

Technikerprojekt: E-Bike ECU

Leon Reeh

6. Februar 2025

Vorwort Um das Lesen zu vereinfachen, wurde die Dokumentation in vier Teile unterteilt. Diese vier Abschnitte beschäftigen sich jeweils vollständig mit einem Aspekt des Projekts und stehen für sich:

Teil 1: Einleitung und Beschreibung des vollständigen Systems.

Teil 2: Dokumentation des fertigen Schaltplans, Layouts, der Software und Mechanik im Detail.

Teil 3: Der Entwicklungsprozess mit Designänderungen und deren Hintergründen.

Teil 4: Anhänge, einschließlich des vollständigen Schaltplans und Quellausschnitte.

Die vollständige Dokumentation sowie der vollständige und durchgehend verfügbare Software-Quellcode, Schaltpläne und weitere Dateien stehen ebenfalls in digitaler Form auf GitHub unter <https://github.com/leonreeh/E-Bike-ECU> zur Verfügung.

Lizenz Dieses Projekt und die hier beiliegende Dokumentation fallen unter die MIT-Lizenz (X11-Lizenz) *Leon Reeh 2025*

Inhaltsverzeichnis

I. Einleitung und Projektbeschreibung	5
1. Einleitung	6
1.1. Technische Spezifikation	6
2. Systemdesign	7
2.1. Funktionsbeschreibung	7
2.2. Bedienung	8
2.3. Sicherheitsfunktionen	9
II. Schaltplan, Layout und Software	10
3. Schaltplan	11
3.0.1. Kernbauteile	11
3.1. Netzteil	11
3.1.1. Powertrain	12
3.1.2. DC/DC	12
3.1.3. Protection circuits	13
3.1.4. Hardware Interrupt	14
3.1.5. Mikrocontroller	15
3.1.6. BLDC	17
3.1.7. Beleuchtung	19
3.1.8. Temperatur	20
3.1.9. IO	21
4. Layout	22
4.1. Design Regeln	23
4.2. PCB Dimensionierung	23
4.3. Layer Design	24
5. Software	26
5.1. Prozesse & Softwarearchitektur	27
5.1.1. Zustandsautomat/State Machine	28
5.2. Source Code	30
5.2.1. Main.c/h	30
5.2.2. BLDC.c/h	32
5.2.3. liquidcrystal_i2c.c/h	32

5.2.4. Mymath.c/h	33
5.2.5. stm32f4xx_it.c/h	34
5.3. Controller Konfiguration	35
6. Gehäuse	37
6.1. Mechanische Eigenschaften	37
6.2. Konstruktion	38
6.3. Gehäuse Komponenten	38
III. Entwicklungsprozess und Relevante Konzepte	41
7. Entwicklungsprozess	43
8. Designphasen	45
8.1. Designphase 1	45
8.2. Designphase 2	46
8.3. Designphase 3	47
8.4. Abschluss der Entwicklung	48
9. Relevante Konzepte	49
9.1. BLDC	49
IV. Anhang	52
10. System Design	53
11. Entwicklung	55
12. Schaltplan	57
13. Layout	65
14. BOM	75
15. Source Code	82
15.1. Main.c/h	82
15.2. BLDC.c/h	94
15.3. liquidcrystal_i2c.c/h	99
15.4. Mymath.c/h	102
15.5. stm32f4xx_it.c/h	104
16. Quellen	105
16.1. Danksagungen	105

Teil I.

Einleitung und Projektbeschreibung

1. Einleitung

Ziel des Projekts ist die Entwicklung eines Steuergeräts für einen akkubetriebenen BLDC-Heckmotor mit einer Nennleistung von 1000 W bei 48 V für ein E-Bike. Das Steuergerät soll alle relevanten Bedienelemente für den Motorbetrieb sowie die Steuerung der Beleuchtung integrieren. Die Umsetzung erfolgt mit einer selbst entwickelten Platine auf Mikrocontroller-Basis und einer eigens programmierten Software. **Motivation** Warum habe ich dieses Projekt gewählt? Nach zehn Jahren Firmenzugehörigkeit bei Panasonic Lüneburg – einem Unternehmen mit Fokus auf dedizierte Elektroniklösungen – war für mich klar, dass die Entwicklung einer eigenen Elektronik im Mittelpunkt meines Techniker Projekts stehen sollte. Elektrische Antriebe sind nach wie vor ein aktuelles Thema, weshalb die Idee eines E-Bike-Antriebs nahe lag. Warum 1kW als Herausforderung? Mit großer Leistung kommt große Verantwortung! Da ich das Projekt privat und unter selbst gestellten Anforderungen entwickeln wollte, setzte ich bewusst auf eine hohe Leistung von 1 kW. Bei Strömen über 20 A ist die Fehlertoleranz gering. Die Bauteile müssen sorgfältig ausgewählt werden, Leiterbahnen und Kabelquerschnitte müssen korrekt dimensioniert sein, und Sicherheits- sowie Abschaltfunktionen müssen sowohl in Hardware als auch in Software sicher implementiert werden, um die Funktion und den Personenschutz zu gewährleisten.

1.1. Technische Spezifikation

E-Bike:

Reichweite: 20-40km abhängig von Fahrweise

Höchstgeschwindigkeit: >40km/h

Leistung/Ps: 1KW/1,36PS

Akku:

Ausgangsspannung: 56V-43V

Ausgangstrom: 30A

Energie: 15A/h

Steuergerät:

Eingangsspannung: 13V-64V

Stromaufnahme: standby 0.10A/ max 22 A

Leistungsaufnahme: 1,3W - 1.218W

Temperaturbereich Umgebung(Tu): 10°C - 40°C

Temperaturbereich Elektronik: Tu - 120°C

2. Systemdesign

Das fertige System besteht aus sechs Baugruppen, die über das eigens entwickelte Steuergerät vereint werden (siehe Anhang System Design 10.1). Dadurch entsteht ein vollständig integriertes System, das dem Nutzer die Bedienung des E-Bikes erleichtert. **Akku:** 48 V, 15 Ah Lithium-Ionen-Batterie **Motor:** Viribus 26-Zoll BLDC-Heckmotor mit 1000 W Nennleistung **Display:** 20x4 LCD-Display zur Anzeige von Geschwindigkeit und Systeminformationen **Bedienelemente:** „Motorrad-Style“-Lenkerbedienung für alle Funktionen **Beleuchtung:** Abblendlicht, Rücklicht, Bremslicht und Blinker **Steuergerät:** Das Fokus Element der Techniker Arbeit.

Siehe Anhang System Design 10 für eine Übersicht der relevanten System Bauteile.

2.1. Funktionsbeschreibung

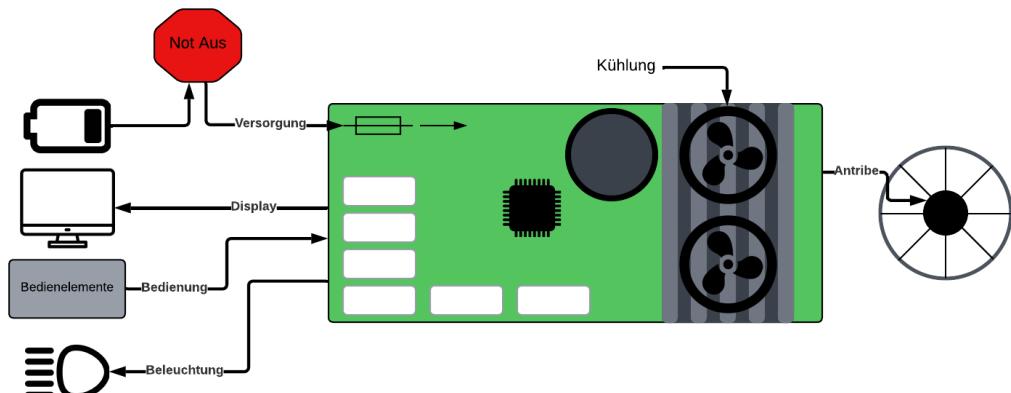


Abbildung 2.1.: Vereinfachtes Steuergerät

Das Steuergerät des E-Bikes bietet folgende Hauptfunktionen: BLDC-Heckmotor-Antrieb: 3-Phasen H brücke für den Antrieb eines bürstenlosen Gleichstrommotor (Brushless DC/BLDC). Beleuchtungselemente: Integrierte LED-Vorder- und Rücklichter sorgen für Sichtbarkeit und Sicherheit. Blinkerelemente und Bremslicht für Richtungsanzeigen. Darstellung wichtiger Betriebsdaten über LCD: Spannung, Strom, Temperatur, Geschwindigkeit, Fehler Code. Integrierte Temperatur-Überwachung mit diskreter Lüfter Ansteuerung.

2.2. Bedienung

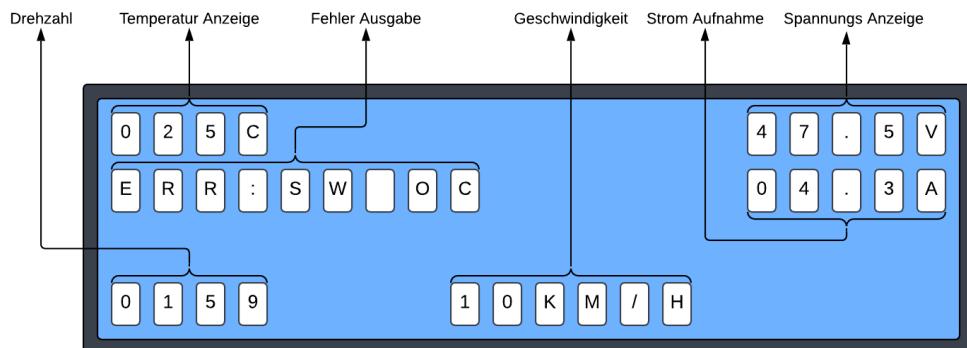


Abbildung 2.2.: Display Ausgabe

Die verschiedene Bedienelemente für den Benutzer befinden sich am Lenker in einem Motorrad ähnlichen system: **E-Stop**: Deaktiviert das Selbstthalte-Relais und schaltet das Steuergerät aus. **E-Start**: Aktiviert das Gerät und betätigts das Selbstthaltesystem, wodurch das E-Bike in den Betriebsmodus versetzt wird. **Bremshebel**: Aktiviert die Bremslichter. Schaltet die Motorbremse am Hinterrad ein, um eine zusätzliche Verzögerung zu bieten. **Gasgriff**: Reguliert die Motorleistung und steuert die Geschwindigkeit. **Beleuchtung**: **Taster Ablendlicht**: Schaltet das Ablendlicht und den Rückstrahler ein. **Taster Blinker Links/Blinker Rechts**: Steuert die Blinkerelemente für die jeweilige Fahrtrichtung. Diese Funktionen und Eingabemöglichkeiten stellen sicher, dass das E-Bike sowohl sicher als auch benutzerfreundlich im Betrieb ist.

2.3. Sicherheitsfunktionen

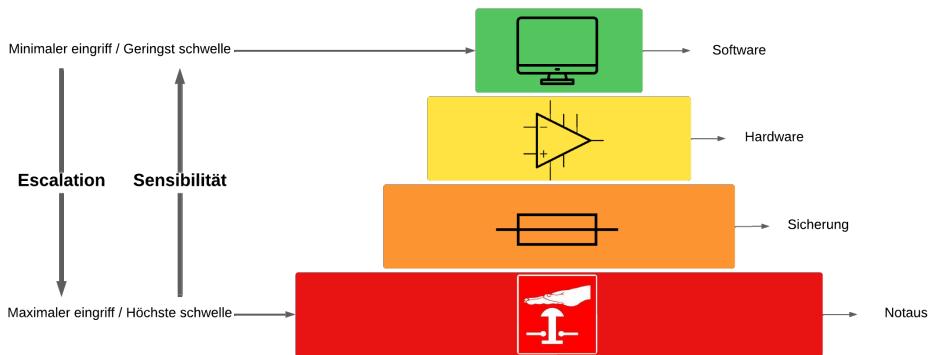


Abbildung 2.3.: Sicherheitsfunktion

1 Software

Auslösung: Messvorrichtungen überwachen Spannung, Strom und Temperatur. Überschreiten diese Werte festgelegte Schwellen, wird der Lastbetrieb automatisch abgeschaltet. Eingriff: Das Gerät wechselt in einen Sicherheitszustand, aus dem es sich selbstständig zurücksetzen kann, sobald die Bedingungen wieder im Normalbereich liegen.

2 Hardware

Auslösung: Hardwarebasierte Messvorrichtungen erkennen Spannungen, Ströme oder Temperaturen außerhalb der spezifizierten Werte und schalten das Gerät ab. Eingriff: Das Gerät wechselt in einen Sicherheitszustand, der nur durch einen manuellen Neustart wieder verlassen werden kann.

3 Sicherung

Auslösung: Stromkreise für 48V (25A) und 12V (3A) sind durch Sicherungen gegen Kurzschlüsse geschützt. Eingriff: Bei Überlast oder Kurzschluss wird das Gerät außer Betrieb gesetzt. Zum Wiederinbetriebnehmen müssen die betroffenen Stecksicherungen ersetzt werden.

4 Notaus

Auslösung: Im Notfall betätigt der Nutzer den Notaus-Schalter. Eingriff: Das Gerät wird sofort vom Akku getrennt und vollständig stromlos gemacht, um maximale Sicherheit zu gewährleisten.

Hinweis zu BMS-Schutzfunktionen

Das im Akku integrierte Battery Management System (BMS) verfügt zusätzlich über Schutzvorrichtungen gegen Überstrom, Unterspannung und Überhitzung. Diese Funktionen sind unabhängig von den hier beschriebenen Sicherheitsmaßnahmen und waren nicht Teil der Projektarbeit. Dennoch tragen sie wesentlich zur Gesamtsicherheit des Systems bei.

Teil II.

Schaltplan, Layout und Software

3. Schaltplan

In diesem Abschnitt der Dokumentation werden die einzelnen Teile des Schaltplans vorgestellt und detailliert beschrieben. Der vollständige, ununterbrochene Schaltplan ist im Anhang oder digital auf GitHub unter: github.com/leonreeh/E-Bike-ECU → RESC_V2 Hardware → RESC_V2 Schematic.pdf

Zur besseren Übersicht und Verständlichkeit des Schaltplans wird die folgende Namensgebung für Signale verwendet:

SENSE – Signal eines analogen Ausgangs (z. B. Spannungsmessung).

DI – Digitaler Eingang (Digital-Logik-Eingang des Mikrocontrollers).

DO – Digitaler Ausgang (Digital-Logik-Ausgang des Mikrocontrollers).

— – Invertierung (Signal arbeitet als Active Low – ausgelöst bei Pegel 0).

3.0.1. Kernbauteile

Bauteil	Name	Beschreibung
Mikrocontroller	STM32F405RGT6	32bit ARM 1024 FLASH 192kB SRAM
Abwärtswandler	LMR36520	2,5A DC/DC Converter 4-65V
Brücken Treiber	IR2101S	Bootstrap Brückentreiber
Power Mosfet	IPB035N08N3GATMA	80V/100A NMos
Leistungs Relais	JD2912-1Z-12VDC	12V/40A DC
Akkustecker	XT90	500V/40A DC
Motorstecker	MT60	500V/30A DC
Systemstecker	JST-XH Serie	250V/3A DC

3.1. Netzteil

Das Netzteil bildet das Herzstück der Energieversorgung und stellt sicher, dass alle elektronischen Komponenten des Systems mit den benötigten Spannungen und Strömen versorgt werden. Es gewährleistet einen stabilen Betrieb der Elektronik. **Funktionalität:**

Selbsthalterelais für zuverlässiges Einschalten, für zusätzlichen Schutz gegen Vibrationen und erschüttern wurde hier ein KFZ Relais gewählt das eine besseres Haltemoment als herkömmliche printrelais bietet.

48V Hochleistungsversorgung für den Motor

DC/DC-Wandler (12V) für Beleuchtung und leistungs Elemente (max. 2.5A)

DC/DC-Wandler (5V) für Display und logik Elemente (max. 2A)

LDO-Regler (3.3V) für eine besonders stabile Versorgung auf Logik-Ebene.

Besonderheiten: Messung des Eingangsstroms Messung der Eingangsspannung Hardwarebasierte Überstromerkennung Hardwarebasierte Unterspannungserkennung.

3.1.1. Powertrain

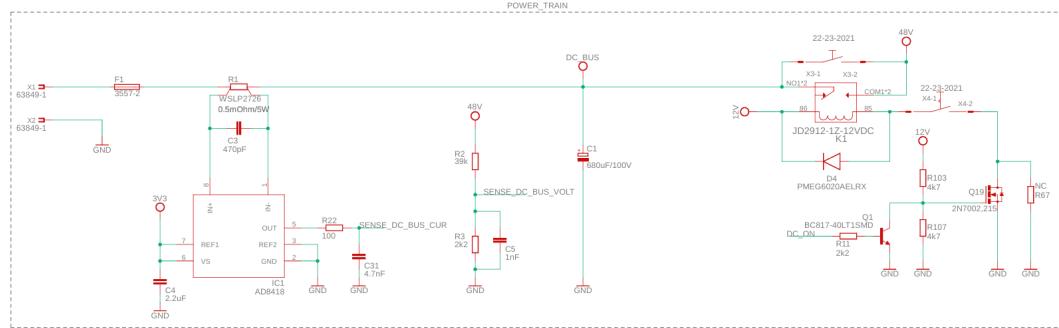


Abbildung 3.1.: RESC_V2 Schaltplan Seite 2

In der ersten Funktionsgruppe, dem „Power Train“, sind alle Komponenten zusammengefasst, die in direkter Verbindung mit dem Akku stehen und den Hochleistungspfad für den Motortreiber bilden. KFZ-Stecksicherung F1: Schützt den Akku vor Überstrom und Kurzschlägen. Shunt R1: Misst den Eingangsstrom und ermöglicht eine präzise Erfassung des Stromverbrauchs. Spannungsteiler R2/R3: Dient zur Messung der Eingangsspannung und leitet die Daten an die Steuerung weiter. Bulk-Kondensator C1: Glättet Spannungsspitzen und stabilisiert die Eingangsspannung, um Schwankungen auszugleichen. Selbstthalte-KFZ-Relais K1: Schaltet die Versorgungsspannung des Niederleistungspfads und sorgt dafür, dass das System sicher ein und ausgeschaltet werden kann. Diese Funktionsgruppe ist essenziell für die sichere Versorgung des Motortreibers und bildet die Grundlage für die gesamte Energieversorgung des E-Bikes.

3.1.2. DC/DC

Um unnötige Verlustleistung zu vermeiden und die Komplexität des Netzteils gering zu halten, werden für die Versorgung der Elektronik lediglich drei Spannungsebenen benötigt:

12V – System-Leistungsspannung als Versorgung der Beleuchtung, Lüfter, Relais und des Motortreibers.

5V – Sensor- und Displayversorgung für Motor-Hall-Sensoren und des LCD-Displays.

3.3V – System-Logikspannung für die Versorgung des Mikrocontrollers, Operationsverstärkers und der analogen Signale.

Durch die Reduzierung auf diese drei Spannungsspegel wird der Aufbau des Netzteils vereinfacht, während gleichzeitig eine stabile und effiziente Energieversorgung für die verschiedenen Baugruppen gewährleistet wird.

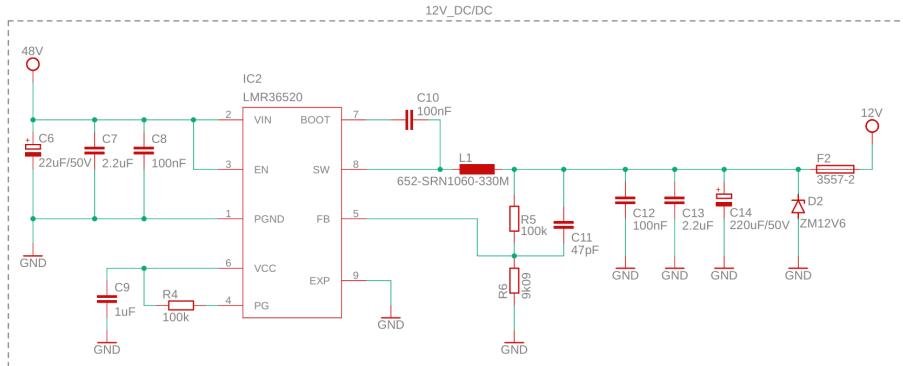


Abbildung 3.2.: 12V Abwärtswandler RESC_V2 Schaltplan Seite 2

Die Beschaltung des LMR36520 entspricht dem TI-Referenzdesign (Datenblatt Abschnitt 9.2 Typical Application) und wurde um eine 12V6 Suppressordiode (Z-Diode) ergänzt, um das System vor Überspannung zu schützen. Diese zusätzliche Schutzmaßnahme trägt zur Erhöhung der Zuverlässigkeit und Langlebigkeit der Spannungsversorgung bei, indem sie Spannungsspitzen effektiv absorbiert und so empfindliche Schaltungsteile schützt. Die Induktivität L1 wurde gemäß den Richtlinien im Datenblatt des LMR36520 dimensioniert (Datenblatt Abschnitt 9.2.1.2.3 – Inductor Selection).

3.1.3. Protection circuits

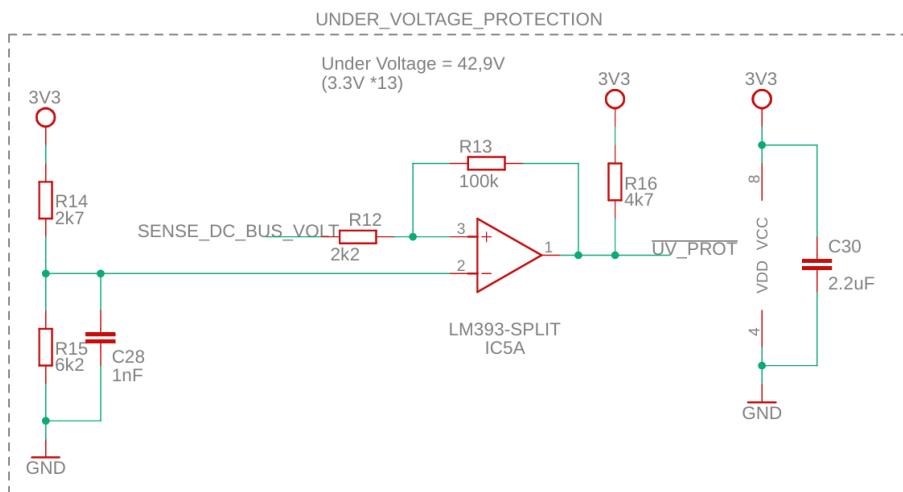


Abbildung 3.3.: Unterspannungsschutz RESC_V2 Schaltplan Seite 2

Für die Umsetzung der Sicherheitsfunktion auf Hardware-Ebene wurde eine einfache Komparatorschaltung implementiert, die die drei Messgrößen (Spannung, Strom,

Temperatur) überwacht. Beim Überschreiten vordefinierter Schwellenwerte sendet sie ein „Fehler“-Signal an den Mikrocontroller. Im Beispiel in Abbildung 3.3 wurde zum Schutz vor Tiefentladung des Akkus eine Spannung von 42,9 V festgelegt. Bei 13 in Reihe geschalteten Zellen ergibt sich daraus eine minimale Zellenspannung von 3,3 V. Über-Strom und -Temperatur Erkennung werden mit der gleichen Schaltung implementiert und nach relevanten Kriterien dimensioniert.

3.1.4. Hardware Interrupt

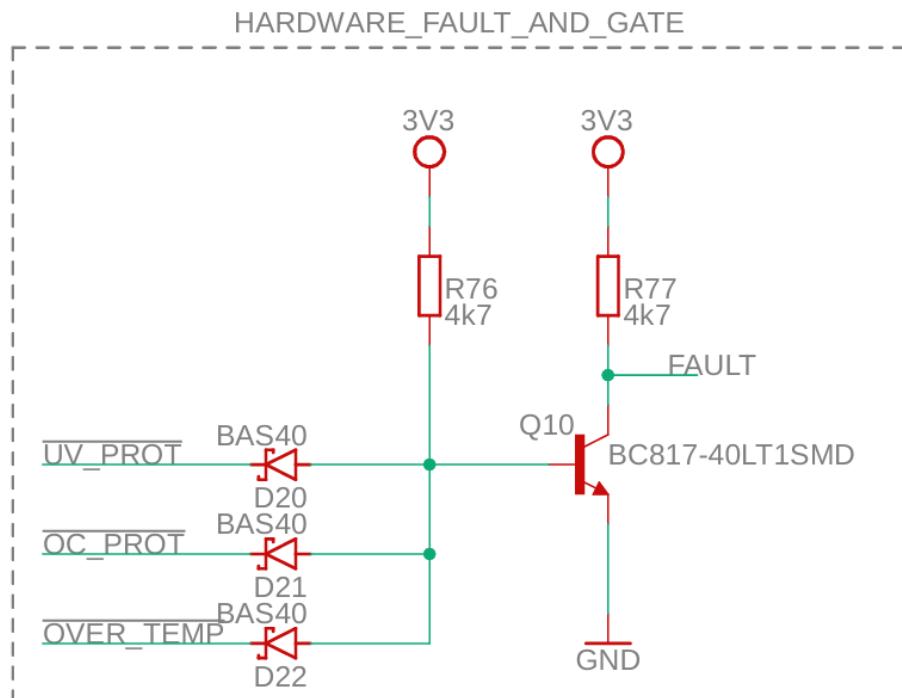


Abbildung 3.4.: Oder Gatter RESC_V2 Schaltplan Seite 3

Um sicherzustellen, dass die Elektronik im Falle eines Hardware-Fehlers immer konsistent reagiert, werden die drei Hardware-Fehlersignale (Strom, Spannung, Temperatur) über ein ODER-Gatter zusammengefasst. Das resultierende **"FAULT"**-Signal wird auf einen Interrupt-Pin des Mikrocontrollers gelegt, sodass dieser bei einem Fehler schnellstmöglich reagieren kann. Durch diese einfache Hardwareschaltung können zwei zusätzliche Interrupt-Pins eingespart werden, da alle drei Fehlersignale zum selben Sicherheitszustand führen sollen und daher keinen eigenen Interrupt-Pin benötigen.

3.1.5. Mikrocontroller

Der STM32F405RGT6 bildet das zentrale Steuerungselement des Systems und koordiniert die gesamte Elektronik. Er übernimmt die Verarbeitung und Weiterleitung von Signalen, die Überwachung der Systemparameter, die Kommunikation mit anderen Modulen und nicht zuletzt die Ansteuerung der Motor H-Brücke.

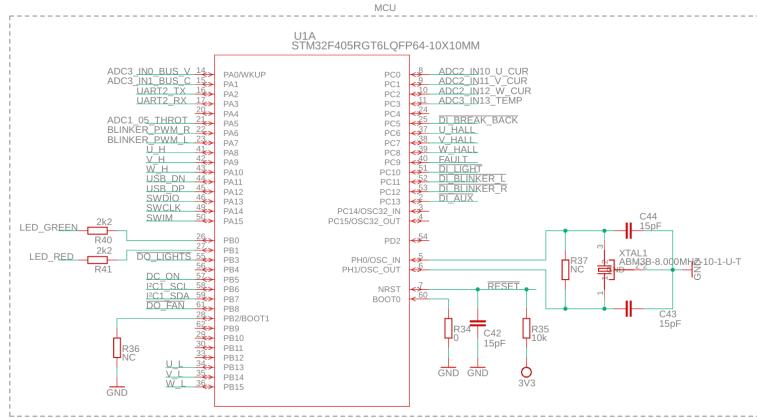


Abbildung 3.5.: Mikrocontroller Pinbelegung RESC_V2 Schaltplan Seite 3

Die Pin-Belegung des Mikrocontrollers wurde mit Hilfe des STM32CubeIDE-Konfigurators festgelegt. Dabei wurden die Timer-PWM-Kanäle, Interrupt-Pins und ADCs konfiguriert. Als Referenz diente der **VESC 6.4 Schaltplan**, der maßgeblich zur Auswahl des Mikrocontrollers beitrug. Diese Vorgehensweise ermöglichte die Funktions-kritisch Aspekte der notwendigen Peripheriefunktionen für die Motorsteuerung und Systemüberwachung zu implementieren.

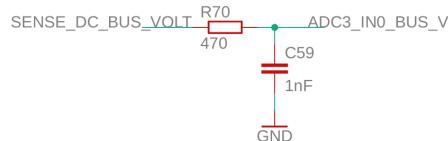


Abbildung 3.6.: RC-Filter RESC_V2 Schaltplan Seite 3

Zur Entstörung der Analogsignale sind alle ADC-Pins des Mikrocontrollers mit einem RC-Glied ausgestattet. Das RC-Glied dient dazu, hochfrequente Störungen zu filtern und sicherzustellen, dass die an den ADC-Eingängen gemessenen Werte stabil und präzise sind. Dies trägt zur Verbesserung der Signalqualität und zur Reduzierung von Messfehlern bei.

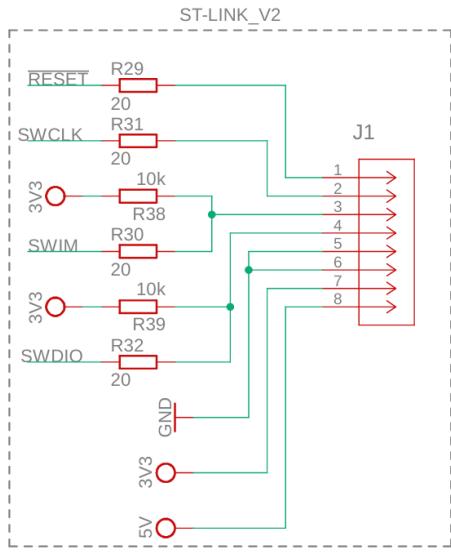


Abbildung 3.7.: ST-Link RESC_V2 Schaltplan Seite 3

Standard SWD-Schnittstelle zum Programmieren und Debugging von STM-Mikrocontrollern. Die Schnittstelle ist vollständig kompatibel mit dem ST-Link V2 über das Serial Wire Debug (SWD)-Protokoll.

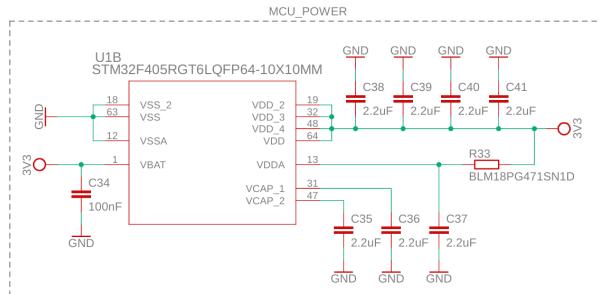


Abbildung 3.8.: Mikrocontroller Versorgung RESC_V2 Schaltplan Seite 3

Jeder Versorgungspin des Mikrocontrollers ist mit einem eigenen 2,2 μ F Kondensator ausgestattet, um eine möglichst stabile Spannungsversorgung zu gewährleisten. Der ADC-Referenzpin (VDDA) ist zusätzlich mit einem Ferritfilter gegen Störungen geschützt.

3.1.6. BLDC

Für die Implementierung des BDLCs wird eine drei Phasen H-Brücke benötigt. dies wurde durch den **Infineon 2101S** Bootstrap-H-Brücken-Treiber und leistungsstarke **IPB035N08N3GATMA**-Power-MOSFETs realisiert.

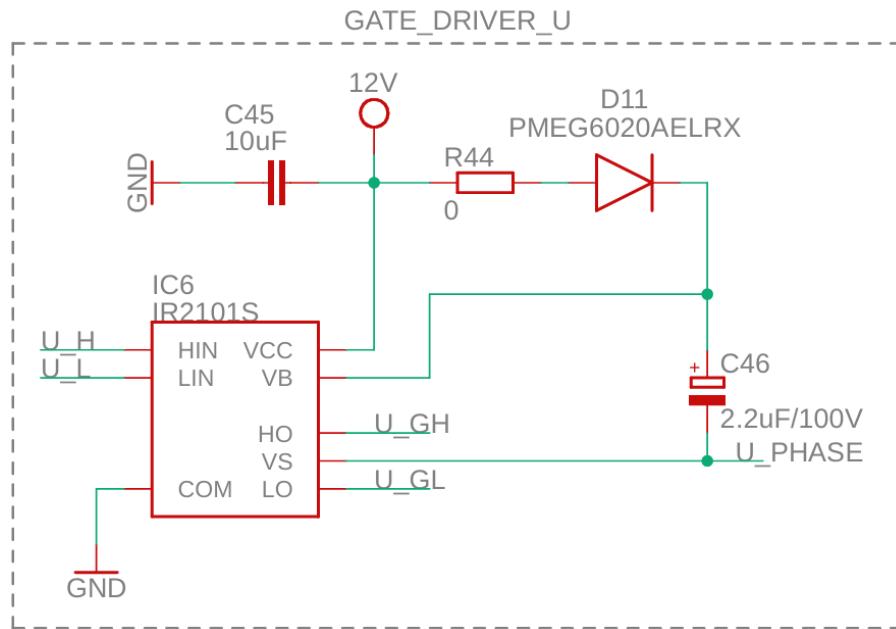


Abbildung 3.9.: H-Brücken Treibe RESC_V2 Schaltplan Seite 4

Das Infineon 2101S Treiber IC wurde gewählt für seine einfache Implementierung, Ansteuerung, integrierte Bootstrap-Schaltung und Short Protection. Die Referenzschaltung wurde um einen Vorwiderstand R44 erweitert der es ermöglicht den Ladestrom von C46 zu dimensionieren.

Dimensionierung des Bootstrap-Kondensators C46 gemäß der Faustregel:

$C_{bootstrap} \geq 10 \times C_{gs}$ ($C_{gs} = 2.2\text{nF}$ Gate-Kapazität der MOSFETs) Diese Dimensionierung stellt sicher, dass der Treiber auch bei hohen Schaltfrequenzen zuverlässig arbeitet und eine stabile Gate-Spannung bereitgestellt wird.

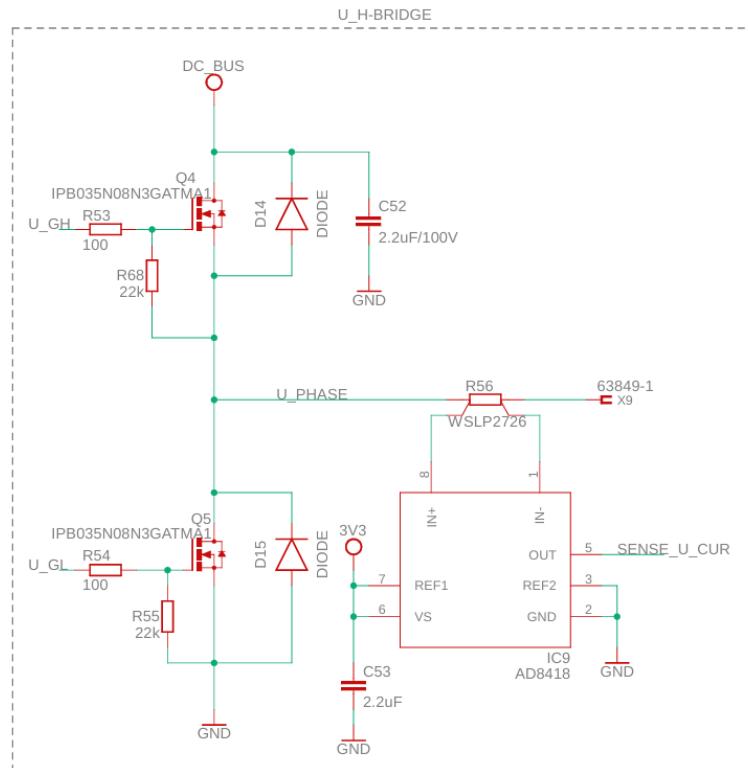


Abbildung 3.10.: H-Brück RESC_V2 Schaltplan Seite 4

Die Strom- und Drehmomentregelung erfolgt durch eine Shunt-Messung direkt am Phasenabgriff der H-Brücke, wodurch der tatsächliche Phasenstrom präzise erfasst und eine genauere Motorregelung ermöglicht wird. Zur Entlastung der MOSFET-internen Body-Dioden und zur Reduzierung der thermischen Belastung, insbesondere bei schnellen Schaltvorgängen und hohen Strömen, werden externe Freilaufdioden (D16/D17) parallel zu den MOSFETs geschaltet. Jedes MOSFET ist mit einem Pull-Down-Widerstand am Gate versehen, um ungewolltes Einschalten durch Leckströme zu verhindern und einen sicheren Sperrzustand zu gewährleisten. Zusätzlich ist jede H-Brücke mit einem 2,2 μ F Kondensator ausgestattet, der Schalt- und Spannungsspitzen reduziert, die MOSFETs vor Überspannungen schützt und zur Glättung der Phasenspannung beiträgt.

3.1.7. Beleuchtung

Die Beleuchtungseinheit steuert die LED-Beleuchtungselemente mittels diskreter MOSFET-Treibern. Ihr primärer Zweck ist die Gewährleistung der gesetzlich vorgeschriebenen sowie funktionalen Fahrzeugbeleuchtung für den Straßenverkehr, wodurch Sicherheit und Sichtbarkeit erhöht werden. Die Beleuchtung umfasst Abblendlicht, Rücklicht, Bremslicht und Blinker, wobei letztere durch Software-PWM geregelt werden, um eine flexible Lichtsteuerung zu ermöglichen.

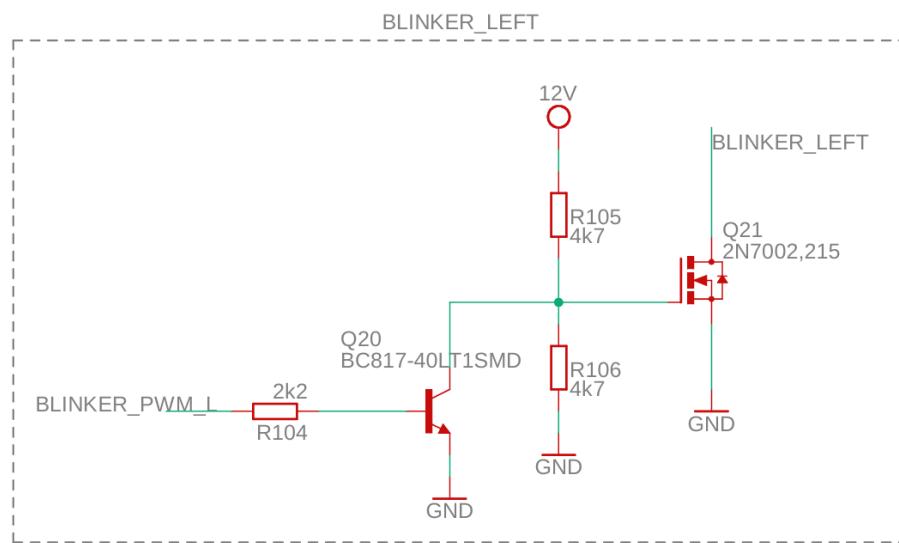


Abbildung 3.11.: H-Brücken Treiber RESC_V2 Schaltplan Seite 5

Alle Beleuchtungs-LEDs verwenden die gleiche Treiberbeschaltung. Der einzige Unterschied zwischen den Treibern liegt in der Größe der MOSFETs, die entsprechend der Leistungsaufnahme der jeweiligen LED-Gruppen dimensioniert werden.

3.1.8. Temperatur

Die Temperaturüberwachung dient dem Schutz der Leistungskomponenten durch eine Messung und Steuerung der Kühlung. Ein NTC-Sensor am Kühlkörper erfasst die Temperatur, während eine MOSFET-basierte Lüftersteuerung, mit optionalem PWM-Betrieb, eine bedarfsgerechte Kühlung ermöglicht. Eine OP-basierte Hardware-Schaltung erkennt Übertemperaturen und reagiert entsprechend, um Schäden an den Leistungskomponenten zu verhindern.

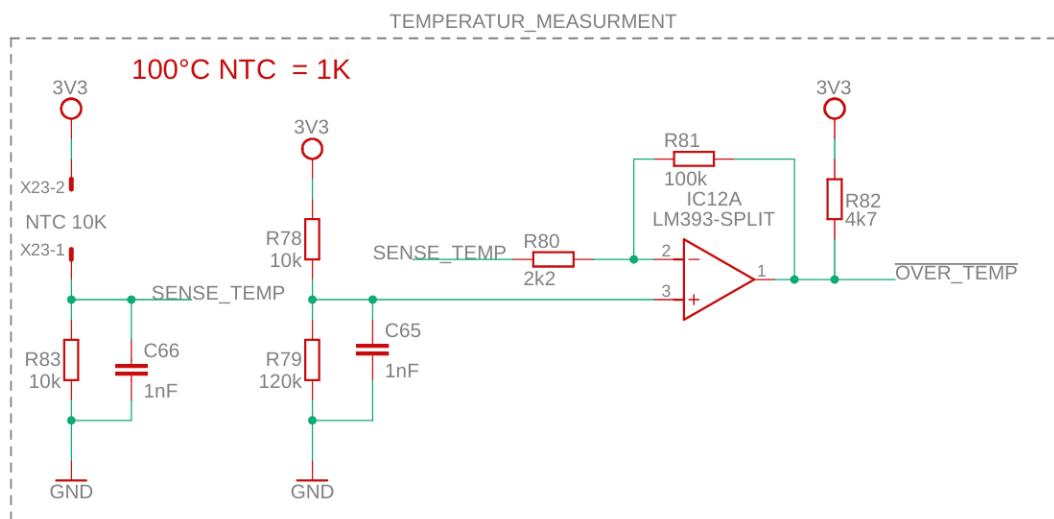


Abbildung 3.12.: Temperaturmessung RESC_V2 Schaltplan Seite 6

Der **NTC-Widerstand (R84)** ist direkt am Kühlkörper befestigt, um eine kontinuierliche Überwachung der Systemtemperatur zu gewährleisten. Die Lüftersteuerung erfolgt softwareseitig über eine **Hysterese**, wobei der Lüfter eingeschaltet wird, sobald die Temperatur **80°C** übersteigt, und wieder abgeschaltet wird, sobald sie unter **60°C** fällt. Falls die softwaregesteuerte Regelung nicht ausreicht, um den Temperaturanstieg zu begrenzen, greift eine hardwarebasierte Fehlererkennung bei **120°C** ein. Diese löst automatisch Schutzmaßnahmen aus, um die MOSFETs vor Überhitzung und möglichen Schäden zu bewahren.

Für das Ansteuern von zwei 40mm PC-Lüftern wird ein MOSFET-Treiber verwendet. Die Steuerung der Lüfter erfolgt über den Digital Output DO-FAN, der wahlweise im PWM-Betrieb oder im Digitalbetrieb arbeiten kann.

3.1.9. IO

Das Eingabe- und Ausgabe-Modul ermöglicht die Interaktion zwischen Benutzer und System durch die Erfassung von Eingabeelementen wie Taster sowie die Ausgabe von Systeminformationen auf einem Kommunikationsschnittstelle. Die Kommunikationsschnittstelle kann wahlweise über eine I²C oder UART Daten übertragen, wodurch eine flexible Integration in verschiedene Systemarchitekturen ermöglicht wird wie z.b LDC-Displays oder mehr Fortgeschrittene TFT Touch Tabletts.

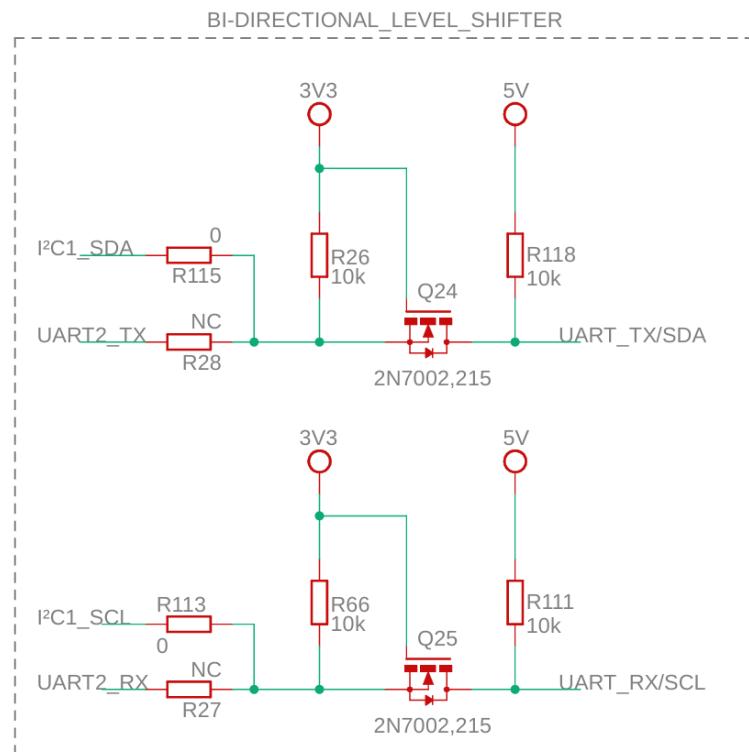


Abbildung 3.13.: Bidirektonaler-Levelshifter RESC_V2 Schaltplan Seite 7

Die Konfigurierung der Kommunikationsschnittstelle, erfolgt über das bestücken der entsprechenden Widerstände (R28, R27 oder R115, R113). Diese Anpassungsmöglichkeit ermöglicht die Anbindung einer Vielzahl kompatibler Displays und Peripheriegeräte. Zur Sicherstellung der Kompatibilität mit Arduino-Komponenten, die auf einem 5V-Logiklevel arbeiten, ist die Schnittstelle zusätzlich mit einem Bidirektionalem 3.3V/5V-Level-Shifter ausgestattet. Dieser gewährleistet eine zuverlässige Kommunikation zwischen den Komponenten, unabhängig vom verwendeten Logikpegel.

4. Layout

In diesem Kapitel wird das Layout der Leiterplatte (PCB) detailliert beschrieben. Ziel ist es, die wichtigsten Designentscheidungen, Regelwerke und Dimensionierungen zu dokumentieren, um die optimale Funktion und Zuverlässigkeit des Systems sicherzustellen. Das angefertigte Layout finden Sie im Anhang oder digital auf: github.com/leonreeh/E-Bike-ECU ⇒ RESC_V2 Hardware ⇒ RESC_V2 PCB.pdf

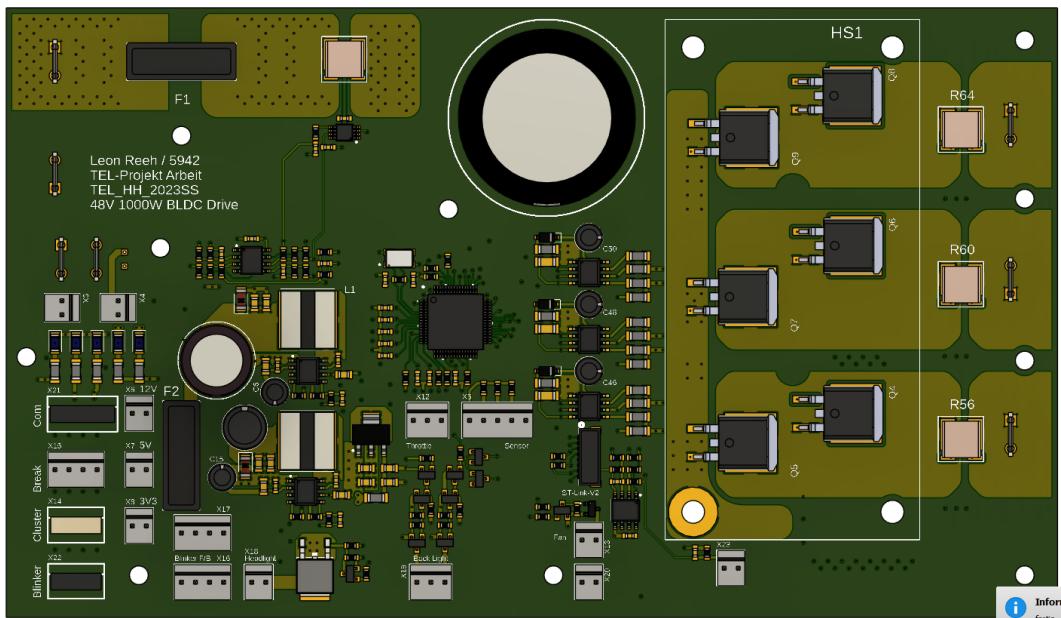


Abbildung 4.1.: RESC_V2 Leiterplatte

4.1. Design Regeln

Für die Leiterplattenentwicklung wurden folgende Design-Regeln definiert:

- Leiterbahnbreiten und -abstände:** Signal-Leitungen werden mit einer Breite von 8 mil ausgeführt, während Versorgungsleitungen im Kleinspannungsbereich (≥ 12 V) 20 mil breit sind.
- Via-Dimensionierung:** Die Vias werden gemäß den Vorgaben des PCB-Herstellers (JLCPCB) dimensioniert und besitzen eine Bohrung von 0,3 mm.
- Isolationsabstände:** Im Signalbereich beträgt der Isolationsabstand 16 mil (entspricht dem Zweifachen der Signal-Leitungsbreite), im Versorgungsnetz 20 mil und im Hochleistungsbereich 100 mil.
- Bauteilplatzierung:** Die Bauteile werden entsprechend ihren Schaltplangruppen angeordnet, sodass jede Gruppe ein schlüssiges Layout bildet. Dabei gilt die Regel: Komponenten, die zusammenarbeiten, werden auch physisch zusammen platziert. Beispielsweise werden RC-Filter (Widerstand und Kondensator) direkt nebeneinander und so nah wie möglich am zu filternden Eingang positioniert.
- Lagenaufteilung:** Der Top Layer (Layer 1) dient als Hauptbestückungslayer, während der Bottom Layer (Layer 6) als Hilfslayer genutzt wird, insbesondere für die Platzierung von Abblock-Kondensatoren nahe an den Versorgungspins des Mikrocontrollers.

4.2. PCB Dimensionierung

Die Platine besitzt eine Abmessung von 190×110 mm. Die einzige kritische Vorgabe bezüglich der Außenmaße war, dass sie kleiner als 210×210 mm bleibt, damit das PCB-Trägergehäuse auf ein 3D-Druckbett passt.

Die Befestigung erfolgt über mehrere M3-Schrauben, die das PCB sicher im Gehäuse fixieren. Besondere Aufmerksamkeit wurde auf die Befestigung großer und schwerer Bauteile gelegt, ebenso wie auf Bereiche, die unter mechanischer Belastung stehen, beispielsweise Stecker und Kabel. Diese Maßnahmen minimieren Biegestress und Vibrationen der Elektronik.

Die Design-Toleranzen richten sich nach den Fertigungsvorgaben von JLCPCB:

PCB-Outline: $\pm 0,2$ mm

Pad-Pad-Abstand: 6 mil

Pad-Via-Abstand: 6 mil

Die Steckerplatzierung wurde so gestaltet, dass eine einfache Handhabung ermöglicht wird. Stecker sind daher möglichst nah am Platinenrand positioniert, wobei die Orientierung der Platine innerhalb des Gehäuses berücksichtigt wurde. Beispielsweise sind:

Stecker für den Motor auf der Rückseite, da sich der Motor hinter der Elektronik befindet. Stecker für Bedienelemente auf der Vorderseite, da diese leicht zugänglich sein müssen. Eine Ausnahme bilden der Stecker für den Motor-Hall-Sensor sowie der Stecker für den Analog-Gasgriff. Diese wurden in der Nähe des Mikrocontrollers platziert, um eine optimale Signalanbindung zu gewährleisten.

4.3. Layer Design

Um unerwünschte Störeinflüsse zu minimieren und die Signalqualität zu gewährleisten, wurden die sechs zur Verfügung stehenden Layer in spezifische Signalgruppen aufgeteilt. Dies ermöglicht eine klare Trennung von Leistungs- und Signalleitungen mit dem Ziel elektromagnetische Störungen (EMI) zu reduzieren. Die Verteilung der Layer folgt einem logischen Aufbau, der sich an bewährten Industriepraktiken orientiert:

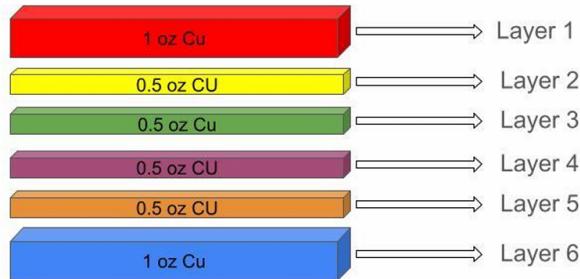


Abbildung 4.2.: 6-Layer PCB Stackup RESC_V2 Layout

Layer:

1. Baugruppen- und Power-Layer
2. Signal-Ground-Layer
3. Primär Digital-Signal-Layer
4. Analog-Signal-Layer
5. Kleinspannungs-Power-Layer (12V, 5V, 3.3V)
6. Primär Power-Layer

Layer 1 (Top-Layer – Hochstrom und Leistungsführung): Layer 1&6 tragen die Hauptlast der Leistungsversorgung. Da diese Layer die dickste Kupferschicht aufweist (1 oz Kupfer pro Layer), werden sie für die Führung der 48V Versorgung und der Motorphasenströme genutzt. Leiterbahnen, die große Ströme führen, wie z. B. die H-Brücken für den BLDC-Motor, werden auf diesem Layer platziert. Breite Leiterbahnen und großzügige Kupferflächen gewährleisten einen niedrigen Widerstand und vermeiden unerwünschte Erwärmung. Der Platzbedarf für Hochstromverbindungen wird optimiert, um Spannungsabfälle und Leistungsverluste zu minimieren.

Layer 2 (Signal Ground Layer): Layer 2 dient als dedizierte Massefläche für die gesamte Leiterplatte. Diese Fläche erfüllt zwei Hauptfunktionen: **Potentialreferenz:** Komponenten, die auf Massepotential arbeiten, können über Vias direkten Zugang zu diesem Layer erhalten, wodurch Massewege verkürzt werden und störungsfreier Betrieb gewährleistet ist. **Schirmung:** Die großflächige Massefläche direkt unter den aktiven Komponenten dient als elektromagnetischer Schirmschild. Sie verhindert, dass hochfrequente Signale oder Störfelder von den oberflächennahen Komponenten in tiefere Layer eindringen und dort empfindliche Signale beeinträchtigen.

Layer 3&4 (Signal-Layer – Analog- und Digitalsignale): Layer 3 und 4 sind für die Führung von Steuersignalen und Kommunikationsleitungen vorgesehen. Diese Layer wurden aufgrund ihrer Position in der Mitte des PCB-Stacks gewählt, da sie dort vor externen Störquellen am besten geschützt sind. **Layer 3 (Digital-Signale):** Hier werden digitale Steuersignale wie Kommunikationsleitungen (UART, I²C) geführt. Durch die Trennung von analogen und digitalen Signalen wird sichergestellt, dass hochfrequente Schaltimpulse die Integrität der analogen Signale nicht beeinträchtigen. **Layer 4 (Analog-Signale):** Dieser Layer führt empfindliche analoge Signale, z. B. Spannungs- und Strommessleitungen, um sicherzustellen, dass diese möglichst störungsfrei und fern von digitalen Schaltfrequenzen verlaufen.

Layer 5 (Kleinspannungs-Versorgungsebene): Layer 5 führt großflächige Versorgungsnetze für 12V, 5V und 3.3V. Diese Spannungen versorgen Mikrocontroller, Sensoren und andere Niederspannungskomponenten. Die großzügigen Kupferflächen minimieren Spannungsabfälle und ermöglichen eine gleichmäßige Spannungsverteilung über die gesamte Platine. Vias ermöglichen es, die Spannungen direkt an die benötigten Stellen zu führen, wodurch die Anzahl langer Leiterbahnen reduziert wird.

Layer 6 (Bottom-Layer – Hochstrom und Leistungsführung): Layer 6 fungiert als ergänzende Leistungsebene zu Layer 1 und trägt ebenfalls Hochstrompfade, jedoch auf der Unterseite der Platine. Er wird für die Rückführung von Leistungsströmen und zur Anbindung an Motorphasen verwendet. Durch die symmetrische Anordnung von Layer 1 und Layer 6 wird die Strombelastung der Platine gleichmäßig verteilt und die thermische Belastung reduziert.

Diese strukturierte Aufteilung gewährleistet eine klare Trennung von Leistungs- und Signalpfaden, reduziert EMV-Probleme und verbessert die allgemeine Zuverlässigkeit der Schaltung. Besondere Aufmerksamkeit wurde darauf gelegt, kritische Signale nicht unter Leistungsinduktivitäten oder großen Leistungstransistoren zu führen, um unerwünschte Koppelungen und Störungen zu vermeiden.

Die Leiterbahnbreite der Hochstromleitungen beträgt mindestens 700 mil, um Spannungsabfälle und eine übermäßige Erwärmung zu minimieren. Der MOSFET-Heatsink ist zusätzlich mit der Power-Ground-Fläche verbunden, um die Wärmeableitung zu verbessern. Thermisch kritische Bauteile, wie beispielsweise Schaltregler, sind mit speziellen Kupferflächen versehen. Für eine optimale Wärmeabfuhr wurden In-Pad-Vias integriert, um die Wärme effizient von den Bauteilen zur Massefläche abzuleiten.

5. Software

Dieses Kapitel bietet eine detaillierte Beschreibung der Softwarearchitektur und geht auf die wichtigsten Module ein, die die Funktionalität des Systems gewährleisten. Von der Initialisierung der Hardware über die Motorsteuerung bis hin zur Benutzerinteraktion deckt die Software eine Vielzahl von Aufgaben ab, die in den folgenden Unterkapiteln erläutert werden.

Den vollständigen Quellcode zu diesem Projekt finden Sie auf GitHub unter github.com/leonreeh/E-Bike-ECU ⇒ BLDC-ESC-Firmware.

main.c/h Board Initialisierung, Mainloop und state machine

BLDC.c/h Interpretation des Hall Sensoren und setzen der Mosfet-Brücke

liquidcrystal_i2c.c/h Kommunikation mit dem LCD display

Mymath.c/h mathematische Hilfsfunktionen und Berechnungen

stm32f4xx_it.c/h Timer Interrupt-Routine, für zeitabhängige Funktionen (zb Geschwindigkeitsberechnung)

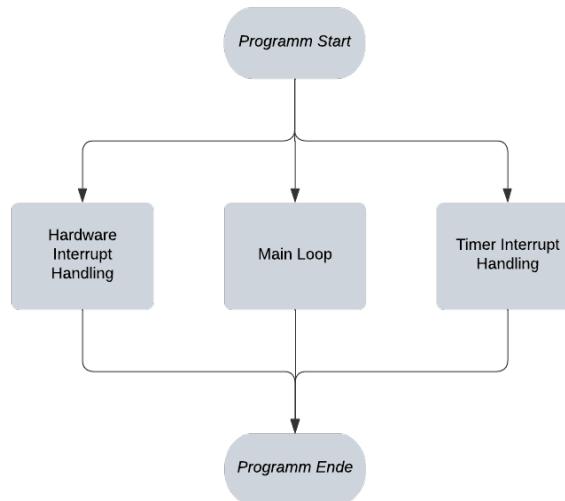


Abbildung 5.1.: Prozess-Parallelität

5.1. Prozesse & Softwarearchitektur

Das gesamte Programm wird in vereinfachter Form durch drei parallel arbeitende Prozesse gesteuert: der **Main Loop**, den **Timer Interrupt** und den **Hardware Interrupt**. Diese Prozesse übernehmen unterschiedliche Aufgaben, um eine effiziente und stabile Steuerung sicherzustellen.

1. Der Hardware Interrupt wird für zeitkritische Ereignisse verwendet und reagiert unmittelbar auf bestimmte Signale. Diese Funktionen beinhalten: Interpretation der Motor-Hall-Sensoren (Ermittlung der Motorposition) Erkennung des Hinterrad-Bremshebels (Einleitung des Bremsvorgangs) Verarbeitung des Hardware Fault Signals (Kritische Fehlererkennung).

2. Der Timer Interrupt arbeitet mit festen Zeitintervallen und dient der periodischen Ausführung zeitgesteuerter Aufgaben. Wie der Berechnung der Drehzahl (alle 100 ms), Übertragung von Daten an das Display (alle 500 ms), Software Fault Timeout (30s).

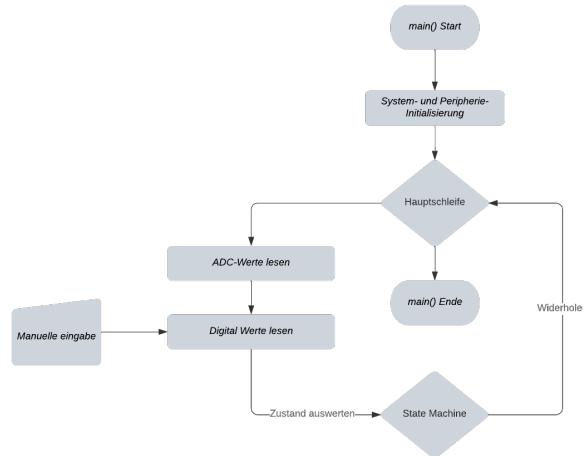


Abbildung 5.2.: Ablauf-Diagramm Main Loop

3. Die Main Loop bearbeitet zyklisch alle nicht zeitkritischen Aufgaben. Dies beinhaltet: Einlesen&Konvertieren der ADC-Messwerte, Abfrage der Bedientaster, Verarbeitung der Zustandsmaschine (State Machine), Setzen der digitalen Ausgänge (z. B. Beleuchtung, Lüftersteuerung).

5.1.1. Zustandsautomat/State Machine

Um die Vielzahl der Funktionen des Mikrocontrollers effizient und sicher zu verwalten, wurde in der Main Loop ein Zustandsautomat (State Machine) implementiert. Dieses Prinzip basiert auf der Definition von klaren Zuständen und Übergängen zwischen diesen, abhängig von bestimmten Bedingungen oder Ereignissen. Ein Zustandsautomat ist ein Kontrollflussmodell, das aus folgenden Elementen besteht:

Zustände: Definieren die aktuellen Betriebsmodi oder Aufgaben des Systems (z. B. Initialisierung, Normalbetrieb, Fehlerzustand).

Übergänge: Beschreiben die Bedingungen, unter denen das System von einem Zustand in einen anderen wechseln kann.

Ereignisse oder Bedingungen: Lösen die Übergänge zwischen Zuständen aus (z. B. Sensorwerte, Timer, Benutzereingaben).

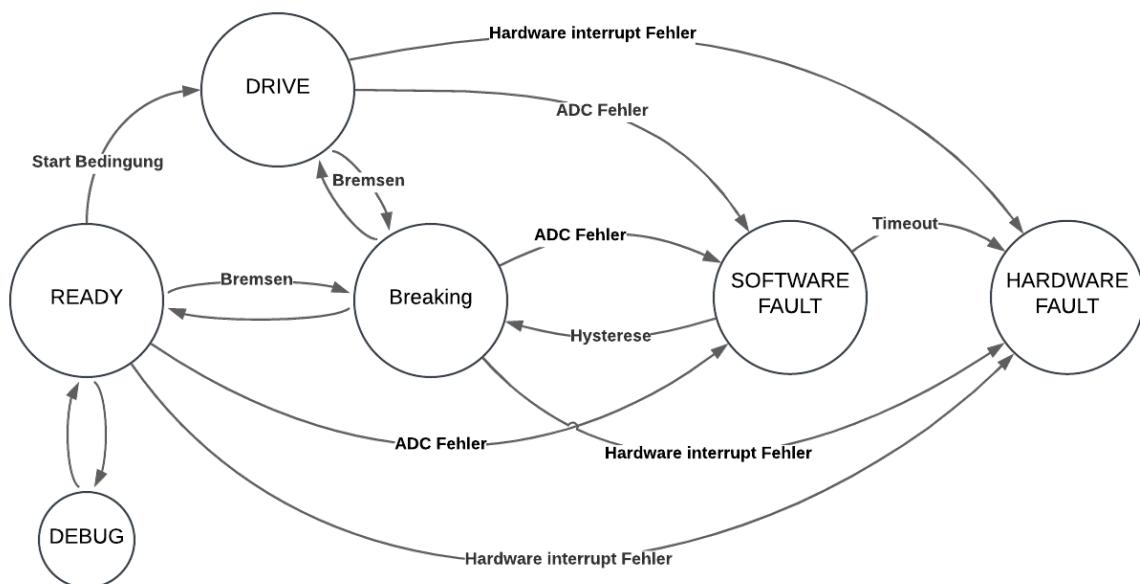


Abbildung 5.3.: Zustandsautomaten Diagramm

Die in Abbildung 5.3 dargestellten Zustände:

READY:

Nach Abschluss der Systeminitialisierung wechselt der Controller in den „Standby“-Zustand. In diesem Zustand bietet das Gerät folgende Funktionen: Steuerung der Beleuchtung, Initialisierung des Motors, Aktivierung der Lüfter bei Temperaturen ($T \geq 60^\circ\text{C}$).

DRIVE:

Dieser Zustand repräsentiert den Fahrbetrieb. Mit den Funktionen: Steuerung der Beleuchtung, Volle Motorleistung, Aktivierung der Lüfter bei $T \geq 60^\circ\text{C}$.

BRAKING:

Dieser Zustand repräsentiert das aktive Motorbremsen. Zustandsfunktionen: Steuerung der Beleuchtung, Motor im Bremsbetrieb, Aktivierung der Lüfter bei $T \geq 60^\circ\text{C}$.

SOFTWARE FAULT:

In diesem Zustand wird der Motor gestoppt, Beleuchtungselemente werden deaktiviert, um Last zu reduzieren.

HARDWARE FAULT:

Dieser Zustand gilt als kritisch, er kann nur durch Aus- und erneutes Einschalten des Geräts verlassen werden und Blockiert alle Funktionen wie Motor und Beleuchtungssteuerung. Optional kann das Gerät sich selbstständig abschalten durch öffnen des Selbsthalterelais.

Übergänge wie in Abbildung 5.3 dargestellte:

Start:

Bedingung: Das Gerät befindet sich im READY-Zustand, Drehzahl = 0, Der Nutzer drückt den Start-Taster oder bewegt den Gas-Griff, woraufhin das Gerät in den DRIVE-Zustand wechselt und die Fahrt beginnt.

Bremsen:

Der Nutzer betätigt während der Fahrt den Bremshebel, wodurch ein Hardware-Interrupt ausgelöst wird. Der Motor wechselt vom Antriebsmodus in den Bremsmodus. Sobald der Motor bis zum Stillstand abgebremst ist, wechselt das Gerät zurück in den READY-Zustand. Wird der Bremshebel losgelassen, bevor die Drehzahl 0 erreicht, kehrt das Gerät in den DRIVE Zustand zurück.

ADC-Fehler:

Tritt ein ADC-Messwert über einen definierten Grenzwert (z. B. Kühlkörpertemperatur $> 100^\circ\text{C}$), betritt das System den SOFTWARE FAULT-Zustand. Dieser Zustand kann nur verlassen werden in dem die Temperatur wieder unter 80°C fällt. Dies passiert über den BRAKE Zustand, um versehentliches Start des Motors zu vermeiden.

Timeout:

Mit dem betreten des SOFTWARE FAULT Zustands wird ein 30s Timer gestartet, wenn dieser Zustand nicht vor ablaufen dieses Timers verlässt. Wechselt es den HARDWARE FAULT-Zustand.

Hardware-Interrupt-Fehler: Beispiel Die Spannung Des Akkus fällt unter 42.9V löst die Hardware Messschaltung (siehe Schaltplan Abbildung 3.3) aus und betätigt damit den Fault Interrupt-Pin (siehe Schaltplan Abbildung 3.4).

5.2. Source Code

Um die Dokumentation kompakt und übersichtlich zu gestalten, konzentriert sich dieser Abschnitt ausschließlich auf die Funktionen des Mikrocontrollers, die für die Hauptlogik und Steuerung des Systems entscheidend sind. Allgemeine Quellcode-Elemente wie die Konfiguration von GPIOs, Timern oder Middleware werden bewusst vernachlässigt. Die hier erklärten Source Code ausschnitte finden sie im Anhang unter Source Code

5.2.1. Main.c/h

Main Loop siehe Anhang Source Code 15.2

implementiert eine Endlosschleife (while(1)), die eine Zustandsmaschine zur Steuerung verschiedener Betriebszustände verwendet. In jedem Schleifendurchlauf werden zunächst Eingaben durch readADCs(), readDI() erfasst und mit doADCs() setDO() verarbeitet. Anschließend wird ein switch-Statement basierend auf der aktuellen Systemzustandsvariable (STATE) ausgeführt. Je nach Zustand (READY, DRIVE, BREAK, etc) werden entsprechende Funktionen aufgerufen, um den Systemstatus zu verwalten, beispielsweise ready(). In fehlerhaften Zuständen (SWFAULT und HWFAULT) werden die digitalen Ausgänge (DO) mit resetDO() zurückgesetzt, während in regulären Betriebszuständen setDO() aktiviert wird. Zusätzlich wird ein LCD-Display alle 500 ms aktualisiert, wenn der Zähler timcc den Wert 5 erreicht. Am Ende jedes Schleifendurchlaufs wird eine Verzögerung von 25 ms mit HAL_Delay(25); eingefügt, um den Prozessor zu entlasten.

readADCs() siehe Anhang Source Code 15.3

Die Funktion readADCs liest analoge Werte von drei unterschiedlichen Quellen (Spannung, Strom und Temperatur) ein und speichert die Ergebnisse in einem globalen Array ADC_VAL.

doADCs() siehe Anhang Source Code 15.4

Die Funktion doADCs analysiert die erfassten ADC-Werte, überprüft Grenzwerte und steuert Systemkomponenten wie den Lüfter basierend auf den Messwerten. Sie verwaltet zudem Zustandsübergänge und erkennt Fehlerzustände.

readDI() siehe Anhang Source Code 15.5

Die Funktion readDI liest den Zustand von vier digitalen Eingängen (Tastern) ein und aktualisiert ein globales Array but, das diese Zustände speichert.

setDO() siehe Anhang Source Code 15.6

Die Funktion setDO setzt digitale Ausgänge basierend auf dem aktuellen Zustand der Taster (gespeichert im globalen Array but) und steuert damit Systemkomponenten wie Licht, Blinker und einen Debug-Modus.

ready() siehe Anhang Source Code 15.7

Die Funktion ready prüft den Gasgriff und initialisiert den Motor, falls die Startbedingungen erfüllt sind. Wenn der Gasgriff eine bestimmte Schwelle überschreitet, wird das System in den Zustand DRIVE versetzt und der BLDC-Motor gestartet.

drive() siehe Anhang Source Code 15.8

Die Funktion drive liest den Gasgriffwert ein, berechnet den entsprechenden Duty-Cycle und aktualisiert diesen, um die Motorgeschwindigkeit während des Fahrmodus zu steuern. Bei einem Fehler in der ADC-Konvertierung wird der Duty-Cycle auf einen sicheren Wert gesetzt, und ein Fehlerzustand ausgelöst.

breaking() siehe Anhang Source Code 15.9

Die Funktion breaking setzt den Motor in den Bremsmodus, indem sie die PWM-Signale stoppt und die Motorphasen kurzschließt, um die Bewegung zu verlangsamen oder zu stoppen

swfault() siehe Anhang Source Code 15.10

Die Funktion swfault wird aufgerufen, wenn ein Softwarefehler erkannt wird. Sie setzt das System in einen sicheren Zustand durch Aktivierung des Bremsmodus und überwacht, ob der Fehlerzustand über einen definierten Timeout hinaus anhält.

hwfault() siehe Anhang Source Code 15.11

Die Funktion hwfault behandelt einen dauerhaften Hardwarefehlerzustand. Sie stoppt das System vollständig und zeigt eine Fehlermeldung an, während sie auf einen Neustart (Power-On-Reset) wartet.

HAL_GPIO_EXTI_Callback() siehe Anhang Source Code 15.12

Die Funktion HAL_GPIO_EXTI_Callback wird aufgerufen, wenn ein externer GPIO-Interrupt ausgelöst wird. Sie verarbeitet verschiedene Arten von Interrupts, darunter Brems-, Hardwarefehler- und Hall-Sensor-Interrupts, abhängig vom Zustand des Systems und dem auslösenden Pin.

handleHardwareFaultInterrupt() siehe Anhang Source Code 15.13

Wechselt bei Hardwarefehlern in den entsprechenden Fehlerzustand. Optional: Fehlerdiagnose kann implementiert werden.

handleBreakInterrupt() siehe Anhang Source Code 15.14

Die Funktion handleBreakInterrupt verarbeitet den Interrupt, der durch das Betätigen oder Loslassen des Bremshebels ausgelöst wird. Sie wechselt den Systemzustand zwischen BREAK und READY.

handleHallSensorInterrupt() siehe Anhang Source Code 15.15

Die Funktion handleHallSensorInterrupt verarbeitet Interrupts, die durch Änderungen an den Hall-Sensoren ausgelöst werden. Sie zählt die Rotationsschritte des Motors, ermittelt die aktuelle Rotorposition und steuert die Motorphasen basierend auf trapezförmiger Kommutierung.

5.2.2. BLDC.c/h

hallState() siehe Anhang Source Code 15.16

Die Funktion hallState interpretiert die Zustände der Hall-Sensoren (ein Array mit drei Werten) und ermittelt die aktuelle Position des Rotors in Bezug auf die elektrische Kommutationsstufe (zwischen 0 und 5).

commutator() siehe Anhang Source Code 15.17

Die Funktion commutator steuert die drei Phasen eines BLDC-Motors basierend auf der Kommutatorposition, dem gewünschten Duty-Cycle (PWM-Signal) und der Drehrichtung des Motors.

initBLDC() siehe Anhang Source Code 15.18

Die Funktion initBLDC initialisiert den BLDC-Motor basierend auf den aktuellen Hall-Sensor-Zuständen.

BLDCbreak() siehe Anhang Source Code 15.19

Die Funktion BLDCbreak versetzt den BLDC-Motor in den Bremsmodus, indem sie alle Phasen des Motors kurzschließt und die PWM-Ausgabe deaktiviert.

5.2.3. liquidcrystal_i2c.c/h

lcd struct siehe Anhang Source Code 15.20

Die Struktur lcd_ar wird verwendet, um Daten und Cursorpositionen für die Aktualisierung eines LCD-Displays zu speichern. Sie enthält die wichtigsten Systemparameter sowie deren Positionen auf dem Display.

HD44780_SetCursor() siehe Anhang Source Code 15.21

Die Funktion HD44780_SetCursor setzt die Position des Cursors auf einem HD44780-kompatiblen LCD-Display. Die Position wird durch die Spalte (col) und die Zeile (row) angegeben.

HD44780_PrintStr() siehe Anhang Source Code 15.22

Die Funktion HD44780_PrintStr schreibt eine Zeichenkette (String) auf ein HD44780-kompatibles LCD-Display.

Init_lcd_ar() siehe Anhang Source Code 15.23

Die Funktion Init_lcd_ar initialisiert eine Instanz der Struktur lcd_ar mit Standard-Cursor-Positionen und voreingestellten Anzeigewerten auf einem HD44780-kompatiblen LCD-Display.

update_lcd_val() siehe Anhang Source Code 15.24

Die Funktion update_lcd_val aktualisiert sowohl die Daten in einer lcd_ar-Struktur als auch die Anzeige auf einem HD44780-kompatiblen LCD-Display mit neuen Sensorwerten.

5.2.4. Mymath.c/h

pid struct siehe Anhang Source Code 15.25

Die Struktur pid_f_t wird verwendet, um die Parameter und Zustandsvariablen eines PID-Reglers (Proportional-Integral-Differential-Regler) zu speichern.

map() siehe Anhang Source Code 15.26

Die Funktion map skaliert einen Wert aus einem Bereich (Eingabebereich) in einen anderen Bereich (Zielbereich). Dies wird häufig verwendet, um Werte zwischen unterschiedlichen Skalen umzuwandeln, z. B. ADC-Werte in eine PWM-Skala.

Formel:

$$\text{mappedvalue} = \left(\frac{x - \text{in}_{\min}}{\text{in}_{\max} - \text{in}_{\min}} \right) \times (\text{out}_{\max} - \text{out}_{\min}) + \text{out}_{\min}$$

adc_volt() siehe Anhang Source Code 15.27

Die Funktion adc_volt berechnet die Eingangsspannung (Vin) basierend auf einem Rohwert des ADC (val). Sie verwendet ein Spannungsteiler-Netzwerk, um die tatsächliche Spannung am Eingang zu bestimmen.

Formel:

$$V_{adc} = \frac{val}{4095} \times V_{cc} ; V_{in} = V_{adc} \times \left(\frac{R_1 + R_2}{R_2} \right)$$

adc_cur() siehe Anhang Source Code 15.28

Die Funktion adc_cur berechnet den Strom in Ampere basierend auf einem Rohwert des ADC (val), einem Shunt-Widerstand und einem Verstärkungsfaktor.

Formel:

$$V_{adc} = \frac{val}{4095} \times V_{cc} ; V_{adc_adjusted} = V_{adc} - 1,65V ; V_{shunt} = \frac{V_{adc_adjusted}}{\text{amplification_factor}} ; \text{current} = \frac{V_{shunt}}{R_{shunt}}$$

adc_temp() siehe Anhang Source Code 15.29

Die Funktion adc_temp berechnet die Temperatur in Grad Celsius basierend auf einem Rohwert des ADC (val), der Spannungsteilerkonfiguration eines NTC-Thermistors (R1) und der Steinhart-Hart-Gleichung (vereinfachte Form mit Beta-Parameter).

Formel:

$$V_{adc} = \frac{val}{4095} \times V_{cc} ; R_1 = R_2 \times \left(\frac{V_{cc}}{V_{adc}} - 1.0 \right) ; T = \frac{1.0}{\frac{1}{T_0} + \frac{1}{B} \times \log(\frac{R_1}{R_0})}$$

rpm_tokmh() siehe Anhang Source Code 15.30

Die Funktion rpm_tokmh berechnet die Geschwindigkeit in Kilometern pro Stunde (km/h) basierend auf der Motordrehzahl (RPM) und dem Radumfang(U).

Formel:

$$m/min = RPM \times U; kmh = \frac{m/min \times 60min}{1000m}$$

5.2.5. stm32f4xx_it.c/h

TIM2_IRQHandler() siehe Anhang Source Code 15.31

Die Funktion TIM2_IRQHandler ist der Interrupt-Handler für den Timer TIM2, der mit einer Frequenz von 100 ms ausgelöst wird. Sie führt mehrere systemkritische Aufgaben aus, einschließlich der Berechnung der Drehzahl (RPM) und Geschwindigkeit sowie der Verwaltung von Zeitmessungen und Zuständen. Motor Sensor interrupt zähler (hallCC), Motor Polpaare (24).

Formel:

$$x = hallCC \times 10 ; rev = \frac{x}{24} ; RPM = rev \times 60$$

5.3. Controller Konfiguration

Controller Pinning und Konfiguration basiert auf dem opensource VESC 6.4 Schaltplan und wurden mit Hilfe des STM32CubeIDE Configuratos erweitert.

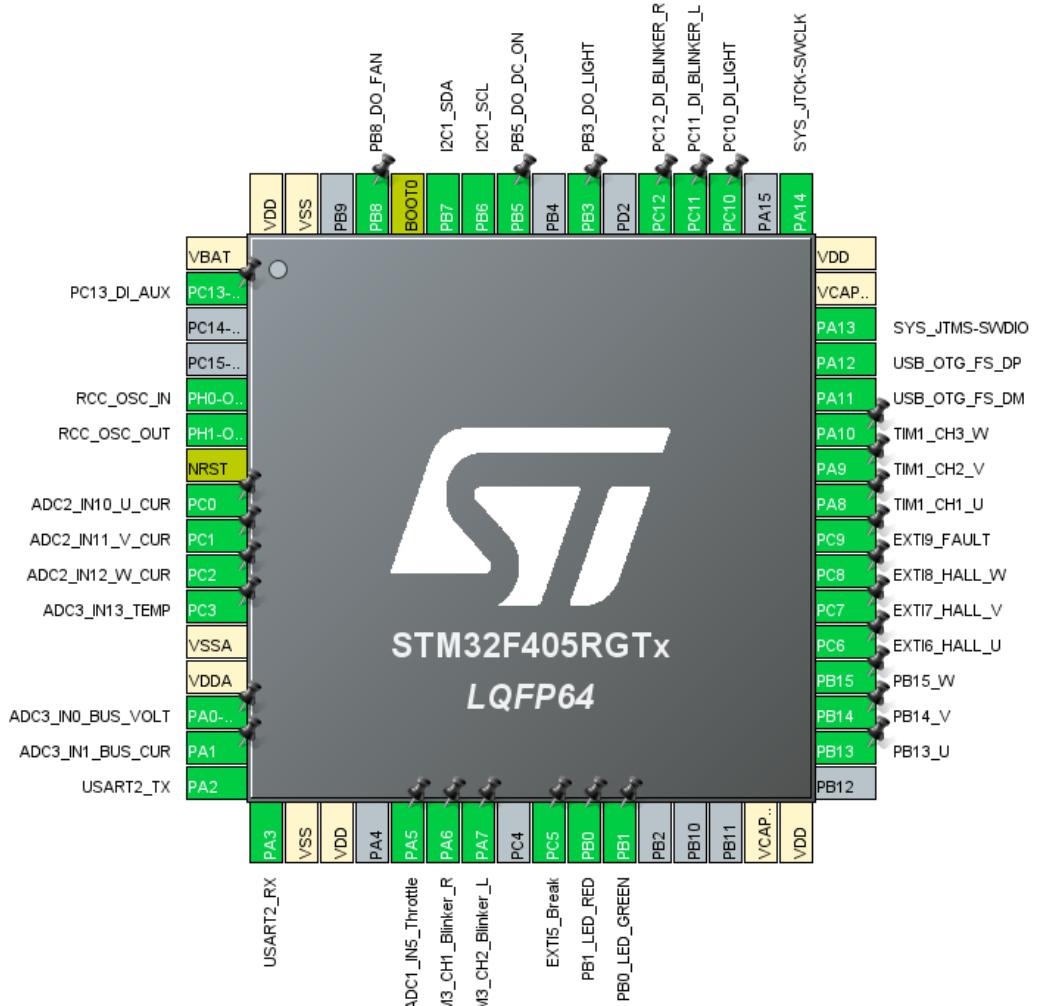


Abbildung 5.4.: Pin Konfiguration

Timer1 (10 KHz Motor PWM):

Counter Settings

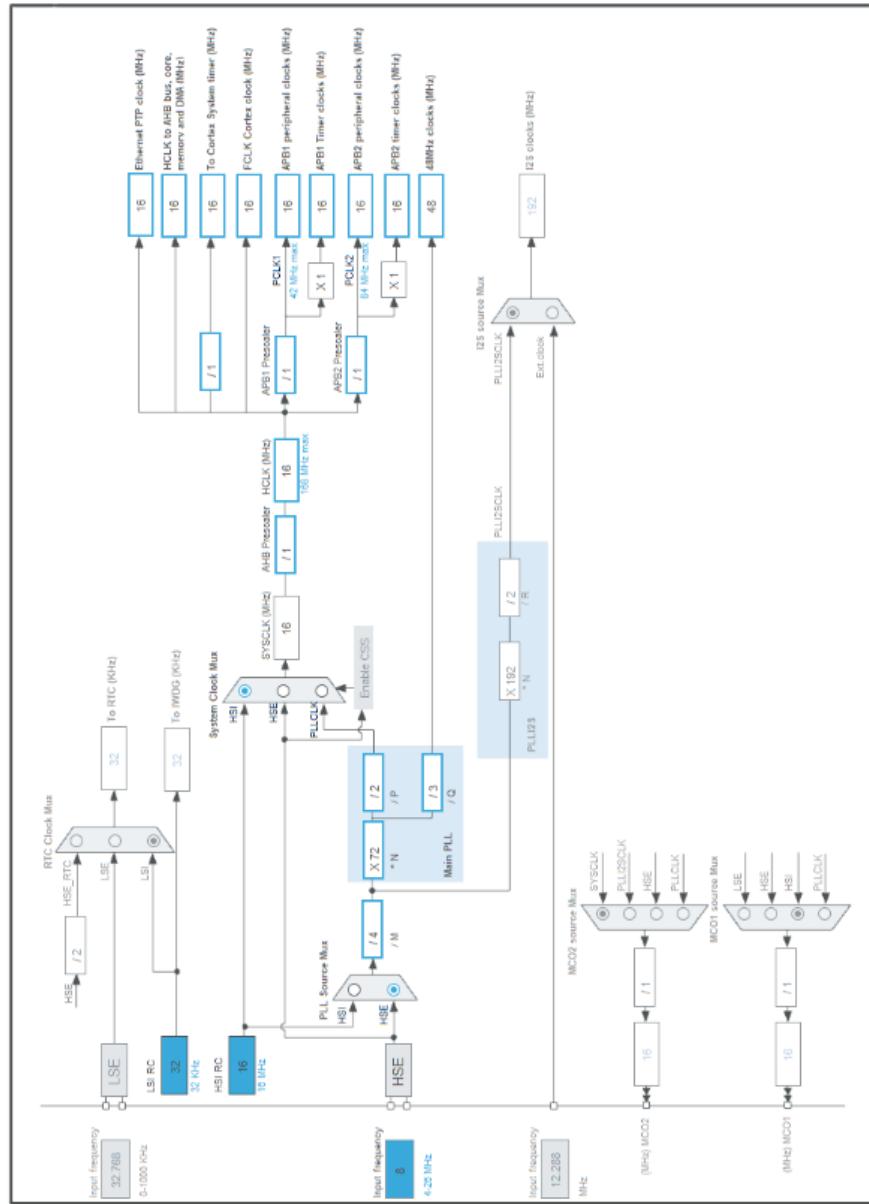
Prescaler (PSC - 16 bits val... 15	Up
Counter Mode	99
Counter Period (AutoReload... 99	No Division
Internal Clock Division (CKD)	No Division
Repetition Counter (RCR - ... 0	Disable
auto-reload preload	Disable

Timer2 (100 ms Interrupt):

Counter Settings

Prescaler (PSC - 16 bits val... 15999	Up
Counter Mode	36
Counter Period (AutoReload... 99	No Division
Internal Clock Division (CKD)	No Division
Repetition Counter (RCR - ... 0	Disable
auto-reload preload	Disable

System Clock:



Timer3 (1Hz Blinker PWM):

Counter Settings

Prescaler (PSC - 16 bits val... 15999	Up
Counter Mode	499
Counter Period (AutoReload... 499	No Division
Internal Clock Division (CKD)	No Division
Repetition Counter (RCR - ... 0	Disable
auto-reload preload	Disable

Abbildung 5.5.: System Clock Konfiguration

6. Gehäuse

Die STL-Dateien der Gehäuse sind auf der Projekt-GitHub-Repository github.com/leonreeh/E-Bike-ECU ⇒ Gehäuse STL verfügbar.



Abbildung 6.1.: Gehäuse

6.1. Mechanische Eigenschaften

Material: ASA (Acrylnitril-Styrol-Acrylat) wurde aufgrund seiner robusten mechanischen Eigenschaften gewählt. Es bietet hohe Temperatur- und UV-Beständigkeit und lässt sich einfach weiterverarbeiten. Das Material ist besonders geeignet für Anwendungen, die ständiger Witterung oder mechanischer Belastung ausgesetzt sind.

Dimensionen: Die Außenmaße betragen $272 \times 140 \times 77$ mm, was ausreichend Platz für die notwendigen Elektronikkomponenten bietet und eine kompakte Integration ermöglicht.

6.2. Konstruktion

Das Gehäuse besteht aus vier modularen Einzelteilen: **Platinenträger:** Hier wird die Elektronik mit Hilfe von M3 Schrauben befestigt. **Vorder- und Rückseite:** Diese Teile werden mithilfe von 3×60 mm Nägeln und einem speziellen ASA-Kleber dauerhaft mit dem Platinenträger verbunden. Dies sorgt für eine robuste und langanhaltende Fixierung. **Deckel:** Der Deckel wird mit M4-Schrauben befestigt, was einen einfachen Zugriff auf die Elektronik bei Wartungsarbeiten ermöglicht.

Zur Anbindung von verschraubbaren Teilen werden sogenannte **Heatset-Inserts** verwendet. Diese bestehen aus Messing und können mithilfe eines Lötkolbens in das Kunststoffmaterial eingebracht werden. Dadurch entsteht ein langlebiges und stabiles Metallgewinde, das sich ideal für wiederholtes Verschrauben eignet. M4 Inserts: Zum sicheren Verschrauben des Deckels. M3 Inserts: Für die Montage der Platine und Lüfter.

6.3. Gehäuse Komponenten



Abbildung 6.2.: Front

Lufteinlässe: Passive Kühlung durch strategisch platzierte Lufteinlässe für optimale Wärmeableitung. Anschlussbuchse: Eine XT90-Buchse dient zur Verbindung mit dem Akku. Diese ist für hohe Ströme ausgelegt und sorgt für sichere Energieübertragung. Kabelzugentlastung: Eine PG17-Zugentlastung stellt sicher, dass die Leitungen für Bedienelemente und das Display sicher fixiert sind und keine mechanischen Belastungen auf die internen Verbindungen wirken. Befestigung: Die Frontseite wird mit 5 Nägeln am Platinenträger montiert.

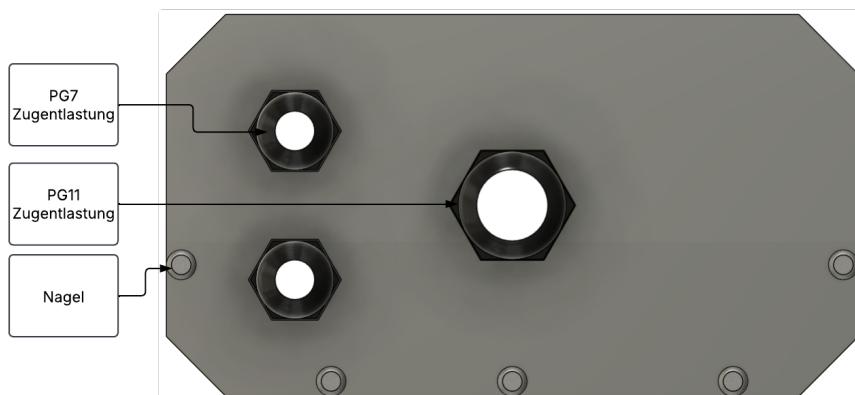


Abbildung 6.3.: Rückseite

Kabelzugentlastungen: PG11 für die Motorleitungen, PG7 für die Heckbeleuchtung, PG7 für die Heckblinker, Befestigung: Auch die Rückseite wird mit 5 Nägeln sicher am Platinenträger fixiert. Heatset Inserts: M4 Inserts für die Montage des Deckels, M3

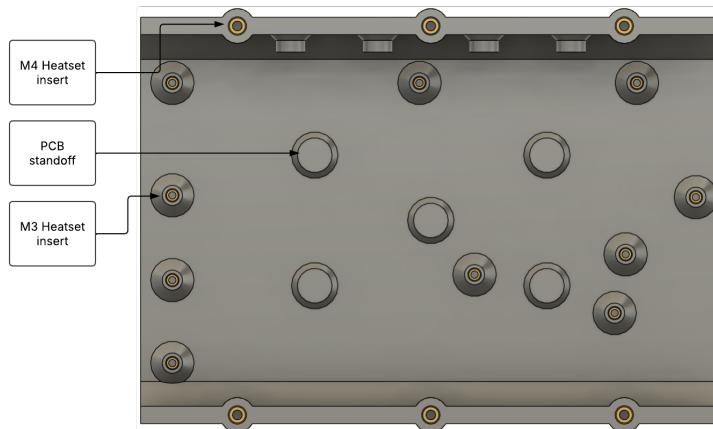


Abbildung 6.4.: Platinenträger

Inserts für die sichere Befestigung der Platine. PCB-Standoffs: Integrierte PCB-Standoffs sorgen für zusätzliche Stabilität und unterstützen die Elektronik gegen Verbiegen. Diese Konstruktion minimiert mechanische Belastungen auf die Platine.

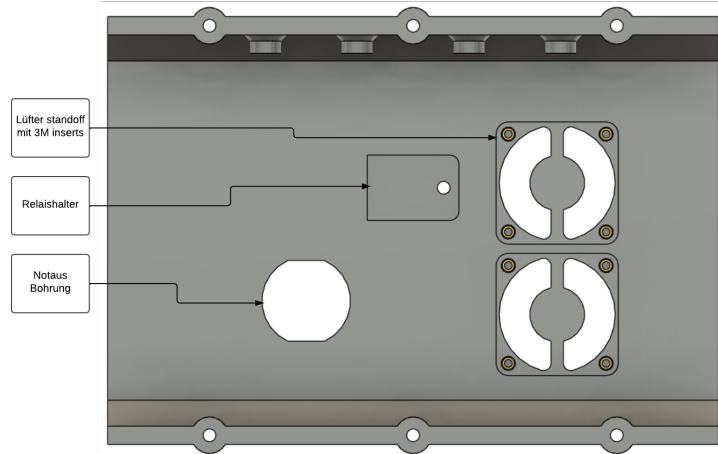


Abbildung 6.5.: Deckel

Lüfter-Standoff: Ein erhöhter Bereich zur direkten Positionierung eines Lüfters über dem Kühlkörper, um die Wärmeableitung zu verbessern. Relaishalter: Speziell designet Aussparungen sorgen für die sichere Positionierung und Fixierung des Relais. Befestigungsbohrung: Eine Bohrung mit Registrierung ermöglicht die Montage eines Not-Aus-Schalters an einer zugänglichen Stelle.

Gehäuse Stückliste:

Name	Stückzahl	Beschreibung
Platinenträger	1	STL file für 3D druck
Deckel	1	STL file für 3D druck
Front	1	STL file für 3D druck
Rückseite	1	STL file für 3D druck
M3S Gewindeeinsatz	20	M3 gewinde insert
M4 Gewindeeinsatz	6	M4 gewinde insert
XT90E-M	1	Hochstrom Einbaustecker
PG17	1	Zugentlastung
PG11	1	Zugentlastung
PG7	2	Zugentlastung

Teil III.

Entwicklungsprozess und Relevante Konzepte

In diesem Kapitel widme ich mich dem **Entwicklungsprozess** und den **relevanten Konzepten**, die für die Realisierung des Projekts von zentraler Bedeutung sind. Ziel ist es, einen Einblick in die methodische Herangehensweise und die zugrunde liegenden Entscheidungen zu geben, die den Weg von der ersten Idee bis hin zu einem funktionalen Prototypen geebnet haben.

Der Entwicklungsprozess folgt einer klar strukturierten Methodik – von der Planung über die Designphase bis hin zur Umsetzung. Zentrale Herausforderungen, wie die Auswahl geeigneter Hardware-Komponenten und die Entwicklung eines zuverlässigen Steuerungssystems, wurden durch einen systematischen Ansatz und eine speziell auf das Projekt zugeschnittene Strategie bewältigt. Dabei wurde das komplexe Gesamtsystem in kleinste funktionale Einheiten unterteilt, die in einzelnen Entwicklungsphasen überprüft wurden. Ziel war es, Designfehler frühzeitig zu erkennen und die Integration in das Gesamtsystem zu erleichtern.

7. Entwicklungsprozess

Planung Nach der Auswahl des Projektthemas begann die Recherche zu den grundlegenden Aspekten des E-Bikes. Wesentliche Fragestellungen, die beantwortet werden mussten, waren:

Welcher Motor und wie wird er angesteuert?

Im Bereich der E-Mobilität ist die Wahl der Motorart eindeutig: Der **Brushless DC Motor (BLDC)**. Aufgrund seines wartungsfreien Designs und seiner hohen Leistungsdichte eignet er sich ideal für Antriebssysteme und wird in verschiedensten Anwendungen von E-Bikes bis hin zu Elektroautos eingesetzt.

Für dieses Projekt fiel die Entscheidung auf einen diskreten BLDC-Hub-Motor. Ein Hub-Motor ist direkt in der Radnabe integriert, wodurch keine Änderungen am Fahrradrahmen erforderlich sind. Diese Bauweise erleichtert die Integration in bestehende Fahrräder erheblich. Zudem wird der Kettenantrieb des Fahrrads nicht belastet, da die Kraftübertragung direkt über das Rad erfolgt.

Welcher Akku wird verwendet?

Die am häufigsten in E-Bikes vertretenen Akkusysteme sind 7S (24V), 9S (32V) und 13S (48V). Mit einer Zielsetzung von 1 kW Leistung fiel die Wahl auf ein 48V-System. Diese Entscheidung ermöglicht es, den Stromfluss gering zu halten und dadurch die ohmschen Verluste in Leitungen und Steckverbindungen zu minimieren.

Basierend auf dieser Spannung konnten erste Hardware-Anforderungen dimensioniert werden, beispielsweise der Eingangsspannungsbereich des Netzteils, um eine stabile und effiziente Energieversorgung sicherzustellen. Siehe Anhang Entwicklung 11.1

Welche Zusatzfunktionen?

Zusätzliche Features wie Beleuchtung und ein Display sollten integriert werden, um den Funktionsumfang zu erweitern und die Benutzererfahrung zu verbessern.

Designphase Für die Motorsteuerung wurde das Open-Source-„VESC-Projekt“ als Basis gewählt. Dieses ist bekannt für seine robuste Hardware für Akku betriebene Antriebssysteme im Kilowatt Bereich, sowie eine breite und modulare Softwarearchitektur. Durch diese Entscheidung wurde eine der komplexesten Herausforderungen des Projekts – die Auswahl eines geeigneten Mikrocontrollers – erheblich erleichtert. Die Vielfalt an verfügbaren Mikrocontrollern, z. B. von Herstellern wie Renesas, STMicro und NXP, und die unterschiedlichen Spezifikationen wie Speichergröße, 16/32-Bit-Systeme, Anzahl von Interrupts und ADC-Pins, machten diese Entscheidung besonders anspruchsvoll.

Aus dem VESC 6.4-Referenzdesign wurde die Mikrocontroller-Auswahl sowie dessen Pinbelegung übernommen, während der restliche Schaltplan und die gesamte Software vollständig eigenständig entwickelt wurden.

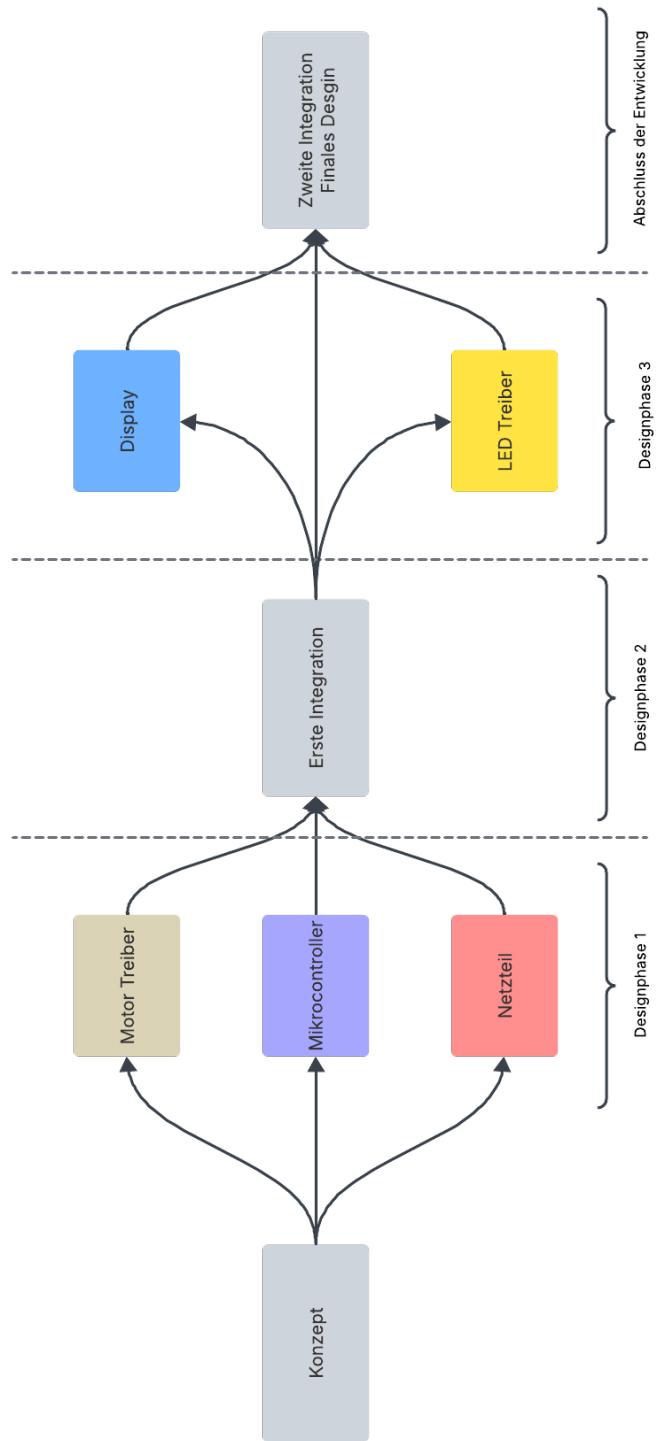


Abbildung 7.1.: Designphasen

8. Designphasen

8.1. Designphase 1

Mikrocontroller Für die Inbetriebnahme des Mikrocontrollers wurde eine kompakte 2-Layer-Leiterplatte entworfen, die alle notwendigen Komponenten enthält, um eine funktionale und stabile Umgebung für die Softwareentwicklung zu gewährleisten. Die Leiterplatte umfasst: **3.3V-Versorgung** zur Stromversorgung des Mikrocontrollers. **Clock-Oszillator** für die Systemtaktung. **Debug-LEDs**, die den Status oder Fehlerzustände anzeigen. **Programmierschnittstelle**, um den Mikrocontroller mit der Entwicklungs-Umgebung zu verbinden. **USB-Debug-Kommunikation**, die eine einfache Fehlersuche und Überwachung ermöglicht. Lötpads für **GPIO-Anschlüsse**, um externe Komponenten flexibel verbinden zu können.

Diese Schaltung blieb über die gesamte Entwicklungsphase unverändert, da das Design von Anfang an stabil und zuverlässig lief.

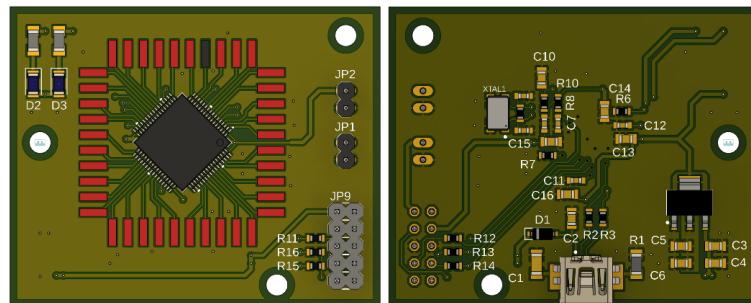


Abbildung 8.1.: Designphasen

Motor Treiber Die Aufgabe des Motortreibers besteht in der Ansteuerung der drei BLDC-H-Brücken, der Phasenstrommessung für den Sinusbetrieb und der Drehmomentregelung. Die erste Version des Motortreibers basierte auf dem TI DRV8302 Treiber-IC, das aus dem VESC-Referenzdesign übernommen wurde. Dieses IC bot mehrere Vorteile: Integrierte **Spannungsversorgung** 12-60V. Integrierte **Verstärker** für die Phasenstrommessung. **Fehlererkennung** für Bootstrap-Spannung, Phasenstrom und Phasenspannung über Integrierte SPI Kommunikation. Einfache Integration durch ein gut dokumentiertes Design.

Netzteil Das Netzteil übernimmt die Aufgabe, aus der Akku-Spannung von 42–58V (siehe Anhang Entwicklung 11.1) alle systemrelevanten Spannungen zu erzeugen: 12V (min. 1A): Versorgung von Beleuchtung, Lüftern und Relais. 5V (min. 0.5A): Stromquelle für das Display und die Motorsensoren. 3.3V (min. 0.3A): Energieversorgung für den Mikrocontroller und weitere logische Komponenten.

In der ersten Version des Netzteils wurde ein DC/DC-Spannungsregler für 12V und LDOs (Low-Dropout-Regler) für 5V und 3.3V verwendet. Die LDOs schienen aufgrund des geringen Beschaltungsaufwand eine kostengünstige Lösung zu sein. Allerdings zeigten die Tests, dass der Spannungsabfall auf dem 5V-LDO zu hoch war, um die notwendige Leistung zuverlässig bereitzustellen. Dies führte zu einer Überarbeitung des Designs in Designphase 2.

8.2. Designphase 2

Integration Für die ersten Inbetriebnahmen wurde ein vereinfachter Testaufbau konzipiert, um die Funktion der einzelnen Baugruppen zu verifizieren und Schwachstellen im Design frühzeitig zu identifizieren.

Zur Sicherstellung des Personenschutzes kamen ein Labornetzteil mit Überstromschutz sowie ein kleiner, leistungsschwacher 50W NEMA-Motor zum Einsatz. Das Labornetzteil reduziert das Risiko elektrischer Fehler im Vergleich zur direkten Nutzung eines Akkus. Zudem minimiert der nur 300 g leichte Motor die Gefahren durch rotierende mechanische Teile im Vergleich zum deutlich leistungsstärkeren 36-Zoll-Hub-Motor.

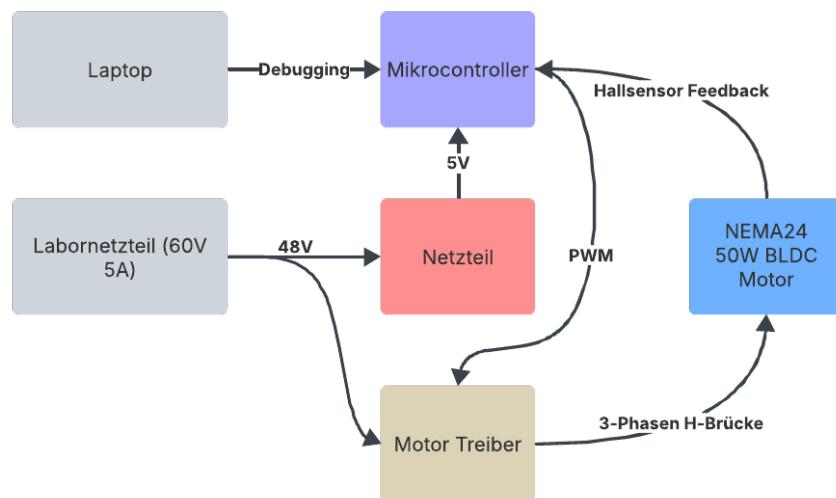


Abbildung 8.2.: Testaufbau Blockdiagramm

Motor Treiber Redesign Trotz der Vorteile wurde der TI DRV8302 nicht für die finale Version verwendet, hauptsächlich aus folgenden Gründen: Hohe Kosten des Treiber-ICs (ca. 8 € pro Stück bei Distributoren wie Digikey). Der Wunsch, das Design stärker von der Referenzlösung abzuheben und eigene Entwicklungen einzubringen.

In der zweiten Version wurde der Infineon IR2101S eingesetzt, ein Halbbrückenregler, der zu einem Bruchteil der Kosten (unter 1 €) erhältlich ist. Für die Phasenstrommessung war eine zusätzliche Verstärkerschaltung erforderlich, die mithilfe des AD8418 realisiert wurde. Dieses Design ermöglichte eine deutlich günstigere und individuellere Umsetzung.

Netzteil Redesign in der zweiten Version wurde der 5V-LDO durch einen weiteren DC/DC-Wandler ersetzt, um dessen Verluste zu minimieren. Hierfür wurde ein neuer DC/DC-Wandler(LMR36520) mit variabler Ausgangsspannung eingeführt, um die selben Komponenten für 12V & 5V Versorgung zu nutzen.

8.3. Designphase 3

Nachdem die grundlegenden Funktionen Gestalt angenommen hatten, konnten nun die Zusatzfunktionen in den Fokus genommen werden.

Display Für die Anbindung von Display-Modulen wurde eine spezielle bidirektionale Level-Shifter-Schaltung entwickelt (siehe Schaltplan Abbildung 3.13). Ziel war es, eine Schnittstelle zu schaffen, die das breite Angebot an Arduino-, ESP32- und Raspberry Pi-Displays voll ausnutzen kann. Dafür musste sowohl UART als auch I2C mit 3.3V- und 5V-Logik-Level unterstützt werden.

Es ist wichtig zu erwähnen, dass die verbaute Schaltung die Übertragungsrate auf die Schaltfrequenz des 2N7002-MOSFETs limitiert. Diese liegt mit rund 250 kHz jedoch weit über den benötigten 10–100 kHz (I2C Normal/Fast Speed) bzw. 9,6–115,2 kBaud (UART), die für herkömmliche Display-Module erforderlich sind. Das im finalen Design verwendete 2004 LCD benötigt lediglich 10 kHz I2C, sodass die gewählte Schaltung mehr als ausreichend ist.

LED Treiber Für die Steuerung der Beleuchtung war lediglich eine einfache MOSFET-Treiber-Schaltung erforderlich (siehe Schaltplan Abbildung 3.11). Diese konnte problemlos auf einer Lochrasterplatine evaluiert werden und ermöglicht die Ansteuerung gängiger 12V-Fahrrad- und Motorradbeleuchtungselemente.

Die gleiche Schaltung wurde zudem für den Betrieb der Lüfter sowie des Selbsthalterelais verwendet. Hier war lediglich die Ergänzung einer Freilaufdiode erforderlich, um Spannungsspitzen beim Schalten der induktiven Lasten zu vermeiden.

8.4. Abschluss der Entwicklung

Nach der Fertigstellung der finalen Designs der einzelnen Funktionsgruppen wurde eine Gesamthardware entwickelt, die alle Komponenten nahtlos miteinander verbindet. Diese schrittweise Integration erwies sich als äußerst effizient, da sie mehrere Vorteile bot:

Kostensparnis: Durch die Iteration auf kleinen, kostengünstigen 2-Layer-PCBs, die weniger Bestückungsaufwand erforderten.

Zeitersparnis: Die Integration basierte auf stabilen und getesteten Funktionsgruppen, wodurch bereits fertige Layouts einfach miteinander verbunden werden konnten.

Erhöhte Zuverlässigkeit: Potenzielle Fehlerquellen wurden bereits in der zweiten Designphase identifiziert und behoben, was die Robustheit der Gesamthardware deutlich verbesserte.

Die erfolgreiche Integration aller Funktionsgruppen in eine einzige, optimierte Hardware stellte somit einen entscheidenden Meilenstein in der Realisierung des Projekts dar.

9. Relevante Konzepte

9.1. BLDC

Grundlagen Ein BLDC-Motor (bürstenloser Gleichstrommotor) funktioniert durch ein rotierendes Magnetfeld, das von Spulen im Stator erzeugt wird. Der Rotor mit Permanentmagneten wird von diesem Magnetfeld angezogen und in Drehung versetzt.

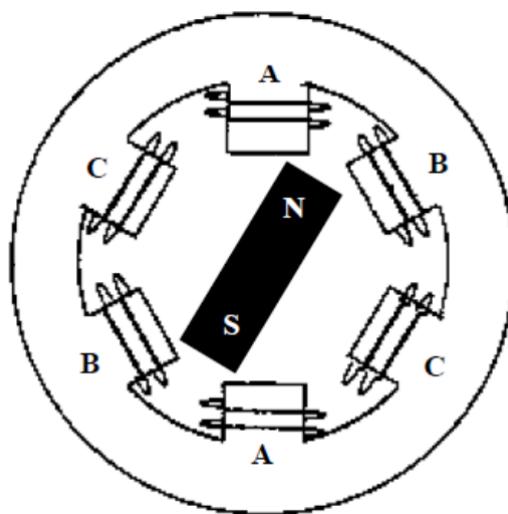


Abbildung 9.1.: (Texas Instruments. (2015). Trapezoidal Control of BLDC Motors Using Hall Effect Sensors . C2000 Systems and Applications Team, Bilal Akin, Manish Bhardwaj, & Jon Warriner.)

Zur Synchronisation des elektrischen Stroms mit der Position des Rotors verwenden BLDC-Motoren häufig Hallsensoren oder andere Positionssensoren. Diese Sensoren ermitteln die exakte Position des Rotors und geben diese Information an den Controller weiter, sodass die Spulen präzise im richtigen Moment angesteuert werden. Diese Sensorik machen einen der erheblichen Vorteile von modernen BLDC motoren gegenüber herkömmlichen Wechselstrom motoren aus.

Ansteuerung Die Steuerung von BLDC-Motoren (Brushless DC-Motoren) ist ein zentraler Aspekt ihrer Anwendung und bestimmt maßgeblich die Effizienz, Laufruhe und Steuerbarkeit der Motoren. In diesem Kapitel werden die wichtigsten Steuerungsmethoden für BLDC-Motoren beschrieben: Sinussteuerung, FOC-Steuerung und Trapezsteuerung.

Trapez Steuerung ist eine einfachere und kostengünstigere Methode zur Steuerung von BLDC-Motoren. Sie verwendet rechteckige (trapezförmige) Spannungs- oder Stromwellen, um die Phasen des Motors zu betreiben. **Funktionsweise:** Der Rotor wird durch das Erzeugen eines trapezförmigen Magnetfelds angetrieben. Die Kommutierung erfolgt diskret in Schritten von 60° elektrischem Winkel. Die Rotorposition wird meist mithilfe von Hall-Sensoren erfasst. **Vorteile:** Einfache Umsetzung, Hohes Drehmoment, Geringste Schaltverluste, **Nachteile:** Höhere Geräusch- und Vibrationsentwicklung im Vergleich zu Sinus- oder FOC-Steuerung. Weniger effizient, besonders bei hoher Drehzahl.

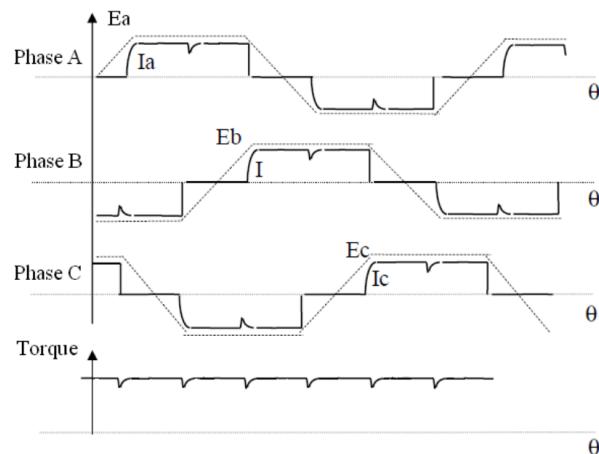


Abbildung 9.2.: (Texas Instruments. (2015). Trapezoidal Control of BLDC Motors Using Hall Effect Sensors . C2000 Systems and Applications Team, Bilal Akin, Manish Bhardwaj, & Jon Warriner.)

Sinus Steuerung basiert darauf, die Ströme in den Phasen des Motors so zu regeln, dass sie sinusförmig verlaufen. Dadurch wird ein gleichmäßiges Drehmoment erzeugt, was sich in einer hohen Laufruhe und geringen Vibrationen äußert. **Funktionsweise:** Die Sinussteuerung verwendet Sinuswellen als Referenzsignal für die Phasenströme. Diese werden über Pulsweitenmodulation (PWM) in die Wicklungen eingespeist. **Vorteile:** Sehr leiser und vibrationsarmer Betrieb. Bessere Effizienz bei hohen Drehzahlen. Geeignet für Anwendungen, bei denen Laufruhe entscheidend ist. **Nachteile:** Erfordert komplexe Elektronik und präzise Sensorik zur Bestimmung der Rotorposition. Kann weniger effizient sein als andere Methoden bei hoher Last im unteren Drehzahl-Bereich.

FOC Steuerung Field-Oriented Control (Vektorregelung) ist eine fortschrittliche Methode zur Steuerung von BLDC-Motoren. Sie nutzt mathematische Transformationen, um die Drehmoment- und Magnetisierungsströme unabhängig voneinander zu regeln.

Funktionsweise: Die Rotorposition wird kontinuierlich überwacht, oft mithilfe von Hall-Sensoren oder Encodern. Mithilfe einer Park- und Clarke-Transformation werden die Phasenströme von einem dreiphasigen in ein zweidimensionales Koordinatensystem (d-q-Achsen) transformiert. Dies ermöglicht die gezielte Regelung des Drehmoments (q-Achse) und der Magnetisierung (d-Achse). **Vorteile:** Sehr hohe Effizienz bei variabler Drehzahl und Last. Präzise Steuerung des Drehmoments. Unterstützt hohe Drehzahlen und bietet bessere Leistung als andere Methoden. **Nachteile:** Hohe Rechenleistung erforderlich, da fortschrittliche Algorithmen verwendet werden. Aufwendige Sensorik und Kalibrierung notwendig.

In diesem Projekt wurde die Trapezsteuerung als Steuerungsmethode ausgewählt. Ausschlaggebend hierfür waren die einfache Implementierung, das hohe Drehmoment und die vorhandene Sensorik des Fahrradmotors, der nur eine Auflösung von 60° bietet. Die Hardware des Systems ist jedoch vollständig ausgestattet, um auch eine Sinussteuerung zu realisieren. Eine Umsetzung der FOC-Steuerung ist hingegen mit dem aktuellen Motor nicht möglich, da hierfür entweder präzisere Hall-Sensoren oder eine Back-EMF-Messschaltung erforderlich wären.

Teil IV.

Anhang

10. System Design

System Komponenten:

Bauteil	Name	Beschreibung
Motor	Viribus LY23048911	1KW BLDC Motor
Akku	Unit Pack S039-3	48V 15Ah 750W/h
Display	GeeekPi I2C 2004 LCD	20x4 LCD I ² C display
Bedienelemente	QWork WD7435 Greluma ZL277LUM VGEBY B08KFC114S	Lenkerschalter-Beleucht Lenker E-Stop Schalter Analog Gasgriff
Scheinwerfer	Filmmer 49002	Fahrrad Scheinwerfer
Rücklicht	evermotor A08051-3	Motorrad 50mm Brems-Rücklicht
Blinker	Dowrap B0CK6ZTGS3 evermotor A08023	12V Led blinker Vorne 12V Led blinker Hinten
Steuergerät	RESC_V2	13-64V 30A E-Bike Steuergerät
Notaus	Tawvelm SPST	48V 100A DC Schalter
Lüfter	2x GDA4010	40x10mm 12V Lüfter
Kühlkörper	VPR138/94-M3	94x46x33mm Kühlkörper

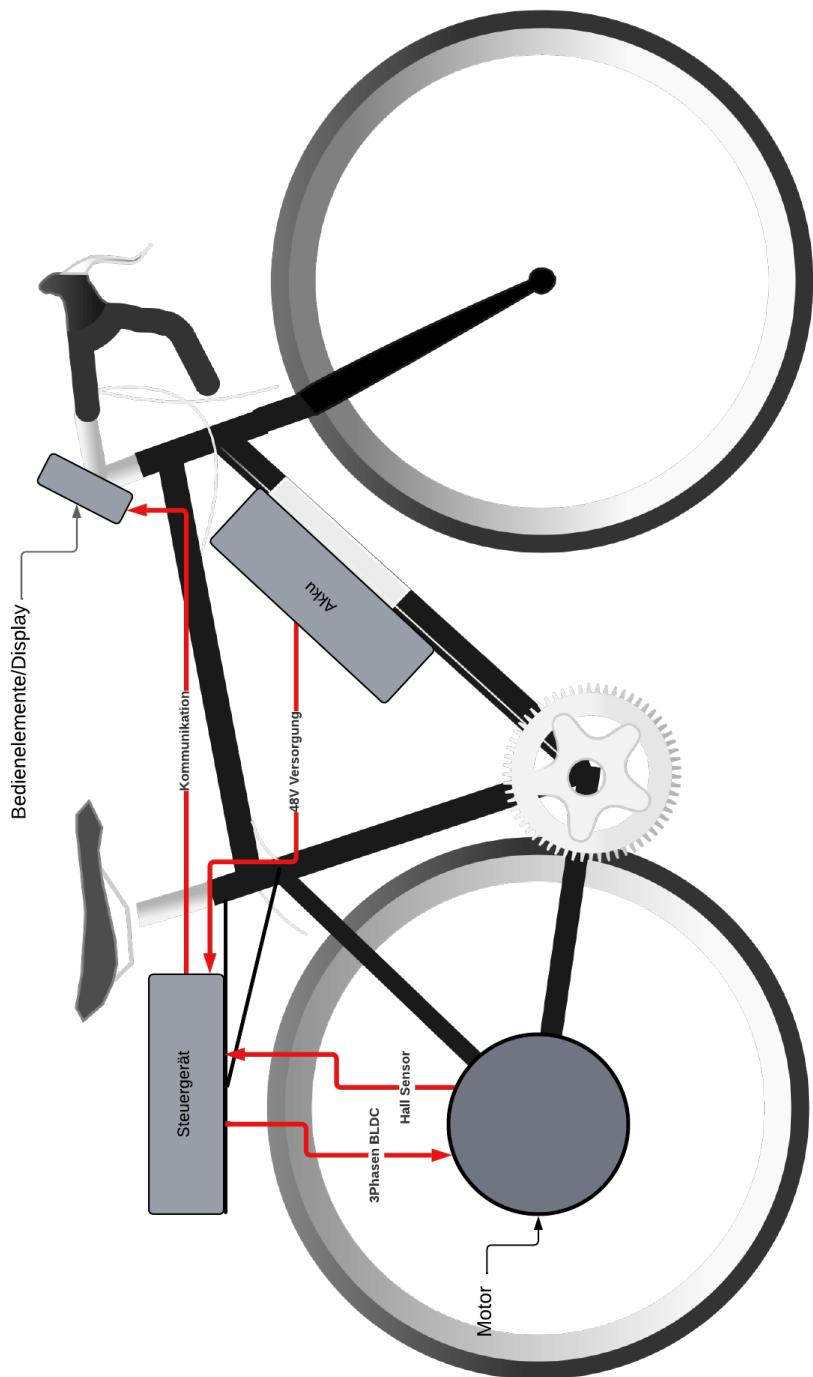


Abbildung 10.1.: Vereinfachtes System Diagramm

11. Entwicklung

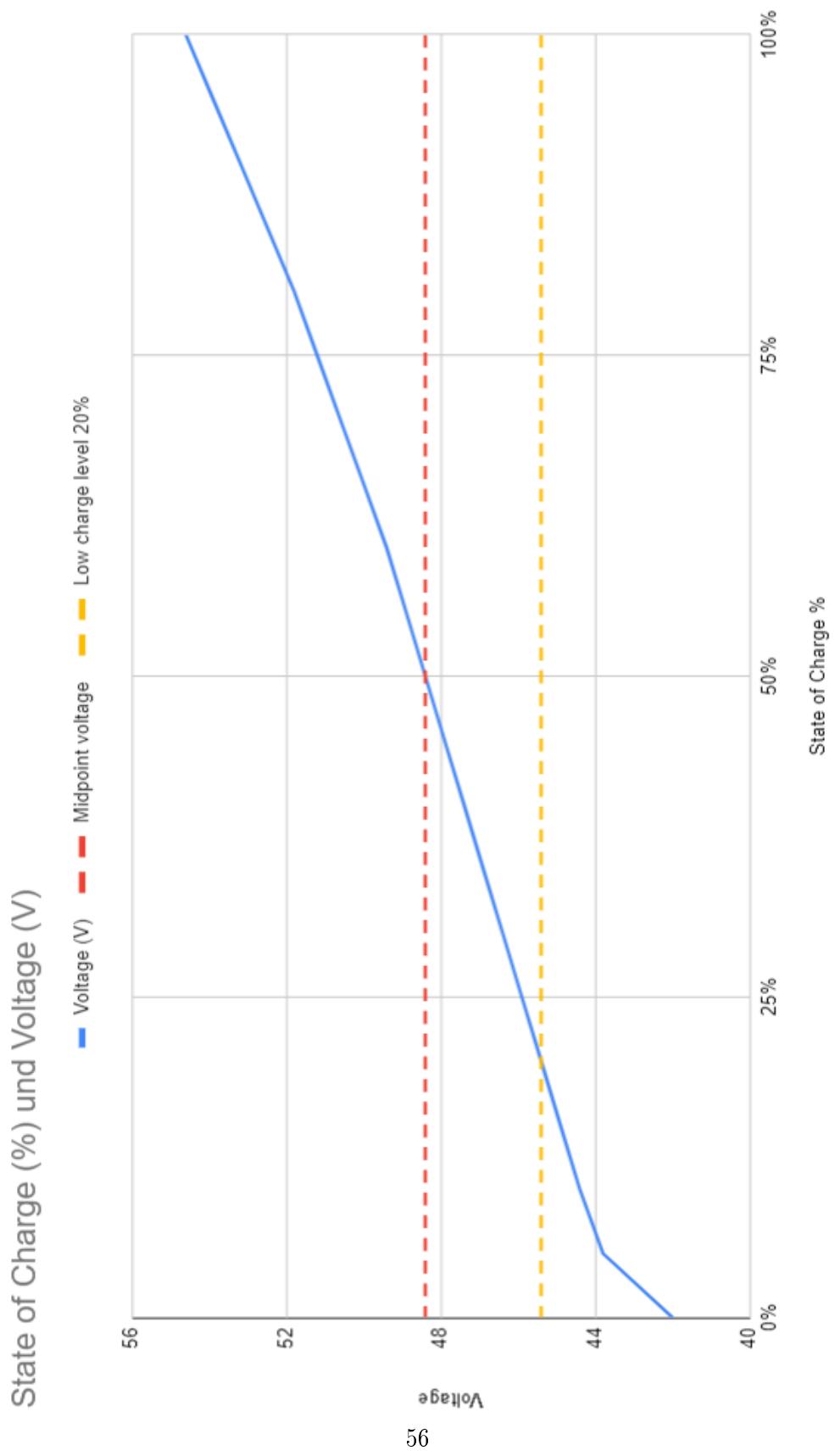
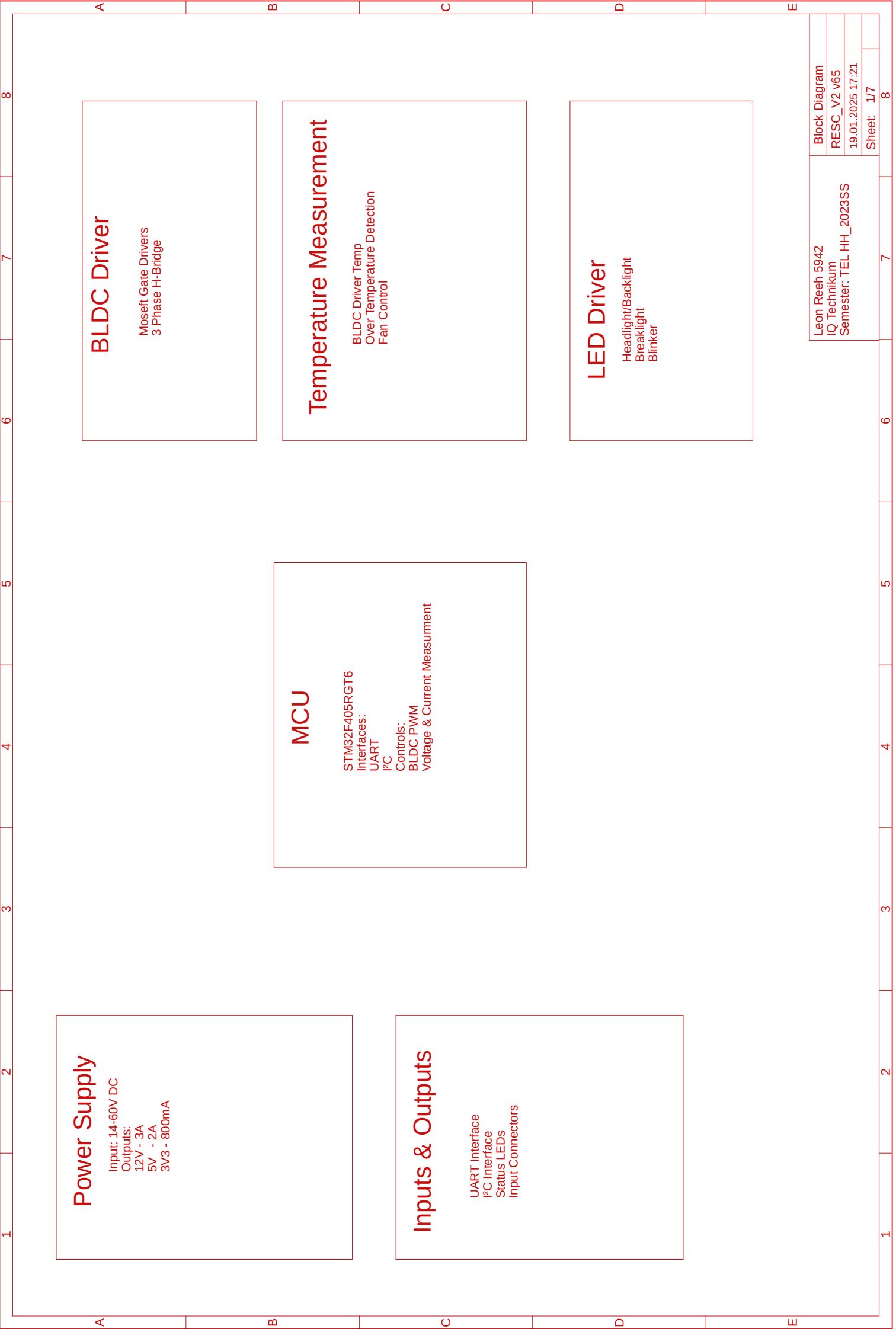
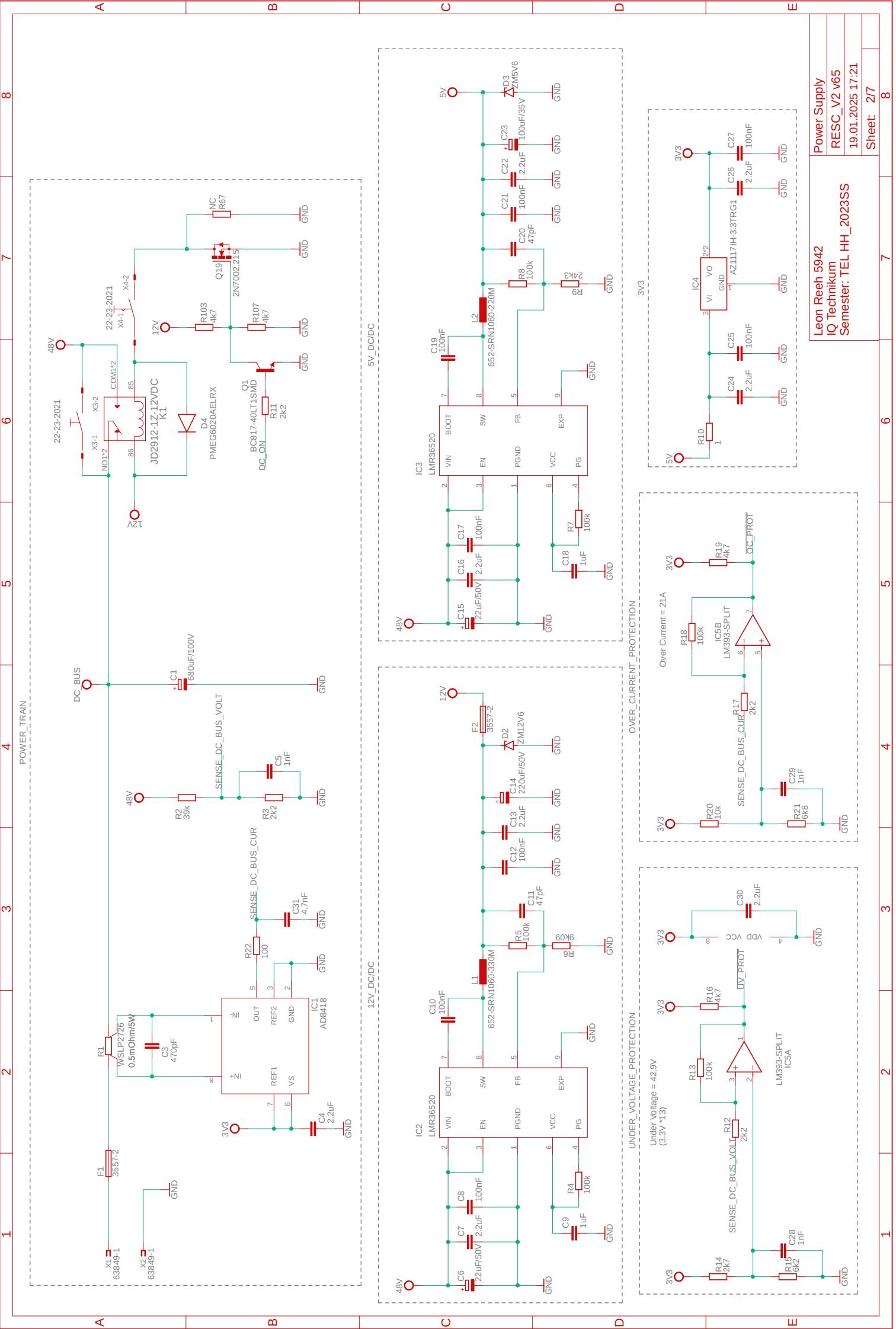
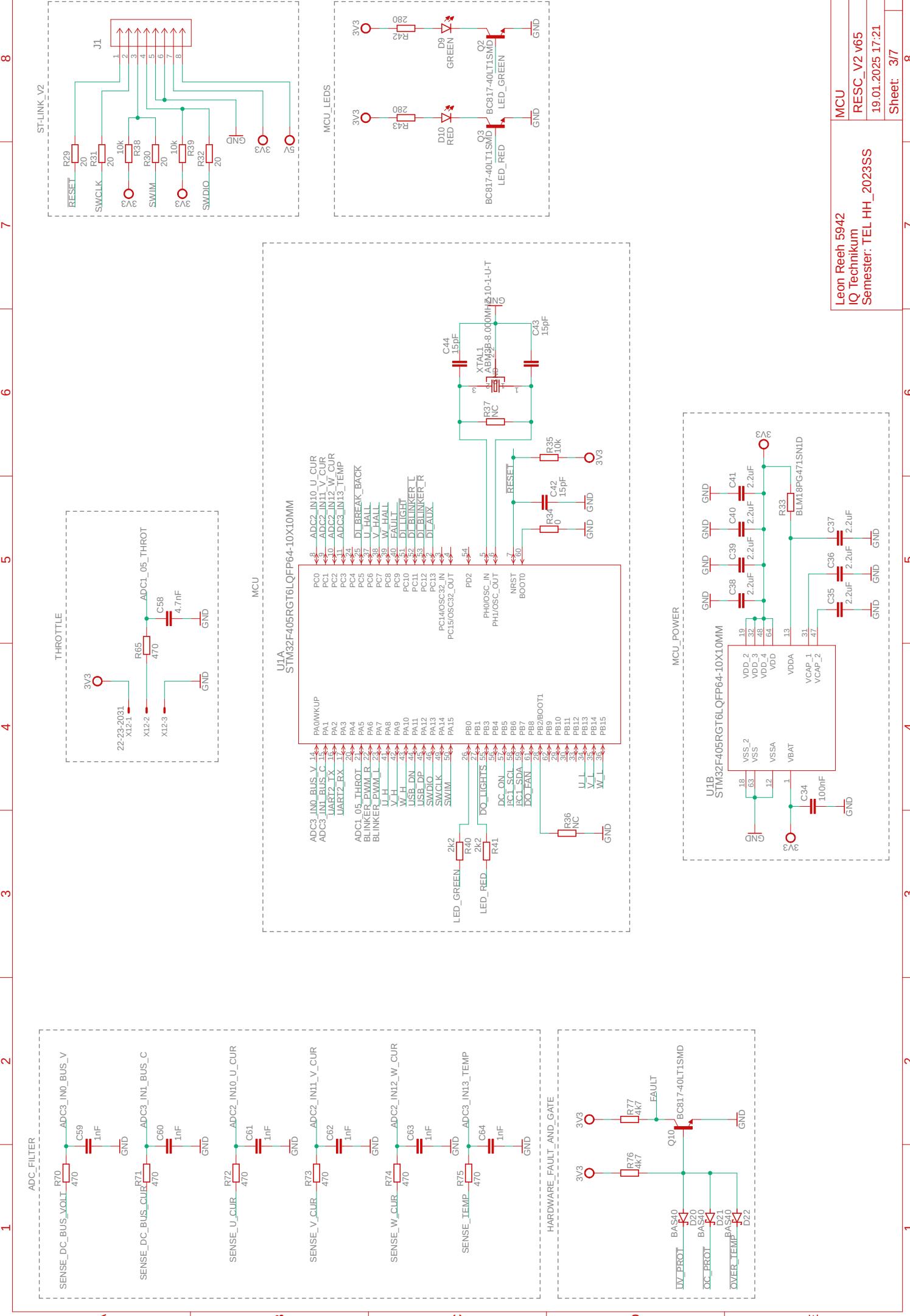


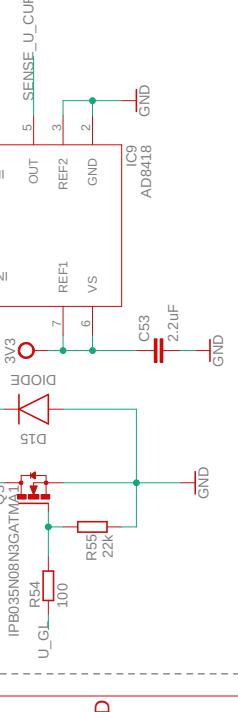
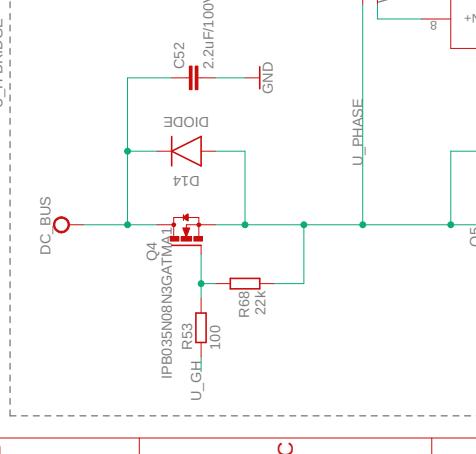
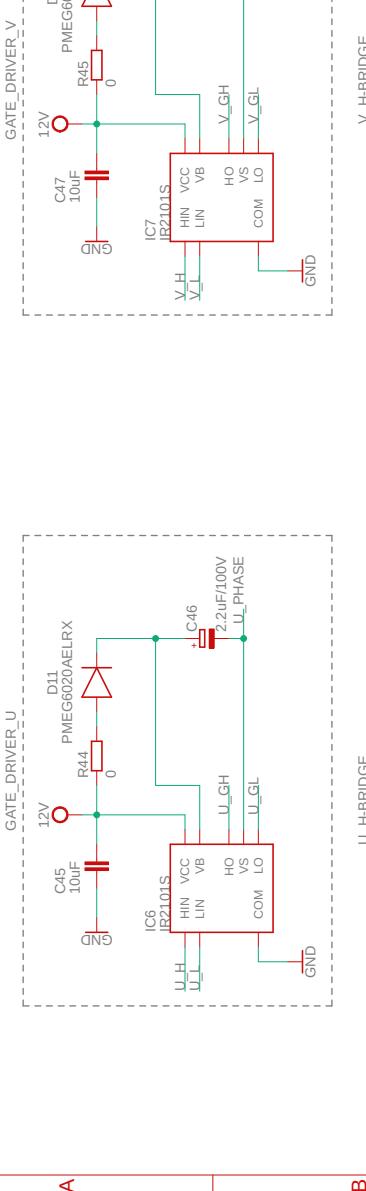
Abbildung 11.1.: 48V Akku Spannungskurve

12. Schaltplan







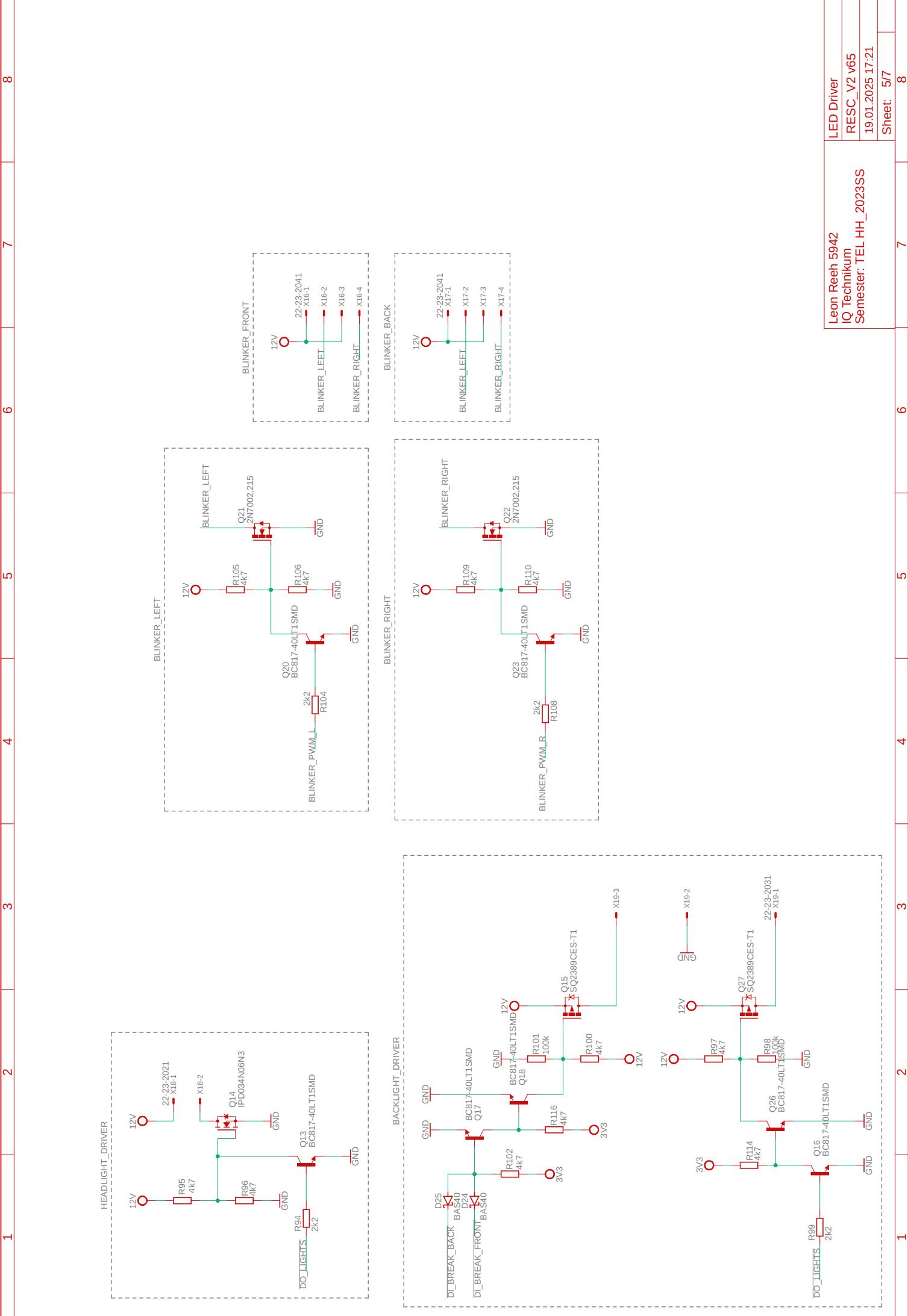


HS1
VPR138/94-M3

Leon Reeh 5942
IQ Technikum
Semester: TEL HH_2023SS

BLDC Driver
RESC_V2 v65
19.01.2025 17:21

Sheet: 4/7



8

7

6

5

4

3

2

1

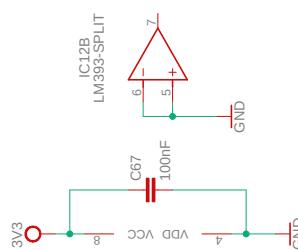
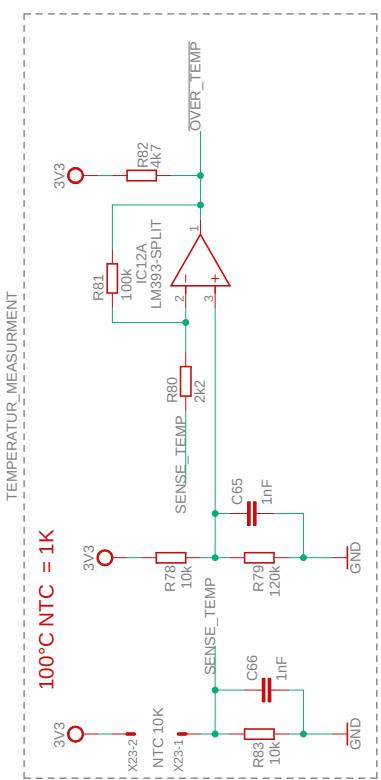
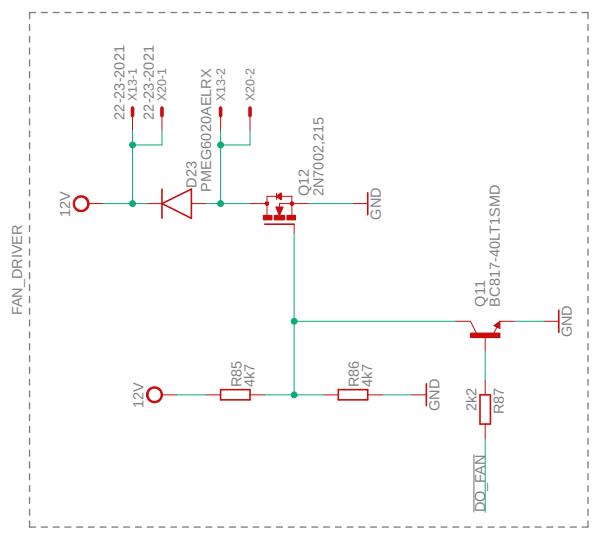
A

B

C

D

E



Leon Reeh 5942
IQ Technikum
Semester: TEL_HH_2023SS
Temperature Measurement
RESC_V2 v65
19.01.2025 17:21
Sheet: 6/7

8

7

6

5

4

3

2

1

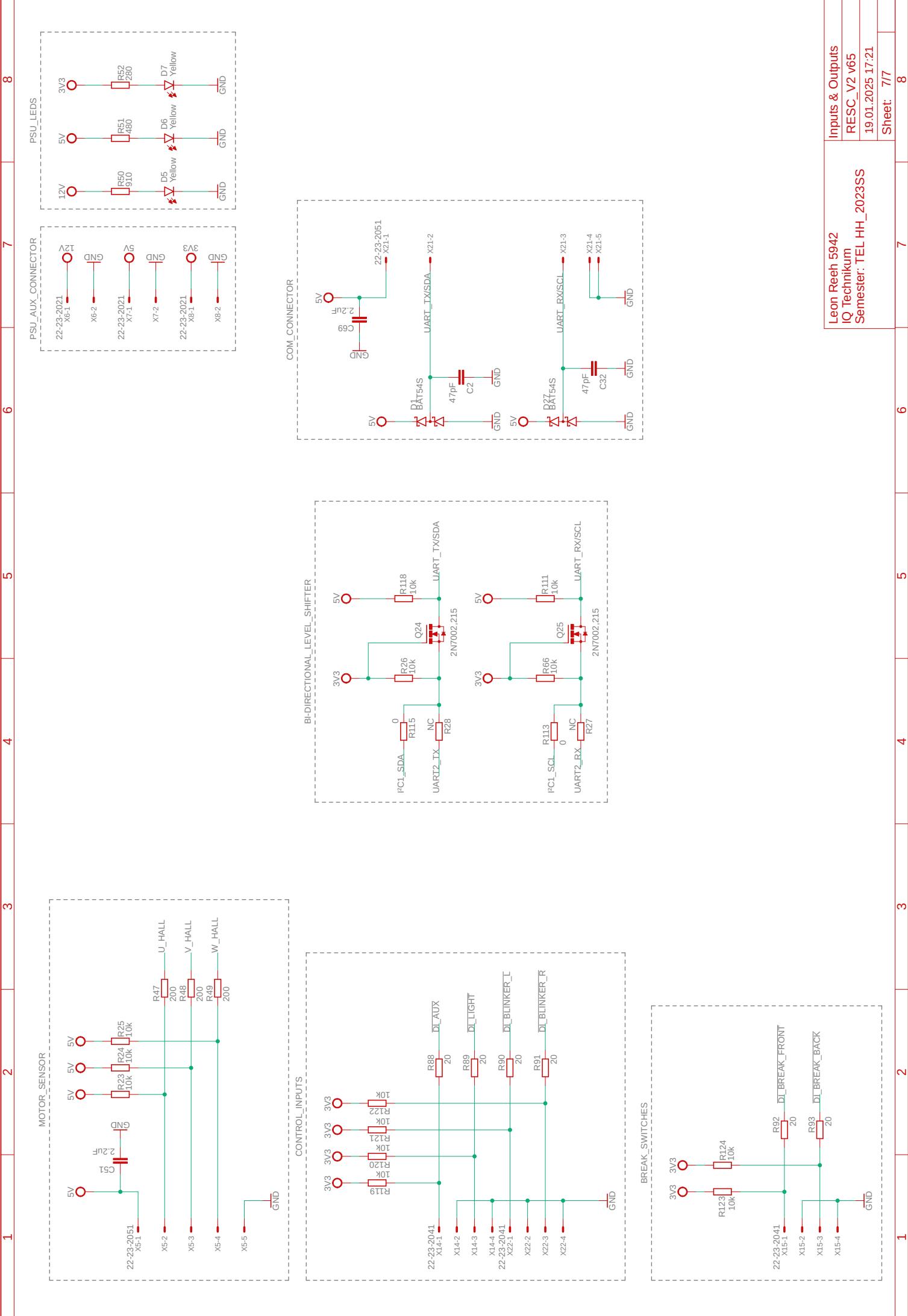
A

B

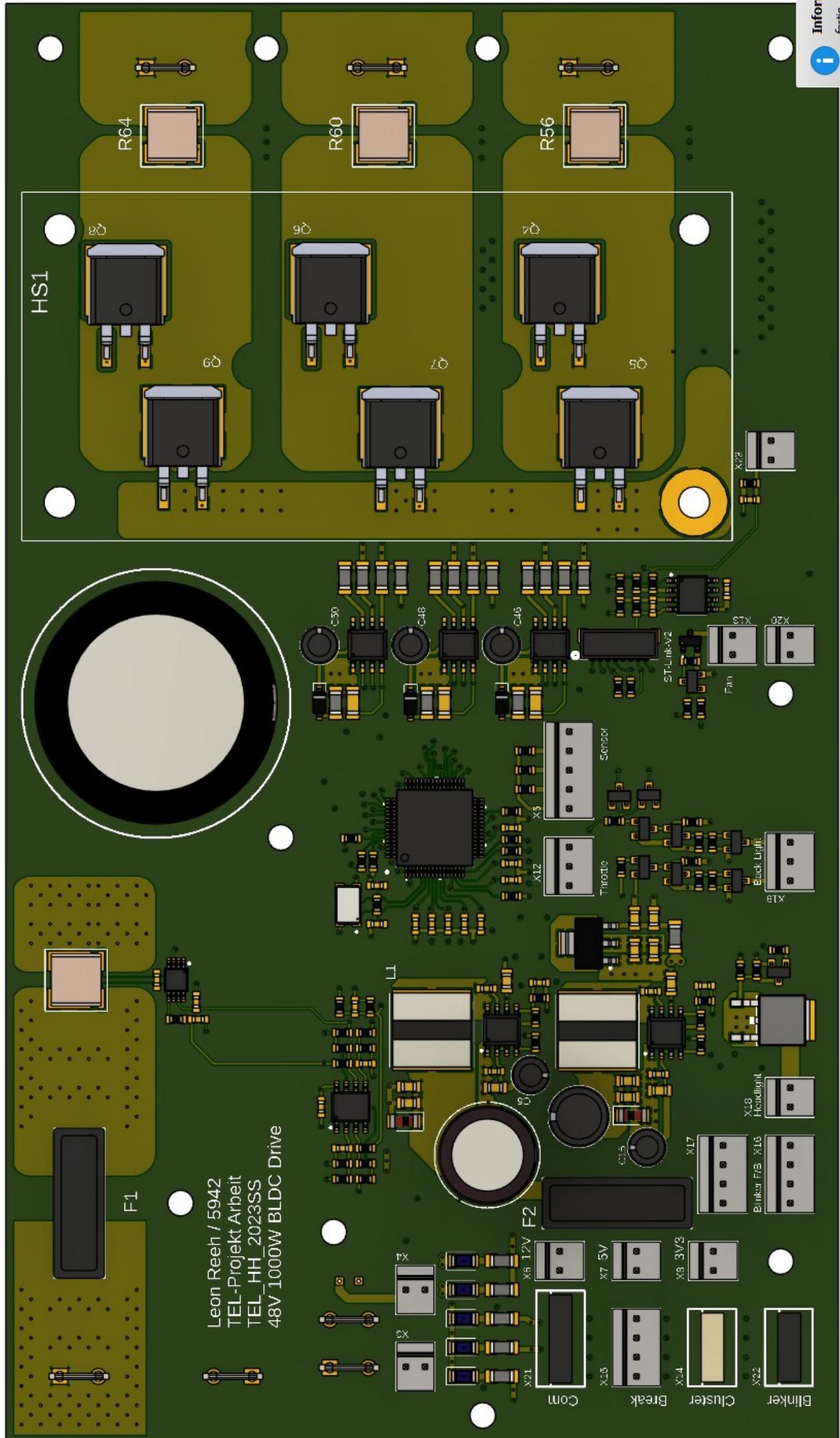
C

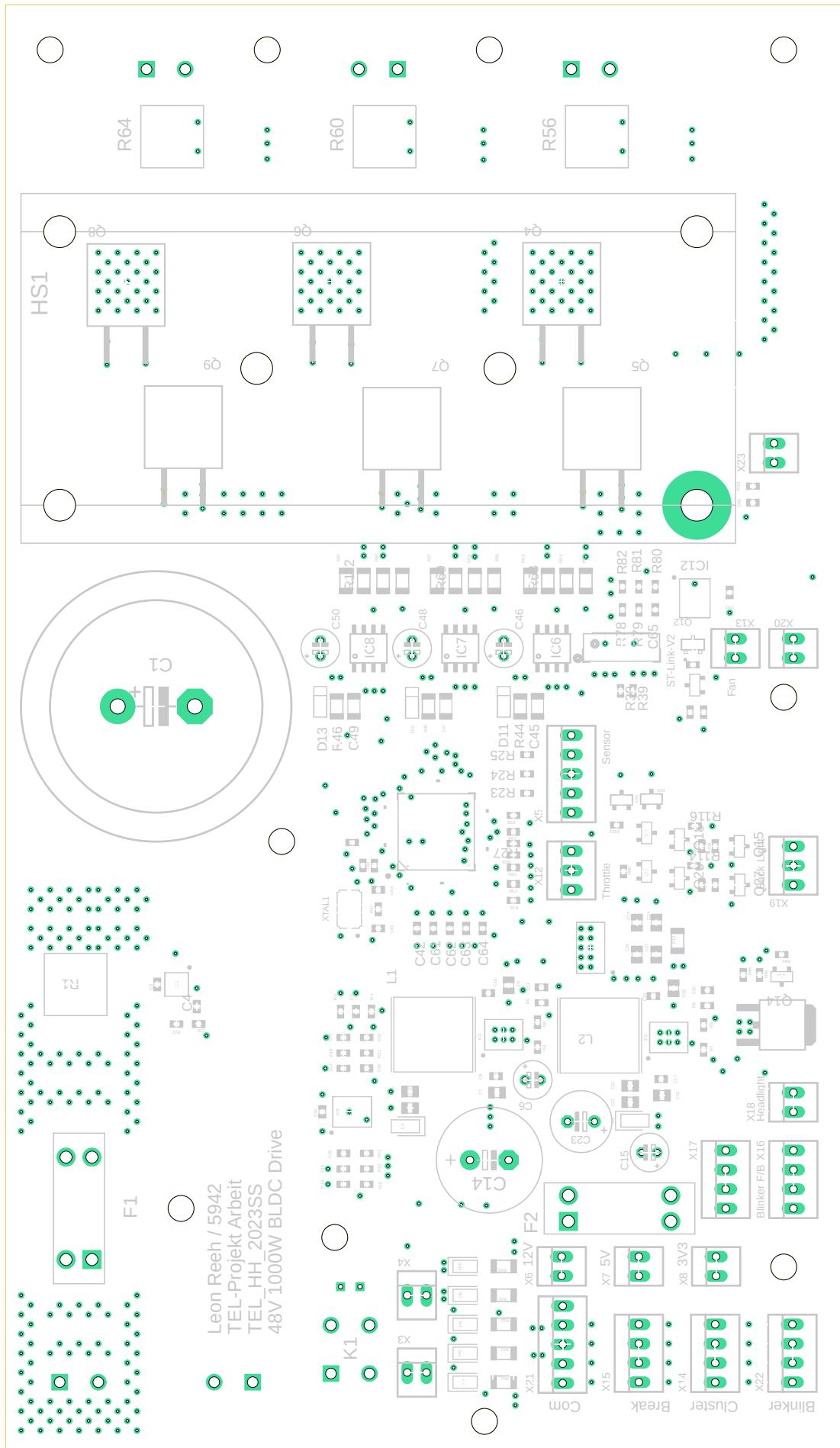
D

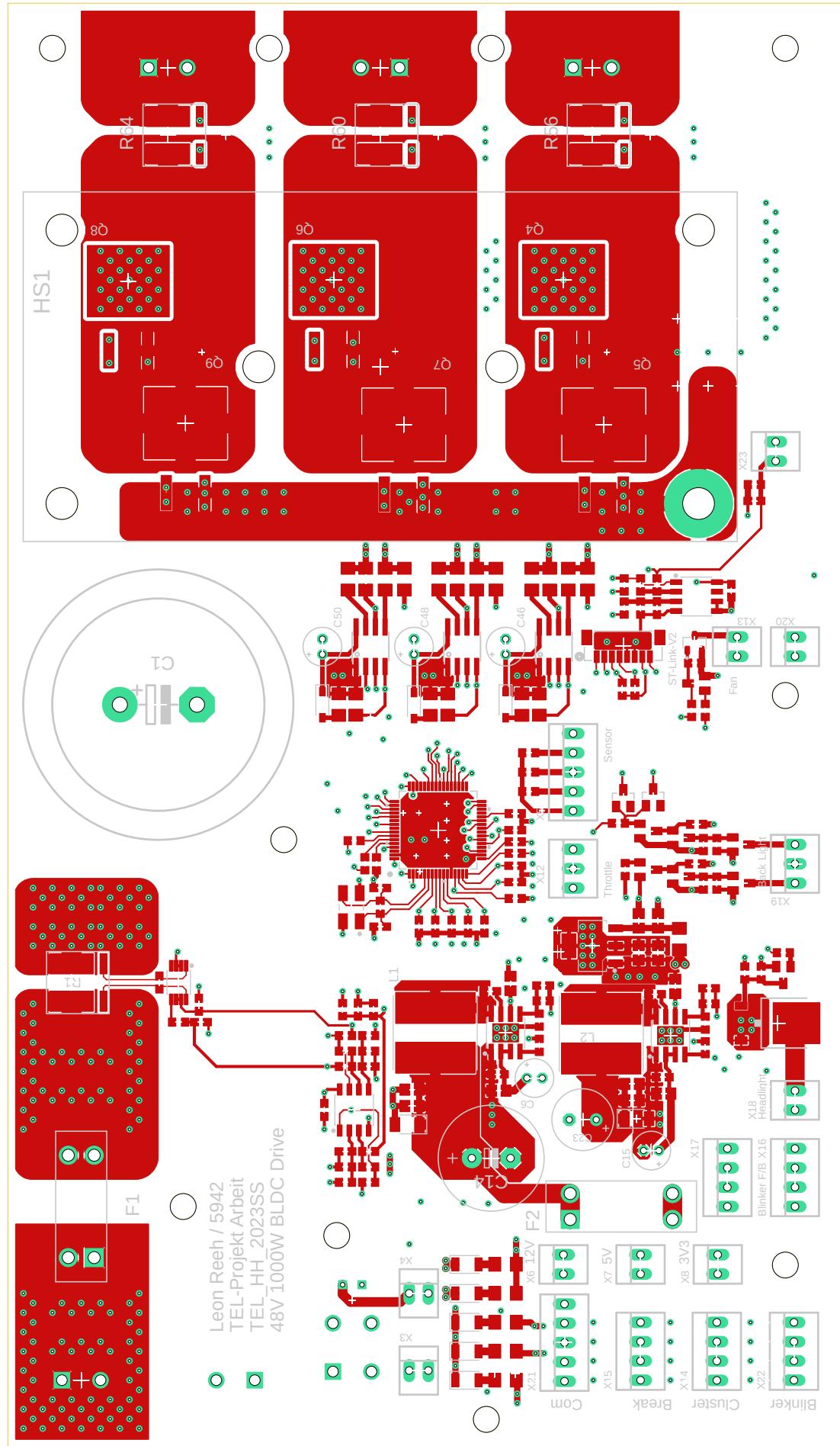
E

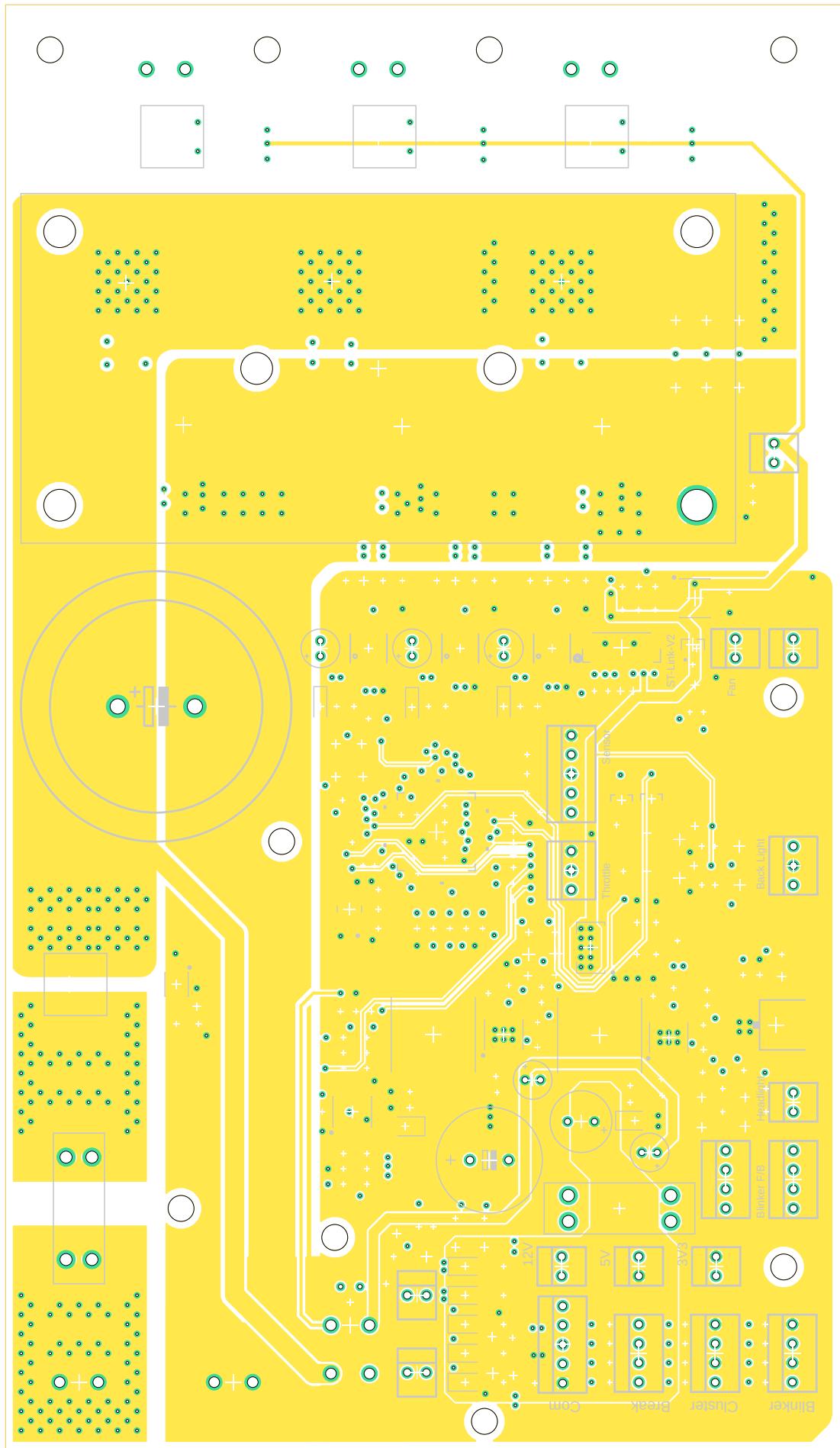


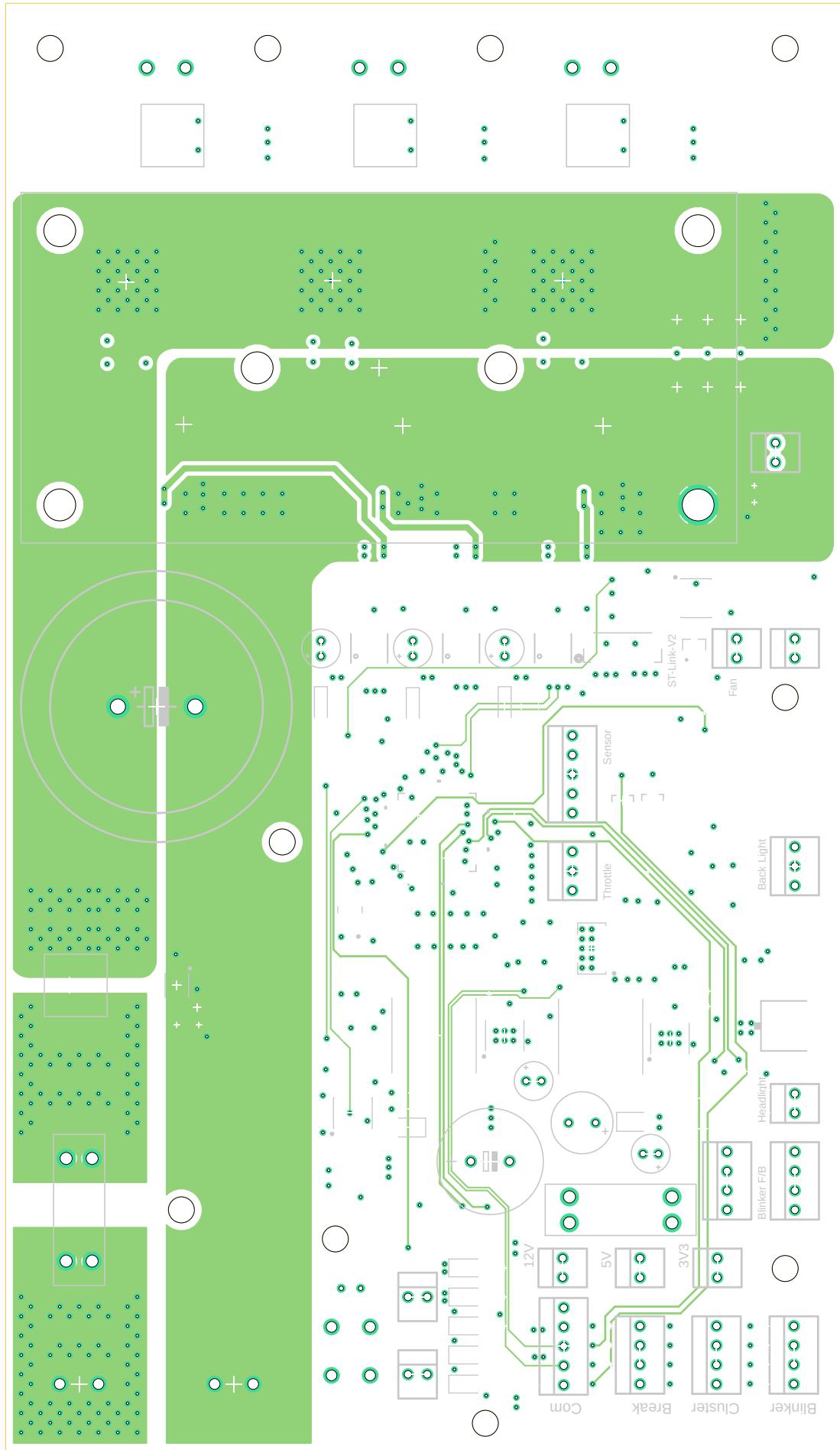
13. Layout

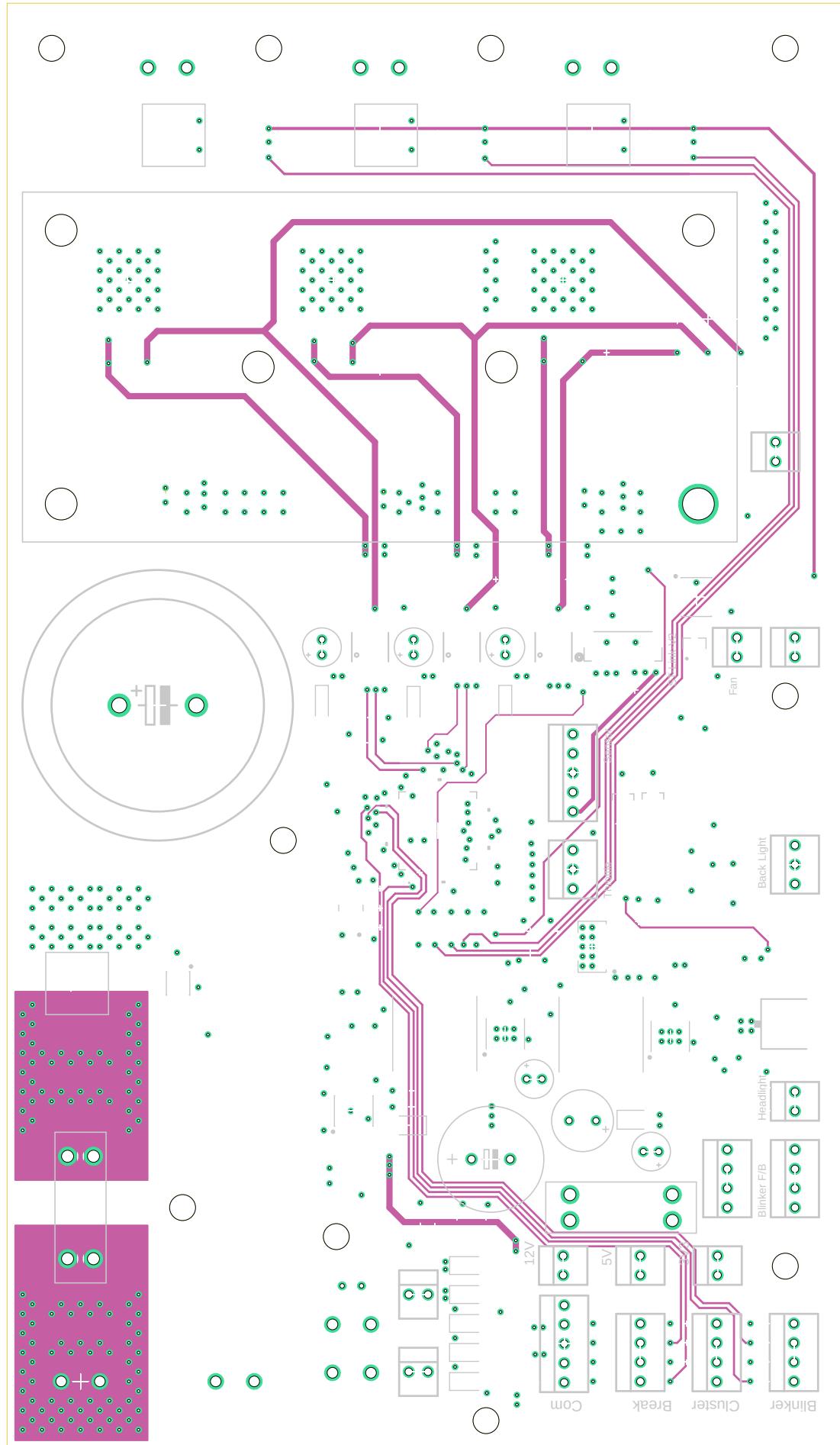


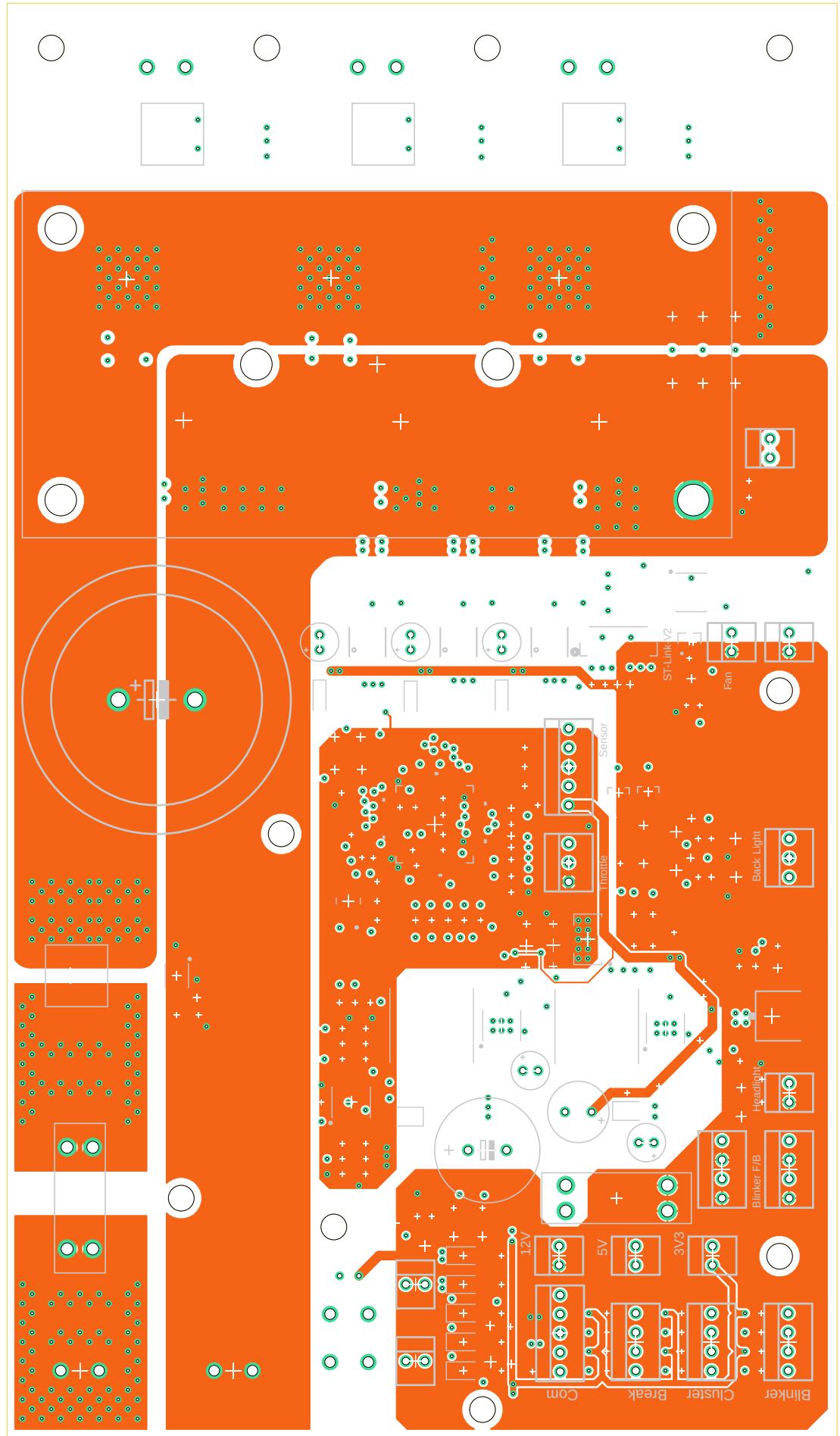


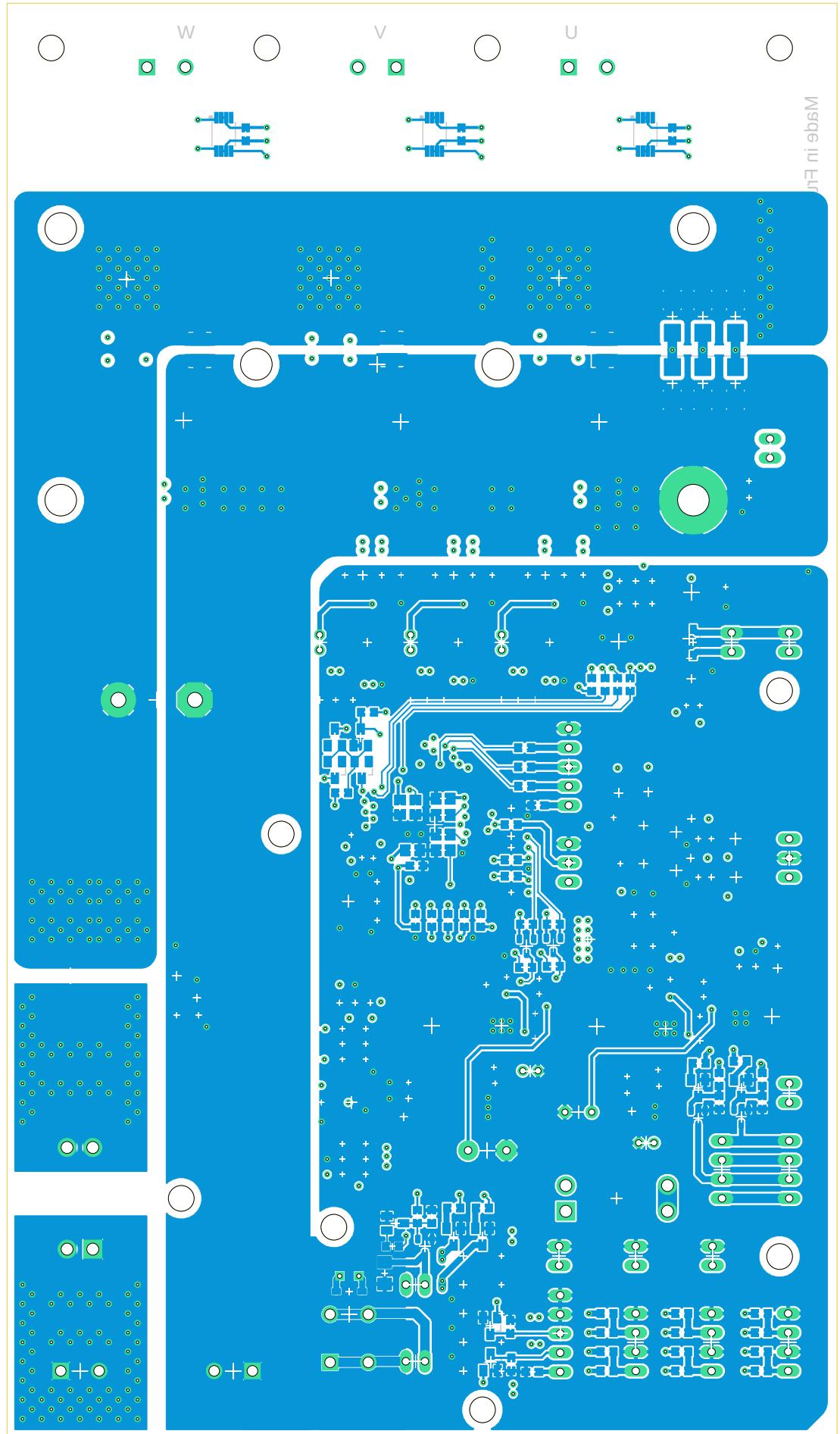


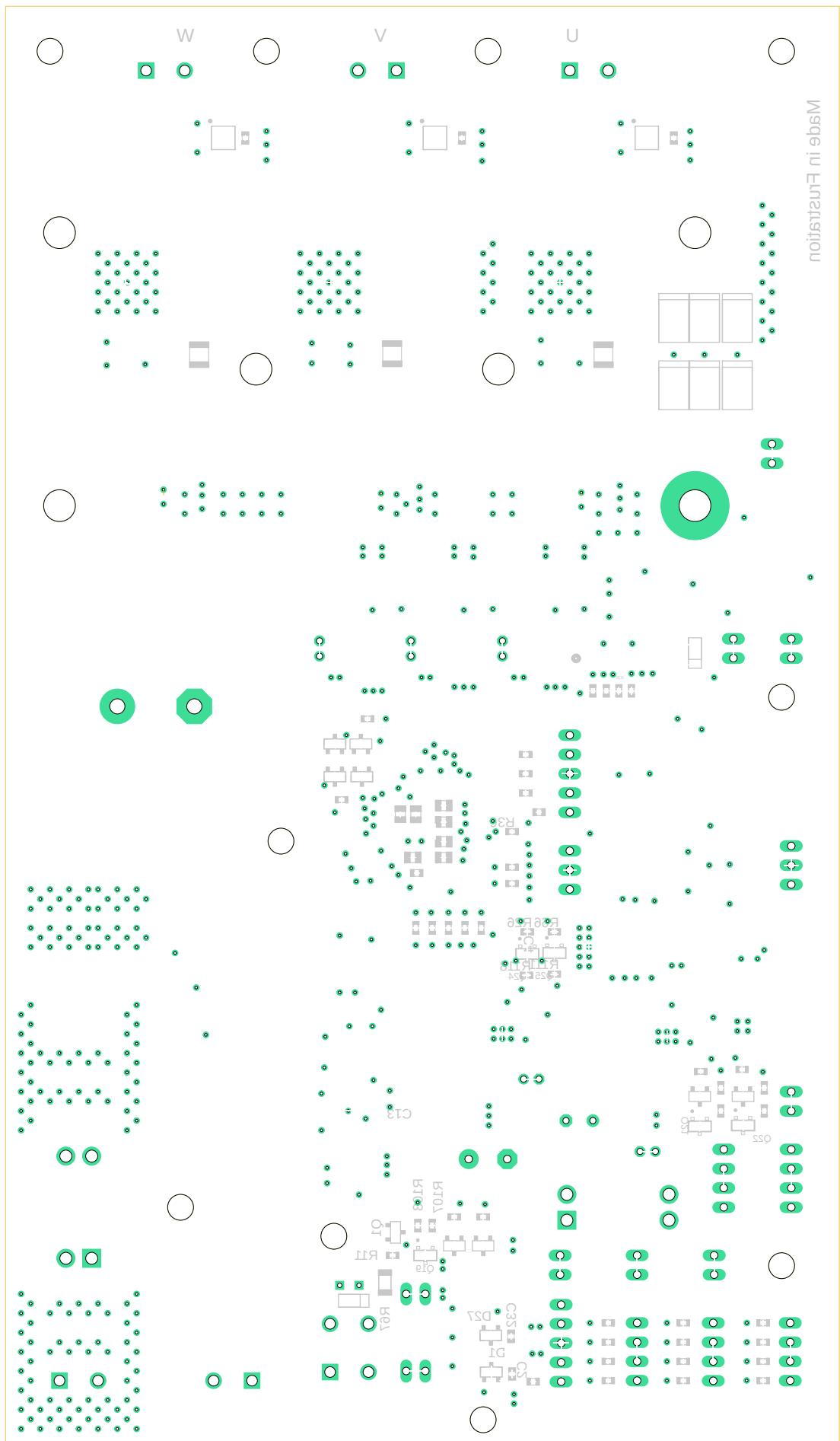












14. BOM

Stückliste exportiert aus C:/Users/Leon/AppData/Local/Temp/Neutron/ElectronFileOutput/14372/sch-06216c19-e44b-450a-8892-8c042def1b8f/RESC_V2 v60.sch am 19.01.2025 17:21

Qty	Value	Device	Footprint Name	Parts	Detailed Description
3	0	R-EU_R0603	R0603	R34, R113, R115	RESISTOR, European symbol
3	0	R-EU_R1206	R1206	R44, R45, R46	RESISTOR, European symbol
1	1	R-EU_R1206	R1206	R10	RESISTOR, European symbol
1	100	R-EU_R0603	R0603	R22	RESISTOR, European symbol
6	100	R-EU_R1206	R1206	R53, R54, R57, R58, R61, R62	RESISTOR, European symbol
9	100k	R-EU_R0603	R0603	R4, R5, R7, R8, R13, R18, R81, R98, R101	RESISTOR, European symbol
6	100nF	C-EUC0603	C0603	C8, C10, C17, C19, C34, C67	CAPACITOR, European symbol
4	100nF	C-EUC0805	C0805	C12, C21, C25, C27	CAPACITOR, European symbol
1	100uF/35V	CPOL-EUE3.5-8	E3,5-8	C23	POLARIZED CAPACITOR, European symbol
19	10k	R-EU_R0603	R0603	R20, R23, R24, R25, R26, R35, R38, R39, R66, R78, R83, R111, R118, R119, R120, R121, R122,	RESISTOR, European symbol

3	10uF	C-EUC1206	C1206	R123, R124
1	120k	R-EU_R0603	R0603	C45, C47, C49 RESISTOR, European symbol
3	15pF	C-EUC0603	C0603	C42, C43, C44 CAPACITOR, European symbol
11	1nF	C-EUC0603	C0603	C5, C28, C29, C59, C60, C61, C62, C63, C64, C65, C66 CAPACITOR, European symbol
2	1uF	C-EUC0805	C0805	C9, C18 CAPACITOR, European symbol
7	2.2uF	C-EUC0603	C0603	C4, C30, C51, C53, C55, C57, C69 CAPACITOR, European symbol
13	2.2uF	C-EUC0805	C0805	C7, C13, C16, C22, C24, C26, C28, C30, C32, C34, C36, C37, C38, C39, C40, C41 CAPACITOR, European symbol
3	2.2uF/100V	C-EUC1210	C1210	C52, C54, C56 CAPACITOR, European symbol
3	2.2uF/100V	CPOL-EUE2-5	E2-5	C46, C48, C50 POLARIZED CAPACITOR, European symbol
10	20	R-EU_R0603	R0603	R29, R30, R31, R32, R88, R89, R90, R91, R92, R93 RESISTOR, European symbol

3	200	R-EU_R0603	R0603	R47, R48, R49	RESISTOR, European symbol
8	22-23-2021	22-23-2021	22-23-2021	X3, X4, X6, X7, X8, X13, X18, X20	.100" (2.54mm) Center Header - 2 Pin
2	22-23-2031	22-23-2031	22-23-2031	X12, X19	.100" (2.54mm) Center Header - 3 Pin
5	22-23-2041	22-23-2041	22-23-2041	X14, X15, X16, X17, X22	.100" (2.54mm) Center Header - 4 Pin
2	22-23-2051	22-23-2051	22-23-2051	X5, X21	.100" (2.54mm) Center Header - 5 Pin
1	220uF/50V	CPOL-EU	E5-13	C14	POLARIZED CAPACITOR, European symbol
6	22k	R-EU_R1206	R1206	R55, R59, R63, R68, R69, R112	RESISTOR, European symbol
2	22uF/50V	CPOL-EUE2-5	E2-5	C6, C15	POLARIZED CAPACITOR, European symbol
1	24k3	R-EU_R0603	R0603	R9	RESISTOR, European symbol
3	280	R-EU_R1206	R1206	R42, R43, R52	RESISTOR, European symbol
6	2N7002,215	2N7002,215	SOT23-3	Q12, Q19, Q21, Q22, Q24, Q25	MOSFET N- CH 60V 300MA SOT- 23
12	2k2	R-EU_R0603	R0603	R3, R11, R12, R17, R40, R41, R80, R87, R94, R99,	RESISTOR, European symbol

				R104, R108
1	2k7	R-EU_R0603	R0603	R14 RESISTOR, European symbol
2	3557-2	3557-2	M60	F1, F2 RESISTOR,
1	39k	R-EU_R0603	R0603	R2 European symbol
2	4.7nF	C-EUC0603	C0603	C31, C58 CAPACITOR, European symbol
				R65, R70, R71, R72, R73, R74, R75
7	470	R-EU_R0603	R0603	RESISTOR, European symbol
1	470pF	C-EUC0603	C0603	C3 CAPACITOR, European symbol
4	47pF	C-EUC0603	C0603	C2, C11, C20, C32 CAPACITOR, European symbol
1	480	R-EU_R1206	R1206	R51 RESISTOR, European symbol
				R16, R19, R76, R77, R82, R85, R86, R95, R96, R97, R100, R102, R103, R105, R106, R107, R109, R110, R114, R116
20	4k7	R-EU_R0603	R0603	RESISTOR, European symbol
5	63849-1	63849-1	63849-1	X1, X2, X9, X10, X11
1	652-SRN1060-220M	7847870SRN1016	SRN1016	L2
1	652-SRN1060-330M	7847870SRN1016	SRN1016	L1
1	680uF/100V	CPOL-EUE10-35	EB35D	C1 POLARIZED CAPACITOR, European symbol

1	6k2	R-EU_R0603	R0603	R15	RESISTOR, European symbol
1	6k8	R-EU_R0603	R0603	R21	RESISTOR, European symbol
1	910	R-EU_R1206	R1206	R50	RESISTOR, European symbol
1	9k09	R-EU_R0603	R0603	R6	RESISTOR, European symbol
1	ABM3B-8.000MHZ-10-1- U-T	7BFREQUENCY_8MHZ	OSCSC500X320X105- 4N	XTAL1	CRYSTAL 8MHZ to 40MHZ 18PF SMD
4	AD8418	AD8418	SOP65P490X110-8N	IC1, IC9, IC10, IC11	
1	AZ1117IH-3.3TRG1	AZ1117IH-3.3TRG1	SOT230P700X170-4	IC4 D20, D21, D22, D24, D25	
5	BAS40	BAS40	SOT23	D20, D21, D22, D24, D25	Silicon Schottky Diodes
2	BAT54S	BAT54S	SOT23	D1, D27	Schottky Diodes
12	BC817-40LT1SMD	BC817-40LT1SMD	SOT23-BEC	Q1, Q2, Q3, Q10, Q11, Q13, Q16, Q17, Q18, Q20, Q23, Q26	NPN Transistor
1	BLM18PG471SN1D	R-EU_R0603	R0603	R33	RESISTOR, European symbol
1	BM08B-SRSS-TB(LF) (SN)CONN_BM08B- SRSS-TB(LF)(SN)_JST	BM08B-SRSS-TB(LF) (SN)CONN_BM08B-SRSS-TB(LF) (SN)_JST	CONN_BM08B-SRSS- TB(LF)(SN)_JST	J1	
6	DIODE	DIODEDO-214AA(SMB)	DIOM5336X265N	D14, D15, D16, D17, D18, D19	Diode Rectifier - Generic
1	GREEN	CHIP-FLAT-B_1206	LEDC3216X75N_FLAT- B	D9	
6	IPB035N08N3GATMA1	IPB035N08N3GATMA1	PG-T0263-3	Q4, Q5, Q6, Q7, Q8, Q9	MOSFET N- CH 80V 100A D2PAK
1	IPD034N06N3	IPD034N06N3	TO252	Q14	

3	IR2101S	IR2101S	SO08	IC6, IC7, IC8	HIGH AND LOW SIDE DRIVER
1	JD2912-1Z-12VDC	JD2912-1Z-12VDC	JD2912-1Z-12VDC	K1	
2	LM393-SPLIT	LM393-SPLIT	SOIC127P600X175-8	IC5, IC12	
2	LMR36520	LMR36520	HSOIC	IC2, IC3	
4	NC	R-EU_R0603	R0603	R27, R28, R36, R37	RESISTOR, European symbol
1	NC	R-EU_R1206	R1206	R67	RESISTOR, European symbol .100" (2.54mm) Center Header - 2 Pin
1	NTC 10K	22-23-2021	22-23-2021	X23	
5	PMEG6020AELRX	DIODE_MMSD4148T1G_SOD123G SOD3715X135		D4, D11, D12, D13, D23	Diode Rectifier - Popular parts
1	RED	CHIP-FLAT-B_1206	LEDC3216X75N_FLAT-B	D10	
2	SQ2389CES-T1	PMOSFET_SOT23-GSD	SOT23	Q15, Q27	P-Channel MOSFET - Generic
1	STM32F405RGT6LQFP64-10X10MM	STM32F405RGT6LQFP64-10X10MM	LQFP64-10X10MM	U1	
1	VPR138/94-M3	VPR138/94-M3	VPR138/94-M3	HS1	
4	WSLP2726	WSLP2726	WSLP2726	R1, R56, R60, R64	
3	Yellow	CHIP-FLAT-B_1206	LEDC3216X75N_FLAT-B	D5, D6, D7	
1	ZM12V6	ZENER_TZM5231BDO-213AA(SOD-80)	DIOMELF3516	D2	Zener Diode - Popular parts
1	ZM5V6	ZENER_TZM5231BDO-213AA(SOD-80)	DIOMELF3516	D3	Zener Diode - Popular parts

15. Source Code

15.1. Main.c/h

```
1 /* USER CODE BEGIN Private defines */
2 //Machine States
3 #define READY 0
4 #define DRIVE 1
5 #define BREAK 2
6 #define SWFAULT 3
7 #define HWFAULT 4
8 #define DEBUGST 6
9 //BLDC
10 #define THROTTLE_THRESHOLD 30 //Threshold for initial start to prevent
    adc drift
11 #define MAXDUTY 100 //Motor PWM Duty Cycle
12 #define MINDUTY 0 //Motor PWM Duty Cycle
13 #define MAXADC 3170 //throttle MAX 2,55V
14 #define MINADC 1055 //throttle MIN 0.85V
15 #define MAXRPM 4500 //Max RPM of the given motor
16 #define MINRPM 25 //Min RPM of the given motor
17 //Software limits for ADC measurements
18 #define SW_OC 28 //Over Current 28A
19 #define SW_UV 44 //Under Voltage 44V
20 #define SW_OV 60 //Over Voltage 60V
21 #define SW_OT 100 //Over Temperature 100C
22 #define Temp_FAN_ON 80 //Turn on fan here 80C
23 #define Temp_FAN_OFF 60 //Turn off fan here 60C
24 //Blinker PWM
25 #define BLINKER_START 250 // Run blinker at nom. PWM
26 #define BLINKER_STOP 500 // Turns off blinker PWM
27 //other
28 #define ADC_TIMEOUT 20
29 #define SWFAULT_TIMEOUT 300
30 /* USER CODE END Private defines */
```

Source Code 15.1: **Private Defines**

Die Benutzerdefiniert-Konstanten dienen der Konfiguration von Grenzwerten und Betriebsparametern. Änderungen sind zentral und einfach möglich.

```

1  /* USER CODE BEGIN WHILE */
2  while (1)
3  {
4      //Read inputs
5      readADCs();
6      readDI();
7      doADCs();
8      /* STATE MACHINE */
9      switch(STATE){
10          case READY:
11              ready();
12              setDO();
13              break;
14          case DRIVE:
15              drive();
16              setDO();
17              break;
18          case BREAK:
19              breaking();
20              setDO();
21              break;
22          case SWFAULT:
23              resetDO();
24              swfault();
25              break;
26          case HWFAULT:
27              resetDO();
28              hwfault();
29              break;
30          case DEBUGST:
31              debug();
32              setDO();
33              break;
34          default:
35              resetDO();
36              hwfault();
37              break;
38      }
39      //Update LCD every 500ms
40      if(timcc >=5){
41          update_lcd_val(&lcd_val,ADC_VAL);
42          writeState();
43          timcc = 0;
44      }
45      HAL_Delay(25);
46      /* USER CODE END WHILE */
47 }

```

Source Code 15.2: Main Loop

Die Endlosschleife (while (1)) ist der zentrale Ablauf des Programms und führt kontinuierlich die Eingabelesung, Zustandsmaschine und die Anzeigeaktualisierung aus.

```

1  /**
2   * @brief Reads ADC values for voltage, current, and temperature, and
3   * updates corresponding variables
4   * @param None
5   * @retval void
6   */
7  void readADCs(){
8      uint16_t a =0; //Int dump for ADC
9      float x = 0.0;
10     //READ Voltage
11     ADC3_Select_CH(0);
12     HAL_ADC_Start(&hadc3);
13     HAL_ADC_PollForConversion(&hadc3, ADC_TIMEOUT);
14     a =HAL_ADC_GetValue(&hadc3);
15     HAL_ADC_Stop(&hadc3);
16     x = adc_volt(a);
17     ADC_VAL[0] = 0.15 * x + (1.0 - 0.05)*ADC_VAL[0];
18
19     //READ Current
20     ADC3_Select_CH(1);
21     HAL_ADC_Start(&hadc3);
22     HAL_ADC_PollForConversion(&hadc3, ADC_TIMEOUT);
23     a =HAL_ADC_GetValue(&hadc3);
24     HAL_ADC_Stop(&hadc3);
25     x = adc_volt(a);
26     ADC_VAL[1] = 0.15 * x + (1.0 - 0.05)*ADC_VAL[1];
27
28     //READ Temperature
29     ADC3_Select_CH(2);
30     HAL_ADC_Start(&hadc3);
31     HAL_ADC_PollForConversion(&hadc3, ADC_TIMEOUT);
32     a =HAL_ADC_GetValue(&hadc3);
33     HAL_ADC_Stop(&hadc3);
34     x = adc_volt(a);
35     ADC_VAL[2] = 0.15 * x + (1.0 - 0.05)*ADC_VAL[2];
36 }

```

Source Code 15.3: **void readADCs()**

```

1 /**
2  * @brief Processes ADC values to monitor and control system states,
3  * including fault detection and fan control
4  * @param None
5  * @retval void
6 */
7 void doADCs() {
8     // Ensure ADC_VAL array has enough elements
9     if (sizeof(ADC_VAL) / sizeof(ADC_VAL[0]) < 3) {
10         // Handle error (e.g., log it, set state to fault)
11         STATE = SWFAULT;
12         HD44780_SetCursor(0, 1);
13         HD44780_PrintStr("ERR:ADC VAL SIZE");
14         return;
15     }
16     // Release SW_FAULT if all conditions are normal
17     if (STATE == SWFAULT) {
18         if (ADC_VAL[0] < SW_OV && ADC_VAL[0] > SW_UV + 2 && // Voltage
19             OK
20             ADC_VAL[1] <= 1 && // Current
21             OK
22             ADC_VAL[2] <= Temp_FAN_ON) { // Temperature OK
23                 STATE = BREAK;
24             }
25     }
26     // Device is in normal condition
27     if (STATE != SWFAULT) {
28         // Control the fan based on temperature
29         if (ADC_VAL[2] >= Temp_FAN_ON) {
30             HAL_GPIO_WritePin(GPIOB, PB8_DO_FAN_Pin, GPIO_PIN_RESET); // FAN ON
31         } else if (ADC_VAL[2] <= Temp_FAN_OFF) {
32             HAL_GPIO_WritePin(GPIOB, PB8_DO_FAN_Pin, GPIO_PIN_SET); // FAN OFF
33         }
34         // Check for software faults and set error messages
35         if (ADC_VAL[0] >= SW_OV) {
36             setFaultState("ERR:SW OV");
37         } else if (ADC_VAL[0] <= SW_UV) {
38             setFaultState("ERR:SW UV");
39         } else if (ADC_VAL[1] >= SW_OC) {
40             setFaultState("ERR:SW OC");
41         } else if (ADC_VAL[2] >= SW_OT) {
42             setFaultState("ERR:SW OT");
43         }
44     }
45 }

```

Source Code 15.4: **void doADCs()**

```

1 /**
2  * @brief Reads digital input values from GPIO pins and updates button
3  * states
4  * @param None
5  * @retval void
6 */
7 void readDI(){
8     // Button data[Light, Blinker L, Blinker R, Aux]
9     uint16_t but_new[4];
10    but_new[0] = (GPIOC->IDR & GPIO_IDR_ID10) ? 0x0001 : 0x0000;
11    but_new[1] = (GPIOC->IDR & GPIO_IDR_ID11) ? 0x0001 : 0x0000;
12    but_new[2] = (GPIOC->IDR & GPIO_IDR_ID12) ? 0x0001 : 0x0000;
13    but_new[3] = (GPIOC->IDR & GPIO_IDR_ID13) ? 0x0001 : 0x0000;
14
15    for (int i = 0; i < 4; i++) {
16        but[i] = but_new[i];
17    }
18}

```

Source Code 15.5: **void readDI()**

```

1 /**
2  * @brief Sets digital output states based on button inputs and updates
3  * system behavior accordingly
4  * @param None
5  * @retval void
6 */
7
8 void setDO() {
9     // Data error
10    if (sizeof(but) / sizeof(but[0]) < 4) {
11        setFaultState("ERR: DO SIZE");
12    }
13
14    // Button data: [Light, Blinker L, Blinker R, Aux]
15    // Toggle lights based on button state
16    HAL_GPIO_WritePin(GPIOB, PB3_DO_LIGHT_Pin, but[0] == 0 ?
17        GPIO_PIN_RESET : GPIO_PIN_SET);
18
19    // Handle Blinker Left (PWM Control)
20    if (but[1] == 1) {
21        TIM3->CCR2 = BLINKER_START; // Start PWM
22    } else {
23        TIM3->CCR2 = BLINKER_STOP; // Stop PWM
24    }
25
26    // Handle Blinker Right (PWM Control)
27    if (but[2] == 1) {
28        TIM3->CCR1 = BLINKER_START; // Start PWM
29    } else {
30        TIM3->CCR1 = BLINKER_STOP; // Stop PWM
31    }
32
33    // Handle Aux button action
34    if (but[3] == 0) {
35        HAL_GPIO_TogglePin(PB1_LED_RED_GPIO_Port, PB1_LED_RED_Pin); // //
```

Source Code 15.6: **void setDO()**

```

1 /**
2  * @brief Prepares the system for motor operation by initializing
3  * throttle input and PWM control
4  * @param None
5  * @retval void
6  */
7 void ready() {
8     // Start ADC conversion
9     HAL_ADC_Start(&hadc1);
10    HAL_ADC_PollForConversion(&hadc1, 20);
11
12    // Get throttle raw value
13    rawThrot = HAL_ADC_GetValue(&hadc1);
14
15    // Map throttle value to duty cycle range
16    int THrotduty = map(rawThrot, MINADC, MAXADC, MINDUTY, MAXDUTY);
17
18    // Check if throttle duty cycle exceeds the threshold
19    if (THrotduty >= THROTTLE_THRESHOLD) {
20        // Initialize PWM outputs to zero
21        TIM1->CCR1 = 0;
22        TIM1->CCR2 = 0;
23        TIM1->CCR3 = 0;
24
25        // Start PWM for all three channels
26        HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_1);
27        HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_2);
28        HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_3);
29
30        // Initialize BLDC motor
31        initBLDC();
32
33        // Set initial duty cycle
34        duty = 10; // Set initial duty cycle; modify as needed
35
36        // Change state to DRIVE
37        STATE = DRIVE;
38    }
}

```

Source Code 15.7: **void ready()**

```

1 /**
2  * @brief Controls the motor drive by reading throttle input and
3  * updating the PWM duty cycle
4  * @param None
5  * @retval void
6 */
7 void drive() {
8     // Start ADC conversion
9     HAL_ADC_Start(&hadc1);
10
11    // Wait for ADC conversion to complete with a timeout
12    if (HAL_ADC_PollForConversion(&hadc1, ADC_TIMEOUT) == HAL_OK) {
13        // Get raw throttle value
14        rawThrot = HAL_ADC_GetValue(&hadc1);
15
16        // Map the raw throttle value to a duty cycle range
17        int THrotDuty = map(rawThrot, MINADC, MAXADC, MINDUTY, MAXDUTY);
18
19        // Update the duty cycle
20        duty = THrotDuty;
21    } else {
22        // Handle ADC conversion error (optional)
23        duty = 0; // Set duty to a safe value in case of failure
24        setFaultState("ERR: ADC");
25        // Optionally log or display an error message
26    }
27}

```

Source Code 15.8: **void drive()**

```

1 /**
2  * @brief Engages the braking mechanism by stopping PWM outputs and
3  * activating braking GPIO pins
4  * @param None
5  * @retval void
6 */
7 void breaking() {
8     // Stop all PWM channels
9     TIM1->CCR1 = 0;
10    TIM1->CCR2 = 0;
11    TIM1->CCR3 = 0;
12    duty = 0; // Reset duty cycle to zero
13
14    HAL_TIM_PWM_Stop(&htim1, TIM_CHANNEL_1);
15    HAL_TIM_PWM_Stop(&htim1, TIM_CHANNEL_2);
16    HAL_TIM_PWM_Stop(&htim1, TIM_CHANNEL_3);
17
18    // Set GPIO pins for braking mode
19    HAL_GPIO_WritePin(GPIOB, PB13_U_Pin, GPIO_PIN_SET);
20    HAL_GPIO_WritePin(GPIOB, PB14_V_Pin, GPIO_PIN_SET);
21    HAL_GPIO_WritePin(GPIOB, PB15_W_Pin, GPIO_PIN_SET);
22}

```

Source Code 15.9: **void breaking()**

```

1 /**
2  * @brief Handles the software fault state by engaging braking and
3  * transitioning to HWFAULT after a timeout
4  * @param None
5  * @retval void
6 */
7 void swfault() {
8     // Perform breaking to ensure the system is in a safe state
9     braking();
10
11    // Check if the timeout for SWFAULT has elapsed (30s = 300 units of
12    // 100ms)
13    if (swfault_time_counter >= SWFAULT_TIMEOUT) {
14        // Transition to HWFAULT state
15        STATE = HWFAULT;
16    }
17}

```

Source Code 15.10: **void swfault()**

```

1 /**
2  * @brief Handles hardware faults by engaging braking, displaying an
3  * error message, and halting execution
4  * @param None
5  * @retval void
6 */
7 void hwfault(){
8     STATE = HWFAULT;
9     braking();
10    HD44780_SetCursor(0,1);
11    HD44780_PrintStr("ERROR:HW FAULT");
12    // Stuck until Power on reset
13    while(1){
14        HAL_Delay(100);
15    }
16}

```

Source Code 15.11: **void hwfault()**

```

1 // Interrupt Pin Function
2 void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
3     // Handle Break Signal Interrupt
4     if (GPIO_Pin == EXTI5_Break_Pin) {
5         handleBreakInterrupt();
6     }
7     // Handle Hardware Fault Interrupt
8     else if (GPIO_Pin == EXTI9_FAULT_Pin) {
9         handleHardwareFaultInterrupt();
10    }
11    // Handle Hall Sensor Interrupt
12    else if (STATE == DRIVE || STATE == READY) {
13        if (GPIO_Pin == EXTI6_HALL_U_Pin || GPIO_Pin == EXTI7_HALL_V_Pin
14        || GPIO_Pin == EXTI8_HALL_W_Pin) {
15            handleHallSensorInterrupt(GPIO_Pin);
16        }
17    } else {
18        // No operation for other states
19        __NOP();
20    }
21}

```

Source Code 15.12: **void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)**

```

1 /**
2  * @brief Handles hardware fault interrupts, transitioning motor state
3  * to HWFAULT
4  * @param None
5  * @retval void
6  */
7 void handleHardwareFaultInterrupt() {
8     STATE = HWFAULT;
9     // Optional: Identify Fault Reason and shutdown Device
10    // Example readADCs() and determine fault cause;
11}

```

Source Code 15.13: **void handleHardwareFaultInterrupt()**

```

1 /**
2  * @brief Handles the brake interrupt signal , transitioning motor state
3  * between BREAK and READY
4  * @param None
5  * @retval void
6 */
7 void handleBreakInterrupt() {
8     uint16_t breakSignal = (GPIOC->IDR & GPIO_IDR_ID5) ? 0x0001 : 0x0000
9     ;
10    if (breakSignal == 0) {
11        // Transition to BREAK state
12        STATE = BREAK;
13        HAL_GPIO_WritePin(PB0_LED_GREEN_GPIO_Port, PB0_LED_GREEN_Pin,
14                           GPIO_PIN_RESET);
15        HAL_GPIO_WritePin(PB1_LED_RED_GPIO_Port, PB1_LED_RED_Pin,
16                           GPIO_PIN_SET);
17    } else {
18        // Transition to READY state if motor is fully stopped
19        if(rpm >=5){
20            STATE = DRIVE;
21        }else{
22            STATE = READY;
23        }
24    }
25 }
```

Source Code 15.14: **void handleBreakInterrupt()**

```

1 /**
2  * @brief Handles hall sensor interrupts, updates commutator step for
3  * motor control
4  * @param GPIO_Pin      = Pin number that triggered the interrupt
5  * @retval void
6 */
7 void handleHallSensorInterrupt(uint16_t GPIO_Pin) {
8     hallCC++; // Increment hall sensor counter
9
10    // Read hall sensor states
11    uint16_t hall[3];
12    hall[0] = (GPIOC->IDR & GPIO_IDR_ID6) ? 0x0001 : 0x0000; // Sensor U
13    hall[1] = (GPIOC->IDR & GPIO_IDR_ID7) ? 0x0001 : 0x0000; // Sensor V
14    hall[2] = (GPIOC->IDR & GPIO_IDR_ID8) ? 0x0001 : 0x0000; // Sensor W
15
16    // Trapezoidal control
17    uint16_t commutatorStep = hallState(hall);
18    commutator(commutatorStep, duty, dir);
19
20    // Optional: Implement sine control as needed
21    /*
22     * uint16_t commutatorStep = hallState(hall);
23     * uint16_t phaseAngle = electricalAngle(commutatorStep);
24     * FOCcommutator(phaseAngle, duty);
25     */
26}

```

Source Code 15.15: **void handleHallSensorInterrupt(uint16_t GPIO_Pin)**

15.2. BLDC.c/h

```
1  /**
2   * @brief Takes hallsensor states and determines the commutator
3   * position
4   * @param hall[]      = arry containing the values read from the 3 Hall
5   * Sensors
6   * @retval      = The Communator step between 0-5 used to set BLDC gates
7   * Hall Sensor State Rotor Position Electrical Angle (Degrees)
8   *      001    0 - 60      0 - 1440
9   *      101    60 - 120    1440 - 2880
10  *      100    120 - 180    2880 - 4320
11  *      110    180 - 240    4320 - 5760
12  *      010    240 - 300    5760 - 7200
13  *      011    300 - 360    7200 - 8640
14 */
15
16 uint16_t hallState(uint16_t hall[]){
17
18     uint16_t commutatorState = -1;
19     if ((hall[0] == 1) && (hall[1] == 0) && (hall[2] == 1)) {
20         //Mechanical Angle 0-60
21         commutatorState = 0;
22     }
23     else if ((hall[0] == 0) && (hall[1] == 0) && (hall[2] == 1)) {
24         //Mechanical Angle 60-120
25         commutatorState = 1;
26     }
27     else if ((hall[0] == 0) && (hall[1] == 1) && (hall[2] == 1)) {
28         //Mechanical Angle 120-180
29         commutatorState = 2;
30     }
31     else if ((hall[0] == 0) && (hall[1] == 1) && (hall[2] == 0)) {
32         //Mechanical Angle 180-240
33         commutatorState = 3;
34     }
35     else if ((hall[0] == 1) && (hall[1] == 1) && (hall[2] == 0)) {
36         //Mechanical Angle 240-300
37         commutatorState = 4;
38     }
39     else if ((hall[0] == 1) && (hall[1] == 0) && (hall[2] == 0)) {
40         //Mechanical Angle 300-360
41         commutatorState = 5;
42     }
43     return commutatorState;
44 }
```

Source Code 15.16: `uint16_t hallState(uint16_t hall[])`

```

1 /**
2 * @brief Sets Three Phase gate driver according to commutator position
3 * @param commutatorStep      = 0-5 determined by FOC hall sensor
4 * @param duty                = PWM duty cycle 0-100
5 * @param dir                 = motor direction: 1 = fwr; -1= rws
6 * @retval void
7 */
8 void commutator(int commutatorStep, int duty, int dir){
9
10 if(dir ==1){
11     switch(commutatorStep){
12
13     //U PWM
14     //Hall state:      U = HIGH   V = LOW    W = HIGH
15     //Phase Current:  U = I      V = -I     W = 0
16     //High Side       U = PWM    V = OFF    W = OFF
17     //Low Side        U = OFF   V = ON     W = OFF
18     case 0:
19         //Low Side
20         HAL_GPIO_WritePin(GPIOB,PB13_U_Pin,GPIO_PIN_RESET);
21         HAL_GPIO_WritePin(GPIOB,PB14_V_Pin,GPIO_PIN_SET);
22         HAL_GPIO_WritePin(GPIOB,PB15_W_Pin,GPIO_PIN_RESET);
23         //High Side
24         TIM1->CCR1 = duty;
25         TIM1->CCR2 = 0;
26         TIM1->CCR3 = 0;
27
28         break;
29
30     //W PWM
31     //Hall state:      U = LOW    V = LOW    W = HIGH
32     //Phase Current:  U = 0      V = -I     W = I
33     //High Side       U = OFF    V = OFF    W = PWM
34     //Low Side        U = OFF   V = ON     W = OFF
35     case 1:
36         //Low Side
37         HAL_GPIO_WritePin(GPIOB,PB13_U_Pin,GPIO_PIN_RESET);
38         HAL_GPIO_WritePin(GPIOB,PB14_V_Pin,GPIO_PIN_SET);
39         HAL_GPIO_WritePin(GPIOB,PB15_W_Pin,GPIO_PIN_RESET);
40         //High Side
41         TIM1->CCR1 = 0;
42         TIM1->CCR2 = 0;
43         TIM1->CCR3 = duty;
44
45         break;
46
47     //W PWM
48     //Hall state:      U = LOW    V = HIGH   W = HIGH
49     //Phase Current:  U = -I     V = 0      W = I
50     //High Side       U = OFF    V = OFF    W = PWM
51     //Low Side        U = ON     V = OFF    W = OFF
52     case 2:
53         //Low Side

```

```

54     HAL_GPIO_WritePin(GPIOB ,PB13_U_Pin ,GPIO_PIN_SET);
55     HAL_GPIO_WritePin(GPIOB ,PB14_V_Pin ,GPIO_PIN_RESET);
56     HAL_GPIO_WritePin(GPIOB ,PB15_W_Pin ,GPIO_PIN_RESET);
57     //High Side
58     TIM1->CCR1 = 0;
59     TIM1->CCR2 = 0;
60     TIM1->CCR3 = duty;
61
62     break;
63
64 //V PWM
65 //Hall state:      U = LOW      V = HIGH    W = LOW
66 //Phase Current:   U = -I       V = I       W = 0
67 //High Side        U = OFF      V = PWM     W = OFF
68 //Low Side         U = ON       V = OFF     W = OFF
69 case 3:
70     //Low Side
71     HAL_GPIO_WritePin(GPIOB ,PB13_U_Pin ,GPIO_PIN_SET);
72     HAL_GPIO_WritePin(GPIOB ,PB14_V_Pin ,GPIO_PIN_RESET);
73     HAL_GPIO_WritePin(GPIOB ,PB15_W_Pin ,GPIO_PIN_RESET);
74     //High Side
75     TIM1->CCR1 = 0;
76     TIM1->CCR2 = duty;
77     TIM1->CCR3 = 0;
78
79     break;
80
81 //V PWM
82 //Hall state:      U = HIGH;    V = HIGH    W = LOW
83 //Phase Current:   U = 0        V = I       W = -I
84 //High Side        U = OFF      V = PWM     W = OFF
85 //Low Side         U = OFF      V = OFF     W = ON
86 case 4:
87     //Low Side
88     HAL_GPIO_WritePin(GPIOB ,PB13_U_Pin ,GPIO_PIN_RESET);
89     HAL_GPIO_WritePin(GPIOB ,PB14_V_Pin ,GPIO_PIN_RESET);
90     HAL_GPIO_WritePin(GPIOB ,PB15_W_Pin ,GPIO_PIN_SET);
91     //High Side
92     TIM1->CCR1 = 0;
93     TIM1->CCR2 = duty;
94     TIM1->CCR3 = 0;
95
96     break;
97
98 //U PWM
99 //Hall state:      U = HIGH    V = LOW     W = LOW
100 //Phase Current:  U = I       V = 0       W = -I
101 //High Side        U = PWM     V = OFF     W = OFF
102 //Low Side         U = OFF     V = OFF     W = ON
103 case 5:
104     //Low Side
105     HAL_GPIO_WritePin(GPIOB ,PB13_U_Pin ,GPIO_PIN_RESET);
106     HAL_GPIO_WritePin(GPIOB ,PB14_V_Pin ,GPIO_PIN_RESET);
107     HAL_GPIO_WritePin(GPIOB ,PB15_W_Pin ,GPIO_PIN_SET);

```

```

108     //High Side
109     TIM1->CCR1 = duty;
110     TIM1->CCR2 = 0;
111     TIM1->CCR3 = 0;
112
113     break;
114
115     //i have no idea how you ended up here pls stop motor
116     //High Side      U = OFF    V = OFF    W = OFF
117     //Low Side       U = HIGH   V = HIGH   W = HIGH
118     default:
119         HAL_GPIO_WritePin(GPIOB,PB13_U_Pin,GPIO_PIN_SET);
120         HAL_GPIO_WritePin(GPIOB,PB14_V_Pin,GPIO_PIN_SET);
121         HAL_GPIO_WritePin(GPIOB,PB15_W_Pin,GPIO_PIN_SET);
122
123         TIM1->CCR1 = 0;
124         TIM1->CCR2 = 0;
125         TIM1->CCR3 = 0;
126         break;
127     }
128 }
129 if(dir !=1){
130     //no implimentation for reverse needed yet just break
131     HAL_GPIO_WritePin(GPIOB,PB13_U_Pin,GPIO_PIN_SET);
132     HAL_GPIO_WritePin(GPIOB,PB14_V_Pin,GPIO_PIN_SET);
133     HAL_GPIO_WritePin(GPIOB,PB15_W_Pin,GPIO_PIN_SET);
134
135     TIM1->CCR1 = 0;
136     TIM1->CCR2 = 0;
137     TIM1->CCR3 = 0;
138 }
139 }
```

Source Code 15.17: **void commutator(int commutatorStep, int duty, int dir)**

```

1 /**
2  * @brief Initializes the BLDC motor by reading hall sensor values and
3  * setting the commutator step
4  * @param None
5  * @retval void
6  */
7 void initBLDC(){
8     uint16_t hall[3];
9     hall[0] = (GPIOC->IDR & GPIO_IDR_ID6)? 0x0001 : 0x0000; // Sensor A
10    hall[1] = (GPIOC->IDR & GPIO_IDR_ID7)? 0x0001 : 0x0000; // Sensor B
11    hall[2] = (GPIOC->IDR & GPIO_IDR_ID8)? 0x0001 : 0x0000; // Sensor C
12
13    int step= hallState(hall);
14    commutator(step, 15,1);
15 }
```

Source Code 15.18: **void initBLDC()**

```

1 /**
2  * @brief Engages the brake for the BLDC motor by setting all output
3  * phases high and stopping PWM signals
4  * @param None
5  * @retval void
6  */
7 void BLDCbreak(){
8     HAL_GPIO_WritePin(GPIOB ,PB13_U_Pin ,GPIO_PIN_SET);
9     HAL_GPIO_WritePin(GPIOB ,PB14_V_Pin ,GPIO_PIN_SET);
10    HAL_GPIO_WritePin(GPIOB ,PB15_W_Pin ,GPIO_PIN_SET);
11
12    TIM1->CCR1 = 0;
13    TIM1->CCR2 = 0;
14    TIM1->CCR3 = 0;
15 }
```

Source Code 15.19: **void BLDCbreak()**

15.3. liquidcrystal_i2c.c/h

```
1 /* lcd struct */
2 typedef struct{
3 //Values for updating the display
4     char volt[4] /*Storage DC Voltage */;
5     char temp[3] /*Storage Drive temperature*/;
6     char amp[4] /*Storage DC Current*/;
7     char speed[2] /*Storage bike Speed*/;
8     char erpm[4] /*Storage Motor rpm*/;
9     char pwm[2] /*Storage PWM value*/;
10
11 //Cursor position for respective Values
12     uint8_t cur_volt[2];
13     uint8_t cur_temp[2];
14     uint8_t cur_amp[2];
15     uint8_t cur_speed[2];
16     uint8_t cur_erpm[2];
17     uint8_t cur_pwm[2];
18 } lcd_ar;
```

Source Code 15.20: **lcd struct**

```
1 /**
2 * @brief Sets the cursor position on the HD44780 LCD display
3 * @param col      = Column position (0-based)
4 * @param row      = Row position (0-based)
5 * @retval void
6 */
7 void HD44780_SetCursor(uint8_t col, uint8_t row)
8 {
9     int row_offsets[] = { 0x00, 0x40, 0x14, 0x54 };
10    if (row >= dpRows)
11    {
12        row = dpRows -1;
13    }
14    SendCommand(LCD_SETDDRAMADDR | (col + row_offsets[row]));
15 }
```

Source Code 15.21: **void HD44780_SetCursor(uint8_t col, uint8_t row)**

```
1 /**
2 * @brief Prints a string to the HD44780 LCD display
3 * @param c      = Pointer to a null-terminated string to display
4 * @retval void
5 */
6 void HD44780_PrintStr(const char c[])
7 {
8     while(*c) SendChar(*c++);
9 }
```

Source Code 15.22: **void HD44780_PrintStr(const char c[])**

```

1 /**
2  * @brief Initializes the lcd_ar struct with default cursor positions
3  * and display values
4  * @param lcd      = Pointer to an lcd_ar struct to initialize
5  * @retval void
6 */
7 void Init_lcd_ar(lcd_ar* lcd){
8     //Load cursor positions
9     lcd->cur_volt[0] =15;
10    lcd->cur_volt[1] =0;
11
12    lcd->cur_temp[0]=0;
13    lcd->cur_temp[1]=0;
14
15    lcd->cur_amp[0] =15;
16    lcd->cur_amp[1] =1;
17
18    lcd->cur_speed[0]=9;
19    lcd->cur_speed[1]=3;
20
21    lcd->cur_erpm[0]=0;
22    lcd->cur_erpm[1]=3;
23
24    lcd->cur_pwm[0]=1;
25    lcd->cur_pwm[1]=2;
26
27    //Set up Display for array mode
28    //Init Volt
29    HD44780_SetCursor(15,0);
30    HD44780_PrintStr("42.0V");
31    //Init Temp
32    HD44780_SetCursor(0,0);
33    HD44780_PrintStr("069C");
34    //Init Current
35    HD44780_SetCursor(15,1);
36    HD44780_PrintStr("02.5A");
37    //Init Speed
38    HD44780_SetCursor(9,3);
39    HD44780_PrintStr("10KM/H");
40    //Init ERPM
41    HD44780_SetCursor(0,3);
42    HD44780_PrintStr("4000");
43    //Init PWM
44    //HD44780_SetCursor(1,2);
45    //HD44780_PrintStr("00%");
46}

```

Source Code 15.23: **void Init_lcd_ar(lcd_ar* lcd)**

```

1 /**
2  * @brief Updates the lcd_ar struct and LCD display with new sensor
3  * values
4  * @param ar      = Pointer to an lcd_ar struct containing cursor
5  * positions
6  * @param val     = Array of float values to update [Voltage, Current,
7  * Temperature, Speed, (optional: PWM)]
8  * @retval void
9  */
10 void update_lcd_val(lcd_ar* ar, float val[]){
11     //Set Voltage
12     sprintf(ar->volt, 5, "%04.1f", val[0]);
13     HD44780_SetCursor(ar->cur_volt[0],ar->cur_volt[1]);
14     HD44780_PrintStr(ar->volt);
15     //Set Current
16     sprintf(ar->amp, 5, "%04.1f", val[1]);
17     HD44780_SetCursor(ar->cur_amp[0],ar->cur_amp[1]);
18     HD44780_PrintStr(ar->amp);
19     //Set Temp
20     sprintf(ar->temp, 4, "%03.0f", val[2]);
21     HD44780_SetCursor(ar->cur_temp[0],ar->cur_temp[1]);
22     HD44780_PrintStr(ar->temp);
23     //Set Speed
24     sprintf(ar->speed, 3, "%02.0f",val[3] );
25     HD44780_SetCursor(ar->cur_speed[0],ar->cur_speed[1]);
26     HD44780_PrintStr(ar->speed);
27     //Set ERPM
28     sprintf(ar->erpm, 5, "%04.0f",rpm);
29     HD44780_SetCursor(ar->cur_erpm[0],ar->cur_erpm[1]);
30     HD44780_PrintStr(ar->erpm);
31     //Set PWM
32     //sprintf(ar->pwm,3,"%02.0f",val[4] );
33 }

```

Source Code 15.24: **void update_lcd_val(lcd_ar* ar, float val[])**

15.4. Mymath.c/h

```
1 typedef struct{
2     float max /*! Max manipulated value */;
3     float min /*! Minimum manipulated value */;
4     float e /*! Error value */;
5     float i /*! Integrator value */;
6     float kp /*! Proportional constant */;
7     float ki /*! Integrator constant */;
8     float kd /*! Differential constant */;
9 } pid_f_t;
```

Source Code 15.25: pid struct

```
1 /**
2 * @brief Re-maps a number from one range to another
3 * @param x      = The number to map.
4 * @param in_min = The lower bound of the values current range.
5 * @param in_max = The upper bound of the values current range.
6 * @param out_min = The lower bound of the values target range.
7 * @param out_max = The upper bound of the values target range.
8 * @retval       = The mapped value
9 */
10 uint16_t map(uint16_t x, uint16_t in_min, uint16_t in_max, uint16_t
11   out_min, uint16_t out_max) {
12     return (x - in_min) * (out_max - out_min) / (in_max - in_min) +
13       out_min;
14 }
```

Source Code 15.26: uint16_t map()

```
1 /**
2 * @brief Converts a raw ADC value to the corresponding input voltage
3 * @param val      = Raw 12-bit ADC value (0-4095)
4 * @retval float   = Calculated input voltage
5 */
6 float adc_volt(uint16_t val){
7   //((val/(39000+2200))*2200) /12bitADC
8   float Vcc = 3.3;
9   float R1 = 37000.0;
10  float R2 = 2200.0;
11  // Convert ADC value to voltage across R2
12  float Vout = (val / 4095.0) * Vcc;
13
14  // Calculate the total voltage across R1 and R2
15  float Vin = Vout * (R1 + R2) / R2;
16
17  return Vin;
18 }
```

Source Code 15.27: float adc_volt(uint16_t val)

```

1 /**
2  * @brief Converts a raw ADC value to the corresponding current based
3  * on shunt resistor and amplification factor
4  * @param val      = Raw 12-bit ADC value (0-4095)
5  * @retval float   = Calculated current in amperes
6 */
7 float adc_cur(uint16_t val){
8     float Vcc = 3.3;
9     float amplification_factor = 20.0;
10    float R_shunt = 0.004;
11    float offset_voltage = Vcc / 2.0;
12
13    // Convert ADC value to the amplified voltage
14    float Vadc = (val / 4095.0) * Vcc;
15
16    // Adjust for the offset voltage
17    float Vadc_adjusted = Vadc - offset_voltage;
18
19    // Determine the actual voltage drop across the shunt
20    float Vshunt = Vadc_adjusted / amplification_factor;
21
22    // Calculate the current through the shunt
23    float current = Vshunt / R_shunt;
24    return current;
}

```

Source Code 15.28: **float adc_cur(uint16_t val)**

```

1 /**
2  * @brief Converts a raw ADC value to the corresponding temperature in
3  * Celsius using an NTC thermistor
4  * @param val      = Raw 12-bit ADC value (0-4095)
5  * @retval float   = Calculated temperature in Celsius
6 */
7 float adc_temp(uint16_t val){
8     float Vcc = 3.3;
9     float R2 = 10000.0;
10    float T0 = 298.15; // 25C in Kelvin
11    float R0 = 10000.0; // Resistance at 25C
12    float B = 2904.0; // Beta parameter
13
14    // Convert ADC value to voltage
15    float Vadc = (val / 4095.0) * Vcc;
16
17    // Calculate the resistance of the NTC thermistor
18    float R1 = R2 * (Vcc / Vadc - 1.0);
19    // Calculate temperature in Kelvin using B-parameter equation
20    float T = 1.0 / ((1.0 / T0) + (1.0 / B) * log(R1 / R0));
21    // Convert Kelvin to Celsius
22    float T_Celsius = T - 273.15;
23    return T_Celsius;
}

```

Source Code 15.29: **float adc_temp(uint16_t val)**

```

1 /**
2  * @brief Converts motor RPM to speed in kilometers per hour (km/h)
3  * based on wheel circumference
4  * @param rpm      = Motor RPM (revolutions per minute)
5  * @retval float   = Calculated speed in kilometers per hour (km/h)
6 */
7
8 float rpm_tokmh(float rpm){
9     //36inch wheel
10    //91,44 cm wheel
11    //U = 2*pi*r
12    //U = 287,267cm
13    //U = 2,87267m
14    float circumference = 2.87267;
15    // distance traveled per minute in meters
16    float distance_per_minute = rpm * circumference;
17    // Convert distance to kilometers per hour
18    float speed_kmh = distance_per_minute * 60 / 1000;
19    return speed_kmh;
20 }
```

Source Code 15.30: **float rpm_tokmh(float rpm)**

15.5. stm32f4xx_it.c/h

```

1 /**
2  * @brief This function handles TIM2 global interrupt.
3 */
4 void TIM2_IRQHandler(void)
5 {
6     int x = hallCC*10;
7     float rev = x/24;
8     rpm = rev*60;
9     hallCC = 0;
10    ADC_VAL[3] = rpm_tokmh(rpm);
11    //HAL_GPIO_TogglePin(PB1_LED_RED_GPIO_Port, PB1_LED_RED_Pin);
12    if (timcc >= 10){
13        timcc = 0;
14    } else{
15        timcc++;
16    }
17    // Increment counter only if in SWFAULT state
18    if (STATE == SWFAULT) {
19        swfault_time_counter++;
20    } else {
21        // Reset the counter if leaving SWFAULT state
22        swfault_time_counter = 0;
23    }
24    HAL_TIM_IRQHandler(&htim2);
25 }
```

Source Code 15.31: **void TIM2_IRQHandler(void)**

16. Quellen

- Vedder, B. (2019). VESC 4.12 Open Source BLDC ESC. VESC-Project.
- C2000 Systems and Applications Team, Akin, B., Bhardwaj, M., & Warriner, J. (2015). Trapezoidal Control of BLDC Motors Using Hall Effect Sensors. Texas Instruments.
- Hein, M. (2020). Demystifying BLDC motor commutation: Trap, Sine, & FOC. Texas Instruments.
- Abacan, A. (2020). Sensored 3-Phase BLDC Motor Control Using Sinusoidal Drive. MicroChip.
- Simple Projects. (2017). Sensored brushless DC motor control. Simple Circuits.
- Stratify Labs. (2013). Motor Control using PWM and PID. Stratify Labs.

16.1. Danksagungen

Dieses Projekt wurde durch mehrere Ressourcen inspiriert und unterstützt. Ein besonderer Dank gilt allen Mitwirkenden und der Open-Source-Community für ihre wertvollen Ressourcen und ihre Unterstützung.