



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:

30.07.2021

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 Hochschule 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

Vorwort und Projekt-Links	II
Inhaltsverzeichnis	III
Abkürzungsverzeichnis	V
1 Einführung	1
1.1 Zielsetzung	1
1.2 Konzept	1
2 Aufbau des Fahrzeugs	3
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
3 Controllerplatine und ProgrammierTool	17
4 Antrieb des Fahrzeugs	18
4.1 BLDC-Antrieb und Motorcontroller	18
4.2 Montage der Antriebskomponenten	19
4.3 Konfiguration der Motorcontroller	20
4.4 Programmierung des Antriebsbausteins	22
4.5 Drehzahlmessung	26
4.5.1 Erörterung der Notwendigkeit einer Drehzahlmessung	26
4.5.2 Auswahl des Messprinzips	30
4.5.3 Hardware für die Drehzahlmessung	32
4.5.4 Ausblick Schaltungserweiterung	38
4.5.5 Programmierung des Drehzahlmessungsbausteins	40
5 Lenkung des Fahrzeugs	45
5.1 Servoantrieb	45
5.2 Montage der Lenkungskomponenten	46
5.3 Programmierung des Lenkungsbausteins	47

6 Bedienungsboard	50
6.1 Schaltplan	50
6.2 Programmierung der Steuerelemente	51
6.2.1 Taster auswerten	52
6.2.2 Drehencoder auswerten	53
6.3 Programmierung der Anzeige	54
6.3.1 Funktionsweise des organischen Leuchtdioden-Displays	54
6.3.2 I2C Ansteuerung	55
6.3.3 Konfiguration des Displays	56
6.4 Darstellungen auf dem organischen Leuchtdioden-Display	57
6.4.1 Pixelansteuerung	58
6.4.2 Rechteck zeichnen	58
6.4.3 Buchstabe darstellen	58
6.4.4 Text darstellen	58
6.4.5 Bilder darstellen	58
6.5 Menüführung	59
7 Streckenerkennung des Fahrzeugs	60
7.1 Kamera des Fahrzeugs	60
7.2 Ablauf einer Bildaufnahme	62
7.3 Programmierung der zyklischen Bildaufnahme	63
8 Verteilerplatinen	72
8.1 Leistungsverteiler auf der unteren Fahrzeugebene	72
8.2 Signalverteilerplatine auf der oberen Fahrzeugebene	72
9 Zusammenfassung, Fazit und Ausblick	74
9.1 Zusammenfassung und Fazit	74
9.2 Ausblick	74
Abbildungsverzeichnis	VI
Quellenverzeichnis	IX
Anhang	X

Abkürzungsverzeichnis

BLDC-Motor	Brushless Direct Current Motor
DC-Motor	Gleichstrommotor
RC	Remote-Control
ESC	Electronic Speed Controller
PWM	Pulsweitenmodulation
NXP	Next Experience
CPU	Central Processing Unit
ABS-Kunststoff	Acrylnitril-Butadien-Styrol Copolymer
SLA-Drucker	3D-Drucker mit Stereolithographie-Verfahren
OLED-Display	organisches Leuchtdioden-Display
I2C	Inter Integrated-Circuit
RAM	Random-Access Memory
DMA	Direct Memory Access
EEPROM	Electrically Erasable Programmable Read-Only Memory
PWL-File	Piecewise Linear Function in LTspice
OPV	Operationsverstärker
TI	Texas Instruments
op amp	(Dual) Operational Amplifier

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 des Fahrzeugs und dessen 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 Verteilerplatten (Lochrasterplatten) mit Steck- und Schraubkontakten 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).

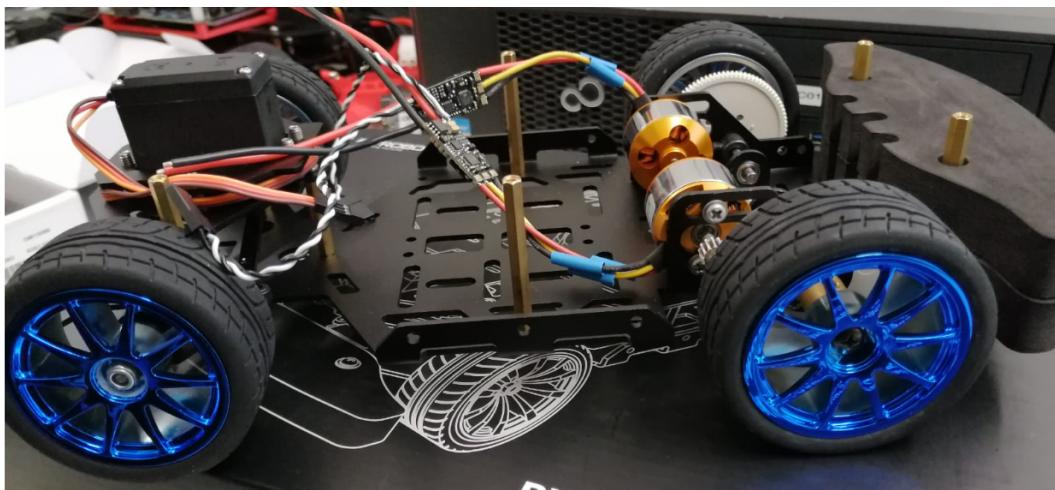


Abbildung 1: Grundplatte des Standardbausatzes für das Fahrzeug mit BLDC-Motoren, Motorcontrollern, Servomotor und Reifen

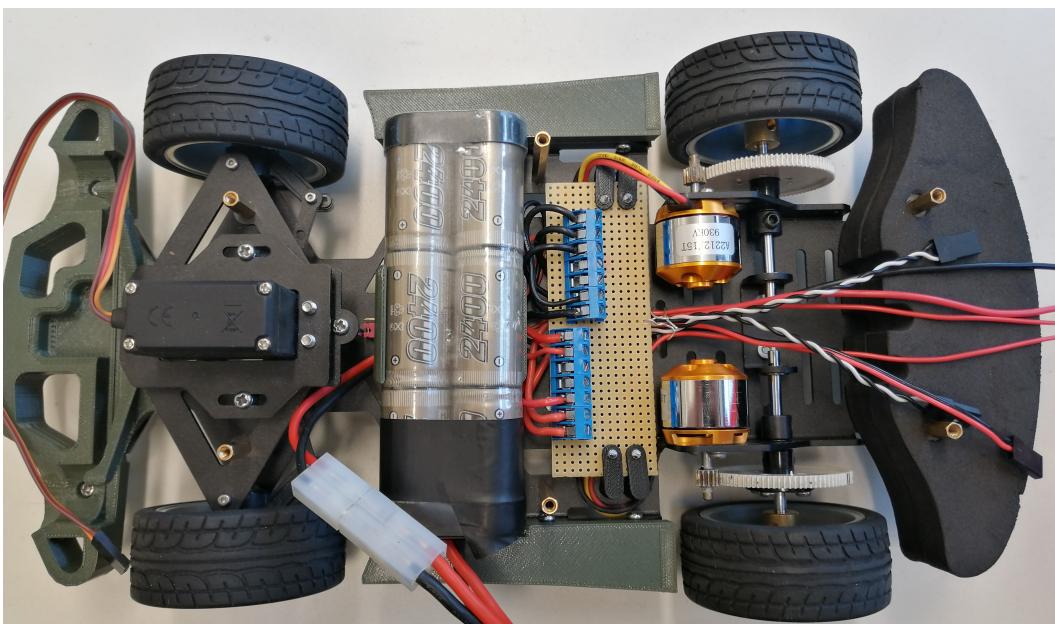


Abbildung 2: Grundplatte des Fahrzeugs mit Akku, Strom-Verteilerplatine und bereits montierter Fahrzeug-Peripherie

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

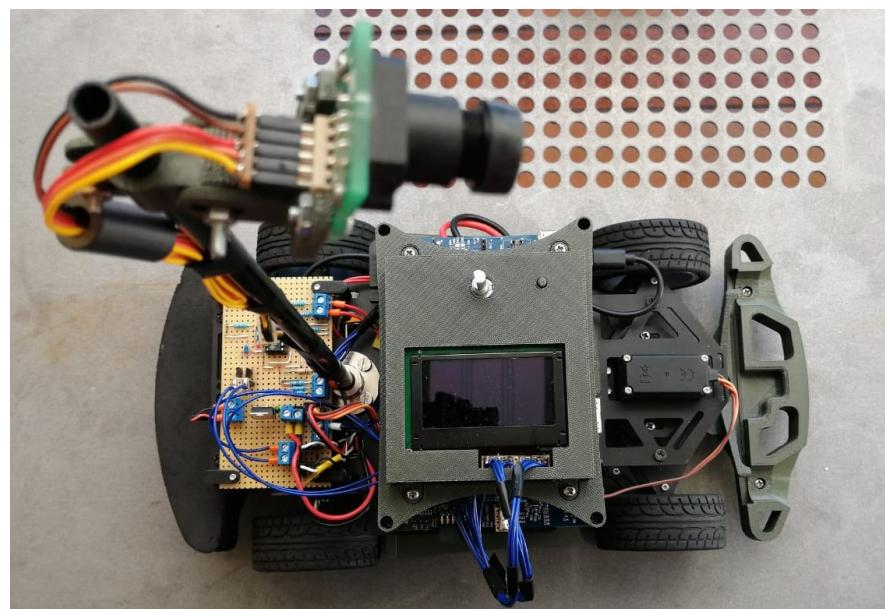


Abbildung 4: Obere Ebene des Fahrzeugs mit Controller, Bedienungsboard, Kamera und Verteilerplatine

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 Hochschule 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.

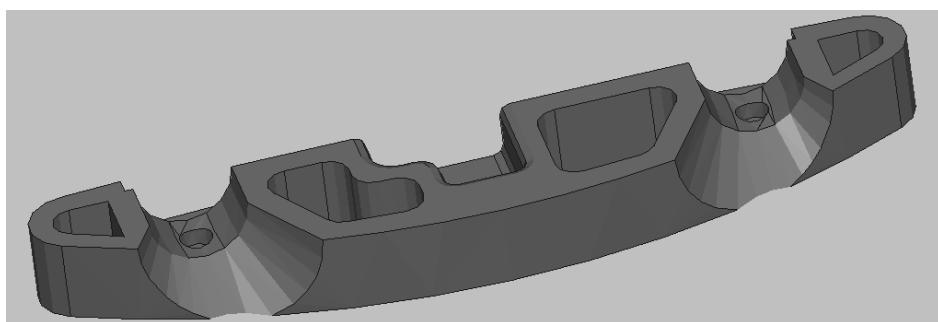


Abbildung 5: Konstruktionsbild des oberen Stoßstangenteils

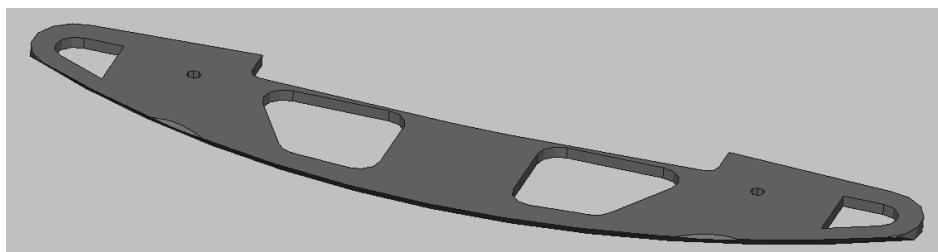


Abbildung 6: Konstruktionsbild des unteren Stoßstangenteils

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

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

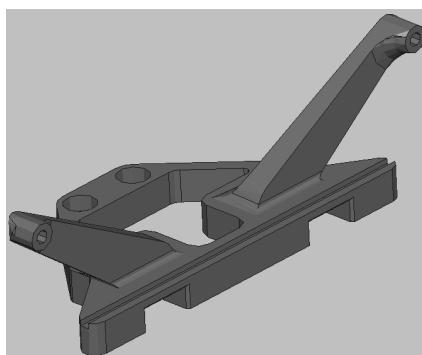


Abbildung 8: Konstruktionsbild der Halterung des Ultraschallboards



Abbildung 9: Gedruckte Halterung des Ultraschallboards

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.

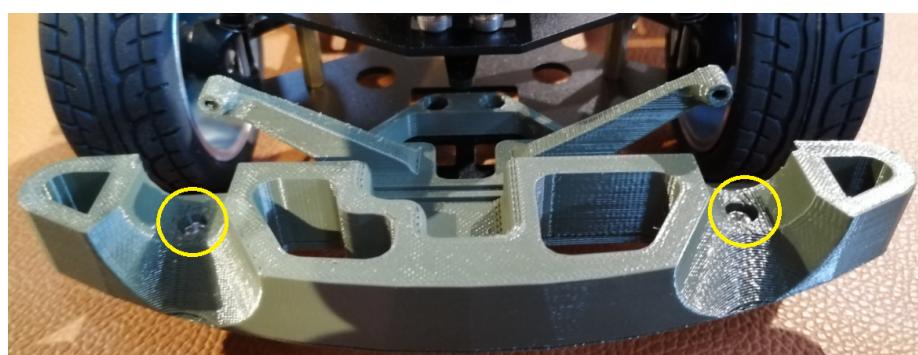


Abbildung 10: Draufsicht der Stoßstange und der Ultraschallboard-Halterung; Befestigungsschrauben der unteren Stoßstangenkomponente in gelb



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 Gasgeben 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 sauber 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 4 Schrauben auf der Grundplatte des Fahrzeugs montiert.

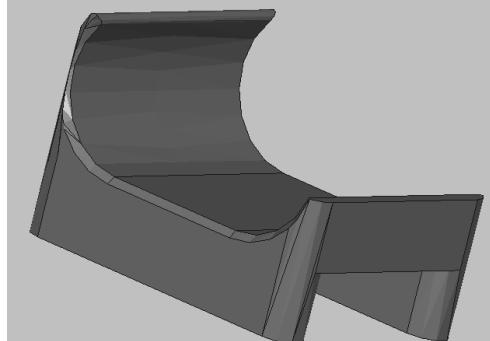


Abbildung 12: Konstruktionsbild der Akku-Halterung mit Kabeldurchführung

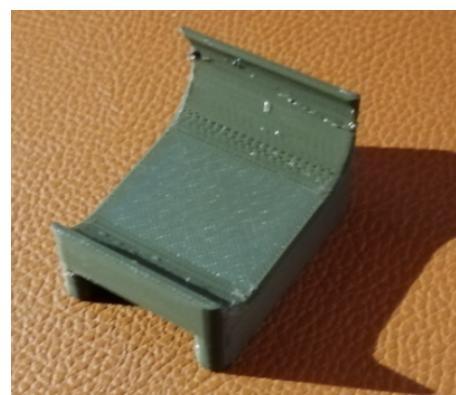


Abbildung 13: Gedruckte Akku-Halterung

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.

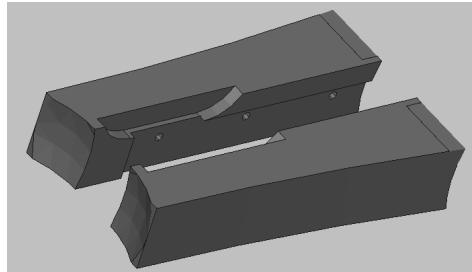


Abbildung 14: Konstruktionsbild der Seitenschweller, welche auch als Sicherung des Akkus zu den Seiten dienen



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.

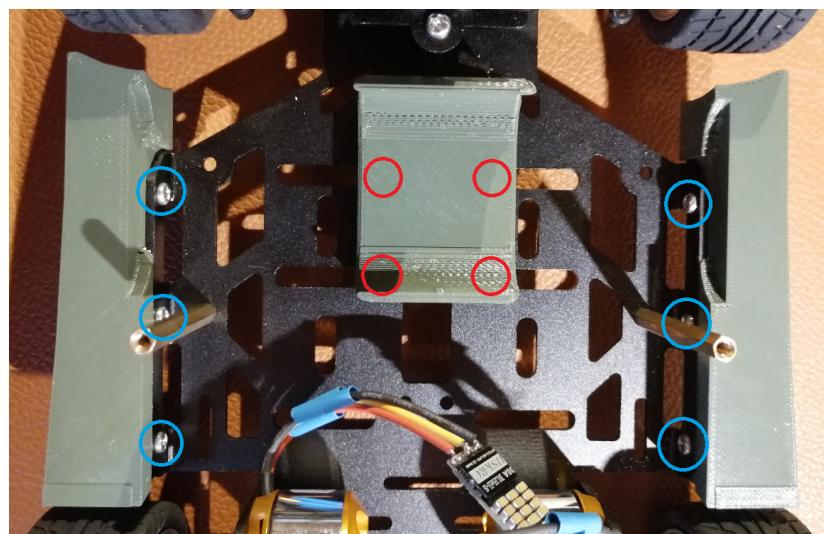


Abbildung 16: Montierte Akku-Halterung und Seitenschweller; Befestigungsschrauben der Seitenschweller in blau und Befestigungsschrauben der Akkuhalterung in rot (von unten, nicht sichtbar)

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 Bedientaster und Drehencoder für die Menüsteuerung Platz (Bedienungsboard). Zusätzlich zur Halterung der Platine (Konstruktionsbild siehe Abbildung 18) ist eine Abdeckung notwendig (Konstruktionsbild siehe Abbildung 19).

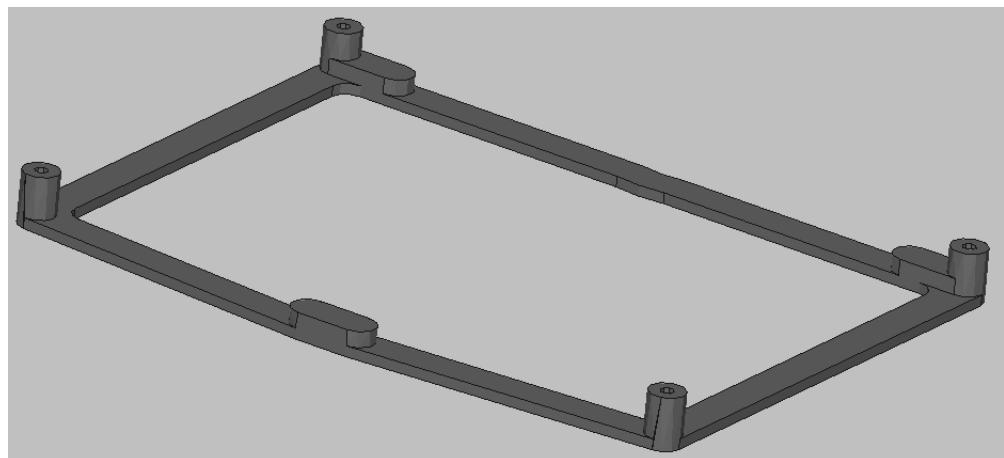


Abbildung 17: Konstruktionsbild der Controller-Halterung

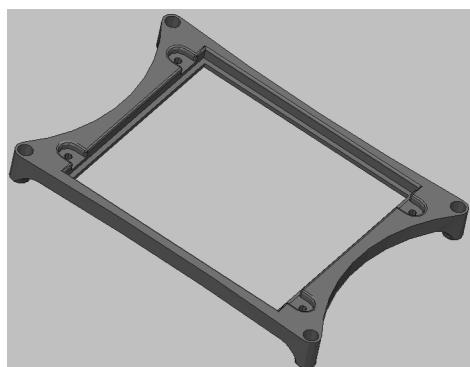


Abbildung 18: Konstruktionsbild der Platinen-Halterung des Bedienungsboards

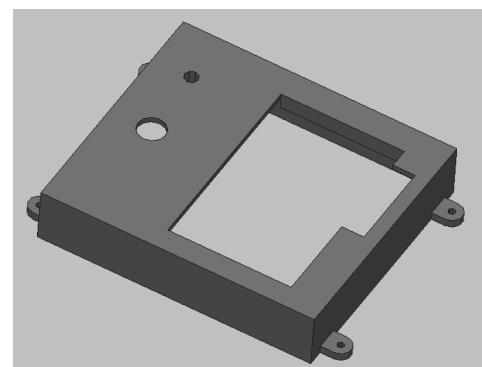


Abbildung 19: Konstruktionsbild der Abdeckung des Bedienungsboards

Zum Bedienen des Drucktasters 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.

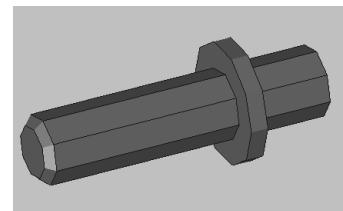


Abbildung 20: Konstruktionsbild des Drucktasters



Abbildung 21: Gedruckte Controller-Halterung



Abbildung 22: Gedruckte Drucktaster-Verlängerung

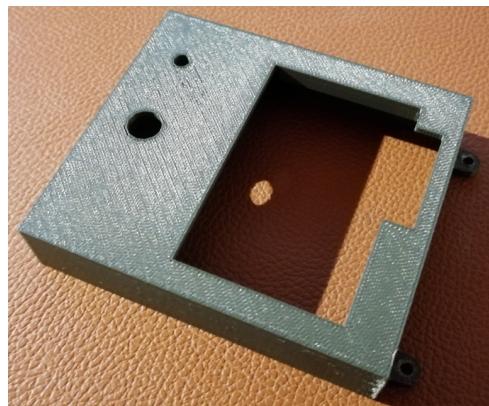


Abbildung 23: Gedruckte Platinen-Abdeckung des Bedienungsboards



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 mit roten Kreisen, die der Platinenhalterung mit blauen Kreisen und die der Abdeckung mit gelben Kreisen hervorgehoben.

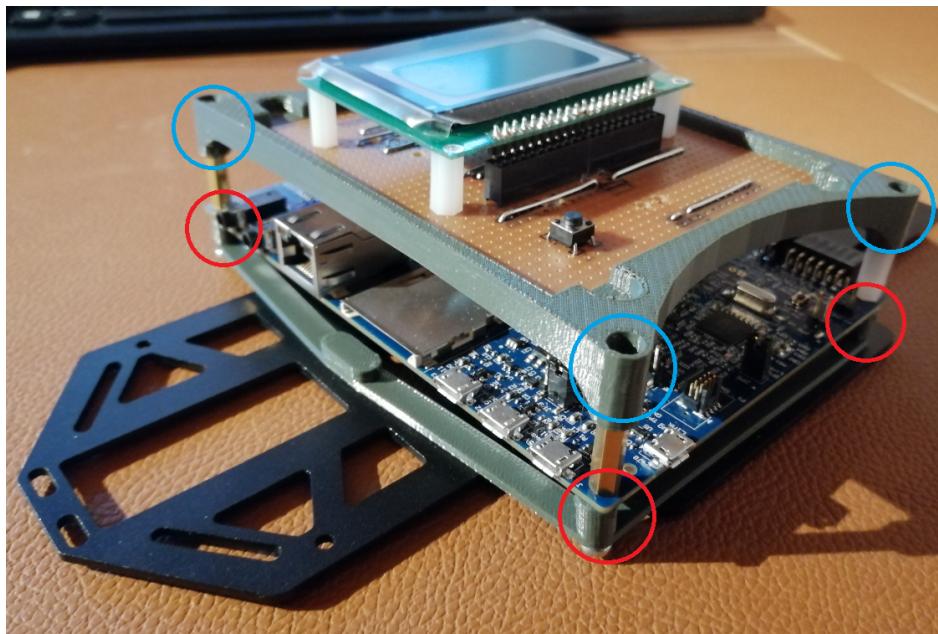


Abbildung 25: Fertig montierte Controller- und Bedienungsboard-Halterung; Befestigungsschrauben des Controllers in rot, Befestigung der Platinen-Halterung in blau

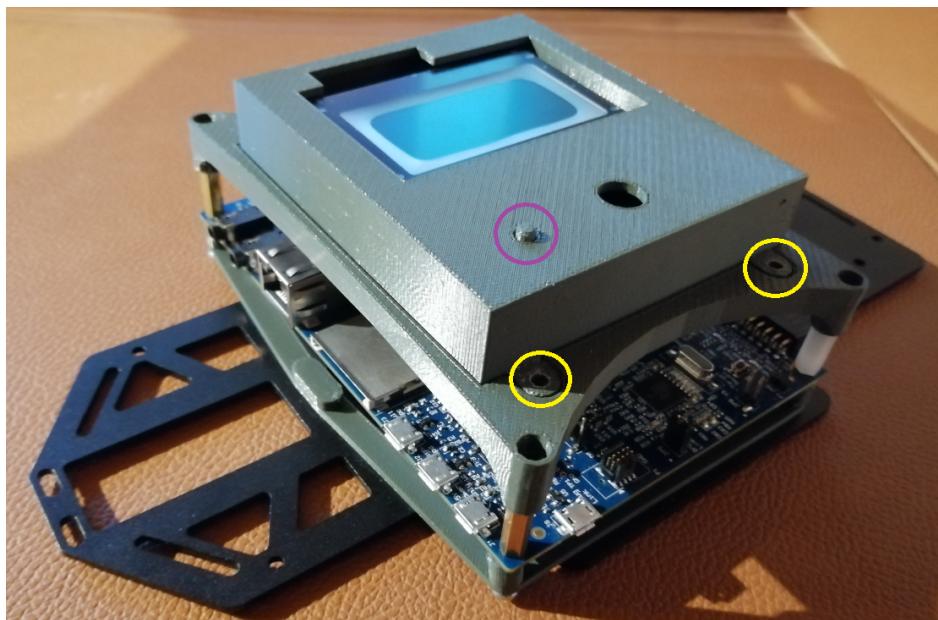


Abbildung 26: Fertig montierte Controller- und Bedienungsboard-Halterung mit Abdeckung; Befestigungsschrauben der Abdeckung in gelb, Taster in violett

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.

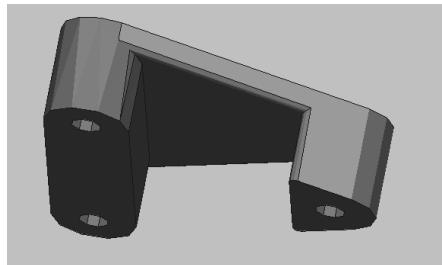


Abbildung 27: Konstruktionsbild einer Motorcontroller-Halterung



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.

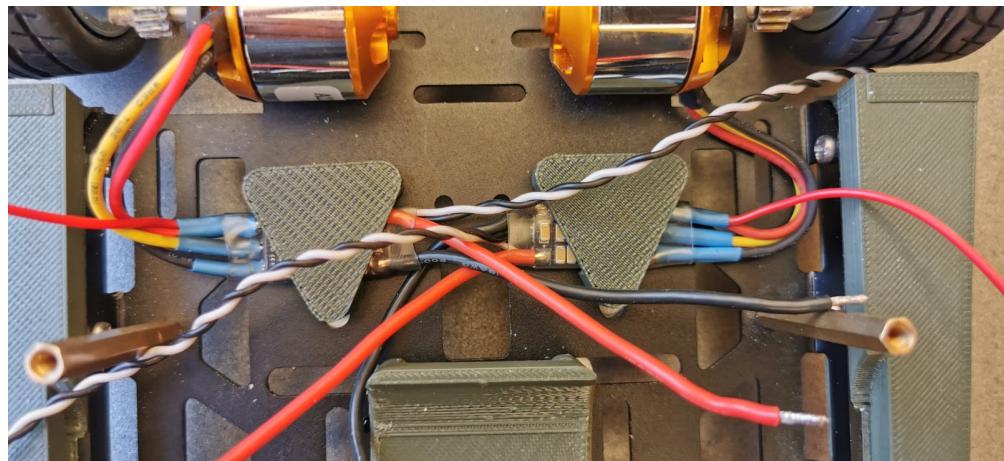


Abbildung 29: Die auf der Grundplatte montierten Motorcontroller

2.2.5 Einfache Platinenhalterungen

Für die Befestigung der Verteiler-Platine auf der unteren Ebene wird eine Halterung benötigt. Mit vier einfachen Halterungen (Konstruktionsbild siehe Abbildung 30) kann die Lochrasierplatine an vier Stellen mit etwas Abstand zur Grundplatte auf dieser befestigt werden. Die fertig gedruckten Platinen-Halterungen sind in Abbildung 31 abgebildet.

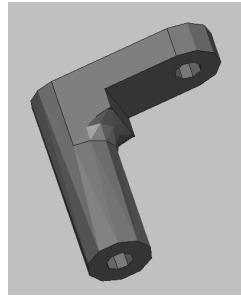


Abbildung 30: Konstruktionsbild einer einfachen Platinenhalterung



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.

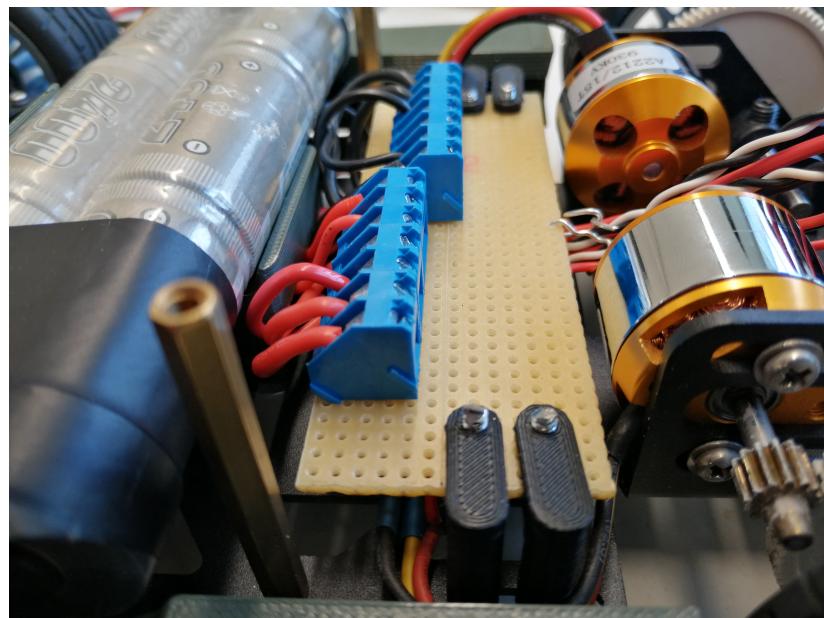


Abbildung 32: Mit den einfachen Platinenhalterungen montierte Verteilerplatine für die Versorgungsspannung

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.

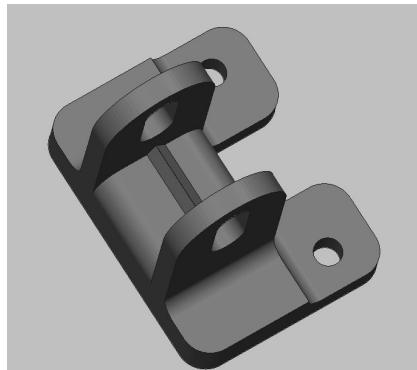


Abbildung 33: Kameramontage-Komponente der Kamera-Halterung

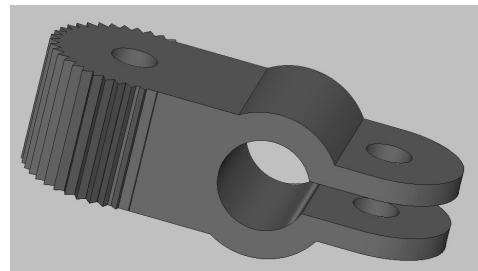


Abbildung 34: Stangenmontage-Komponente der Kamera-Halterung

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.

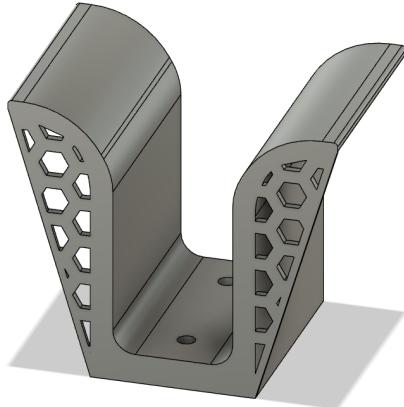


Abbildung 36: Konstruktionsbild der Antriebsabdeckung



Abbildung 37: Gedruckte Abdeckung der Antriebe

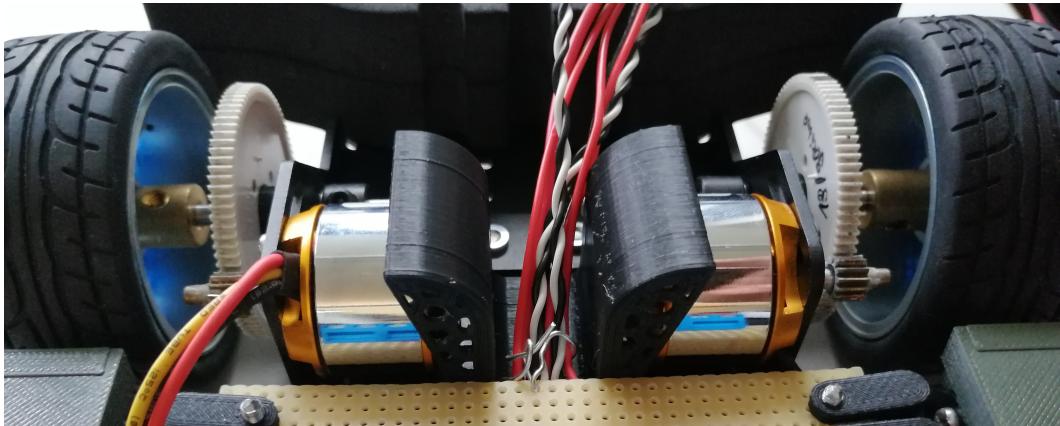


Abbildung 38: Fertig montierte Abdeckung der Antriebe zur Verhinderung von Reibung zwischen den Antrieben und den Kabeln, die nach oben durchgeführt werden

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.

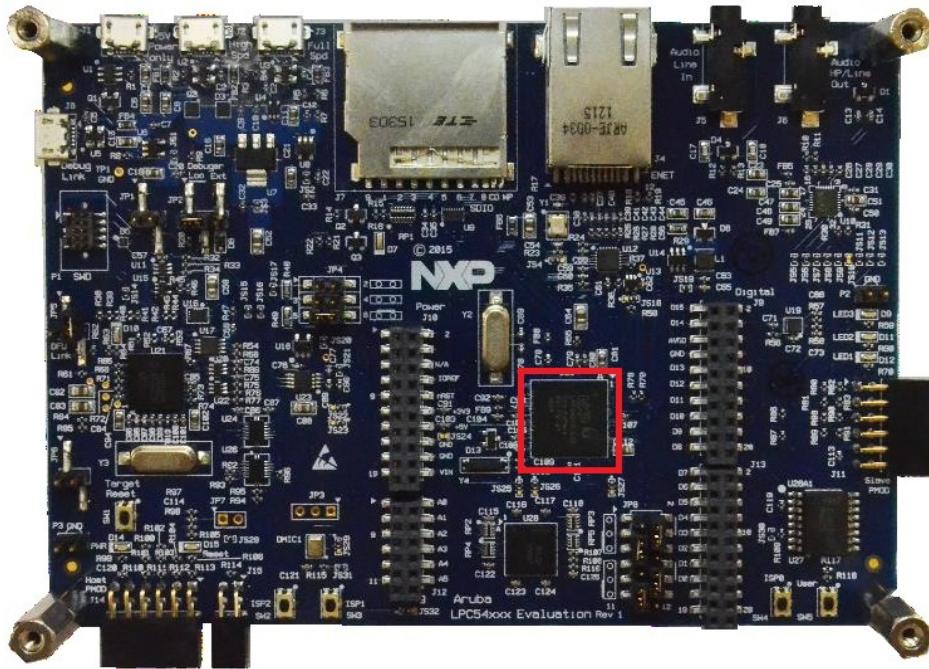


Abbildung 39: Draufsicht auf die Controllerplatine mit dem Mikrocontroller LPC54628 (in rot hervorgehoben) als Hauptbestandteil [Sem19]

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.

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.

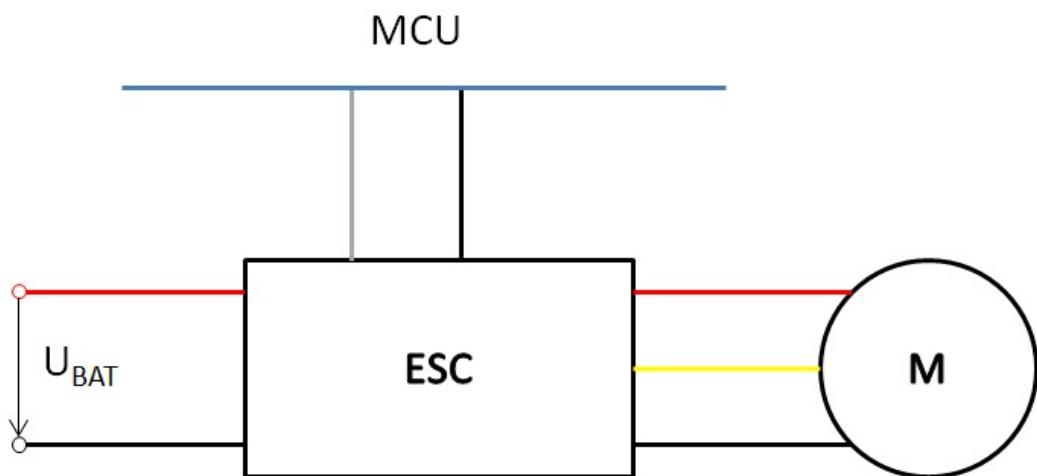


Abbildung 40: Skizze zur Beschaltung eines ESCs und BLDC-Motors; Versorgungsspannung des ESCs in rot und schwarz (Masse), PWM-Signalleitungen in grau und schwarz (Masse) und Anschlussleitungen des BLDC-Motors in rot, gelb und schwarz

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.

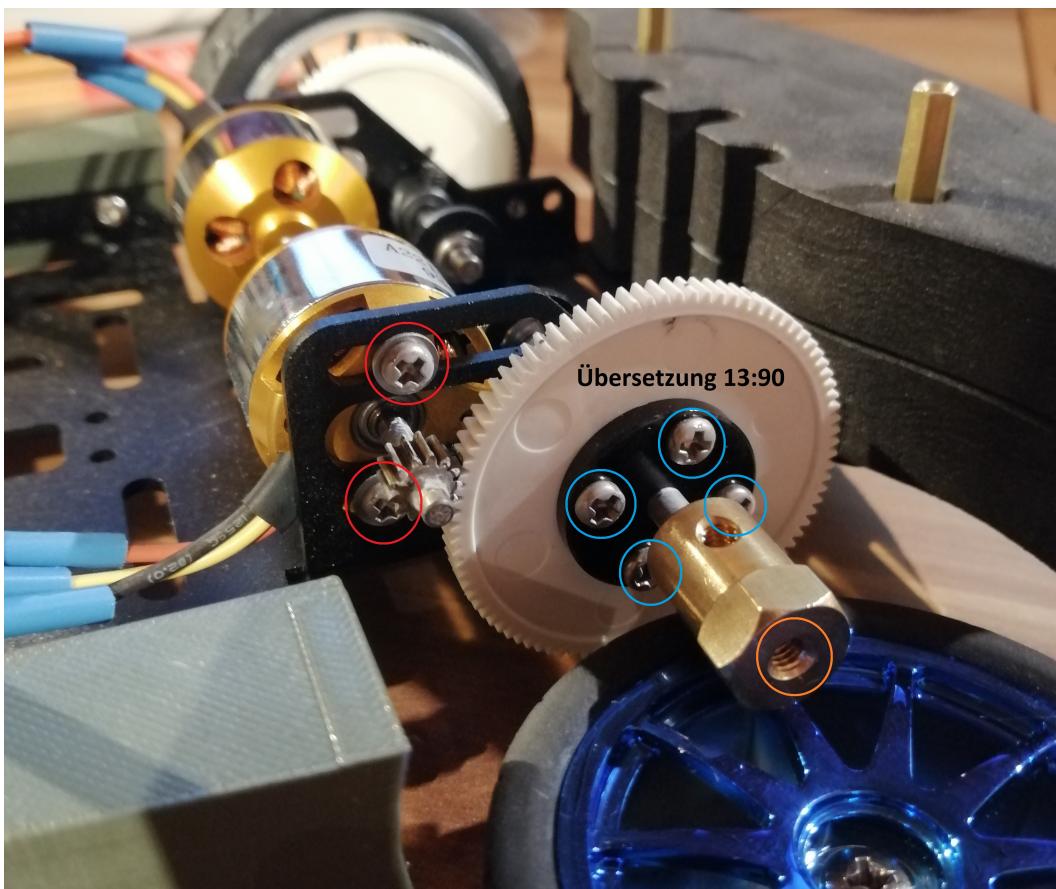


Abbildung 41: Montage der BLDC-Motoren und Übersetzung auf die Antriebsachse; In rot die Befestigung des BLDC-Motors, in blau die Befestigung des Zahnrads an der Antriebswelle und in orange die Befestigung des Reifens

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.

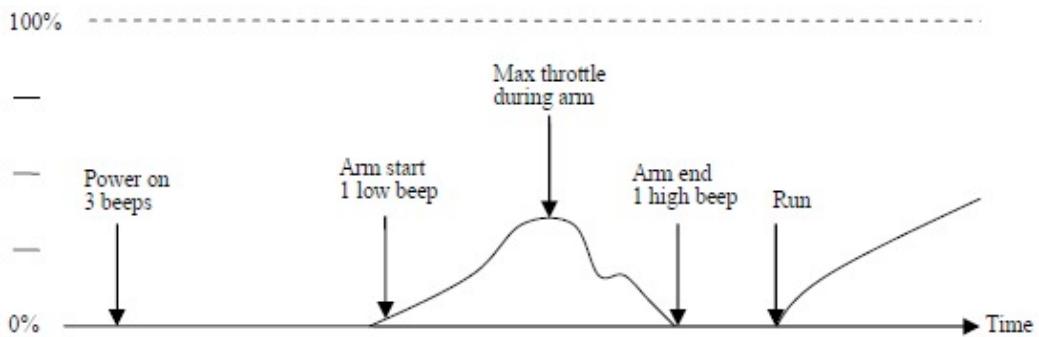


Abbildung 42: Initialisierungssequenz der ESCs; Wert an der Signalleitung für die Initialisierungssequenz über der Zeit; 0% entspricht dem PWM-Tastgrad für den Stillstand und 100% dem für die maximal erreichbare Drehzahl

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.

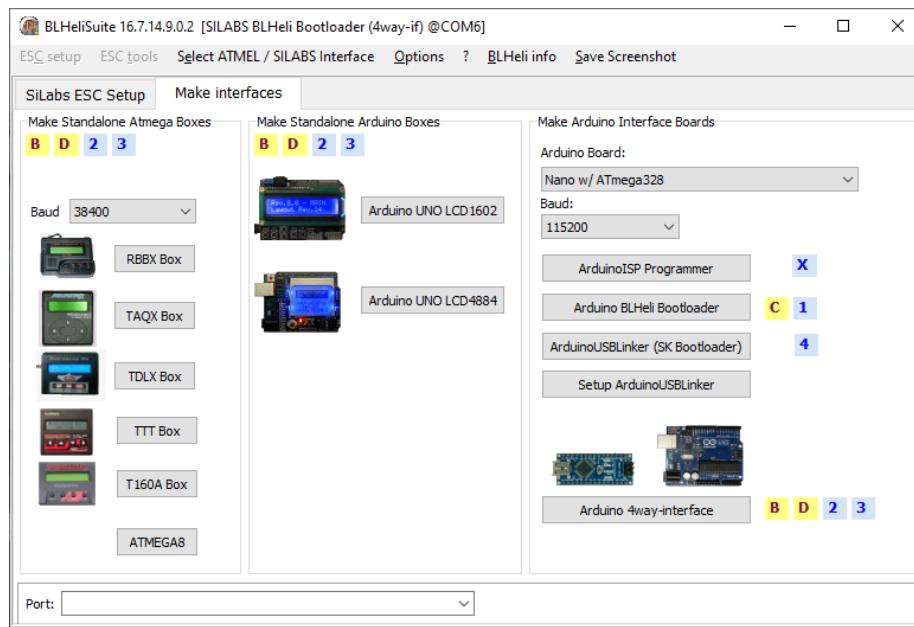


Abbildung 43: Programmierung des Arduino Nano zur ESC-Konfiguration mit der Software BLHeliSuite

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.

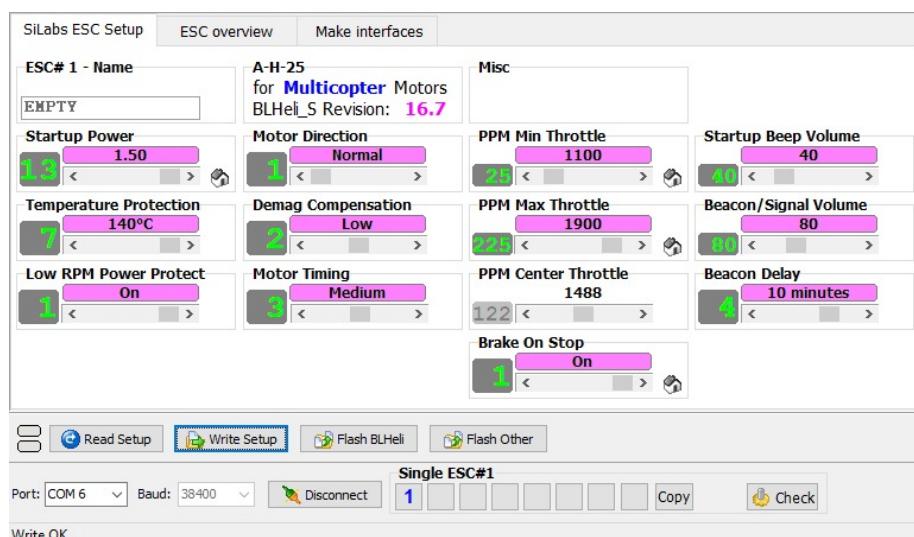


Abbildung 44: Konfiguration der ESCs mit der Software „BLHeliSuite“

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 (BLDC.MaxValue, 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 (BLDC.MaxValue, ca. 240.000 = 1,09ms). 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“

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, Channel). Die Channel werden auf das Timer Match Register 2 (rechter Antrieb, „kCTIMER_Match_2“) und auf das Timer Match Register 0 (linker Antrieb, „kC-TIMER_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.

```

/*
 * 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] &= 0xFFFFFFF0; //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] &= 0xFFFFFFF0; //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
    //***** End of BLDC Initialization *****

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

```

Abbildung 47: Funktionen BLDC _ Init und CTIMER3 _ Init der Datei „drive.c“

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).

```

/*
 * 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

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.

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 wurde, 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 wurden, 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.

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 wurde. Damit das Fahrzeug möglichst stabil auf dem Prüfstand 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 Fahrzeuges sind ebenfalls am Prüfstand fixiert (Abbildung 44 gelbe Markierungen).

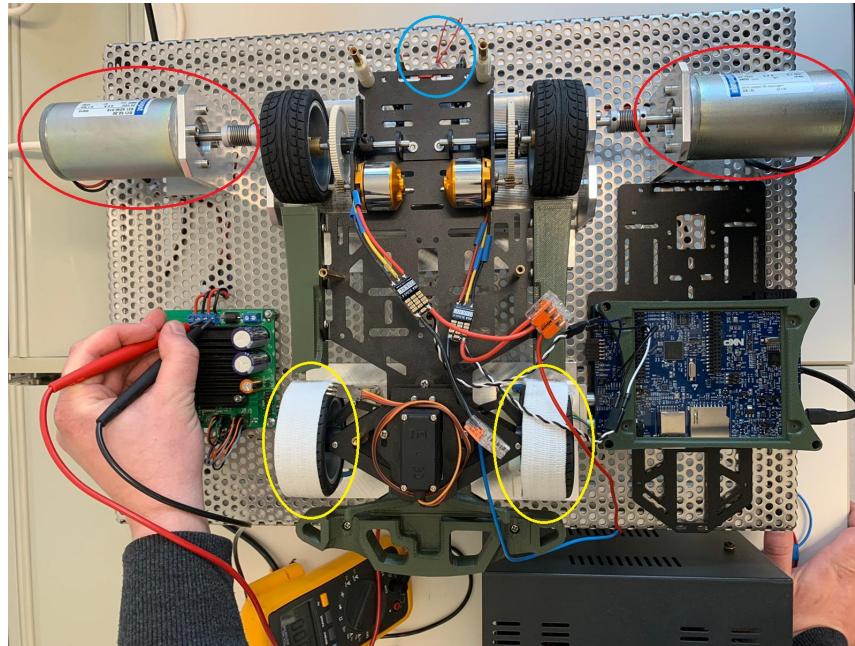


Abbildung 49: Messaufbau zur Prüfung der Spannungsabhängigkeit der BLDC-Motoren; in blau die Befestigung des Hecks an der Grundplatte des Prüfstands, in gelb die Befestigung der Vorderreifen am Prüfstand, in rot die DC-Motoren zur Spannungsmessung

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_{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.

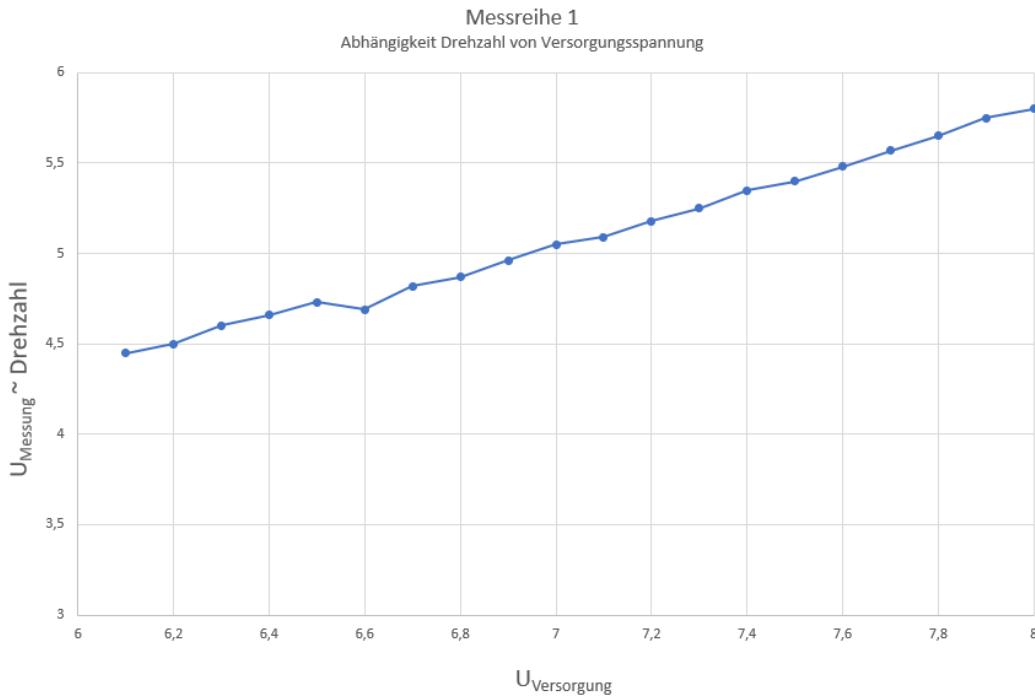


Abbildung 50: Spannungsabhängigkeit der Drehzahl der BLDC-Motoren von der Versorgungsspannung

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 HS-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 Fahrdauer (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 Zahnrädern) 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.

derlich, um ein zuverlässiges und sicheres Fahrverhalten zu gewährleisten.

4.5.2 Auswahl des Messprinzips

Nachdem bereits festgestellt werden konnte, 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).

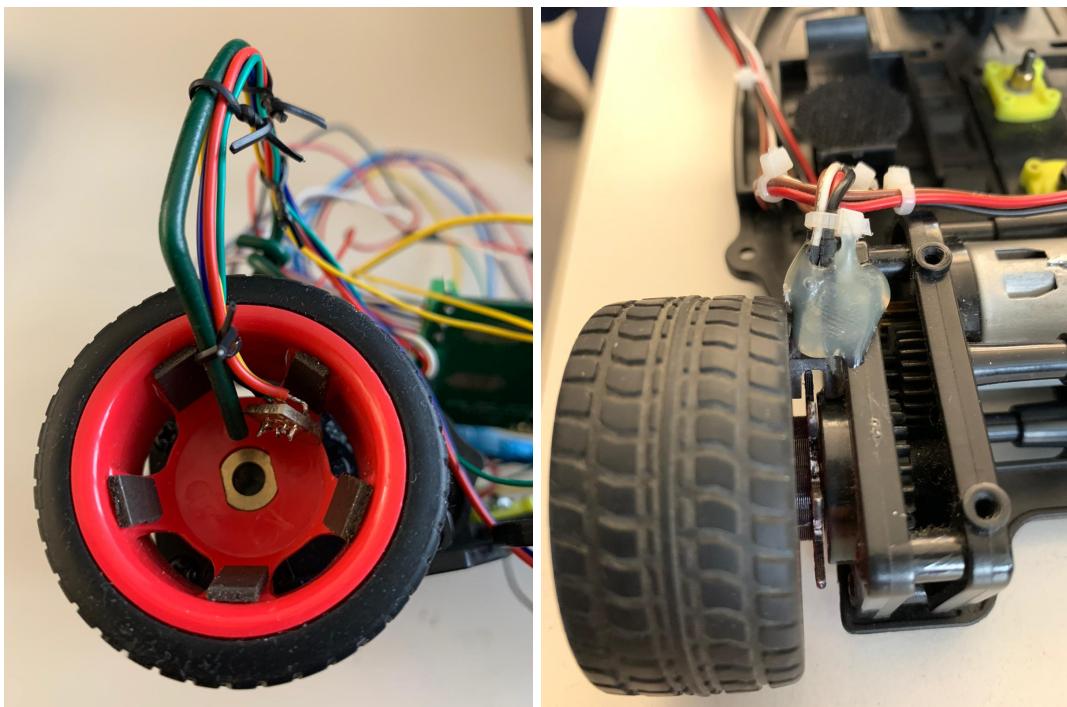


Abbildung 51: Varianten der Drehzahlerfassung bei vorherigen Fahrzeugversionen; links: Hallsensoren; rechts: Lichtschranke

Ein Hallsensor bietet den Vorteil, dass neben der Erfassung der Drehzahl auch die Drehrichtung mit Hilfe eines zweiten um 90° versetzten Hallsensors bestimmt werden kann. Dazu muss nicht einmal ein zweiter Sensor gekauft werden, da bereits zwei Sensoren mit 90° -Versetzung in einem Hallsensorgehäuse 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. Darüber hinaus ist die Entscheidung gegen die Verwendung eines Hallsensors darin begründet, dass die Drehzahlerfassung bei kleinen Drehzahlen, beispielsweise unter 50rpm sehr lange dauert, da durch den Abstand der Magnete viel Zeit zwischen zwei Signalen vergeht bis Winkel und Drehzahl ermittelt werden.

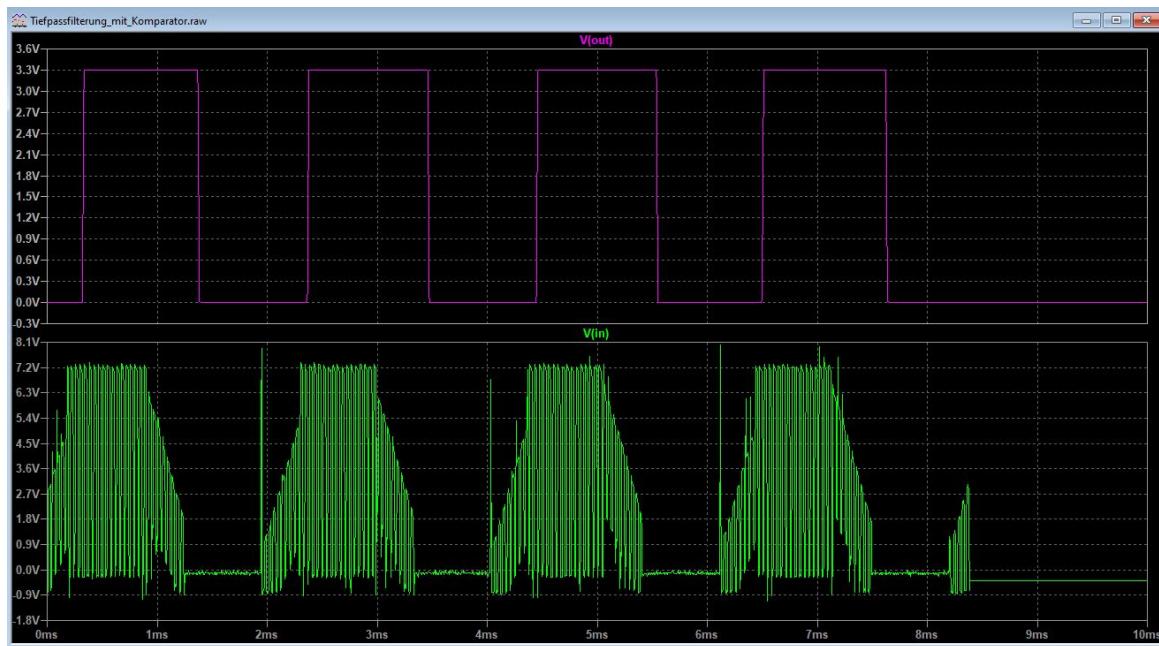


Abbildung 52: Phasensignal eines Antriebs ($V_{(in)}$, grün) bei schneller Fahrt und Darstellung des simulierten Ausgangssignals ($V_{(out)}$, pink) aus der Tiefpass-/Komparatorschaltung; erstellt und simuliert mit dem Schaltungssimulationstool LTspice XVII

Die Alternativmethode zur Erfassung der Drehzahl ist die Anbindung eines Tiefpassfilters mit nachgeschaltetem Komparator an eine der BLDC-Motor-Phasen je Antriebsseite. 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 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° 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.

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 61 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 Operationsverstärkerereingang IN^- nur einmal realisiert werden muss, worauf im weiteren noch detaillierter eingegangen wird.

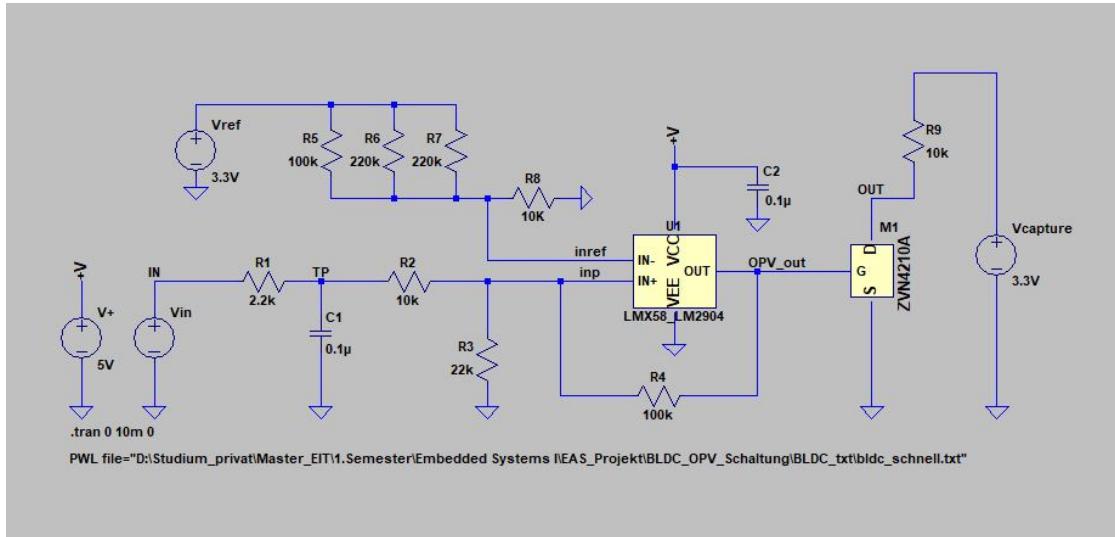


Abbildung 53: Schaltplan der Tiefpass-/Komparatorschaltung; erstellt für die Simulation eines Phasensignals in LTspice XVII

Das Phasensignal wird von der Spannungsquelle V_{in} repräsentiert. Die Signale wurden vor der Schaltungsentwicklung an einer Phase eines ESCs 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).

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 1,1kΩ Widerstand (R_1) und einem 100nF Kondensator (C_1). Dabei beträgt die Grenzfrequenz f_G des Tiefpasses nach Gleichung 2 ca. 1,45kHz. Der aus der Simulation resultierende Spannungsverlauf sowie der am Oszilloskop gemessene Spannungsverlauf nach der Tiefpassfilterung sind in den Abbildungen 54 und 55 dargestellt.

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

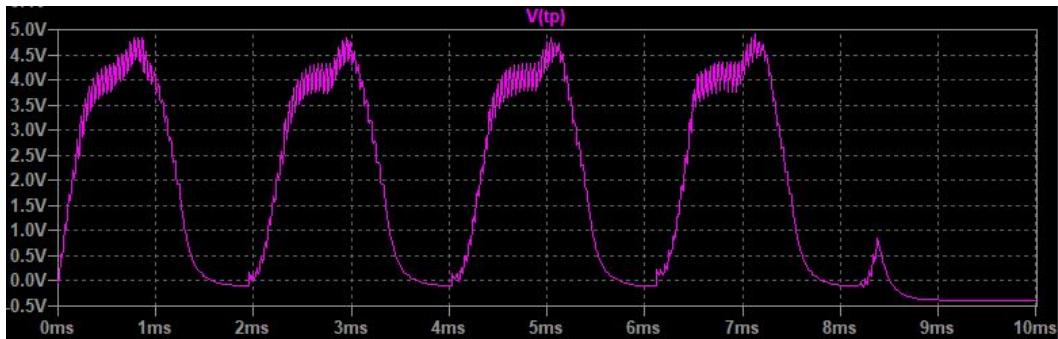


Abbildung 54: Simulationsergebnis des Phasensignals nach dessen Tiefpassfilterung bei schneller Fahrt

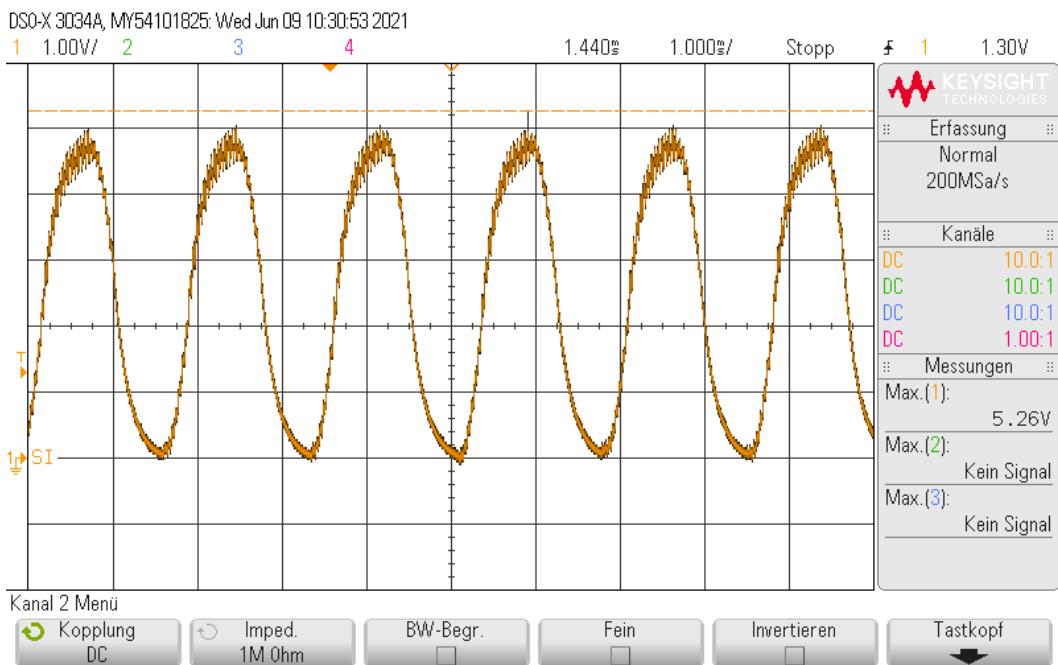


Abbildung 55: Messergebnis des Phasensignals nach dessen Tiefpassfilterung bei schneller Fahrt

Das zentrale Bauelement der Schaltung ist ein von Texas Instruments (TI) hergestellter Operationsverstärker LM358A, welcher als Komparator verwendet. Hierbei handelt sich um einen Standard (Dual) Operational Amplifier (op amp), welcher im Single-Supply betrieben wird. Der Operationsverstärker (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.

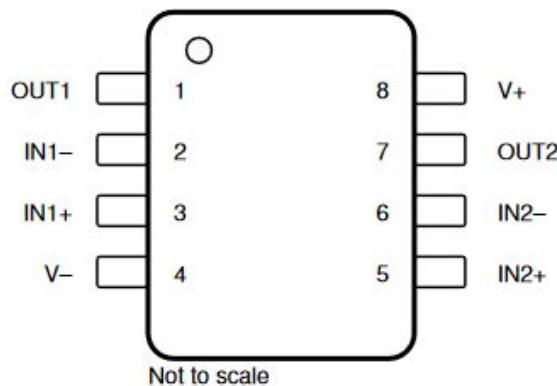


Abbildung 56: Pinbelegung des Operationsverstärkers LM358A

Da der OPV in der Schaltung als Komparator verwendet wird, müssen einige schaltungs-technische 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 $R_4 = 100\text{k}\Omega$. Die Kombination der Widerstände wurde durch Verwendung des Schaltungssimulationstools LTspice XVII und anhand der verfügbaren Widerstandsgrößen ermittelt.

Grundsätzlich vergleicht ein Komparator ständig seine Eingangsgrößen (IN^+ , IN^-) und zeigt digital an, welcher Eingang die größere Spannung besitzt. Da ein Operationsverstärker einen nahezu unendlichen Verstärkungsfaktor besitzt, reicht eine geringe Eingangsspannungs differenz, um den Ausgang in die Sättigung gehen zu lassen. Der hier verwendete OPV LM358A besitzt einen open-loop voltage gain von 100V/mV . Allerdings kann der Ausgang des OPV nur Werte im Bereich der Zustände V^+ und V^- annehmen. Versorgt man den Operationsverstärker mit einer Versorgungsspannung von $+3,3\text{ V}$ und legt V^- auf Masse, so erhält man im Idealfall am Ausgang ein TTL-Signal mit der maximalen Höhe der Versorgungsspannung zur digitalen Weiterverarbeitung. Sind beide Eingangsspannungen annähernd gleich, so kippt der Ausgang bei der kleinsten Störung oder Veränderung hin und her. Um dieses Problem zu vermeiden, wird eine Hysterese eingebaut. Das bedeutet, dass bei erreichen einer bestimmten Referenzspannung oder einem bestimmten Pegel am invertierenden Eingang eingeschaltet, und bei einem wiederholten erreichen der Referenzspannung wieder ausgeschaltet wird (siehe Abbildung 52 Schnittpunkte IN^+ (grün) und IN^- (blau)).

Wie bereits erwähnt kann der Ausgang des OPV im Idealfall seine Versorgungsspannung V^+ und V^- annehmen, da aber das verwendete OPV-Modell lediglich über eine Single Supply Funktion und nicht über eine Rail-to-Rail Funktion verfügt, bleibt die max. Ausgangsspannung stets unter der angeschlossenen Versorgungsspannung. Hierzu ist in Abbildung 52 der nach unten abweichende Spannungsverlauf OPV out (gelb) dargestellt, die Versorgungsspannung bei dieser Aufnahme beträgt $V^+ 5\text{V}$.

Da zu Beginn der Schaltungsentwicklung in der Simulation jedoch keine signifikanten Ab-

weichungen festgestellt werden, wird der OPV zunächst mit einer Spannung V+ von 3,3V versorgt. Erst eine Messung mit dem Ozilloskop macht ein deutlich abweichendes Verhalten im Vergleich zur Simulation deutlich. Es werden am Ausgang des OPV bei FULLTHROTTLE lediglich 2,08V und bei „STOPTHROTTLE“ nur ca. 0,1V gemessen. Dies führt dazu, dass insbesondere im niedrigen Drehzahlbereich, sprich ab einer Unterschreitung der Referenzspannung Vref von 540 mV kein PWM-Signal mehr erzeugt wird. Darüber hinaus ist im zugehörige Datenblatt des verwendeten Mikrocontrollers ersichtlich, dass die Timer2 Capture Input Pins(P0.24 Antrieb rechts, P0.25 Antrieb links) eine Mindestspannung von 2,0V benötigten um zuverlässig Signale verarbeiten zu können. Da dieser Spannungswert insbesondere im niedrigen Drehzahlbereichs nicht erreicht wird, ist die Schaltung in dieser Form für den Anwendungszweck nicht ausreichend.

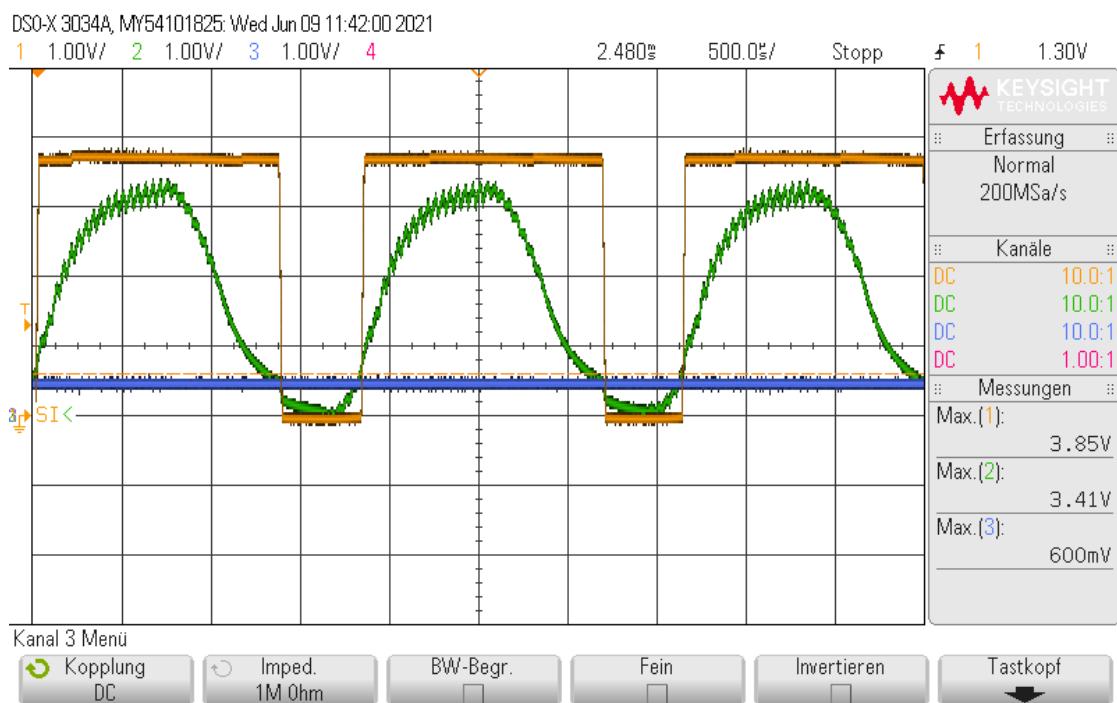


Abbildung 57: Beispielhafte Messung des OPV Ausgangssignals OPV out (gelb) bei einer Versorgungsspannung von V+ 5,0V Single Supply, IN+ (grün), Vref am IN- (blau)

Als erste Lösung der Problematik wird versucht die Spannungsversorgung des OPV auf V+ 5V zu erhöhen. Durch die Erhöhung der Versorgungsspannung kann grundsätzlich eine höhere Ausgangsspannung erzielt werden, dennoch gilt dies weiterhin nur für größere Drehzahlbereiche. Eine Spannung größer 2,0 V, wird vor allem im Bereich der niedrigeren Drehzahlen nach wie vor nicht zuverlässig erreicht.

Damit eine zuverlässige Bereitstellung einer konstanten Signalspannung für die Timer2 Capture Input Pins am Mikrocontroller größer als 2,0 V erreicht werden kann, wird ein zusätzlicher n-Kanal Mosfet ZVN4210A am Ausgang des OPV implementiert. Der in Abbildung 49 dargestellte Widerstand R9 (10 kOhm) vor dem Drain-Anschluss des Mosfets, stellt den

intern einstellbaren Pull-up Widerstand der Timer2 Capture Input Pins des Mikrocontrollers dar. Je nach Konfiguration können diese zu- abgeschalten werden. Im Zuge der Entwicklung wird der Pull-up Widerstand R9 auf einem Whiteboard angebracht und entsprechend verdrahtet.

Zur Bereitstellung der am invertierenden OPV-Eingang (IN-) benötigten Referenzspannung, auch Schwellspannung genannt, wird ein zusätzlicher Spannungsteiler mit einer vom PIN X.Y. kommenden +3,3V Spannungsversorgung vorgeschalten. Der Spannungsteiler besteht aus drei parallel geschalteten Widerständen R5 (100 kOhm), R6 und R7 (220 kOhm) in Reihe zum Widerstand R8 (10 kOhm). Die dadurch bereitgestellte Spannung beträgt in etwa 540mV. Wie bereits erwähnt, ermöglicht die Referenzspannung (Vinref) zusammen mit dem Eingangssignals am nicht-invertierenden Eingang (Vinp), dass durchschalten von V+ am OPV Ausgang. Die bereitgestellte Referenzspannung von 540mV ist in der Theorie ausreichend um für alle Drehzahlen ein zuverlässiges PWM-Signal zu erzeugen. In der Praxis treten dennoch wiederholt Probleme auf. Die Problematik sowie deren Lösung werden im nachfolgenden Abschnitt 4.5.4 „Ausblick Schaltungserweiterung“ beschrieben.

In Abbildung 53 links, ist die auf der Verteilerplatine befindliche Drehzahlmessschaltung im Original zu sehen. Auf der rechten Seite von Abbildung 53 befindet sich der mit dem Programm BlackBoard Circuit Designer erstellte zugehörige Stromverlaufsplana, welcher eine schnelle Übersicht ermöglicht. Die grünen Verbindungen stellen die Pfade zwischen Vin (Eingangssignal von den ESCs) bis zum OPV Eingang IN+ dar, Vref (gelb) Pfad der Referenzspannung zu den invertierenden Eingängen IN1-, IN2- des OPV. Der Pfad zur Spannungsversorgung des OPV (V+) ist mit violett dargestellt und alle schwarzen Verbindungen stellen GND-Verbindungen dar. Zuletzt kann der Verlauf der Ausgangsspannung des OPV out anhand der roten Verbindungen nachvollzogen werden.

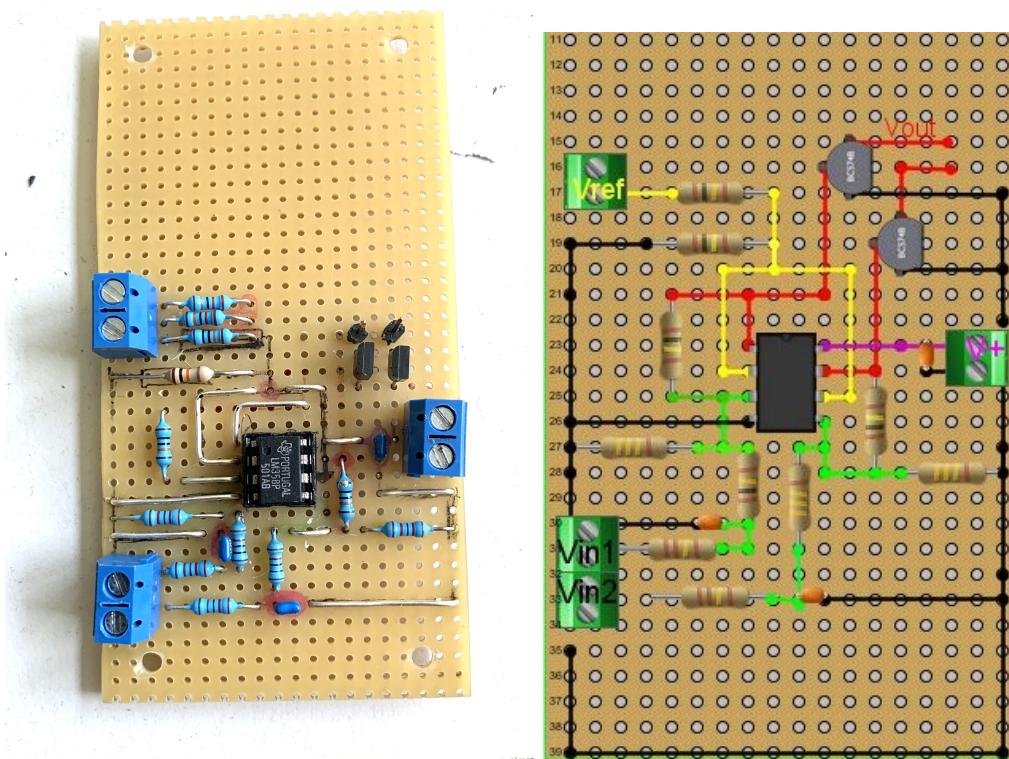


Abbildung 58: Schaltung Drehzahlmessung auf Verteilerplatine (links), Übersicht Platinenaufbau mit markierten Stromverlaufspfaden

4.5.4 Ausblick Schaltungserweiterung

Nachdem die Referenzspannung V_{ref} in der Schaltung so dimensioniert ist, sodass in der Theorie alle Drehzahlen zuverlässig als eindeutiges PWM-Signal dargestellt werden können, treten in Praxis noch Probleme auf. Denn nach Messung der Spannung mit dem Oszilloskop am PIN X.Y, wurden anstelle der exakten +3,3V, Spannungen bis zu 3,65V gemessen. Dies führt zu einer schwankenden Referenzspannung von etwa 540mV bis 600mV. Das Problem, Drehzahlen im sehr niedrigen Drehzahlbereich insbesondere im „STOPTHROTTLE“, 1,09 ms bis 1,10ms ton-Zeit, werden nicht immer zuverlässig als eindeutiges PWM-Signal dargestellt, da die Referenzspannung keine eindeutigen Schnittpunkte mit dem Eingangssignal besitzt (siehe Abbildung 54).

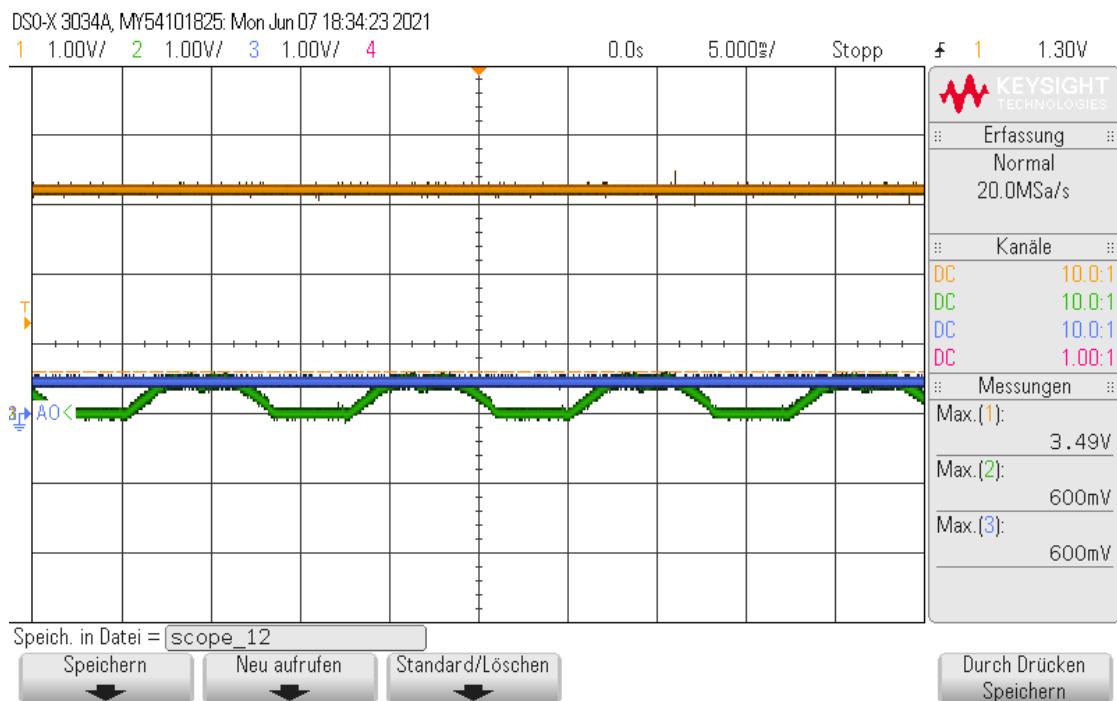


Abbildung 59: Beispielhafte Messung bei nicht übereinstimmenden Schaltschwellen für das Eingangssignal - Folge: kein Schalten des OPV möglich

Eine mögliche Lösung der Problematik, ist die Erweiterung des Spannungsteiler zur Einstellung der Referenzspannung vor dem invertierenden Eingang des OPV um einen n-Kanal Mosfet gegen Masse (siehe Abbildung 55). Diese Maßnahme ermöglicht es, die Referenzspannung, für hohe und niedrige Drehzahlen flexibel anzupassen um ein zuverlässiges Ausgangssignal zu erhalten.

Durch die Vorgabe eines bestimmten Modus, in der Schaltung als MCU-Mode bezeichnet, kann durch die Definierung von zwei Drehzahlbereichen, beispielsweise ton-Zeit 1,09ms bis 1,43ms für langsame Drehzahlen eine Referenzspannung zwischen 400mV-500mV eingestellt werden. Der Vorteil ist, dass man dadurch einen bestimmten Toleranzbereich für Schwankungen der Versorgungsspannung vom Pin erzeugt und die Schaltung dadurch weniger anfällig für Abweichungen ist.

Softwareseitig kann hierzu eine Interrupt Service Routine programmiert werden, welche bei erkennen des niedrigen Drehzahlbereichs einen weiteren Pin des Mikrocontrollers so schaltet, sodass dieser die Gate Spannung des Mosfets liefert und den Widerstand R7 folglich gegen Masse schaltet. Durch den zusätzlichen parallelen Widerstand kann die Referenzspannung verringert werden. Zur Bestimmung der Widerstandswerte für R5, R6 und R7 können folgende Formeln (3), (4) verwendet werden.

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

$$V_{ModeHigh} = \frac{R2||R3 + R_{DS0N}}{R2||R1 + R_{DS0N} + R5} \cdot 3,3V \quad (4)$$

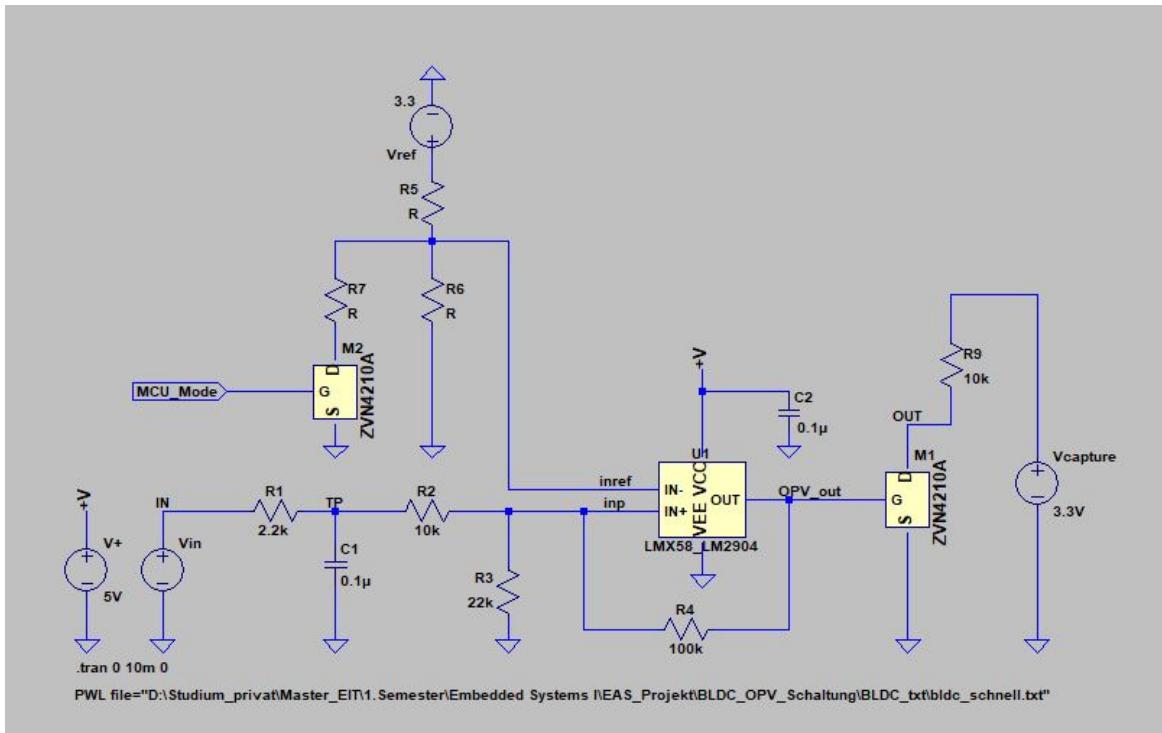


Abbildung 60: Schaltungsaufbau mit einstellbarer Referenzspannung für die invertierenden Eingänge IN1-, IN2-

4.5.5 Programmierung des Drehzahlmessungsbausteins

Damit das durch die Schaltung bereitgestellte PWM-Signal vom Mikrocontroller softwareseitig erfasst werden kann, wird grundsätzlich ein Timer Capture Register benötigt. Dieses Register speichert bei einem Ereignis, bspw. das Auftreten einer steigenden Flanke, am angeschlossenen Input Pin den aktuellen Timerwert und kann daraufhin einen Interrupt auslösen.

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“. Hier werden die beiden Channel zur Erfassung der Daten auf das Timer Caputre Register 0 (rechter Antrieb, „kCTIMER_Caputre_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 Pin 4 und der Pin P0.24 über den Pin 6 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 61: Relevante Zeilen der Datei „rpmMeas.h“ zur Initialisierung des Timers 2 für das PWM Signal sowie Prototypen

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 57).

In der Funktion RPMMEAS_Init wird zuerst die Funktion CTIMER2_Init aufgerufen, welche unter anderem die beiden vorher festgelegten Kanäle des Timers C2 („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 der Trigger der beiden Kanäle auf die steigende Flanke gesetzt („kCTIMER_Capture_RiseEdge“) und im Anschluss der Enable für die Capture Interrupts auf 1 gesetzt.

Damit beim Auftreten einer steigenden Flanke des PWM-Signals, ein Interrupt Flag ausgelöst werden kann, benötigt man bei MCUXpresso zunächst einen cTimer_callback_table. Der Callback-Table wird als Parameter initialisiert und muss mit den verwendeten Interrupt Service Routinen rpmMeas_Left_ISR und rpmMeas_Right_ISR verknüpft werden. Der Callback-Table wird als Pointer-Array in der CTIMER2_Init der Funktion CTIMER_RegisterCallBack übergeben. Bei einer Callback-Funktion auch Rückruffunktion genannt, wird einer Funktion als Parameter eine andere Funktion 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 Channel ein Callback durchgeführt werden kann. Zuletzt wird in der CTIMER2_Init der Timer C2 gestartet.

Nach Vollendung der Initialisierungsfunktionen des CTIMER2_Init werden in der Funktion RPMMEAS_Init zusätzlich die Pins P0.24 (Antrieb links) und P0.25 (Antrieb rechts) für die Verwendung als PWM-Input Capture Pin des Timers C2 konfiguriert.

```

#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[10] = {0,0,0,0,0,0,0,0,0,0}; /*array for average value determination left BLDC*/
uint8_t ileft = 0;                            /*increment array number left side*/
uint32_t measright[10];/*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]     &= 0xFFFFFFF0;      //clear FUNC bits of P0.24
    IOCON->PIO[0][24]     |= 0x3;            //set FUNC bits to CTIMER2_CAP0 function FUNC3 P0.24
    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]     &= 0xFFFFFFF0;      //clear FUNC bits of P0.25
    IOCON->PIO[0][25]     |= 0x3;            //set FUNC bits to CTIMER2_CAP1 function FUNC3 P0.25
    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 62: Funktionen RPMMEAS_Init und CTIMER2_Init in der Datei Relevante Zeilen der Datei „rpmMeas.c“

Zur Erfassung der Drehzahl wird im Programm jeweils eine Interrupt Service Routine rpmMeas_Left_ISR und rpmMeas_Right_ISR für die jeweilige Antriebsseite benötigt. Sobald eine steigende Flanke getriggert wird, wird durch das setzen eines Interrupt Flags die ISR ausgelöst und die vom Timer C2 erfasste Zeit zwischen zwei steigenden Flanken mit dem Befehl CTIMER_GetTimerCountValue an die ISR übergeben. Die aktuell gemessene Zeit wird mit der vorherig gemessenen Zeit (oldvalright, oldvalleft) subtrahiert und die resultierende Differenz (diffright oder diffleft) 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 63: Interrupt Service Routinen zur Erfassung der Capture Daten

Die Werte des in den Arrays („measright[]“ und „measleft[]“) abgelegten Werte, werden daraufhin der Funktion „rpmMeas_average_val_right()“, „rpmMeas_average_val_left()“ übergeben. Dabei wird aus zehn erfassten Werten der Mittelwert gebildet. Der berechnete Mittelwert „averageval_right“, „averageval_left“ wird in den weiteren Schritten zur Bestimmung 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 64: Funktionen zur Berechnung des Mittelwerts aus den erfassten Capture Daten

Im letzten Abschnitt der RPMMEAS_Init werden an die zwei Funktionen „rpmMeas_average_val_left“

und „rpmMeas_average_val_right“ die Arrays mit den erfassten Drehzahldaten übergeben. Aus den erfassten Daten kann daraus der Mittelwert „averageval_left“ bzw. „averageval_right“ gebildet werden und im weiteren einer entsprechenden Funktion zur Drehzahlregelung weitergegeben werden.

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.

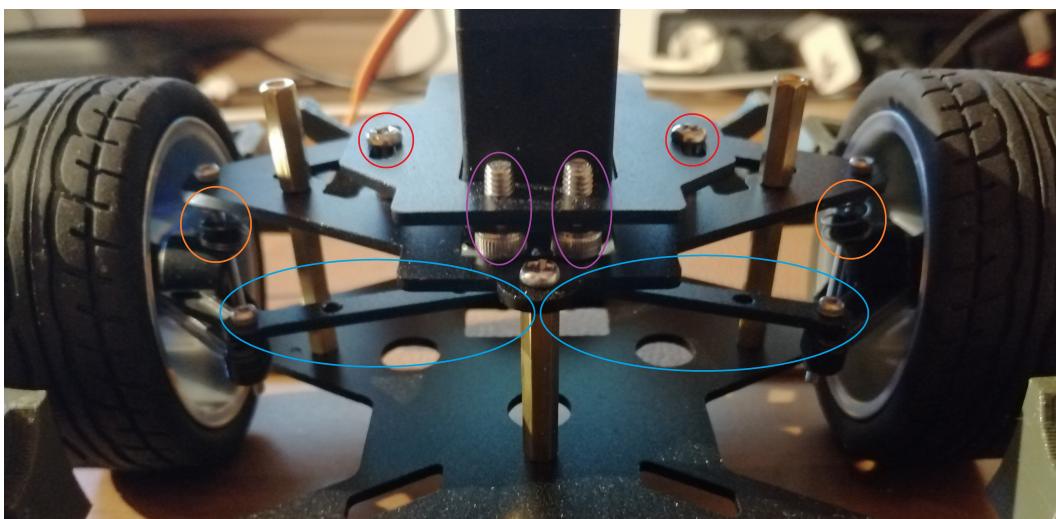


Abbildung 66: Montage des Servo-Motors, des Lenkgestänges und der Vorderreifen; Befestigung des Servo-Motors in violett, Befestigungsschrauben des losen Chassis-Teils in rot, Lenkgestänge in blau und Federung in orange

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 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
}

/*
 * 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 sollen. 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 C 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 Linkslenken, 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 Bedienungsboard

Das Bedienungsboard ist auf der oberen Fahrzeugebene über dem Controller verbaut. Es enthält ein Display, einen Drehencoder mit Button, einen separaten Button 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 Bedienungsboard ist eine selbst bestückte Lochrasterplatine, die die Anschlüsse der einzelnen Komponenten auf eine Stiftleiste zusammenführt. In den Abbildungen 71 und 72 ist der Schaltplan des Bedienungsboards 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 Stiftleisten J1 und J2 geführt, über die die Verbindung zur Controllerplatine hergestellt wird. Die Einbindung des Bedienungsboards in das Gesamtsystem des Fahrzeugs ist über den Anhang „Anhang 1: Schaltplan“ nachvollziehbar.

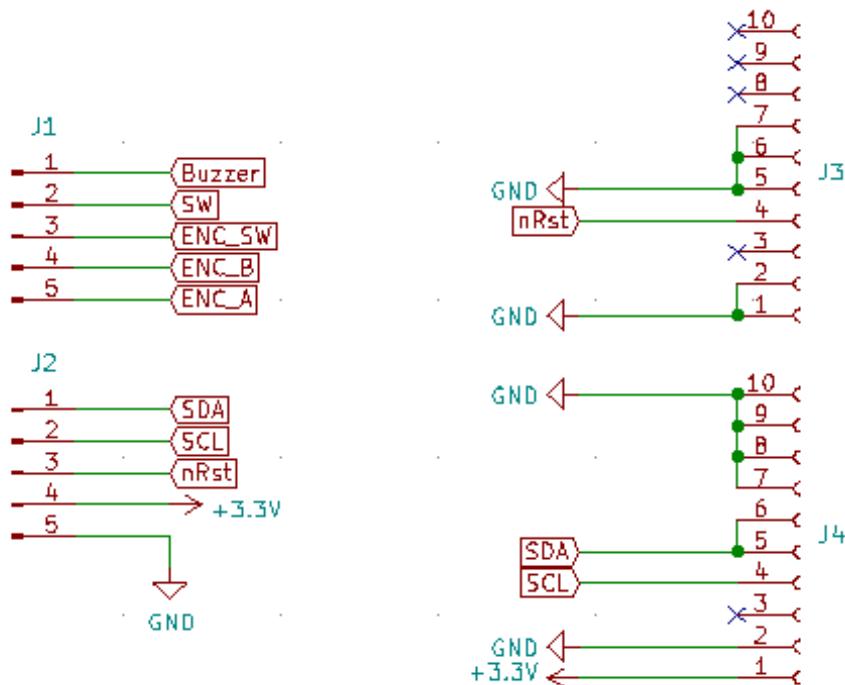


Abbildung 71: Schaltplan des Bedienungsboards mit den Stiftleisten J1 und J2 zur Verbindung mit dem Mikrocontroller und den Buchsenleisten J3 und J4 für das Display

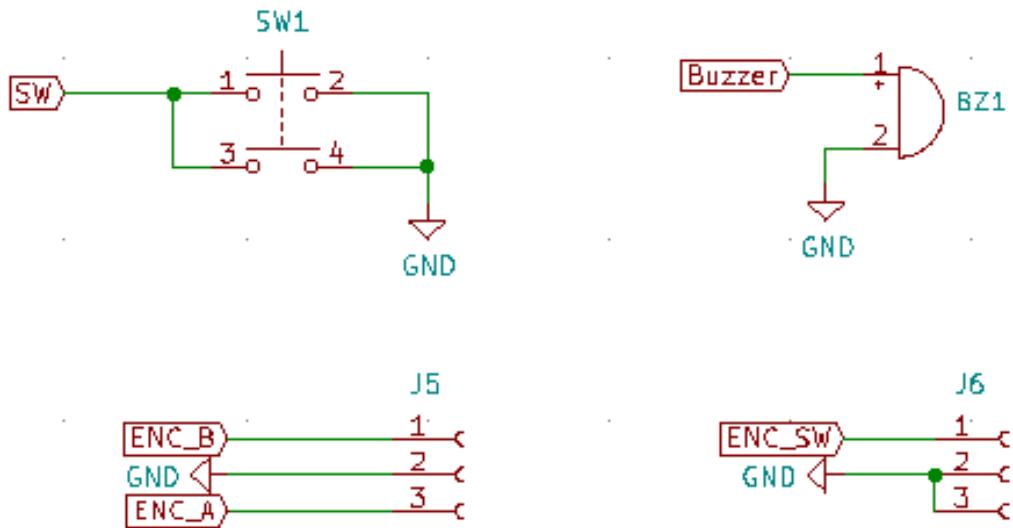


Abbildung 72: Schaltplan des Bedienungsboards mit dem Drehencoder, dem Taster und dem Summer

6.2 Programmierung der Steuerelemente

Die drei Signale des Drehencoders ENC_A, ENC_B und ENC_SW und das Signal SW des Tasters fungieren als Pulldown Anschlüsse. Diese Signale werden im Mikrocontroller über einen internen Pull-Widerstand auf 3,3 V gezogen. Die beiden Signale ENC_A und ENC_B liefern ein Rechtecksignal, dessen Frequenz sich mit der Drehgeschwindigkeit ändert. Durch einen Phasenversatz der beiden Signale 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, kann die Drehrichtung und die Anzahl der Schritte bestimmt werden.

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.

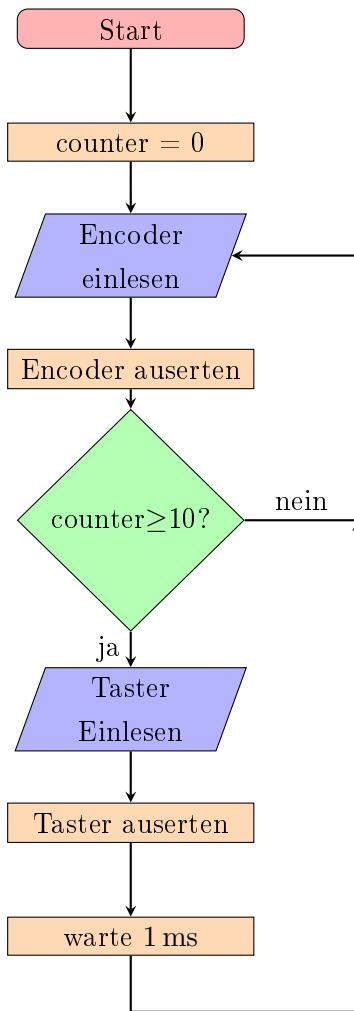


Abbildung 73: Flussablauf zum Einlesen des Encoders jede Millisekunde und der Taster alle zehn Millisekunden

6.2.1 Taster auswerten

Im Flussablaufdiagramm in Abbildung 73 werden die beiden Taster mit einer Abtastzeit von 10 ms eingelesen. Da Taster mechanische Bauteile sind kommt es bei jedem Betätigen zu ungewollten Pegelwechseln, dem sogenannten Prellen. Dies muss verhindert werden, da dadurch ein Tastendruck als mehrere interpretiert werden kann. Soll zum Beispiel ein Zähler per Tastendruck um eins inkrementiert werden, wird bei einem prellendem Taster der Zähler pro Druck nicht um eins erhöht sondern um zum Beispiel fünf. Die Anzahl um wie viel der Zähler erhöht wird hängt davon ab, wie viele Pegelwechsel durch das Prellen erzeugt wird und mit welcher Frequenz der Taster ausgelesen wird. Um das Prellen zu verhindern können spezielle prellfreie Taster verwendet werden, da diese recht teuer sind und der Taster im Drehencoder nicht ausgetauscht werden kann, wird auf eine softwareseitige 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 wichtig, dass der Taster mindestens die vierfache Zeit der Abtastzeit

gedrückt 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 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.

Im C-Code wird der Zähler mit zwei 32 Bit Variablen implementiert. Dabei wird aus jeweils einem Bit der beiden Variablen ein zwei Bit 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 ...).

6.2.2 Drehencoder auswerten

Im Flussablaufdiagramm in Abbildung 73 werden die beiden um 90° Phasenverschobenen Signale des Drehencoders mit einem Zeitabstand von 1 ms ausgelesen. Im Anschluss darauf 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 Drehencoder Anschlüsse 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 vier Bit große Zahl bildet den Index eines Array, das Anzeigt, ob der Drehencoder nach links oder nach rechts gedreht wird. Steht in der Variable zum Beispiel der Wert 1101_2 (13_{10}) bedeutet das, dass die vorherigen Zustände beide eins waren, der aktuelle Zustand von ENC_A LOW und der von ENC_B HIGH ist. 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 darstellt. 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.

6.3 Programmierung der Anzeige

Für die Anzeige von Fahrzeugparametern wird ein 64 mal 128 Pixel großes OLED-Display verwendet, dessen Ansteuerung über eine Inter Integrated-Circuit (I2C) Schnittstelle erfolgt. Mit dem Pin 6 von J3 wird die I2C Adresse auf $3C_{16}$ (GND) oder $3E_{16}$ (3,3 V) festgelegt. Dem Schaltplan in Abbildung 71 kann entnommen werden, dass die Adresse auf $3C_{16}$ festgelegt ist. Außerdem wird das Reset Signal des Controllers an den Pin 4 der Stifteleiste J3 angelegt.

6.3.1 Funktionsweise des organisches 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 an. Das Display besitzt 64 COM-Anschlüsse für die Kathoden der LEDs und 128 Segmentanschlüsse für die Anoden der LEDs. Jeder COM-Anschluss entspricht einer Zeile des Displays und jeder Segmentanschluss einer Spalte (siehe Schaltplan in Abbildung 74). Somit 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 (COM-Anschluss) des Displays und jeder Spalte des Speichers ist 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 zusammengefassst. Bit null eines Wertes entspricht der ersten Zeile der Seite und Bit sieben der achten Zeile. 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. Diese wird bis zum letzten Segment fortgeführt. Ist dies erreicht wird wieder das Segment SEG0 ausgewählt. Diesmal werden die Pixelwerte jedoch an die Anschlüsse COM8 bis COM15 angelegt. Nach dem das letzte Segment und die letzten acht COM-Anschlüsse angesteuert wurden, beginnt der Durchlauf von vorn. In den Einstellungen des Displaycontroller kann eingestellt werden von welcher Startseite bis zu welcher Stopseite die COM-Anschlüsse angesteuert werden. Dadurch können Seiten am oberen und unteren Rand ausgelassen und somit der Displaybereich eingegrenzt werden. Das Start- und Stoppsegment kann ebenfalls bestimmt werden und somit der rechte und linke Displayrand beschränkt werden. Außerdem kann die horizontale Orientierung festgelegt werden, in dem man die Zuordnung des Segments null 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 vom Segment SEG127. Die Einstellung der vertikale Orientierung erfolgt mit der Richtung, mit der die Zeilen des Speichers auf die COM-Anschlüsse gelegt werden.

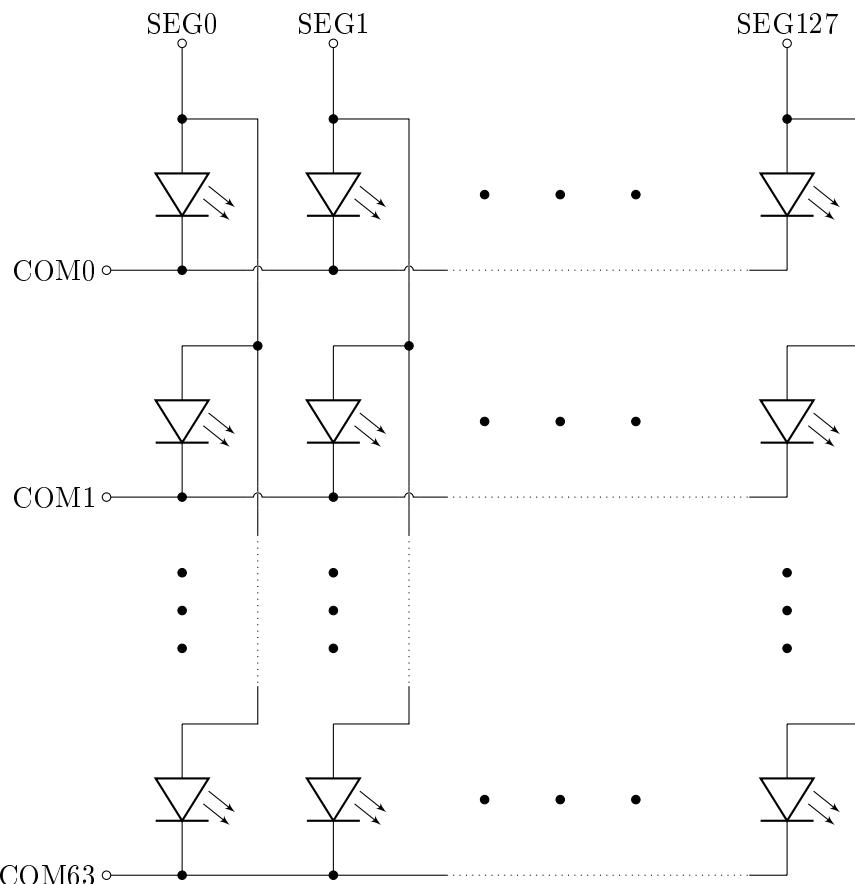


Abbildung 74: Repräsentativer Schaltplan des OLED-Displays mit Segment- und COM-Anschlüssen

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. Diese Module besitzen mehrere Schnittstellen wie zum Beispiel UART, SPI oder I2C. Zu Beginn wird der Pin 3.24 als SCL und der Pin 3.25 als SDA des Flexcomm2 Moduls konfiguriert. Danach wird das Modul selbst mit der Funktion „I2C_MasterInit“ von NXP so initialisiert, dass die Taktfrequenz des SDA Signals ca. **333 kHz** beträgt. Dabei ist das Problem entstanden, dass das Modul zuerst keine Daten sendet. Nach dem Debuggen und Überprüfen jedes Registers des Moduls wird festgestellt, dass der Takt für das Modul deaktiviert ist. Dies ist ärgerlich, da bei der Konfiguration auf Funktionen zurückgegriffen wird, die von NXP stammen. Nachdem der Takt für das Flexcomm2 Modul aktiviert ist, werden die Daten wie gewünscht über die Schnittstelle gesendet.

Um den Prozessor zu entlasten wird auf die zusehenden Daten per Direct Memory Access (DMA) zugegriffen. Dadurch werden diese parallel zum Prozessor in das Senderegister des Flexcomm2 Moduls geladen und der Prozessor wird nicht mit einfaches hin- und herschie-

ben von Daten aufgehalten. Zu aller erst wird das DMA Modul DMA0 mit der von NXP zur Verfügung gestellten Funktion „DMA_Init“ initialisiert. Dem Datenblatt kann in Tabelle 300 entnommen werden, dass der Kanal fünf mit der I2C Funktion des Flexcomm2 Moduls verbunden ist. Deshalb wird mit der ebenfalls Bereitgestellten Funktion „DMA_EnableChannel“ der Kanal fünf freigegeben. Dann wird mit den beiden Funktionen „DMA_CreateHandle“ und „I2C_MasterTransferCreateHandleDMA“ die Funktionalität des DMAs mit dem Flexcomm2 Modul freigegeben.

Jetzt kann mit der Funktion „I2C_MasterTransferNonBlocking“ eine Übertragung über I2C und DMA begonnen werden. Dieser Funktion wird in einer Variable ein Pointer auf die zusehende Variable und deren Länge übergeben. Sollen anzuzeigende Daten an das OLED-Display übertragen werden wird der Funktion ein Pointer auf einen zuvor definierten Speicherbereich und dessen Länge übergeben. Soll eine Konfiguration gesendet werden, muss der Funktion ein Pointer auf eine acht Bit große 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, ob es sich um eine Konfiguration oder um Daten handelt und wie die Start- und Stopbits gesendet werden. Die Auswahl, ob es sich um eine Konfiguration handelt oder nicht erfolgt über die sogenannte Subadresse diese wird nach der Slaveadresse übertragen. Deshalb muss der Funktion ebenfalls die Länge der Subadresse 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 eins zu eins auf dem Display angezeigt wird. Eine andere Möglichkeit wäre hier, dass der Inhalt des RAMs ignoriert wird. Das Display wird im normalem Modus gestartet. Dies bedeutet, dass eine eins im Speicher die Pixel einschaltet. Im Falle des horizontalen Modus wird das Bild, das auf dem Display angezeigt werden soll wie im Kapitel 6.3.1 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 beziehungsweise unten geschoben werden. Mit dem nächsten Kommando wird die horizontale Orientierung eingestellt. In diesem Fall wird der Inhalt des Speichers von Segment 127 nach Segment null ausgegeben. Mit dem Befehl „scan from COM N-1 to COM0“ wird die vertikale Orientierung verändert. Wobei N dem „MUX ratio“ (Anzahl der COM-Anschlüsse) entspricht. Die Hardware der COM Pins wird auf ihren Reset Wert gestellt, was bedeutet dass die geraden Zeilen mit den ersten 32 COM-Anschlüssen und die ungeraden mit den restlichen 32 COM-Anschlüssen verbunden ist. 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 Phase 2 auf jeweils zwei Takte gestellt. In Phase 1 wird die Spannung des

Pixels entladen 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.

AE	Display off	A8 3F	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 7F	set column address start address end address	D5 F0	increace F_{OSC} frequency
22 00 07	set page address to reset value	D9 22	Phase 1 and 2 period to reset
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

Nach der Reset Konfiguration muss noch die Konfiguration erfolgen, die für die Anwendung auf dem Fahrzeug passend ist. Dazu wird die horizontale Orientierung so verstellt, dass die Spaltenadresse 127 im Speicher dem Segment SEG127 entspricht. Ebenso wird die vertikale Orientierung auf den Wert eingestellt, dass das Display von den COM-Anschlüssen COM0 nach COM64 angesteuert werden. Danach wird das Display eingeschaltet.

6.4 Darstellungen auf dem organisches Leuchtdioden-Display

Die im Folgenden genannten Funktionen werden zur Darstellung von Formen und Text beziehungsweise Zahlen verwendet. Beim Aufruf einer dieser Funktionen wird eine Variable gesetzt, die anzeigt, dass dem OLED-Display neue Daten geschickt werden müssen. In einem Echtzeittask wird diese alle 30 ms geprüft und wenn notwendig die neuen Daten gesendet.

6.4.1 Pixelansteuerung

Mit der Funktion „ssd1309_set_pixel“ wird ein Pixel an einer vorgegebenen Position gesetzt oder gelöscht. Dabei wird das anzusprechende in der globalen Variable für den Displaypuffer gesetzt oder gelöscht.

6.4.2 Rechteck zeichnen

Zum Darstellen von Rechtecken wird die Funktion „ssd1309_draw_rect“ verwendet. Dieser werden die Koordinaten der linken oberen und rechten unteren Ecke, eine Variable die angibt ob das Rechteck ausgefüllt ist oder nur der Rand angezeigt werden soll. Außerdem muss angegeben werden, ob die Pixel des Rechtecks angezeigt oder gelöscht werden sollen. In der Funktion werden mit zwei Schleifen alle Pixel innerhalb des Rechtecks durchgegangen und für jeden Pixel bestimmt, ob dieser angezeigt werden soll oder nicht. Soll das Pixel angezeigt werden, wird mit der Funktion „ssd1309_set_pixel“ aus Kapitel 6.4.1 dargestellt.

6.4.3 Buchstabe darstellen

Der Funktion „ssd1309_write_char“ wird eine char Variable übergeben, die 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 der übergebene Charakter 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 jedes Pixel innerhalb der Höhe und Breite des Zeichens geprüft, ob dieses angezeigt werden soll. Pixel, die angezeigt werden, werden mit der Funktion „ssd1309_set_pixel“ aus Kapitel 6.4.1 dargestellt. Die Position wird über einen Cursor eingestellt, der vor Funktionsaufruf gesetzt werden muss.

6.4.4 Text darstellen

Die Funktion „ssd1309_write_str“ stellt am Display einen Text dar, der mit einem Pointer auf ein Charakterarray übergeben wird. Da in der Funktion die Funktion „ssd1309_write_char“ in einer Schleife, die sofort ausgeführt wird, wie viele Elemente das Array hat, verwendet wird, muss der Funktion zum Darstellen eines Textes auch die Höhe und Breite der Zeichen in Form einer „ssd1309_font_t“ Variablen übergeben werden. Die Position des Textes wird wie bei den Buchstaben über den Cursor eingestellt, der vor Funktionsaufruf gesetzt werden muss.

6.4.5 Bilder darstellen

Die Funktion „ssd1309_draw_img“ stellt Bilder dar, die als Hexadezimal Wert abgespeichert sind. Die Position der linken oberen Ecke muss der Funktion übergeben werden. Das Bild selbst wird in einer „struct“ Variable mit dem Namen „ssd1309_img_t“ übergeben, die ebenfalls die Breite und Höhe des Bildes enthält. Angezeigt wird das Bild mit zwei Schleifen.

fen, in denen jedes Pixel überprüft wird, das angezeigt werden soll. Dargestellt werden die einzelnen Pixel mit der Funktion „ssd1309_set_pixel“ aus Kapitel 6.4.1.

6.5 Menuführung

Der Mikrocontroller muss die Menüführung auf Basis der Werte des Drehencoders und der beiden Taster vornehmen. Ein Zustandsautomat entscheidet je nach dem ob der Hauptbildschirm oder ein Menü angezeigt wird, was die einzelnen Bedienelemente bewirken. Bevor der Zustandsautomat ausgeführt werden kann, müssen die beiden Zustände der entprellten Taster sowie die Anzahl der Schritte, die der Drehencoder in welch Richtung gemacht hat, gespeichert werden. Der Zustandsautomat berechnet alle 20 ms das aktuelle Menü und ändert den Displaypuffer ab, sodass das aktuelle Menü dargestellt wird.

Es gibt insgesamt vier verschiedenen Arten von Menüs.

- MENU_LINK
- MENU_CHECK

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 die Zeilenkamera verwendet wurde, wird für dieses Projekt dieselbe 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 Wichtigkeit.

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 dann sein, wenn die additive Kamera im Gegensatz zur Zeilenkamera auf der Strecke weiter voraussieht, um Streckenverlaufs-Änderungen früher zu erkennen. 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.

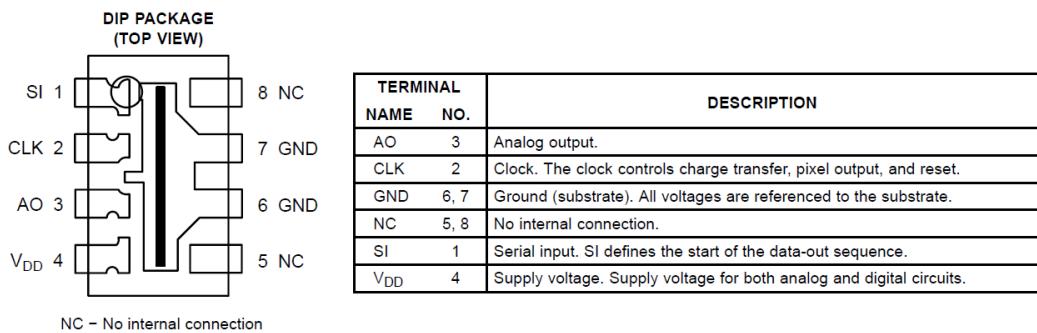


Abbildung 75: Pinbelegung der Taos Zeilenkamera TSL1401R-LF [Tao06]

In Abbildung 75 ist die Anschlussbelegung des Kamerachips und in Abbildung 76 der zeitliche Verlauf eines Bildtransfers aus Sicht der Kamera abgebildet. Je höher die Taktfrequenz der Kamera-Clock (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-Clock 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 Clock-Flanke ereignen. Je Clock-Zyklus 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 Clock-Zyklus zu Ende sein. Die Konversion hingegen hat Zeit bis zum nächsten ADC Trigger. Die Clock-Frequenz wird also von der Dauer einer ADC-Konversion beschränkt und nicht durch die technische Grenze von 8MHz.

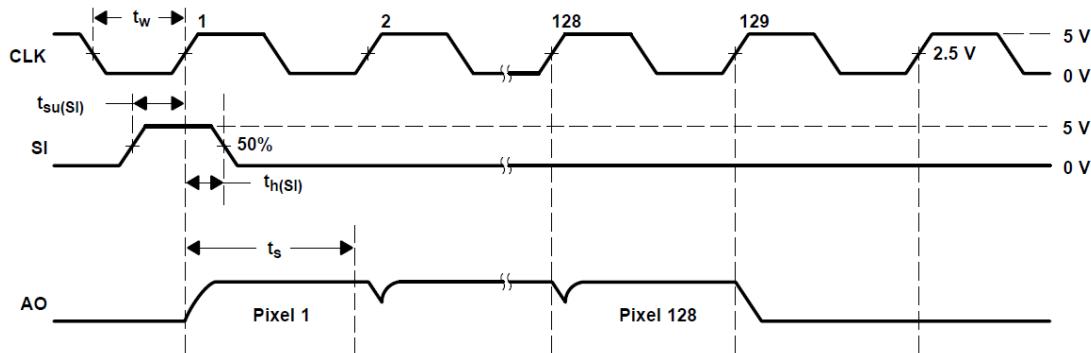


Abbildung 76: Zeitverlauf einzelner Pixeltransfers aus Sicht der Kamera TSL1401R-LF relativ zur vom Controller bereitgestellten Kamera-Clock [Tao06]

Nach dem 128. Clock-Zyklus ist die Aufnahme eines Bildes eigentlich abgeschlossen. Trotzdem darf die nächste Bildaufnahme erst nach 129 Clock-Zyklen und einer zusätzlichen „Pixel Charge Transfer Time“ (t_{qt}) von 20 μ s erfolgen (siehe Abbildung 77).

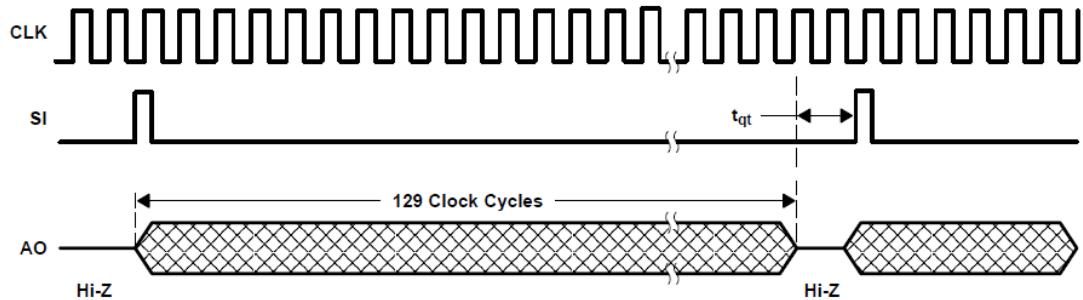


Abbildung 77: Zeitverlauf eines vollständigen Bildtransfers aus Sicht der Kamera TSL1401R-LF mit der Mindestdauer von 129 Clock-Zyklen und einer Wartezeit von t_{qt} („Pixel Charge Transfer Time“ = min. 20 μ s) [Tao06]

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 werden die Spannungspegel des analogen Ausgangs der Kamera 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 Bildaufnahmegeschwindigkeit eines einzelnen Bildes!

Abbildung 78 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-Clock-Signals 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 zur Clock fest. Das Rücksetzen des SI-Signals erfolgt immer, 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 Clock-Zyklus. Die Speicherung der Werte erfolgt allerdings nur dann, wenn gerade eine Aufnahme getätigt wird, also insgesamt 128 mal nach dem Erscheinen des SI-Signals.

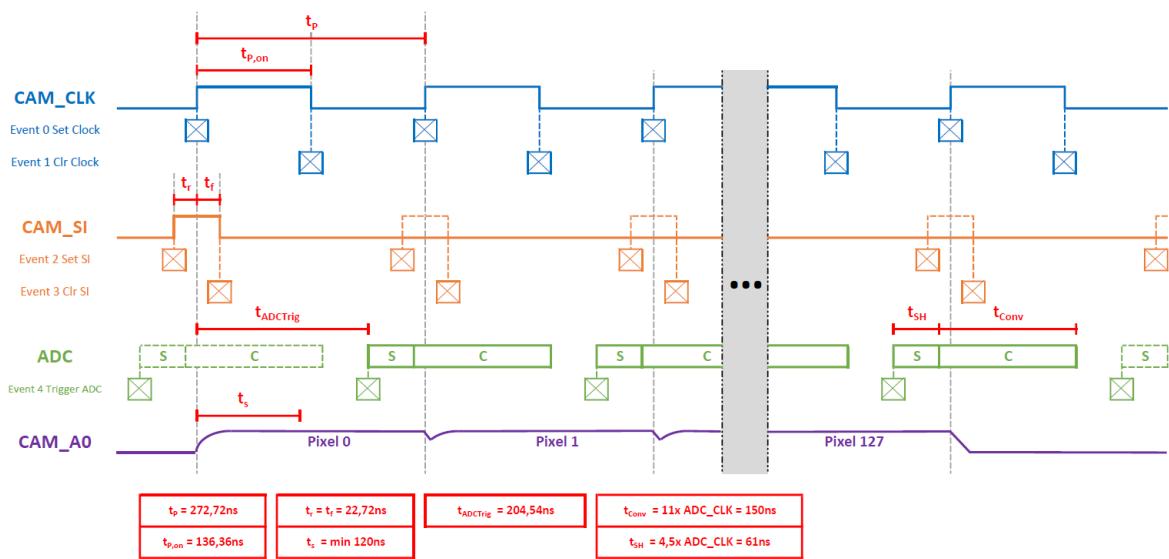


Abbildung 78: Zeitlicher Ablauf einer einzelnen Bildaufnahme aus Sicht des Controllers

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 79). 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 79: Prototypen der Kamerafunktionen aus „camera.c“ in der Datei „camera.h“

Die Primärfunktion der Kamera-Initialisierung („CAM_Init“, siehe Abbildung 80) ruft aus Übersichtlichkeitsgründen lediglich alle anderen notwendigen Initialisierungs- und Konfigurationsfunktionen auf und startet an ihrem Ende den SCTimer SCT0, welcher die Clock-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 80: 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 statthen geht, ist die des CTIMER0 („CTIMER0_Init“, siehe Abbildung 81). 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 81: 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 angepasst werden und das am besten nach jeder Aufnahme. Belichtet wird bei der Kamera immer dann, wenn nicht gerade eine Aufnahme getätigt wird. Um die Belichtungszeit anzupassen, muss also lediglich der Überlaufwert des Timers im Shadow Register verändert werden (Anpassung der Regelmäßigkeit des die 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 -> Belichtungszeit verkürzen
Wenn errechnete mittlere Helligkeit niedriger als 128 -> 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 82). Darin wird die Clock für den SCTimer SCT0 mit einer Frequenz von 44MHz festgelegt. Die maximale Clock Frequenz für den SCTimer beträgt zwar 100MHz, jedoch reicht die Zeitauflösung bei 44MHz leicht für die Genauigkeit der Zeiteinstellung der einzelnen Events.

```
/****************************************************************************
 * 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 82: Funktion „SCTimer_Clock_Config“ für die Konfiguration der Modul-Clock des SCTimers; Teil der Datei „camera.c“

Als nächstes wird mit der Funktion „SCTimer_CamCLK_Init“ das Kamera-Clock Signal erstellt (siehe Abbildung 83), also die Events 0 und 1 konfiguriert. Die Clock 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 die Kamera-Clock 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). Beide Events sind immer aktiv, die Kamera-Clock läuft also 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 83: Funktion „SCTimer_CamCLK_Init“ für die Initialisierung der Events für die Kamera-Clock; 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 84). 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 der Kamera-Clock 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 der Clock 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 = 0xFFFFFFF;         //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 84: Funktion „SCTimer_SIEvents_Init“ für die Initialisierung der Events für den seriellen Eingang der Kamera; Teil der Datei „camera.c“

```
/*
 * 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 = 0xFFFFFFF;           //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 85: 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“

Wird bei einem Timer Überlauf des Timers CTIMER0 ein Interrupt Request ausgelöst, wird daraufhin die Interrupt Service Routine „CTIMER0_IRQHandler“ ausgeführt (siehe Abbildung 85). 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 kann. 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 Belichtungsduer.

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 86). 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 86: 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 87). 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 die Clock für den ADC aktiviert und eine Kalibrierung gestartet (siehe Funktion „ADC_Calibration“). Ist die Kalibrierung des ADC zu Ende, wird der ADC mit einer Clock 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 dann für eine 8bit Auflösung $(11+4,5) \cdot ADC_{CLKCycle}$, also insgesamt 211ns. Im Anschluss daran wird noch die Trigger-Quelle auf SCT0_OUT4 festgelegt und die Interrupt Service Routine des ADC für jede fertige Konversion aktiviert.

```

/*
 * Configure ADC
 ****
 void ADC_Config(void)
{
    //*****
    //Configure Pin P[0][16] (J12 Pin2) (ADC0IN4)
    IOCON->PIO[0][16] &= ~0xFFFFFFF0; //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(200, 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(); //ADC Calibration Sequence

    //*****ADC fundamental configuration
    ADC0->CTRL |= 3-1; //CLK Divider 220MHz/3 = 73,33MHz (80MHz max)
    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 87: Funktion „ADC_Config“ für die Konfiguration des Analog-Digital-Wandlers; Teil der Datei „camera.c“

Die ADC-Kalibrierung folgt einem festen Ablauf. Damit es egal ist, wann die Kalibrierfunktion „ADC_Calibration“ (siehe Abbildung 88) 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 Kalibrationszyklus gestartet.

Nach jeder ADC-Konversion wird ein Interrupt Request ausgelöst und die Interrupt Service Routine „ADC0_SEQA_IRQHandler“ (siehe Abbildung 89) 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 cycles
    if (ADC_CALIB_CALIB_MASK == (ADC0->CALIB & ADC_CALIB_CALIB_MASK)) {
        return; // Calibration timed out.
    }
    //***** //Rewrite ADC Configurations

    ADC0->CTRL = tmp;
}
```

Abbildung 88: 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 89: Interrupt Service Routine „ADC0_SEQA_IRQHandler“ für die Ergebniswertaufnahme nach jeder ADC-Konversion; Teil der Datei „camera.c“

Aus den vom ADC gemessenen Werten können mit dem Algorithmus aus Abbildung 90 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 90: 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 ein Verrechnung nebeneinanderliegender Pixelwerte für die bessere Erkennung der Fahrbahnbegrenzungen.

8 Verteilerplatinen

Anders als beim Fahrzeug von Herrn Arne Kullina werden bei dieser Fahrzeugversion zwei statt nur einer Verteilerplatine verbaut. Die Leistungsverteilung findet auf der Grundplatte statt und die steckbare 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 91) 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 zerlegt 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 91: Leistungsverteilerplatine auf der unteren Fahrzeugebene

8.2 Signalverteilerplatine auf der oberen Fahrzeugebene

Auf der Signalverteilerplatine befindet sich außer 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 der Abbildung 92 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.

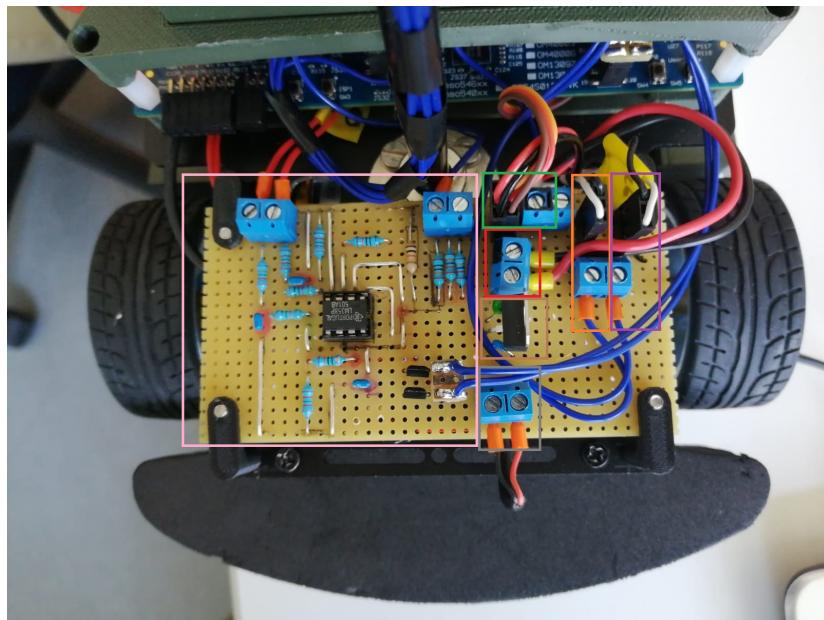


Abbildung 92: Signalverteilerplatine auf der oberen Fahrzeugebene

9 Zusammenfassung, Fazit und Ausblick

9.1 Zusammenfassung und Fazit

9.2 Ausblick

- **Test verschiedener Kameralinsen
- **Programmierung der automatischen Belichtungszeitanpassung (evtl. Regelung)
- **Optimierung der Drehzahlmessung mit einer zweistufigen OPV-Referenzspannungsschaltung
- **Implementierung der Drehzahlregelung
- **Implementierung der Streckenregelung
- **Inbetriebnahme des Fahrzeugs
- **3D-Druck Fahrzeuggehäuse?
- **Obere Verteilerplatine professionell fertigen?
- **zweite Kamera für weiter vorausschauende Streckenerkennung (hilft z.B. bei Kreuzungen)
- **Stop bei Ziellinie implementieren
- **Hinderniserkennung mit Ultraschallboard
- **Anderes Batteripaket?
- **Kamera -> Ergebnisanzeige auf Display (Visualisierung)
- **Fahrgeräusche reduzieren?

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 des Drucktasters	9
21	Gedruckte Controller-Halterung	10
22	Gedruckte Drucktaster-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 Versorgungsspannungs-Verteilerplatine	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	18
41	Montage der Antriebskomponenten	19
42	Initialisierungssequenz der ESCs	20
43	Programmierung des Arduino Nano zur ESC-Konfiguration	21
44	Parameter der neuen ESC-Konfiguration	21
45	Relevante Zeilen der Datei „drive.h“	22
46	Extremwerte der Antriebsgeschwindigkeit als Parameter aus dem EEPROM .	23
47	Funktionen BLDC_Init und CTIMER3_Init der Datei „drive.c“	24
48	Funktion BLDC_Init_Task der Datei „drive.c“	25
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	31
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	inp_inref_opv_out	36
58	Schaltungsaufbau_mit_Mosfet	38
59	Problematik_stop_throttle	39
60	Schaltungserweiterung	40
61	rpmDefines	41
62	rpmmeas_init_ctimer2_init	42
63	isr	43
64	calculation	43
65	Servomotor mit Anschlussleitungen [DIY21]	45
66	Montage der Lenkungskomponenten	46
67	Relevante Zeilen der Datei „servo.h“	47
68	Extremwerte der Lenkwinkel als Parameter aus dem EEPROM	47
69	Funktionen SERVO_Init, CTIMER1_Init und SERVO_Demo der Datei „drive.c“	48
70	SERVO_Demo Task der Datei „drive.c“	49
71	Schaltplan des Bedienungsboards 1	50
72	Schaltplan des Bedienungsboards 2	51
73	Flussablaufdiagramm Taster und Drehencoder einlesen	52
74	Schaltplan OLED	55
75	Pinbelegung der Taos Zeilenkamera TSL1401R-LF [Tao06]	60
76	Zeitverlauf einzelner Pixeltransfers aus Sicht der Kamera [Tao06]	61
77	Zeitverlauf eines vollständigen Bildtransfers aus Sicht der Kamera [Tao06] .	61
78	Zeitlicher Ablauf einer einzelnen Bildaufnahme aus Sicht des Controllers . .	62
79	Prototypen der Kamerafunktionen in der Datei „camera.h“	63
80	Funktion „CAM_Init“ aus der Datei „camera.c“	63

81	Funktion „CTIMER0_Init“ aus der Datei „camera.c“	64
82	Funktion „SCTimer_Clock_Config“ aus der Datei „camera.c“	65
83	Funktion „SCTimer_CamCLK_Init“ aus der Datei „camera.c“	66
84	Funktion „SCTimer_SIEvents_Init“ aus der Datei „camera.c“	67
85	Interrupt Service Routine „CTIMER0_IRQHandler“ aus der Datei „camera.c“	67
86	Funktion „SCTimer_ADCTrigger_Init“ aus der Datei „camera.c“	68
87	Funktion „ADC_Config“ aus der Datei „camera.c“	69
88	Funktion „ADC_Calibration“ aus der Datei „camera.c“	70
89	Interrupt Service Routine „ADC0_SEQA_IRQHandler“ aus der Datei „camera.c“	70
90	Algorithmus für die Bestimmung der Spannungswerte und logischen Werte der Pixel	71
91	Leistungsverteilerplatine auf der unteren Fahrzeugebene	72
92	Signalverteilerplatine auf der oberen Fahrzeugebene	73

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
- [Sem19] *LPC546xx User Manual. : LPC546xx User Manual*, 2019
- [Tao06] *TSL1401R-LF User Manual. : TSL1401R-LF User Manual*, 2006
- [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

