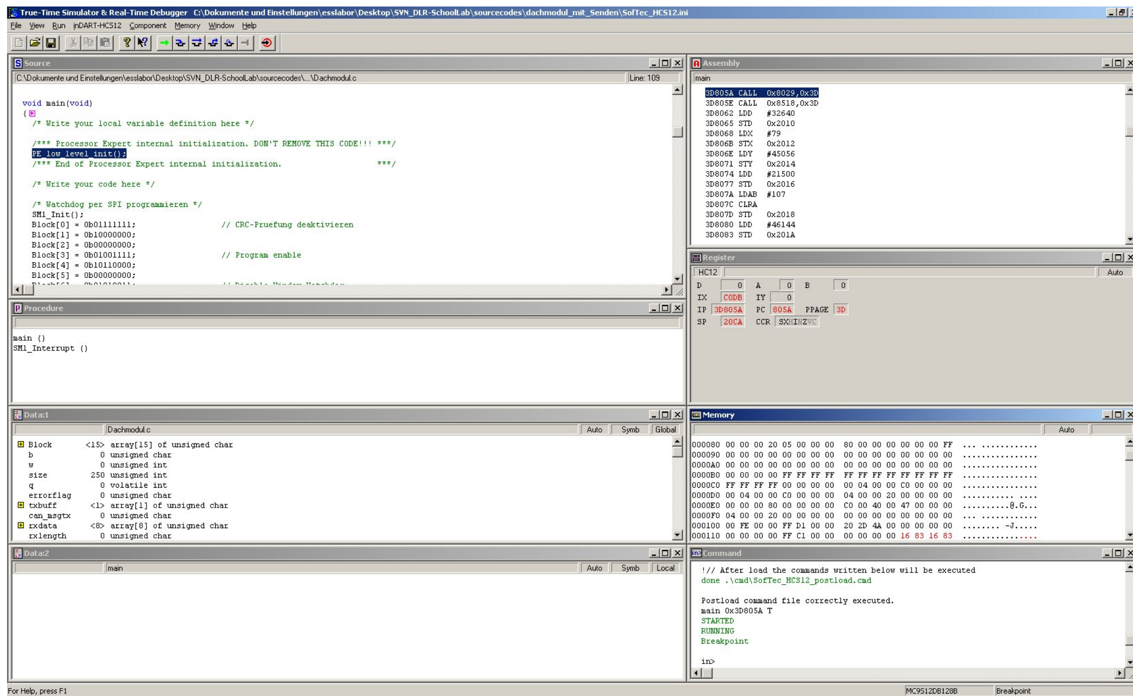


Abbildung A.9: CodeWarrior-Oberfläche zum Debuggen

Nach erfolgreichem Abschluss der Programmierung erscheint das Fenster aus Abbildung A.9, in dem die Debug-Fähigkeiten der BDM-Schnittstelle benutzt werden können, um das Programm beispielsweise schrittweise ablaufen zu lassen. Nun kann das Steuergerät ausgeschaltet und von der BDM-Schnittstelle getrennt werden.

A.3.2 Flashen eines TC1796

Um ein selbst geschriebenes Programm auf ein TC1796 zu flashen, sind ein paar Schritte notwendig. Zunächst muss das Programm auf einer der SUN-Rays geschrieben und mit dem passenden Linkerskript für den Flash des TC1796 kompiliert werden. Damit die so erzeugte elf-Datei auf das TriBoard eingespielt werden kann, muss am Labor-PC die virtuelle Maschine debian_usb gestartet werden und das TC1796 über USB an den Labor-PC angeschlossen werden. Nach dem Starten der virtuellen Maschine muss das TC1796 noch dieser verbunden werden.
Alle weiteren Schritte können von einer der SUN-Rays ausgeführt werden. Als erstes wird eine SSH-Sitzung zu der virtuellen Maschine geöffnet und sich als der Benutzer „debian“ eingeloggt. Hierzu kann der Befehl aus Listing A.1 benutzt werden. Das Passwort ist im SVN hinterlegt. Um nun den JTAG-Server innerhalb der virtuellen Maschine

zu starten, muss mit `su` in den Kontext des Benutzers root gewechselt und anschließend die Datei `/root/jtagsrv_1796_USB.sh` ausgeführt werden.

```
ssh debian@129.217.43.45
```

Listing A.1: Befehl um eine SSH-Sitzung auf der virtuellen Maschine aufzubauen

Das eigentliche Flashen des TC1796 erfolgt auf der SUN-Ray mit Hilfe des Programms „tricore-insight“. Dieses Programm kann innerhalb einer Konsole durch den Aufruf von `/opt/trigcc345/bin/tricore-insight` gestartet werden. Nach dem ersten Start muss es für die Verwendung des JTAG-Servers konfiguriert werden. Dies geschieht über den Menüpunkt `File-Target Settings`. Die notwendigen Einstellungen können der Abbildung A.10 entnommen werden.

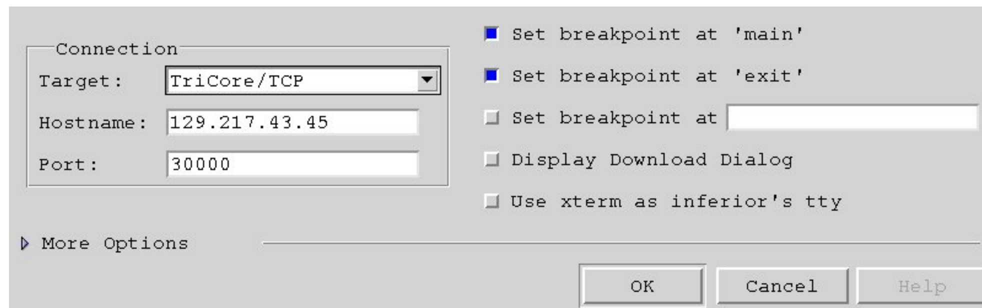


Abbildung A.10: Target-Einstellungen

Damit das eigene Programm in den Flash gespielt werden kann, ist ein Umweg über ein anderes Programm notwendig. Dieses Programm wird auf dem TC1796 ausgeführt und nimmt das zu flashende Programm entgegen. In tricore-insight muss das Programm `flashIntern.elf` aus dem Verzeichnis `triboard_flashen/Tools/Flash/Intern/obj/` geöffnet werden.

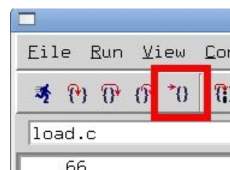


Abbildung A.11: Forsetzen des Programms, nach Erreichen des Breakpoints

Mit einem Klick auf das kleine Männchen in der Symbolleiste, wird das Programm auf dem TC1796 ausgeführt. Es hält dann am Breakpoint der main-Methode an und muss durch einen Klick auf das in Abbildung A.11 markierte Symbol fortgesetzt werden. Dies ist allerdings nur notwendig, wenn bei den Target-Einstellungen der Punkt „Set breakpoint at 'main'“ beibehalten wurde.

Abbildung A.12: Beispiel für Scheinwerfer-Programm

Nach der Fortsetzung des Programms wird eine Eingabe auf der Virtual-IO-Konsole erwartet. Gegebenenfalls muss diese noch über das Menü `View` angezeigt werden. Als Pfad muss der absolute Pfad des zu flashenden Programms eingegeben werden, wie in Abbildung A.12 am Beispiel des Scheinwerfer-Programms gezeigt ist. Nach dem erfolgreichen Flashen erscheint eine Erfolgsmeldung und das TC1796 kann verwendet werden.

A.4 Spannungsversorgung der einzelnen Komponenten

In diesem Abschnitt sind die einzelnen Komponenten mit ihrer benötigten Versorgungsspannung aufgelistet. Zusätzlich ist der Tabelle A.3 eine Empfehlung für den Anschluss der Komponenten an eine geeignete Spannungsquelle zu entnehmen.

Komponente	**Versorgungsspannung**	**empfohlene Spannungsquelle**
Netzteil	220V - 240V	Steckdose
PC	230V	Steckdose
USBprog	5V	USB-Anschluss vom PC
CAN-SPI-Adapter	5V	Anschluss an USBprog
CAN-Gateway	5V	Netzteil des FPGA
AVM-Board	5,5V - 60V	Netzteil oder Steckernetzteil
Pedalboard	5,5V - 60V	Netzteil oder Steckernetzteil
Pedale	3,3V	Anschluss an Pedalboard
Scheinwerferboard	5,5V - 60V	zusammen mit Transistorkästchen
Transistorkästchen	13,8V	Netzteil
Scheinwerfer	13,8V	zusammen mit Transistorkästchen bzw. an Transitorkästechen
Lenksäulenmodul	13,8V	Netzteil
Dachmodul	13,8V	Netzteil
Gangwahlschalter	13,8V	Netzteil

Tabelle A.3: Spannungsversorgung der verschiedenen Komponenten

A.5 CAN-Nachrichten der Experimentierplattform

Die hier aufgelisteten Nachrichten sind komplett der Dokumentation, die im DLR-SchoolLab-SVN enthalten ist, entnommen. Sie werden hier nur der Vollständigkeit halber aufgelistet.

A.5.1 Gangwahlschalter

Ereignisse: Intervall 1 ms - (CAN-ID 200h)

Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:	+		-		Sport	Park	Schaltknopf	Modus
Wert:	0,1,2		0,1,2		0,1	0,1	0,1	0,1

Modus: 1 = Automatik / 0 = Tiptronic
Im Automatik-Modus muss zum Einlegen einer Fahrstufe der Schalthebel ganz nach vorne (R) bzw. ganz nach hinten (D) bewegt werden → ++ bzw. - - (dezimal „2“ im Feld + bzw. -).
Dezimal „1“ bedeutet für das Getriebe die Stufe „Neutral“.
Im Tiptronic-Modus ist nur „einfaches“ Schalten möglich + bzw. - (dezimal „1“ im Feld + bzw. -).

Blockierung Parkmodus: (CAN-ID 201h)

Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:								Block_P
Wert:								0,1

Blockierung Gasse (Automatik → Tiptronic): (CAN-ID 202h)

Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:								Block_G
Wert:								0,1

Sport LED an/aus: (CAN-ID 203h)

Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:								Sport_LED
Wert:								0,1

Park LED an/aus: (CAN-ID 204h)

Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:								P_LED
Wert:								0,1

Neutral LED an/aus: (CAN-ID 205h)

Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:								N_LED
Wert:								0,1

Drive LED an/aus: (CAN-ID 206h)

Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:								D_LED
Wert:								0,1

Reverse LED an/aus: (CAN-ID 207h)

Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:								R_LED
Wert:								0,1

Manual shift LED an/aus: (CAN-ID 208h)

Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:								M/S_LED
Wert:								0,1

Hintergrundbeleuchtung an/aus: (CAN-ID 209h)

Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:								Hintergrund_bel.
Wert:								0,1

A.5.2 Dachmodul

Ereignisse: Intervall 75 ms - (CAN-ID 210h)

Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:		Schieber_ gedrueckt	Schieber			Taste_ rechts	Taste_ mitte	Taste_ links
Wert:		0,1	0,1,2,3,4			0,1	0,1	0,1

Schieber: 1 = vor / 2 = ganz vor / 3 = zurück / 4 = ganz zurück

Leselampe links an/aus: (CAN-ID 211h)

Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:								Leselampe_links
Wert:								0,1

Leselampe rechts an/aus: (CAN-ID 212h)

Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:								Leselampe_rechts
Wert:								0,1

Innenbeleuchtung an/aus: (CAN-ID 213h)

Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:								Innenbel.
Wert:								0,1

LEDs an/aus: (CAN-ID 214h)

Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:								LEDs
Wert:								0,1

A.5.3 Pedaleinheit

Ereignisse: Intervall ca. 50 ms - (CAN-ID 220h)

Byte 0								
Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:	Bremse (in Prozent)							
Wert:	0...100							

Byte 1								
Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:	Gas (in Prozent)							
Wert:	0...100							

A.5.4 Simulation

Ereignisse: Intervall ca. 100 ms - (CAN-ID 290h)

Byte 0								
Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:							Kollision	
Wert:							0,1,2	

Kollision: 0 = links / 1 = rechts / 2 = frontal

Byte 1								
Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:					Gang			
Wert:					-1,0,1,2,3,4,5,6			

Gang: -1 = R / 0 = N / 1 = 1 / 2 = 2 / 3 = 3 / 4 = 4 / 5 = 5 / 6 = 6

Byte 2								
Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:	Geschwindigkeit (in kph) [7:0]							
Wert:	...512							

Byte 3								
Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:							Geschw. (in kph) [9:8]	
Wert:							-511...	

Byte 4								
Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:	Drehzahl (in rpm) [7:0]							
Wert:	...16384							

Byte 5								
Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:					Drehzahl (in rpm) [13:8]			
Wert:					0...			

Getriebe-Steuernachricht: (CAN-ID 291h)

Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:						Gang		
Wert:						1,2,3,4,5,6		

Gang: 1 = N / 2 = D / 3 = R / 4 = M / 5 = Up / 6 = Down

Licht-Steuernachricht: (CAN-ID 292h)

Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:								Licht
Wert:								0,1

Lenkwinkel-Steuernachricht: (CAN-ID 293h)

Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:	Lenkwinkel (in Prozent)							
Wert:	-100...100							

Gang: 1 = N / 2 = D / 3 = R / 4 = M / 5 = Up / 6 = Down

Brems-Steuernachricht: (CAN-ID 294h)

Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:	Bremse (in Prozent)							
Wert:	-100...100							

Gang: 1 = N / 2 = D / 3 = R / 4 = M / 5 = Up / 6 = Down

Motor-Steuernachricht: (CAN-ID 295h)

Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:	Gas (in Prozent)							
Wert:	-100...100							

Gang: 1 = N / 2 = D / 3 = R / 4 = M / 5 = Up / 6 = Down

A.5.5 Scheinwerfer

Ein Teil der CAN-Nachrichten der Scheinwerfer unterliegt der NDA von Audi und ist deshalb nur in der internen, ergänzenden Dokumentation im SVN zu finden. Die hier aufgezählten Nachrichten sind die CAN-Nachrichten, die vom Scheinwerferboard verarbeitet werden.

A.5.5.1 Scheinwerfer links

Hauptscheinwerfer An/Aus: (CAN-ID 231h)

Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:								An/Aus
Wert:								0,1

Fernlicht An/Aus: (CAN-ID 232h)

Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:								Fernl.
Wert:								0,1

Blinker An/Aus: (CAN-ID 233h)

Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:								Blinken
Wert:								0,1

Tagfahrlicht An/Aus + Helligkeit (0 - 15): (CAN-ID 234h)

Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:				Tagfahrl._Helligkeit				Tagfahrl.
Wert:				0...15				0,1

Kurvenlicht: Position (CAN-ID 235h)

Byte 0								
Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:	Sollposition Kurvenlicht [7:0]							
Wert:	...2047							

Byte 1								
Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:					Sollposition Kurvenlicht [11:8]			
Wert:					-2047...			

Position: -x = rechts / +x = links → 0xff0...0x7ff...0x0 (links...geradeaus...rechts)

Leuchtweite: Höhe (CAN-ID 236h)

Byte 0								
Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:	Sollposition Leuchtweitenregulierung [7:0]							
Wert:	...2047							

Byte 1								
Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:					Sollposition Leuchtweitenr. [11:8]			
Wert:					-2047...			

Höhe: -x = Nah / +x = Fern → 0xff0...0x7ff...0x0 (oben...mitte...unten)

A.5.5.2 Scheinwerfer rechts

Hauptscheinwerfer An/Aus: (CAN-ID 241h)

Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:								An/Aus
Wert:								0,1

Fernlicht An/Aus: (CAN-ID 242h)

Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:								Fernl.
Wert:								0,1

Blinker An/Aus: (CAN-ID 243h)

Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:								Blinken
Wert:								0,1

Tagfahrlicht An/Aus + Helligkeit (0 - 15): (CAN-ID 244h)

Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:				Tagfahrl._Helligkeit				Tagfahrl.
Wert:				0...15				0,1

Kurvenlicht: Position (CAN-ID 245h)

Byte 0								
Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:	Sollposition Kurvenlicht [7:0]							
Wert:	...2047							

Byte 1								
Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:					Sollposition Kurvenlicht [11:8]			
Wert:					-2047...			

Position: -x = rechts / +x = links → 0xff0...0x7ff...0x0 (links...geradeaus...rechts)

Leuchtweite: Höhe (CAN-ID 246h)

Byte 0								
Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:	Sollposition Leuchtweitenregulierung [7:0]							
Wert:	...2047							

Byte 1								
Bitposition:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Funktion:					Sollposition Leuchtweitenr. [11:8]			
Wert:					-2047...			

Höhe: -x = Nah / +x = Fern → 0xff0...0x7ff...0x0 (oben...mitte...unten)

A.5.6 Lenksäulenmodul

Die CAN-Nachrichten des Lenksäulenmoduls unterliegen der NDA von Audi und sind deshalb nur in der internen, ergänzenden Dokumentation im SVN zu finden.

A.6 Softwarezuordnung

Diesem Abschnitt ist die Zuordnung der verwendeten Software zu den einzelnen Komponenten zu entnehmen. Die Software ist im DLR-SchoolLab-SVN im Unterordner zu dieser Bachelorarbeit hinterlegt bzw. verlinkt.

Komponente	**Software aus Verzeichnis**
Dachmodul	dachmodul_mit_Senden
Gangwahlschalter	gangwahlschalter
Scheinwerferboard	scheinwerfer_TC1796
Pedalboard	TC1796_pedale_interrupt
AVM-Board	ciaoos_avm
USBprog	usbprog
CAN-Gateway	CAN-Gateway/C_Code

Literaturverzeichnis

[1] PG AutoLab: *PG AutoLab: Abschlussbericht*. http://ess.cs.tu-dortmund.de/Teaching/PGs/autolab/AutoLab-Endbericht.pdf. Version: September 2008

[2] PG CoaCh: *PG CoaCh: Abschlussbericht*. http://ess.cs.tu-dortmund.de/Teaching/PGs/coach/CoaCh-Abschlussbericht.pdf. Version: September 2009

[3] M. Engel ; O. Spinczyk: System-on-chip integration of embedded automotive controllers. In: *Proceedings of the First Workshop on Isolation and Integration in Embedded Systems*, ACM Press, April 2008

[4] Reif, Konrad: *Automobilelektronik*. Vieweg+Teubner, 2009. – ISBN 3834804460

[5] Lavet, Nicolas ; Simonot-Lion, Francoise: *Automotive Embedded Systems Handbook*. CRC Press, 2008. – ISBN 084938026X

[6] Zimmermann, Werner ; Schmidgall, Ralf: *Bussysteme in der Fahrzeugtechnik*. Vieweg & Sohn Verlag, 2007. – ISBN 3834802352

[7] LIN Konsortium: *LIN Specification Package Version 2.1*. November 2006

[8] Wallentowitz, Henning ; Reif, Konrad (Hrsg.): *Handbuch Kraftfahrzeugelektronik*. Vieweg & Sohn Verlag, 2006. – ISBN 352803971X

[9] Grzenba, Andreas ; von der Wense, Hans-Christian: *LIN-Bus*. Franzis Verlag, 2005. – ISBN 3772340091

[10] Robert Bosch GmbH: *CAN Specification Version 2.0*. http://www.semiconductors.bosch.de/pdf/can2spec.pdf. Version: September 1991

[11] Borgeest, Kai: *Elektronik in der Fahrzeugtechnik*. Vieweg+Teubner, 2010. – ISBN 3834805483

[12] Etschberger, Konrad: *Controller-Area-Network : Grundlagen, Protokolle, Bausteine, Anwendungen*. Hanser Verlag, 2002. – ISBN 3446217762

[13] Infineon Technologies: *TC1796 Data Sheet*. http://www.infineon.com/dgdl/TC1796_ds_V10.pdf?folderId=db3a304412b407950112b408e8c90004\&fileId=db3a304412b407950112b41beed22cd3. Version: April 2008

[14] Xilinx, Inc.: *Spartan-3E FPGA Starter Kit Board User Guide*. http://www.xilinx.com/support/documentation/boards_and_kits/ug230.pdf. Version: Juni 2008

[15] embedded projects GmbH : *USBprog - Open-Source Universalwerkzeug*. http://www.embedded-projects.net/index.php?page_id=135

[16] Elektronik-Atelier Kallen: *CAN-SPI-Adapter - Data Sheet*. http://www.avrcard.com/Documents/can/can_spi_datasheet.pdf

[17] Maxim Integrated Products: *MAX3051 - Datasheet*. http://datasheets.maxim-ic.com/en/ds/MAX3051.pdf

[18] Maxim Integrated Products: *MAX3054-MAX3056 - Datasheet*. http://datasheets.maxim-ic.com/en/ds/MAX3054-MAX3056.pdf

[19] PG AutoLab: *PG AutoLab: internes Wiki*. https://ess.cs.tu-dortmund.de/autolab-wiki/

[20] Infineon Technologies: *Datenblatt SPP80P06P*. http://www.datasheetcatalog.org/datasheet/infineon/1-SPP80P06P.pdf

[21] Philips Semiconductors: *Datenblatt BC550C*. http://www.datasheetcatalog.org/datasheet/philips/BC549_550_3.pdf

[22] Homann, Matthias: *OSEK*. mitp-Verlag, 2005. – ISBN 3826615522

[23] Wikipedia: *Inkrementalgeber*. http://de.wikipedia.org/wiki/Inkrementalgeber. Version: Juni 2010

[24] Infineon Technologies: *TC1796 User's Manual System and Peripheral Units, V2.0*. http://www.infineon.com/dgdl/tc1796_um_v2.0_2007_07.pdf?folderId=db3a304412b407950112b41bc4432cb0\&fileId=db3a304412b407950112b41bc4972cb1. Version: Juli 2007

[25] Lohmann, D. ; Hofer, W. ; Schröder-Preikschat, W. ; Erlangen-Nuremberg, FAU ; Streicher, J. ; Spinczyk, O. ; Dortmund, TU: CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In: *Proceedings of the 2009 USENIX Annual Technical Conference*, 2009, S. 215–228

[26] Wikipedia: *Background Debug Mode interface*. http://en.wikipedia.org/wiki/Background_Debug_Mode_interface. Version: März 2010

Abbildungsverzeichnis

2.1 Struktur eines LIN-Bussystems (Quelle: [6] Seite 44) 3
2.2 High-Speed-CAN-Bus (Quelle: [4] Seite 19) 5
2.3 Signalpegel des CAN-Bus (Quelle: [8] Seite 206) 6
2.4 Daten-Frame im Standard-Format 7
2.5 Daten-Frame im Extended-Format 8
2.6 Arbitrierung beim CAN-Bus (Quelle: [11] Seite 93) 11
2.7 FlexRay-Bus in Zwei-Kanal-Linien-Struktur (Quelle: [6] Seite 58) 12
2.8 FlexRay-Bus in Ein-Kanal-Stern-Struktur (Quelle: [6] Seite 58) 13

3.1 Aufbau der Experimentierplattform 15

4.1 Scheinwerfer aus dem Audi A4 19
4.2 Lenksäulenmodul aus dem Audi A8 20
4.3 Dachmodul aus dem BMW X5 20
4.4 Gangwahlschalter aus dem BMW X5 21
4.5 TriBoard TC1796 21
4.6 FPGA Spartan-3E S500 22
4.7 Pedaleinheit vom Saitek R440 Force Feedback Wheel 23
4.8 DELL OPTIPLEX 755 23
4.9 Schnittstelle vom PC zum CAN-Bus 24
4.10 Verwendetes Netzteil 24

5.1 Zuordnung der Anschlüsse zu den CAN-Controllern 25
5.2 Batterieanschluss des Spartan-3E 27
5.3 Menüeintrag PROM File Formatter 28
5.4 Prepare PROM Files Dialog 29
5.5 Erzeugte Konfiguration für die PROM-Datei 30
5.6 Fertig zum Programmieren der dauerhaften Konfiguration 30
5.7 Transistorschaltung der Projektgruppe „AutoLab“ (Quelle: [19] - Abschnitt Transistorkästchen) 31
5.8 Fertiges Transistorkästchen 32
5.9 Unterteilung des Lenkkreises in Segmente 34
5.10 Platine im Inneren der Pedaleinheit 38
5.11 Signale der Pedaleinheit 39
5.12 Ein Teil der Eingänge der External Request Unit (Quelle: [24] Seite 5-14) 40
5.13 grafisches Programmiertool unter Ubuntu 43

A.1 Gesamtübersicht CAN-Gateway 54
A.2 Scheinwerferboard in seiner Kiste 55
A.3 Pedalboard in seiner Kiste mit Anschluss für Pedale 55
A.4 TC1796 mit Virtual Machine in seiner Kiste 56
A.5 Dachmodul in seiner Kiste mit Schalter für Betriebsmodus 56
A.6 Gangwahlschalter in seiner Kiste 57
A.7 Lenksäule in ihrer Kiste 57
A.8 CodeWarrior-Oberfläche zum Programmieren 58
A.9 CodeWarrior-Oberfläche zum Debuggen 59
A.10 Target-Einstellungen 60
A.11 Forsetzen des Programms, nach Erreichen des Breakpoints 60
A.12 Beispiel für Scheinwerfer-Programm 61