



**HOCHSCHULE LANDSHUT**  
HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN

**HOCHSCHULE FÜR ANGEWANDTE  
WISSENSCHAFTEN LANDSHUT**

FAKULTÄT ELEKTROTECHNIK UND WIRTSCHAFTSINGENIEURWESEN

Masterstudium

Elektrotechnik

**Projektbericht**

Eingebettete Autonome Systeme

**Aufbau und Programmierung eines Modellfahrzeugs für den NXP Cup  
Gesamtdokumentation**

vorgelegt von:

Ambrosch Markus, Ecker Christian, Summer Matthias

eingereicht am:

10.07.2021

Betreuer: Prof. Dr. Mathias Rausch

## Vorwort und Projekt-Links

Dieser Bericht umfasst die bei der Durchführung des Projekts „Aufbau und Programmierung eines Modellfahrzeugs für den Next Experience (NXP)-Cup“ über den Zeitraum von zwei Semestern gesammelten Erfahrungen und Ergebnisse. Wir als Projektteam bedanken uns herzlich bei Herrn Prof. Dr. Rausch und Herrn Arne Kullina für die Unterstützung bei diesem Projekt.

Wir freuen uns außerdem darüber, dass wir bei der Innovation Challenge von electromaker.io den dritten Platz mit unserem Projekt belegen konnten. Da Electromaker ein Partner von NXP ist, muss jedes Team ihr Projekt auf der Plattform electromaker.io veröffentlichen, um am NXP-Cup teilnehmen zu können. Besonders gut dokumentierte oder interessante Fahrzeug-Projekte können bei der Innovation Challenge im Vorfeld des NXP-Cups ein Preisgeld von bis zu 1000€ gewinnen.

**Unsere Projektdokumentation auf electromaker.io  
findet sich unter diesem Link:**

<https://www.electromaker.io/project/view/fast-and-furious>

Sowohl die gesamte Dokumentation (Stromlaufpläne, Bilder, Pinbelegungen, 3D-Druck-Dateien, Datenblätter, ...), als auch den Programmcode, haben wir auf GitHub veröffentlicht. Wir hoffen, dass das nächste Team der HAW Landshut (Hochschule für angewandte Wissenschaften Landshut) auf unsere ausführlich dokumentierte Grundlage aufbauen und den NXP-Cup gewinnen wird.

**Der Programmcode und die Projektdokumentation  
finden sich unter diesen Links:**

<https://github.com/CEcker94/nxp-aes-proj>

<https://github.com/CEcker94/nxp-aes-documentation>

## Inhaltsverzeichnis

<b>Vorwort und Projekt-Links</b>	<b>II</b>
<b>Inhaltsverzeichnis</b>	<b>III</b>
<b>Abkürzungsverzeichnis</b>	<b>V</b>
<b>1 Einführung</b>	<b>1</b>
1.1 Zielsetzung . . . . .	1
1.2 Konzept . . . . .	1
<b>2 Aufbau des Fahrzeugs</b>	<b>3</b>
2.1 Grundaufbau . . . . .	3
2.2 Anbaukomponenten aus dem 3D-Druck . . . . .	5
2.2.1 Stoßstange und Ultraschallboard-Halterung . . . . .	5
2.2.2 Akku-Halterung und Seitenschweller . . . . .	7
2.2.3 Halterung der Controllerplatine und des Bedienungsboards . . . . .	9
2.2.4 Halterung für die Motorcontroller . . . . .	12
2.2.5 Einfache Platinenhalterungen . . . . .	13
2.2.6 Kamerahalterung . . . . .	14
2.2.7 Antriebs-Abdeckung . . . . .	15
<b>3 Controllerplatine und ProgrammierTool</b>	<b>17</b>
<b>4 Antrieb des Fahrzeugs</b>	<b>19</b>
4.1 BLDC-Antrieb und Motorcontroller . . . . .	19
4.2 Montage der Antriebskomponenten . . . . .	20
4.3 Konfiguration der Motorcontroller . . . . .	21
4.4 Programmierung des Antriebsbausteins . . . . .	23
4.5 Drehzahlmessung . . . . .	27
4.5.1 Erörterung der Notwendigkeit einer Drehzahlmessung . . . . .	27
4.5.2 Auswahl des Messprinzips . . . . .	30
4.5.3 Hardware für die Drehzahlmessung . . . . .	33
4.5.4 Programmierung des Drehzahlmessungsbausteins . . . . .	38
4.5.5 Ausblick Schaltungserweiterung . . . . .	42
<b>5 Lenkung des Fahrzeugs</b>	<b>44</b>
5.1 Servoantrieb . . . . .	44
5.2 Montage der Lenkungskomponenten . . . . .	45
5.3 Programmierung des Lenkungsbausteins . . . . .	46

<b>6 Bedienungs-Board</b>	<b>49</b>
6.1 Schaltplan . . . . .	49
6.2 Programmierung der Steuerelemente . . . . .	50
6.2.1 Taster auswerten . . . . .	50
6.2.2 Drehencoder auswerten . . . . .	52
6.3 Programmierung der Anzeige . . . . .	53
6.3.1 Funktionsweise des organischen Leuchtdioden-Displays . . . . .	53
6.3.2 I2C Ansteuerung . . . . .	54
6.3.3 Konfiguration des Displays . . . . .	56
6.4 Darstellungen auf dem organischen Leuchtdioden-Display . . . . .	57
6.4.1 Pixelansteuerung . . . . .	57
6.4.2 Rechteck zeichnen . . . . .	57
6.4.3 Buchstabe darstellen . . . . .	58
6.4.4 Text darstellen . . . . .	58
6.4.5 Bilder darstellen . . . . .	58
6.5 Menüführung . . . . .	58
6.6 Aufbau des Human-Machine Interfaces . . . . .	59
6.7 Anleitung zum Hinzufügen eines neuen Menüs . . . . .	60
6.8 Verwendung des Electrically Erasable Programmable Read-Only Memorys . . . . .	63
6.9 Ausblick zur Programmierung des Displays . . . . .	64
<b>7 Streckenerkennung des Fahrzeugs</b>	<b>65</b>
7.1 Kamera des Fahrzeugs . . . . .	65
7.2 Ablauf einer Bildaufnahme . . . . .	67
7.3 Programmierung der zyklischen Bildaufnahme . . . . .	68
<b>8 Verteilerplatinen</b>	<b>77</b>
8.1 Leistungsverteiler auf der unteren Fahrzeugebene . . . . .	77
8.2 Signalverteilerplatine auf der oberen Fahrzeugebene . . . . .	78
<b>9 Zusammenfassung, Fazit und Ausblick</b>	<b>79</b>
9.1 Zusammenfassung und Fazit . . . . .	79
9.2 Ausblick . . . . .	80
<b>Abbildungsverzeichnis</b>	<b>VI</b>
<b>Quellenverzeichnis</b>	<b>IX</b>
<b>Anhang</b>	<b>X</b>

## Abkürzungsverzeichnis

<b>BLDC-Motor</b>	Brushless Direct Current Motor
<b>DC-Motor</b>	Gleichstrommotor
<b>RC</b>	Remote-Control
<b>ESC</b>	Electronic Speed Controller
<b>PWM</b>	Pulsweitenmodulation
<b>NXP</b>	Next Experience
<b>CPU</b>	Central Processing Unit
<b>ABS-Kunststoff</b>	Acrylnitril-Butadien-Styrol Copolymer
<b>SLA-Drucker</b>	3D-Drucker mit Stereolithographie-Verfahren
<b>OLED-Display</b>	organisches Leuchtdioden-Display
<b>I2C</b>	Inter Integrated-Circuit
<b>RAM</b>	Random-Access Memory
<b>DMA</b>	Direct Memory Access
<b>EEPROM</b>	Electrically Erasable Programmable Read-Only Memory
<b>PWL-File</b>	Piecewise Linear Function in LTspice
<b>OPV</b>	Operationsverstärker
<b>TI</b>	Texas Instruments
<b>HMI</b>	Human-Machine Interface
<b>UART</b>	Universal Asynchronous Receiver Transmitter

# 1 Einführung

## 1.1 Zielsetzung

Das Ziel der Projektarbeit ist der Aufbau und die Programmierung eines Modellfahrzeugs für den NXP-Cup mit einem eigens gebauten, selbstfahrenden Fahrzeug. Bei diesem Wettbewerb müssen die im Maßstab 1:18 angefertigten Fahrzeuge einen Parcours in möglichst kurzer Zeit selbstständig durchfahren. Diese werden von Studenten aus Europa, dem mittleren Osten und Afrika auf Grundbasis eines Bausatzes entwickelt. Die Fahrbahn wird von zwei schwarzen Streifen begrenzt, die auf einem weißen Hintergrund aufgebracht sind. Das Fahrzeug muss diese Begrenzungen erkennen und anhand deren Auswertung die Geschwindigkeit und den Lenkwinkel anpassen. Angetrieben wird das Auto mithilfe zweier Brushless Direct Current Motoren (BLDC-Motoren). Die Lenkung wird mittels Servoantrieb und Lenkgestänge realisiert. Auf dem Fahrzeug dürfen beliebig viele Prozessoren und Bauteile von NXP Semiconductors, welche den Wettbewerb ausrichten, verwendet werden [Wil20]. Sind für das eigene Fahrzeug benötigte Komponenten nicht im Portfolio von NXP Semiconductors vorzufinden, können auch eigene oder die Produkte anderer Hersteller verwendet werden.

## 1.2 Konzept

Zu Beginn soll das Fahrzeug mit dem Standardbausatz zusammengebaut werden. Die Software wird aus bereits vorhandenen, vorherigen Projekten zusammengesetzt und optimiert. Ein großes Augenmerk liegt dabei auf einem übersichtlicheren Aufbau des Programms und besserer Nachvollziehbarkeit durch Kommentation und eingängigerer Benennung der Funktionen und Parameter.

Die Programmierung soll Stück für Stück vorgenommen werden. Zu Beginn wird die Ansteuerung der BLDC-Motoren bearbeitet. Ist der Punkt erreicht, an dem die Motoren angesteuert und die Drehzahl über die Puls-Weiten-Modulation variiert werden kann, soll eine Möglichkeit der Drehzahlerkennung erarbeitet werden. Im Anschluss an die Inbetriebnahme der Antriebe wird die Software für den Servomotor der Lenkung erstellt. Die Bedienung des Fahrzeugs soll über ein organisches Leuchtdioden-Display (OLED-Display), einen Drehencoder und einen Taster realisiert werden. Zur Streckenerkennung wird eine Zeilenkamera verwendet. Bei Bedarf kann zusätzlich oder ersatzweise eine größere Kamera eingesetzt werden. Nach der Inbetriebnahme der Einzelkomponenten werden deren Ansteuerung und Auswertung mithilfe einer Regelung verknüpft. Die Software des Fahrzeugs soll am Ende so optimiert werden, dass es schnellstmöglich, aber auch sicher durch den Parcours fährt.

Nach der erfolgreichen Entwicklung des Fahrzeugs für das Durchfahren des Parcours soll für eine zusätzliche Wettbewerbsdisziplin eine Objektdetektion mithilfe eines Ultraschallboards realisiert werden, welches bereits in den vorherigen Semestern von anderen Studierenden entwickelt wurde. Mithilfe dieses Ultraschall-Bords soll das Fahrzeug Hindernisse

erkennen und um diese herumfahren können.

Karosserieteile, wie beispielsweise eine Akkuhalterung oder eine Stoßstange mit der Möglichkeit zur Befestigung des Ultraschallboards, werden mit einem 3D-Druck-Verfahren erstellt. Dafür werden vor allem Teile gedruckt, welche bereits von Herrn Arne Kullina im Rahmen seines Praktikums bei Herrn Prof. Dr. Mathias Rausch im Sommersemester 2020 konstruiert wurden.

Für eine übersichtlichere Kabelführung und einfachere De- und Montage werden zwei Verteilerplatinen (Lochrasterplatinen) mit Steck- und Schraubkontakte erstellt, an welchen die Einzelkomponenten angeschlossen werden.

## 2 Aufbau des Fahrzeugs

### 2.1 Grundaufbau

Das Fahrzeug besteht zum größten Teil aus einem Standarbausatz. Auf der unteren Ebene, der Grundplatte, sind die BLDC-Motoren für den Antrieb, der Servomotor und die Reifen montiert (siehe Abbildung 1). In einem weiteren Schritt werden hier auch die Anbauteile des Fahrzeugs aus dem 3D-Druck befestigt, wie beispielsweise die Stoßstange und die Seitenschweller (näheres in Kapitel 2.2).

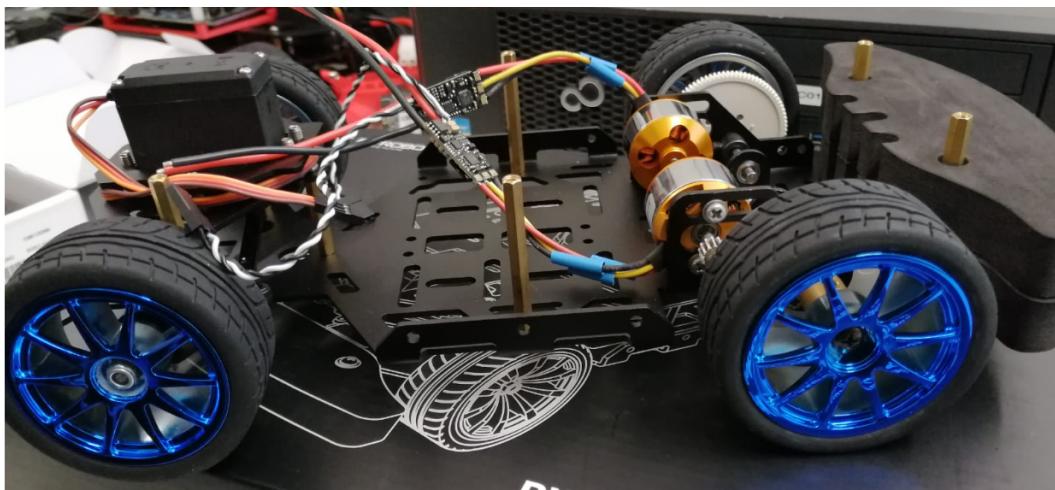


Abbildung 1: Grundplatte des Standardbausatzes für das Fahrzeug mit BLDC-Motoren, Motorcontrollern, Servomotor und Reifen

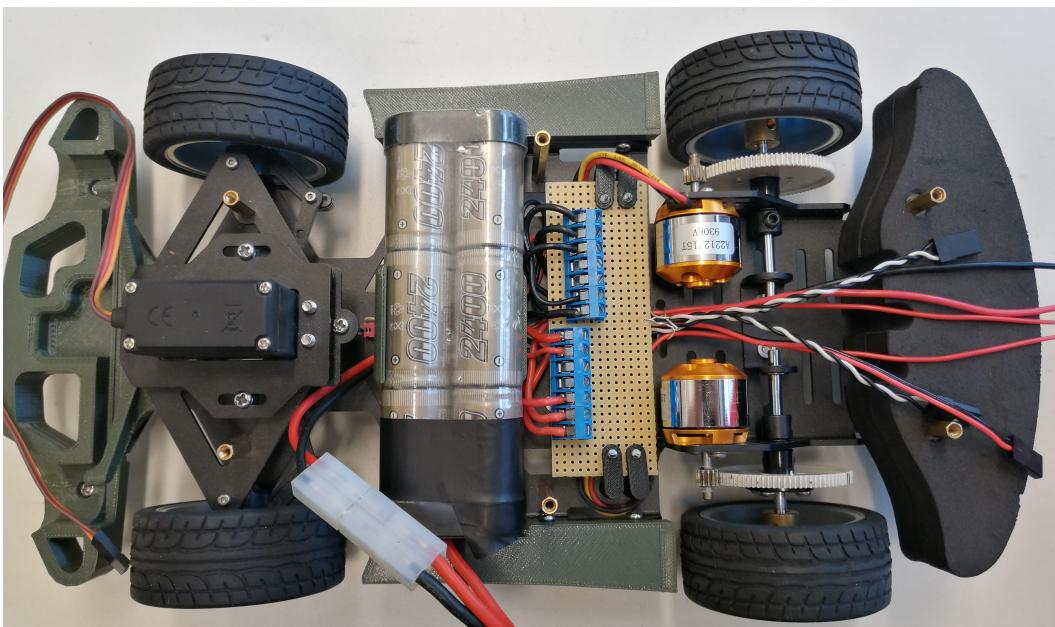


Abbildung 2: Grundplatte des Fahrzeugs mit Akku, Strom-Verteilerplatine und bereits montierter Fahrzeug-Peripherie

Auf der Grundplatte ist zusätzlich eine Verteilerplatine für die Batteriespannung montiert. Auch der Akku findet auf dieser Ebene seinen Platz (siehe Abbildung 2). Die Steckmöglichkeiten für alle Komponenten des Fahrzeugs werden, anders als bei Herrn Kullinas Fahrzeug, nicht auf der Grundplatte realisiert, sondern auf der oberen Ebene. Dass auf der Grundplatte nur noch Komponenten Platz finden, zu denen man selten Zugang benötigt, hat den Vorteil, dass man das Fahrzeug nur noch in Ausnahmefällen auseinandernehmen muss.

Oberhalb der Grundplatte des Fahrzeugs, auf der die Antriebe, die Lenkung, die 3D-Druck Anbaukomponenten und der Akku platziert sind, wird mit einer weiteren Montageplatte eine zweite Ebene aufgespannt (siehe Abbildung 3). Auf dieser oberen Ebene werden der Controller mit dem Bedienungsboard, die Kamera und die Verteilerplatine mit Steckmöglichkeiten für die Fahrzeugkomponenten montiert (siehe Abbildung 4).



Abbildung 3: Obere Ebene des Standardbausatzes für das Fahrzeug ohne Montage des Controllers und der Kamera

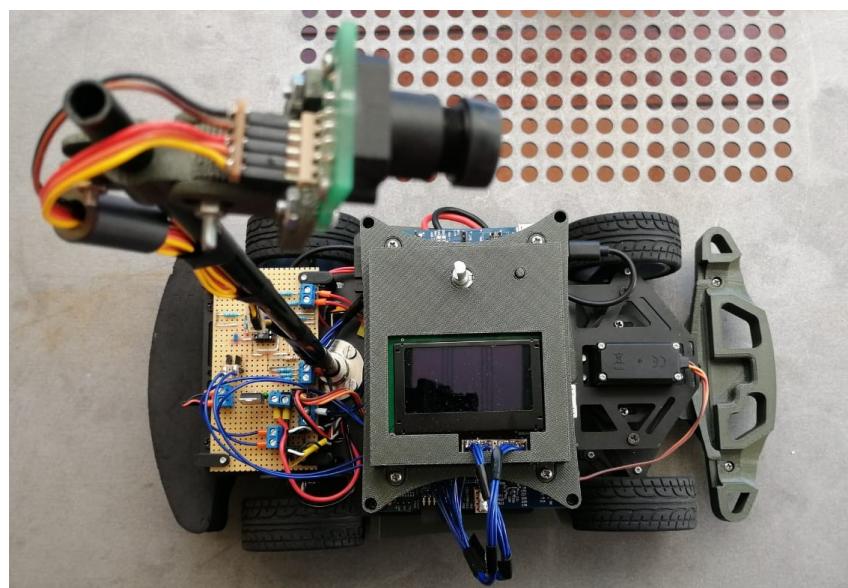


Abbildung 4: Obere Ebene des Fahrzeugs mit Controller, Bedienungsboard, Kamera und Verteilerplatine

## 2.2 Anbaukomponenten aus dem 3D-Druck

Zusätzlich zum Standardbausatz werden auch Anbaukomponenten verbaut, die mithilfe eines 3D-Druckers gefertigt und dann am Fahrzeug angebracht werden. Die Folgekapitel zeigen alle gedruckten Einzelteile und erläutern deren Zweck näher. Die Anbauteile wurden größtenteils von Herrn Arne Kulinna konstruiert, der im Rahmen eines Praktikums bei Herrn Prof. Dr. Mathias Rausch an der HAW Landshut bereits eine erste Version des Fahrzeugs mit demselben Bausatz erstellt hat.

### 2.2.1 Stoßstange und Ultraschallboard-Halterung

Das Fahrzeug benötigt eine Stoßstange, damit bei der Kollision mit einem Hindernis kein wichtiges Teil des Fahrzeugs, wie beispielsweise der Servo-Motor oder das Ultraschall-Board, beschädigt wird. Da das zur Hinderniserkennung zu verwendende Ultraschallboard frontal befestigt wird, müssen in der Stoßstange kegelförmige Aussparungen eingeplant werden, damit die Stoßstange nicht fälschlicherweise als Hindernis erkannt wird. In den Abbildungen 5 und 6 sind die Konstruktionsbilder der beiden Stoßstangenteile abgebildet.

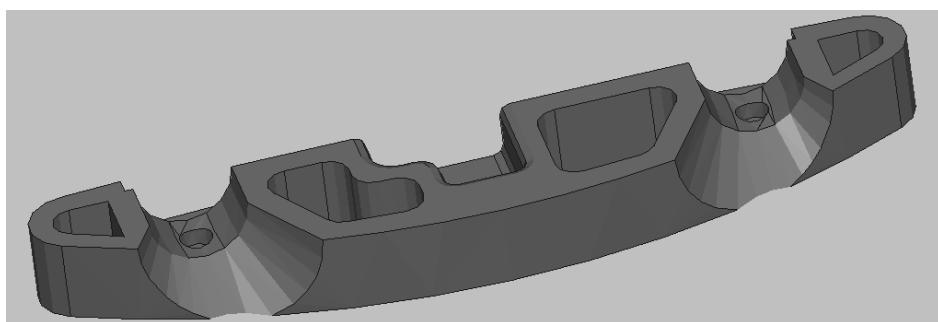


Abbildung 5: Konstruktionsbild des oberen Stoßstangenteils

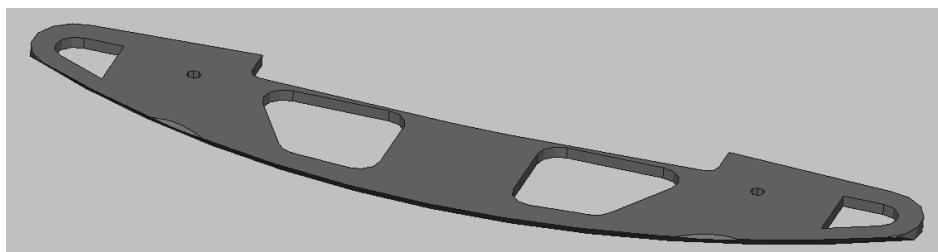


Abbildung 6: Konstruktionsbild des unteren Stoßstangenteils

Abbildung 7 zeigt die fertig gedruckten Stoßstangenkomponenten. Der untere, flache Teil der Stoßstange wird mit zwei Schrauben an der oberen Komponente befestigt. Die gesamte Stoßstange wird ebenfalls an nur zwei Stellen mit der Karosserie verbunden. Zusätzlichen Halt bekommt die Stoßstange von der Halterung des Ultraschallboards.



Abbildung 7: Gedrucktes oberes und unteres Stoßstangenteil

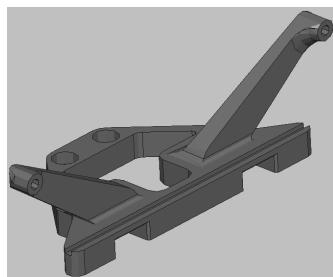


Abbildung 8: Konstruktionsbild der Halterung des Ultraschallboards



Abbildung 9: Gedruckte Halterung des Ultraschallboards

Wie bereits erwähnt, wird das Ultraschallboard frontal am Fahrzeug montiert. Dazu wird eine Aufnahme benötigt, an welcher das Board befestigt wird. In den Abbildungen 8 und 9 sind die Konstruktionszeichnung und die fertig gedruckte Halterung abgebildet.

In den Abbildung 10 und 11 sind die am Fahrzeug fertig montierte Stoßstange und Ultraschallboard-Halterung abgebildet. Die Stellen, an denen diese Bauteile am Fahrzeug fixiert sind, sind in den Abbildungen farblich hervorgehoben.

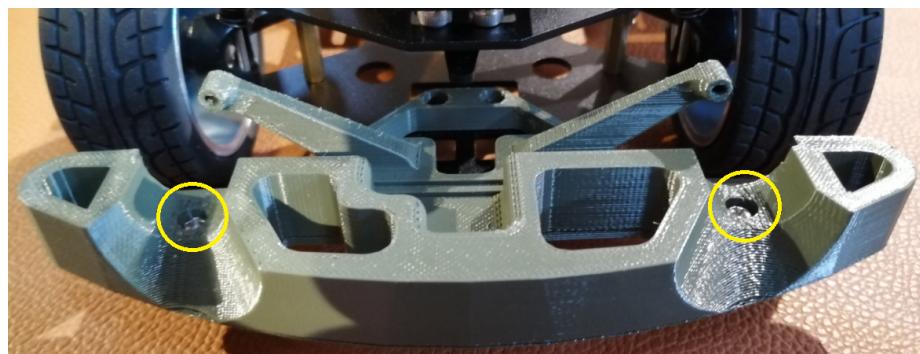


Abbildung 10: Draufsicht der Stoßstange und der Ultraschallboard-Halterung; Befestigungsschrauben der unteren Stoßstangenkomponente in gelb



Abbildung 11: Untersicht der Stoßstange und der Ultraschallboard-Halterung; Befestigungsschrauben der Stoßstange in blau, Schrauben der Ultraschallboard-Halterung in rot und Schrauben zur Befestigung der unteren Stoßstangenkomponente in gelb

### 2.2.2 Akku-Halterung und Seitenschweller

Der Akku für das Fahrzeug soll auf der unteren Ebene Platz finden, um den Schwerpunkt des Fahrzeugs niedrig zu halten. Damit der Akku einen festen Sitz hat und nicht beim Beschleunigen oder Bremsen verrutscht, wird auf der unteren Ebene in der Mitte eine Halterung installiert (Konstruktionszeichnung in Abbildung 12). Die Akku-Halterung fungiert auch als Kabeldurchführung, damit die Drähte und Kabel gebündelt werden können und nicht frei in der Luft geführt werden. In Abbildung 13 ist die fertig gedruckte Akku-Halterung zu sehen. Die Halterung wird von unten mit vier Schrauben auf der Grundplatte des Fahrzeugs montiert.

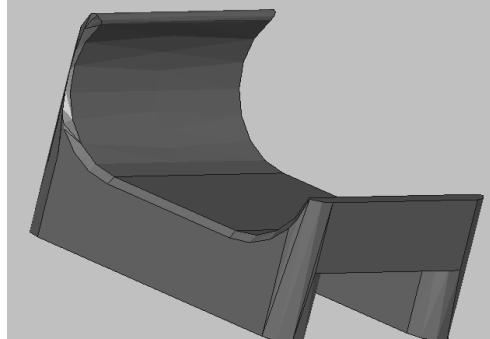


Abbildung 12: Konstruktionsbild der Akku-Halterung mit Kabeldurchführung

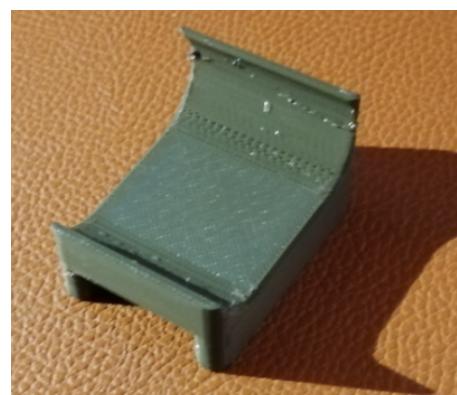


Abbildung 13: Gedruckte Akku-Halterung

Während die Akku-Halterung den Akku vor dem nach vorne und hinten rutschen schützt, tragen kleine Aussparungen an den Innenseiten der Seitenschweller zur Sicherung des Akkus zu beiden Seiten bei (Konstruktionszeichnung in Abbildung 14). Sie werden mit je drei Schrauben

an der Grundplatte des Fahrzeugs befestigt. Die fertig gedruckten Seitenschweller sind in Abbildung 15 abgebildet.

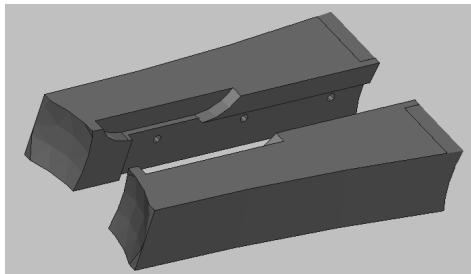


Abbildung 14: Konstruktionsbild der Seitenschweller, welche auch als Sicherung des Akkus zu den Seiten dienen



Abbildung 15: Gedruckte Seitenschweller

In Abbildung 16 sind die zum Fixieren des Akkus notwendigen Teile, die Seitenschweller und die Akku-Halterung, am Fahrzeug fertig montiert, abgebildet. Der Akku wird quer zur Fahrtrichtung eingesetzt. So bleibt hinter dem Akku noch Platz für eine Verteilerplatine, an der die elektrischen Anschlüsse der verschiedenen Fahrzeugkomponenten angesteckt werden können.

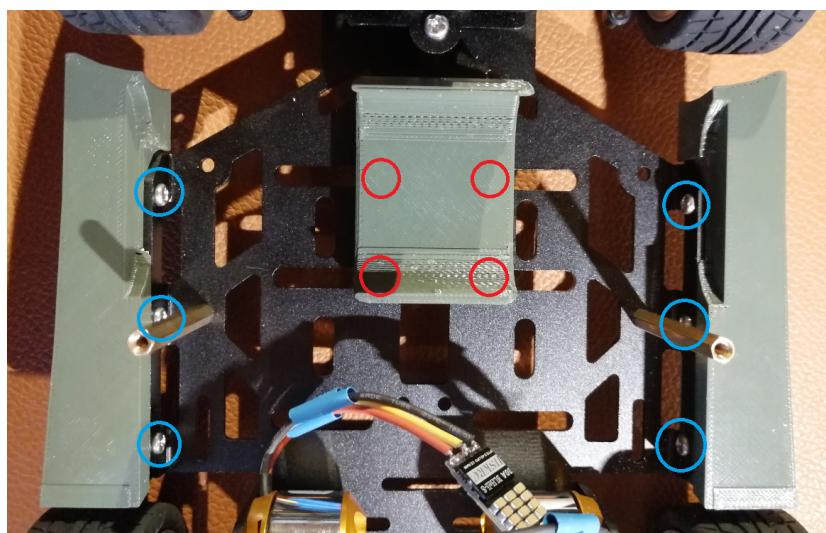


Abbildung 16: Montierte Akku-Halterung und Seitenschweller; Befestigungsschrauben der Seitenschweller in blau und Befestigungsschrauben der Akkuhalterung in rot (von unten, nicht sichtbar)

### 2.2.3 Halterung der Controllerplatine und des Bedienungsboards

Die Controllerplatine wird auf der oberen Fahrzeugplattform befestigt. Dafür wird eine Controller-Halterung benötigt (Konstruktionsbild siehe Abbildung 17). Oberhalb des Controllers findet das Display mit Taster und Drehencoder für die Menüsteuerung Platz (Bedienungsboard). Zusätzlich zur Halterung der Platine (Konstruktionsbild siehe Abbildung 18) ist eine Abdeckung notwendig (Konstruktionsbild siehe Abbildung 19).

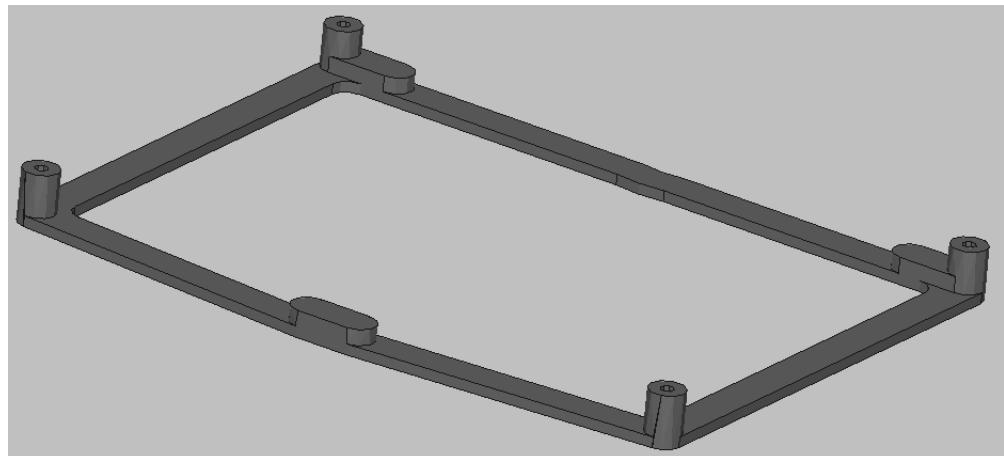


Abbildung 17: Konstruktionsbild der Controller-Halterung

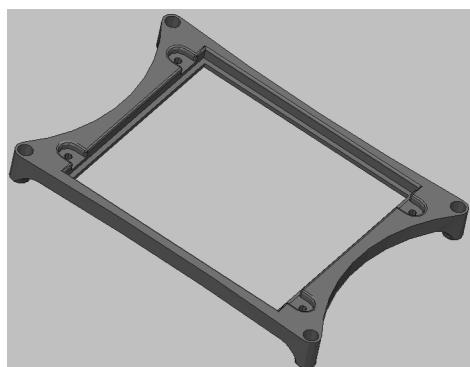


Abbildung 18: Konstruktionsbild der Platinen-Halterung des Bedienungsboards

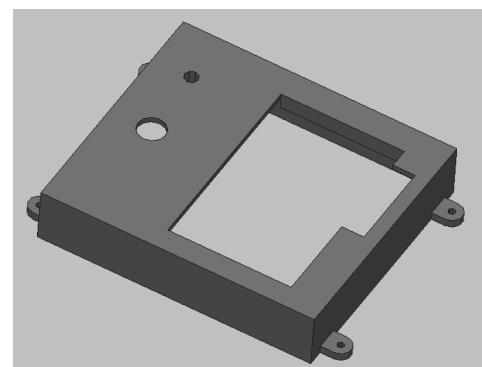


Abbildung 19: Konstruktionsbild der Abdeckung des Bedienungsboards

Zum Bedienen des Tasters auf der Platine wird eine Verlängerung benötigt, damit der Knopf außerhalb der Abdeckung betätigt werden kann (Konstruktionszeichnung siehe Abbildung 20). In den Abbildungen 21 bis 22 sind die fertig gedruckten Teile, die Controller-Halterung, die Platinen-Halterung und die Abdeckung, abgebildet.

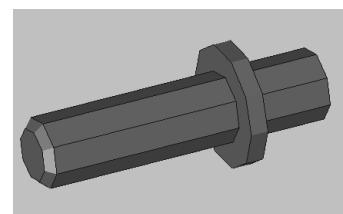


Abbildung 20: Konstruktionsbild der Taster-Verlängerung



Abbildung 21: Gedruckte Controller-Halterung



Abbildung 22: Gedruckte Taster-Verlängerung

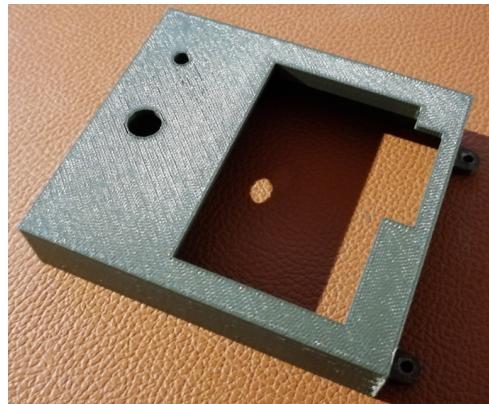


Abbildung 23: Gedruckte Platinen-Abdeckung des Bedienungsboards



Abbildung 24: Gedruckte Platinen-Halterung für das Bedienungsboard

Die Montage der Teile erfolgt, wie bereits erwähnt, auf der oberen Fahrzeugebene. In den Abbildungen 25 und 26 sind die fertig montierten Komponenten für die Befestigung des Controllers und die der Platine für die Fahrzeugbedienung abgebildet. Damit für die Komponenten auf der Controllerplatine ausreichend Platz ist, wird die Platinen-Halterung des Bedienungsboards über Abstandshalterungen montiert. Die Befestigungsschrauben für den Controller sind in den Abbildungen in rot, die der Platinenhalterung in blau und die der Abdeckung in gelb hervorgehoben.

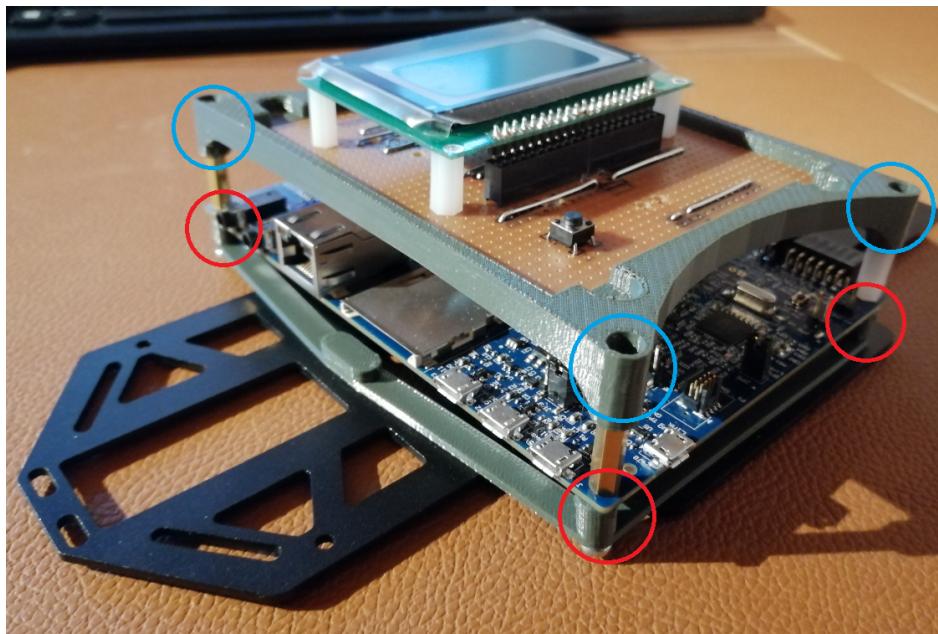


Abbildung 25: Fertig montierte Controller- und Bedienungsboard-Halterung; Befestigungsschrauben des Controllers in rot, Befestigung der Platinen-Halterung in blau

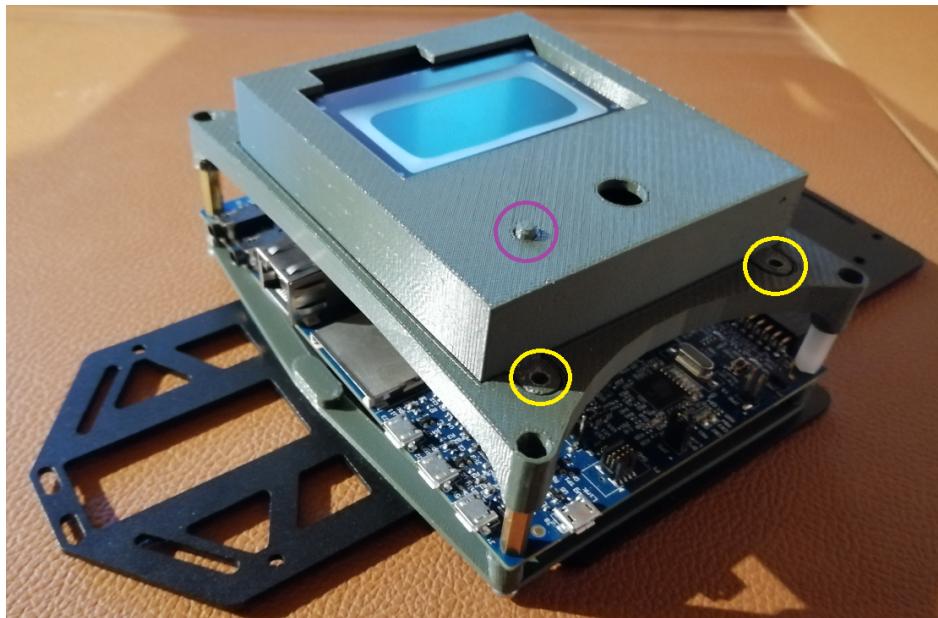


Abbildung 26: Fertig montierte Controller- und Bedienungsboard-Halterung mit Abdeckung; Befestigungsschrauben der Abdeckung in gelb, Taster in violett

### 2.2.4 Halterung für die Motorcontroller

Die Motorcontroller, die vor die BLDC-Motoren geschaltet sind, werden auf der Grundplatte befestigt. Mit zwei dreiecksförmigen Halterungen werden die Motorcontroller an jeweils drei Stellen an der Grundplatte angeschraubt. Die Konstruktionszeichnung einer solchen Motorcontroller-Halterung ist in Abbildung 27 und das Druckergebnis in Abbildung 28 einsehbar.

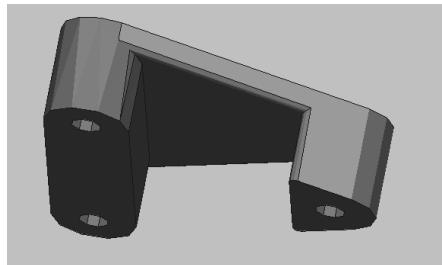


Abbildung 27: Konstruktionsbild einer Motorcontroller-Halterung



Abbildung 28: Fertig Gedruckte Motorcontroller-Halterungen

Abbildung 29 zeigt die auf der Grundplatte montierten Motorcontroller-Halterungen. Die Controller sind am Boden fixiert und durch den transparenten Schrumpfschlauch vor Kurzschlüssen über die metallische Grundplatte geschützt.

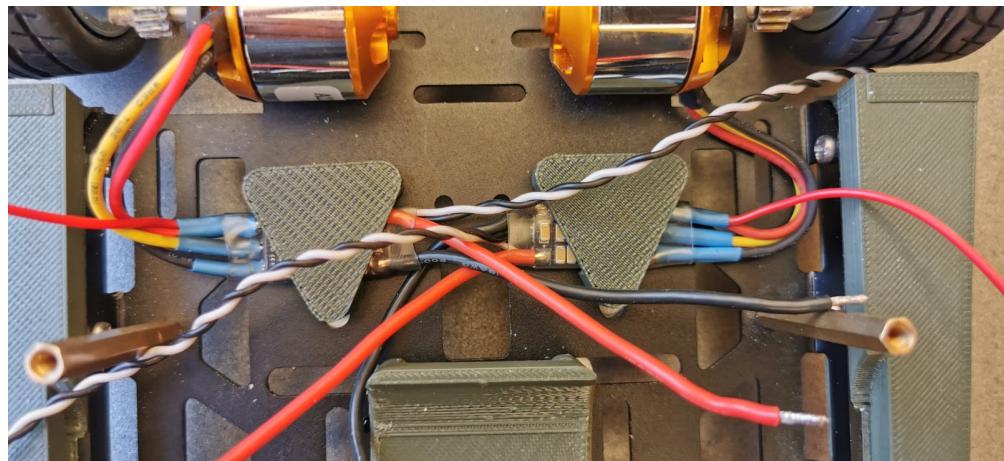


Abbildung 29: Die auf der Grundplatte montierten Motorcontroller

### 2.2.5 Einfache Platinenhalterungen

Für die Befestigung der Verteiler-Platten auf der oberen und unteren Ebene wird eine Halterung benötigt. Mit vier von diesen Halterungen (Konstruktionsbild siehe Abbildung 30) können die Lochrasterplatten an vier Stellen mit etwas Abstand auf dem Fahrzeug befestigt werden. Die fertig gedruckten Platinen-Halterungen sind in Abbildung 31 abgebildet.

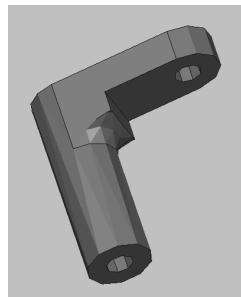


Abbildung 30: Konstruktionsbild einer einfachen Platinenhalterung



Abbildung 31: Gedruckte Platinenhalterungen

In Abbildung 32 ist die fertig montierte Verteiler-Platine sichtbar. Sie ist mit vier einfachen Platinen-Halterungen an der Grundplatte des Fahrzeugs montiert. Der aufgrund der Halterungen resultierende Abstand zur Grundplatte ermöglicht die Montage der Motorcontroller-Halterungen unter der Verteiler-Platine.

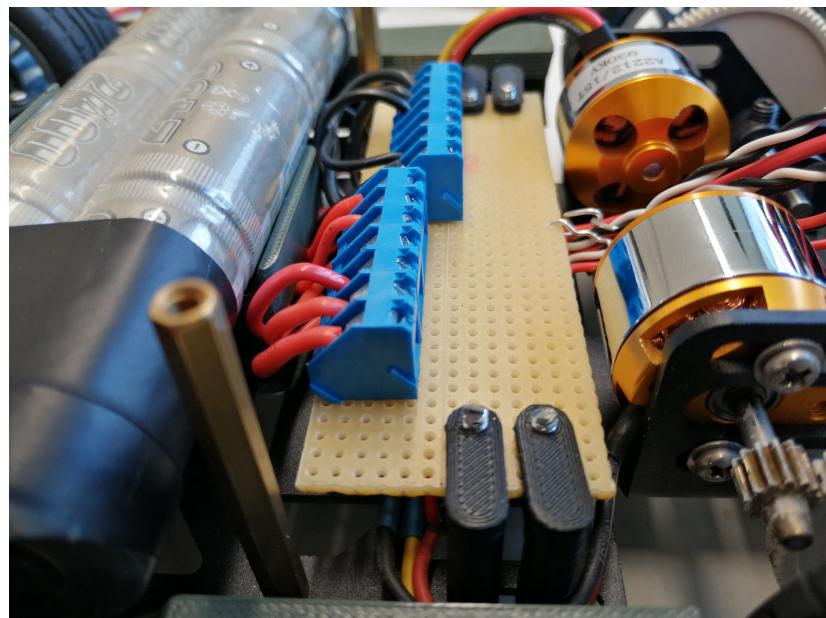


Abbildung 32: Mit den einfachen Platinenhalterungen montierte Leistungsverteilerplatine

### 2.2.6 Kamerahalterung

Die Kamera soll sich bei Kollisionen nicht in ihrer Position verstellen können. Deshalb wird eine eigens dafür konstruierte Halterung gedruckt. Im Gegensatz zu den bisherigen 3D-Druckteilen sind die Kamerahalterung und die Antriebsabdeckung (siehe Kapitel 2.2.7) selbst konstruiert und nicht vom Vorgänger übernommen. Die Konstruktionsbilder der beiden Einzelteile der Kamerahalterung sind in den Abbildungen 33 und 34 einsehbar. Die ineinander greifenden Zacken garantieren dabei eine stabile Kameraposition.

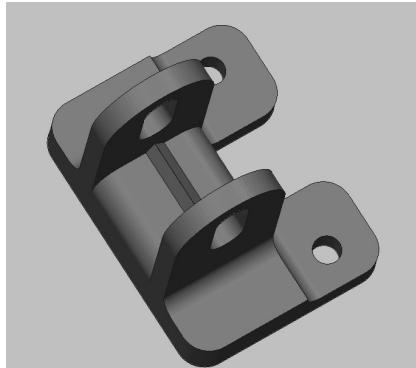


Abbildung 33: Kameramontage-Komponente der Kamera-Halterung

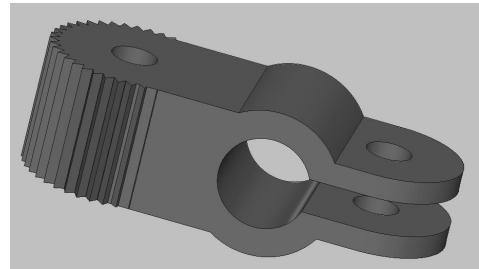


Abbildung 34: Stangenmontage-Komponente der Kamera-Halterung

Schon bei der Konstruktion zeichnet sich ein potentielles Problem für den Druck der Teile ab. Die Verzahnung ist sehr fein gewählt, um die Kamera auch genau einzustellen zu können. Für einen 3D-Drucker kann das bedeuten, dass dieser an seine Grenzen in der Genauigkeit der Fertigung kommt. Die Sorge, dass der verwendete 3D-Drucker, welcher Acrylnitril-Butadien-Styrol Copolymer (ABS-Kunststoff) verwendet, die Verzahnung nicht fein genug fertigen kann, ist zum Teil begründet, da von je zwei gedruckten Teilen eines jeder Komponente von ungenügender Genauigkeit ist. Die beiden in ausreichender Qualität gefertigten Teile sind in Abbildung 35 abgebildet. Mit der Erwartung, genauere Ergebnisse zu erzielen, müsste auf den 3D-Drucker mit Stereolithographie-Verfahren (SLA-Drucker) der Fakultät Maschinenbau zurückgegriffen werden. In diesem Fall ist das allerdings nicht notwendig.

Bei der Montage taucht allerdings ein anderes, unerwartetes Problem auf. Bei der Konstruktion der Teile wurde an einer Stelle entweder ein falsches Maß aufgenommen oder das Maß falsch in die CAD-Zeichnung eingegeben. Die betreffende Stelle ist die Freifläche zwischen den beiden Befestigungsstellen der Kamera, welche um etwa 1,5mm zu kurz geraten ist. Nach einer Anpassung mit einer Säge sind die Teile ohne Probleme verwendbar und die Komponente muss nicht abermals gedruckt werden. Zur Vermeidung dieser Anpassung für Nachfolgemodelle des Fahrzeugs ist die Konstruktionszeichnung angepasst.

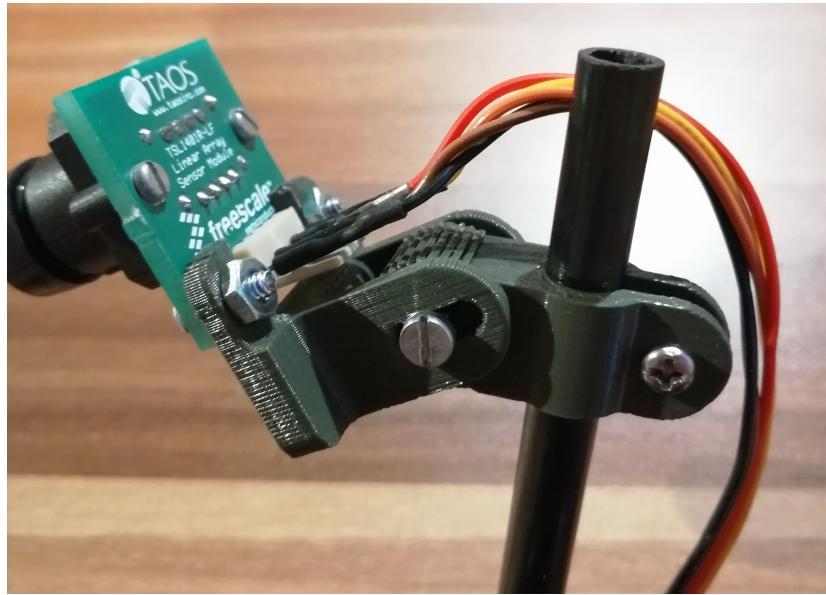


Abbildung 35: Fertig montierte Kamera mit Kamerahalterung

### 2.2.7 Antriebs-Abdeckung

Da die Kabel der Komponenten auf der Grundplatte, welche auf der oberen Ebene angesteckt werden, nicht mittig zwischen den beiden Antrieben auf die obere Ebene durchgeführt werden können, weil dort der Sockel der Kamerastange Platz findet, müssen die Kabel direkt über den Antrieben durchgeführt werden. Das führt allerdings dazu, dass sich die Kabelisolierungen durch die Reibung an den sich drehenden Antrieben über die Zeit abtragen können. Aus diesem Grund wird eine Antriebs-Abdeckung benötigt. Das Konstruktionsbild und das der fertig gedruckten Antriebsabdeckung sind in den Abbildungen 36 und 37 einsehbar. Die fertig montierte Antriebsabdeckung ist in Abbildung 38 abgebildet.

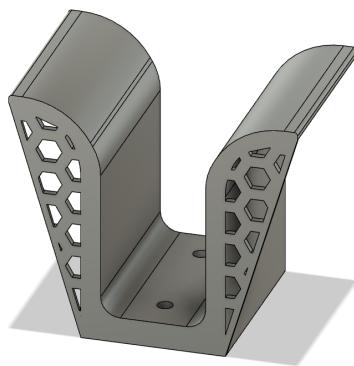


Abbildung 36: Konstruktionsbild der Antriebsabdeckung



Abbildung 37: Gedruckte Abdeckung der Antriebe

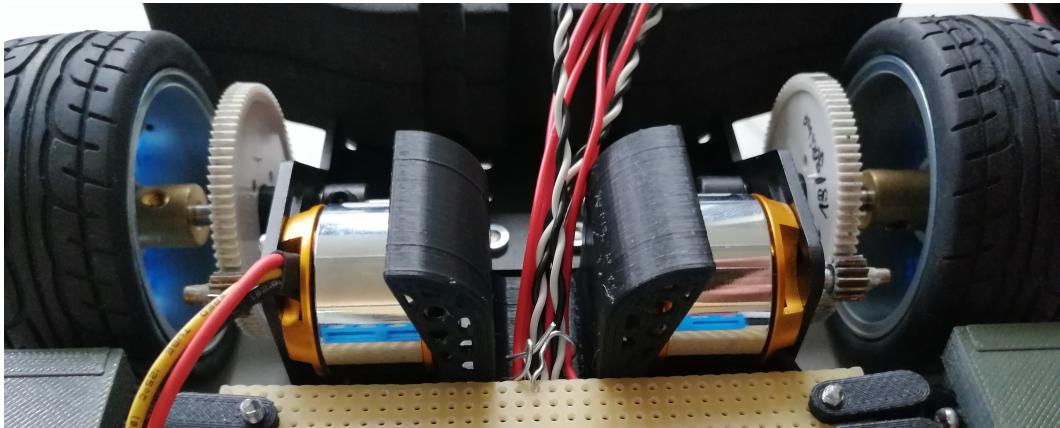


Abbildung 38: Fertig montierte Abdeckung der Antriebe zur Verhinderung von Reibung zwischen den Antrieben und den Kabeln, die nach oben durchgeführt werden

### 3 Controllerplatine und Programmiertool

Für dieses Projekt wird die Controllerplatine LPCXpresso54628 von NXP Semiconductors verwendet. Das Herzstück dieser Platine ist der auf dem ARM Cortex-M4 basierende Mikrocontroller LPC54628. Dieser Controller wird wegen seiner schnellen Taktfrequenz und seiner großen Anzahl an Peripherie für die Ansteuerung der Motoren und des Servos sowie für die Sensorik zur Bestimmung des Streckenverlaufs verwendet. Der Vorteil einer fertigen Controllerplatine liegt darin, dass der Schaltplan sowie das dazugehörige Layout professionell erstellt sind, wodurch dieser aufwendige Schritt bei der Entwicklung des Autos entfällt und die einwandfreie Funktionsweise des Controllers sichergestellt wird. Außerdem ist darauf bereits ein Debugger zur Programmierung und Fehlersuche verbaut. Durch das Flashen dieses Debuggers über das Programm „LPCScrypt\_2.1.2\_57“ kann zwischen dem J-Link Debugger (Standardinstallationspfad + scripts/program\_JLINK.cmd) und dem NXP-eigenen LinkServer Debugger (Standardinstallationspfad + scripts/program\_CMSIS.cmd) gewählt werden.

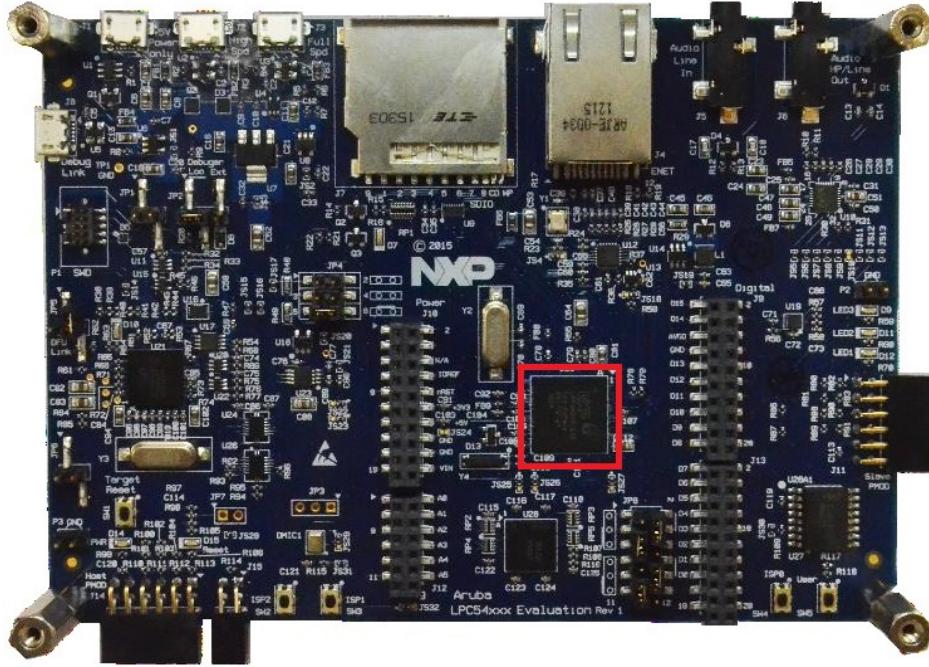


Abbildung 39: Draufsicht auf die Controllerplatine mit dem Mikrocontroller LPC54628 (in rot hervorgehoben) als Hauptbestandteil [Sem19]

Zur Programmierung des Controllers dient das auf Eclipse basierende Tool „MCUXpresso“, welches von NXP Semiconductors zur Verfügung gestellt wird. Darin sind die für den verwendeten Controller notwendigen Konfigurationsdateien, wie beispielsweise das Startup-Skript und einige Beispiele, bereits enthalten. Das erleichtert den Einstieg in die Programmierung erheblich. Außerdem kann das Konfigurationstool „MCUXpresso Config Tools“ verwendet werden. Mit diesem kann die Initialisierung der Peripheriebausteine über eine graphische Benutzeroberfläche vorgenommen werden.

Die Pinbelegungen für die in dieser Fahrzeugversion verwendeten Peripherie (Antriebe, Servo usw.) sind im Anhang „Anhang 2: Pin Assignment“ einsehbar.

## 4 Antrieb des Fahrzeugs

Wie in Kapitel 2.1 beschrieben, wird das Fahrzeug von zwei BLDC-Motoren angetrieben, die im Folgenden genauer erklärt werden. Zusätzlich dazu wird in diesem Kapitel auch näher auf die Montage der Antriebskomponenten, die Programmierung des Antriebsbausteins, die Konfiguration der Motorcontroller und die Drehzahlmessung eingegangen.

### 4.1 BLDC-Antrieb und Motorcontroller

BLDC-Motoren sind im Wesentlichen wie permanent erregte Synchronmaschinen aufgebaut. Sie besitzen Magnete im Rotor und Einzelzahnwicklungen im Stator. Da solche Motoren häufig im Remote-Control (RC)-Bereich Einsatz finden, gibt es zu deren Ansteuerung bereits vorgefertigte Bausteine, sogenannte Electronic Speed Controller (ESC). Diese haben zwei Signalpins für ein Pulsweitenmodulation (PWM)-Signal und zwei Anschlüsse zur Spannungsversorgung des Motorcontrollers, aus welchen die drei Strangspannungen für die Phasen des Antriebs generiert werden. Aus diesen Strangspannungen kann im übrigen auch die Motordrehzahl ermittelt werden, was im Rahmen dieser Projektarbeit auch eines der Ziele darstellt. An den Signalpins wird ein PWM-Signal mit einer Frequenz von 50Hz angelegt, dessen Pulsbreite zwischen 1ms und 2ms liegen darf. Daraus resultiert ein Tastgrad zwischen 5% (Stillstand) und 10% (maximal erreichbare Drehzahl). Die maximale Drehzahl des Antriebs hängt von der Spannung des Akkus ab. Je geringer diese Spannung ist, desto geringer ist die maximal erreichbare Drehzahl. Der Anschluss der Motoren erfolgt nach der in Abbildung 40 gezeigten Weise.

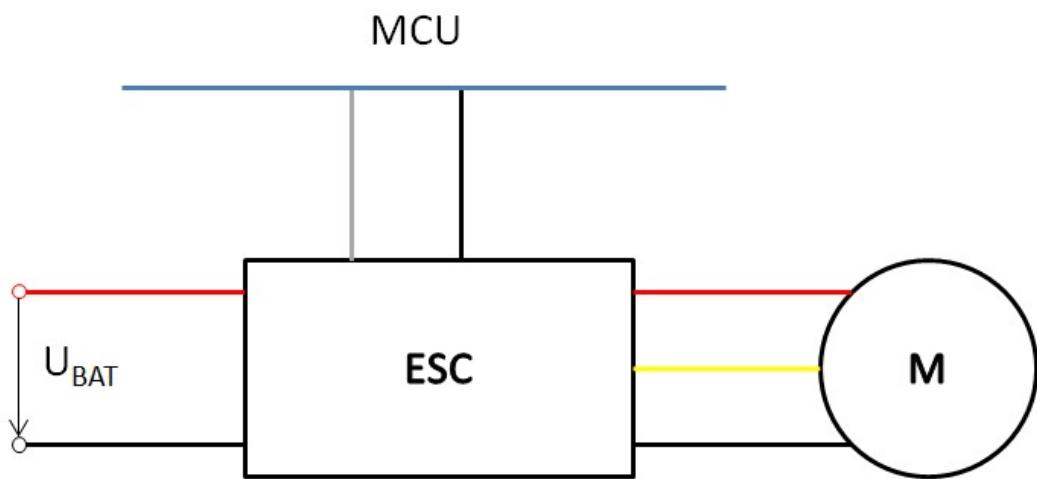


Abbildung 40: Skizze zur Beschaltung eines ESCs und BLDC-Motors; Versorgungsspannung des ESCs in rot und schwarz (Masse), PWM-Signalleitungen in grau und schwarz (Masse) und Anschlussleitungen des BLDC-Motors in rot, gelb und schwarz

## 4.2 Montage der Antriebskomponenten

Die Komponenten des Fahrzeugantriebs, welcher über zwei BLDC-Motoren realisiert ist, sind auf der unteren Fahrzeugebene montiert. Die Motoren selbst sind mit zwei Schrauben an der Karosserie befestigt. An der Welle der Motoren befindet sich je ein Zahnrad mit 13 Zähnen, welches ein weiteres Zahnrad mit 90 Zähnen antreibt, das an der Antriebswelle befestigt ist. Das heißt, dass sich die Reifen bei 6,923 Umdrehungen der Motorwelle genau einmal drehen.

In Abbildung 41 sind die montierten Komponenten des linken Antriebs abgebildet. Die Befestigungsschrauben der Motoren sind dabei in rot hervorgehoben, die Befestigungsschrauben des Zahnrads der Antriebswelle in blau und die Befestigung des Reifens in orange. Das Zahnrad der Motorwelle ist durch Erhitzen geweitet und auf die Welle aufgesetzt worden. Zur Sicherheit dient hier etwas Sekundenkleber dazu, dass sich das Zahnrad auf der Welle nicht durchdreht. Sekundenkleber ist hier völlig ausreichend, da aufgrund der Übersetzung nur 1/7 des auf die Reifen wirkenden Moments auf das Zahnrad der Motorwelle wirkt.



Abbildung 41: Montage der BLDC-Motoren und Übersetzung auf die Antriebsachse; In rot die Befestigung des BLDC-Motors, in blau die Befestigung des Zahnrads an der Antriebswelle und in orange die Befestigung des Reifens

### 4.3 Konfiguration der Motorcontroller

Die beiden Motorcontroller erwarten nach dem Zuschalten der Spannungsversorgung eine Initialisierungssequenz. Das Power-On-Ereignis wird vom ESC mit drei Tönen (tief, mittel, hoch) signalisiert. Im Anschluss daran soll der Wert an der Signalleitung größer als 0% sein. Das bedeutet, dass ein PWM-Signal angelegt werden muss, welches eine Drehzahl  $N \geq 0\text{rpm}$  repräsentiert. Der ESC quittiert das Erkennen des PWM-Signals mit einem tiefen Ton. Die Initialisierungssequenz ist allerdings erst dann beendet, wenn das PWM-Signal an der Signalleitung zuerst vergrößert und dann auf 0% verringert wird ( $N = 0\text{rpm}$ ). Dabei gibt der ESC einen letzten, hohen Ton von sich. Der Ablauf der Initialisierungssequenz ist in Abbildung 42 dargestellt. Nach dem letzten Ton ist die Initialisierung beendet und der BLDC-Motor dreht sich in Abhängigkeit des an der Signalleitung anliegenden PWM-Signals.

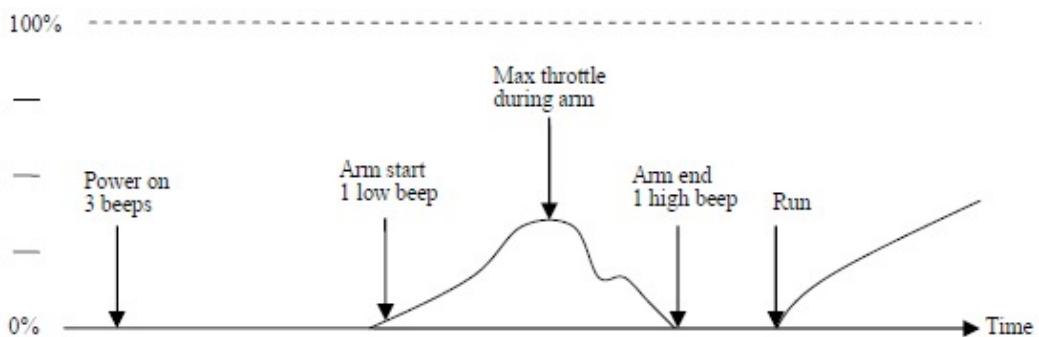


Abbildung 42: Initialisierungssequenz der ESCs; Wert an der Signalleitung für die Initialisierungssequenz über der Zeit; 0% entspricht dem PWM-Tastgrad für den Stillstand und 100% dem für die maximal erreichbare Drehzahl

Da die ESCs individuell konfiguriert werden können und in der Dokumentation zu wenige Angaben gemacht werden, ist der Tastgrad für die Werte 0% (Stillstand) und 100% (maximal erreichbare Drehzahl) unbekannt. Deshalb ist es nicht möglich zu wissen, welche PWM-Tastgrade für die Initialisierungssequenz verwendet werden müssen. Über die Konfiguration der ESCs können die Grenzen für 0% und 100% selbst festgelegt werden. Für das Flashen der Motorcontroller wird die Software „BLHeliSuite“ verwendet. Die Verbindung zwischen den ESCs und der Software stellt ein Arduino Nano her.

Der Arduino Nano muss zuvor mit einer neuen Software beschrieben werden. Um die ESCs flashen zu können wird der Signalpin des zu programmierenden ESC mit dem Pin D3 und der Massepin der Signalleitung mit einem Massepin des Arduino Nano verbunden. Danach wird in der Software „BLHeliSuite“ im Reiter „Make Interface“ eine neue Schnittstelle erstellt (siehe Abbildung 43). Auf der rechten Seite des Programm-Fensters wird ein Arduino Nano als Schnittstelle eingerichtet. Mit einem Klick auf den Button „Arduino 4way-interface“ wird die neue Software, durch die die ESCs neu konfiguriert werden können, auf den Arduino Nano geladen.

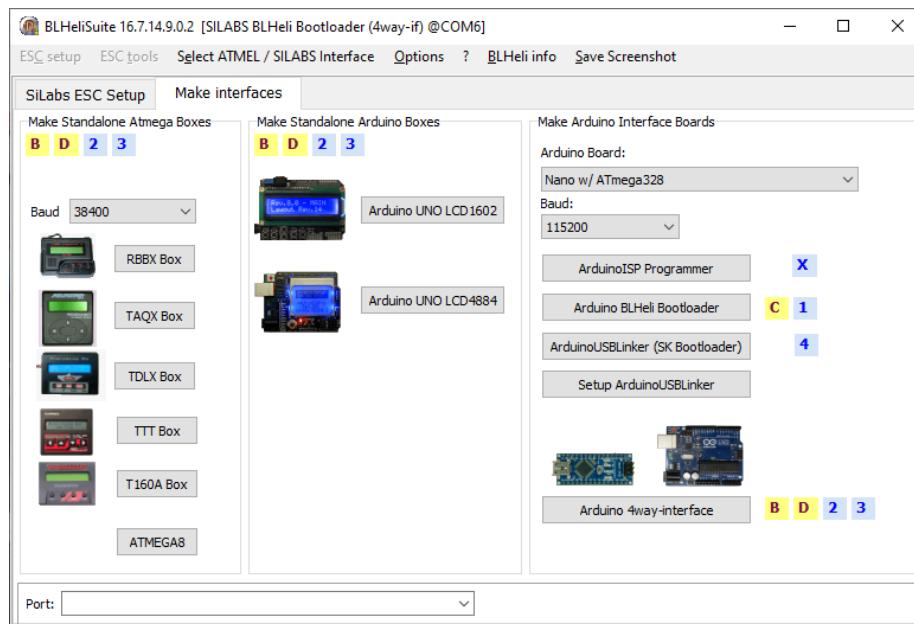


Abbildung 43: Programmierung des Arduino Nano zur ESC-Konfiguration mit der Software BLHeliSuite

Nach dem Programmieren des Arduino Nano wird die Kommunikation der BLHeliSuite-Software mit dem ESC hergestellt. Über die Schaltfläche „Read Setup“ werden die voreingestellten Parameter des ESC ausgelesen. Im nächsten Schritt werden die Zeiten für die Werte „PPM min Throttle“ (0% Aussteuergrad) und „PPM max Throttle“ (100% Aussteuergrad) auf 1100µs und 1900µs angepasst. Alle vorgenommenen Einstellungen sind in Abbildung 44 einsehbar. Die Werte werden über die Betätigung der Schaltfläche „Write Setup“ auf den ESC geladen. Da jetzt die Werte für 0% und 100% Aussteuerung bekannt sind, können die einzelnen Schritte der Initialisierung im Programm des Fahrzeugs abgearbeitet werden.

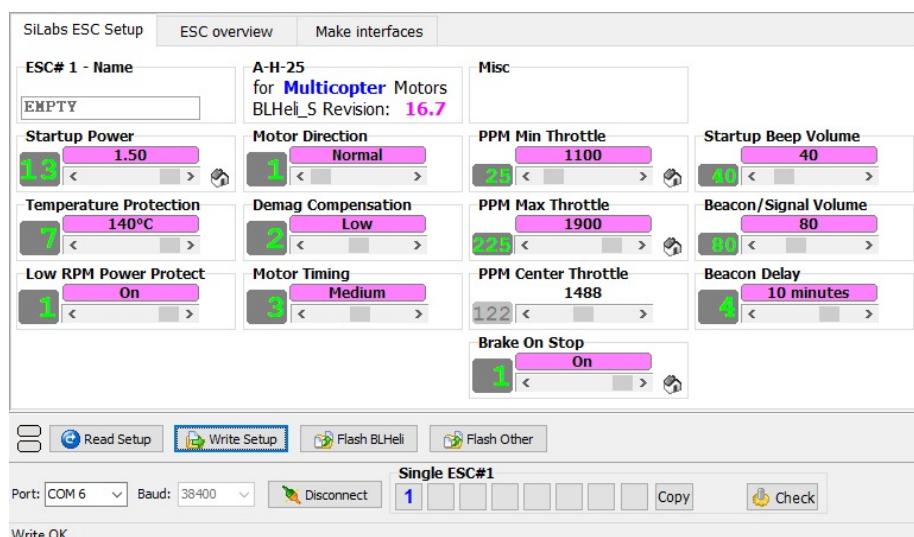


Abbildung 44: Konfiguration der ESCs mit der Software „BLHeliSuite“

#### 4.4 Programmierung des Antriebsbausteins

Der Antriebsbaustein der Software ist in zwei Dateien unterteilt, die Dateien „drive.c“ und „drive.h“. Die Datei „drive.h“ enthält alle relevanten Bibliotheken und Prototypen für die Datei „drive.c“.

Außer der Einbindung der Bibliotheken und der Prototypen der Funktionen aus der Datei „drive.c“ sind hier auch die Parameter für die Initialisierungssequenz der ESCs und für die Initialisierung des Timers für die PWM-Signale hinterlegt (Abbildung 45).

```
/*****************************************************************************  
 * Defines  
*****/  
#define SOURCE_CLOCK CLOCK_GetFreq(kCLOCK_CoreSysClk) //Get Main-CLK Frequency  
  
/* Channel definitions */  
#define CTIMER3_PWM_LEFT_CHANNEL kCTIMER_Match_0      //Definition of channel 0 ID  
#define CTIMER3_PWM_RIGHT_CHANNEL kCTIMER_Match_2     //Definition of channel 2 ID  
  
/* BLDC PWM initialization value definitions */  
#define BLDC_PWM_INIT_HIGH_VALUE 330000 //Init High value Sequence 1,5ms  
#define BLDC_PWM_INIT_LOW_VALUE 220000 //Init Low value Sequence 1,0ms  
#define CTIMER3_PWM_0_DUTY 0          //Definition of initialization duty  
#define CTIMER3_PWM_PERIOD 4400000    //Definition of PWM period
```

Abbildung 45: Relevante Zeilen der Datei „drive.h“ mit den Parametern für die Initialisierungssequenz der ESCs und für die Initialisierung des PWM-Timers

Für die PWM-Periodendauer wird bei der Initialisierung ein Wert von 4.400.000 Takte festgesetzt, woraus mit einer Central Processing Unit (CPU)-Taktfrequenz von 220MHz (220.000.000 Takte pro Sekunde) eine Periodendauer von 20ms resultiert. Die Pulsbreite wird während des Programmablaufs regelmäßig überschrieben.

Der ESC-Initialisierungswert für den Stillstand („BLDC\_PWM\_INIT\_LOW\_VALUE“, 220.000) entspricht hier einer PWM-Pulsbreite von 1,0ms und der Initialisierungswert für die in etwa mittlere Aussteuerung („BLDC\_PWM\_INIT\_HIGH\_VALUE“, 330.000) einer Breite von 1,5ms. Die volle Aussteuerung der Motoren wird, wie über die „BLHeliSuite“ festgelegt, bei einer Pulsdauer von 1,9ms erreicht (BLDCMaxValue 1900µs, ca. 418.000). Eine Drehzahl von N = 0rpm wird theoretisch über die bei der ESC-Konfiguration festgelegten 1,1ms ermöglicht (242.000). Da das PWM-Signal leicht abweicht, wird ein etwas geringerer Wert veranlagt (BLDCMinValue 1090µs, ca. 240.000). Damit wird sichergestellt, dass sich die Räder im Stillstand nicht drehen.

Die Parameter BLDC.MaxValue und BLDC.MinValue existieren im Code für die linke und rechte Antriebsseite separat („BLDCLeft.MaxValue“, „BLDCLeft.MinValue“ & „BLDCRight.MaxValue“, „BLDCRight.MinValue“). Die Werte dieser Parameter kommen direkt aus dem Electrically Erasable Programmable Read-Only Memory (EEPROM) und können mit dem Bedienungsboard eingestellt werden (siehe Abbildung 46).

```

int32_t* BLDCLeftInitValue= &((all_param_t*)&const_all_param)->motors.BLDCLeft.init; //left BLDC init value
int32_t* BLDCLeftMinValue= &((all_param_t*)&const_all_param)->motors.BLDCLeft.min; //left BLDC min value
int32_t* BLDCLeftMaxValue= &((all_param_t*)&const_all_param)->motors.BLDCLeft.max; //left BLDC max value

int32_t* BLDCRightInitValue= &((all_param_t*)&const_all_param)->motors.BLDCRight.init; //right BLDC init value
int32_t* BLDCRightMinValue= &((all_param_t*)&const_all_param)->motors.BLDCRight.min; //right BLDC min value
int32_t* BLDCRightMaxValue= &((all_param_t*)&const_all_param)->motors.BLDCRight.max; //right BLDC max value

```

Abbildung 46: Extremwerte der Antriebsgeschwindigkeit als Parameter aus dem EEPROM, deren Werte über das Bedienungsboard individuell einstellbar sind; Teil der Datei „drive.c“

```

/****************************************************************************
 * Parameters
 *****/
const ctimer_config_t BLDC_config = {
    .mode = kCTIMER_TimerMode, /* TC is incremented every rising APB bus clock edge */
    .input = kCTIMER_Capture_0, /*!< Timer capture channel 0 */
    .prescale = 0 /*!< Prescale value 0 --> */
};

/****************************************************************************
 * BLDC Main Initialization function
 *****/
void BLDC_Init(void)
{
    CTIMER3_Init(); //TIMER3 Initialization

    //Configure Pin P[3][10] (J13 Pin7) (Left BLDC)
    CTIMER3->MCR |= CTIMER_MCR_MR0RL_MASK; //Reload MR0 with content of shadow register at timer overflow
    IOCON->PIO[3][10] &= 0xFFFFFFFF0; //Clear FUNC bits of P3.10
    IOCON->PIO[3][10] |= 0x3; //Set FUNC bits to CTIMER3_MAT0 function ALT3 P3.10
    GPIO->DIR[3] |= 1<<10; //Set P3.10 to output
    CTIMER3->MSR[0] = CTIMER3_PWM_PERIOD - BLDC_PWM_INIT_LOW_VALUE; //Initialize MSR with BLDC_PWM_INIT_LOW_VALUE value
    //*****


    //Configure Pin P[0][27] (J13 Pin12) (Right BLDC)
    CTIMER3->MCR |= CTIMER_MCR_MR2RL_MASK; //Reload MR2 with content of shadow register at timer overflow
    IOCON->PIO[0][27] &= 0xFFFFFFFF0; //Clear FUNC bits of P0.27
    IOCON->PIO[0][27] |= 0x3; //Set FUNC bits to CTIMER3_MAT2 function ALT3 P0.27
    GPIO->DIR[0] |= 1<<27; //Set P0.27 to output
    CTIMER3->MSR[2] = CTIMER3_PWM_PERIOD - BLDC_PWM_INIT_LOW_VALUE; //Initialize MSR with BLDC_PWM_INIT_LOW_VALUE value
    //*****


    //Create ESC initialization task
    if (xTaskCreate(ESC_Init_Task, "ESC_Init_Task", configMINIMAL_STACK_SIZE + 100, NULL, 1, NULL) != pdPASS)
    { LED3_ON(); } //LED3 is Error
    //*****


    /*
     * TIMER3 Initialization function
     *****/
void CTIMER3_Init(void)
{
    /* CTIMER1 peripheral initialization */
    CTIMER_Init(CTIMER3, &BLDC_config);

    /* PWM channel 0 of CTIMER3 peripheral initialization (Left BLDC)*/
    CTIMER_SetupPwmPeriod(CTIMER3, CTIMER3_PWM_LEFT_CHANNEL, CTIMER3_PWM_PERIOD, CTIMER3_PWM_PERIOD - CTIMER3_PWM_0_DUTY, false);

    /* PWM channel 2 of CTIMER3 peripheral initialization (Left BLDC)*/
    CTIMER_SetupPwmPeriod(CTIMER3, CTIMER3_PWM_RIGHT_CHANNEL, CTIMER3_PWM_PERIOD, CTIMER3_PWM_PERIOD - CTIMER3_PWM_0_DUTY, false);

    /* Start TIMER3 */
    CTIMER_StartTimer(CTIMER3);
}

```

Abbildung 47: Funktionen BLDC\_Init und CTIMER3\_Init der Datei „drive.c“

Auch die beiden PWM-Timer (einer je Motorcontroller) benötigen bei der Initialisierung einige Parameter, deren Werte in der Datei „drive.h“ festgelegt sind (PWM-Periodendauer, PWM-Pulsdauer, Kanäle). Die Kanäle werden auf das Timer Match Register 2 (rechter Antrieb, „kCTIMER\_Match\_2“) und auf das Timer Match Register 0 (linker Antrieb, „kCTIMER\_Match\_0“) festgelegt, was bei dem verwendeten Controller den Pins P0.27 (rechts)

und P3.10 (links) entspricht. Der Pin P3.10 wird auf der Controllerplatine über den Pin 7 und der Pin P0.27 über den Pin 12 der Buchsenleiste J13 nach außen geführt. Der Anschluss der Motorcontroller ist über den Anhang „Anhang 1: Schaltplan“ nachvollziehbar.

Die Datei „drive.c“ enthält die Funktionen zur Initialisierung der für die Verwendung der Antriebe notwendigen Controller-Peripherie (siehe Abbildung 47) und zur Initialisierung der Motorcontroller (siehe Abbildung 48).

In der Funktion BLDC\_Init wird zuerst die Funktion CTIMER3\_Init aufgerufen, welche die beiden vorher festgelegten Kanäle des Timers C3 („kCTIMER\_Match\_0“ und „kCTIMER\_Match\_2“) mit den in der Datei „drive.h“ festgelegten Parametern als PWM-Timer mit einer Periodendauer von 20ms und einer Pulsdauer von 0ms initialisiert. Im Anschluss daran wird einzeln für beide Kanäle festgelegt, dass bei einem Timer-Überlauf die neuen Daten für die Pulslängen aus den Shadow-Registern geladen werden sollen. Zum Ändern der Geschwindigkeit eines der beiden Antriebe muss deshalb lediglich ein neuer Wert in das Shadow-Register geschrieben werden. Hier muss allerdings aufgepasst werden, da das Register nicht die Pulsbreite (On-Time) sondern die Off-Time erwartet. Deshalb muss der Wert, der eingetragen wird, der Periodendauer abzüglich der Pulsdauer entsprechen. Zusätzlich werden in der Funktion BLDC\_Init auch die Pins P3.10 und P0.27 für die Verwendung als PWM-Ausgang des Timers C konfiguriert. Am Ende wird noch der erste ESC-Initialisierungswert in die beiden Shadow-Register geschrieben, bevor die ESC-Initialisierungsfunktion aufgerufen wird (siehe Abbildung 48).

Der Task BLDC\_Init\_Task beginnt mit dem Befüllen der Shadow-Register mit dem ersten Initialisierungswert. Nach einer Wartezeit von 2s wird der zweite Initialisierungswert in die Register geschrieben. Ebenfalls nach einer Zeit von 2s wird dann wieder der erste Initialisierungswert in die Shadow-Register geschrieben und die Initialisierung der ESCs ist abgeschlossen. Die Wartezeiten sind notwendig, damit die ESCs Zeit haben, die Änderungen zu erfassen.

```

/*
 * ESC Initialization Task
 */
void ESC_Init_Task(void *pvParameters)
{
    while(1)
    {
        //Initialize Sequence for BLDC-Motors
        CTIMER3->MSR[0] = CTIMER3_PWM_PERIOD - BLDC_PWM_INIT_LOW_VALUE;      //Low throttle value
        CTIMER3->MSR[2] = CTIMER3_PWM_PERIOD - BLDC_PWM_INIT_LOW_VALUE;      //Low throttle value
        vTaskDelay(2000);

        CTIMER3->MSR[0] = CTIMER3_PWM_PERIOD - BLDC_PWM_INIT_HIGH_VALUE;     //Half throttle value
        CTIMER3->MSR[2] = CTIMER3_PWM_PERIOD - BLDC_PWM_INIT_HIGH_VALUE;     //Half throttle value
        vTaskDelay(2000);

        CTIMER3->MSR[0] = CTIMER3_PWM_PERIOD - BLDC_PWM_INIT_LOW_VALUE;      //Low throttle value
        CTIMER3->MSR[2] = CTIMER3_PWM_PERIOD - BLDC_PWM_INIT_LOW_VALUE;      //Low throttle value
        vTaskDelay(3500);
    }
}

//Testing Sequence for BLDC-Motors

/*CTIMER3->MSR[0] = CTIMER3_PWM_PERIOD - (*BLDCLeft.MaxValue);           //Set motor to full speed
CTIMER3->MSR[2] = CTIMER3_PWM_PERIOD - (*BLDCRight.MaxValue);            //Set motor to full speed
vTaskDelay(1000);
CTIMER3->MSR[0] = CTIMER3_PWM_PERIOD - (*BLDCLeft.MinValue);             //Stop motor
CTIMER3->MSR[2] = CTIMER3_PWM_PERIOD - (*BLDCRight.MinValue);            //Stop motor*/
}

vTaskSuspend(NULL); //suspend Task
}
}

```

Abbildung 48: Funktion BLDC\_Init\_Task der Datei „drive.c“ zum Durchlaufen der Initialisierungssequenz der ESCs

## 4.5 Drehzahlmessung

Im folgenden Abschnitt zur Drehzahlmessung werden deren Notwendigkeit sowie Entwicklungsschritte und Realisierung erarbeitet. Des weiteren wird auch ein Ausblick für die Weiterentwicklung Schaltung gegeben.

### 4.5.1 Erörterung der Notwendigkeit einer Drehzahlmessung

Die Quellspannung von Akkus verringert sich mit steigender Betriebszeit, weshalb bereits bei vorherigen Fahrzeug-Versionen eine Drehzahlmessung benötigt wird, um über eine Regelung die Spannungsabhängigkeit der Drehzahl bei Gleichstrommotoren (DC-Motoren) ausgleichen zu können. Die Spannungsabhängigkeit von DC-Motoren bereitet insbesondere beim Überfahren eines Hügels Probleme. Da die in den vorangegangenen Fahrzeugmodellen verwendeten DC-Motoren durch BLDC-Motoren ersetzt werden, stellt sich allerdings erneut die Frage nach der Notwendigkeit einer Drehzahlmessung. Die Variation der Drehzahl erfolgt bei DC-Motoren über die Änderung der Betriebsspannung, was auch die Abhängigkeit von der Versorgungsspannung erklärt. Die Drehzahlvariation bei einem BLDC-Motor wird hingegen über eine Frequenzänderung der Strangspannungen realisiert. Zur Klärung der Frage nach einer Spannungsabhängigkeit der BLDC-Motoren, ist eine Messung am Prüfstand erforderlich.

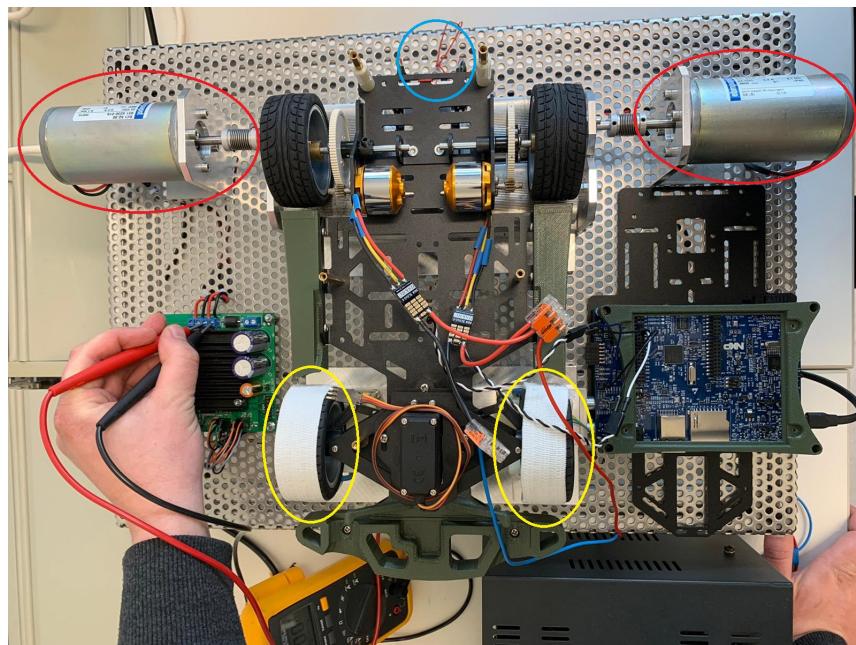


Abbildung 49: Messaufbau zur Prüfung der Spannungsabhängigkeit der BLDC-Motoren; in blau die Befestigung des Hecks an der Grundplatte des Prüfstands, in gelb die Befestigung der Vorderreifen am Prüfstand, in rot die DC-Motoren zur Spannungsmessung

Für die Durchführung der Messung wird ein bereits vorhandener Prüfstand für Modelfahrzeuge verwendet (siehe Abbildung 49), welcher im Rahmen einer Bachelor-Abschlussarbeit an der HAW Landshut erstellt worden ist. Damit das Fahrzeug möglichst stabil auf dem Prüf-

stand steht und die Räder des Fahrzeugs ihre Rotationsbewegung besser auf die DC-Motoren des Prüfstands übertragen können, wird das Heck des Fahrzeugs mit Hilfe eines Drahtes am Prüfstand befestigt (Abbildung 49 blaue Markierung). Die vorderen Räder des Fahrzeugs sind ebenfalls am Prüfstand fixiert (Abbildung 44 gelbe Markierungen).

Zur Überprüfung einer eventuellen Abhängigkeit der Drehzahl von der Versorgungsspannung, wird die Spannung an den vorhandenen DC-Motoren des Prüfstands gemessen, da diese direkt proportional zur Drehzahl ist (siehe Gleichung 1). Die Abhängigkeit der gemessenen Spannung von der Drehzahl ist allerdings nur annähernd linear, da die Drehmomentkonstante  $k_i$  bei hohen Drehzahlen leicht einbricht. Aufgrund der für die Messung konstant eingestellten Pulsbreite von 1,5ms, spielt der Einbruch der Drehmomentkonstante allerdings keine große Rolle, da sich die Drehzahl, wenn überhaupt, nur sehr gering ändert. Dieser Zusammenhang kann deshalb vereinfachend als linear angenommen werden.

Die Versorgungsspannung der ESCs wird für die Aufnahme verschiedener Messwerte zwischen 6,1V und 8V variiert. Ändert sich die gemessene Spannung mit der Variation der Versorgungsspannung, so ist das ein hinreichender Beweis dafür, dass die Drehzahl der BLDC-Motoren spannungsabhängig ist. Die Ergebnisse der Messreihe aus Tabelle 1 sind zur einfacheren Auswertung in Abbildung 50 visualisiert.

$$U_A = k_i \cdot N \cdot \frac{\pi}{30} \quad (1)$$

Messung	1	2	3	4	5	6	7	8	9	10
$U_{\text{Versorgung}} [V]$	8.0	7.9	7.8	7.7	7.6	7.5	7.4	7.3	7.2	7.1
$U_{\text{Messung}} [V]$	5.80	5.75	5.65	5.57	5.48	5.40	5.35	5.25	5.18	5.09
Messung	11	12	13	14	15	16	17	18	19	20
$U_{\text{Versorgung}} [V]$	7.0	6.9	6.8	6.7	6.6	6.5	6.4	6.3	6.2	6.1
$U_{\text{Messung}} [V]$	5.05	4.96	4.87	4.82	4.69	4.73	4.66	4.60	4.50	4.45

Tabelle 1: Messreihe 1: Messwerte für Überprüfung einer eventuellen Spannungsabhängigkeit der Drehzahl

Aus Abbildung 50 kann die Erkenntnis abgeleitet werden, dass die Drehzahl der BLDC-Motoren, welche proportional zu der an den DC-Motoren gemessenen Ankerspannung  $U_{\text{Messung}}$  ist, bei einem Abfallen der Versorgungsspannung nicht konstant bleibt, denn andernfalls wäre die resultierende Gerade eine Horizontale. Deshalb ist die Drehzahl der BLDC-Motoren wie die der DC-Motoren abhängig von der Versorgungsspannung.

Die sich aus der Messreihe ergebende Erkenntniss, führt zu dem Entschluss, dass eine Drehzahlregelung nicht nur sinnvoll, sondern auch erforderlich ist. Denn während der Durchfahrt des Parcours wird nicht zu jedem Zeitpunkt die volle Drehzahl benötigt. Folglich hat

eine Regelung den Vorteil, dass die vorgegebene Drehzahl auch bei einer fortschreitenden Entladung des Akkus weiterhin aufrecht gehalten werden kann. Wird eine feste Geschwindigkeit vorgegeben, kann die Regelung bei fallender Versorgungsspannung der ESCs das PWM-Signal bis zur Stellgrenze (Pulsbreite = 1,9ms) erhöhen. Erst bei Erreichen der Stellgrenze nimmt die Drehzahl linear mit der Versorgungsspannung ab.

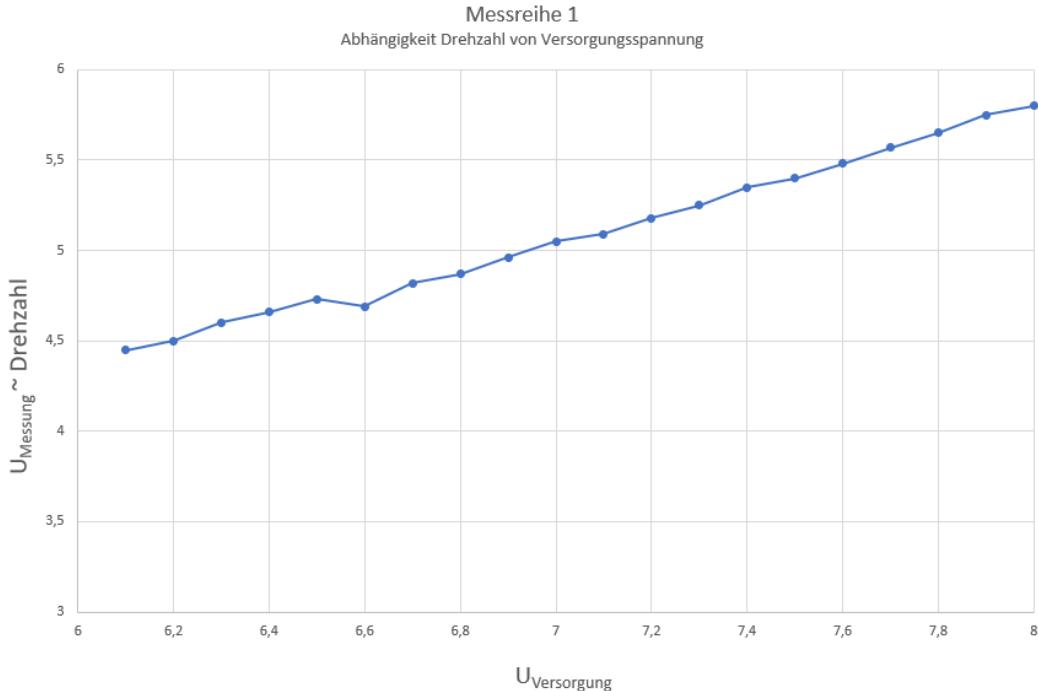


Abbildung 50: Spannungsabhängigkeit der Drehzahl der BLDC-Motoren von der Versorgungsspannung

Des weiteren muss das Fahrzeug im Rahmen des NXP-Cups nach dem Überfahren der Ziellinie innerhalb von zwei Metern anhalten können. In der Konfiguration der ESCs bietet sich dafür die Einstellung „Break On Stop“ an, mit der das Fahrzeug bei einer PWM-Pulsdauer von 1,1ms (BLDCMinValues: BLDCLeftMinValue, BLDCRightMinValue) aktiv bremst, indem die drei Phasen der BLDC-Motoren auf Masse geschlossen werden.

Zur Verifizierung eines ordnungsgemäßen Bremsverhaltens, wird eine Probefahrt auf einer langen geraden Strecke an der HAW Landshut durchgeführt. Die Antriebe sind dabei so konfiguriert, dass das Fahrzeug nach der Initialisierung der BLDC-Motoren unmittelbar mit maximaler Geschwindigkeit fortbewegt und nach kurzer Fahrtzeit (ca. 3s) stoppt (BLDCMinValues). Hierbei stellt man fest, dass die Bremskraft ausreicht, um das Fahrzeug innerhalb von ca. 30 cm zum Stehen zu bringen.

Während des Testens fällt auf, dass sich die Reifen trotz gleicher ESC-Konfiguration unterschiedlich schnell drehen. Der Grund für dieses Problem lässt darauf schließen, dass die beiden Antriebsseiten einen unterschiedlichen Reibungswert aufweisen. Das kann daran liegen, dass

bei den Achsen keine Lager verwendet werden und bei der Übersetzung das Getriebespiel (zwischen den Zahnräden) auf beiden Antriebsseiten unterschiedlich groß ist, was ebenfalls die Reibung beeinflusst.

Die Spannungsabhängigkeit der BLDC-Motoren sowie das unterschiedliche und reibungsabhängige Drehverhalten der beiden Antriebe, machen eine Drehzahlregelung zwingend erforderlich, um ein zuverlässiges und sicheres Fahrverhalten zu gewährleisten.

#### 4.5.2 Auswahl des Messprinzips

Aufgrund der Erkenntnis, dass eine Drehzahlregelung der BLDC-Motoren erforderlich ist, gilt es in diesem Abschnitt entsprechende Messprinzipien zur Drehzahlerfassung gegenüberzustellen und die für dieses Projekt sinnvollste Messmethode zu bestimmen.

Die in bisherigen Fahrzeugversionen am häufigsten verwendeten Methoden zur Drehzahlerfassung sind die Verwendung von Hallsensoren oder Lichtschranken (siehe Abbildung 51). Hallsensoren und Lichtschranken eignen sich hervorragend für eine berührungslose Drehzahlerfassung. Für die Drehzahlmessung mithilfe von Hallsensoren werden kleine Magnete auf der Innenseite der Hinterräder angebracht, welche vom Hallsensor erfasst werden (Abbildung 51, links). Anhand der Zeit zwischen den Zuständen „Magnet vorhanden“ oder „Magnet nicht vorhanden“ (HIGH/LOW-Signal), kann unter Berücksichtigung der jeweiligen Getriebeübersetzung und Menge der Magnete die Drehzahl der BLDC-Motoren bestimmt werden. Die Drehzahlmessung mit einer Lichtschranke funktioniert in der Auswertung genau wie die Messung mit Hallsensoren, lediglich die Messmittel sind anders (Abbildung 51, rechts).

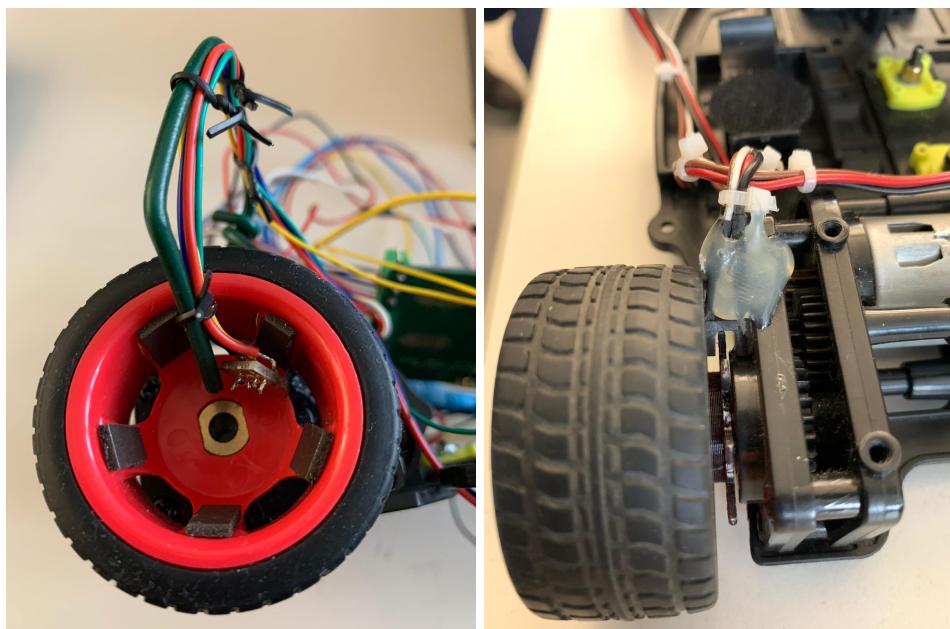


Abbildung 51: Varianten der Drehzahlerfassung bei vorherigen Fahrzeugversionen; links: Hallsensoren; rechts: Lichtschranke

Ein Hallsensor bietet den Vorteil, dass neben der Erfassung der Drehzahl auch die Drehrichtung mit Hilfe eines zweiten um  $90^\circ$  versetzten Hallsensors bestimmt werden kann. Dazu muss nicht einmal ein zweiter Sensor gekauft werden, da bereits zwei Hallsensoren in einem Gehäuse verbaut sind. Im Zuge dieser Projektarbeit ist eine Erfassung der Drehrichtung jedoch nicht erforderlich, da die für die BLDC-Motoren verwendeten ESCs so konfiguriert sind, dass nur eine Fahrt in Vorwärtsrichtung möglich ist. Ein weiterer Vorteil des Hallsensors aber auch einer Lichtschranke ist, dass die für die GPIO-Pins notwendigen Rechtecksignale direkt vorgegeben werden und keine zusätzliche Umformung des Signals mehr notwendig ist.

Aufgrund der Klebefestigung der Magnete besteht das Risiko, dass diese sich nach einer gewissen Zeit ablösen und somit die Daten zur Drehzahlerfassung verfälschen. Wie in Abbildung 51 zusätzlich zu erkennen ist, ist die Montage des Hallsensors, die lediglich über die Versorgungsdrähte oder eine zusätzliche Drahtstütze realisiert ist, sehr anfällig auf Erschütterungen. Eine zuverlässige und weniger erschütterungsempfindliche Montage des Hallsensors erfordert deshalb eine zusätzliche Halterung (3D-Druck Anbauteil). Diese Halterung muss erst konstruiert, gedruckt und montiert werden. Um mechanische Probleme, wie sie durch Erschütterungen entstehen können, vollständig zu eliminieren, bietet sich die Anwendung eines anderen, bisher noch nicht erwähnten Messprinzips an, auf welches in weiteren Verlauf genauer eingegangen wird.

Die Alternativmethode zur Erfassung der Drehzahl ist die Anbindung eines Tiefpassfilters mit nachgeschaltetem Komparator an eine der BLDC-Motor-Phasen. Die Schaltung filtert die hochfrequenten Anteile des Phasensignals heraus. Mithilfe einer Komparatorschaltung wird daraus ein Rechtecksignal mit steilen Flanken erzeugt. Hierbei wird das auf der Phase vom ESC bereitgestellte, hochfrequente Signal (Abbildung 52, grüner Signalverlauf V(in)) abgegriffen. Durch die mit dem Schaltungssimulationstool LTSpice entwickelte und auf einer Lochrasterplatine in Hardware realisierte Tiefpass- und Komparatorschaltung kann dieses abgegriffene Signal in ein eindeutiges Rechtecksignal umgeformt werden (Abbildung 52, pinker Signalverlauf V(out)).

Bei der Drehzahlmessung wird der zeitliche Abstand zwischen zwei aufeinanderfolgenden, steigenden Flanken des aus der Tiefpass-/Komparatorschaltung generierten Rechtecksignals mit Hilfe eines Timer-Zählers bestimmt. Die erfassten Zeiten zwischen den Flanken sind proportional zur Drehzahl und können direkt für die Drehzahlregelung verwendet werden, ohne die tatsächliche Drehzahl errechnen zu müssen.

Es gilt zu erwähnen, dass auch bei dieser Methode eine Erfassung der Drehrichtung möglich wäre. Bei drei Phasen, welche um  $120^\circ$  zueinander versetzt sind, kann die Drehrichtung anhand der Pulsreihenfolge der einzelnen Phasen bestimmt werden, wobei nur zwei der drei Phasen benötigt werden. Aufgrund dessen, dass die ESCs lediglich eine feste Drehrichtung erlauben (harte Konfiguration), wird die Drehrichtungsbestimmung nicht benötigt.

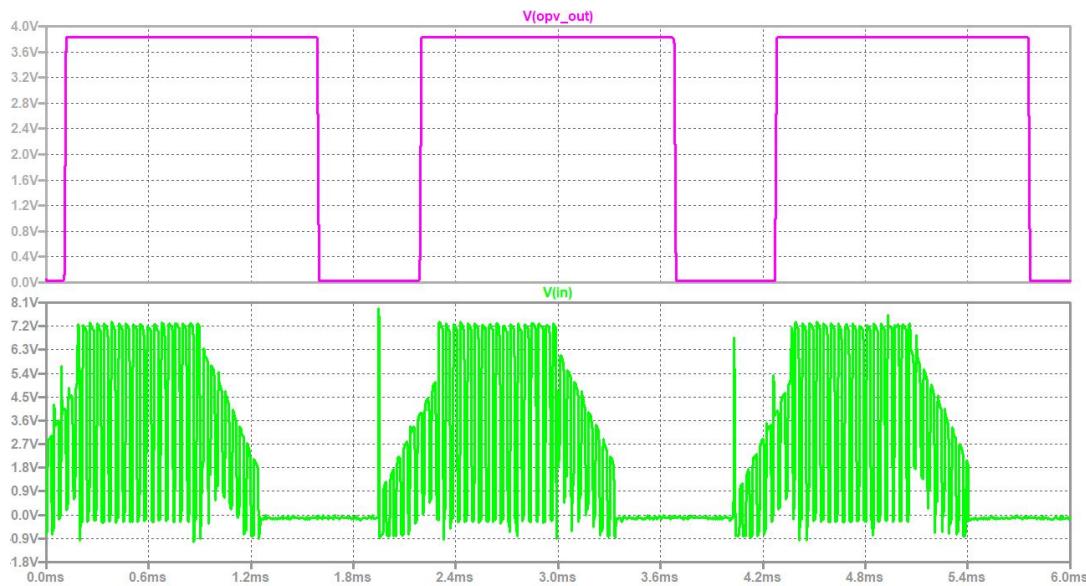


Abbildung 52: Phasensignal eines Antriebs ( $V(\text{in})$ , grün) bei schneller Fahrt und Darstellung des simulierten Ausgangssignals ( $V(\text{out})$ , pink) aus der Tiefpass-/Komparatorschaltung; erstellt und simuliert mit dem Schaltungssimulationstool LTspice XVII

Da auch bei diesem Messprinzip ein hoher Aufwand (Entwicklungs- und Simulationsaufwand) erforderlich ist, gilt es abzuwägen, welche dieser Methoden am vorteilhaftesten für das Gesamtprojekt ist. Unter Betrachtung der Vor- und Nachteile aller Messprinzipien bietet sich die Tiefpass-/Komparatorschaltung am meisten an. Insbesondere die Unempfindlichkeit gegen Erschütterungen und die Verwendung der ohnehin von den ESCs bereitgestellten Phasensignale, spricht für die Realisierung der Drehzahlmessung mit dieser Methode. Als weitere Gründe gegen die sonstigen Messprinzipien sprechen die unzuverlässige Montage von Hallsensoren oder Lichtschranken und die fehlende Notwendigkeit einer Erfassung der Drehrichtung.

#### 4.5.3 Hardware für die Drehzahlmessung

In diesem Abschnitt wird die Entwicklung der Schaltung zur Drehzahlmessung beschrieben. Dabei wird sowohl auf die Probleme eingegangen, die sich während der Schaltungsentwicklung ergeben als auch auf deren Behebung. Damit das Phasensignal in ein eindeutiges Rechtecksignalsignal umgesetzt werden kann, bedarf es der im vorherigen Abschnitt erörterten Entwicklung einer Tiefpass- / Komparatorschaltung. Es soll deutlich gemacht werden, welche Bauteile verwendet werden und welchen Zweck diese in der Schaltung erfüllen. Zur Entwicklung und Simulation der Drehzahl-Messschaltung wird das Schaltungssimulationstool LTspice XVII und zur Erstellung des Platinenlayouts die Freeware BlackBoard Circuit Designer verwendet.

Zu Beginn wird die Schaltung für ein Phasensignal in LTspice realisiert. Der daraus resultierende Schaltplan zur Drehzahlmessung ist in Abbildung 53 sichtbar und dient als Überblick über die verwendeten Bauteile mit deren entsprechenden Kennwerten. Es gilt zu erwähnen, dass der Spannungsteiler ( $R_5$ ,  $R_6$ ,  $R_7$ ,  $R_8$ ) zwischen  $V_{ref}$  und dem invertierenden Operationsverstärkereingang  $IN^-$  nur einmal realisiert werden muss, worauf im weiteren noch detaillierter eingegangen wird.

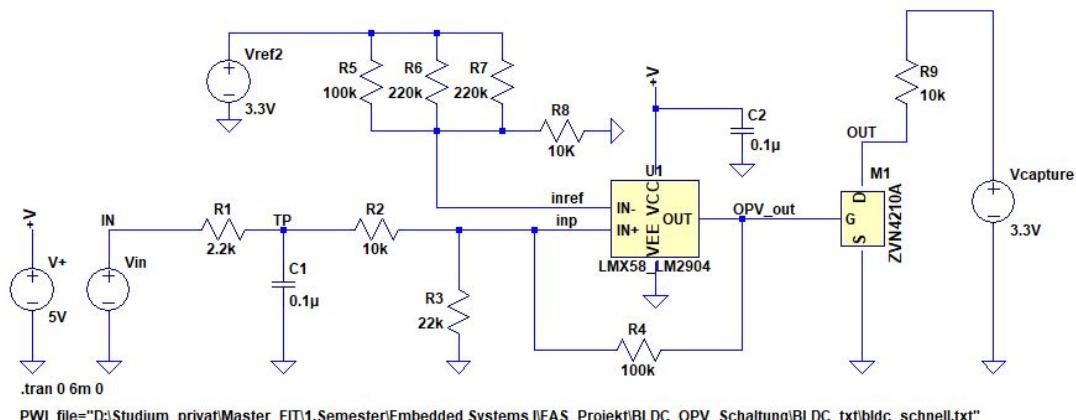


Abbildung 53: Schaltplan der Tiefpass-/Komparatorschaltung; erstellt für die Simulation eines Phasensignals in LTspice XVII

Das Phasensignal wird von der Spannungsquelle Vin repräsentiert. Die Signale werden vor der Schaltungsentwicklung zwischen einer Phase eines ESCs und dem Massepotential mit dem Oszilloskop gemessen. Hierzu werden die Antriebe auf den Modell-Prüfstand gestellt und je einmal mit hoher und geringer Drehzahl angesteuert. Es gilt zu erwähnen, dass die erfassten Signale für schnelle und langsame Fahrt nicht den maximalen und minimalen Drehzahlen entsprechen. Das Oszilloskop (DSO-X 3034A) bietet die Möglichkeit die gemessenen Ausgangssignale als .csv-Datei auf einem USB-Stick zu speichern. Das gespeicherte .csv-Datei wird daraufhin als Piecewise Linear Function in LTspice (PWL-File) importiert und einer

Spannungsquelle übergeben (rechter Mausklick auf Spannungsquelle / PWL-File anwählen / Pfad .csv-Datei wählen). Bevor die .csv-Datei importiert werden kann, muss die Datei umformatiert werden.

Das ungefilterte Phasensignal besitzt eine Frequenz von ca. 23 kHz. Um die hohen Frequenzanteile des Signals im ersten Schritt zu filtern, wird zunächst ein Tiefpass erster Ordnung benötigt. Die Dimensionierung des Tiefpasses erfolgt mit einem 1,1kOhm Widerstand ( $R_1$ ) und einem 100nF Kondensator ( $C_1$ ). Dabei beträgt die Grenzfrequenz  $f_G$  des Tiefpasses nach Gleichung 2 723Hz. Der aus der Simulation resultierende Spannungsverlauf sowie der am Oszilloskop gemessene Spannungsverlauf nach der Tiefpassfilterung sind in den Abbildungen 54 und 55 dargestellt (unterschiedliche Drehzahlen!).

$$f_G = \frac{1}{2 \cdot \pi \cdot R_1 \cdot C_1} \quad (2)$$

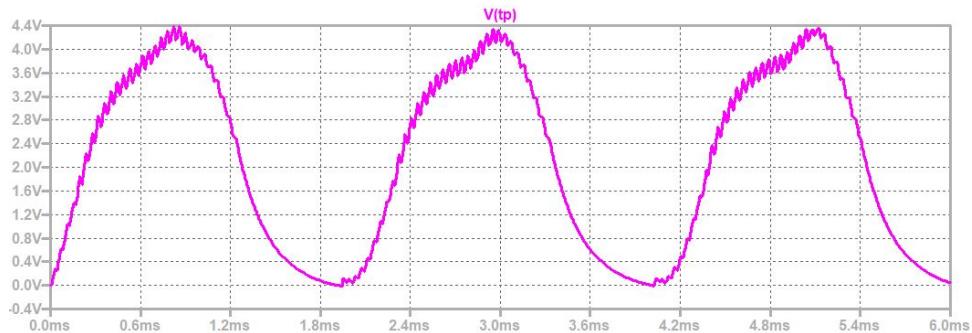


Abbildung 54: Simulationsergebnis des Phasensignals nach dessen Tiefpassfilterung bei schneller Fahrt

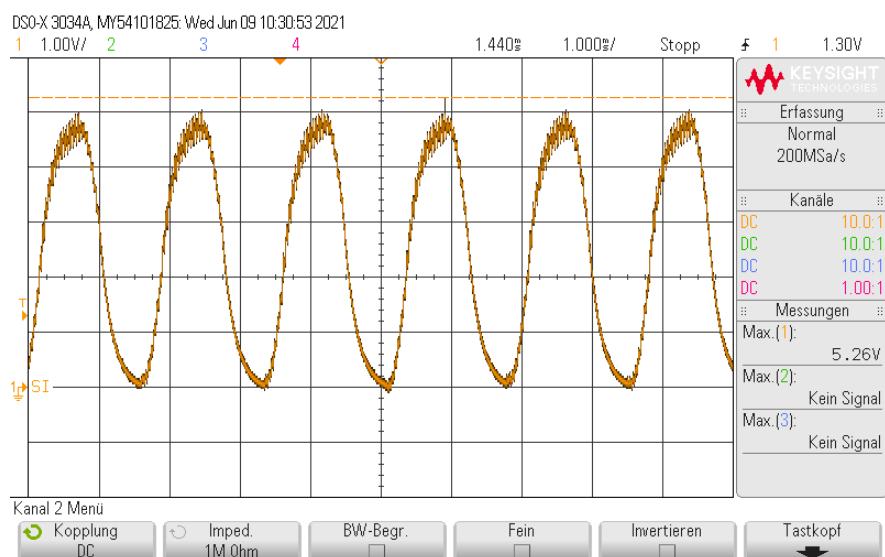


Abbildung 55: Messergebnis des Phasensignals nach dessen Tiefpassfilterung bei schneller Fahrt

Das zentrale Bauelement der Schaltung ist ein von Texas Instruments (TI) hergestellter Operationsverstärker LM358A, welcher als Komparator verwendet wird. Für eine maximal realistische Simulation wird ein Simulations-Modell des Herstellers in LTspice XVII eingebunden [TI]. Hierbei handelt sich um einen Operationsverstärker (OPV), welcher im Single-Supply betrieben wird. Der OPV wird zunächst mit einer Versorgungsspannung  $V_+$  von 5V betrieben, welche wie die Versorgungsspannung des Controllers vom Linearspannungsregler abgegriffen wird. Für die Drehzahlmessung wird jeweils eine Schaltung für das Phasensignal des linken und rechten Antriebs benötigt. Der OPV ermöglicht es, beide Signale in einem Bauteil zu verarbeiten. Dazu besitzt er zwei invertierende Eingänge (IN1-, IN2-) und zwei nicht-invertierende Eingänge (IN1+, IN2+) sowie zwei Ausgänge (OUT1, OUT2) (siehe Abbildung 56). Dadurch kann auf der Verteilerplatine zusätzlicher Platz eingespart werden.

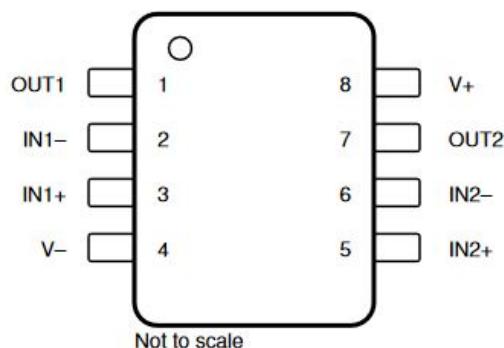


Abbildung 56: Pinbelegung des Operationsverstärkers LM358A

Da der OPV in der Schaltung als Schmitt-Trigger verwendet wird, müssen einige schaltungstechnische Maßnahmen, wie z.B. ein Spannungsteiler mit Rückführwiderstand hinzugefügt werden. Die Realisierung des Spannungsteilers erfolgt mit den Widerständen  $R_2 = 10\text{k}\Omega$ ,  $R_3 = 22\text{k}\Omega$  sowie einem Rückführwiderstand  $R_4 = 100\text{k}\Omega$ .

Grundsätzlich verstärkt ein OPV ständig die Differenz seiner Eingangsgrößen (IN+, IN-). Da ein Operationsverstärker einen theoretisch unendlichen Verstärkungsfaktor besitzt, reicht eine geringe Eingangsspannungsdifferenz aus, um den Ausgang bei dieser Schmitt-Trigger Schaltung in die Versorgungsspannungsbegrenzung zu treiben (nur Werte bis  $V_+$ ). Der verwendete OPV LM358A besitzt einen „open-loop voltage gain“ von  $100\text{V/mV}$ . Die Hysterese wird durch den Rückkopplungsanteil ( $R_4$ ) der Ausgangsspannung des OPV bestimmt. Aus dieser Art der Rückführung entsteht eine Schmitt-Triggerschaltung welche den Vorteil hat, dass sie unempfindlich gegen Schwankungen innerhalb des Hysteresebandes ist (Schwankungen wie z.B. Rauschen). So wird ein unregelmäßiges Schaltverhalten am OPV-Ausgang verhindert und ein sauberes Ausgangssignal erzeugt (siehe Abbildung 57).

Wie bereits erwähnt kann der Ausgang des OPV im Idealfall seine Versorgungsspannung  $V_+$  und  $V_-$  annehmen. Da das verwendete OPV-Modell nicht über einen Rail-to-Rail Ausgang

verfügt, bleibt die maximale Ausgangsspannung stets mit einem nicht zu vernachlässigbaren Abstand unter der Versorgungsspannung. Bei einer OPV-Versorgungsspannung  $V_+$  von 3,3V werden am Ausgang des OPV ungefähr 2V gemessen. Dies führt dazu, dass das Phasen-Signal im unteren und oberen Drehzahlbereich die Referenzspannung  $V_{ref}$  von 540mV am invertierenden Eingang des OPV nicht durchschreitet. Dadurch wird am OPV-Ausgang kein Rechtecksignal mehr erzeugt. Darüber hinaus ist im zugehörigen Datenblatt des verwendeten Mikrocontrollers ersichtlich, dass die Capture Input Pins vom Timer Modul CTIMER2 (J13 Pin4 / Port-Pin P0.25 für Messung rechts, J13 Pin6 / Port-Pin P0.24 Messung links) eine Mindestspannung von 2,0V benötigen, um zuverlässig einen HIGH-Pegel erkennen zu können. Da dieser Spannungswert insbesondere im niedrigen Drehzahlbereich nie erreicht wird, muss die Versorgungsspannung des OPV hochgesetzt werden. In Abbildung 57 wird die Abweichung bei einer Versorgungsspannung von 5V deutlich. Das Ausgangssignal des Operationsverstärkers (Abbildung 57, gelb) erreicht einen Spannungspegel von 3.85V. Durch die Erhöhung der Versorgungsspannung kann also grundsätzlich eine höhere Ausgangsspannung erzielt werden.

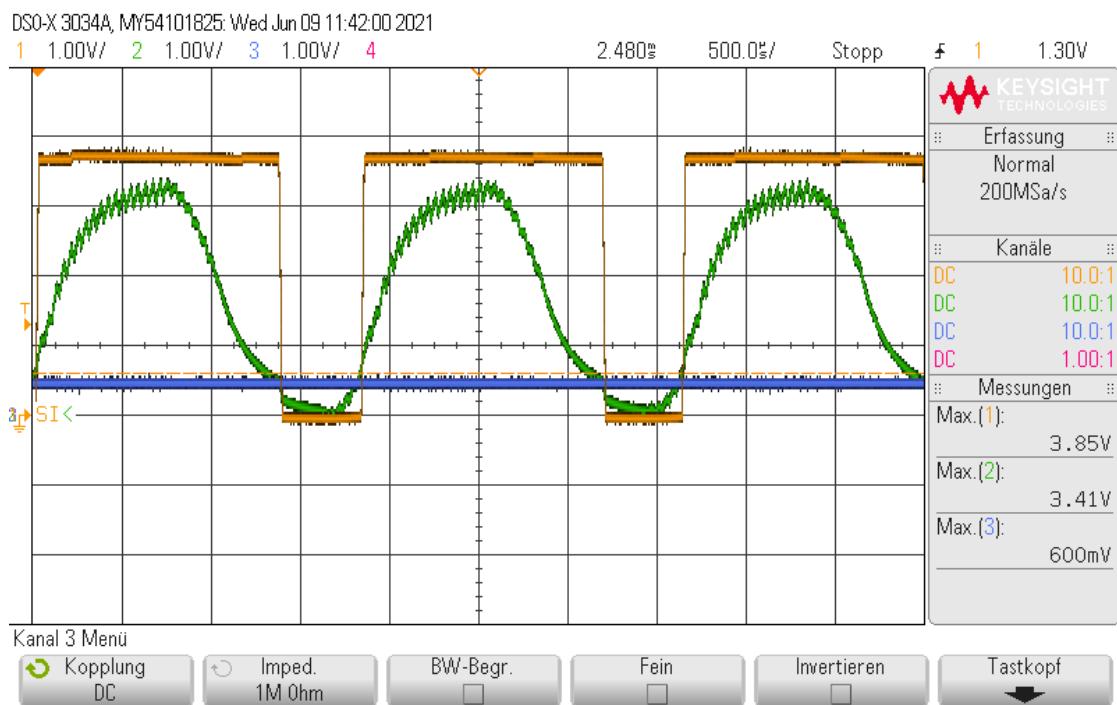


Abbildung 57: Beispielhafte Messung des OPV-Ausgangssignals (gelb) bei einer Versorgungsspannung von  $V_+ = 5,0\text{V}$ ; IN+ in grün und Vref an IN- in blau

Damit eine Signalspannung mit mehr als 2,0 V für die Capture Input Pins des Timer Moduls CTIMER2 am Mikrocontroller zuverlässig bereit gestellt wird, wird ein n-Kanal MOSFET (ZVN4210A) an jedem der OPV-Ausgänge platziert. Der Widerstand  $R_9 = 10 \text{ k}\Omega$  vor dem Drain-Anschluss des MOSFETs in Abbildung 53 steht stellvertretend für den internen Pull-up Widerstand der TIMER2 Capture Input Pins des Mikrocontrollers. Je nach Konfiguration können diese zu- oder abgeschalten werden.

Für die Bereitstellung der am invertierenden OPV-Eingang (IN-) benötigten Referenzspannung, auch Schwellspannung genannt, wird ein Spannungsteiler mit einer vom Pin12 der Buchsenleiste J10 stammenden +3,3V Spannungsversorgung vorgeschalten. Der Spannungsteiler besteht aus drei parallel geschalteten Widerständen  $R_5 = 100\text{k}\Omega$  und  $R_6 = R_7 = 220\text{k}\Omega$  in Reihe zum Widerstand  $R_8 = 10\text{k}\Omega$ . Die dadurch erzeugte Referenzspannung beträgt etwa 540mV. Diese Referenzspannung ist in der Theorie ausreichend, um für alle Drehzahlen ein zuverlässiges Rechtecksignal zu erzeugen. Trotz aller bisher getroffenen Optimierungen treten dennoch immer wieder Probleme bei der Drehzahlmessung auf. Die Problematik sowie deren Lösung werden im Kapitel 4.5.5 Ausblick Schaltungserweiterung beschrieben.

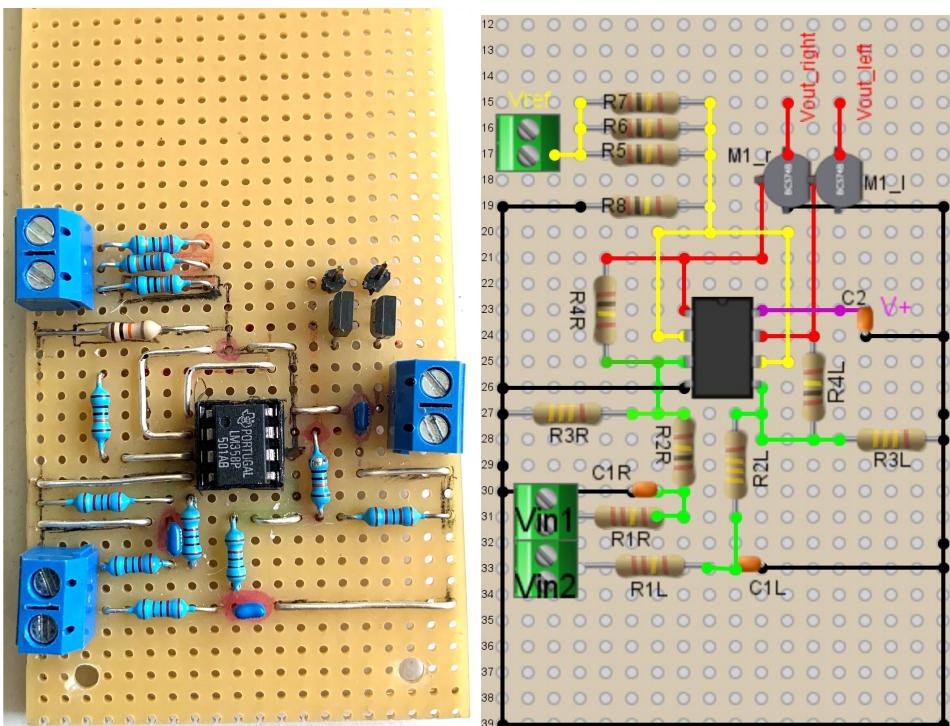


Abbildung 58: Schaltung der Drehzahlmessung auf der Signalverteilerplatine (links) und zugehöriges Layout (rechts)

Auf der linken Seite der Abbildung 58 ist die auf der Signalverteilerplatine befindliche Drehzahl-Messschaltung im Original zu sehen. Auf der rechten Seite der Abbildung 58 befindet sich das dafür mit dem Programm BlackBoard Circuit Designer erstellte, zugehörige Layout. Die grünen Verbindungen stellen die Pfade zwischen  $V_{in}$  (Phasensignale) und den OPV Eingängen  $IN1+$  und  $IN2+$  dar. Gelbe Linien repräsentieren den Pfad der Referenzspannung  $V_{ref}$  zu den invertierenden Eingängen  $IN1-$  und  $IN2-$ . Der Pfad zur Spannungsversorgung des OPV ( $V_+$ ) ist in violett dargestellt und alle schwarzen Verbindungen stellen GND-Verbindungen dar. Zuletzt kann der Verlauf der Ausgangsspannungen vom OPV an die Capture Eingänge des Timers CTIMER2 anhand der roten Verbindungen nachvollzogen werden.

#### 4.5.4 Programmierung des Drehzahlmessungsbausteins

Damit die durch die Schaltung bereitgestellten Rechtecksignale (links, rechts) vom Mikrocontroller erfasst werden können, werden die Capture Register des Timer-Moduls CTIMER2 verwendet. Dieses Modul löst bei einem Flanken-Ereignis (steigende Flanke) am jeweiligen Capture Input Pin einen Interrupt aus, in welchem die verstrichene Zeit seit der letzten steigenden Flanke errechnet wird und speichert dann den aktuellen Timerwert ab.

Der Softwarebaustein zur Drehzahlerfassung ist in zwei Dateien unterteilt, die Dateien „rpmMeas.c“ und „rpmMeas.h“. Die Datei „rpmMeas.h“ enthält alle relevanten Bibliotheken und Prototypen für die Datei „rpmMeas.c“ (siehe Abbildung 59). Hier werden auch die beiden Kanäle zur Erfassung der Daten auf das Timer Caputure Register 0 (rechter Antrieb, „kCTIMER\_Capture\_0“) und auf das Timer Capture Register 1 (linker Antrieb, „kCTIMER\_Capture\_1“) festgelegt. Bei dem verwendeten Mikrocontroller entspricht dies den Pins P0.25 (rechts) und P0.24 (links). Der Pin P0.25 wird auf der Controllerplatine über den Pin4 und der Pin P0.24 über den Pin6 der Buchsenleiste J13 nach außen geführt.

```

/* Definition of channel 0 ID for right BLDC*/
#define CTIMER2_RPMMEAS_LEFT_CHANNEL KCTIMER_Capture_0

/* Definition of channel 1 ID for left BLDC*/
#define CTIMER2_RPMMEAS_RIGHT_CHANNEL KCTIMER_Capture_1

/*************************
 * Prototypes
 *****/
void RPMMEAS_Init(void);
void CTIMER2_Init(void);
void rpmMeas_Left_ISR(uint32_t);
void rpmMeas_Right_ISR(uint32_t);
uint32_t rpmMeas_average_val_left(uint32_t a[]);
uint32_t rpmMeas_average_val_right(uint32_t b[]);
```

Abbildung 59: Relevante Zeilen der Datei „rpmMeas.h“ zur Initialisierung des Timers CTIMER2 für die Zeiterfassung zwischen zwei Flanken Signal sowie die Prototypen für die Funktionen aus der Datei „rpmMeas.c“

Die Datei „rpmMeas.c“ enthält die Funktionen zur Initialisierung der für die Verwendung der Drehzahlerfassung notwendigen Controller-Peripherie als auch die Definition und Initialisierungen der notwendigen Parameter zur Erfassung und Berechnung der benötigten Drehzahlwerte (siehe Abbildung 60).

In der Funktion RPMMEAS\_Init wird zuerst die Funktion CTIMER2\_Init aufgerufen, welche unter anderem die beiden vorher festgelegten Kanäle des Timers CTIMER2 („kCTIMER\_Capture\_0“ und „kCTIMER\_Capture\_1“) initialisiert. Bei der Initialisierung der Capture Channel (CTIMER\_SetupCapture) wird das entsprechende Ereignis, bei dem ein Capture stattfinden soll, festgelegt. In diesem Fall wird dieses Ereignis für beide Kanäle auf die steigende Flanke gesetzt und die Capture Interrupt Requests aktiviert („kCTIMER\_Capture\_RiseEdge“).

```

#include "rpmMeas.h"

const ctimer_config_t RPMMEAS_config = {
    .mode = kCTIMER_TimerMode, /* TC is incremented every rising APB bus clock edge */
    .input = kCTIMER_Capture_0, /*!< Timer capture channel 0 */
    .prescale = 0 /*!< Prescale value 0 --> */
};

/* **** Parameters **** */
volatile uint32_t oldvalleft = 0;
volatile uint32_t diffleft = 0;
volatile uint32_t oldvalright = 0;
volatile uint32_t diffright = 0;
volatile uint32_t sumleft = 0;
volatile uint32_t averageval_left = 0;
volatile uint32_t sumright = 0;
volatile uint32_t averageval_right = 0;

uint32_t measleft[100] = {0}; /*array for average value determination left BLDC*/
uint8_t ileft = 0; /*increment array number left side*/
uint32_t measright[100] = {0}; /*array for average value determination right BLDC*/
uint8_t iright = 0; /*increment array number right side*/

/*CTIMER callback table contains functions for Match Event 0 to 3 and Capture Event 0 to 3*/
ctimer_callback_t ctimer_callback_table[] = {NULL, NULL, NULL, NULL, rpmMeas_Left_ISR, rpmMeas_Right_ISR, NULL, NULL};

/* **** Speed measurement main initialization function **** */
void RPMMEAS_Init()
{
    CTIMER2_Init();
    //Configure Pin P[0][24] (J13 Pin6) (Left BLDC Phase 1)
    IOCON->PIO[0][24]     &= 0xFFFFFFFF0;      //clear FUNC bits of P0.24
    IOCON->PIO[0][24]     |= 0x3;           //set FUNC bits to CTIMER2_CAP0 function FUNC3 P0.24
    IOCON->PIO[0][24]     |= 0xB2<<4;
    GPIO->DIR[0]          &= ~1<<12;        //set P0.24 pin to input

    //Configure Pin P[0][25] (J13 Pin4) (Right BLDC Phase 1)
    IOCON->PIO[0][25]     &= 0xFFFFFFFF0;      //clear FUNC bits of P0.25
    IOCON->PIO[0][25]     |= 0x3;           //set FUNC bits to CTIMER2_CAP1 function FUNC3 P0.25
    IOCON->PIO[0][25]     |= 0xB2<<4;
    GPIO->DIR[0]          &= ~1<<12;

    //set P0.25 pin to input
    //*****
}

/* **** TIMER2 Initialization function **** */
void CTIMER2_Init(void)
{
    /* CTIMER1 peripheral initialization */
    CTIMER_Init(CTIMER2, &RPMMEAS_config);
    /* Set Callback functions */
    CTIMER_RegisterCallBack(CTIMER2, &ctimer_callback_table[0], kCTIMER_MultipleCallback);
    /*Setup Capture Left Drive Capture Channel 0*/
    CTIMER_SetupCapture(CTIMER2, CTIMER2_RPMMEAS_LEFT_CHANNEL, kCTIMER_Capture_RiseEdge, 1);
    /*Setup Capture Right Drive Capture Channel 1*/
    CTIMER_SetupCapture(CTIMER2, CTIMER2_RPMMEAS_RIGHT_CHANNEL, kCTIMER_Capture_RiseEdge, 1);
    /* Start the timer */
    CTIMER_StartTimer(CTIMER2);
}

```

Abbildung 60: Funktionen RPMMEAS\_Init und CTIMER2\_Init in der Datei „rpmMeas.c“

Damit beim Auftreten einer steigenden Flanke der Rechtecksignale ein Interrupt Request ausgelöst werden kann, benötigt man zunächst den Parameter cTimer\_callback\_table. Die Callback-Table enthält die bei verschiedenen Events (Match0-3 und Capture0-3) aufzurufenden Interrupt Service Routinen rpmMeas\_Left\_ISR (Capture Event 0) und rpmMeas\_Right\_ISR (Capture Event 1). Die Callback-Table wird als Pointer-Array der Funktion CTIMER\_RegisterCallBack übergeben. Bei einer Callback-Funktion, auch Rückruffunktion

genannt, wird einer Funktion als Parameter eine weitere Funktion oder wie hier eine Liste von Funktionen übergeben. Des weiteren muss in der RegisterCallBack-Funktion der Typ des Callbacks festgelegt werden. In diesem Fall wird der Callback als „kCTIMER\_MultipleCallback“ festgelegt, was bedeutet, dass pro Kanal ein Callback durchgeführt werden kann. Zuletzt wird in der CTIMER2\_Init der Timer CTIMER2 gestartet.

Nach Vollendung der Initialisierungen in der Funktion CTIMER2\_Init werden in der Funktion RPMMEAS\_Init zusätzlich die Pins P0.24 (Messung links) und P0.25 (Messung rechts) für die Verwendung als Capture Input Pin des Timers CTIMER2 konfiguriert.

Zur Erfassung der Drehzahl wird im Programm eine Interrupt Service Routine rpmMeas\_Left\_ISR und rpmMeas\_Right\_ISR für die jeweilige Antriebsseite benötigt. Sobald auf eine steigende Flanke getriggert wird, wird durch das Setzen eines Interrupt Flags die Interrupt Service Routine ausgelöst. In der jeweiligen Service Routine wird der aktuelle Zählerwert des Timers CTIMER2 mit dem Befehl CTIMER\_GetTimerCountValue ausgelesen, der Wert bei der letzten Flanke abgezogen und somit die verstrichene Zeit zwischen diesen Flanken berechnet (diffright und diffleft) und den Arrays zur Mittelwertberechnung (measright[ ], measleft[ ]) übergeben.

```
/****************************************************************************
 * Interrupt Service Routine for rising edge of Phase 1 of left BLDC
 ****/
void rpmMeas_Left_ISR(uint32_t flags)
{
    diffleft = CTIMER_GetTimerCountValue(CTIMER2) - oldvalleft;
    measright[iright] = diffleft;
    oldvalleft = CTIMER_GetTimerCountValue(CTIMER2);
    if(ileft <= 100)
    {
        ileft++;
    }
    else
    {
        ileft = 0;
    }
}

/****************************************************************************
 * Interrupt Service Routine for rising edge of Phase 1 of right BLDC
 ****/
void rpmMeas_Right_ISR(uint32_t flags)
{
    diffright = CTIMER_GetTimerCountValue(CTIMER2)-oldvalright;
    measright[iright] = diffleft;
    oldvalright = CTIMER_GetTimerCountValue(CTIMER2);

    if(iright <= 9)
    {
        iright++;
    }
    else
    {
        iright = 0;
    }
}
```

Abbildung 61: Interrupt Service Routinen für Zeitermittlung zwischen zwei Flanken der Capture Inputs des Timers CTIMER2

Von den in den Arrays („measright[ ]“ und „measleft[ ]“) gespeicherten Werten können in den Funktionen „rpmMeas\_average\_val\_right()“ und „rpmMeas\_average\_val\_left()“ die Mittelwerte gebildet werden. Die Mittelwerte „average\_val\_right“ und „average\_val\_left“ werden in den weiteren Schritten zur Implementierung der Drehzahlregelung benötigt.

```

/************************
*Calculation function for average rpm value left BLDC
****************************/
uint32_t rpmMeas_average_val_left(uint32_t measleft[])
{
    uint8_t i = 0;
    sumleft = 0;

    for(i=0; i < 10; i++){
        sumleft += measleft[i];           //sum all array elements left BLDC
    }

    averageval_left = sumleft/10;
    return averageval_left;           //return average value left BLDC
}

/************************
*Calculation function for average rpm value right BLDC
****************************/
uint32_t rpmMeas_average_val_right(uint32_t measright[])
{
    uint8_t j = 0;
    sumright = 0;

    for(j=0; j < 10; j++)
    {
        sumright += measright[j];      //sum all array elements right BLDC
    }

    averageval_right = sumright/10;
    return averageval_right;         //return average value right BLDC
}

```

Abbildung 62: Funktionen zur Berechnung des Mittelwerts der erfassten Zeiten zwischen den Flanken der Capture Inputs des Timers CTIMER2

#### 4.5.5 Ausblick Schaltungserweiterung

Nachdem die Referenzspannung Vref in der Schaltung so dimensioniert ist, dass in der Theorie alle Drehzahlen zuverlässig als eindeutiges Rechtecksignal dargestellt werden können, treten in der Praxis dennoch Probleme auf. Drehzahlen im sehr niedrigen Drehzahlbereich (ca. 1,09 ms bis 1,10ms PWM-Pulsbreite) oder auch im höheren Drehzahlbereich (über 1,5ms) werden nicht immer zuverlässig zu einem eindeutiges Rechtecksignal verarbeitet, da die Referenzspannung keine eindeutigen Schnittpunkte mit dem Eingangssignal besitzt (siehe Abbildung 63). Auch ein Abfallen der Batteriespannung bei längerem Betrieb des Fahrzeugs kann ebenfalls das Problem verursachen, dass verschiedene Drehzahlen (vor allem an den Maxima) nicht mehr detektiert werden können.

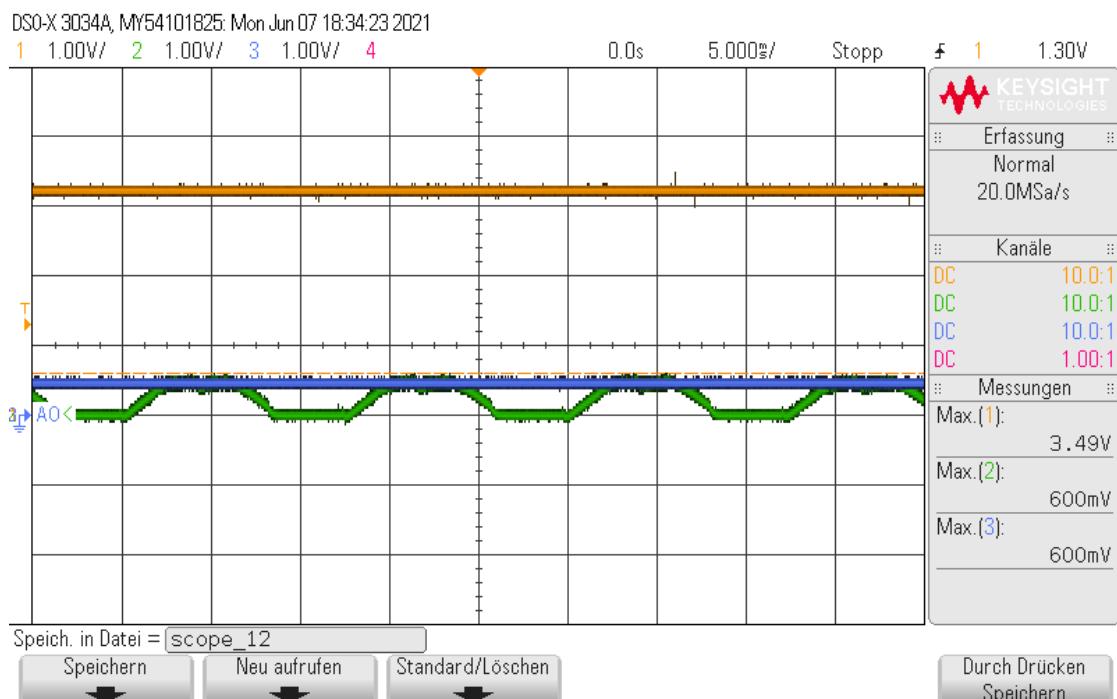


Abbildung 63: Fehlerhafte Rechtecksignal-Erzeugung (in diesem Fall kein Rechteck) bei niedrigen Drehzahlen, wenn die Schaltschwellen nicht erreicht werden

Eine mögliche Lösung der Problematik ist die Erweiterung des Spannungsteilers für die Referenzspannung des OPV um eine zusätzliche und zuschaltbare Teilerstufe (1x für hohe Drehzahlen; 1x für niedrige Drehzahlen) mit einem n-Kanal MOSFET gegen Masse (siehe Abbildung 64). Diese Maßnahme ermöglicht es, die Referenzspannung, für hohe und niedrige Drehzahlen flexibel anzupassen um in jedem Fall ein zuverlässiges Ausgangssignal für die Messung der Drehzahl zu erhalten.

Durch die Vorgabe eines bestimmten Modus (MCU-Mode), kann durch die Definition von zwei Drehzahlbereichen (z.B. Pulsbreite 1,09ms - 1,43ms langsam; Pulsbreite 1,43ms - 1,9ms schnell) eine individuelle Referenzspannung von ca. 500mV für langsame Drehzahlen

und min. 1V für hohe Drehzahlen eingestellt werden. Der Vorteil ist, dass man dadurch einen bestimmten Toleranzbereich für Schwankungen der Versorgungsspannung erzeugt und die Schaltung dadurch weniger anfällig für Abweichungen ist.

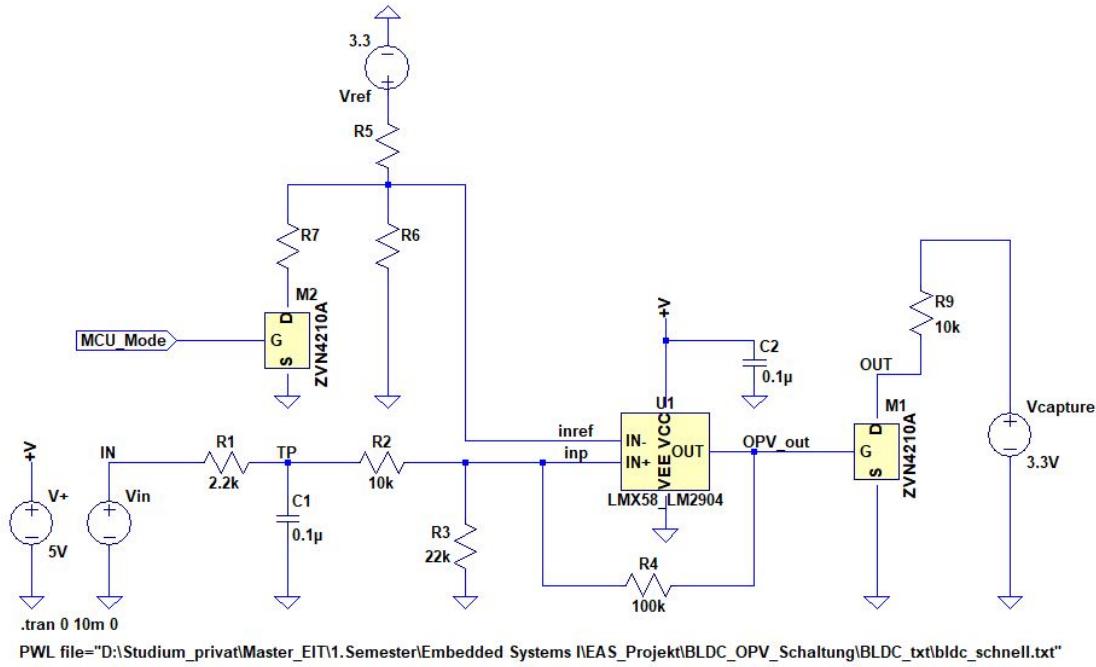


Abbildung 64: Schaltplan der Tiefpass-/Komparatorschaltung nach der Schaltungserweiterung um eine zweite Referenzspannungsstufe; erstellt für die Simulation eines Phasen-signal in LTspice XVII

In der Software muss hierzu in der Interrupt Service Routine der Drehzahlwert-Erfassung eine Umschalterkennung programmiert werden. Diese stellt über einen Ausgang des Mikrocontrollers die Referenzspannungsstufen über einen n-Kanal MOSFET so ein, wie sie bei der aktuellen Drehzahl gerade benötigt werden. Zur Bestimmung der Widerstandswerte für R5, R6 und R7 können die Gleichungen 3 und 4 verwendet werden.

$$V_{ModeLow} = \frac{R6}{R6 + R5} \cdot 3,3V \quad (3)$$

$$V_{ModeHigh} = \frac{R7||R6}{(R7||R6) + R5} \cdot 3,3V \quad (4)$$

## 5 Lenkung des Fahrzeugs

Wie in Kapitel 2.1 beschrieben, wird das Lenkgestänge des Fahrzeugs von einem Servomotor angetrieben, der im Folgenden näher erklärt wird. Auch auf die Montage der Antriebskomponenten und die Programmierung des Lenkungsbausteins wird in diesem Kapitel näher eingegangen.

### 5.1 Servoantrieb

Zur Lenkung des Fahrzeugs wird ein Servomotor (siehe Abbildung 65) verwendet, der mit den Achsen über ein Lenkgestänge verbunden ist. Ein Servo besteht aus einem Motor, dessen Welle mit einem Potentiometer verbunden ist. Die interne Elektronik regelt automatisch die Istwert-Vorgabe des Potentiometers auf den an der Signalleitung anliegenden Sollwert aus. Dadurch können Lenkwinkelpositionen genau angefahren werden. Die Sollwert-Vorgabe erfolgt, wie die Drehzahlvorgabe bei den BLDC-Motoren, mit einem 50Hz PWM-Signal und einem Tastgrad von 5% bis 10%, wobei dieser proportional zum Lenkwinkel ist. Bei einem Tastgrad von 7,5% befindet sich die Lenkung in der Mittelstellung. Kleinere Werte bewirken einen Lenkeinschlag nach links und größere Werte einen Einschlag nach rechts. Der PWM-Signalleitung und der dazugehörigen Masseleitung schließt sich noch eine Versorgungsleitung an (5V-Versorgung).



Abbildung 65: Servomotor mit Anschlussleitungen; Versorgungsspannung rote Leitung, PWM-Signal orangene Leitung und Masseleitung in braun [DIY21]

## 5.2 Montage der Lenkungskomponenten

Die Komponenten der Lenkung, welche über einen Servomotor und ein mit den Reifen verbundenes Lenkgestänge realisiert ist, sind wie die Antriebskomponenten auf der unteren Fahrzeugebene montiert. Der Servomotor ist mit vier Schrauben an einem losen Teil des Chassis befestigt, welches über zwei Schrauben durch Langlöcher von oben am Fahrzeug montiert ist. Über die Langlöcher kann der Servomotor so platziert werden, dass die Vorderreifen parallel zueinander stehen. An der Welle des Servomotors ist dann das Lenkgestänge befestigt, welches mit den Reifen verbunden ist.

In Abbildung 66 sind die montierten Komponenten der Fahrzeuglenkung abgebildet. Die Befestigung des Servomotors ist in violett hervorgehoben, die Befestigung des losen Chassis-Teils in rot, die Lenkgestänge in blau und die Federung in orange.

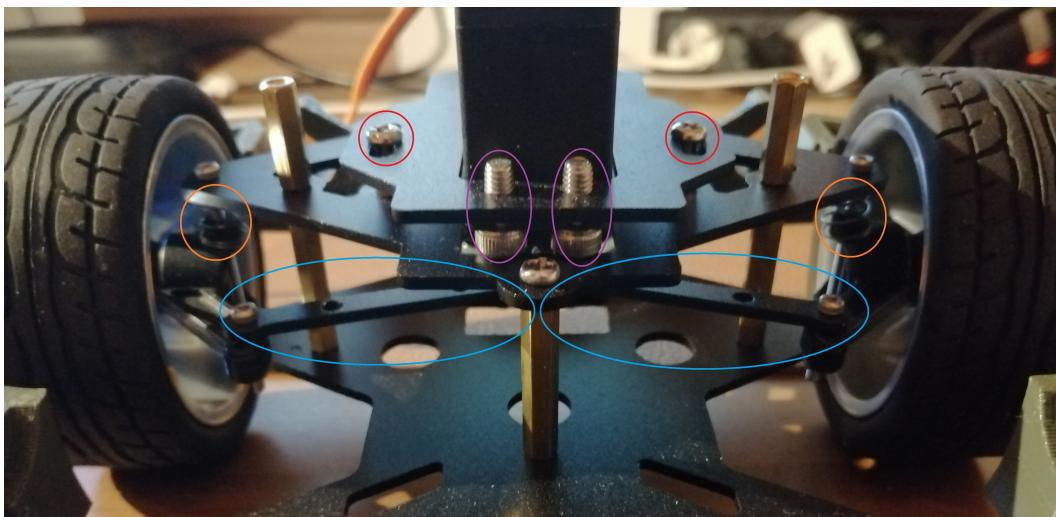


Abbildung 66: Montage des Servo-Motors, des Lenkgestänges und der Vorderreifen; Befestigung des Servo-Motors in violett, Befestigungsschrauben des losen Chassis-Teils in rot, Lenkgestänge in blau und Federung in orange

### 5.3 Programmierung des Lenkungsbausteins

Der Lenkungsbaustein der Software ist wie der Antriebsbaustein in zwei Dateien unterteilt, die Dateien „servo.c“ und „servo.h“. Die Datei „servo.h“ enthält alle relevanten Bibliotheken und Prototypen für die Datei „servo.c“. Außer der Einbindung der Bibliotheken und der Prototypen der Funktionen aus der Datei „servo.c“ sind hier auch die Parameter für die Initialisierung des Timers für das PWM-Signal hinterlegt (Abbildung 67).

```
/****************************************************************************
 * Defines
 ****/
/* Channel definitions */
#define CTIMER1_PWM_0_CHANNEL kCTIMER_Match_2 //Definition of channel 2 ID

/* Servo PWM initialization value definitions */
#define CTIMER1_PWM_0_DUTY 0 //Definition of channel 2 duty --> on ticks
#define CTIMER1_PWM_PERIOD 4400000 //Definition of PWM period --> whole period ticks
```

Abbildung 67: Relevante Zeilen der Datei „servo.h“ mit den Parametern für die Initialisierung des PWM-Timers

Wie schon beim Antrieb, wird für die PWM-Periodendauer bei der Initialisierung ein Wert von 4.400.000 Takten festgesetzt, woraus mit einer CPU-Taktfrequenz von 220MHz (220.000.000 Takte pro Sekunde) eine Periodendauer von 20ms resultiert. Die Pulsbreite wird während des Programmablaufs regelmäßig überschrieben, weshalb für die Initialisierung ein Wert von 0 gewählt werden kann. So bewegt sich der Servomotor bei der Initialisierung nicht und es kommt dabei nicht zum Crash.

Der Wert für den linken maximalen Lenkeinschlag (ca. 200.000) entspricht einer PWM-Pulsbreite von 0,91ms und der des maximalen rechten Lenkeinschlags (ca. 459.000) einer Breite von 2,08ms. Auch wenn der Servomotor einen Wert zwischen 1ms und 2ms erwartet, kann mit den eben genannten Einstellungen noch ein wenig mehr Lenkeinschlag gewonnen werden. Mit dem Wert der Mittelstellung (ca. 327.000), welcher durch Probieren angenähert wurde, beläuft sich die Pulsbreite auf 1,486ms. Alle drei Extremwerte („servoMiddleValue“, „servoLeftValue“, „servoRightValue“) sind Parameter, deren Werte aus dem EEPROM kommen und welche in der Einheit µs über das Bedienungsboard eingestellt werden können (siehe Abbildung 68).

```
int32_t* servoMiddleValue= &((all_param_t*)&const_all_param)->motors.servo.init; //servo init value
int32_t* servoLeftValue= &((all_param_t*)&const_all_param)->motors.servo.min; //servo min value
int32_t* servoRightValue= &((all_param_t*)&const_all_param)->motors.servo.max; //servo max value
```

Abbildung 68: Extremwerte der Lenkwinkel als Parameter aus dem EEPROM, deren Werte über das Bedienungsboard individuell einstellbar sind; Teil der Datei „servo.c“

Auch der PWM-Timer benötigt bei der Initialisierung einige Parameter, deren Werte in der Datei „servo.h“ festgelegt sind (PWM-Periodendauer, PWM-Pulsdauer, Channel). Der Channel wird auf das Timer Match Register 2 festgelegt („kCTIMER\_Match\_2“), was bei dem verwendeten Controller dem Pin P3.2 entspricht. Der Pin P3.2 wird auf der Controllerplatine über den Pin 11 der Buchsenleiste J13 nach außen geführt.

```
*****
 * Parameters
 *****
const ctimer_config_t SERVO_config = {
    .mode = kCTIMER_TimerMode, /* TC is incremented every rising APB bus clock edge */
    .input = kCTIMER_Capture_0, /*!< Timer capture channel 0 */
    .prescale = 0, /*!< Prescale value 0 --> */
};

*****
 * Servo Main Initialization function
 *****
void SERVO_Init(void)
{
    CTIMER1_Init(); //TIMER1 Initialization

    //Configure Pin P[3][2] (J13 Pin11) (Servo PWM)
    CTIMER1->MCR |= CTIMER_MCR_MR2RL_MASK; //Reload MR2 with content of shadow register at timer overflow
    IOCON->PIO[3][2] &= 0xFFFFFFFF0; //Clear FUNC bits of P3.2
    IOCON->PIO[3][2] |= 0x4; //Set FUNC bits to CTIMER1_MAT2 function ALT4 P3.2
    GPIO->DIR[3] |= 1<<2; //Set P3.2 pin to output
    CTIMER1->MSR[2] = CTIMER1_PWM_PERIOD - (*servoMiddleValue)*CTIMER1_PWM_PERIOD/20000; //Initialize MSR with SERVO_PWM_Middle_Value value
//*****


    //*****Create SERVO_Demo task
    if (xTaskCreate(SERVO_Demo, "SERVO_Demo", configMINIMAL_STACK_SIZE + 100, NULL, 1, NULL) != pdPASS)
    { LED3_ON(); } //LED3 is Error
    //*****


}

*****
 * TIMER1 Initialization function
 *****
void CTIMER1_Init(void)
{
    /* CTIMER1 peripheral initialization */
    CTIMER_Init(CTIMER1, &SERVO_config);

    /* PWM channel 2 of CTIMER1 peripheral initialization */
    CTIMER_SetupPwmPeriod(CTIMER1, CTIMER1_PWM_0_CHANNEL, CTIMER1_PWM_PERIOD, CTIMER1_PWM_PERIOD - CTIMER1_PWM_0_DUTY, false);

    /* Start the timer */
    CTIMER_StartTimer(CTIMER1); //Timer starten
}
```

Abbildung 69: Funktionen SERVO\_Init und CTIMER1\_Init und Demonstrations-Task SERVO\_Demo der Datei „drive.c“

Die Datei „servo.c“ enthält die Funktionen zur Initialisierung der für die Verwendung des Servos notwendigen Controller-Peripherie. In der Funktion SERVO\_Init wird zuerst die Funktion CTIMER1\_Init aufgerufen, welche den Timer mit den in der Datei „servo.h“ festgelegten Parametern als PWM-Timer mit einer Periodendauer von 20ms und einer Pulslänge von 0ms initialisiert. Im Anschluss daran wird festgelegt, dass bei einem Timer-Überlauf die neuen Daten für die Pulslänge aus dem Shadow-Register geladen werden. Zum Ändern des Lenkwinkels muss deshalb lediglich ein neuer Wert in das Shadow-Register geschrieben werden. Hier muss allerdings aufgepasst werden, da das Register nicht die Pulsbreite (On-Time) sondern die Off-Time erwartet. Deshalb muss der Wert, der eingetragen wird, der Periodendauer abzüglich der Pulsdauer entsprechen. Zusätzlich wird in der Funktion SERVO\_Init auch der Pin 3.2 für die Verwendung als PWM-Ausgang des Timers CTIMER1 konfiguriert (siehe Abbildung 69).

Am Ende wird der Wert für die Mittelstellung des Servos in das Shadow-Register geschrieben und ein Task für die Demonstration der Lenkung gestartet (SERVO\_Demo), welcher nach einmaligem Linksslenken, Rechtslenken und nach der Rückkehr in die Mittelstellung ausgesetzt wird. Der Task „SERVO\_Demo“ ist in Abbildung 70 einsehbar.

```
/*****************************************************************************  
 * Servo Steering Demonstration Task  
*****/  
void SERVO_Demo(void *pvParameters)  
{  
  
    while(1)  
    {  
        //*****  
        //Demo Sequence for Servo-drive  
        CTIMER1->MSR[2] = CTIMER1_PWM_PERIOD - (*servoLeftValue);      //Steer left  
        vTaskDelay(500);  
        CTIMER1->MSR[2] = CTIMER1_PWM_PERIOD - (*servoRightValue);     //Steer right  
        vTaskDelay(800);  
        CTIMER1->MSR[2] = CTIMER1_PWM_PERIOD - (*servoMiddleValue); //Steering angle zero degree  
        //*****  
  
        vTaskSuspend(NULL); //suspend Task  
    }  
}
```

Abbildung 70: SERVO\_Demo Task der Datei „drive.c“

## 6 Bedienungs-Board

Das Bedienungs-Board ist auf der oberen Fahrzeugebene über dem Controller verbaut. Es enthält ein Display, einen Drehencoder mit Taster, einen separaten Taster und einen Summer. Diese Komponenten dienen zum einen der Eingabe von Parametern und der Bedienung des Fahrzeugs durch den Benutzer und zum anderen zum Ablesen der Fahrzeugdaten, wie beispielsweise der Streckenerkennungsdaten der Kamera.

### 6.1 Schaltplan

Das Bedienungs-Board ist eine selbst bestückte Lochrasterplatine, die die Anschlüsse der einzelnen Komponenten auf zwei Stifteleisten zusammenführt. In den Abbildungen 71 und 72 ist der Schaltplan des Bedienungs-Boards zusehen. Die Buchsenleisten J3 und J4 sind die Anschlüsse des Displays. An die Buchsenleisten J5 und J6 sind die Pins des Drehencoders geführt. Dieser gibt zwei Signale (ENC\_A, ENC\_B) für die Erkennung der Drehrichtung und ein Signal (ENC\_SW) für den Taster aus. Die Signale des Displays, des Drehencoders, des Tasters und des Summers werden an die Stifteleisten J1 und J2 geführt, über die die Verbindung zur Controllerplatine hergestellt wird. Die Einbindung des Bedienungs-Boards in das Gesamtsystem des Fahrzeugs ist über den Anhang „Anhang 1: Schaltplan“ nachvollziehbar.

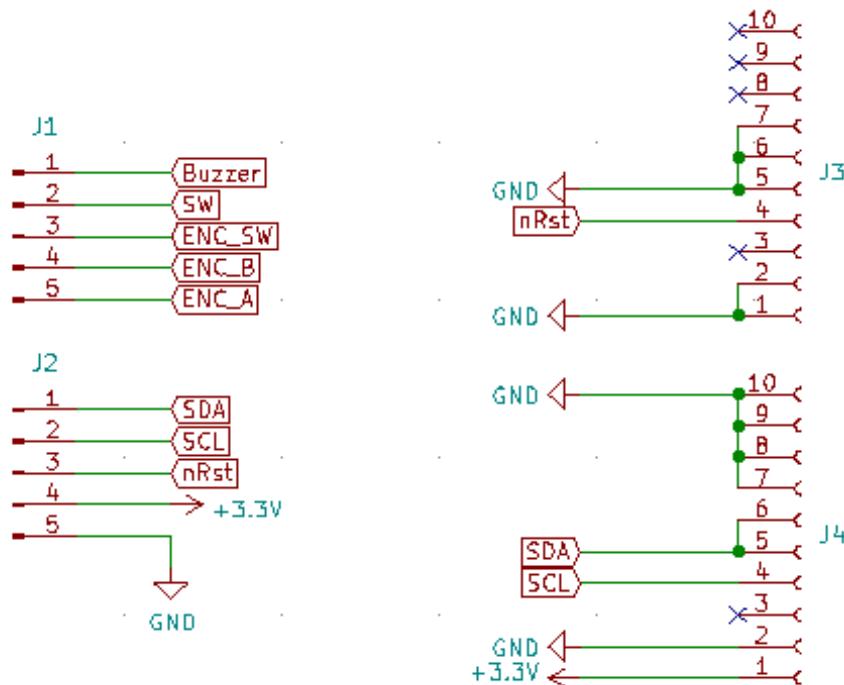


Abbildung 71: Schaltplan des Bedienungs-Boards mit den Stifteleisten J1 und J2 zur Verbindung mit dem Mikrocontroller und den Buchsenleisten J3 und J4 für das Display

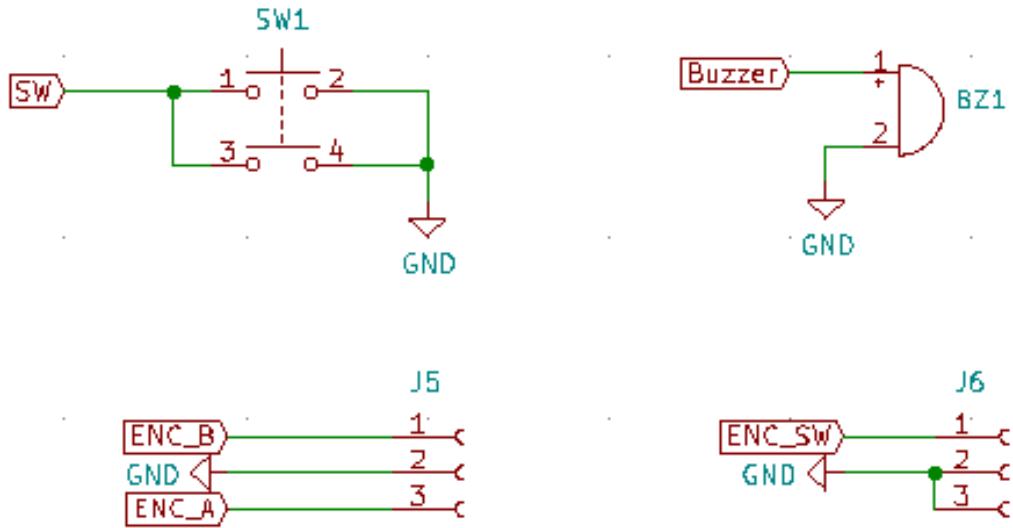


Abbildung 72: Schaltplan des Bedienungs-Boards mit dem Drehencoder, dem Taster und dem Summer

## 6.2 Programmierung der Steuerelemente

Die drei Signale des Drehencoders ENC\_A, ENC\_B und ENC\_SW und das Signal SW des Tasters sind als Pulldown-Anschlüsse ausgeführt. Diese Signale werden im Mikrocontroller über einen internen Pull-Widerstand auf 3,3 V gezogen. Die beiden Signale ENC\_A und ENC\_B liefern zwei um 90° verschobene Rechtecksignale, dessen Frequenz sich mit der Drehgeschwindigkeit ändert. Durch den Phasenversatz um 90° kann die Drehrichtung bestimmt werden. Dabei wird bei einer Flanke vom Signal „ENC\_A“ der Spannungspegel vom Signal „ENC\_B“ bestimmt. Aus der Information, ob es sich um eine fallende oder steigende Flanke des Signals ENC\_A handelt und ob der Spannungspegel des Signals ENC\_B 3,3 V oder 0 V ist, wird die Drehrichtung bestimmt.

Das Signal Buzzer lässt bei anlegen einer Spannung den internen Quarz des Summers mit seiner charakteristischen Frequenz schwingen. Dadurch können Eingaben des Benutzers akustisch bestätigt werden.

### 6.2.1 Taster auswerten

Im Flussablaufdiagramm in Abbildung 73 ist zu erkennen, dass beide Taster mit einer Abtastzeit von 10 ms eingelesen werden. Da Taster mechanische Bauteile sind, kommt es bei jedem Betätigen zu ungewollten Pegelwechseln, dem sogenannten Prellen. Dies muss verhindert werden, da dadurch ein einzelner Tastendruck als mehrere interpretiert wird. Soll zum Beispiel ein Zähler per Tastendruck inkrementiert werden, wird bei einem prellenden Taster der Zähler pro Druck nicht um einen sondern um mehrere Schritte erhöht. Die Anzahl, um wie viel der Zähler erhöht wird, hängt davon ab, wie viele Pegelwechsel durch das Prellen

erzeugt und mit welcher Frequenz der Taster ausgelesen wird. Um das Prellen zu verhindern, wird auf eine Software-Entprellung zurückgegriffen. Dabei wird ein Tastendruck nur erkannt, wenn sich der Spannungspiegel des Tasters für vier Abtastzyklen nicht ändert. Dies entspricht bei einer Abtastzeit von 10 ms einer Zeit von 40 ms. Bei diesem Verfahren ist es wichtig, dass der Taster mindestens die vierfache Zeit der Abtastzeit betätigt wird. Beim Entprellen wird zuerst abgefragt, ob sich der eingelesene Zustand des Tasters zum vorherigen entprellten Zustand geändert hat. Ist dies der Fall wird ein Zähler dekrementiert. Der Zähler zählt dabei von drei bis null. Erreicht dieser den Wert null oder wird ein Zustand eingelesen, der dem zuvor entprelltem Zustand entspricht, wird der Zähler auf den Wert drei zurückgesetzt. Wechselt der Zähler von null auf drei, wird der Tastendruck als solcher erkannt.

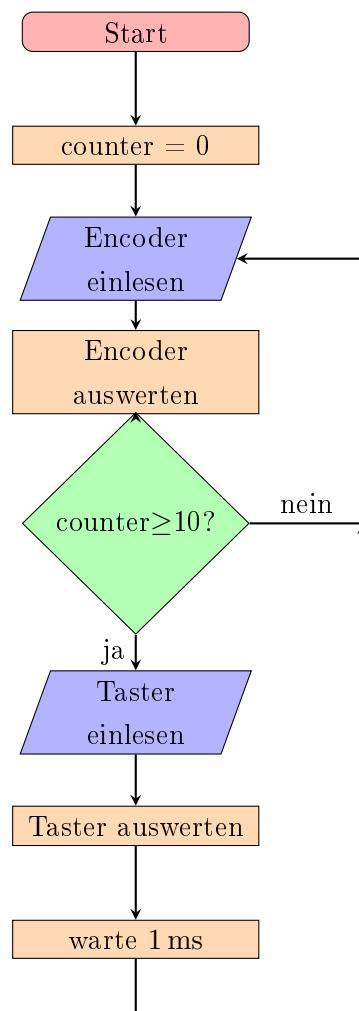


Abbildung 73: Flussablaufdiagramm zum Einlesen des Encoders jede Millisekunde und der Taster alle zehn Millisekunden

Im C-Code wird der Zähler mit zwei 32 Bit Variablen implementiert. Dabei wird aus jeweils einem Bit der beiden Variablen ein 2bit Zähler gebildet. Dadurch entstehen mit nur zwei Variablen insgesamt 32 Zähler, die von drei bis null zählen können. Dies macht jedoch die

Implementierung der Zählfunktion schwieriger, da der Zähler nicht mit einem Postdecrement und if-Abfragen realisiert werden kann. Diese Funktionen müssen durch Bitoperationen ersetzt werden, die die ganze 32 Bit Variable abfragen und ändern können (siehe [mikb], Kapitel „Timer-Verfahren (nach Peter Dannegger)“).

### 6.2.2 Drehencoder auswerten

Im Flussablaufdiagramm in Abbildung 73 werden die beiden um  $90^\circ$  phasenverschobenen Signale des Drehencoders mit einem Zeitabstand von 1 ms ausgelesen. Im Anschluss daran müssen daraus die Drehrichtung und die Anzahl der gedrehten Schritte ermittelt werden. Dazu wird in einer Variable der aktuell gemessene und der jeweils letzte Wert der beiden Drehencoderanschlüsse als je ein Bit gespeichert. Dabei liegt in Bit0 der aktuelle Zustand von ENC\_B, in Bit1 der aktuelle Zustand von ENC\_A, in Bit2 der letzte Zustand von ENC\_B und in Bit3 der letzte Zustand von ENC\_A. Die daraus entstehende 4bit-Zahl bildet den Index eines Arrays, welches zeigt, ob der Drehencoder nach links oder nach rechts gedreht worden ist. Steht in der Variable zum Beispiel der Wert  $1101_2$  ( $13_{10}$ ) bedeutet das, dass die vorherigen Zustände HIGH, der aktuelle Zustand ENC\_A LOW und ENC\_B HIGH sind. Daraus ist abzuleiten, dass eine fallende Flanke des Signals ENC\_A und ein HIGH Pegel am Signal ENC\_B anliegt. In dem Array an der Stelle 13 steht der Wert -1, was eine Drehung im Uhrzeigersinn repräsentiert. Im Falle von  $1000_2$  ( $8_{10}$ ) liegt eine fallende Flanke an ENC\_A und ein LOW Pegel an ENC\_B an. Daraus ergibt sich im Array der Wert 1 an der Stelle acht, was eine Drehrichtung gegen den Uhrzeigersinn darstellt (siehe [mika]).

### 6.3 Programmierung der Anzeige

Für die Anzeige von Fahrzeugparametern wird ein 64x128 Pixel großes OLED-Display verwendet, dessen Ansteuerung über eine Inter Integrated-Circuit (I2C) Schnittstelle erfolgt. Mit dem Pin6 von Buchsenleiste J3 wird die I2C Adresse auf  $3C_{16}$  (GND) oder  $3E_{16}$  (3,3 V) festgelegt. Dem Schaltplan in Abbildung 71 ist zu entnehmen, dass die Adresse auf  $3C_{16}$  festgelegt ist. Außerdem wird das Reset Signal des Controllers an den Pin4 der Buchsenleiste J3 angelegt.

#### 6.3.1 Funktionsweise des organischen Leuchtdioden-Displays

Auf der Platine des Displays ist der Displaycontroller SSD1309 von SOLOMON SYSTECH verbaut. Dieser kommuniziert mit dem Mikrocontroller und steuert die Segment- und COM-Anschlüsse des Displays. Das Display besitzt 64 COM-Anschlüsse für die Kathoden der LEDs und 128 Segmentanschlüsse für die Anoden. Jeder COM-Anschluss entspricht einer Zeile des Displays und jeder Segmentanschluss einer Spalte (siehe Schaltplan in Abbildung 74). So mit kann durch Auswahl einer Reihe mit einem COM-Anschluss und durch ansteuern des Segmentanschlusses jedes Pixel angesprochen werden. Im Displaycontroller ist ein Random-Access Memory (RAM) mit 64 Zeilen und 128 Spalten verbaut. Jeder Zeile des Speichers ist eine Zeile des Displays (COM-Anschluss) und jeder Spalte des Speichers eine Spalte des Displays (Segmentanschluss) zugeordnet. Somit ist jedem Bit im RAM ein Pixel auf dem Display zugewiesen. Acht Zeilen des Speichers werden zu einer Seite zusammengefasst. Dadurch werden acht Zeilen mit einer 8bit großen Variable angesprochen. Bit0 entspricht der ersten Zeile und Bit7 der achten Zeile der Seite. Ein Bild auf dem Display wird aufgebaut indem zuerst das Segment SEG0 mit einem HIGH Pegel ausgewählt wird und die ersten acht im RAM gespeicherten Pixelwerte an den Anschlüssen COM0 bis COM7 angelegt werden. Danach wird das Segment SEG1 ausgewählt und die zweiten acht Pixelwerte an die gleichen COM-Anschlüsse angelegt. Dies wird bis zum letzten Segment fortgeführt. Im Anschluss daran wird wieder das Segment SEG0 ausgewählt. Diesmal werden die Pixelwerte jedoch an die Anschlüsse COM8 bis COM15 angelegt. Nachdem das letzte Segment und die letzten acht COM-Anschlüsse angesteuert worden sind, beginnt der Durchlauf von vorne. In den Einstellungen des Displaycontrollers kann eingestellt werden, von welcher Startseite bis zu welcher Stoppsseite die COM-Anschlüsse angesteuert werden. Dadurch können Seiten am oberen und unteren Displayrand ausgelassen und somit der Displaybereich vertikal beschränkt werden. Das Start- und Stoppsegment kann ebenfalls bestimmt und somit der rechte und linke Displayrand beschränkt werden. Außerdem kann die horizontale Orientierung festgelegt werden, indem man die Zuordnung des Segments SEG0 auf die Speicherspalte null oder 127 legt. Dadurch werden die ersten Pixelwerte im Speicher nicht in die Spalte des Segments SEG0 sondern in die Spalte von Segment SEG127 geschrieben. Die Einstellung der vertikalen Orientierung erfolgt mit der Richtung, mit der die Zeilen des Speichers auf die COM-Anschlüsse gelegt werden.

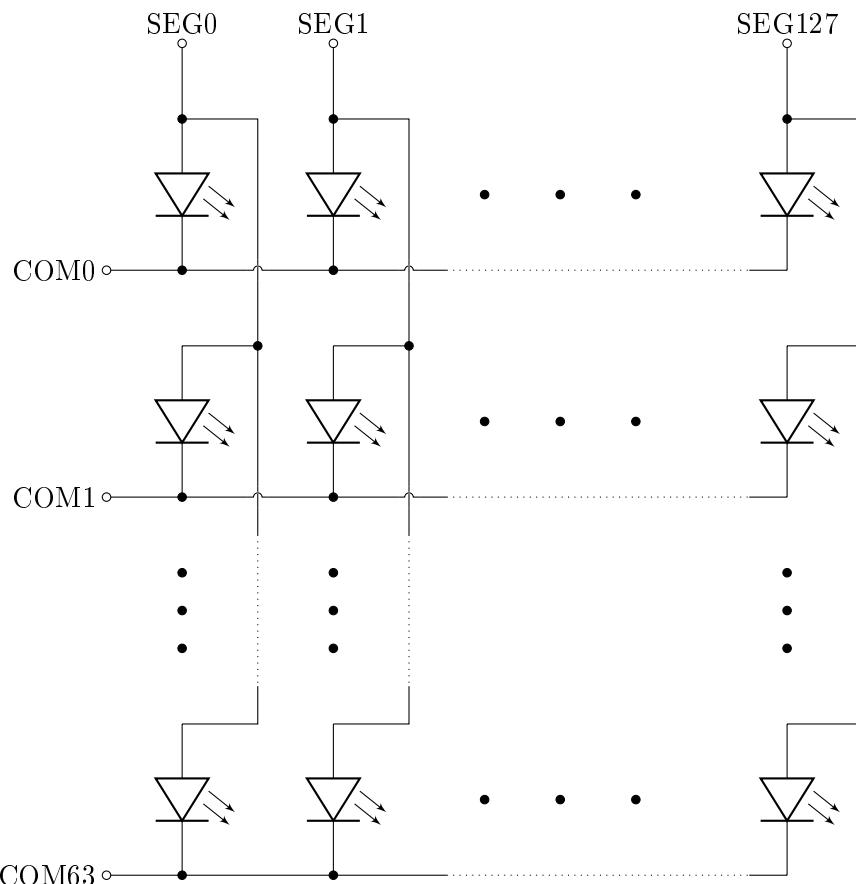


Abbildung 74: Repräsentativer Schaltplan des OLED-Displays mit Segment- und COM-Anschlüssen

### 6.3.2 I2C Ansteuerung

Die Kommunikation mit dem OLED-Display erfolgt mit einer I2C Schnittstelle. Dazu wird auf dem Mikrocontroller ein sogenanntes Flexcomm Modul verwendet. Dieses Modul beinhaltet mehrere Schnittstellen wie zum Beispiel Universal Asynchronous Receiver Transmitter (UART) oder I2C. Zu Beginn wird der Port-Pin P3.24 als SCL und der Port-Pin P3.25 als SDA des Flexcomm2 Moduls konfiguriert. Danach wird das Modul mit der Funktion „I2C\_MasterInit“ von NXP so initialisiert, dass die Taktfrequenz des SDA Signals 400 kHz beträgt. Dabei ergibt sich Problem, dass das Modul zunächst keine Daten sendet. Nach dem Debuggen und Überprüfen jedes Registers des Moduls wird festgestellt, dass der Takt für das Modul deaktiviert ist. Ärgerlich ist das, da bei der Konfiguration auf Funktionen zurückgegriffen wird, die von NXP entwickelt worden sind und funktionieren sollten. Nachdem der Takt (12 MHz) für das Flexcomm2 Modul aktiviert ist, werden die Daten wie gewünscht über die Schnittstelle gesendet. Jedoch beträgt die von der Initialisierungsfunktion eingestellte Taktfrequenz statt der angegebenen 400 kHz nur 333 kHz. Um den Fehler zu finden, wird in den Registern überprüft, welche Werte für die Erzeugung der Taktfrequenz verwendet werden. Zur Erzeugung wird die Taktfrequenz des Flexcomm2 Moduls mit einem Teiler erniedrigt.

Mit zwei Werten werden für die HIGH und LOW Zeiten die Anzahl der Takte (Flexcomm-Takt) angegeben. Nach der Initialisierungsfunktion wird ein Teiler von drei und eine HIGH und LOW Zeit von 5 Takten festgelegt. Die Taktfrequenz berechnet sich über Gleichung 5. Setzt man darin die Werte der Register ein, ergibt sich die gewollte Frequenz von 400 kHz.

$$\frac{f_{flexcomm}}{DIV_{CLK} \cdot (t_{cycHIGH} + t_{cycLOW})} \quad (5)$$

Im Datenblatt des Mikrocontrollers sind Werte angegeben, mit denen, bei einer Flexcomm-Taktfrequenz von 12 MHz, eine Taktfrequenz der I2C-Schnittstelle von 400 kHz erzeugt wird. Dabei wird ein Teiler von sechs, eine HIGH Zeit von zwei und eine LOW Zeit von drei Taktzyklen verwendet (vgl. [Sem19] Seite 462, Tabelle 496). Mit der Gleichung 5 berechnet sich die Taktfrequenz der Schnittstelle zu 400 kHz. Die drei richtigen Werte werden nach der Initialisierungsfunktion per Hand in den Registern gesetzt.

Um den Prozessor zu entlasten, wird auf die zu sendenden Daten per Direct Memory Access (DMA) zugegriffen. Dadurch werden diese parallel zum Prozessor in das Senderegister des Flexcomm2 Moduls geladen und der Prozessor hält sich nicht mit hin- und herschieben von Daten auf. Zuallererst wird das DMA Modul DMA0 mit der von NXP zur Verfügung gestellten Funktion „DMA\_Init“ initialisiert. Dem Datenblatt wird in Tabelle 300 entnommen, dass der Kanal 5 mit der I2C Funktion des Flexcomm2 Moduls verbunden ist. Deshalb wird mit der ebenfalls bereitgestellten Funktion „DMA\_EnableChannel“ der Kanal 5 freigegeben. Im Anschluss daran wird mit den beiden Funktionen „DMA\_CreateHandle“ und „I2C\_MasterTransferCreateHandleDMA“ die Funktionalität des DMAs mit dem Flexcomm2 Modul freigegeben.

Mit der Funktion „I2C\_MasterTransferNonBlocking“ wird eine Übertragung über I2C und DMA begonnen. Dieser Funktion wird in einer Variable ein Pointer auf die zu sendende Variable und deren Länge übergeben. Sollen anzuzeigende Daten an das OLED-Display übertragen werden wird der Funktion ein Pointer auf einen zuvor definierten Displaypuffer und dessen Länge übergeben. Soll eine Konfiguration gesendet werden, muss der Funktion ein Pointer auf eine 8bit Variable und die Länge eins übergeben werden. Die Übergabe in die Funktion erfolgt mit einer „struct“ Variable, die neben dem Pointer und der Länge noch die Slaveadresse des Displays, eine Variable für die Richtung der Übertragung (schreiben oder lesen) und eine Variable enthält, die angibt, wie die Start- und Stoppbits gesendet werden. Die Auswahl, ob es sich bei den gesendeten Daten um Konfigurations- oder Anzeigedaten handelt, erfolgt über die sogenannte Subadresse. Diese wird im Anschluss an die Slaveadresse übertragen. Deshalb muss der Funktion ebenfalls die Subadresse und deren Länge übergeben werden.

### 6.3.3 Konfiguration des Displays

Zu Beginn werden einige Resetparameter (siehe Tabelle 2) übertragen, die das Display zuerst ausschalten und den Kontrast einstellen. Danach wird ein Kommando gesendet, das den Speicher des OLED-Display Controllers so einstellt, dass dessen Inhalt direkt auf dem Display angezeigt wird. Eine andere Möglichkeit ist es, den Inhalt des RAMs zu ignorieren. Das Display wird im Standard-Modus (Normal Mode) gestartet. Dies bedeutet, dass eine 1 im Speicher die Pixel-LED einschaltet. Wird der horizontale Modus konfiguriert, wie es in dieser Anwendung der Fall ist, wird das Bild, das auf dem Display angezeigt werden soll, wie im Kapitel 6.3.1 beschrieben ist, aufgebaut. Der Befehl  $40_{16}$  bestimmt die Zeile des Displays, welche dem Anschluss COM0 zugewiesen wird. Durch Ändern dieses Wertes kann der Inhalt des Displays nach oben bzw. unten geschoben werden. Mit dem nächsten Kommando wird die horizontale Orientierung eingestellt. In diesem Fall wird der Inhalt des Speichers von Segment SEG127 nach SEG0 ausgegeben. Mit dem Befehl „scan from COM N-1 to COM0“ wird die vertikale Orientierung festgelegt, wobei N dem „MUX ratio“ (Anzahl der COM-Anschlüsse) entspricht. Die Hardware der COM Pins wird auf ihren Reset Wert eingestellt, was bedeutet, dass die geraden Zeilen mit den ersten 32 COM-Anschläüssen und die ungeraden mit den restlichen 32 COM-Anschläüssen verbunden sind. Danach werden die GPIO Pins als Eingänge konfiguriert. Das nächste Kommando stellt die Frequenz des Displays auf ihren korrekten Wert. Außerdem wird die Anzahl der Takte in Phase 1 und 2 auf jeweils zwei Takte gesetzt. In Phase 1 wird das Pixel ent- und in Phase 2 auf ihren neuen Wert aufgeladen. Diese Werte müssen den Kapazitätswerten der Pixel angepasst werden. Das letzte Kommando stellt die Spannung, bezogen auf die Versorgungsspannung des Displays, ein, wenn die COM-Anschlüsse deaktiviert sind. Alle Konfigurationseinstellungen sind im Datenblatt des Display Controllers zu finden (vgl [SSD11]).

Nach der Konfiguration des Resets erfolgt die Konfiguration, welche für die Anwendung auf dem Fahrzeug passend ist. Dazu wird die horizontale Orientierung so eingestellt, dass die Spaltenadresse 127 im Speicher dem Segment SEG127 entspricht. Ebenso wird die vertikale Orientierung auf einen Wert festgelegt, sodass das Display von den COM-Anschläüssen COM0 nach COM64 angesteuert wird. Am Ende wird das Display eingeschaltet und ist einsatzbereit.

	AE	Display off	A8	set MUX ratio to 64MUX
81	7F	Contrast Controll	C8	scan from COM N-1 to COM 0
A4		display on, output RAM content	D3 00	no vertical shift
A6		Normal display mode	DA 12	set COM pins hardware to reset
20	00	Horizontal mode	DC 02	GPIO pins are inputs
21	00	set column address start address	D5	increace FOSC frequency
7F		end address	F0	
22	00	set page address to reset value	D9 22	Phase 1 and 2 period to reset
07				
40		set display start line to top line	DB 34	set $V_{COMH}$ to $0.78 \cdot V_{cc}$
A1		column address 127 is SEG0		

Tabelle 2: Werte, die der Reihe nach über I2C zum OLED-Display gesendet werden

## 6.4 Darstellungen auf dem organisches Leuchtdioden-Display

Die im Folgenden vorgestellten Funktionen werden zur Darstellung von Formen, Text und Zahlen verwendet. Beim Aufruf einer dieser Funktionen wird eine Variable gesetzt, welche anzeigt, dass dem OLED-Display neue Daten gesendet werden müssen. In einem Echtzeittask wird diese Variable alle 30 ms geprüft und, wenn notwendig, die neuen Daten gesendet und dargestellt. Die Daten werden in einer globalen Variable, dem Displaypuffer „s\_disp\_0\_buffer“, gespeichert.

### 6.4.1 Pixelansteuerung

Mit der Funktion „ssd1309\_set\_pixel“ wird ein Pixel an einer vorgegebenen Position, also das anzusprechende Bit im Displaypuffer, gesetzt oder gelöscht.

### 6.4.2 Rechteck zeichnen

Zur Darstellung von Rechtecken wird die Funktion „ssd1309\_draw\_rect“ verwendet. Dieser werden die Koordinaten der linken oberen und rechten unteren Ecke und eine Variable

übergeben, die festlegt, ob das Rechteck ausgefüllt ist oder nur die Begrenzungslinie angezeigt werden soll. Außerdem muss angegeben werden, ob die Pixel des Rechtecks gesetzt oder gelöscht werden sollen. In der Funktion werden mit zwei Schleifen alle Pixel innerhalb des Rechtecks durchlaufen und für jedes Pixel bestimmt, ob dieses angezeigt werden soll oder nicht. Jedes Pixel wird mit der Funktion „ssd1309\_set\_pixel“ aus Kapitel 6.4.1 im Displaypuffer gesetzt oder gelöscht.

#### 6.4.3 Buchstabe darstellen

Der Funktion „ssd1309\_write\_char“ wird eine char Variable übergeben, welche am Display angezeigt werden soll. In „struct“-Variablen mit dem Namen „ssd1309\_font\_t“ sind für alle darstellbaren Buchstaben und Zeichen Werte hinterlegt, die diese in unterschiedlichen Größen darstellen. In der Funktion selbst wird zu Beginn überprüft, ob das übergebene Zeichen ein darstellbares Ascii-Zeichen ist. Ist dies der Fall, muss abgefragt werden, ob das Zeichen innerhalb des Displays liegt. Wenn ja, wird mit zwei Schleifen für jedes Pixel innerhalb der Höhe und Breite des Zeichens überprüft, ob dieses angezeigt werden soll oder nicht. Pixel, die angezeigt werden, werden mit der Funktion „ssd1309\_set\_pixel“ aus Kapitel 6.4.1 angesprochen. Die Position des Zeichens wird über einen Cursor eingestellt, der vor dem Funktionsaufruf gesetzt werden muss. Nachdem die einzelnen Pixel des Zeichens im Displaypuffer gesetzt bzw. gelöscht worden sind, wird der Cursor automatisch inkrementiert.

#### 6.4.4 Text darstellen

Die Funktion „ssd1309\_write\_str“ stellt am Display einen Text dar, der über einen Pointer auf ein Zeichenarray übergeben wird. Dazu wird die in Kapitel 6.4.3 beschriebene Funktion für jedes im Text enthaltene Zeichen ausgeführt. Die Position des Textes wird, wie bei den Buchstaben, über den Cursor eingestellt, welcher vor dem Funktionsaufruf gesetzt werden muss.

#### 6.4.5 Bilder darstellen

Die Funktion „ssd1309\_draw\_img“ stellt Bilder dar, die als Hexadezimalwert abgespeichert sind. Die Position der linken oberen Ecke muss der Funktion übergeben werden. Das Bild wird in einer „struct“-Variable mit dem Namen „ssd1309\_img\_t“ übergeben, die die Breite und Höhe des Bildes enthält. Ausgegeben wird das Bild mithilfe zweier Schleifen, die für jedes Pixel des Bildes überprüfen, ob dieses angezeigt werden soll oder nicht. Dargestellt werden die einzelnen Pixel mit der Funktion „ssd1309\_set\_pixel“ aus Kapitel 6.4.1.

### 6.5 Menüführung

Auf die einzelnen Parameter und Funktionen des Fahrzeugs wird mit verschiedenen Menüs zugegriffen. Zuallererst wird das Hauptmenü aufgerufen. Darin sind die einzelnen Funktionen

in unterschiedliche Menüpunkte aufgeteilt. Der Mikrocontroller nimmt die Menüführung auf Basis der Werte des Drehencoders und der beiden Taster vor. Ein Zustandsautomat entscheidet je nach Bedienung, ob der Hauptbildschirm oder ein Menü angezeigt werden soll. Bevor der Zustandsautomat ausgeführt wird, werden die beiden Zustände der entprellten Taster sowie die Anzahl der Schritte gespeichert, die der Drehencoder in eine bestimmte Richtung getätigigt hat. Der Zustandsautomat bestimmt das aktuelle, anzuzeigende Menü und ändert den Displaypuffer so ab, dass dieses dargestellt wird.

**Es gibt insgesamt vier verschiedenen Arten von Menüs:**

- MENU\_LINK
- MENU\_CHECK
- MENU\_PAGE
- MENU\_VALUE

Jedes Menü besitzt eine „struct“-Variable, in der immer die drei gleichen Eigenschaften enthalten sind; die Bezeichnung, die auf dem OLED-Display angezeigt wird, die Menüart und eine Kennzeichnung, ob das Menü angeklickt werden kann oder nicht. Ist das angeklickte Menü ein MENU\_LINK, so öffnet sich das ebenfalls in der „struct“-Variable enthaltene Folge-Menü. Ist es von der Art MENU\_CHECK, wird mit einem Klick auf dieses eine, in der „struct“-Variable als Pointer gespeicherte, Variable inkrementiert. Durch eine spätere, binäre UND-Operation mit 1, erhält man eine Variable, die bei einem Klick aktiviert bzw. deaktiviert wird ( $01_2 \& 1 = 1$ ;  $10_2 \& 1 = 0$ ). Mit der Betätigung eines Menüs der Art MENU\_PAGE, wird eine darin verlinkte Funktion ausgeführt. Wird ein Menü der Sorte MENU\_VALUE betätigkt, wird der Wert einer in der „struct“-Variable als Pointer verknüpften Variable mit dem Drehencoder erhöht oder erniedrigt. Außerdem wird eine Funktion aufgerufen, wenn sich der Wert dieser Variable verändert. Dies wird zum Beispiel bei der Einstellung der maximalen und minimalen Drehzahl der BLDC-Motoren verwendet, um den Wert in das Register des jeweiligen Timer Match Registers zu schreiben. Somit kann z.B. die Einstellung der Drehzahlwerte überprüft werden.

## 6.6 Aufbau des Human-Machine Interfaces

Das Human-Machine Interface (HMI) wird in einem eigenen Task aufgebaut. Beim ersten Durchlauf wird zuallererst das OLED-Display sowie die benötigten Menüs initialisiert. Danach werden im Abstand von einer Sekunde zuerst das Logo von NXP und danach das Logo der HAW Landshut angezeigt. Nach wiederum einer Sekunde wird der Hauptbildschirm geöffnet. Dort kann mit dem Taster in das Hauptmenü gewechselt und zwischen dem Drive-, Hardware-, Expert-, About- und Hide Menü gewählt werden. Aus zeitlichen Gründen sind nur das Hardware und Hide Menü implementiert. Das Hide Menü wechselt lediglich vom Hauptmenü zurück auf den Hauptbildschirm.

Das Hardware Menü enthält Untermenüs für die Kamera, die Motoren und den Summer. Die Menüs für die Kamera und den Summer sind noch nicht implementiert. Mit einem Klick auf Motoren wird in das Motorenmenü gewechselt, in dem die beiden BLDC-Motoren sowie der Servo-Motor enthalten sind. Im Servo Menü wird in der ersten Zeile der Port und Pin angezeigt, an welchem das PWM-Signal für die Servo-Lenkung ausgegeben wird. In der zweiten Zeile befindet sich ein Menü zum Testen der Lenkwinkel. Dabei wird ein Wert zwischen -100 und 100 mit einer Schrittweite von 1 vom Drehencoder eingestellt. Der Servo lenkt beim Wert -100 auf den vollen Linkseinschlag und beim Wert 100 auf den vollen Rechtseinschlag ein. Im nächsten Menüeintrag wird mit dem Drehencoder der Initialisierungswert, also die Mittelstellung, eingestellt. Dieser Wert wird nach dem Einschalten des Mikrocontrollers angesteuert. Mit den letzten beiden Einträgen wird der linke bzw. rechte, maximale Lenkeinschlag bestimmt.

Wählt man im Motor Menü einen der beiden Antriebe aus, gelangt man in das Menü für den jeweiligen BLDC-Motor. Der erste Menüeintrag zeigt, wie beim Servo Menü, lediglich den Port-Pin des ausgewählten Motors an. Ebenso wie beim Servo ist bei den BLDC-Motor Menüs ein Testmenü enthalten, mit welchem man eine Drehzahl zwischen 0 und 100 vorgeben kann. Die letzten beiden Untermenüs stellen die obere und untere Drehzahl ein.

Jedes Menü enthält als letzten Eintrag ein Untermenü, mit welchem ins jeweils vorangegangene Menü gewechselt werden kann. Ein Klick auf den Taster lässt das OLED-Display zum Hauptbildschirm zurückspringen.

## 6.7 Anleitung zum Hinzufügen eines neuen Menüs

1. Hinzufügen eines Eintrags in dem Menü, in dem das neue Menü aufgerufen werden soll („menu\_data.c“)
2. Erhöhen der Anzahl der Einträge um eins („menu\_data.c“)
3. Erstellen einer Funktion, welche das neue Menü aufruft („screen.c“)
4. Erstellen einer Liste mit allen im neuen Menü enthaltenen Untermenüs sowie eines neuen Handlers („menu\_data.c“)
5. Deklarieren der notwendige Variablen und Funktionen für die neuen Untermenüs („menu\_data.c“)
6. Initialisieren des neuen Menüs („hmi.c“)

Soll zum Beispiel im Hardware Menü ein neues Untermenü mit dem Namen „new menu“ hinzugefügt werden, so muss dafür ein neuer Eintrag (Abbildung 75, roter Rahmen) im Hardwaremenü erstellt werden. Außerdem muss die Variable „entry\_count“ im zugehörigen „menu\_rtos\_handle\_t“ inkrementiert werden (siehe Abbildung 76). Im Anschluss daran muss

eine Funktion implementiert werden, welche das neue Menü öffnet. Diese ersetzt nur das aktuelle Menü durch das neue Menü (siehe Abbildung 77).

```
const static menu_entry_t menu_main.hardware_entries[] = {
{
    /*****
     * Entry for new menu
     *****/
    .str = "new menu", //display "new menu"
    .type = MENU_LINK, //type is MENU_LINK
    .link = menu_open_new_menu, //link to new menu
    .en = true, //element is clickable
},
{
    .str = "Camera", //display "Camera"
}
```

Abbildung 75: Schritt 1: Erstellen eines neuen Menüeintrags durch Hinzufügen der im roten Rahmen dargestellten Code-Zeilen

```
menu_rtos_handle_t menu_main.hardware_handle = {
    .drv_handle = {
        .entry_cnt = 5,
        .entry_list = (menu_entry_t*)menu_main.hardware_entries,
        .draw = menu_list_draw,
    },
};
```

Abbildung 76: Schritt 2: Erhöhen der Anzahl der Menüeinträge um eins (entry\_cnt)

```
void menu_open_new_menu() {
    menu_rtos_switch_handle(&curr_menu_handle, &menu_new_handle);
    menu_reset(&curr_menu_handle->drv_handle);
}
```

Abbildung 77: Schritt 3: Erstellen einer Funktion zum Menüaufruf

Außerdem müssen in der Datei „menu\_data.c“ die Einträge des neuen Menüs definiert werden (Schritt 4). Dazu wird zuallererst eine Liste für die Einträge erstellt. Das erste Untermenü mit dem Namen „new1“ öffnet in diesem Beispiel den Hauptbildschirm. Das zweite Menü „new2“ aktiviert bzw. deaktiviert die Variable „variableCheck“. Mit dem Menü „new3“ wird die Variable „variableValue“ mit dem Drehencoder verändert und die Funktion „func\_value“ ausgeführt. Das letzte Menü mit dem Namen „new4“ führt die Funktion „func\_page“ aus, welche den kompletten Bildschirm füllt (alle Pixel-LEDs ein). Außerdem muss in diesem Schritt der Handler definiert werden, der die eben erstellte Liste mit den Einträgen, deren Anzahl sowie die bereits vorhandene Funktion zum Darstellen der Untermenüs enthält. Wichtig ist, dass der Handler in der Datei „menu\_data.h“ bekannt ist. Nur so kann in der Datei „screen.c“ das neue Menü geöffnet werden (siehe Abbildung 79).

```

const static menu_entry_t menu_new_entries[] = {
{
    .str = "new1",                                //display "new1"
    .type = MENU_LINK,                            //type is MENU_LINK
    .link = menu_close,                          //pointer to main screen
    .en = true,                                  //element is clickable
},
{
    .str = "new2",                                //display "new2"
    .type = MENU_CHECK,                           //type is MENU_LINK
    .val = &variableCheck,                      //pointer to variable
    .en = true,                                  //element is clickable
},
{
    .str = "new3",                                //display "new3"
    .type = MENU_VALUE,                           //type is MENU_VALUE
    .val = &variableValue,                      //pointer to variable
    .min_val=0,                                 //min value for variableValue
    .max_val=10,                                //max value for variable Value
    .func = func_value,                          //call func_value on each change
    .en = true,                                  //element is clickable
},
{
    .str = "new4",                                //display "new4"
    .type = MENU_PAGE,                           //type is MENU_PAGE
    .func = func_page,                          //pointer to function
    .en = true,                                  //element is clickable
},
};

menu_rtos_handle_t menu_new_handle = {
    .drv_handle = {
        .entry_cnt = 4,                         //5 entries
        .entry_list = (menu_entry_t*)menu_new_entries, //list of all entries
        .draw = menu_list_draw,                  //entry draw function
    }
};

```

Abbildung 78: Schritt 4.1: Liste mit Einträgen des neuen Menüs erstellen

```

/*****************
 * handle for new
 *****/
extern menu_rtos_handle_t menu_new_handle;

```

Abbildung 79: Schritt 4.2: Hinzufügen des Menü Handlers im Header („menu\_dat.h“)

Die Funktion „func\_page“ (siehe Abbildung 80) wird in der Datei „screen.c“ angegeben (Schritt 5). Diese nimmt den Semaphor des Display, setzt jedes Bit im Displaypuffer, bis auf die Kopfzeile der Anzeige, und gibt den Semaphor wieder frei.

```

void func_page(uint8_t refresh){

    ssd1309_rtos_lock(&g_disp_0);
    ssd1309_draw_rect(&g_disp_0.disp_obj,0,13,127,63,true,ON);
    ssd1309_rtos_unlock(&g_disp_0);

}

```

Abbildung 80: Schritt 5: Deklarieren der notwendige Variablen und Funktionen für die neuen Untermenüs

Nach der Initialisierung des neuen Menüs (Schritt 6) kann dieses aufgerufen werden. Das neu erzeugte Menü ist in Abbildung 81 und der mit der Funktion „func\_page“ voll gefüllte Bildschirm in Abbildung 82 dargestellt.

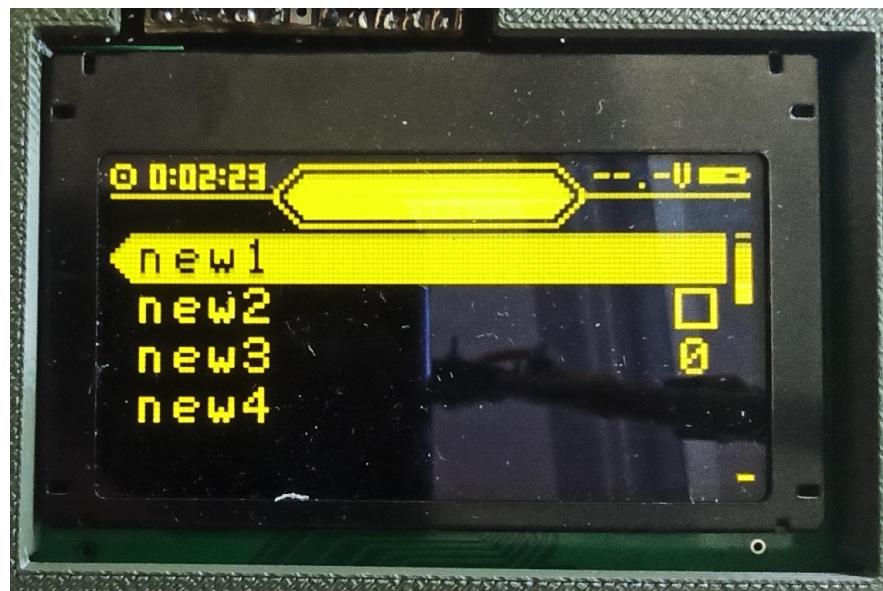


Abbildung 81: Anzeige des neu erstellten Menüs auf dem OLED-Display



Abbildung 82: Mit der Funktion „func\_Page“ erzeugter, voll gefüllter Bildschirm

## 6.8 Verwendung des Electrically Erasable Programmable Read-Only Memorys

Die Fahrzeugparameter, die mit dem OLED-Display eingestellt werden können, sollen auch nach der Trennung von der Versorgungsspannung erhalten bleiben. Deshalb werden diese in das Controller-eigene EEPROM gespeichert.

Das EEPROM wird gleich nach dem Start des Mikrocontrollers mit der Funktion „EEPROM\_Init“ initialisiert. Danach werden die im EEPROM gespeicherten Werte mit der „memcpy“-Funktion in ein Array mit allen Parametern kopiert. Diese Funktion kopiert einen ganzen Speicherbereich mit einer vorgegebenen Länge ab einer Startadresse an eine ande-

re Adresse. Sollen Daten ins EEPROM gespeichert werden, wird zuerst das Array mit den Parametern mit der „memcpy“-Funktion in den Speicherbereich des EEPROMs geschrieben. Danach müssen diese mit der Funktion „EEPROM\_WritePage“ fest in den Speicher geschoben werden. Die Daten werden automatisch gespeichert, wenn von einem Menü in den Hauptbildschirm gewechselt wird. Im Hardwaremenü gibt es einen Eintrag „Restore def.“, mit dem die Parameter auf deren Ursprungswerte zurückgesetzt werden können.

## 6.9 Ausblick zur Programmierung des Displays

Eine sinnvolle Funktionsweise, die mit dem OLED-Display relativ einfach realisierbar ist, ist die Anzeige der Daten, die die Kamera liefert (näheres dazu in Kapitel 7.3 „Programmierung der zyklischen Bildaufnahme“). Dazu kann die Menüart „MENU\_PAGE“ verwendet werden. Die Anzahl der Kamerapixel stimmt zufälligerweise mit der Anzahl der Pixel auf dem Display überein. In der Funktion des „MENU\_PAGE“-Menüs, kann für jeden Kamerapixel ein Displaypixel mit unterschiedlicher Höhe verwendet werden. Die Höhe kann dabei dem Wert entsprechen, den die Kamera für diesen Pixel liefert.

## 7 Streckenerkennung des Fahrzeugs

Für die Streckenerkennung des Fahrzeugs stehen zwei Kamera-Typen zur Verfügung. Zum einen kann die Pixy2 Kamera verwendet werden, welche eine Auflösung von 640x400 Bildpunkten besitzt. Die andere Möglichkeit ist eine einfache Zeilenkamera mit einer Auflösung von 1x128 Bildpunkten.

Da in vorherigen Projekten meist eine Zeilenkamera verwendet worden ist, wird für dieses Projekt ebenfalls eine Zeilenkamera eingesetzt. Ein Grund dafür ist, dass die Verwendung von Bildern von so großer Auflösung, wie bei der Pixy2 Kamera, viel Rechenressourcen und Zeit kostet. Eine geringere Auflösung ist deshalb für die Schnelligkeit des Fahrzeugs von großer Bedeutung.

Da mehr als eine Kamera am Fahrzeug erlaubt ist, kann bei der Weiterführung des Projekts natürlich die Funktion einer weiteren Kamera implementiert werden. Sinnvoll kann das vor allem für die vorausschauende Streckenverlaufs-Erkennung sein, damit auf Änderungen im Verlauf frühzeitig reagiert werden kann. Auch bei Kreuzungen, an denen die Seitenlinien kurz verschwinden, kann eine zweite Kamera dabei helfen, den Streckenverlauf nicht zu verlieren.

### 7.1 Kamera des Fahrzeugs

Wie in der Einführung dieses Kapitels bereits erwähnt, wird eine Zeilenkamera mit einer Auflösung von 1x128 Bildpunkten verwendet (TAOS TSL1401R-LF). Die Kamera kann mit einer maximalen Taktfrequenz von 8MHz betrieben werden.

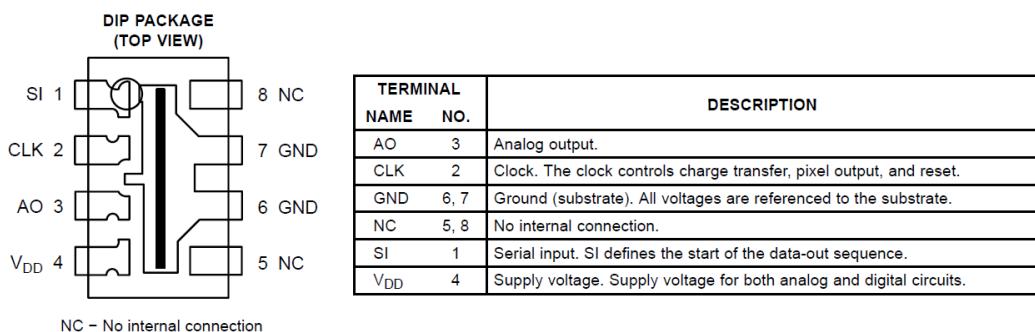


Abbildung 83: Pinbelegung der Taos Zeilenkamera TSL1401R-LF [Tao06]

In Abbildung 83 ist die Anschlussbelegung des Kamerachips und in Abbildung 84 der zeitliche Verlauf eines Bildtransfers aus Sicht der Kamera abgebildet. Je höher die Taktfrequenz der Kamera-Takt (CLK), desto schneller kann eine einzelne Aufnahme getätigt werden (Bildaufnahmegeschwindigkeit). Der serielle Eingang der Kamera (SI) startet die Aufnahme. Damit die Kamera das Startsignal erkennt, muss dieses mindestens 20ns vor der steigenden Flanke der Kamera-Takt anliegen ( $t_{su(SI)}$ , „Setup time, serial input“). Da die „Hold time“ des

seriellen Eingangs ( $t_{h(SI)}$ ) im Datenblatt 0ns beträgt, kann sich das Rücksetzen des SI-Signals gleichzeitig mit der Taktflanke ereignen. Je Taktzyklus wird ein einzelnes Pixel ausgelesen. Für die ADC-Messung ist unbedingt die „Settling Time“ zu beachten ( $t_{ts}$ ), vor deren Ablauf der ADC nicht getriggert werden darf. Die Sampling-Zeit muss im Gegensatz zur Konversion vor Beginn des nächsten Taktzyklus zu Ende sein. Die Konversion hingegen hat Zeit bis zum nächsten ADC Trigger. Die Taktfrequenz wird also von der Dauer einer ADC-Konversion beschränkt und nicht durch die technische Grenze von 8MHz.

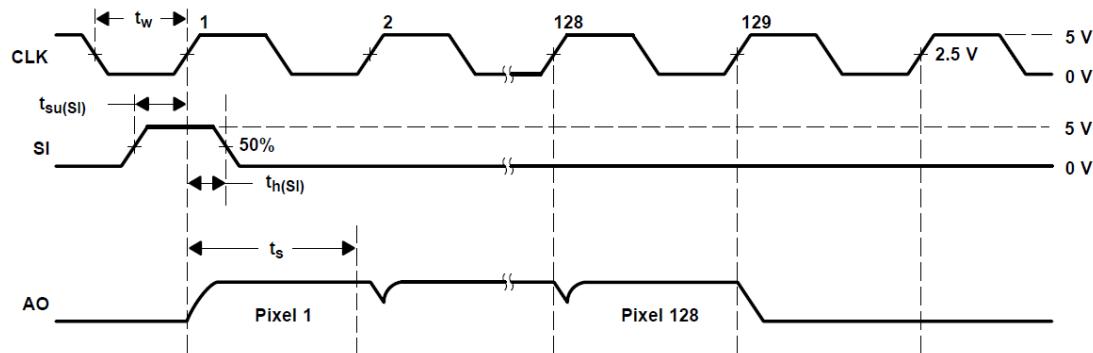


Abbildung 84: Zeitverlauf einzelner Pixeltransfers aus Sicht der Kamera TSL1401R-LF relativ zur vom Controller bereitgestellten Kamera-Takt [Tao06]

Nach dem 128. Taktzyklus ist die Aufnahme eines Bildes eigentlich abgeschlossen. Trotzdem darf die nächste Bildaufnahme erst nach 129 Taktzyklen und einer zusätzlichen „Pixel Charge Transfer Time“ ( $t_{qt}$ ) von 20 $\mu$ s erfolgen (siehe Abbildung 85).

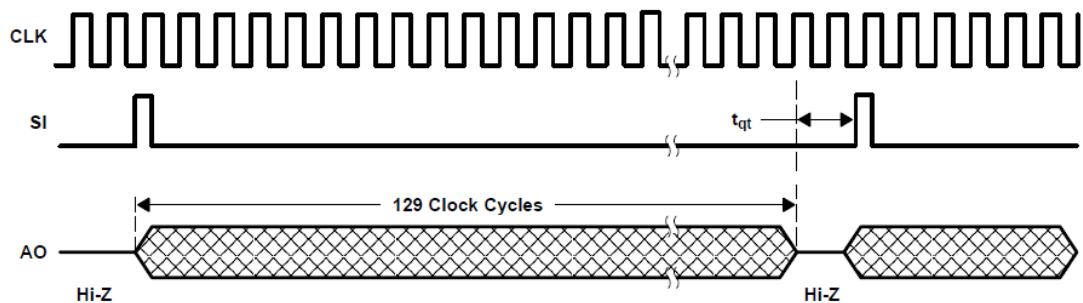


Abbildung 85: Zeitverlauf eines vollständigen Bildtransfers aus Sicht der Kamera TSL1401R-LF mit der Mindestdauer von 129 Taktzyklen und einer Wartezeit von  $t_{qt}$  („Pixel Charge Transfer Time“ = min. 20 $\mu$ s) [Tao06]

## 7.2 Ablauf einer Bildaufnahme

Für das Erstellen eines zeitlichen Ablaufs für die Bildaufnahme des Fahrzeugs werden drei Module des Mikrocontrollers verwendet. Für die Nachbildung der zeitlichen Abläufe, die die Kamera benötigt (SI, CLK), wird der SCTimer des Controllers eingesetzt. Für diesen kann eine Vielzahl von Events konfiguriert werden. Jedes dieser Events kann andere Controller-Module triggern oder Ausgänge des Controllers setzen oder rücksetzen. Das zweite Modul, welches verwendet wird, ist der Analog-Digital-Wandler (ADC). Mithilfe dessen wird zyklisch der Spannungspegel des analogen Kameraausgangs (CAM\_AO) gemessen. Diese Spannungen repräsentieren die Helligkeiten der einzelnen Pixel. Das dritte eingesetzte Modul ist ein einfacher Timer, der die Bildaufnahmerate festlegt. Die Bildaufnahmerate ist die Frequenz, mit der neue Bilder aufgenommen werden und ist nicht zu verwechseln mit der Bildaufnahmefrequenz eines einzelnen Bildes!

Abbildung 86 zeigt den zeitlichen Verlauf einer Bildaufnahme aus Sicht des Controllers mit den in dieser Fahrzeugversion festgelegten Zeiten und Events. Event 0 und 1 sind für die Erstellung des Kamera-Taktsignals zuständig (Event 0: Set CAM\_CLK / Event 1: Clear CAM\_CLK). Die Events 2 (Set CAM\_SI) und 3(Clear CAM\_SI) legen die Relativposition des SI-Signals zum Kamera-Takt fest. Das Rücksetzen des SI-Signals erfolgt in jedem Takt-Zyklus, während das Setzen nur dann erlaubt wird, wenn der Timer, der für die Bildaufnahmerate zuständig ist, einen Interrupt auslöst. Das Event 4 triggert den ADC, welcher spätestens bis zum nächsten Trigger-Event ein fertiges Ergebnis liefern muss. Der ADC misst die Spannung an CAM\_AO in jedem Taktzyklus. Die Speicherung der Werte erfolgt allerdings nur dann, wenn gerade eine Aufnahme getätigt wird, also insgesamt 128 mal nach dem Erscheinen des SI-Signals.

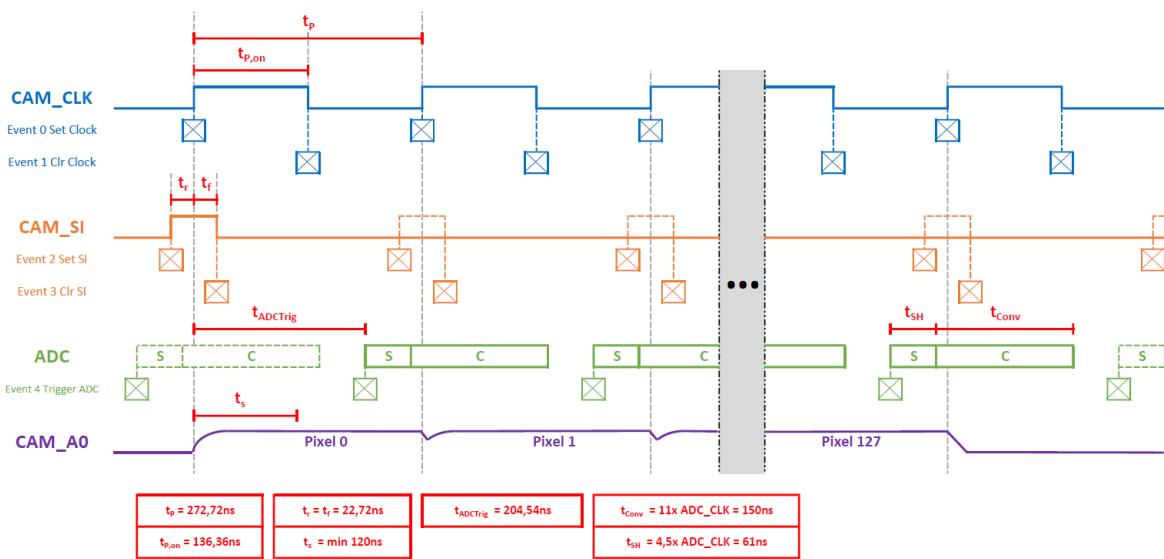


Abbildung 86: Zeitlicher Ablauf einer einzelnen Bildaufnahme aus Sicht des Controllers

### 7.3 Programmierung der zyklischen Bildaufnahme

Der gesamte Code für die Inbetriebnahme der Kamera ist wie bei der Servo-Lenkung und bei den Antrieben in zwei Dateien unterteilt. In der Datei „camera.h“ befinden sich die Funktionsprototypen und die Einbindung der für die Kamera notwendigen Bibliotheken (Prototypen siehe Abbildung 87). Die Datei „camera.c“ enthält hingegen die gesamten Funktionen, die für die Realisierung der zyklischen Bildaufnahme notwendig sind und im Folgenden näher erläutert werden.

```
*****
* Prototypes
*****
void CAM_Init(void);
void SCTimer_Clock_Config(void);
void SCTimer_CamCLK_Init(void);
void SCTimer_SIEvents_Init(void);
void SCTimer_ADCTrigger_Init(void);
void ADC_Config(void);
void ADC_Calibration(void);
void CTIMER0_Init(void);
```

Abbildung 87: Prototypen der Kamerafunktionen aus „camera.c“ in der Datei „camera.h“

Die Primärfunktion der Kamera-Initialisierung („CAM\_Init“, siehe Abbildung 88) ruft aus Übersichtlichkeitsgründen lediglich alle anderen notwendigen Initialisierungs- und Konfigurationsfunktionen auf und startet an ihrem Ende den SCTimer SCT0, welcher die CLK-Events, SI-Events und das ADC-Trigger-Event generiert.

```
*****
* Camera Main Initialization function
*****
void CAM_Init(void)
{
    //*****
    //Camera Initialization Functions
    CTIMER0_Init();           //CTIMER0 Initialization
    SCTimer_Clock_Config();   //SCTimer Clock Configuration
    SCTimer_CamCLK_Init();    //Initialize PWM Signal for Camera Clock (3,63MHz)
    SCTimer_SIEvents_Init();  //Initialize Start Signal for Camera (SI)
    SCTimer_ADCTrigger_Init(); //Initialize ADC Trigger Event
    ADC_Config();             //Configure ADC
    //*****

    SCT0->CTRL &= ~(1 << 2); //Unhalt SCT0 by clearing bit 2 of CTRL
}
```

Abbildung 88: Funktion „CAM\_Init“ als primäre Einstiegsfunktion zur Kamera-Initialisierung; Teil der Datei „camera.c“

Die erste Modul-Initialisierung, die in der Funktion „CAM\_Init“ von stattfinden geht, ist die des CTIMER0 („CTIMER0\_Init“, siehe Abbildung 89). Das Timer-Modul CTIMER0 ist, wie bereits erwähnt, für die Bildaufnahmerate verantwortlich. Zuerst wird festgelegt, was bei

einem Timer-Überlauf passieren soll. Bei einem Überlauf, also dann, wenn der Wert des Timer-Zählers gleich dem Wert aus dem Match Register 0 ist, wird sowohl ein Interrupt Request gestartet, der Zähler des Timers auf 0 gesetzt und ein neuer Wert für das Match Register 0 aus dem Shadow Register geladen. Außerdem wird in der Initialisierungsfunktion des Timers die Interrupt Service Routine für das Timer Modul CTIMER0 aktiviert und der Timer gestartet.

```
/*****************************************************************************  
 * CTIMER0 Initialization function  
*****/  
void CTIMER0_Init(void)  
{  
    /* CTIMER0 peripheral initialization */  
    CTIMER_Init(CTIMER0, &TakeShots_config);  
  
    CTIMER0->MCR = 0;                                //Delete current Configuration  
    CTIMER0->MCR |= (1<<0)|(1<<1)|(1<<24); //Interrupt, Timer Counter reset and reload MR with MSR at Match0  
  
    CTIMER0->MSR[0] = 220000;                         //Initialize MSR0 with 220000 --> Timer overflow every 1ms  
    //*****  
    //ADC Interrupt configuration  
    //CTIMER0->INTEN |= (1<<0);  
    NVIC_SetPriority(CTIMER0 IRQn, 0);      //Enable NVIC interrupt for sequence A.  
    EnableIRQ(CTIMER0 IRQn);           //Enable ADC Sequence A Interrupt  
    //Enabling NVIC will block DMA trigger!!!!  
    //*****  
  
    CTIMER_StartTimer(CTIMER0); //Start CTIMER0  
}
```

Abbildung 89: Funktion „CTIMER0\_Init“ für die Initialisierung des Timer Moduls CTIMER0, welches für die Bildaufnahmerate zuständig ist; Teil der Datei „camera.c“

Bei einer Bildaufnahme in regelmäßigen Abständen gibt es einen problematischen Effekt, der die Pixelwerte betrifft. Da die Strecke nie gleichmäßig beleuchtet ist, schwankt die durchschnittliche Helligkeit des Bildes erheblich. Wenn zu lange belichtet wird, kann es sogar zu einer Überhöhung der Spannung einiger Pixel kommen, was zu falschen Ergebnissen bei allen Pixeln führt. In beiden Fällen ist die Dauer der Belichtung ungünstig gewählt. Um diesem Problem entgegenzuwirken, muss die Belichtungszeit am besten nach jeder Aufnahme angepasst werden. Belichtet wird bei der Kamera immer dann, wenn nicht gerade eine Aufnahme getätigkt wird. Um die Belichtungszeit anzupassen, muss also lediglich der Überlaufwert des Timers im Shadow Register verändert werden (Anpassung der Regelmäßigkeit des Kamera-startenden SI-Events). Da aus Zeitgründen die Anpassung der Belichtungszeit nicht mehr implementiert werden konnte, ist im Folgenden der theoretisch erarbeitete Ablauf der Belichtungszeitanpassung aufgeführt.

### Ablauf der Belichtungszeitanpassung nach jeder Aufnahme:

- Abzählen der hellen und dunklen Pixel  
Schwelle hell/dunkel bei 8bit ADC-Auflösung ist  $256/2 = 128$
- Gewichten der Pixelwerte  
Gewichtung so, dass gleich viele Helligkeitswerte von dunklen und hellen Pixeln in die Mittelwertbestimmung einfließen
- Mittlere Helligkeit des Bildes berechnen  
Addition aller Werte und Bildung des Gradienten mit der Werteanzahl (Werteanzahl variiert je nach Gewichtung)
- Erörterung, ob bei der nächsten Aufnahme länger oder kürzer belichtet werden muss  
Wenn errechnete mittlere Helligkeit höher als 128  $\rightarrow$  Belichtungszeit verkürzen  
Wenn errechnete mittlere Helligkeit niedriger als 128  $\rightarrow$  Belichtungszeit verlängern
- Anpassung des Überlauf-Werts des Timers CTIMER0 im Shadow Register von Match 0

Im Anschluss an die Initialisierung des Timers CTIMER0 wird in „CAM\_Init“ die Funktion „SCTimer\_Clock\_Config“ aufgerufen (siehe Abbildung 90). Darin wird der Takt für den SCTimer SCT0 mit einer Frequenz von 44MHz festgelegt. Die maximale Taktfrequenz für den SCTimer beträgt zwar 100MHz, jedoch reicht die Zeitauflösung bei 44MHz für die Genauigkeit der Zeiteinstellung der einzelnen Events aus.

```
/****************************************************************************
 * SCTimer Clock Configuration
 ****/
void SCTimer_Clock_Config(void)
{
    //*****
    //Configure CLK for SCTimer/PWM
    SYSCON->SCTCLKSEL = 0x00;           //Main Clock for SCTimer/PWM
    SYSCON->SCTCLKDIV = (5-1);          //CLK Divider 5 -> 220MHz/5 = 44MHz (SCMax = 100MHz)
    SYSCON->AHBCLKCTRL[1] |= 1<<2;    //SCTimer/PWM SCT0 CLK Enable
    SYSCON->PRESETCTRLCLR[1] |= 1<<2; //Clear the SCTimer/PWM peripheral reset
    //*****

    //*****
    //SCT0 --> Use Configured CLK
    SCT0->CONFIG |= 0x3<<1;           //CLKMODE asynchronous with input (->CKSEL)
    SCT0->CONFIG |= 0xE<<3;           //CKSEL Input 7 Rising Edges
    SCT0->CONFIG |= 1<<17;            //Auto limit (& two 16-bit timers)
    //*****
}
```

Abbildung 90: Funktion „SCTimer\_Clock\_Config“ für die Konfiguration des Modultakts des SCTimers; Teil der Datei „camera.c“

Als nächstes wird mit der Funktion „SCTimer\_CamCLK\_Init“ das Kamera-Taktsignal erstellt (siehe Abbildung 91), also die Events 0 und 1 konfiguriert. Der Kamera-Takt wird mithilfe des SCTimer-Ausgangs SCT0\_OUT1 am Pin 13 der Buchsenleiste J13 nach außen geführt (Port-Pin P3.27). Das Event 0 setzt den Ausgang, während das Event 1 den Ausgang zurücksetzt. Um mit dem ADC eine 8 bit Auflösung zu erreichen, kann der Kamera-Takt nicht mit der technischen Grenze von 8MHz betrieben werden. Deshalb wird die Frequenz auf 3,63MHz festgesetzt. Das entspricht einer Periodendauer von 272,72ns (Event 0) und einer On-Zeit von 136,36ns (Event 1). Da beide Events immer aktiv sind, läuft der Kamera-Takt die ganze Zeit durch.

```
/****************************************************************************
 * Initialize PWM Signal for Camera Clock (3,676MHz)
 ****/
void SCTimer_CamCLK_Init(void)
{
    //*****
    //Configure Pin P[3][27] (J13 Pin13) (CAM_CLK/SCT0_OUT1)
    IOCON->PIO[3][27] &= 0xFFFFFFFF0;      //Clear FUNC bits of P3.27
    IOCON->PIO[3][27] |= 0x02;            //Set FUNC bits to SCT0_OUT1 function FUNC2 P3.27
    GPIO->DIR[3] |= 1<<27;             //Set PIO3_27 (SCT0_OUT1) to output
    //*****

    //*****
    //Set PWM at PIO3_27 to 3,676MHz (Cam_CLK_frequency max=8MHz)
    //*****
    //Event 0 for Counter Limit
    SCT0->MATCHREL[0] = (12-1);           //Match 0 @ 12/44MHz = 272,72ns Limit Counter
    SCT0->EV[0].STATE = 0xFFFFFFFF;       //Event 0 happens in all states
    SCT0->EV[0].CTRL = (1 << 12);        //Match 0 condition only
    SCT0->OUT[1].SET = (1 << 0);         //Event 0 will set SCT0_OUT1
    //Event 1 for PWM Duty Cycle
    SCT0->MATCHREL[1] = (6-1);            //Match 1 @ 6/44MHz = 136,36ns
    SCT0->EV[1].STATE = 0xFFFFFFFF;       //Event 1 happens in all states
    SCT0->EV[1].CTRL = (1 << 0) | (1 << 12); //Match 1 condition only
    SCT0->OUT[1].CLR = (1 << 1);         //Event 1 will clear SCT0_OUT1
    //*****
}
```

Abbildung 91: Funktion „SCTimer\_CamCLK\_Init“ für die Initialisierung der Events für den Kamera-Takt; Teil der Datei „camera.c“

Um die Bildaufnahme starten zu können, muss mit dem seriellen Eingang der Kamera die Bildaufnahme getriggert werden, was über die Events 2 und 3 des SCTimers realisiert wird. Diese Events werden in der Funktion „SCTimer\_SIEvents\_Init“ initialisiert (siehe Abbildung 92). Wie bereits erwähnt, muss das Setzen des SI-Ausgangs, welcher über den SCTimer-Ausgang SCT0\_OUT0 am Pin 15 der Buchsenleiste J13 nach außen geführt ist (Port-Pin P3.26), mindestens 20ns vor der steigenden Flanke des Kamera-Takts erfolgen (Event 2, Set CAM\_SI). Das Rücksetzen des SI-Signals könnte theoretisch auch mit dem Event 0 (Set CAM\_CLK) erfolgen. Da im SCTimer allerdings noch genügend Events zur Verfügung stehen, wird das Event 3 (Clear CAM\_SI) so eingestellt, dass das SI-Signal mit der gleichen Zeit nach der steigenden Flanke des Kamera-Takts zurückgesetzt wird, mit der es vor der steigenden

Flanke gesetzt wurde. Deshalb wird für das Event 2 eine Zeit von 250ns (250ns - 272,72ns = -22,72ns) und für das Event 3 eine Zeit von 22,72ns eingestellt.

Das Event 3 wird in jedem Zyklus ausgeführt, während das Event 2 zum Setzen des SI-Signals in der Interrupt Service Routine des Timers CTIMER0 nur zyklisch erlaubt wird. Deshalb wird für das Event 2 im EV[2].State Register nicht 0xFFFFFFFF (Event wird in jedem Status ausgelöst) sondern 0 eingetragen (Auslösung des Events nur in Status 0). Da der Status während des Betriebs des SCTimers in dieser Anwendungsform nie verändert wird, wirkt eine 0 im Status Register wie eine vollständige Deaktivierung des Events, die in der Interrupt Service Routine des Timers kurzzeitig für ein einmaliges Auslösen aufgehoben wird. Der zyklische Eintritt in die Interrupt Service Routine legt daher die Bildaufnahmerate fest.

```
/****************************************************************************
 * Initialize Start Signal for Camera (SI)
 ****/
void SCTimer_SIEvents_Init(void)
{
    //*****
    //Configure Pin P[3][26] (J13 Pin15) (CAM_SI/SCT0_OUT0)
    IOCON->PIO[3][26] &= 0xFFFFFFF0;      //clear FUNC bits of P3.26
    IOCON->PIO[3][26] |= 0x02;           //Set FUNC bits to SCT0_OUT0 function FUNC2 P3.26
    GPIO->DIR[3] |= 1<<26;            //set PIO3_26 (SCT0_OUT0) to output
    //*****


    //*****
    //Event 2 for SI Set Event
    SCT0->MATCHREL[2] = (11-1);          //Match 2 @ 11/44MHz = 250ns
    SCT0->EV[2].STATE = 0;                //Event 2 happens only in State 0
    SCT0->EV[2].CTRL = (2 << 0)|(1 << 12); //Match 2 condition only
    SCT0->OUT[0].SET = (1 << 2);        //Event 2 will set SCT0_OUT0
    //*****


    //*****
    //Event 3 for SI reset Event
    SCT0->MATCHREL[3] = (1-1);           //Match 3 @ 1/44MHz = 22,727ns
    SCT0->EV[3].STATE = 0xFFFFFFFF;     //Event 3 happens in every state
    SCT0->EV[3].CTRL = (3 << 0)|(1 << 12); //Match 3 condition only
    SCT0->OUT[0].CLR = (1 << 3);       //Event 3 will clear SCT0_OUT0
    //*****
}
```

Abbildung 92: Funktion „SCTimer\_SIEvents\_Init“ für die Initialisierung der Events für den seriellen Eingang der Kamera; Teil der Datei „camera.c“

Wird bei einem Timer Überlauf des Timers CTIMER0 ein Interrupt Request ausgelöst, wird daraufhin die Interrupt Service Routine „CTIMER0\_IRQHandler“ ausgeführt (siehe Abbildung 93). In ihr wird nicht nur das Event 2 kurzfristig aktiviert, sondern auch der Pixel-Zähler zurückgesetzt, damit bei einer neuen Bildaufnahme die vom ADC aufgenommenen Spannungswerte wieder den zugehörigen Pixeln zugeordnet werden. Die Häufigkeit des Eintretens in die Interrupt Service Routine wird durch den Überlaufwert des Timers CTIMER0 im Match Register 0 festgelegt und beeinflusst direkt die Belichtungsdauer.

```
/*
 * Allow Start Signal for Camera (SI) to happen
 * Allows Event 2 to happen once every interrupt (Set and Clear SI)
 */
void CTIMER0_IRQHandler(uint32_t flags)
{
    pixelCounter = 0; //new picture start at pixel 0

    /*
    SCT0->EV[2].STATE = 0xFFFFFFFF;           //Event 2 happens in all states
    SCT0->CTRL &= ~(1 << 2);                //Unhalt SCT0 by clearing bit 2 of CTRL
    SCT0->EV[2].STATE = 0;                     //Event 2 happens only in State 0
    SCT0->CTRL &= ~(1 << 2);                //Unhalt SCT0 by clearing bit 2 of CTRL
    */

    CTIMER_ClearStatusFlags(CTIMER0,kCTIMER_Match0Flag);
}
```

Abbildung 93: Interrupt Service Routine „CTIMER0\_IRQHandler“ für die kurzzeitige Aktivierung des Events 2 (Set CAM\_SI) und das Rücksetzen der Pixel Zähler-Variable; Teil der Datei „camera.c“

Zuletzt muss der SCTimer auch die Aufgabe des ADC-Triggers erfüllen. Die Initialisierung des dafür zuständigen Event 4 wird in der Funktion „SCTimer\_ADCTrigger\_Init“ realisiert (siehe Abbildung 94). Anders als die SCTimer-Ausgänge SCT0\_OUT0 (CAM\_SI) und SCT0\_OUT1 (CAM\_CLK) wird der ADC-Trigger (SCT0\_OUT4) nicht auf einen Pin nach Außen geführt, sondern lediglich der ADC so konfiguriert, dass der SCTimer Ausgang SCT0\_OUT4 als Hardware Trigger für den ADC verwendet wird.

```
/*
 * Initialize Trigger Event for ADC
 */
void SCTimer_ADCTrigger_Init(void)
{
    /*
    //Event 4 for ADC Trigger Event
    SCT0->MATCHREL[4] = (9-1);           //Match 4 @ 9/44MHz = 204,54ns (Cam_AO settlingTime Min120ns)
    SCT0->EV[4].STATE = 0xFFFFFFFF;      //Event 4 happens in all states
    SCT0->EV[4].CTRL = (4 << 0)|(1 << 12); //Match 4 condition only

    //SCT0_OUT4 is ADC hardware trigger
    SCT0->OUT[4].SET = (1 << 4);        //Event 4 will set SCTx_OUT4
    SCT0->OUT[4].CLR = (1 << 0);        //Event 0 will clear SCTx_OUT4
    */
}
```

Abbildung 94: Funktion „SCTimer\_ADCTrigger\_Init“ für die Initialisierung des Events für den Start einer ADC-Konversion; Teil der Datei „camera.c“

Nach der Konfiguration der Module SCT0 und CTIMER0 muss lediglich noch der ADC konfiguriert werden (Funktion „ADC\_Config“, siehe Abbildung 95). Als Messeingang für das Analogsignal der Kamera (CAM\_AO) wird der ADC-Eingang ADC0IN4 verwendet (J12 Pin2, Port-Pin P0.16). Zuerst wird die ADC Peripherie aktiviert und die Referenzspannungen eingestellt. Im Anschluss daran wird der Takt für den ADC aktiviert und eine Kali-

brierung gestartet (siehe Funktion „ADC\_Calibration“). Ist die Kalibrierung des ADC zu Ende, wird der ADC mit einem Takt von 73,33MHz initialisiert (max. 80MHz), eine Auflösung von 8bit festgelegt (Wertebereich 0-256) und die Sampling-Zeit eingestellt (hier 4,5x ADC Clock Cycle = 61,26ns). Die Konversionszeit beträgt hier für eine 8bit Auflösung  $(11 + 4,5) \cdot ADC_{CLKCycle} = 211\text{ns}$ . Im Anschluss daran wird noch die Trigger-Quelle auf SCT0\_OUT4 festgelegt und die Interrupt Service Routine des ADC für jede vollendete Konversion aktiviert.

```
/**************************************************************************
 * Configure ADC
 **************************************************************************/
void ADC_Config(void)
{
    //*****
    //Configure Pin P[0][16] (J12 Pin2) (ADC0IN4)
    IOCON->PIO[0][16] &= 0xFFFFFFFF; //Clear FUNC bits of P0.16 Func 0 is ADC0_4
    IOCON->PIO[0][16] &= ~(1 << 8); //Disable DIGIMODE --> Analog input
    GPIO->DIR[0] &= ~(1 << 16); //Set PIO0_16 (ADC0_4) to input
    //*****


    //*****
    //Power up ADC0 peripheral in normal-power mode
    SYSCON->PDRUNCFG[0] &= ~(1 << 10); //Power Up ADC (PDEN_ADC0)
    SYSCON->PDRUNCFG[0] &= ~(1 << 9); //Power Up Analog Supply for ADC (PDEN_VD2_ANA)
    SYSCON->PDRUNCFG[0] &= ~(1 << 19); //VDDA to ADC (PDEN_VDDA)
    SYSCON->PDRUNCFG[0] &= ~(1 << 23); //VREFP to ADC (PDEN_VREFP)
    //*****


    //*****
    //Clock Enable and Peripheral Reset
    SDK_DelayAtLeastUs(20U, SDK_DEVICE_MAXIMUM_CPU_CLOCK_FREQUENCY); //Wait at least 20us after Power Up !!!!!!!!
    SYSCON->AHBCLKCTRL[0] |= (1 << 27); //ADC0 CLK Enable
    SYSCON->PRESETCTRLCLR[0] |= (1 << 27); //Clear ADC0 peripheral reset
    //*****


    ADC_Calibration(); //ADC Calibration Sequence

    //*****
    //ADC fundamental configuration
    ADC0->CTRL |= 3-1; //CLK Divider 220MHz/3 = 73,33MHz (80MHz max)
    //Synchronous Mode System Clock is Reset value
    ADC0->CTRL &= ~ADC_CTRL_RESOL_MASK; //Delete Resolution bits
    ADC0->CTRL |= (0b01 << ADC_CTRL_RESOL_SHIFT); //Resolution 8bit (12bit is max)
    ADC0->CTRL |= (0b010 << 12); //Sampling Time is 4.5x ADC Clock Cycle --> 61,366ns
    //*****


    //*****
    //ADC Sequence A configuration
    ADC0->SEQ_CTRL[0] &= ~(1 << 31); //Sequence A Disable for Configuration
    ADC0->SEQ_CTRL[0] |= (1 << 4); //Select Channel Input 4 for ADC Conversion in Sequence A
    ADC0->SEQ_CTRL[0] |= (3 << 12); //SCTIMER Output 4 Trigger SCT0_OUT4
    ADC0->SEQ_CTRL[0] |= (1 << 18); //TRIGPOL positive Edge
    ADC0->SEQ_CTRL[0] |= (1 << 19); //Bypass Trigger Synchronization
    //ADC0->SEQ_CTRL[0] |= (1 << 30); //Mode: 0(Rst.Value)=End of Conversion / 1=End of Sequence
    ADC0->SEQ_CTRL[0] |= (1 << 31); //Sequence A Enable
    //*****


    //*****
    //ADC Interrupt configuration
    ADC0->INTEN |= (1<<0); //Interrupt Enable ADC0 Sequence A
    NVIC_SetPriority(ADC0_SEQA_IRQn, 0); //Enable NVIC interrupt for sequence A.
    EnableIRQ(ADC0_SEQA_IRQn); //Enable ADC Sequence A Interrupt
    //Enabling NVIC will block DMA trigger!!!
    //*****


}

}
```

Abbildung 95: Funktion „ADC\_Config“ für die Konfiguration des Analog-Digital-Wandlers; Teil der Datei „camera.c“

Die ADC-Kalibrierung folgt einem festen Ablauf. Damit es unerheblich ist, wann die Kalibrierfunktion „ADC\_Calibration“ (siehe Abbildung 96) aufgerufen wird, wird zuerst die aktuelle Konfiguration gespeichert und am Ende der Kalibrierung diese wieder geladen. Dazwischen wird der Clock-Divider auf unter 30MHz umprogrammiert (maximale Kalibrierfrequenz) und der Kalibrierzyklus gestartet.

Nach jeder ADC-Konversion wird ein Interrupt Request ausgelöst und die Interrupt Service Routine „ADC0\_SEQA\_IRQHandler“ (siehe Abbildung 97) ausgeführt. In ihr werden die vom ADC ermittelten Werte der Reihe nach mithilfe des Pixel-Zählers, welcher in der Interrupt Service Routine des CTIMER0 beim Start einer Bildaufnahme jedes mal zurückgesetzt wird, in ein Ergebniswert-Array gespeichert (uint8\_t pixelValues[pixelCounter]).

```
/**************************************************************************
 * ADC Calibration Sequence
 **************************************************************************/
void ADC_Calibration(void)
{
    uint32_t tmp = ADC0->CTRL;                      //Save current ADC configurations

    ADC0->CTRL |= ADC_CTRL_CLKDIV(2U);           //Clock Divider --> 220MHz/3 = 73,33MHz (80MHz max.)
    ADC0->STARTUP = ADC_STARTUP_ADC_ENA_MASK;     // Start ADC module

    if (0UL == (ADC0->STARTUP & ADC_STARTUP_ADC_ENA_MASK)) {
        return; // ADC is not powered up.
    }

    //Calibration frequency and cycle (max. 30MHz!!!!)
    ADC0->CTRL |= ADC_CTRL_CLKDIV(7U);           //Clock Divider --> 220MHz/8
    ADC0->CALIB = ADC_CALIB_CALIB_MASK;          // Launch calibration cycle.
    SDK_DelayAtLeastUs(100U, SDK_DEVICE_MAXIMUM_CPU_CLOCK_FREQUENCY); // Wait for at least 81 ADC conversions
    if (ADC_CALIB_CALIB_MASK == (ADC0->CALIB & ADC_CALIB_CALIB_MASK)) {
        return; // Calibration timed out.
    }
    //Rewrite ADC Configurations
}

ADC0->CTRL = tmp;
```

Abbildung 96: Funktion „ADC\_Calibration“ für die Konfiguration des Analog-Digital-Wandlers; Teil der Datei „camera.c“

```
/*
 * ADC0 Sequence A conversion finished ISR (Get pixel charge values)
 */
void ADC0_SEQA_IRQHandler(void)
{
    if(pixelCounter<128) //Save Pixel Values
    {
        pixelValues[pixelCounter] = ADC0->SEQ_GDAT[0] >> 8; //Reading current pixel
        pixelCounter++; //Next ISR is next pixel
    }

    ADC0->FLAGS = (1<<28); //Delete interrupt flags
    SDK_ISR_EXIT_BARRIER;
}
```

Abbildung 97: Interrupt Service Routine „ADC0\_SEQA\_IRQHandler“ für die Ergebnisaufnahme nach jeder ADC-Konversion; Teil der Datei „camera.c“

Aus den vom ADC gemessenen Werten können mit dem Algorithmus aus Abbildung 98 am Ende einer Bildaufnahme sowohl der reelle Spannungswert, als auch der logische Wert für hell oder dunkel bestimmt werden. Da mit den 8bit Werten eine Regelung qualitativ genauso gut realisierbar ist, wie mit Spannungswerten, ist der Algorithmus lediglich zu Testzwecken in der Datei „camera.c“ eingebunden.

```
//Transfer ADC result values to voltages and logical values
uint8_t transferCounter; //Start with result 0
float cFactor = (float)1/256*(VREFp-VREFn);
for (transferCounter = 0; transferCounter<129; transferCounter++) //Go through all results
{
    pixelVoltages[transferCounter] = (int16_t)((float)pixelValues[transferCounter]*cFactor)+VREFn;
    if(pixelValues[transferCounter] >= 128)
    {
        pixelLogicValues[transferCounter] = 1; //Write logical 1 for "Bright"-Value
    }
    else
    {
        pixelLogicValues[transferCounter] = 0; //Write logical 0 for "Dark"-Value
    }
}
```

Abbildung 98: Algorithmus für die Bestimmung der Spannungswerte und logischen Werte der Pixel aus den vom ADC bestimmten 8bit Werten

Außer der bisher verwendeten Module (ADC0, SCTIMER0, CTIMER0) kann es auch sinnvoll sein, den Direct Memory Access (DMA) für die Ergebnisse des ADCs zu verwenden. Ein Direktzugriff des ADC auf den Speicher spart Rechenressourcen und Zeit, was die maximal mögliche Bildaufnahmerate weiter erhöht. Da bisher lediglich die Bildaufnahme realisiert werden konnte, ist es außerdem notwendig, die Aufbereitung der Messergebnisse zu implementieren. Eine Aufbereitung der Messergebnisse ist beispielsweise die automatische Belichtungszeitanpassung oder eine Verrechnung nebeneinanderliegender Pixelwerte für die bessere Erkennung der Fahrbahnbegrenzungen.

## 8 Verteilerplatinen

Anders als beim Fahrzeug von Herrn Arne Kullina werden bei dieser Fahrzeugversion zwei anstelle nur einer Verteilerplatine verbaut. Die Leistungsverteilung befindet sich auf der Grundplatte und die steck- und schraubbare Verteilung der Signale für die Fahrzeugperipherie auf der oberen Ebene.

### 8.1 Leistungsverteiler auf der unteren Fahrzeugebene

Von der Leistungsverteilerplatine auf der unteren Fahrzeugebene (siehe Abbildung 99) wird die gesamte Fahrzeugperipherie mit der Batteriespannung versorgt. Sowohl der Schaltspannungsregler für die Spannungsversorgung der Servo-Lenkung, als auch der Linearspannungsregler auf der oberen Verteilerplatine und die ESCs erhalten hier ihre Versorgungsspannung. Der Vorteil, dass auf der Grundplatte lediglich der Versorgungsspannungsabgriff erfolgt, liegt darin, dass das Fahrzeug seltener demontiert werden muss. Da auf der Grundplatte keine wichtige Schaltung verbaut ist, können die meisten Probleme gelöst werden, ohne auf die untere Fahrzeugebene zugreifen zu müssen.

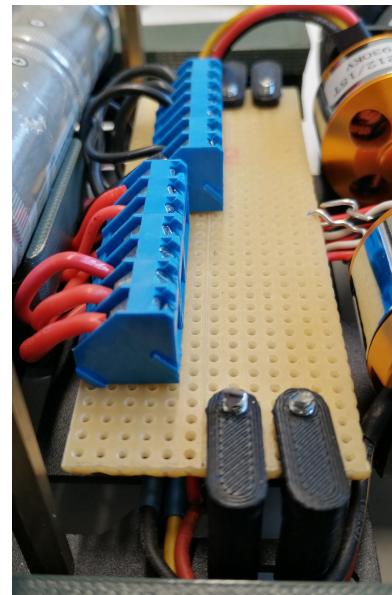


Abbildung 99: Leistungsverteilerplatine auf der unteren Fahrzeugebene

## 8.2 Signalverteilerplatine auf der oberen Fahrzeugebene

Auf der Signalverteilerplatine befindet sich neben der Drehzahlmessung (siehe Kapitel 4.5) auch die Verteilung der PWM-Signale der Antriebe und Servo-Lenkung und der Linearspannungsregler für die Spannungsversorgung des Controllers. In Abbildung 100 ist die gesamte Platine zu sehen. Die Drehzahlmessung ist in rosa hervorgehoben, die Versorgung der Servo-Lenkung in grün, die PWM-Signalverteilung des linken und rechten Antriebs in orange und violett, der Linearspannungsregler in braun, dessen Versorgung von der Leistungsverteilerplatine in rot und die Versorgungsspannungsanschlüsse des Controllers in grau.

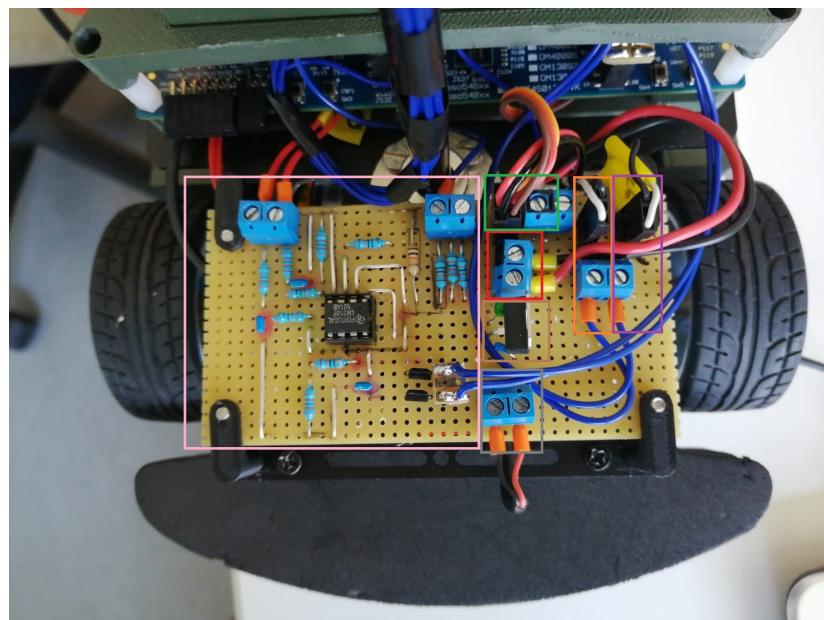


Abbildung 100: Signalverteilerplatine auf der oberen Fahrzeugebene

## 9 Zusammenfassung, Fazit und Ausblick

### 9.1 Zusammenfassung und Fazit

Der Bausatz des Fahrzeugs wurde erfolgreich zusammengesetzt und jedes der 3D-Anbauteile konnte konstruiert, gedruckt und montiert werden. Nach der ebenfalls erfolgreichen Implementierung der Fahrzeuglenkung konnte auch die Software für die Ansteuerung der Antriebe vollständig programmiert werden. Durch die Umkonfiguration der ESCs wurde auch das Problem der nicht funktionierenden Start-Sequenz der BLDC-Motoren gelöst. Jetzt ist es möglich, die Motoren zu initialisieren, anzusteuern und dabei in der Drehzahl zu variieren. Im Anschluss daran sind die Vor- und Nachteile von verschiedenen Drehzahlregelungen eruiert worden. Die Hardware-Implementierung der letztendlich erwählten Methode wurde dann mithilfe einer Lochrasterplatine realisiert, bevor auch die Implementierung der Software dafür erfolgreich beendet werden konnte. Die Entwicklung und Programmierung des Bedienungsboards konnte trotz einiger Schwierigkeiten bei der Initialisierung der verwendeten I2C-Schnittstelle ebenfalls bewältigt werden. Die Menüführung zum Auslesen und Verändern der Fahrzeugparameter ist, soweit bereits möglich, erledigt. Zu guter Letzt wurde auch die zyklische Bildaufnahme mit der Zeilenkamera zur Funktionalität des Fahrzeugs hinzugefügt. Eine Teilnahme am NXP Cup ist mit dem aktuellen Stand des Fahrzeugs allerdings noch nicht möglich, da die Streckenerkennung mithilfe der Kamera-Daten und die Regelung des Lenkwinkels und der Fahrzeuggeschwindigkeit noch nicht implementiert werden konnten. Die Erstellung der Hardware des Fahrzeugs ist, ausgenommen der Hinderniserkennung, vollständig abgeschlossen.

Die bisher realisierten Funktionen sind in diesem Bericht umfangreich erklärt und im Programm ausführlich kommentiert worden. Bei der Programmierung wurde auf eine nachvollziehbare Benennung der Funktionen und Variablen geachtet.

Zusammenfassend ist zu sagen, dass die Entwicklung während der beiden Semester gut vorangeschritten ist. Da das übergeordnete Ziel von Anfang an ein ordentliches und nachvollziehbares Ergebnis war, war die Teilnahme am NXP Cup 2021 leider nicht mehr möglich.

## 9.2 Ausblick

Bei der Fortführung dieses Projekts können und müssen noch einige Funktionen entwickelt bzw. optimiert werden. Zum einen ist es bei der Zeilenkamera sinnvoll, verschiedene Linsen zu testen und deren Vor- und Nachteile zu erörtern. Ein weiterer Entwicklungsschritt bei der Kamera ist die bereits erwähnte, automatische Belichtungszeitanpassung. Auch die Verwendung einer zweiten Kamera kann durchaus von Vorteil sein. Als letzter Punkt in Bezug auf die Kamera sollten die vom ADC aufgenommenen Pixelwerte zur Live-Überwachung auf dem Display dargestellt werden.

Für die Lösung der Problematik, dass nicht der gesamte Drehzahlbereich gemessen werden kann, sollte eine zusätzliche, zuschaltbare Referenzspannungsstufe implementiert werden oder der Tiefpass ersetzt werden. Bei der Ersetzung des Tiefpasses kann entweder die Grenzfrequenz angepasst oder ein aktiver Tiefpass verwendet werden. In jedem Fall ist es durchaus zu empfehlen, die Drehzahlmessschaltung mitsamt der Signalverteilung professionell zu fertigen.

Bei der Inbetriebnahme der Antriebe konnte erkannt werden, dass das Drehverhalten bei der Reifen unterschiedlich ist. Zur Lösung dieses Problems kann entweder das Getriebe ersetzt (z.B. durch einen Keilriemen) oder die Reibung der beiden Antriebe bestmöglich aufeinander abgestimmt werden.

Weitere Entwicklungsschritte, welche für die Teilnahme am NXP-Cup notwendig sind, sind die Implementierung einer Drehzahl- und Lenkwinkelregelung und einer Erkennung der Ziellinie, aber auch die Einbindung des Ultraschallboards zur Hinderniserkennung. Zuallerletzt kann auch der verwendete Nickel-Metallhydrid-Akku durch einen Lithium-Ionen-Akku mit einer größeren Energiedichte ersetzt werden, um bei gleicher Spannung Gewicht zu sparen.

## Abbildungsverzeichnis

1	Grundplatte des Standardbausatzes für das Fahrzeug . . . . .	3
2	Grundplatte des Fahrzeugs . . . . .	3
3	Obere Ebene des Standardbausatzes für das Fahrzeug . . . . .	4
4	Obere Ebene des Fahrzeugs . . . . .	4
5	Konstruktionsbild des oberen Stoßstangenteils . . . . .	5
6	Konstruktionsbild des unteren Stoßstangenteils . . . . .	5
7	Gedrucktes oberes und unteres Stoßstangenteil . . . . .	6
8	Konstruktionsbild der Halterung des Ultraschallboards . . . . .	6
9	Gedruckte Halterung des Ultraschallboards . . . . .	6
10	Draufsicht der Stoßstange und der Ultraschallboard-Halterung . . . . .	6
11	Untersicht der Stoßstange und der Ultraschallboard-Halterung . . . . .	7
12	Konstruktionsbild der Akku-Halterung . . . . .	7
13	Gedruckte Akku-Halterung . . . . .	7
14	Konstruktionsbild der Seitenschweller . . . . .	8
15	Gedruckte Seitenschweller . . . . .	8
16	Montierte Akku-Halterung und Seitenschweller . . . . .	8
17	Konstruktionsbild der Controller-Halterung . . . . .	9
18	Konstruktionsbild der Platinen-Halterung des Bedienungsboards . . . . .	9
19	Konstruktionsbild der Abdeckung des Bedienungsboards . . . . .	9
20	Konstruktionsbild der Taster-Verlängerung . . . . .	9
21	Gedruckte Controller-Halterung . . . . .	10
22	Gedruckte Taster-Verlängerung . . . . .	10
23	Gedruckte Platinen-Abdeckung des Bedienungsboards . . . . .	10
24	Gedruckte Platinen-Halterung für das Bedienungsboard . . . . .	10
25	Fertig montierte Controller- und Bedienungsboard-Halterung . . . . .	11
26	Fertig montierte Controller- und Bedienungsboard-Halterung mit Abdeckung	11
27	Konstruktionsbild einer Motorcontroller-Halterung . . . . .	12
28	Gedruckte Motorcontroller-Halterungen . . . . .	12
29	Die auf der Grundplatte montierten Motorcontroller . . . . .	12
30	Konstruktionsbild einer einfachen Platinenhalterung . . . . .	13
31	Gedruckte Platinenhalterungen . . . . .	13
32	Mit den Platinenhalterungen montierte Leistungsverteilerplatine . . . . .	13
33	Kameramontage-Komponente der Kamera-Halterung . . . . .	14
34	Stangenmontage-Komponente der Kamera-Halterung . . . . .	14
35	Fertig montierte Kamera mit Kamerahalterung . . . . .	15
36	Konstruktionsbild der Antriebsabdeckung . . . . .	15
37	Gedruckte Abdeckung der Antriebe . . . . .	15
38	Fertig montierte Abdeckung der Antriebe . . . . .	16

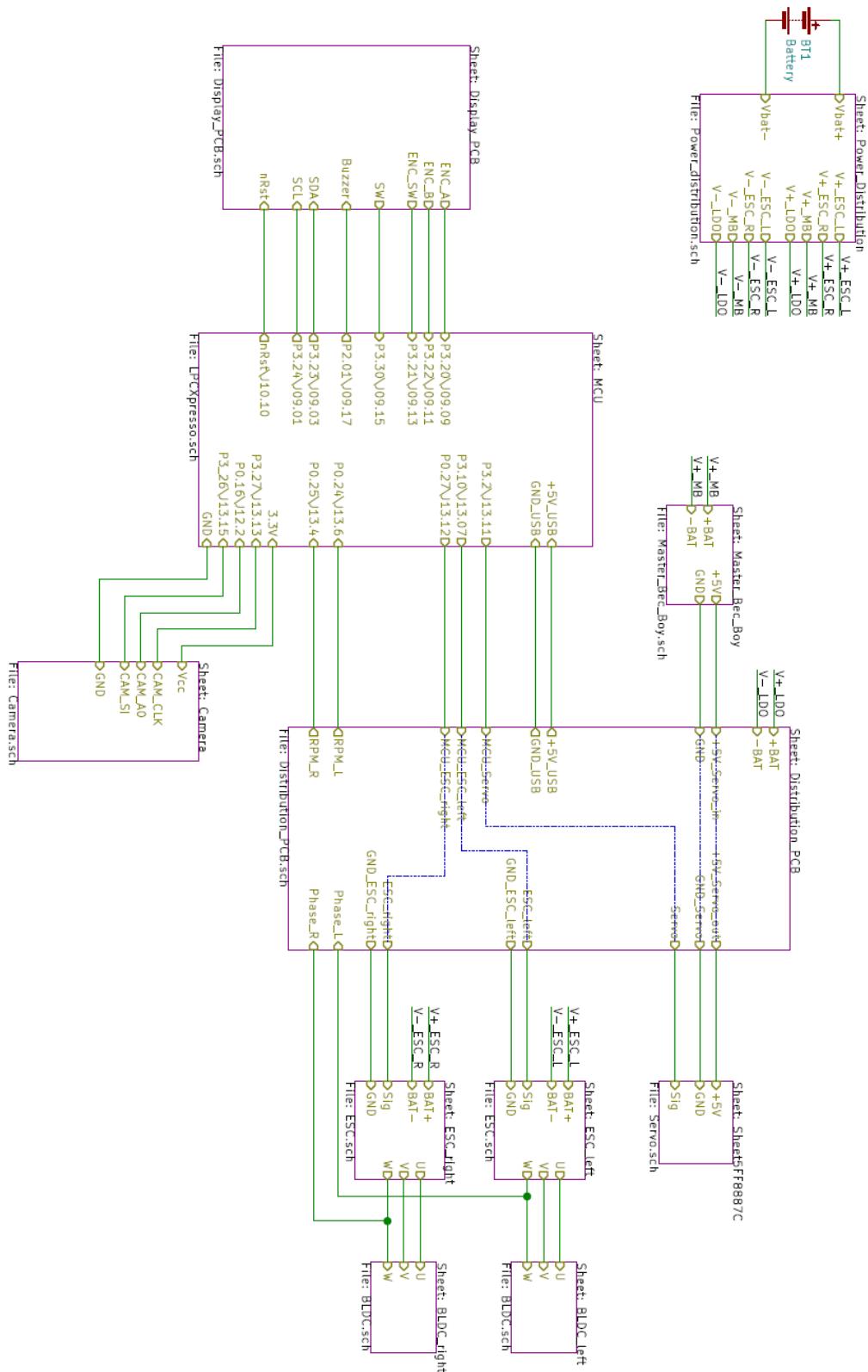
39	Controllerplatine LPCXpresso54628 [Sem19] . . . . .	17
40	Skizze zur Beschaltung eines ESCs und BLDC-Motors . . . . .	19
41	Montage der Antriebskomponenten . . . . .	20
42	Initialisierungssequenz der ESCs . . . . .	21
43	Programmierung des Arduino Nano zur ESC-Konfiguration . . . . .	22
44	Parameter der neuen ESC-Konfiguration . . . . .	22
45	Relevante Zeilen der Datei „drive.h“ . . . . .	23
46	Extremwerte der Antriebsgeschwindigkeit als Parameter aus dem EEPROM . . . . .	24
47	Funktionen BLDC_Init und CTIMER3_Init der Datei „drive.c“ . . . . .	24
48	Funktion BLDC_Init_Task der Datei „drive.c“ . . . . .	26
49	Messaufbau zur Prüfung der Spannungsabhängigkeit der BLDC-Motoren . . . . .	27
50	Spannungsabhängigkeit der Drehzahl von der Versorgungsspannung . . . . .	29
51	Varianten der Drehzahlerfassung bei vorherigen Fahrzeugversionen . . . . .	30
52	Phaseneingangssignal und Ausgangssignal der Tiefpass-/Komparatorschaltung . . . . .	32
53	Schaltplan der Tiefpass-/Komparatorschaltung aus LTspice XVII . . . . .	33
54	Simulationsergebnis des Phasensignals nach dessen Tiefpassfilterung . . . . .	34
55	Messergebnis des Phasensignals nach dessen Tiefpassfilterung . . . . .	34
56	Pinbelegung des Operationsverstärkers LM358A . . . . .	35
57	Beispielhafte Messung des OPV-Ausgangssignals bei V+ = 5,0V; . . . . .	36
58	Schaltung der Drehzahlmessung und zugehöriges Layout . . . . .	37
59	Relevante Zeilen der Datei „rpmMeas.h“ . . . . .	38
60	Funktionen RPMMEAS_Init und CTIMER2_Init in der Datei „rpmMeas.c“ . . . . .	39
61	Interrupt Service Routinen für Zeitermittlung zwischen zwei Capture-Flanken . . . . .	40
62	Funktionen zur Berechnung des Mittelwerts der Zeiten zwischen den Capture-Flanken . . . . .	41
63	Fehlerhafte Rechtecksignal-Erzeugung bei niedrigen Drehzahlen . . . . .	42
64	Schaltplan der Tiefpass-/Komparatorschaltung nach der Schaltungserweiterung . . . . .	43
65	Servomotor mit Anschlussleitungen [DIY21] . . . . .	44
66	Montage der Lenkungskomponenten . . . . .	45
67	Relevante Zeilen der Datei „servo.h“ . . . . .	46
68	Extremwerte der Lenkwinkel als Parameter aus dem EEPROM . . . . .	46
69	Funktionen SERVO_Init, CTIMER1_Init und SERVO_Demo der Datei „drive.c“ . . . . .	47
70	SERVO_Demo Task der Datei „drive.c“ . . . . .	48
71	Schaltplan des Bedienungs-Boards 1 . . . . .	49
72	Schaltplan des Bedienungs-Boards 2 . . . . .	50
73	Flussablaufdiagramm Taster und Drehencoder einlesen . . . . .	51
74	Schaltplan OLED . . . . .	54
75	Schritt 1: Erstellen eines neuen Menüeintrags . . . . .	61
76	Schritt 2: Erhöhen der Anzahl der Menüeinträge um eins . . . . .	61
77	Schritt 3: Erstellen einer Funktion zum Menüaufruf . . . . .	61

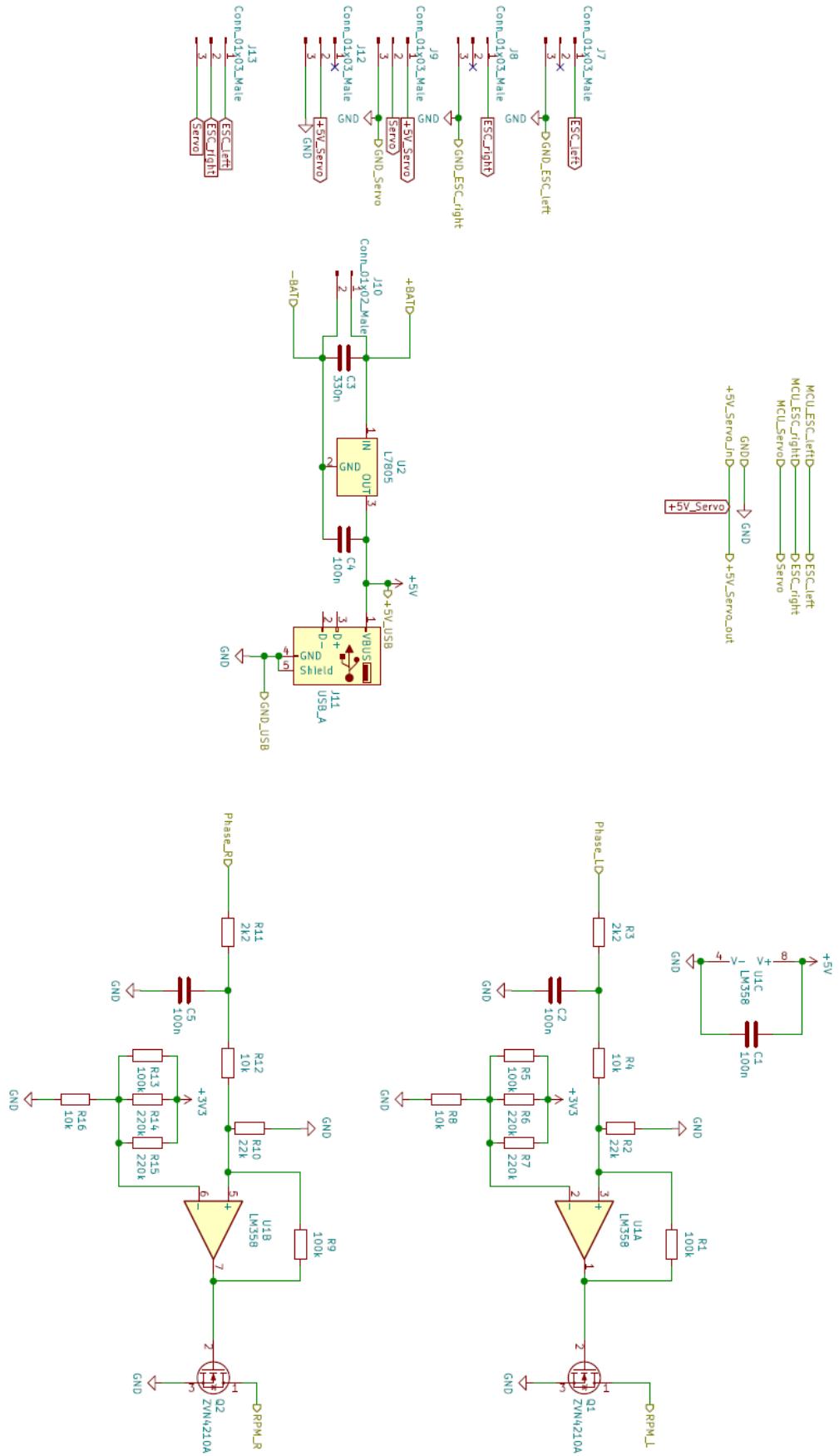
78	Schritt 4.1: Liste mit Einträgen des neuen Menüs erstellen . . . . .	62
79	Schritt 4.2: Hinzufügen des Menü Handlers im Header . . . . .	62
80	Schritt 5: Deklarieren der notwendige Variablen und Funktionen für die neuen Untermenüs . . . . .	62
81	Anzeige des neu erstellten Menüs auf dem OLED-Displayü . . . . .	63
82	Mit der Funktion „func_Page“ erzeugter, voll gefüllter Bildschirm . . . . .	63
83	Pinbelegung der Taos Zeilenkamera TSL1401R-LF [Tao06] . . . . .	65
84	Zeitverlauf einzelner Pixeltransfers aus Sicht der Kamera [Tao06] . . . . .	66
85	Zeitverlauf eines vollständigen Bildtransfers aus Sicht der Kamera [Tao06] . . . . .	66
86	Zeitlicher Ablauf einer einzelnen Bildaufnahme aus Sicht des Controllers . . . . .	67
87	Prototypen der Kamerafunktionen in der Datei „camera.h“ . . . . .	68
88	Funktion „CAM_Init“ aus der Datei „camera.c“ . . . . .	68
89	Funktion „CTIMER0_Init“ aus der Datei „camera.c“ . . . . .	69
90	Funktion „SCTimer_Clock_Config“ aus der Datei „camera.c“ . . . . .	70
91	Funktion „SCTimer_CamCLK_Init“ aus der Datei „camera.c“ . . . . .	71
92	Funktion „SCTimer_SIEvents_Init“ aus der Datei „camera.c“ . . . . .	72
93	Interrupt Service Routine „CTIMER0_IRQHandler“ aus der Datei „camera.c“ . . . . .	73
94	Funktion „SCTimer_ADCTrigger_Init“ aus der Datei „camera.c“ . . . . .	73
95	Funktion „ADC_Config“ aus der Datei „camera.c“ . . . . .	74
96	Funktion „ADC_Calibration“ aus der Datei „camera.c“ . . . . .	75
97	Interrupt Service Routine „ADC0_SEQA_IRQHandler“ aus der Datei „camera.c“ . . . . .	76
98	Algorithmus für die Bestimmung der Spannungswerte und logischen Werte der Pixel . . . . .	76
99	Leistungsverteilerplatine auf der unteren Fahrzeugebene . . . . .	77
100	Signalverteilerplatine auf der oberen Fahrzeugebene . . . . .	78

## Literatur

- [DIY21] DIYELECTRONICS: *MG996R Servo Motor / 120° Servo Motor – 10kg Stalling Torque.* Website, 2021. – Online erhältlich unter <https://www.diyelectronics.co.za/store/servos/766-towerpro-mg996r-servo-motor.html>; zuletzt abgerufen am 27. März 2021
- [mika] MIKROCONTROLLER.NET: *Drehgeber.* Website, . – Online erhältlich unter <https://www.mikrocontroller.net/articles/Drehgeber>; zuletzt abgerufen am 20. Juni 2021
- [mikb] MIKROCONTROLLER.NET: *Entprellung.* Website, . – Online erhältlich unter [http://www.mikrocontroller.net/articles/Entprellung](https://www.mikrocontroller.net/articles/Entprellung); zuletzt abgerufen am 20. Juni 2021
- [Sem19] *LPC546xx User Manual.* : *LPC546xx User Manual*, 2019
- [SSD11] *SSD1309 Advance Information.* : *SSD1309 Advance Information*, 7 2011
- [Tao06] *TSL1401R-LF User Manual.* : *TSL1401R-LF User Manual*, 2006
- [TI] TEXAS INSTRUMENTS, Inc.: *LMx58\_LM2904 PSPICE Model (Rev. A).* – Online erhältlich unter <https://www.ti.com/product/LM358#design-development>; zuletzt abgerufen am 21. Juni 2021
- [Wil20] WILKENS, Matthias: *Autonomous Driving Robot Student Teams Work from Home to Win the NXP Cup Electromaker Innovation Challenge.* Website, 2020. – Online erhältlich unter <https://www.nxp.com/company/blog/autonomous-driving-robot-student-teams-work-from-home-to-win-the-nxp-cup-electromaker-innovation-challenge:BL-AUTONOMOUS-DRIVING-ROBOT-NXP-CUP>; zuletzt abgerufen am 19. März 2021

## Anhang 1: Schaltplan





## Anhang 2: Pin Assignment

