Tim Wiesner und Maximilian Grau

Design und Implementierung eines Backbones für ein autonomes Modellfahrzeug

FAKULTÄT FÜR
INFORMATIK

Intelligent Cooperative Systems
Computational Intelligence

Design und Implementierung eines Backbones für ein autonomes Modellfahrzeug

Bachelorarbeit

Tim Wiesner und Maximilian Grau

27. Juli 2020

Betreuende Professorin: Prof. Dr.-Ing. habil. Sanaz Mostaghim

Betreuer: Dr.-Ing. Christoph Steup

Tim Wiesner und Maximilian Grau:
Design und Implementierung eines Backbones für ein autonomes Modellfahrzeug
Otto-von-Guericke Universität
Intelligent Cooperative Systems
Computational Intelligence
Magdeburg, 2020.

Abstrakt

Diese Arbeit befasst sich mit dem Design und der Implementierung eines neuen Backbones für die nächste Generation des oTToCars, einem autonom agierenden Modellfahrzeug. Das Backbone stellt hier das Zusammenspiel zwischen den einzelnen Hardwareteilen in Form kleinerer Platinen und der dazugehörigen Firmware aus elektrischer und softwaretechnischer Sicht dar. Dabei wird für die Kommunikation zwischen den Platinen auf ein multi-master-fähiges Bussystem gesetzt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis

CAN	Controller Area Network
CMSIS	Cortex Microcontroller Software Interface Standard
CPU	Central Processing Unit
CTAD	Class Template Argument Deduction
DSDL	Data Structure Description Language
DTID	Data Type ID
EEPROM	Electrically Erasable Programmable Read-Only Memory
ESR	Equivalent Series Resistor
FIFO	First In First Out
GCC	GNU Compiler Collection
GDB	GNU Debugger
GPIO	General Purpose Input/Output
HAL	Hardware Abstraction Layer
HDMI	High Definition Multimedia Interface
I^2C	Inter-Integrated Circuit
IB	Interface Board
IC	Integrated Circuit

IDE Integrated Development Environment

ISR Interrupt Service Routine

LDO Low Dropout

LED Light-Emitting Diode

LiPo Lithium-Polymer

LQFP Low-profile Quad Flat Package

MCU Mikrocontroller

MMCX Micro-Miniature Coaxial

MosFET Metall-Oxid-Halbleiter-Feldeffekttransistor

NRZ Non Return to Zero

PDB Power Distribution Board

RF Radio Frequency

RTOS Real-time Operating System

SB Sensor Board

SMA SubMiniature version A

SPI Serial Peripheral Interface

STM STMicroelectronics

SWO Serial Wire Output

TI Texas Instruments Incorporated

TVS Transient Voltage Suppressor

UART Universal Asynchronous Receiver Transmitter

USB Universal Serial Bus

VESC Vedder Electronic Speed Controller

1 Einleitung

Das oTToCar ist ein autonom fahrendes Modellfahrzeug im Maßstab 1:10, das von einem Team Studierender der Otto-von-Guericke-Universität entwickelt und gebaut wird. Ziel des Projektes ist die erfolgreiche Teilnahme am Carolo-Cup der Technischen Universität Braunschweig, weshalb sich der technische Aufbau des Autos im Wesentlichen nach den Regularien dieses Wettbewerbs richtet.

1.1 Carolo-Cup

Der Carolo-Cup ist ein jährlich von der Technischen Universität Braunschweig veranstalteter Hochschulwettbewerb, bei dem studentische Teams mit autonom fahrenden Modellfahrzeugen im Maßstab 1:10 in verschiedenen Disziplinen gegeneinander antreten. Dabei werden die Fahrzeuge von den Studierenden selbstständig mit Hinblick auf Kosten- und Energieeffizienz konzipiert und konstruiert. Ziel des Wettbewerbs ist es, die jeweiligen Disziplinen möglichst schnell und fehlerfrei zu absolvieren [Tec].

Neben einer statischen Disziplin, die vor allem die Präsentation des Fahrzeugkonzepts sowie die Aufschlüsselung der Energie- und Herstellungskosten beinhaltet, gibt es auch die dynamischen Disziplinen, bei denen das Fahrzeug unterschiedliche Aufgaben auf einer vorgegebenen Strecke absolvieren muss. Die Art der Aufgaben richtet sich dabei nach der gewählten Wettbewerbsklasse: dem Basic-Cup oder dem Master-Cup.

Beim Basic-Cup muss im ersten Szenario ein einfacher Rundkurs gefahren werden; dabei kann optional auch bis zu zweimal eingeparkt werden, um mehr Punkte zu erzielen. Beim zweiten Szenario muss sowohl unbewegten als auch bewegten Hindernissen ausgewichen werden [Tec19a]. Der Master-Cup baut auf den Szenarien des Basic-Cups auf, erweitert diese aber unter anderem um

Verkehrszeichen, Steigungen, Fußgängerüberwegen, Verkehrsinseln und Überholverbotsabschnitten [Tec19b].

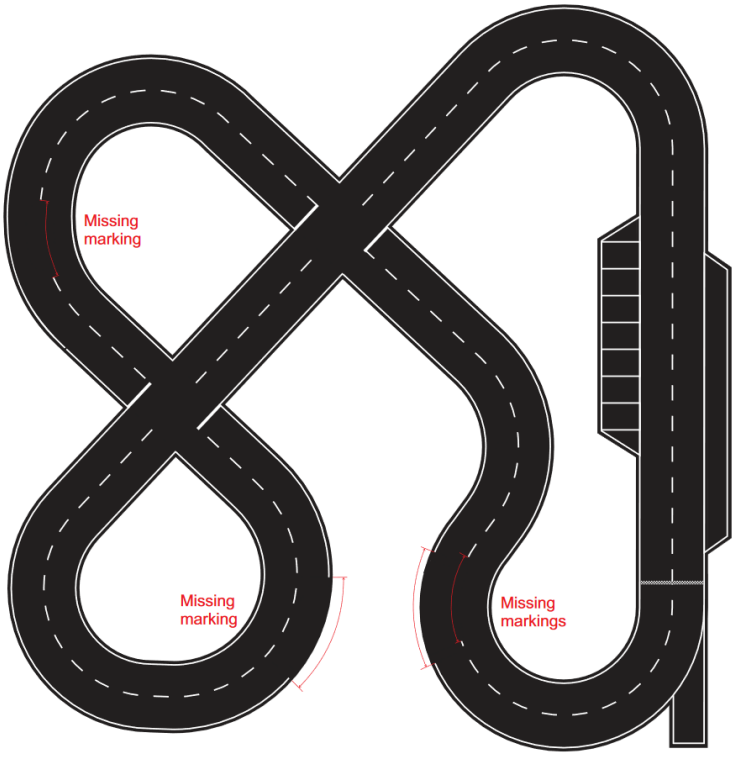


Abbildung 1.1: Beispielstrecke im Basic-Cup [Tec19a]

1.2 Aktueller Stand

Das aktuelle oTToCar der fünften Generation entstand im Jahr 2016 und stellt den zweiten Versuch dar, im Gegensatz zu den Vorgängergenerationen eine modulare Hardwareplattform aufzubauen. Statt auf eine einzelne große Platine, welche sich u. a. mit der Ansteuerung des Motors, der Sensorik und LED-Beleuchtung befasste, wurde auf sechs kleinere Platinen gesetzt, um die Aufgaben zu verteilen. Die Kommunikation zwischen den Platinen findet über einen RS485-Bus statt.

1.2.1 Bus

Die Funktionalität des Fahrzeugs ist auf mehrere Platinen verteilt, die über einen Bus miteinander kommunizieren. Der Bus ist als Master-Slave-Bus auf der Basis von RS485 ausgelegt. Dabei nimmt das *Mainboard*, die Brücke zwischen dem via Universal Serial Bus (USB) angeschlossenen Hauptrechner und dem Bussystem, die Rolle des Masters ein. Bedingt durch diese Architektur müssen sämtliche Aktionen vom Mainboard initiiert werden; andere Busteilnehmer können nicht selbstständig einen Nachrichtenaustausch beginnen. Das hat zur Folge, dass auch ein teilweiser Ausfall des Mainboards zum Totalausfall des Bussystems führen kann und das Fahrzeug in diesem Fall nicht einmal mehr per Fernbedienung gesteuert werden kann. Aufgrund einiger USB-bedingter Probleme konnte eine solche Funktionsstörung in der Vergangenheit bereits mehrfach beobachtet werden.

RS485 spezifiziert nur die elektrischen Eigenschaften des Busses [KUN]. Als Framing-Protokoll wird Universal Asynchronous Receiver Transmitter (UART) benutzt, da alle in den Busgeräten verbauten Mikrocontroller (MCUs) über entsprechende Schnittstellen verfügen. Das auf der Anwendungsschicht benutzte Protokoll ist eine Eigenentwicklung eines ehemaligen oTToCar-Teammitglieds. Die Beschreibung des Aufbaus der verschiedenen Busnachrichten erfolgt in einem eigens dafür angelegten Format, welches dann mit einem Python-Skript zu C-Code umgewandelt wird, der Funktionen für Serialisierung und Deserialisierung der jeweiligen Nachricht enthält. Für das Protokoll und insbesondere die Beschreibungssprache existieren jedoch keine Dokumentation, was eine Erweiterung des aktuellen Systems unmöglich macht. Versuche, das Format durch Reverse Engineering zu verstehen, scheiterten. Testweise vorgenommene Änderungen konnten nicht die gewünschten Ergebnisse erzielen; in einigen Fällen konnte das gesamte System nicht mehr starten.

1.2.2 Stromversorgung

Die Stromversorgung des oTToCars findet über zwei Lithium-Polymer-Akkus statt: ein größerer 4S-Akku wird als primäre Stromquelle für den Motor und die Recheneinheit „Brix“ genutzt; ein kleinerer 2S-Akku ist als sekundäre Stromquelle für die Platinen und Sensoren zuständig. Diese Designentscheidung wurde getroffen, um die ausfallempfindliche Elektrik des Autos so gut wie möglich von den Störungen des Motors zu entkoppeln. Des Weiteren benötigt die

Recheneinheit eine Spannung von 19 V, aus diesem Grund ist ein Aufwärtswandler (Boost-Converter) verbaut. Sowohl für die primären und sekundären Spannungsquellen existiert jeweils eine dedizierte 5V-Versorgung, die per Abwärtswandler (Buck-Converter) bereitgestellt wird. So beziehen energieintensive Lasten wie die LEDs den Strom von der primären 5V-Versorgung, während die Platinen und Sensoren den Strom von der sekundären, in der Regel rauschärmeren 5V-Versorgung beziehen. Die LiPo-Akkus können das Auto je nach Stromverbrauch des Bordcomputers und des Antriebs bis zu 30 Minuten lang mit Strom versorgen. Danach ist ein Wechsel der Akkus erforderlich, wozu das gesamte Fahrzeugsystem samt Linux heruntergefahren werden muss. Erst nach dem Tausch gegen einen voll geladenen Akku kann das System wieder hochgefahren werden, wofür allerdings eine gewisse Zeit benötigt wird.

1.2.3 Firmware

Alle Firmwares sind vollständig in C geschrieben. Als Firmwarebibliothek wird *libopencm3*[1], eine quelloffene Bibliothek, die viele Cortex-M-basierte Mikrocontroller unterstützt, verwendet. Diese wird jedoch nicht aktiv weiterentwickelt und enthält Bugs, die auch Einfluss auf unseren Anwendungszweck haben. Da analog zum Bussystem keine Dokumentation zu den Firmwares existiert, sämtlicher Code von einer einzigen Person geschrieben und dieser weder gut strukturiert noch nach gängigen Clean-Code-Konventionen geschrieben wurde, ist die Funktionsweise zum Teil nur schwer ersichtlich und hat bereits mehrmals zeitintensives Reverse Engineering erforderlich gemacht. Zudem wird ein einfacher, selbst geschriebener Scheduler benutzt, obwohl es viele kostenlose, etablierte Alternativen gibt, die auch sehr gut getestet sind.

1.3 Zielsetzung

Das Hauptziel dieser Arbeit ist das Designen und Implementieren eines neuen Backbones für die nächste oTToCar-Generation. Dazu wird für die Kommunikation zwischen den im Auto verbauten Platinen auf ein multi-masterfähiges Bussystem gesetzt. Dieses soll robust gegenüber Störungen und Fehlern sowohl in den Übertragungspaketen als auch in den Busteilnehmern sein.

[1] `http://libopencm3.org/`

Der Austausch der Daten zwischen den Platinen soll dabei schnell (ggf. mit Priorisierung) und kollisionsfrei stattfinden. Des Weiteren wird ein großes Augenmerk auf eine unterbrechungsfreie Spannungsversorgung gelegt. Diese wird implementiert, um ein bequemes und sicheres Wechseln der Akkus zu gewährleisten, ohne dabei das gesamte Fahrzeugsystem herunterfahren zu müssen. Das Wechseln der Spannungsversorgung soll dabei robust, mit schnellen Umschaltzeiten in der Größenordnung von einigen hundert Nanosekunden und ohne nennenswerte Spannungseinbrüche/-spitzen erfolgen. Bei Spannungseinbrüchen sollen dabei 12 V nicht unterschritten werden und Spannungsspitzen dürfen nicht mehr als 18 V betragen. Bei Kurzschlüssen bzw. Überströmen soll schnell eingegriffen werden, um weitere Schäden des Systems zu verhindern.

Um diese Funktionalitäten implementieren zu können, werden alle Platinen für diese Anforderungen neu designt und die dazugehörigen Firmwares von Grund auf neu entwickelt. Die Firmwares benötigen für ihre Funktionsweise diverse einstellbare Parameter. Diese sollen zur Laufzeit bzw. mindestens bis zum Start der Firmware, aber auf jeden Fall nach dem Kompilieren des Codes änderbar sein. Ferner sollen einheitliche Methoden und Werkzeuge zur Konfiguration und Fehlersuche zur Verfügung gestellt werden sowie eine einheitliche Anzeige von Fehlerzuständen erfolgen.

1.4 Struktur der Arbeit

Diese Arbeit besteht aus fünf Kapiteln. Im Kapitel „Grundlagen“ werden bestimmte Sachverhalte erläutert, um dem Leser den Einstieg in die Arbeit zu erleichtern. Im Kapitel 3 „Konzept“ wird die Strategie zum Aufbau der Platinen bzw. der dazugehörigen Firmwares präsentiert. Kapitel 4 zeigt die Implementierung des Konzeptes und im fünften Kapitel werden die gesetzten Ziele validiert. Das letzte Kapitel fasst die Arbeit zusammen und gibt einen Ausblick.

Da diese Arbeit von zwei Autoren angefertigt wird, werden die Texte des jeweiligen Autors mit seinen Initialen markiert. So werden Texte von Tim Wiesner mit **[TW]** gekennzeichnet, während Texte von Maximilian Grau mit **[MG]** markiert werden. Ist keine Markierung vorhanden, so wurde der Kapitel von beiden Autoren gleichermaßen angefertigt.

2 Grundlagen

Dieses Kapitel erläutert die Grundlagen, um dem Leser einen leichteren Einstieg in die Thematik dieser Arbeit zu vermitteln. Zuerst werden grundlegende Sachverhalte aus dem Hardwarebereich wie z. B. das Power-Multiplexing und der CAN-Bus erläutert. Anschließend werden firmwarespezifische Einzelheiten wie UAVCAN und Echtzeitbetriebssystem erklärt. Die hier behandelten Themen stehen dabei in derselben Reihenfolge, in der sie anschließend im Konzeptteil aufgegriffen werden.

2.1 Power-Multiplexing [MG]

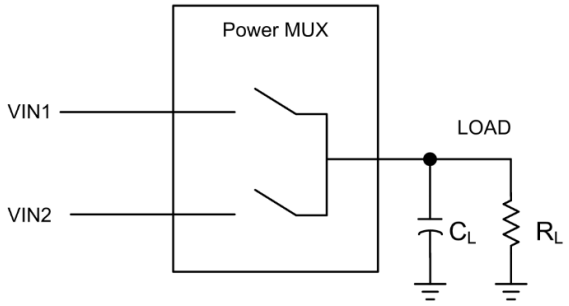


Abbildung 2.1: Power-Multiplexer [Tri18]

Ein Power-Multiplexer wie in der Abb. 2.1 ermöglicht einen unterbrechungsfreien und schnellen Wechsel zwischen zwei oder mehreren Spannungsquellen. So kann ein laufendes System bei sowohl beabsichtigtem (z. B. Akku-Wechsel) als auch unbeabsichtigtem Ausfall einer der Spannungsquellen weiterhin seine Arbeit verrichten. Da Power-Multiplexer in der Lage sind, die Spannungsleitungen zu verwalten, besitzen hochwertige *Integrated Circuit (IC)* integrierte Schutzfunktionen gegen Unter-/Überspannungen und Überströme [Tri18]. Diese versprechen einen Sicherheitsvorteil. Es ist wichtig, dass solch eine Schaltung möglichst energieeffizient arbeitet, da die Energie aus den Akkus bezogen wird.

Power-Multiplexing wird in der Regel mit zwei grundlegenden Schaltungen in Kombination realisiert: *OR-ing* und *HotSwap*.

2.1.1 OR-ing

Um eine Schaltung mit zwei verschiedenen Spannungsquellen betreiben zu können, müssen einige Dinge beachtet werden. Sobald sich zwei Spannungsquellen gegenüber stehen und unterschiedliche Spannungen besitzen, so fließt ein Ausgleichsstrom vom höheren Spannungslevel zum niedrigeren Spannungslevel. Beispielsweise besitzt das Netzteil als zweite Spannungsquelle eine höhere Spannung als der angeschlossene Akku, was zur Folge hat, dass das Netzteil den Akku **unkontrolliert** auflädt. Dies kann große Schäden verursachen und gilt es definitiv zu vermeiden. Um diesen *reverse current* (Rückstrom) zu verhindern, existieren nach [Sel19] mehrere Ansätze:

Dioden

Zum einen können Dioden eingesetzt werden. Diese lassen nur eine Stromrichtung zu und können leicht in Schaltungen (siehe Abb. 2.2) integriert werden. Damit scheinen sie für den Anwendungsfall geeignet zu sein, allerdings

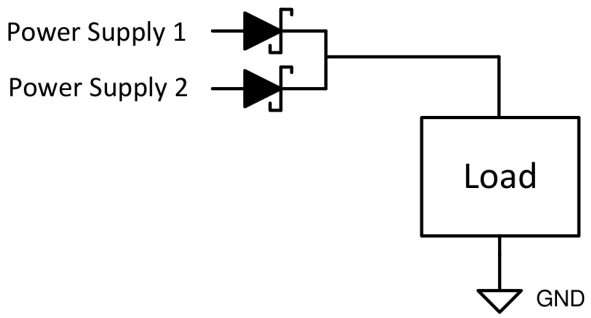
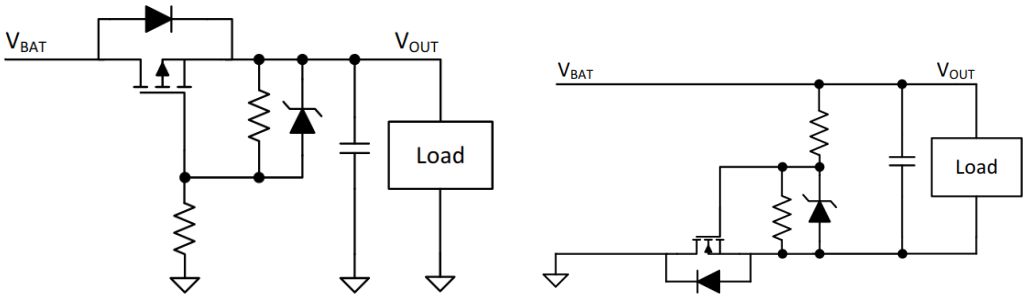


Abbildung 2.2: OR-ing-Schaltung mithilfe von Dioden [Sel19]

findet über diese ein Spannungsabfall statt. Dieser liegt bei Standarddioden auf Siliziumbasis zwischen 0,6 V und 0,7 V [Eleb], während sich der Wert des Spannungsabfalls bei Schottky-Dioden zwischen 0,3 V und 0,5 V [Elea] bewegt. Gemäß der Formel $P = U \cdot I$ entstehen an der Diode bei hohen Durchgangsströmen dementsprechend auch hohe Verlustleistungen. Damit sind sie nur für Anwendungsfälle mit kleinen Strömen bedingt geeignet.

MosFETs

Einen anderen Ansatz bildet das Verwenden von diskreten MosFETs. Deren Vorteil ist der äußerst geringe Durchlasswiderstand $R_{DS(on)}$ in eingeschaltetem/leitendem Zustand. Der Wert von $R_{DS(on)}$ bewegt sich im ein- bis zweistelligen mΩ-Bereich. Aus diesem Grund fällt über diesen kaum Spannung ab und somit wird auch nur wenig Verlustleistung erzeugt. Die Abb. 2.3 zeigt eine mögliche Verschaltung sowohl mit N-Channel-MosFETs als auch mit P-Channel-MosFETs. Allerdings lassen MosFETs in eingeschaltetem Zustand



(a) P-Channel-MosFET (b) N-Channel-MosFET

Abbildung 2.3: Aufbau einer „Reverse Polarity Protection"-Schaltung mithilfe von MosFETs, um die allgemeine Funktionsweise der MosFETs zu verdeutlichen [Sel19]

einen Stromfluss in beide Richtungen[1] durch. Aus diesem Grund erfordern sie eine Logik, welche solche Rückströme erkennen kann und so den MosFET sperrt.

OR-ing-Controller

Ein OR-ing-Controller besitzt diese Logik zur Erkennung von Rückströmen und steuert dementsprechend einen MosFET an. Diese Schaltung vereinigt damit die Vorteile der Dioden und diskreten MosFETs (ohne Logik). Mit der Schaltung wird hier also eine ideale Diode nachgebildet, welche kaum Verlustleistung abwirft.

[1] Hier ist nicht die parasitäre Body-Diode eines MosFETs gemeint, denn der Silizium-Aufbau des MosFETs lässt im Gegensatz zu Bipolartransistoren generell Ströme in beide Richtungen zu.

2.1.2 HotSwap

Wenn Platinen mit größeren Pufferkondensatoren an eine Spannungsquelle angeschlossen werden, entstehen temporär sehr hohe Einschaltströme. Der Grund hierfür liegt in der Eigenschaft eines entladenen Kondensators. Im entladenen Zustand ist für die Spannungsversorgung nur der *Equivalent Series Resistor (ESR)* des Kondensators relevant. Da dieser mit Werten zwischen 10 mΩ und 100 mΩ [Elec] recht klein ist, fließt am Anfang ein hoher Ladestrom von der Spannungsquelle in den Kondensator. Dieser Strom kann für elektrische Funken an den Verbindungssteckern, Durchbrennen/Auslösen der Sicherungen und je nach Auslegung der Spannungsquelle sogar für Spannungseinbrüche an dieser sorgen. Solche Einbrüche können dazu führen, dass andere Subsysteme nicht mehr genug Spannung bekommen und so ihre Arbeit einstellen.

Um dieses Szenario zu vermeiden, muss der Einschaltstrom limitiert werden. Eine einfache Lösung ist ein Widerstand zwischen der Spannungsquelle und dem Kondensator. Dieser limitiert zwar den Einschaltstrom, aber im stationären Betrieb, d. h. der Kondensator ist geladen, wäre diese Limitierung weiterhin existent. Das führt bei Lasten mit höheren permanenten Strömen ($\neq$ temporärer Einschaltstrom) zu Verlustleistungen und Spannungsabfällen an dem Widerstand. Diese Tatsache sorgt für eine Senkung der Energieeffizienz.

Eine bessere Lösung bietet hier der *HotSwap-Controller* an. Dieser verwendet wie in der Abb. 2.4 ersichtlich einen (externen) MosFET und begrenzt mit dessen Linearbetrieb den Einschaltstrom. Mit dieser Betriebsart verhält sich der MosFET wie ein veränderlicher Widerstand, d. h. die Höhe der Gate-Spannung hat einen Einfluss auf die Größe des Durchlasswiderstandes R_{DS}. Im stationären Betrieb ist der MosFET voll leitend – so hat dieser den geringsten Durchlasswiderstand $R_{DS(on)}$ und deswegen fallen kaum Verlustleistungen an diesem ab, was der Energieeffizienz zugute kommt.

HotSwap-Controller haben nach [Max03] zwei Strategien, um ihre Aufgabe umzusetzen. Zum einen kann das Gate des MosFETs mit einer definierten Geschwindigkeit $\frac{dV}{dt}$ aufgeladen werden. So durchläuft der MosFET seinen Linearbetrieb, welcher dafür sorgt, dass der MosFET im ohmschen Bereich ist und somit wie ein größerer Widerstand fungiert. Dieser begrenzt so den Einschaltstrom. Die Höhe und Dauer des Einschaltstromes hängen hier von der Größe der Gate-Ladung ab und wie schnell dieser aufgeladen wird. Nachdem

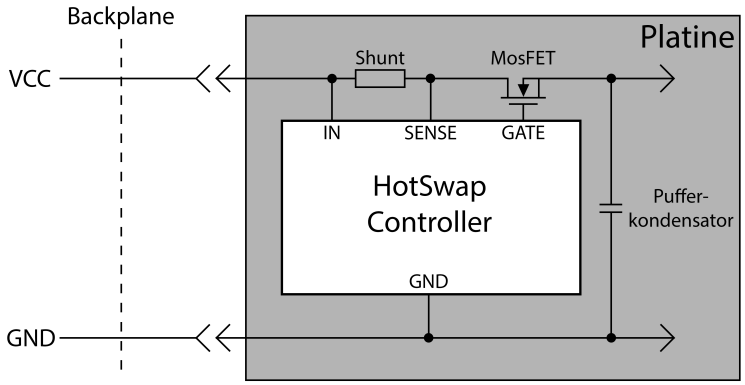


Abbildung 2.4: Beispielschaltung zur Begrenzung des Einschaltstromes mithilfe eines HotSwap-Controllers nach [Max03]

das Gate voll aufgeladen ist, besitzt der MosFET nur einen sehr kleinen Durchlasswiderstand und lässt größere Ströme ohne große Verlustleistung durch.

Zum anderen kann der HotSwap-Controller den Einschaltstrom über einen Messwiderstand *(Shunt)* kontinuierlich messen und dementsprechend den MosFET flexibel ansteuern. Letztere Strategie bietet den Vorteil, dass der Controller unabhängig von der Größe der Gate-Ladung agieren kann und bequemere Einstellmöglichkeiten wie Höhe oder Dauer des Einschaltstromes anbietet. Außerdem besitzen die HotSwap-Controller mit dieser Architektur zusätzlich die Möglichkeit, den Strom im stationären Betrieb zu messen und bei Überstrom als Unterbrecher (E-Fuse) zu fungieren.

2.2 Controller Area Network (CAN)

Der Inhalt dieses Kapitels basiert auf Informationen von [MEM] und [Bie08]. Um die verschiedenen Herausforderungen durch die Vielzahl an elektronischen Steuergeräten in einem automobilen Fahrzeug zu meistern, hat Bosch 1983 für die Kommunikation zwischen den Steuergeräten den CAN-Bus entwickelt. Es entstanden zwei ISO-Standards, *ISO 11898-2*[1] für *high-speed* und *ISO 11898-3*[2] für *low-speed*. Die CAN-Kommunikation findet in Schichten statt.

[1] `https://www.iso.org/standard/67244.html`
[2] `https://www.iso.org/standard/63648.html`

Die unterste Schicht bildet die *Bitübertragungsschicht (Physical Layer)*, welche die grundlegenden physikalischen Aspekte wie die Signalerzeugung, Pegeldifferenzen und Verdrahtung beschreibt. Die Aufgabe dieser Schicht übernehmen CAN-Controller und CAN-Transceiver. Die nächste Schicht bildet die *Sicherungsschicht (Data Link Layer)*, hier werden die Nachrichtenformate des CAN-Protokolls definiert. Die höheren Schichten werden per Software wie z. B. CANopen oder UAVCAN auf dem Mikrocontroller implementiert.

2.2.1 Bitübertragungsschicht

Generell lag das Augenmerk auf der Reduzierung des Verkabelungsaufwands. Deshalb benötigt der CAN-Bus für den Betrieb lediglich ein differentielles Datenpaar (CAN-High und CAN-Low) und eine Masseleitung. An ihm können mehrere gleichberechtigte Busknoten angeschlossen werden und bilden so ein multi-master-fähiges Bussystem. Die Datenübertragung geschieht wie eingangs erwähnt differentiell, d. h. eine Bitinformation wird auf beiden Leitungen gleichzeitig, aber mit einem invertierten Spannungssignal dargestellt. Damit ist der Bus robust gegenüber elektrischen Störungen, da diese auf beide Leitungen gleichermaßen wirken – die Differenz zwischen den Leitungen bleibt somit immer gleich. Dies ist in der Elektrotechnik auch als *Gleichtaktunterdrückung* bekannt.

Die Erzeugung der Datensignale erfolgt durch Transistoren und Widerstände. Über Pegeldifferenzen lassen sich verschiedene Bitzustände darstellen. Eine Pegeldifferenz zwischen den beiden Leitungen entspricht einer logischen **0**, während keine Pegeldifferenz eine logische **1** bedeutet. Da ein Busteilnehmer jederzeit eine vorhandene logische **1** auf dem Bus durch eine logische **0** überschreiben kann, entspricht die logische **0** einem dominanten Zustand.

Mit den dominanten und rezessiven Bits ist nun eine *bitweise Arbitrierung* möglich. Diese ist erforderlich, um eine kollisionsfreie Nachrichtenübertragung zu gewährleisten. Denn jeder Busteilnehmer kann jederzeit eine Nachricht auf dem Bus senden. Natürlich kommt es dann vor, dass mehrere Busteilnehmer mit dem Nachrichtenversand gleichzeitig starten wollen. Um eine Kollision zu verhindern, wird jede Nachricht mit einem eindeutigen *Identifier* gekennzeichnet, welcher gleichzeitig die Priorisierung bildet. Da die logischen Nullen dominant sind, werden Nachrichten mit dem niedrigsten Identifier, d. h. mit der höchsten Priorität, zuerst gesendet. Bei den anderen Nachrichten mit einem

höheren Identifier (niedrigerer Priorität) wird später erneut versucht, sie zu verschicken.

2.2.2 Aufbau der Nachrichten

Bei CAN können fünf verschiedene Arten von Nachrichten (Frames) versendet werden. Am häufigsten sind sogenannte *Data Frames*, die Nutzdaten zusammen mit einem Identifier übertragen, der Rückschlüsse auf die Art der Daten zulässt. Data Frames gibt es in der Standard- und Extended-Variante, die sich lediglich durch die Länge des Identifiers unterscheiden (11 Bit bei Standard Data Frames, 29 Bit bei Extended Data Frames). *Remote Frames* werden benutzt, um Data Frames mit einem bestimmten Identifier anzufordern. Stellt ein beliebiger Busteilnehmer einen Fehler bei einer Datenübertragung fest, sendet dieser einen *Error Frame*, um alle anderen Busteilnehmer darauf aufmerksam zu machen. Die fünfte Art ist der *Overload Frame*, mit welchem den übrigen Busteilnehmern signalisiert wird, dass die Verarbeitung einer empfangenen Nachricht noch nicht abgeschlossen ist und deshalb aktuell keine Empfangsbereitschaft für neue besteht. Das Senden eines Overload Frames verzögert neue Nachrichten um die Dauer eines Frames.

Bei der Übertragung kommt als Verfahren Non Return to Zero (NRZ) zum Einsatz. Das bedeutet, das für jedes Bit einer von zwei möglichen Buspegeln erzeugt wird, je nach Wert des Bits. Dies hat zur Folge, dass sich bei vielen aufeinanderfolgenden Bits des gleichen Werts der Buspegel über eine längere Zeit nicht ändern würde und so eine Synchronisation des Takts nicht mehr möglich wäre. Um dieses Problem zu lösen, wird nach fünf aufeinanderfolgenden, gleichwertigen Bits vom Sender ein Bit mit invertiertem Pegel eingefügt, das für einen Flankenwechsel sorgt und so die Synchronisation wieder möglich macht. Dieses Verfahren wird *Bit-Stuffing* genannt. Das eingefügte Stuff-Bit wird anschließend von den Empfängern automatisch wieder entfernt.

Feld	Länge in bit	Bedeutung
SOF	1	Frame-Start-Marker
Identifier	11	Arbitrierung
RTR	1	Remote Transmission Request
IDE	1	Extended-Identifier-Bit
DLC	4	Länge der Nutzdaten in Bytes
Data	0 bis 64	Nutzdaten
CRC	15	Prüfsumme
DEL	1	CRC-Delimiter
ACK	1	Bestätigungsfeld
DEL	1	ACK-Delimiter
EOF	7	End-of-Frame-Marker
IFS	3	Rezessive Bits zwischen einzelnen Frames

Abbildung 2.5: Aufbau eines Standard-Datenframes ohne Bit-Stuffing mit einer Nutzdatenlänge von einem Byte[1]

2.2.3 Fehlererkennung

Damit der CAN-Bus möglichst robust agiert, muss eine Vielzahl von Fehlerarten detektiert werden können. Nach [Bie08] sind fünf verschiedene Fehlerarten zu unterscheiden, die im Folgenden erläutert werden.

Der Sender prüft bei jedem gesendeten Bit den aktuellen Buspegel. Wird dieser von einem anderen Teilnehmer überschrieben, liegt ein *Bitfehler* vor. Dies gilt nicht für das Arbitrierungsfeld (Identifier) und das Bestätigungsfeld (ACK), da das Überschreiben der Bits dieser Felder spezifiziert ist und i. d. R. keinen Fehler darstellt. Bei einem *Bit-Stuffing-Fehler* wird eine Sequenz von mehr als fünf aufeinanderfolgenden gleichwertigen Bits erkannt, die eigentlich mit einem komplementären Bit hätte aufgefüllt werden müssen. Der End-of-Frame-Marker (EOF) stellt hierbei eine Ausnahme dar.

Für jeden CAN-Frame wird eine Prüfsumme (CRC) berechnet, die in den Frame eingebettet wird. Die Empfänger berechnen die Prüfsumme erneut und

[1] Nach `https://de.wikipedia.org/wiki/Controller_Area_Network#Frame-Aufbau`

vergleichen diese mit der im Frame enthaltenen. Weichen die beiden Werte voneinander ab, ist ein *CRC-Fehler* aufgetreten. Zu *Formatfehlern* kommt es, wenn ein empfangener Frame einen falschen Aufbau besitzt und bspw. bei Bits wie dem CRC-Delimiter (DEL), das immer den Wert logisch **1** besitzt, von der Spezifikation abweicht. Die letzte Fehlerart stellt der *Acknowledgement-Fehler* dar. Dieser tritt auf, wenn ein Sender im Bestätigungsfeld (ACK) kein dominantes Bit feststellt, welches im Falle einer erfolgreichen Transaktion von mindestens einem Busteilnehmer als Bestätigung über den Empfang einer Nachricht erzeugt wird.

2.3 UAVCAN [TW]

UAVCAN[1] ist ein Anwendungsprotokoll auf Basis von CAN für eine verlässliche Kommunikation zur Nutzung in Luft- und Raumfahrt- sowie Robotikanwendungen [KWa].

2.3.1 Grundkonzept

Innerhalb eines UAVCAN-Netzwerkes sind alle Teilnehmer gleichberechtigt; es gibt keinen Master-Knoten, der die Kommunikation koordiniert. Dadurch gibt es auch keinen *Single Point of Failure*, dessen Ausfall zu einem totalen Zusammenbruch der gesamten Kommunikation führen würde. Es können Nutzdaten ausgetauscht werden, die die Framegröße des zugrunde liegenden Transportprotokolls überschreiten. Dazu werden diese vor dem Senden in kleinere Frames aufgeteilt und bei Empfang wieder zu einem einzigen Nutzdatenblock zusammengesetzt. Diese Prozesse geschehen automatisch auf Protokollebene, so dass das Aufteilen und wieder Zusammensetzen nicht von der jeweiligen Anwendung selbst implementiert werden muss. Zudem gibt es bereits vordefinierte Nachrichten und Dienste für einige häufig benötigte Funktionen. Das umfasst unter anderem das Erkennen anderer Netzwerkteilnehmer, die Konfiguration von Netzwerkknoten, das Überwachen des Status aller Netzwerkteilnehmer, die

[1] Während der Anfertigung dieser Arbeit ist die Spezifikation für UAVCAN in der Version *v1.0* erschienen. Da für diese aber zu diesem Zeitpunkt noch keine Referenzimplementierung existierte und bereits mit der Integration der Version *v0* begonnen wurde, bezieht sich die gesamte Arbeit auf UAVCAN *v0*.

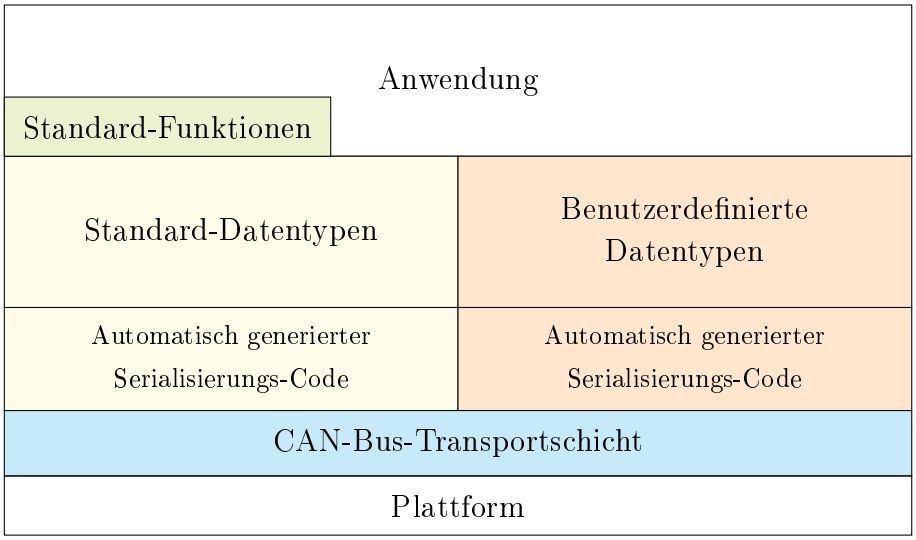


Abbildung 2.6: Schematischer Aufbau von UAVCAN in Anlehnung an [KWa]

netzwerkweite Zeitsynchronisation und das Aktualisieren der Firmware anderer Teilnehmer. UAVCAN kann auf einer Vielzahl von Systemen mit unterschiedlichen Ressourcenbeschränkungen eingesetzt werden, was sowohl linuxbasierte, rechenintensive Systeme als auch MCUs mit stark eingeschränkter Speicherkapazität und Rechenleistung einschließt [KWb].

Abb. 2.6 skizziert grob den schematischen Aufbau von UAVCAN. Die jeweilige Anwendung kann auf Standardfunktionen zurückgreifen, die wiederum die Standard-Datentypen benutzen. Sie kann aber auch eigene Datentypen definieren, um spezifischere Funktionen bereitzustellen. Für beide Arten von Datentypen werden Serialisierungs- und Deserialisierungscode generiert. Die serialisierten Nutzdaten werden dann von der CAN-Bus-Transportschicht in CAN-Frames aufgeteilt und anschließend über den Bus versendet.

Jeder Netzwerkteilnehmer wird eindeutig über eine Identifikationsnummer, die *Node ID* identifiziert. Die Kommunikation erfolgt immer über eine von zwei möglichen Methoden, dem *Message Broadcasting* oder der *Service Invocation*. Beim Message Broadcasting wird eine Nachricht (*Message*) an alle Teilnehmer des Netzwerkes gesendet. Dieses Vorgehen stellt das primäre Mittel zum Datenaustausch in einem UAVCAN-Netzwerk dar. Dabei obliegt es jedem Teilnehmer selbst, die Nachricht entweder zu verarbeiten oder zu ignorieren (*Publish/Subscribe*-Ansatz). Bei einer Service Invocation hingegen erfolgt die Kommunikation in zwei Schritten. Im ersten wird eine Anfrage an einen

spezifischen Netzwerkteilnehmer gestellt (*Request*), der diese verarbeitet und im zweiten Schritt eine Antwort (*Response*) an den anfragenden Teilnehmer sendet [KWa].

2.3.2 Data Structure Description Language (DSDL)

Für den Datenaustausch werden immer vordefinierte Strukturen verwendet, wobei jede Datenstruktur über eine eindeutige Data Type ID (DTID) verfügt. Diese Strukturen werden mit Hilfe der DSDL, einer Beschreibungssprache, definiert. Aus den Definitionen wird dann Code für die Serialisierung und Deserialisierung für die jeweilige Zielprogrammiersprache generiert. Da die so generierten Strukturen statisch sind, d. h. dem System schon zur Kompilierzeit vorliegen, kann die Protokollimplementierung in Hinsicht auf Speicherverbrauch und Rechenzeit optimiert werden kann [KWa]. Es gibt zwei Arten von DSDL-Definitionen, eine für Messages und eine für Services, die sich nur geringfügig voneinander unterscheiden [KWK].

2.4 Echtzeitbetriebssystem [TW]

Dieser Abschnitt soll einen kurzen Überblick über die Notwendigkeit und die Aufgaben von Echtzeitbetriebssystemen geben. Alle darin getätigten Aussagen beziehen sich, sofern nicht anders gekennzeichnet, auf Informationen aus [Gay18] und [Hün19].

Viele MCUs besitzen nur eine Central Processing Unit (CPU), weshalb das gesamte Programm sequenziell in einem einzigen Task unter Verwendung eines einzigen Stacks für alle lokalen Variablen und Rücksprungadressen ausgeführt werden muss. Dabei läuft das Programm in einer Schleife (auch bekannt als „Super Loop“), in deren Iteration die jeweiligen Aufgaben des Controllers ausgeführt werden [Wan17]. Dieses Vorgehen kann unter bestimmten Umständen für simplere Aufgaben, zum Beispiel das regelmäßige Auslesen eines Sensors und der anschließenden Übertragung des ermittelten Wertes, ausreichend sein. Bei komplexeren Szenarien hingegen stößt dieses Vorgehen schnell an seine Grenzen. Dazu gehören beispielsweise eine größere Anzahl von Tasks mit unterschiedlichen Ausführungsintervallen, asynchron auftretende Tasks sowie die Abhängigkeit von mehreren Tasks untereinander.

Hierfür empfiehlt sich der Einsatz eines Real-time Operating System (RTOS). An Stelle der Tasks selbst verwaltet ein RTOS die Hardwareressourcen des Mikrocontrollers, wie etwa die CPU und den Speicher. Da eine echte parallele Ausführung von mehreren Tasks mit nur einer CPU nicht möglich ist, verteilt das RTOS die zur Verfügung stehende Rechenzeit auf die einzelnen Tasks, sodass diese „quasi-parallel“ ausgeführt werden. Dabei können zum Beispiel, je nach konkreter Implementierung, den Tasks auch unterschiedliche Prioritäten zugewiesen werden. Im Wesentlichen werden zwei unterschiedliche Strategien beim Zuteilen von Rechenzeit, dem sogenannten Scheduling, verwendet. Eine Strategie ist das *präemptive Multitasking*, bei dem den Tasks Rechenzeit in Form von kleinen Zeitscheiben zur Verfügung gestellt wird. Ein Task wird solange ausgeführt, bis er das Ende der Zeitscheibe erreicht hat, durch ein Ereignis blockiert wird oder freiwillig die Kontrolle abgibt. Der Scheduler des RTOS bestimmt bei dieser Strategie, welche Tasks wann und in welcher Reihenfolge ausgeführt werden. Beim *kooperativen Multitasking* dagegen kann nur der Task selbst die Kontrolle abgeben, woraufhin das RTOS entscheidet, welcher Task als nächstes Rechenzeit erhält.

Zu den weiteren Aufgaben eines RTOS gehört die Bereitstellung von Kommunikationsprimitiven, um eine Kommunikation der Tasks untereinander zu ermöglichen. Dabei können zum Beispiel *Message Queues* nach dem FIFO-Prinzip zum Einsatz kommen, bei dem ein Task bzw. Interrupt Service Routine (ISR) eine Message in die Queue legt und ein anderer Task oder ISR die Message wieder ausliest. Ein weiterer, vor allem bei der Inter-Task-Kommunikation wichtiger Aufgabenbereich ist die Synchronisierung von Tasks. Durch die Nebenläufigkeit dieser kann es beim Zugriff von mehreren Tasks auf geteilte Ressourcen zu fehlerhaftem Verhalten kommen. So könnten nicht-atomare Rechenoperationen, also aus mehreren Prozessorinstruktionen bestehende, ein falsches Ergebnis liefern, wenn diese zwischen den Einzelinstruktionen etwa durch einen Task-Wechsel unterbrochen werden. Um derartige Fehler zu vermeiden, stellen RTOS Synchronisierungsmechanismen in der Form von *Mutexen* oder *Semaphoren* zur Verfügung. Diese stellen sicher, dass der Zugriff auf eine geteilte Ressource nur von einem bzw. im allgemeinen Fall von einer begrenzten Anzahl von Konsumenten genutzt werden kann.

3 Konzept

Für das Bussystem fällt die Wahl auf CAN, da dieser Bus multi-master-fähig ist und weite Verbreitung in der Automobilindustrie findet. Dazu stellt die Hardware dementsprechend einen CAN-Controller, einen CAN-Transceiver und mehrere Busanschlüsse bereit. Die Firmwares werden neben der Bereitstellung der Grundfunktionalitäten wie CAN-Kommunikation auf die individuellen Anforderungen der Hardware zugeschnitten.

3.1 Hardware [MG]

Da die Implementierung eines neuen Multi-Master-Busses einen anderen MCU mit entsprechendem Bus-Controller und Bus-Treiber erfordert, besteht die Notwendigkeit, die Platinen neu zu designen. Hierbei entsteht die Gelegenheit, Verbesserungen und neue Features auf den neuen Platinen zu implementieren. Dazu flossen auch Wünsche der anderen oTToCar-Teammitglieder ein. Dabei sind Wünsche nach mehr Sicherheit, einem höheren Komfort und verbesserten Möglichkeiten zum Debuggen der Firmwares zusammengekommen. So wird, um die Sicherheit grundlegend zu erhöhen, eine native Implementierung mehrerer Sicherungen umgesetzt. Zudem werden mehrere unterschiedliche Spannungs- und Stromparameter überwacht und bei einem möglichen Fehlerfall wie Unter-/Überspannung bzw. Überstrom soll eingegriffen werden. Um den Komfort eines Akkuwechsels anzubieten, bei dem nicht das gesamte System heruntergefahren werden muss, wird eine Power-Multiplexing-Schaltung implementiert. Des Weiteren werden Features wie das Ausführen des *Serial Wire Output* (SWO) für einfache `print`-Ausgaben sowie eine zweifarbige Status-LED auf allen Platinen verbaut. Die LED dient der verbesserten Wahrnehmung des jeweiligen Status der Platinen. Der Aufbau des neuen Autos bildet im Prinzip gleichzeitig die Struktur des Busses ab. Diese bestimmt im Wesentlichen bereits die Anordnung der Platinen nach bestimmten Kriterien

wie etwa dem Einsatzort der Platinen. Die *Sensor Boards* werden z. B. jeweils an Front und Heck des Autos verbaut und bilden so die beiden Enden im Bussystem.

3.1.1 Spannungsversorgung

Um zu verhindern, dass das NUC samt Linux bei einem Wechsel des Akkus heruntergefahren werden muss, wird das Konzept des *Power-Multiplexings*, einem Wechsel zwischen zwei Spannungsquellen, eingesetzt. Dazu wird eine zweite Spannungsquelle in Form eines weiteren Akkus bzw. Netzteils für den Zeitraum des Akkuwechsels temporär angeschlossen und übernimmt so die Spannungsversorgung des Systems, ohne dass dieses heruntergefahren werden muss. So kann bequem der leere Akku entfernt und durch einen neuen ersetzt werden. Dabei wird auch die Zeit gespart, die früher für das Herunterfahren und Neustarten des Linux-Systems erforderlich war. Die Aufgabe einer unterbrechungsfreien Spannungsversorgung des Autos übernimmt das *Power Distribution Board*. Die Platine ist u. a. für Power-Multiplexing konzipiert. Für die Versorgung der Busteilnehmer, der LED-Beleuchtung und des Touchscreens ist ein 5V-Spannungswandler zuständig, während ein 12V-Spannungswandler die Spannungsversorgung der Servomotoren realisiert. Im Vergleich zum alten Auto wird hier aus Komplexitätsgründen auf eine getrennte Versorgung des Motors bzw. der Recheneinheit vom Rest des Autos verzichtet, d. h. alle Komponenten des Systems beziehen die Energie aus derselben Spannungsquelle. Die Spannungswandler werden zugunsten der Effizienz als Abwärtswandler (auch bekannt als Stepdown-Wandler bzw. Buck-Converter) arbeiten. Da die vier VESCs (Treiberplatinen für die Einzelradmotoren) recht große Kapazitäten besitzen, wird zur Vermeidung hoher Einschaltströme und Bildung von Funken am Akkustecker ein *HotSwap-Controller* verbaut. Da die VESCs rekuperieren und so Strom zurück in den Akku speisen können, wird der HotSwap-Controller nicht an den Power-Multiplexer, sondern direkt an den Akku angeschlossen. Der Power-Multiplexer lässt aufgrund seines Aufbaus (genauer gesagt durch die OR-ing-Schaltung) keine Rückströme zu. Da die zentrale Recheneinheit NUC eine beliebige Eingangsspannung zwischen 12 V und 19 V annimmt [Int20], kann diese ohne zusätzlichen Spannungswandler direkt an den Power-Multiplexer angeschlossen werden. Auf den 5V-, 12V- und NUC-Leitungen werden zur Überwachung und für die Statistik die jeweiligen Spannungen und Ströme gemessen. In einem Fehlerfall wie Unter-/Überspannung

bzw. Überstrom kann eingegriffen werden. Die Abbildung 3.1 zeigt einen Überblick über die einzelne Bestandteile der Spannungsversorgung.

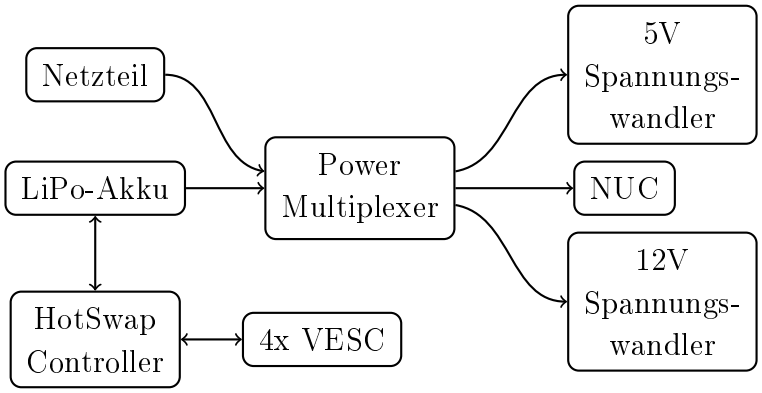


Abbildung 3.1: Schematischer Aufbau der Spannungsversorgung – die Pfeile stellen den möglichen Stromfluss dar

3.1.2 Struktur des Busses

Die Kommunikation zwischen den Platinen findet über *CAN* statt. Die zentrale Recheneinheit, hauptsächlich zuständig für die zentrale Logik des autonomen Fahrens, stellt das *NUC* von Intel dar. Dieses kommuniziert über USB und die *CAN-USB-Bridge* mit den am CAN-Bus angeschlossenen Platinen.

Das Design der *CAN-USB-Bridge* basiert auf einem bereits vorhandenem Projekt[1] auf GitHub und wird für die Verwendung der von uns benutzten Komponenten wie Steckverbindungen und Elektronikkomponenten leicht angepasst. Das Herzstück aller Platinen stellt das *Power Distribution Board* dar, da es für die Spannungsversorgung des Autos zuständig ist. Des Weiteren sammeln zwei *Sensor Boards* (jeweils für die Front und das Heck des Autos) die eingehenden Daten der Distanzsensoren und steuern die LEDs z. B. für Bremslicht und Blinker an. Das *Interface Board* dient als Rückfallebene für den Touchscreen und beinhaltet die Hardware für den 868MHz-Funk, welcher der Kommunikation mit einer oder mehreren Fernbedienungen dient. Eine Fernbedienung wird gebraucht, um das Auto manuell steuern zu können, beispielsweise wenn dieses im autonomen Modus sich nicht korrekt verhält und dementsprechend

[1] `https://github.com/HubertD/candleLight`

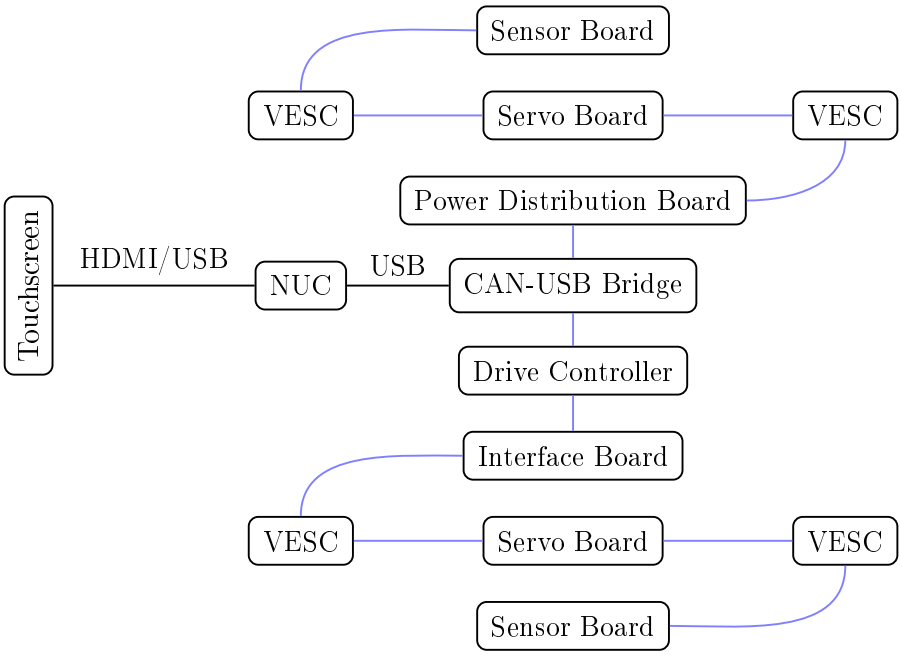


Abbildung 3.2: Möglicher Aufbau des Autos – die CAN-Leitungen werden durch blaue Verbindungslinien dargestellt

vom Team eingegriffen werden muss. Der Touchscreen wird direkt über HDMI am *NUC* angeschlossen und dient zur Interaktion mit dem Auto.

Die Platinen für die Ansteuerung des Servomotors (Servo Board), der Einzelradmotoren (VESC) und für die zentrale Steuerung aller vier Motoren (Drive Controller), sind nicht Bestandteil dieser Bachelorarbeit und werden im Rahmen einer anderen Arbeit [Alb20] behandelt.

Ein möglicher Aufbau des Autos wird in der Abb. 3.2 dargestellt. Es folgt für alle Platinen jeweils ein UML-Komponentendiagramm, welches zum einen den grundlegenden Aufbau der Platine, zum anderen die möglichen Interaktionen (durch Bereitstellung der Schnittstellen) visualisiert. Jede Platine benötigt zwar eine Spannungsversorgung, jedoch wurde dies, um Übersicht zu bewahren, nicht in die Diagramme aufgenommen. Tabelle 3.1 zeigt eine Liste der vorhandenen Interfaces – hier wird CAN nicht mit aufgelistet, da dieses Interface ein Grundbestandteil aller Platinen ist.

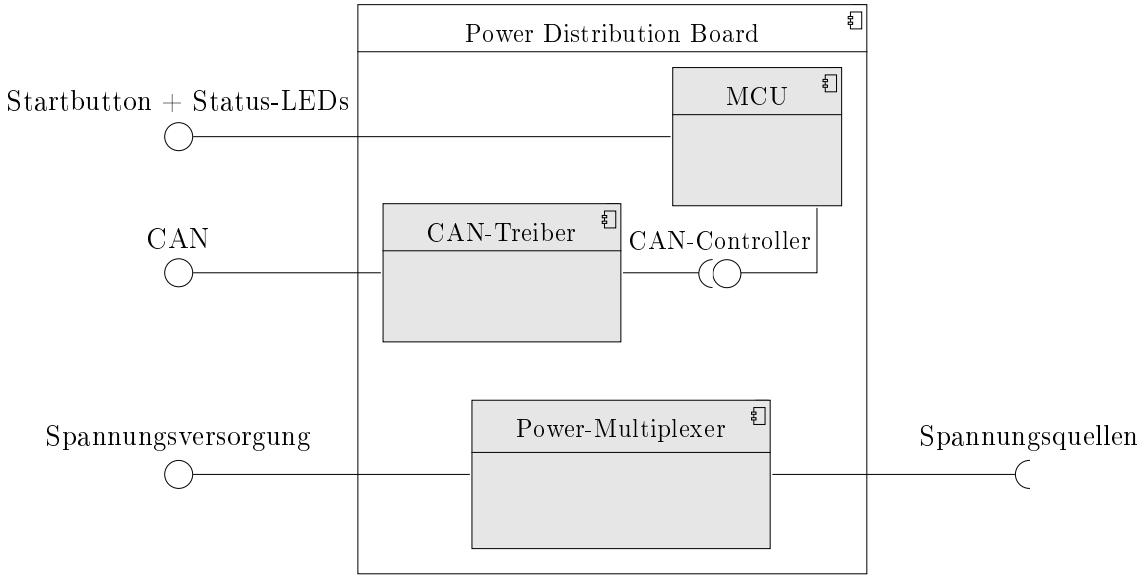


Abbildung 3.3: Angebotene Interfaces: Power Distribution Board

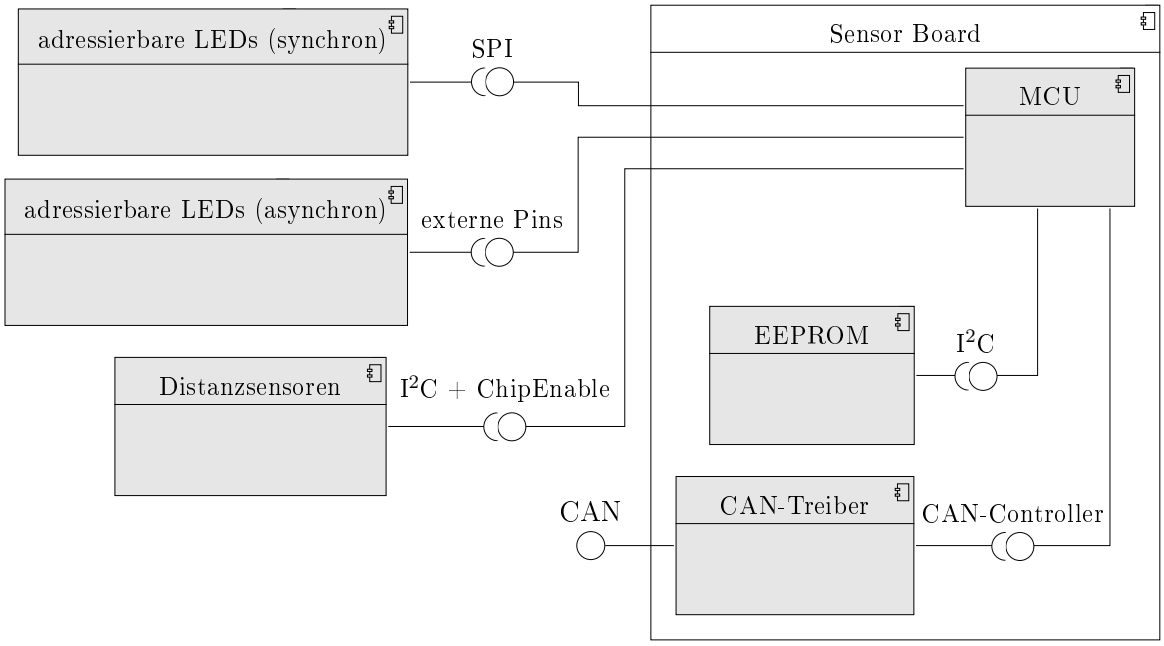


Abbildung 3.4: Angebotene Interfaces: Sensor Board

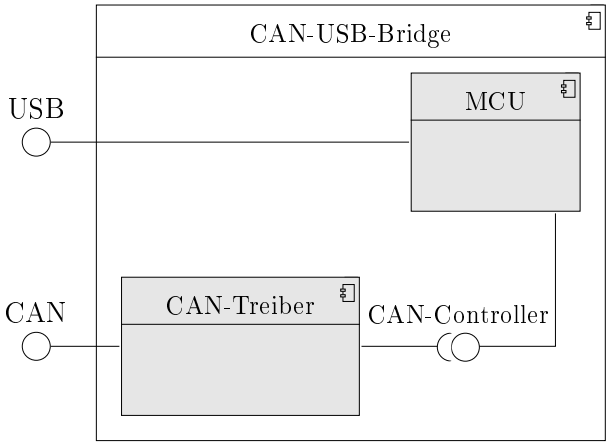


Abbildung 3.5: Angebotene Interfaces: CAN-USB-Bridge

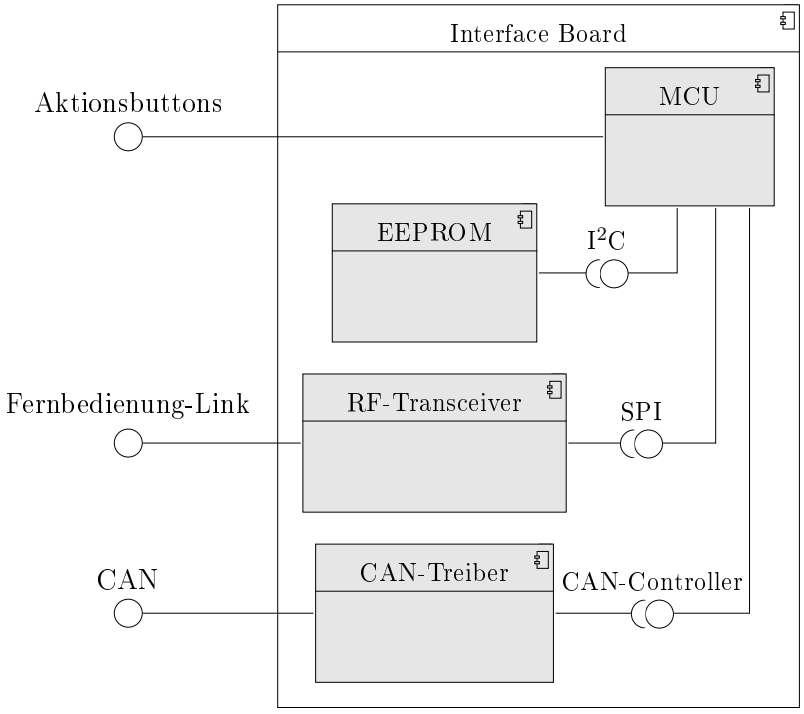


Abbildung 3.6: Angebotene Interfaces: Interface Board

Platine	Interface	Bedeutung
PDB	Spannungsversorgung	Stellt 5V- und 12V-Spannungen zur Verfügung
	Startbutton + Status-LEDs	Erforderlich zum Starten/Abschalten des Fahrzeugsystems. LEDs dienen als Indikator des Status der Spannungsquellen und Versorgungsschienen
	Spannungsquellen	Anschluss der Spannungsquellen wie Akku und externes Netzteil
SB	SPI	Ansteuern der adressierbaren LEDs (synchrone Datenübertragung)
	Externe Pins	Ansteuern der adressierbaren LEDs (asynchrone Datenübertragung)
	I^2C + ChipEnable	Kommunikation mit den Distanzsensoren
IB	Aktionsbuttons	Buttons als Rückfallebene für Touchscreen
	Fernbedienung-Link	Funkschnittstelle zur Fernbedienung

Tabelle 3.1: Übersicht aller Interfaces

3.2 Firmware [TW]

Alle Firmwares setzen bestimmte Basisfunktionalitäten um. Dazu gehört unter anderem die Umsetzung eines UAVCAN-Knotens, mit dem Nachrichten gesendet und empfangen werden können sowie Service-Aufrufe getätigt und entgegengenommen werden können. Ferner soll jeder Teilnehmer über die UAVCAN-Standardfunktionen regelmäßig seinen Status zur Erkennung durch die anderen Teilnehmer auf dem Bus veröffentlichen. Abb. 3.8 zeigt, über welche Nachrichten und Services zwischen den Netzwerkteilnehmern kommuniziert wird, wobei der Austausch ausschließlich über den CAN-Bus erfolgt. Es werden dabei nur die im Rahmen dieser Arbeit behandelten Teilnehmer, ergänzt um das Intel NUC, dargestellt; Netzwerkteilnehmer aus [Alb20] sind nicht enthalten.

Die Funktionsweise der Firmwares hängt wesentlich von einigen konfigurierbaren Parametern ab. Zur Vereinfachung des Konfigurationsprozesses sollen die Firmwares über den CAN-Bus mit Hilfe von dafür in UAVCAN vorgesehenen Mechanismen konfigurierbar sein. So müssen die jeweiligen Parameter nicht fest einprogrammiert sein, sondern können zur Laufzeit von anderen Busteilnehmern angepasst werden. Da auch auf dem Intel NUC ein Busknoten ausgeführt wird, kann die Konfiguration auch über eine grafische Oberfläche erfolgen, was den gesamten Prozess aus Anwendersicht vereinfacht. Die Ergebnisse des Konfigurationsprozesses werden in einen nicht-flüchtigen Speicher auf dem jeweiligen Board geschrieben und spätestens beim nächsten Firmware-Start übernommen.

Zur genauen Identifikation und Rückverfolgung von Firmware-Builds werden Buildinformationen mit im Binärabbild hinterlegt. Die Abfrage dieser erfolgt als Service-Aufruf an den jeweiligen Busteilnehmer. Dieses Vorgehen ermöglicht eine automatische Überprüfung auf die richtigen Programmversionen bzw. die Erkennung von Builds, die von der in der Versionskontrolle hinterlegten Version abweichen.

Jedes Board besitzt mindestens eine zweifarbige Status-LED. Damit das Erkennen und Behandeln von Fehlern einfacher wird, sollen Fehlerzustände der Firmware einheitlich angezeigt werden können. Im Normalbetrieb soll die Status-LED dauerhaft grün leuchten; bei Ausfällen von Teilsystemen, die keinen Einfluss auf die korrekte Funktionsweise des Gesamtsystems haben, soll diese (durch Mischen von rot und grün) gelb leuchten. Bei Fehlerzuständen, die den Ausfall von wichtiger Funktionalität zur Folge haben, soll die LED rot leuchten und bei schwerwiegenden Fehlern, wie einem Prozessor-Fault, rot blinken.

Im Folgenden wird detailliert auf die Funktionsweise der einzelnen Firmwares eingegangen.

3.2.1 Power Distribution Board (PDB)

Der Programmablauf wird in Abb. 3.7 ersichtlich. Sobald das PDB mit Strom versorgt wird, leuchten zunächst alle Status-LEDs kurz auf, um deren Funktionalität zu testen, da diese später kritische Zustände anzeigen sollen. Sämtliche *Power Rails* sind zu diesem Zeitpunkt noch deaktiviert. Ist der Funktionstest abgeschlossen, wechselt das Board in den *Standby*-Zustand, um auf das

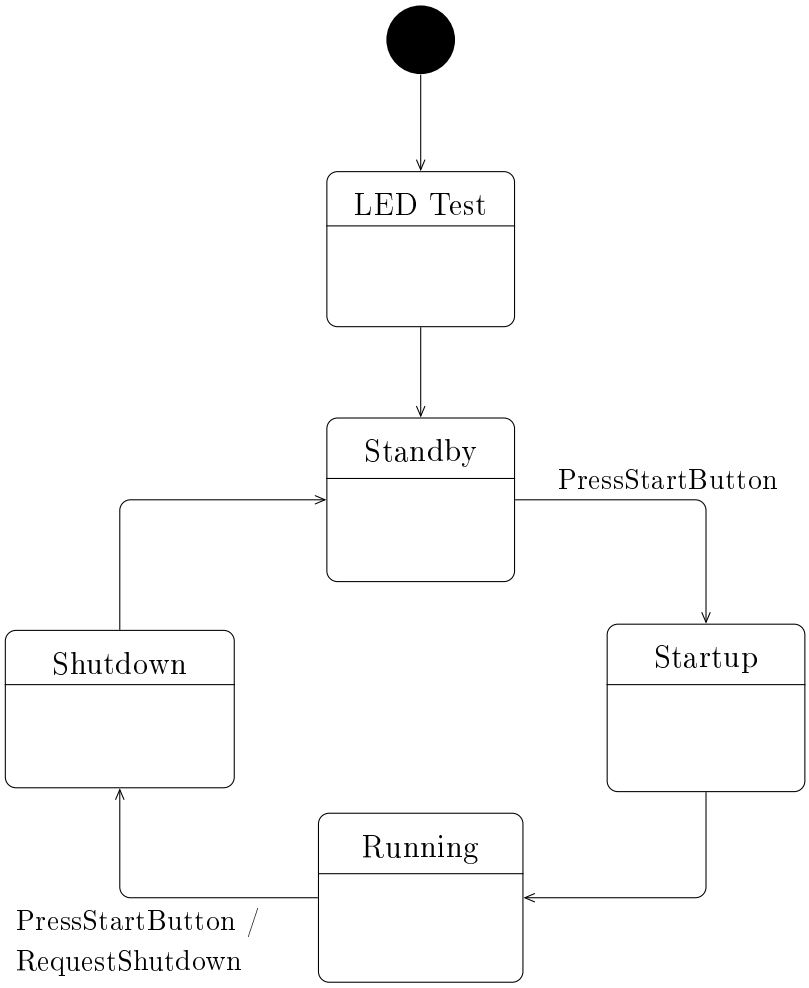


Abbildung 3.7: Zustände des Power Distribution Board

Signal zum Starten des Systems zu warten. Dieser Zustand muss möglichst energiesparend implementiert werden, damit er auch bei Akkubetrieb längere Zeit aufrecht erhalten werden kann. Mit Druck auf den Startknopf wird das System schließlich gestartet. Es wechselt in den *Startup*-Zustand und aktiviert schrittweise und sequenziell die einzelnen Power Rails. Anschließend befindet sich das System im *Running*-Zustand. Hauptaufgabe in diesem Zustand ist es, die in Tabelle 3.2 aufgelisteten elektrischen Parameter zu überwachen.

Die Überwachung erfolgt mit einer Frequenz von 100 Hz, um auch kurzzeitige Strom- und Spannungsspitzen von einigen Millisekunden Dauer messen zu können, ohne zu viel Rechenleistung durch eine zu hohe Abtastrate aufzuwenden. Wird der maximal zulässige Strom auf einer Schiene überschritten,

Symbol	Bedeutung	Min	Typ	Max	Einheit
U_{5V}	Spannung auf 5V-Schiene	4,75	5,0	5,25	V
U_{12V}	Spannung auf 12V-Schiene	11,6	12,0	12,4	V
U_{NUC}	USB-Spannung des Intel NUC	4,75	5,0	5,25	V
U_{prim}	Spannung der Primärquelle	12,0	16,0	18,0	V
U_{sek}	Spannung der Sekundärquelle	13,0	16,0	18,0	V
I_{5V}	Strom auf 5V-Schiene		3,0	5,0	A
I_{12V}	Strom auf 12V-Schiene		1,0	2,0	A
I_{NUC}	Strom auf NUC-Schiene		4,0	5,0	A

Tabelle 3.2: Zu überwachende elektrische Parameter

wird diese, sofern das möglich ist, deaktiviert. Gleiches soll bei der Über- oder Unterschreitung der zulässigen Spannungswerte der Versorgungsschienen passieren. Jede Fehlfunktion soll sowohl durch Anzeige mittels einer Status-LED als auch durch das Senden einer entsprechenden Nachricht auf dem Bus signalisiert werden, sodass andere Busteilnehmer, vor allem das NUC, die Ursache der Fehlfunktion protokollieren können. Die Spannung U_{NUC} soll Auskunft darüber geben, ob sich das NUC im Betriebszustand befindet. Über die Spannung U_{sek} soll der Ladezustand des Akkus approximiert werden, um bei Unterschreitung das System rechtzeitig herunterzufahren und so Schäden, wie etwa ein korruptes Dateisystem, zu vermeiden. Die zweite Aufgabe im *Running*-Zustand ist das Senden der elektrischen Parameter über den Bus. Dies soll anderen Busteilnehmern ermöglichen, die Werte zu akkumulieren und daraus die Leistungsaufnahme des Fahrzeugs für Evaluationszwecke zu berechnen. Ein weiteres Drücken des Startknopfes oder die explizite Shutdown-Anforderung eines anderen Busteilnehmers versetzen das System in den *Shutdown*-Zustand. Das NUC wird durch Simulation eines Drucks auf den Power-Button zum Herunterfahren aufgefordert. Das PDB setzt alle Überwachungsaufgaben solange fort, bis ein vollständiges Abschalten des NUC erkannt wird. Danach werden alle Power Rails deaktiviert und das System wechselt wieder in den *Standby*-Zustand.

3.2.2 Sensor Board (SB)

Die beiden Sensor Boards an der Front- und Rückseite des Fahrzeugs erfüllen prinzipiell die gleichen Aufgaben, weichen aber in der Umsetzung geringfügig

voneinander ab. Die Unterscheidung erfolgt dabei über einen Konfigurationsparameter im persistenten Speicher, welcher beim Start der Firmware eingelesen und ausgewertet wird. Je nach Wert des Parameters wird die entsprechende Funktionalität ausgeführt. Aufgrund dieser Methode kann für beide Varianten die exakt gleiche Firmware benutzt werden, was den Entwicklungsaufwand reduziert und Code-Redundanzen vermeidet.

Bei beiden Varianten werden die angeschlossenen Distanzsensoren mit einer Frequenz von 50 Hz sequenziell abgetastet und die ermittelten Distanzen auf dem Bus veröffentlicht. Da dies bei beiden Board-Varianten nach demselben Prinzip funktioniert, obliegt die Zuordnung der Sensoren zu einem Punkt am Fahrzeug der auswertenden Komponente. Unterschiede zeigen sich vor allem in der Ansteuerung der angeschlossenen Beleuchtung. Da nur an der Frontseite Frontlichter und nur an der Heckseite Rücklichter existieren, müssen diese unterschiedlich angesteuert werden. Die einzelnen Lichtfunktionen sollen als aufrufbare Services zur Verfügung gestellt werden. D. h. dass logische Aktionen wie „Links blinken“ oder „Warnblinklicht an“ als solche von anderen Busteilnehmern angefordert werden. Dieses Konzept steht stark in Kontrast zu dem des vorherigen Fahrzeuges, bei dem die Lichter per Busnachricht auf konkrete Farbwerte im 24-bit-RGB-Format gesetzt wurden. Der Busknoten auf der Recheneinheit hat diese Nachrichten mit einer durchschnittlichen Frequenz von 50 Hz veröffentlicht, was eine unnötige Erhöhung der Busauslastung zur Folge hatte. Da auf der Recheneinheit kein Echtzeitbetriebssystem ausgeführt wird, waren die zeitlichen Abstände zwischen den Nachrichten zudem stark schwankend.

3.2.3 Interface Board (IB)

Das IB kommuniziert durchgehend mit der Fernbedienung und arbeitet primär als Gateway zwischen dem CAN-Bus und dem drahtlosen Netzwerk zwischen den IBs auf den Fahrzeugen und den jeweiligen Fernbedienungen[1]. Die von der Fernbedienung erhaltenen Steuerdaten werden als Nachricht auf dem CAN-Bus veröffentlicht. Das umfasst im Wesentlichen die Zielgeschwindigkeit, den Lenkwinkel und die Information, ob der autonome Modus ein- oder ausgeschaltet ist. Wenn die Fernbedienung die Verbindung verloren hat bzw. außer

[1] Zu einem späteren Zeitpunkt ist die gleichzeitige Verwendung von mehreren Fahrzeugen und mehreren Fernbedienungen vorgesehen.

Reichweite ist oder aus anderen Gründen keine gültigen Daten empfangen werden können, werden trotzdem weiterhin Nachrichten im gleichen Format auf den Bus gesendet, wobei aber die Geschwindigkeit 0 beträgt und ein spezielles Status-Flag gesetzt wird. Das Senden erfolgt in jedem Fall mit einer Frequenz von 50 Hz, um eine flüssige Steuerung zu gewährleisten.

Bei Druck auf einen der drei Buttons wird eine Service-Anfrage an den auf dem Intel NUC laufenden Knoten gesendet, der daraufhin eine entsprechende Aktion ausführt und optional per Service-Anfrage an das IB den gedrückten Button entsprechend beleuchtet.

Konfigurationsparameter

Netzwerk-ID
: Identifier des Funknetzwerkes

Eigene Netzwerkadresse
: Adresse des Empfängers im Funknetzwerk

Netzwerkadresse der Fernbedienung
: Adresse der zugehörigen Fernbedienung im Funknetzwerk

Höchstgeschwindigkeit im manuellen Modus
: Höchstgeschwindigkeit, wenn mit der Fernbedienung gesteuert wird. Wirkt sich nicht auf die Fahrgeschwindigkeit im autonomen Modus aus.

3.2.4 Kommunikationsübersicht

Dieser Abschnitt soll einen Überblick über die softwareseitige Kommunikation zwischen den einzelnen Komponenten geben. Abb. 3.8 zeigt den Nachrichtenfluss und die Benutzung von Services zwischen den Busteilnehmern auf, wobei die Nachrichten in Tab. 3.3 und die Services in Tab. 3.4 erklärt werden.

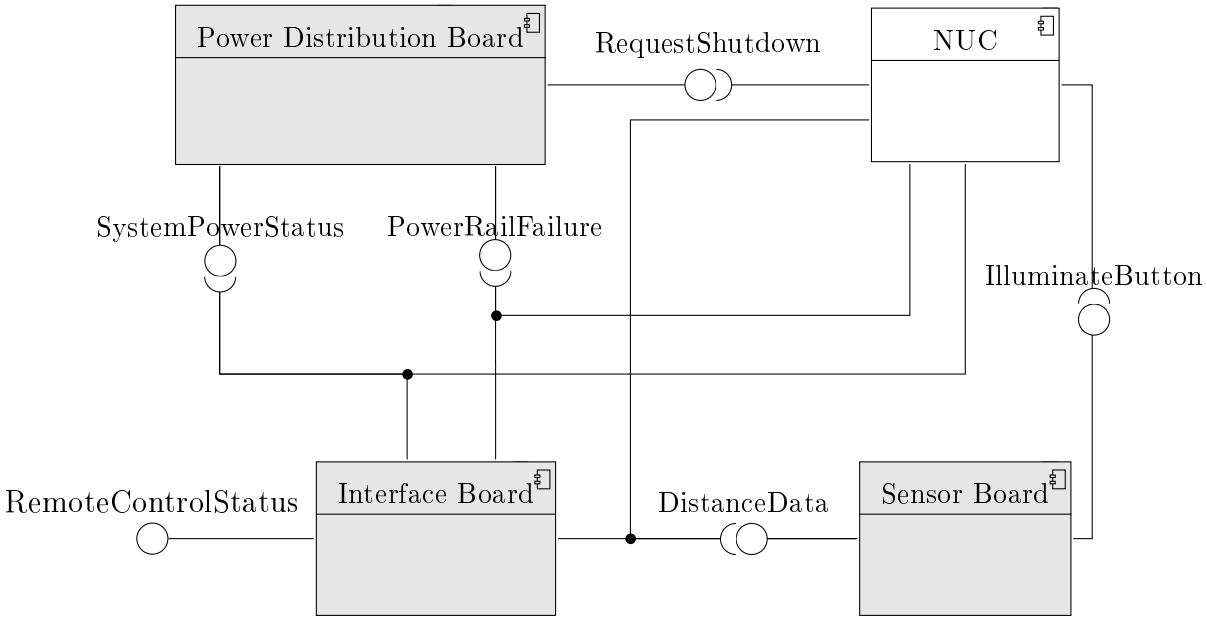


Abbildung 3.8: Zwischen den Netzwerkteilnehmern ausgetauschte Nachrichten und angebotene bzw. konsumierte Services

Sender	Nachricht	Bedeutung
PDB	`SystemPowerStatus`	Enthält die Werte aller in Tabelle 3.2 aufgeführten Größen
	`PowerRailFailure`	Zeigt eine Fehlfunktion einer Versorgungsschiene an
SB	`DistanceData`	Distanzwerte der fünf angeschlossenen Distanzsensoren
IB	`RemoteControlStatus`	Von der Fernbedienung empfangene Steuerdaten für den manuellen Modus

Tabelle 3.3: Übersicht über alle ausgehenden Nachrichten

Anbieter	Service	Bedeutung
PDB	`RequestShutdown`	Fordert eine Abschaltung des gesamten Systems an
IB	`IlluminateButton`	Setzt die Farbe und den Modus der im Button integrierten LED

Tabelle 3.4: Angebotene Services

4 Implementierung

Die Platinen werden gemäß der Anforderungen designt und implementiert. Die Firmwares sind dabei auf die jeweiligen Revisionen der Platinen zugeschnitten. Die dabei verwendeten Softwarewerkzeuge werden hier beschrieben.

4.1 Hardware [MG]

In der Implementierung werden die Konzepte für die Hardware umgesetzt, indem Schaltpläne erstellt und diese mit der Wahl der elektrischen Bauteilen verknüpft werden.

4.1.1 Genereller Aufbau der Platinen

Alle Platinen benötigen für die Buskommunikation einen CAN-Controller und CAN-Transceiver für die Erzeugung und Auswertung der differentiellen Signale. Die Wahl des Mikrocontrollers (MCU) fällt auf den Cortex-M3-basierten *STM32F103RE* [STM18] von STMicroelectronics (STM). Dieser besitzt einen integrierten CAN-Controller. Der Mikrocontroller kommt auch in anderen Projekten von oTToCar zum Einsatz und hat sich dort bewährt. So konnten bereits Erfahrungen mit der Chipserie und ihrer Peripherie gesammelt werden. Er besitzt eine Flashgröße von 512 KiB und eine RAM-Größe von 64 KiB. Damit ist der Chip für unsere Anwendungen ideal, da Softwareteile wie UAVCAN relativ viel Platz im Flash-Speicher benötigen. Als Gehäusetyp wird das 64-Pin LQFP verwendet. Dieses lässt sich einfach mit einem Lötkolben bzw. Heißluft verlöten.

Um den MCU und weitere ICs wie EEPROM, CAN-/Funk-Transceiver etc. betreiben zu können, benötigt man eine Spannung von 3,3 V. Da die Spannungsversorgung der einzelnen Platinen über die 5V-Leitung des CAN-Busses reali-

siert wird, besitzen die Platinen jeweils einen 3,3V-LDO-Regulator (*TC1185-3.3* von Microchip mit maximal 150 mA) für diesen Zweck. Lediglich für Platinen mit einer höheren Stromaufnahme auf der 3,3V-Ebene wird stattdessen einen Stepdown-Wandler *TPS62175* von Texas Instruments Incorporated (TI) mit maximal 500 mA verbaut.

Für die Erzeugung und Auswertung der differentiellen Signale des CAN-Busses ist das IC-Bauteil *SN65HVD230* [Tex18] von TI zuständig. Die Entscheidung fällt auf diesem Chip, da dieser recht klein ist, mit 3,3 V ohne galvanische Trennung arbeitet und bereits in großen Projekten wie z. B. in VESC[1] von Benjamin Vedder erfolgreich eingesetzt wird. Der Abstand zwischen dem Transceiver und der Anschlussbuchse wird so klein wie möglich gehalten und die Leiterbahnen im Platinenlayout als differentielles Leitungspaar ausgeführt.

Es wurde drauf geachtet, dass die Platinen durch Abstecken entfernbar sind, d. h. es sollen so wenig feste Lötverbindungen wie möglich verwendet werden. Für Verbindungen mit hohen Stromflüssen wie solche zwischen den Akkus, VESCs und NUC wird auf *XT30*-, *XT60*- und *MT30*-Steckverbindungen von Amass gesetzt. Für die restlichen Steckverbindungen, allen voran solche für den CAN-Bus, wird die *PicoBlade*-Familie[2] von Molex verwendet. Diese werden mit unterschiedlicher Anzahl an Pins angeboten und deren Vorteile sind zum einen die kleinen Größen der Stecker und Buchsen, zum anderen die recht hohe Strombelastbarkeit dieser. In unserem Fall sind sie mit bis zu 2 A pro Pin/Kabel belastbar. Es werden zudem pro Platine zwei CAN-Anschlüsse bereitgestellt, um eine Platine bequem per Kabel mit einer anderen Platine zu verbinden. Lediglich die Sensor Boards besitzen nur eine CAN-Buchse, da diese in der Verdrahtung als die letzten Busteilnehmer konzipiert sind. Es wird davon ausgegangen, dass die Reflexionen auf dem CAN-Bus trotz dieser Art von Stichleitungen gering genug sind, da die Distanz zwischen den Buchsen nur wenige Millimeter beträgt. Das CAN-Kabel, mit welchem die Platinen miteinander verbunden werden, besteht aus vier Leitungen: CAN-High, CAN-Low, 5V und GND (Masse). Die ersten und letzten beiden Leitungen werden jeweils miteinander verdrillt, um weitere mögliche Störungen zu minimieren.

Alle Schaltpläne und Platinenlayouts werden mit dem OpenSource-Programm *KiCad*[3] erstellt. Das Programm kann sowohl unter Windows als auch Linux

[1] VESC Project: `https://vesc-project.com`
[2] Molex PicoBlade Connector System. 987651-3691 Revision 4. 2019
[3] KiCad EDA - Schematic Capture & PCB Design Software: `https://kicad-pcb.org`

verwendet werden und besitzt viele nützliche Funktionalitäten/Tools, um ein erfolgreiches Design der Platinen voranzutreiben.

4.1.2 Power Distribution Board

Für die Funktion eines unterbrechungsfreien Spannungswechsels wird ein Power-Multiplexer benötigt. Dieser wird mit zwei IC-Chips *TPS24740* [Tex15a] von TI realisiert. Der Chip vereint HotSwap-Controller und OR-ing-Controller in einem und steuert externe MosFETs an. Außerdem bietet er viele Einstellmöglichkeiten, so können z. B. das Limit des Einschaltstromes, Überstromes und die Schwellwerte für Unter-/Überspannung variabel eingestellt werden. Aufgrund der hohen Anzahl an Einstellmöglichkeiten stellt TI

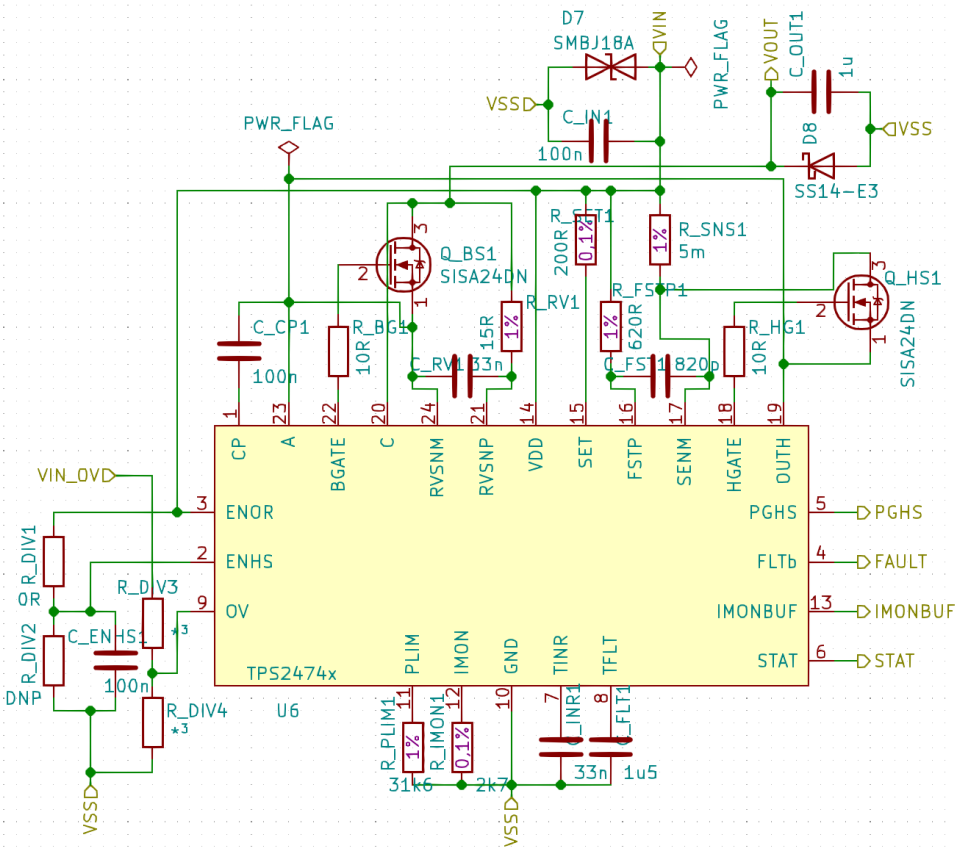


Abbildung 4.1: Ein einzelner TPS2474x-Chip mit den zwei externen MosFETs und der Vielzahl an passiven Komponenten für die Einstellungen der Limits.

zur Vermeidung von Designfehlern eine Excel-Datei als „Design Calculator“ [Tex15b] zur Verfügung, was das Designen einer solchen Schaltung und vor allem das Dimensionieren der nötigen passiven Komponenten enorm erleichtert. Das Resultat der dimensionierten Schaltung ist in der Abb. 4.1 zu sehen. Da ein solcher Chip nur eine Spannungsquelle absichert, werden für zwei Spannungsquellen logischerweise zwei Chips gebraucht. Diese werden mithilfe des *Design Calculators* für den Anwendungsfall des oTToCars dementsprechend dimensioniert und gemäß dem Schema in der Abb. 4.2 zusammen verschaltet. Das Resultat wird in der Abb. 4.3 ersichtlich. Mit dieser Art der Verschaltung

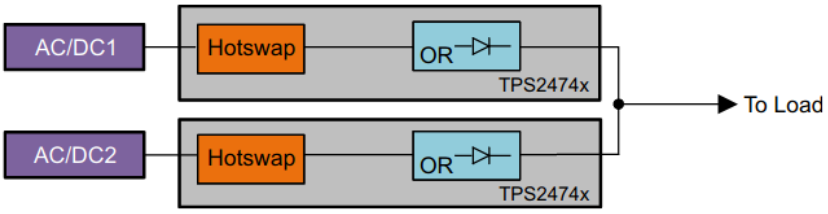


Abbildung 4.2: Schematische Darstellung des Aufbaus der Power-Multiplexer-Schaltung – jede Spannungsquelle wird jeweils mit dem Chip *TPS2474x* abgesichert [Tex15a]

liefert, sofern beide Spannungsquellen angeschlossen sind, physikalisch bedingt immer diejenige Spannungsquelle mit der höchsten Spannung den Strom. Ein nicht gewollter Ausgleichstrom vom höheren zum niedrigeren Spannungslevel wird durch die OR-ing-Schaltung verhindert. Sobald beide Spannungsquellen die gleiche Spannung vorweisen, teilen sich beide den Ausgangsstrom zu einem gleichen Verhältnis. Ursprünglich war geplant, dass die Power-Multiplexing-Schaltung mit einer Priorisierung für die sekundäre Spannungsquelle arbeitet, d. h. unabhängig von der Höhe der Spannung der sekundären Spannungsquelle wird diese immer bevorzugt, sobald sie angeschlossen wird. Allerdings traten beim Testen leider unlösbare Probleme auf, die ein stabiles und sicheres Arbeiten der Power-Multiplexing-Schaltung unmöglich machten. Das Problem liegt hier in der Art und Weise, wie die Priorisierung stattfindet. Dazu wurde die für diesen Anwendungsfall empfohlene Schaltung aus dem Datenblatt [Tex15a] genutzt. Diese Schaltung hat jedoch ihre elektrischen Tücken, welche die Entwickler eventuell nicht bedacht haben – diese führen dazu, dass eine zuverlässige Priorisierung und generell ein robustes Arbeiten nicht garantiert werden können. So wurde auf die Priorisierung verzichtet und stattdessen die Schaltung zu einer, wie in der Abb. 4.2 dargestellt, umgebaut.

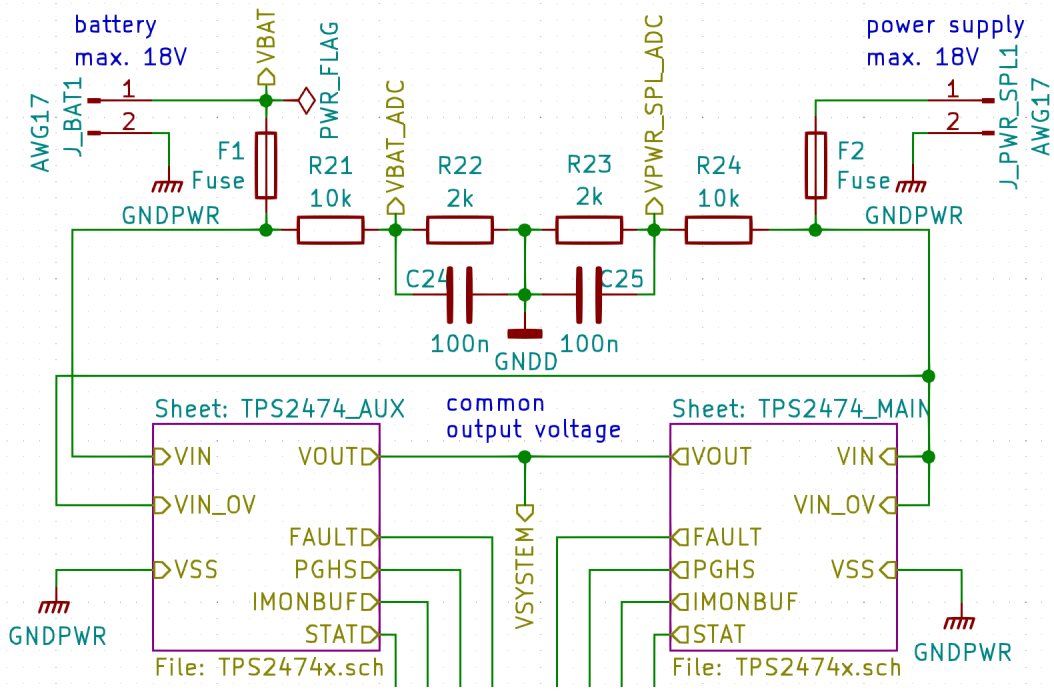


Abbildung 4.3: Komplette Power-Multiplexer-Schaltung – allerdings ist hier die (später verworfene) Priorisierung noch implementiert. Die Spannungsquellen werden, jeweils abgesichert über eine Sicherung, an den jeweiligen Power-Multiplexer-Chip angeschlossen. Die Ausgänge der beiden Power-Multiplexer werden zusammengeschaltet und bildet so den Bezugspunkt der Systemspannung.

Die integrierten *HotSwap-Controller* verhindern hohe Einschaltströme, wenn die Pufferkondensatoren des *Power Distribution Boards* noch nicht aufgeladen sind. Die Power-Multiplexing-Schaltung wird mit 18V-TVS-Dioden abgesichert. Solche Dioden leiten alle Spannungen, welche größer gleich 18 V betragen, sofort ab. So wird die Schaltung robust gegenüber Überschwingen und Spannungsspitzen, die bei Lastwechseln an den Spannungsquellen bzw. beim Rekuperieren der Motoren entstehen können.

Um die 5V- und 12V-Spannungen bereit stellen zu können, wird die Systemspannung, welche der Power-Multiplexer ausgibt, mithilfe zweier Stepdown-Wandler (*RT8298* von Richtek mit maximal 6 A) jeweils auf die anfangs genannten Spannungen herunter transformiert. Beide Spannungswandler benutzen dasselbe Design, welches sich lediglich in der Dimensionierung eines

Spannungsteilers unterscheidet, welcher die Ausgangsspannungen von 5 V und 12 V bestimmt. Die Spannungswandler sind vom MCU zu- und abschaltbar. Es wurde darauf geachtet, dass die Strompfade in Form von Leiterbahnen dementsprechend breit genug sind, um hohe Ströme leiten zu können. Um die Ströme auf den 5V-, 12V- und NUC-Leitungen messen zu können, werden Shunt-Widerstände verbaut, deren Spannungsabfall jeweils mit einem Operationsverstärker (*INA4180* von TI) verstärkt wird, damit sie an den analogen Eingängen des MCU mit einer hohen Auflösung eingelesen werden können. Da eine analoge Messung höhere Ansprüche an die analoge Spannungsversorgung des MCU und Operationsverstärkers erfordert, werden diese mithilfe einer Ferritperle, getrennter Masseflächen und mehrerer Kondensatoren von der restlichen Spannungsversorgung entkoppelt. Eine Ferritperle filtert Störungen, indem sie hochfrequente Spitzen bedingt durch ihre induktive Impedanz nicht durchlässt, während niederfrequente Spannungen ohne Probleme durchgelassen werden. Die analogen Messungen können bei Bedarf durch Bestückung der passiven Komponenten auf der Platine mithilfe eines RC-Glieds (Tiefpassfilter der 1. Ordnung) bzw. LC-Glieds (Tiefpassfilter der 2. Ordnung) geglättet werden.

Um den Einschaltstrom für die VESCs zu begrenzen, wird ein weiterer HotSwap-Controller (*MIC2587* von Micrel) implementiert. Dieser lädt mit einem konstanten Strom das Gate eines MosFETs auf, sodass durch diesen, bedingt durch den ohmschen Bereich, nur ein geringer Strom fließen kann und so die großen Pufferkapazitäten der VESCs langsam auflädt. Die implementierte Schaltung lässt einen Rückstrom zu – dieser ist wichtig, da die Bremsenergie der Motoren zurück in die Batterie rekuperiert werden soll.

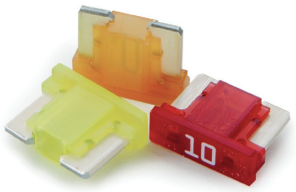


Abbildung 4.4: „Low Profile Mini“ Sicherungen[1]

[1] Littelfuse:
https://www.littelfuse.com/~/media/automotive/datasheets/fuses/passenger-car-and-commercial-vehicle/blade-fuses/littelfuse_lowprofilemini_datasheet.pdf

Für die Absicherungen der Platinen werden die kleinsten im Handel erhältlichen Kfz-Sicherungen der Baureihe „Low Profile Mini“ (siehe Abb. 4.4) verwendet. Diese eignen sich wegen ihrer geringen Steckgröße hervorragend für unsere platzkritische Anwendung.

4.1.3 Sensor Board

Das Sensor Board stellt einen 2,8V-LDO-Regulator (*TC1185-2.8* von Microchip mit maximal 150 mA) und eine I^2C-Schnittstelle für fünf Distanzsensoren (*VL53L0x* von STM) zur Verfügung. Die Limitierung auf fünf ist nicht I^2C-bedingt; alle Distanzsensoren starten nach einem Power-Reset mit derselben I^2C-Adresse, was eine Neuzuweisung der I^2C-Adressen erforderlich macht. Aus diesem Grund müssen die verwendeten Distanzsensoren einzeln nacheinander mit jeweils einer Chip-Enable-Leitung vom MCU aktiviert werden und durch das Beschreiben eines I^2C-Registers wird ihnen eine eindeutige I^2C-Adresse zugewiesen. Für diesen Zweck werden deshalb fünf Chip-Enable-Leitungen ausgeführt. Außerdem bietet das Sensor Board eine Schnittstelle zum Ansteuern der adressierbaren LEDs, welche z. B. für das Bremslicht und den Blinker verwendet werden. Die Ansteuerung kann je nach verwendetem LED-Typ sowohl synchron als auch asynchron erfolgen. Des Weiteren besitzt die Platine einen EEPROM (*24LC64* von Microchip mit 64 Kibit Speicher), um bspw. Konfigurationen und Farbmuster der anzusteuernden LEDs abzuspeichern. Die Abb. 4.5 zeigt die gerenderte Ansicht des Sensor Boards.

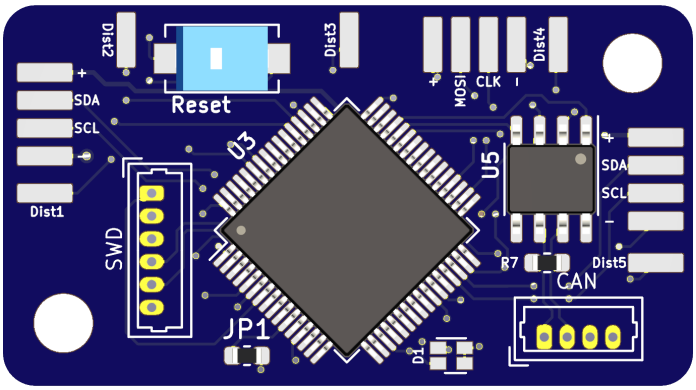


Abbildung 4.5: Sensor Board mit seinen Anschlussmöglichkeiten für die LEDs und Distanzsensoren

4.1.4 Interface Board

Das Interface Board dient als Rückfallebene für das Touchscreen-Display, indem es drei illuminierte Buttons bereitstellt. Für die Illuminierung besitzen die Buttons jeweils eine rote und grüne LED. Eine Mischung der beiden LED-Farben zu Gelb bzw. Orange ist möglich. Des Weiteren wird die RF-Hardware überarbeitet. Auf dem alten Auto wurde diese mit *deRFSam1310* von *dresden elektronik* realisiert, welcher einen Cortex-M3 und RF-Transceiver auf einer kleinen (zusätzlichen) Platine besitzt. Da auf allen Platinen mit dem *STM32F103RE* bereits ein Cortex-M3 verbaut und durch seinen integrierten CAN-Controller unabdingbar ist, ergibt es wenig Sinn, zwei leistungsstarke Cortex-M3-Chips auf derselben Platine zu verbauen. So wird, um Software- und Hardwareaufwand zu minimieren, auf den einfachen Einsatz eines RF-Transceivers AT86RF212B [Atm15] von Atmel/Microchip umgestiegen, welcher direkt über Serial Peripheral Interface (SPI) vom MCU angesprochen werden kann. Durch diesen Umstieg entfällt der Softwareaufwand für die Kommunikation zwischen den beiden Cortex-M3-Chips und für die Firmware des zweiten Chips. Beim Erstellen des RF-Layouts für den Funk wurde Wert auf

Abbildung 4.6: Vergleich: MMCX (unten) und SMA[1]

die Empfehlungen des Herstellers des Funktransceivers gelegt. So wurde unter anderem ein fertiger Balun verbaut, welcher das aufwendige Dimensionieren eines Filternetzwerks für die Antenne erspart. „Ein **Balun** (englisch **bal**anced-**un**balanced) ist [...] ein Bauteil zur Wandlung zwischen einem symmetrischen Leitungssystem und einem unsymmetrischen Leitungssystem.“[Wik19] Am Ende wurde das RF-Layout mit einem Shield abgeschirmt, um RF-Interferenzen und Störungen zu minimieren. Neu ist auch der Anschluss für die Antenne. Die

[1] DELOCK 88580 WLAN Kabel:
`https://www.reichelt.de/wlan-kabel-sma-einbaubuchse-mmcx-stecker-delock-88580-p179753.html`

Entscheidung fiel auf einen Anschluss über *MMCX* (Micro-Miniature Coaxial), welche in der Abb. 4.6 dargestellt wird. Diese zeichnet sich durch ihren kleinen Formfaktor und hervorragende Steckeigenschaften (leichtes Ein- und Abstecken, hohe Anzahl an Steckzyklen) aus. Die Platine hat ebenfalls einen EEPROM zwecks Konfigurationsmöglichkeiten verbaut.

4.1.5 CAN-USB-Bridge

Wie bereits im Kapitel Konzept erwähnt, basiert diese Platine auf einem bereits bestehenden Projekt[1] auf GitHub, welches mit einer dazugehörigen Firmware[2], ebenfalls verfügbar auf GitHub, geflasht wird. Um eine Änderung der Firmware zu vermeiden, wurde der Schaltplan weitestgehend übernommen, inklusive der Wahl des MCUs (*STM32F072C8* von STM), welcher sich von den auf anderen Platinen verbauten MCU unterscheidet. Es wurden lediglich kleinere Anpassungen vorgenommen, was die Wahl der verbauten Komponenten betrifft. So besitzt diese Platine z. B. den gleichen 3,3V-LDO-Regulator und die gleichen Steckverbindungen, so wie sie auch auf den anderen Platinen vorkommen.

[1] `https://github.com/HubertD/candleLight`
[2] `https://github.com/candle-usb/candleLight_fw`

4.2 Softwarewerkzeuge [TW]

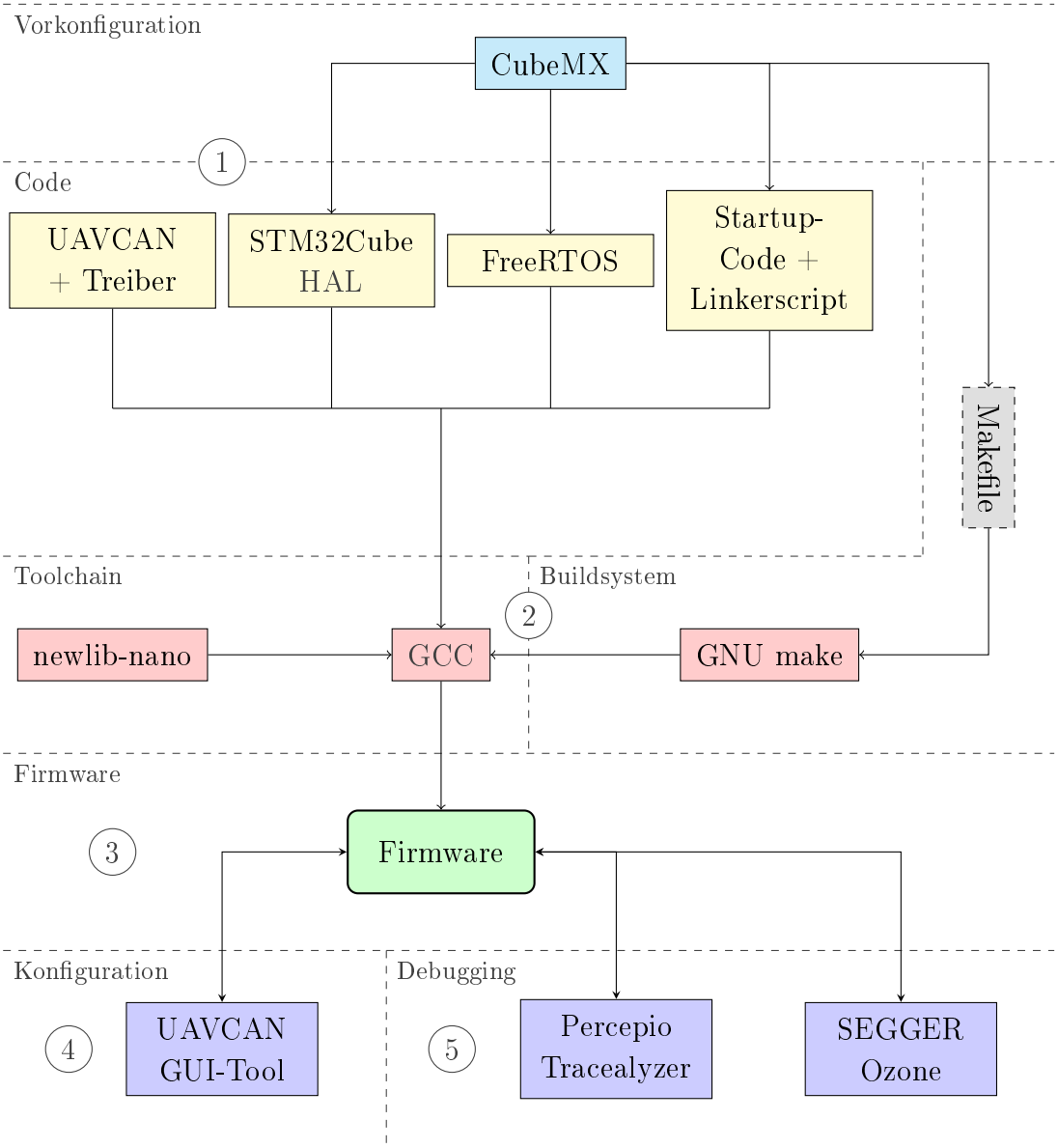


Abbildung 4.7: Übersicht über die zum Erstellen der Firmware verwendeten Softwarewerkzeuge

Dieser Abschnitt soll eine Übersicht über den Erstellungsprozess der einzelnen Firmwares und der daran beteiligten Softwarekomponenten geben. Anschließend werden die einzelnen Softwarekomponenten genauer erläutert. Abbildung

4.7 skizziert dabei grob die Zusammenhänge und Abhängigkeiten der einzelnen Komponenten sowie den Erstellungsprozess; die Nummern in den Kreisen entsprechen dabei den einzelnen Schritten.

1. Mit CubeMX wird der MCU vorkonfiguriert. Das schließt unter anderem die Konfiguration der Pins, Peripheriemodule und Taktquellen ein. Ergebnis des Prozesses sind Initialisierungscode sowie ein Makefile und ein Linkerscript, welche für die nachfolgenden Schritte benötigt werden. Außerdem stellt CubeMX die Komponenten STM32Cube HAL und FreeRTOS in Form von Code zur Verfügung.
2. GNU make führt nun auf Basis des vorher generierten Makefiles verschiedene Aufrufe an die GNU Compiler Collection (GCC) aus. Dabei wird sämtlicher Code kompiliert und anschließend mit dem Linkerscript aus Schritt 1 zu einer Binärdatei zusammengefügt, die später in den Flash-Speicher des MCUs geschrieben wird.
3. Die Binärdatei wird mit einem *SEGGER J-Link*[1] unter Verwendung der zugehörigen Software in den Flash-Speicher des MCU geschrieben[2]. Nach Abschluss des Schreibvorgangs und einem Reset des MCU wird die Firmware nun ausgeführt.
4. Für die Firmwares des IB und des SB kann nun eine Konfiguration über den Bus vorgenommen werden. Dazu werden mit Hilfe des UAVCAN-GUI-Tools die entsprechenden Parameter auf einer grafischen Benutzeroberfläche eingestellt.
5. Zusätzlich kann ein Debugging bzw. eine Laufzeitanalyse der Firmwares mit dem Percepio Tracealyzer und SEGGER Ozone erfolgen.

4.2.1 Toolchain

Als Toolchain wird auf die *GCC* für ARM-Prozessoren ohne Betriebssystem *(arm-none-eabi)*[3] in der Version 9.3 zurückgegriffen, die Compiler, Linker und weitere benötigte Tools enthält. Dabei handelt es sich um eine der wenigen kostenlosen Alternativen; andere Compiler sind in den meisten Fällen für dieses

[1] `https://www.segger.com/products/debug-probes/j-link/`

[2] Dieser Prozess wird im Rahmen dieser Arbeit als trivial erachtet, weshalb auf die dafür verwendete Software nicht weiter eingegangen wird.

[3] `https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads`

```
constexpr std::array VoltageDividerFactors = {
    2.0_, // V_5V
    4.0_, // V_12V
    2.0_, // V_NUC_USB
    6.0_, // V_PWRSPL
    6.0_  // V_BAT
};
```

Listing 4.1: Wegfall von expliziten Template-Parametern durch *CTAD*

Projekt finanziell unerschwinglich oder in ihrer kostenfreien Version, vor allem in Bezug auf die maximal zulässigen Speichergrößen der resultierenden Binärdatei, zu stark limitiert, als dass sie für diese Arbeit geeignet wären. GCC arm-none-eabi wird von ARM selbst aktiv gepflegt und bietet dementsprechend im Vergleich zu diversen anderen Integrated Development Environments (IDEs) die Nutzung aktueller C++17-Sprachfeatures. So wird für die Firmwares auf Sprachfeatures wie *CTAD*[1], *Constexpr If*[2] und *Nested Namespaces*[3] zurückgegriffen.

Listing 4.1 zeigt die Benutzung von CTAD: Vor C++17 mussten hier für `std::array` explizit die beiden Template-Parameter für den zugrunde liegenden Typ und die Arraygröße angegeben werden, im Beispiel also `std::array<units::si::Scalar, 5>`.

```
template <typename Type = float>
constexpr Type getMagnitude() const noexcept
{
    if constexpr (std::is_same<Type, float>::value)
        return magnitude;
    else
        return static_cast<Type>(magnitude);
}
```

Listing 4.2: Auswertung von bedingter Ausführung zur Kompilierzeit mit *Constexpr If*

[1] https://en.cppreference.com/w/cpp/language/class_template_argument_deduction
[2] https://en.cppreference.com/w/cpp/language/if#Constexpr_If
[3] https://en.cppreference.com/w/cpp/language/namespace

Mit *Constexpr If* kann einer der beiden Ausführungszweige bei einer bedingten Verzweigung bereits zur Kompilierzeit verworfen werden, wenn die Bedingung selbst zur Kompilierzeit ausgewertet werden kann. So kann für einige Funktionen der Laufzeit-Overhead reduziert werden. In Listing 4.2 wird so auf die Notwendigkeit einer Typumwandlung geprüft.

```
namespace units::si
{
    using Voltage = Value<SiUnit<2, 1, -3, -1, 0, 0, 0>>;
}
```

Listing 4.3: *Nested namespaces* erlauben eine einfachere Schreibweise bei verschachtelten Namensräumen.

Nested Namespaces ermöglichen eine einfachere Schreibweise von ineinander verschachtelten Namensräumen. Vor C++17 hätte der Code in Listing 4.3 folgendermaßen aussehen müssen: `namespace units { namespace si { ... }}`. Durch die Nutzung dieser Sprachfeatures wird die Lesbarkeit des Codes insbesondere bei häufiger Verwendung von *Templates* und Submodulen (in Form von *Namespaces*) stark verbessert. Bei vielen Linux-Distributionen ist jedoch die GCC-Version (für arm-none-eabi), die aus den offiziellen Paketquellen bezogen werden kann, veraltet und bietet keine Unterstützung für C++17. In diesen Fällen muss die Software aus inoffiziellen Quellen bezogen oder selbst kompiliert werden.

Als Implementierung für die C-Standard-Bibliothek wird *newlib-nano* benutzt, welche ebenfalls in der Toolchain enthalten ist. Sie stellt eine minimale Implementierung speziell für eingebettete Systeme mit beschränkten Ressourcen dar und nimmt deshalb wenig Progammspeicher ein. Bei der Nutzung müssen jedoch einige Dinge beachtet werden. Viele Bibliotheksfunktionen sind nicht thread-safe, d. h. die konkurrente Nutzung einer Funktion von mehreren Tasks kann bei präemptivem Multitasking zu Problemen führen. Zudem rufen einige Funktionen intern `malloc()` zur Allokation von dynamischen Speicher auf, obwohl dies hier nicht erwünscht ist, da auf die Verwendung von dynamischem Speicher weitgehend verzichtet wird. Ausnahmen stellen dabei Initialisierungsfunktionen dar, die nur einmal zum Start der Firmware ausgeführt werden. Diese verwenden allerdings nicht `malloc()`, sondern von FreeRTOS

bereitgestellte Funktionen[1], die einen zeitlich deterministischen Algorithmus zur Allokation verwenden und zusätzlich thread-safe sind.

4.2.2 Hardware Abstraction Layer (HAL)

Um die Hardware des gewählten MCUs anzusprechen, d. h. Peripheriekomponenten und Cortex-M3-interne Komponenten wie den Interrupt-Controller zu benutzen, muss auf deren Register zugegriffen werden. Da manuelle Registerzugriffe im Code aber nur schwer lesbar und unübersichtlich sind, wird für diese Zugriffe eine Firmwarebibliothek bzw. MCU-übergreifend ein HAL verwendet.

In der vorherigen oTToCar-Iteration kam als Firmwarebibliothek *libopencm3*[2] zum Einsatz. Dabei handelt es sich um eine Open-Source-Bibliothek für eine ganze Reihe von ARM-Cortex-M-basierten Mikrocontrollern, die hardwarenahe Zugriffe sowohl auf die Core Peripherals als auch auf herstellerspezifische Peripheriemodule ermöglicht. Ursprünglich war vorgesehen, auch weiterhin libopencm3 im Rahmen dieser Arbeit zu verwenden, jedoch erwies sich die Kombination der Bibliothek und des UAVCAN-Treibers für STM32-Chips als problematisch. Da sowohl der vom Treiber erforderliche Cortex Microcontroller Software Interface Standard (CMSIS) als auch libopencm3 identische Definitionen (wie zum Beispiel Registerdefinitionen) in Form von gleich benannten Präprozessor-Makros mitbringen, führt dies zu einer Vielzahl von Compilerfehlern. Zudem wird libopencm3 nicht aktiv weiterentwickelt und enthält Bugs, von denen ältere Versionen der Firmwares betroffen waren.

Aus diesen Gründen fiel die Wahl eines neuen HAL auf STM32Cube HAL. Diese Bibliothek wird direkt vom Hersteller der eingesetzten MCUs entwickelt und wird auch für ältere Chipgenerationen noch aktiv gepflegt. Durch die weite Verbreitung lassen sich zu vielen Anwendungsszenarien Codebeispiele sowohl von Seiten des Entwicklers als auch von der Community finden. Als größter Vorteil erweist sich für diese Arbeit jedoch die Möglichkeit der Verwendung eines grafischen Tools, mit dem die grundlegende Initialisierung des MCU vorgenommen und automatisiert Code erzeugt werden kann, welcher die nötigen Aufrufe an den HAL enthält.

[1] `https://www.freertos.org/a00111.html`
[2] `http://libopencm3.org/`

4.2.3 STM32CubeMX

Alle für die Funktionsweise der Firmware benötigten Komponenten des MCU wie z. B. SPI, CAN-Controller oder aber auch der Clock Tree müssen vor der Benutzung initialisiert werden. Dies erfolgt über mehrere Aufrufe an den STM32Cube HAL, welche je nach Komponente mehr oder weniger komplex ausfallen und dementsprechend viel Zeit bei der Programmierung erfordern können. Um diese Zeit zu verkürzen und den Initialisierungsprozess zu vereinfachen, wird hier STM32CubeMX[1] verwendet. Dabei handelt es sich um ein grafisches Konfigurationsprogramm von STM für Mikrocontroller der STM32-Familie, das C-Code für die Initialisierung dieser erzeugt. Es ermöglicht die Konfiguration von Clocks, GPIOs, Peripheriemodulen, das Aktivieren von Interrupts und die Einbindung von diversen Middlewares, wie etwa FreeRTOS. Da die Konfigurationen grafisch erfolgen und für viele Einstellungen die möglichen Optionen vorgegeben werden, wird der Workflow gegenüber der klassischen Konfiguration nur durch Code wesentlich vereinfacht. Zudem entfällt häufig das zeitaufwendige Suchen der passenden Optionen in der API-Dokumentation des HAL. Besonders hervorzuheben ist hierbei die Darstellung des Clock Trees in einem eigenen Editor, die der im Datenblatt des jeweiligen MCUs nachempfunden ist. Dieser Editor bietet auch die Möglichkeit, die Taktquellen und -frequenzen für bestimmte Komponenten festzulegen und automatisch alle anderen Parameter so anzupassen, dass eine valide Konfiguration entsteht (siehe Abb. 4.8).

Ferner prüft CubeMX auf Fehlkonfigurationen und weist auf diese hin, wie im Falle eines falsch konfigurierten Clock Trees, oder verhindert diese durch Deaktivieren der Konfigurationsoptionen, die zu einer solchen Konfiguration führen würden. Wenn beispielsweise für ein Peripheriemodul benötigte Pins bereits durch ein anderes Peripheriemodul belegt sind, wird darauf mit einer detaillierten Fehlermeldung aufmerksam gemacht.

Der aus der Konfiguration generierte C-Code enthält alle benötigten Aufrufe an den STM32Cube HAL, um den MCU wie gewünscht zu initialisieren. Außerdem werden alle benötigten Dateien des STM32Cube HAL automatisch mit in das Zielverzeichnis kopiert, sofern dies in den Projektoptionen konfiguriert wird. Neben Initialisierungscode können auch toolchain-spezifische Dateien mitgeneriert werden. Im Rahmen dieser Arbeit werden zusätzlich ein

[1] `https://www.st.com/en/development-tools/stm32cubemx.html`

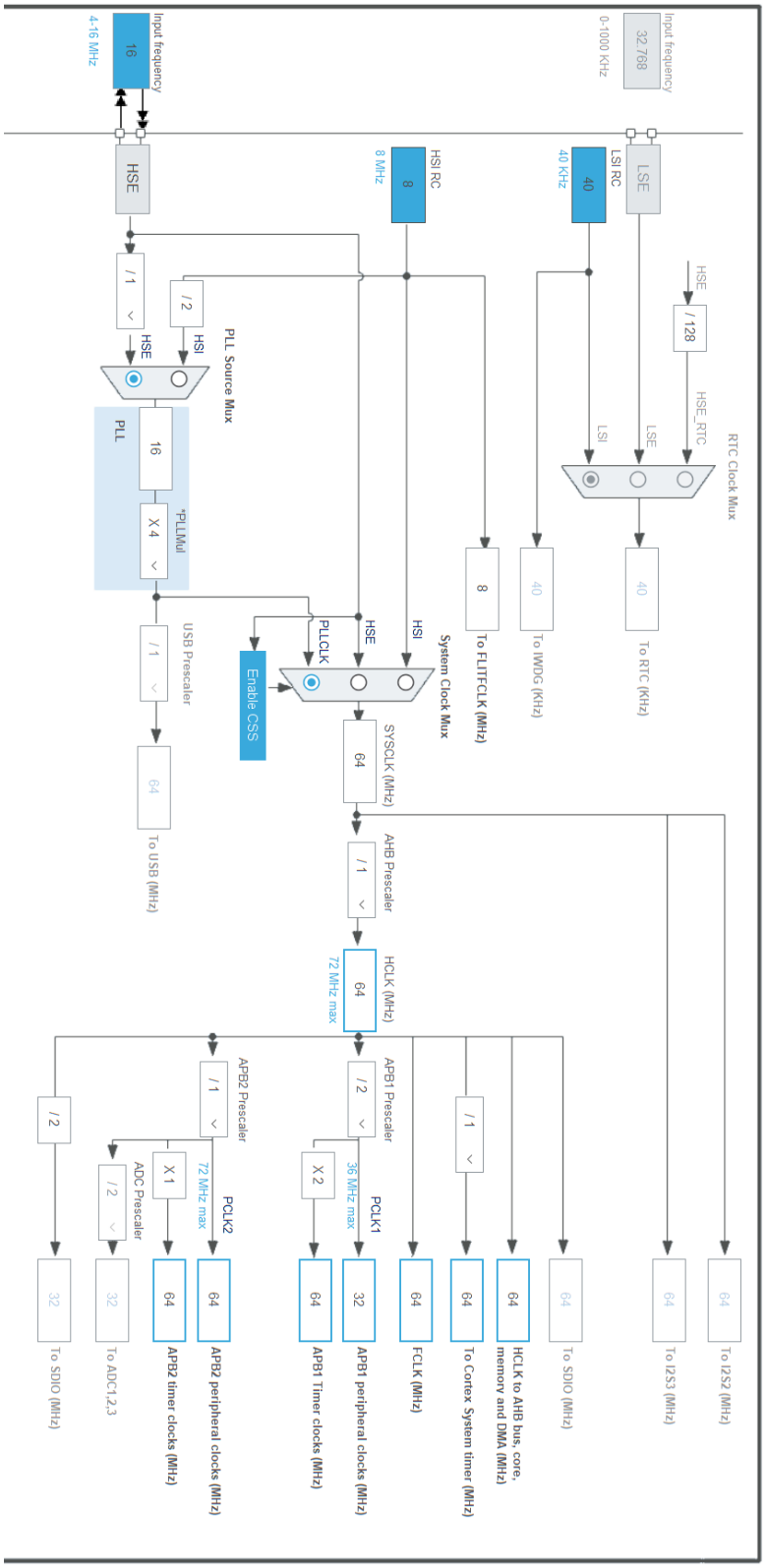


Abbildung 4.8: Ausschnitt aus der Clock-Konfiguration des Interface Boards

Makefile, Startup-Code in Assembler und ein GNU-Linkerscript mitgeneriert. Dabei muss das Makefile jedoch an das für diese Arbeit genutzte, spezifischere Buildsystem angepasst werden.

4.2.4 FreeRTOS

Die Aufgaben der Firmware sind größtenteils zu komplex, als dass sie in einem Super-Loop-Design umgesetzt werden könnten. So verfügt z. B. das IB mit dem Verarbeiten der von der Fernbedienung empfangenen Nachrichten und dem Bus-Handler über zwei Tasks, die unterschiedliche Frequenzen und vor allem verschiedene Prioritäten besitzen. Aus diesem Grund ist es nötig, ein RTOS zu verwenden. Die Entscheidung fällt auf FreeRTOS[1], welches präemptives und kooperatives Multitasking unterstützt und Portierungen für viele Prozessorarchitekturen besitzt, darunter auch die des von uns gewählten MCU implementierte Cortex-M3-Architektur. Es unterstützt Kommunikationsprimitiven wie *Message Queues* und *Stream Buffers* sowie *Mutexe* und *Semaphoren* als wichtigste Synchronisationsmittel. Da FreeRTOS sehr weit verbreitet ist, sind dementsprechend viel Literatur und Beispielcode verfügbar. Zudem wurde es bereits erfolgreich für die Firmware der Fernbedienung eingesetzt. Einen weiteren, erheblichen Vorteil gegenüber den Alternativen stellt die direkte Unterstützung in CubeMX dar: Wenn die entsprechende Option bei der Konfiguration aktiviert wird, wird automatisch der FreeRTOS-Code (über die Abstraktionsschicht *CMSIS OS*) mit eingebunden.

4.2.5 libuavcan

Für die Buskommunikation wird, aufbauend auf CAN, UAVCAN verwendet. Um die Spezifikation des Protokolles nicht selbst implementieren zu müssen, wird auf die Referenzimplementierung *libuavcan* zurückgegriffen. Diese ist in C++ geschrieben und kann sowohl auf eingebetteten Systemen ohne Betriebssystem als auch unter GNU/Linux eingesetzt werden[2]. Zusätzlich zur eigentlichen Protokollimplementierung wird ein für das jeweilige System geschriebener Treiber benötigt, der die Ansteuerung der Register des CAN-Controllers und

[1] `https://www.freertos.org/`
[2] `https://github.com/UAVCAN/libuavcan/tree/legacy-v0`

die Interruptverwaltung im MCU bzw. die Kommunikation mit dem CAN-Netzwerkinterface unter Linux übernimmt. Da wir die Bibliothek auf verschiedenen Plattformen, den STM32-Mikrocontrollern und dem Intel NUC ausführen, benötigen wir auch zwei unterschiedliche Treiber. Diese bringt libuavcan bereits beide als `stm32/libuavcan` und `linux/libuavcan` mit.

4.2.6 UAVCAN GUI Tool

Das UAVCAN GUI Tool[1] ermöglicht die Überwachung und Diagnose eines UAVCAN-Netzwerkes mittels einer grafischen Oberfläche. Es wird auf dem Intel NUC ausgeführt, der über die CAN-USB-Bridge mit dem Netz verbunden ist und stellt einen eigenständigen Netzwerkteilnehmer dar. Neben dem Anzeigen von Statusinformationen zu den einzelnen Netzwerkteilnehmern erlaubt das Tool auch deren Konfiguration durch Ändern von übermittelten Parametern, die dann in den persistenten Speicher des jeweiligen Boards geschrieben werden. Zusätzlich sind erweiterte Funktionen wie das Neustarten eines Knotens oder ein Firmware-Upgrade möglich, sofern die Teilnehmer dies implementieren. Abb. 4.9 zeigt eine Beispielansicht.

Außerdem können alle über den Bus laufenden oder auch nur bestimmte Nachrichten aufgezeichnet werden. Die Inhalte der Nachrichten können dann grafisch in einem Diagramm dargestellt werden, z. B. der Wert einer oder mehrerer Größen über die Zeit.

4.2.7 Percepio Tracealyzer

Bei Entwicklung der Firmware kommt es unausweichlich zu Fehlern. Einige davon können durch das bloße Beobachten des Verhaltens erkannt werden, andere hingegen treten weniger offensichtlich in Erscheinung. Insbesondere bei der Verwendung von FreeRTOS sind Fehlerursachen durch das präemptive Multitasking oft nur schwer mit einem Debugger wie dem GNU Debugger (GDB) auszumachen, da dieser auf Quellcode- bzw. Assembler-Ebene arbeitet und daher keine Kenntnisse über Tasks und deren verwendete Kernel-Objekte besitzt. Deshalb ist es für die Fehlersuche in solchen Fällen wichtig, Einsicht in die internen Abläufe aus Sicht des RTOS zu erlangen. Diese Möglichkeit

[1] `https://github.com/UAVCAN/gui_tool`

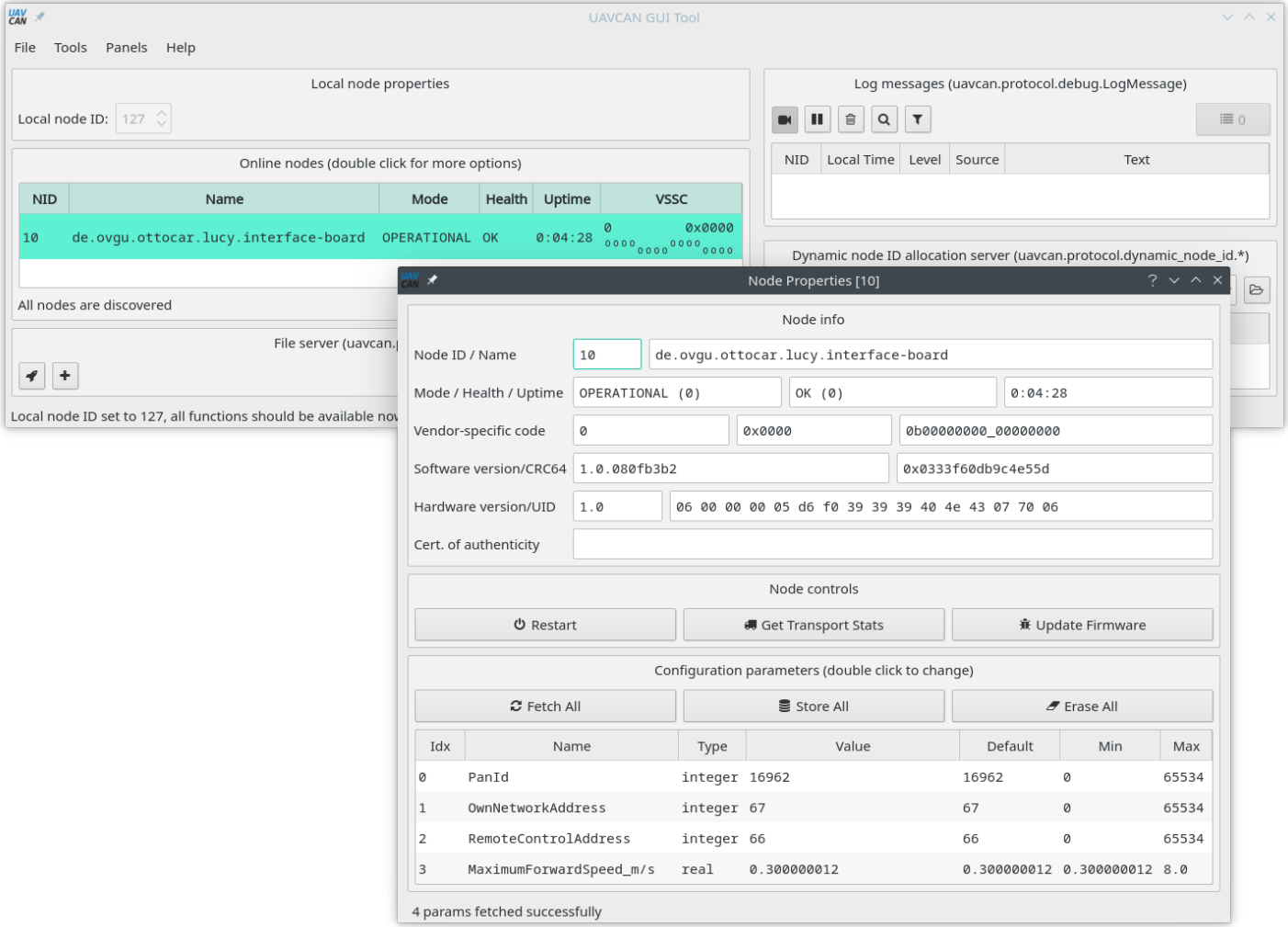


Abbildung 4.9: UAVCAN-Busdiagnose mit dem UAVCAN GUI Tool. Das Fenster im Hintergrund listet alle erkannten Netzwerkteilnehmer zusammen mit ihrem Status auf, das Fenster im Vordergrund liefert detaillierte Informationen zu einem Teilnehmer und dessen konfigurierbaren Parametern (hier zum Interface Board).

wird vom Tracealyzer[1] von Percepio geboten. Tracealyzer ist eine Software für die Diagnose und Visualisierung von internen Abläufen eingebetteter Systeme und insbesondere RTOS. Im Rahmen dieser Arbeit wird sie genutzt, um diverse Ereignisse des in der Firmware benutzten FreeRTOS nachzuverfolgen und zu visualisieren. Das schließt u. a. Taskwechsel, die Benutzung von Kernel-Objekten, das Akquirieren und Freigeben von Semaphoren sowie das Auftreten von Interrupts ein (siehe Abb. 4.10). Ferner kann auch das zeitliche Verhalten der Tasks analysiert werden, um Rückschlüsse auf die Prozessorauslastung und Antwortzeiten zu ziehen und so Probleme wie verzögerte Taskausführungen zu erkennen.

[1] `https://percepio.com/tracealyzer/`

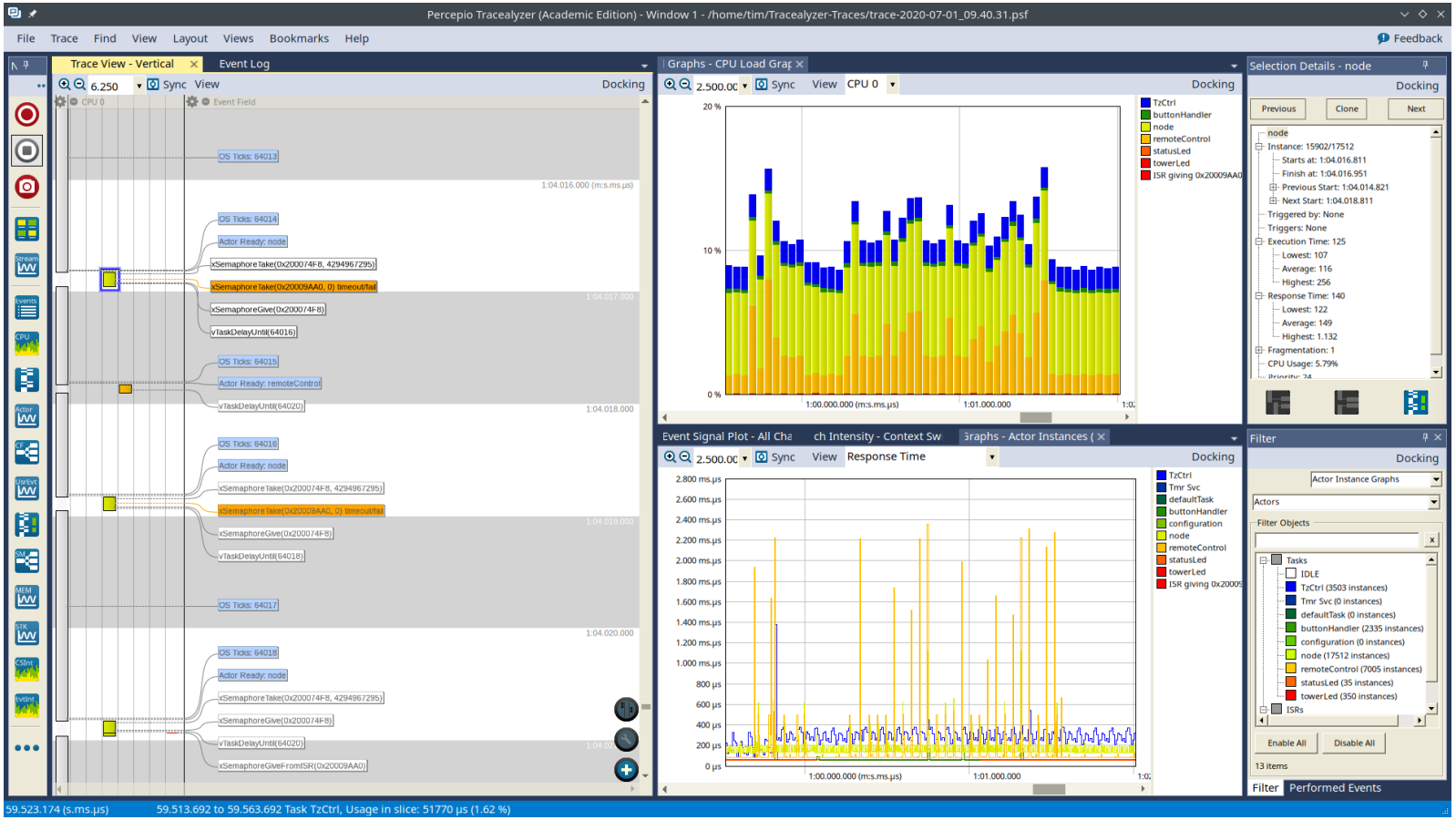


Abbildung 4.10: Durch Tracealyzer visualisierte Laufzeitdaten am Beispiel des Interface Boards. Auf der linken Seite sind die einzelnen Task-Instanzen inklusive der aufgerufenen FreeRTOS-Funktionen in einem Wasserfalldiagramm zu sehen, rechts die CPU-Nutzung und Antwortzeiten.

5 Evaluierung

Um unsere Zielsetzung zu validieren, werden die wichtigsten Komponenten des Backbones evaluiert. Dies geschieht zum einen durch das Testen der Funktionalität der unterbrechungsfreien Spannungsversorgung unter bestimmten Lastbedingungen und zum anderen durch die Evaluierung des CAN-Busses hinsichtlich seiner Robustheit und Fehlerrate.

5.1 Unterbrechungsfreie Spannungsversorgung

Um die Funktionalität der unterbrechungsfreien Spannungsversorgung zu testen, wird ein Wechsel der Spannungsquelle im laufenden Betrieb unter Last durchgeführt und die Spannungsverläufe mithilfe eines Oszilloskops dargestellt.

Zum Anfang wird das System mit einer Spannung von 13,6 V, die von einem Labornetzteil bereitgestellt wird, versorgt und an eine Last angeschlossen. Die Gesamtlast des Systems wird mithilfe von Lastwiderständen als ohmsche Last modelliert. Diese lässt sich in drei Lasten aufteilen: Auf der 5V-Schiene ist eine Last von 2,5 A angeschlossen, während die Last auf der 12V-Schiene 1,2 A beträgt. Um den Verbrauch eines NUCs zu simulieren, wird dort ebenfalls eine ohmsche, konstante Last angeschlossen, welche 3 A verbraucht. Dieser Wert ist aus dem realen Betrieb des NUCs entnommen worden, während dieses mit einem typischen oTToCar-Softwaresetup für Bildverarbeitung und Logik des autonomen Autos lief. Die erforderlichen Stromwerte für 5 V und 12 V wurden geschätzt, aber es wird davon ausgegangen, dass diese Werte im Normalfall auch in der Praxis so auftreten werden. Insgesamt ergibt das eine Gesamtlast von ca. 68 W. Bei der eingangs erwähnten Spannung von 13,6 V wird diese Spannungsquelle mit einem Strom von 5 A belastet.

Nun wird eine zweite Spannungsquelle, ebenfalls ein Labornetzteil, an das System angeschlossen. Ihre Spannung ist mit 15,5 V etwas höher als die der ersten

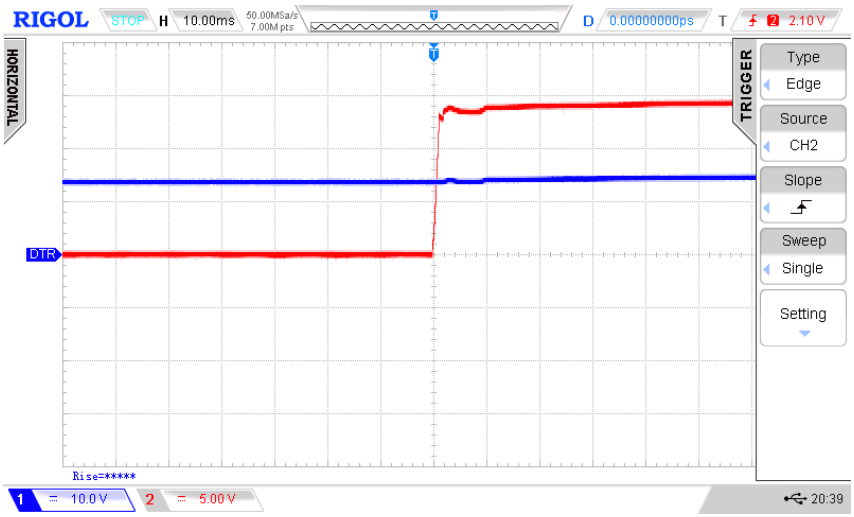


Abbildung 5.1: Zuschalten der zweiten Spannungsquelle (rot) und leichte Anpassung der Systemspannung (blau) an diese

Spannungsquelle. Da an der Stelle des NUCs eine ohmsche Last eingesetzt wurde, steigt der Strom bedingt durch das ohmsche Gesetz auch geringfügig an. Die Lasten auf den 5V- und 12V-Schienen bleiben unverändert. Bedingt durch ihre höhere Spannung übernimmt die zweite Spannungsquelle beim Anschließen umgehend und vollständig die Stromversorgung des Systems. Wie man in der Abb. 5.1 erkennen kann, braucht das Netzteil aufgrund des sprunghaften Lastwechsels von 0 A auf insgesamt 5 A mehrere Millisekunden Zeit, um ihre Spannung zu stabilisieren. Die Systemspannung, also die Spannung hinter dem Power-Multiplexer, erhöht sich auf die Spannung des Netzteils. Es werden dabei keine Spannungseinbrüche bzw. -spitzen produziert.

Interessanter ist aber das Verhalten des Systems beim Entfernen der zweiten Spannungsquelle. Dieser Wechsel muss schnell und ohne Unterbrechung erfolgen. Dieses Szenario wird in der Abb. 5.2 dargestellt.

Beim Abschalten der zweiten Spannungsquelle übernimmt logischerweise die erste Spannungsquelle wieder die Stromversorgung des Systems. Die Systemspannung bricht direkt nach dem Abschalten der zweiten Spannungsquelle für einige Millisekunden um weniger als ein Volt ein, allerdings lässt sich dieses Verhalten auf das Nachregeln/Stabilisieren des Labornetzteils zurückführen und stellt keinerlei Beeinträchtigung für das Gesamtsystem dar. Da die kurzen Umschaltzeiten im Bereich von einigen hundert Nanosekunden durch Pufferkondensatoren überbrückt werden, sind diese nicht zu erkennen. Die Limitie-

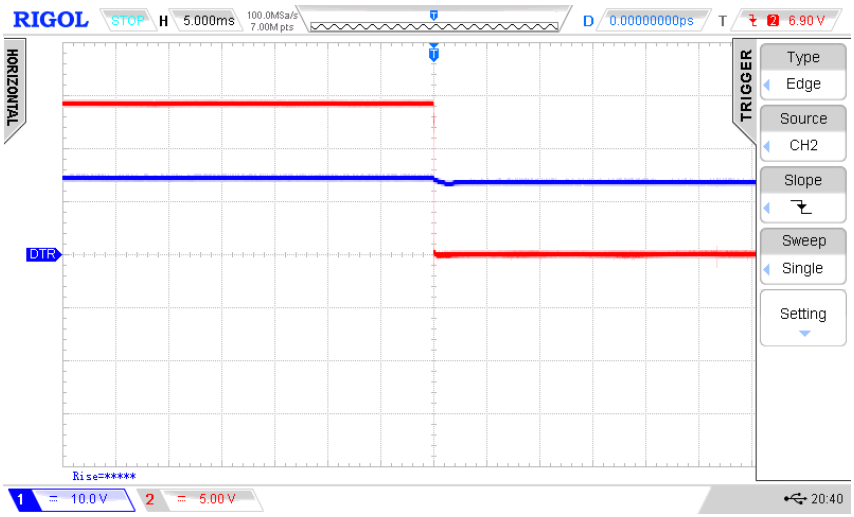


Abbildung 5.2: Abschalten der zweiten Spannungsquelle (rot) – Anpassung der Systemspannung (blau) auf die erste Spannungsquelle

rung des Einschaltstromes und das Abschalten bei einem Überstrom findet ohne Probleme statt.

Nach längeren Tests traten mehrere Probleme in der Ansteuerung der MosFETs auf, die dazu führten, dass diese am Ende irreparabel elektrisch beschädigt wurden, denn diese werden sehr heiß und letztendlich dauerhaft leitend. Auch nach dem Austausch der Komponenten wie MosFETs und ICs treten die Probleme nach einer gewissen Zeit erneut auf und machen ein solides, robustes Arbeiten unmöglich. Trotz intensiver Suche ließ sich die Ursache der Probleme leider nicht feststellen. Auch Lösungsansätze wie das Erhöhen der Gatekapazität des MosFETs, um das Gate gegen ein mögliches Oszillieren der Gatespannung zu schützen, brachten keine Verbesserungen. Aus diesem Grund ist für das *Power Distribution Board* eine neue Revision mit überarbeiteter Schaltung und veränderter Komponentenwahl geplant.

5.2 Multi-Master-Bus

Die Gründe für den Wechsel des Busses auf Basis von RS485 auf CAN liegen zum einen in dem Einsatz eines von der Industrie gut unterstützten multimaster-fähigen Bussystems, zum anderen aber auch in der Steigerung der mög-

lichen Übertragungsgeschwindigkeit bei hoher Robustheit durch Priorisierung und Fehlererkennung der Nachrichten.

Um die Robustheit des CAN-Busses zu evaluieren, wird die Anzahl an Übertragungsfehlern gezählt. Dazu senden fast alle Busteilnehmer Nachrichten mit einer festen Frequenz. Aufgrund der Bustopologie und der Funktionsweise von CAN empfangen alle Teilnehmer alle gesendeten Nachrichten, auch wenn diese möglicherweise nicht weiterverarbeitet werden. Deshalb wird überprüft, ob eine Nachricht bei allen Busteilnehmern korrekt ankommt. Jeder Busteilnehmer besitzt zwei interne Fehlerzähler. Den ersten stellt der CAN-Controller des MCU bereit und zählt die Fehler, die bereits in der Bitübertragungsschicht oder der Sicherungsschicht des CAN-Busses auftreten. Der zweite Fehlerzähler wird von UAVCAN bereitgestellt und registriert Fehler auf Protokollebene.

Für die Evaluierung wird ein Testaufbau errichtet, welcher einen Teil des späteren Autos widerspiegelt. Der Testaufbau ist bzgl. der Verbindungen zwischen den Platinen nicht optimal, da die drei VESCs über Stichleitungen mit jeweils ca. 5 cm Länge angeschlossen sind. Der Grund für die Stichleitungen liegt in der Art der Verkabelung, da die VESCs in ihrer jetzigen Revision nur eine CAN-Buchse besitzen. Um dennoch weitere Platinen hinter den VESCs zu verbinden, wird das Kabel in der Mitte aufgesplittet (Y-Form). Die Tabelle 5.1 zeigt die im Testaufbau genutzten Busteilnehmer mit der zugehörigen Nachricht und ihrer Frequenz. Zusätzlich zu den in der Tabelle 5.1 genannten Busteilnehmern sind zwei weitere, passive Busteilnehmer angeschlossen. Diese senden nicht aktiv Nachrichten auf den Bus, sondern empfangen lediglich die der anderen Teilnehmer. Deren Fehlerzähler wird ebenfalls mit in Betracht bezogen.

Busteilnehmer	Nachricht	Größe	Frequenz
Power Distribution Board	`SystemPowerStatus`	20 Byte	100 Hz
Interface Board	`RemoteControlStatus`	7 Byte	20 Hz
Servo Board	`PositionAndTorque`	4 Byte	250 Hz
VESC1	`VitalStatus`	13 Byte	1000 Hz
VESC2	`VitalStatus`	13 Byte	1000 Hz
VESC3	`VitalStatus`	13 Byte	1000 Hz

Tabelle 5.1: Nachrichten auf dem CAN-Bus während des Testaufbaus

Diese Zusammenstellung ergibt bei einem CAN-Bus, welcher mit 1 MBaud betrieben wird, eine Buslast von 91 %. Dieser Wert für die Buslast wurde mit dem Linux-Tool `canbusload`[1] gemessen. Dazu ist das Bussystem über die CAN-USB-Bridge mit einem Linux-System verbunden und mithilfe des UAVCAN-GUI-Tools können die Fehlerzähler der einzelnen Busteilnehmer und die Anzahl an gesendeten bzw. empfangende Nachrichten ausgelesen werden. Unter dieser Buslast lief der Testaufbau mehr als zwei Stunden lang. In diesem Zeitraum wurden auf dem Bus mehr als 54 Millionen Nachrichten gesendet. Schließlich wurden die Fehlerzähler aller acht angeschlossenen Busteilnehmern ausgewertet und es sind keine Fehler aufgetreten. Angesichts der hohen Buslast in Kombination mit dem Vorhandensein von Stichleitungen ist dies positiv zu bewerten.

Um den Bus unter besonders widrigen Bedingungen zu testen, wurden von außen Spannungsspitzen auf die CAN-Leitungen induziert. Um solche Spitzen zu generieren, reicht es aus, in der Nähe des Aufbaus ein Netzteil mit hoher Kapazität an eine 230V-Steckdose anzuschließen. Bedingt durch die Kapazitäten des Netzteils entsteht in diesem ein hoher Einschaltstrom, welcher als elektromagnetischer Impuls an die Umgebung abgegeben wird. Die dabei induzierten Spannungsspitzen besitzen eine Dauer von wenigen Mikrosekunden und erreichen dabei eine Amplitude von bis zu ±800 mV. Diese werden in der Abb. 5.3 gezeigt. Nun wird überprüft, ob diese Spitzen eine Auswirkung auf

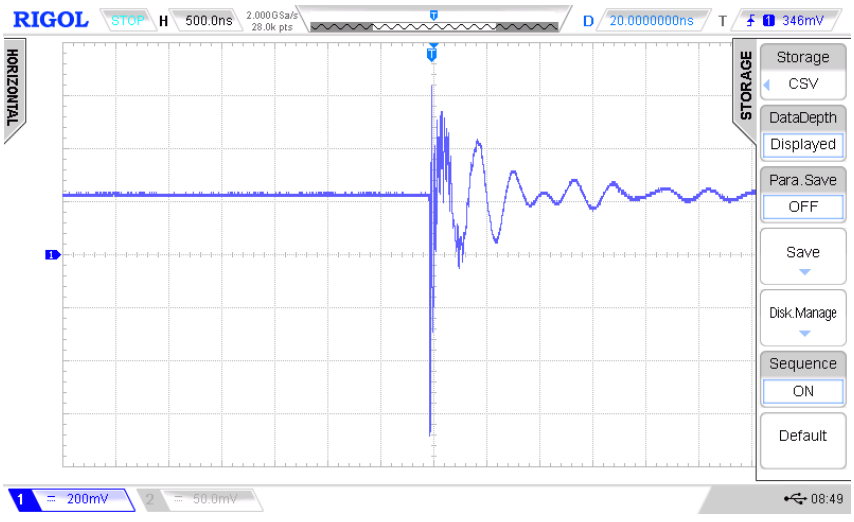
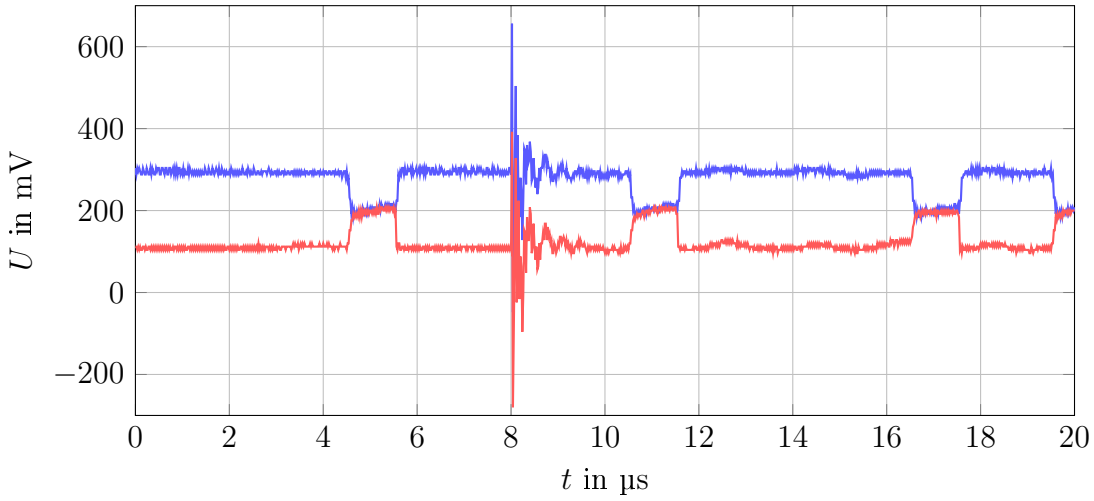


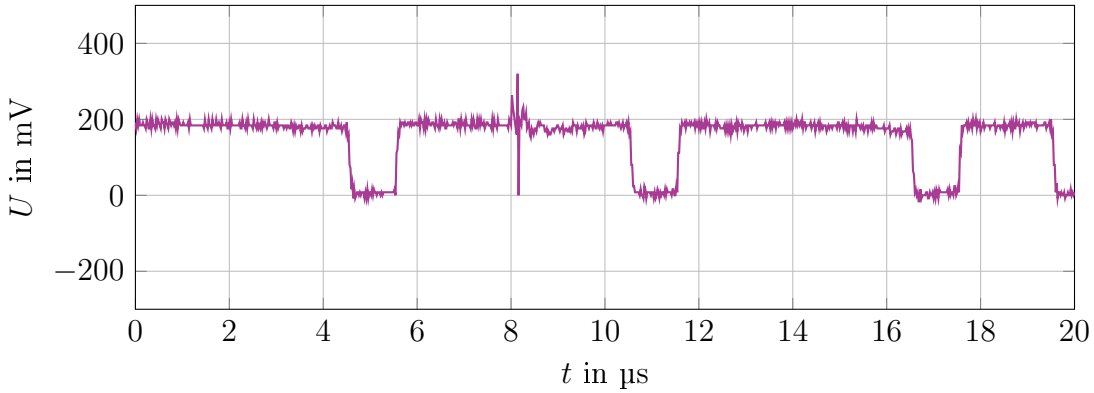
Abbildung 5.3: Spannungsspitze auf einer Leitung

[1] https://github.com/linux-can/can-utils

das Bussystem besitzen. Dazu wurden die Spitzen im laufenden Betrieb provoziert und schließlich die Fehlerzähler aller Busteilnehmer ausgewertet. Dabei waren keine Auswirkungen auf die Fehlerzähler der Busteilnehmer ersichtlich. Der Grund für das robuste Verhalten liegt in den Stärken der differentiellen Leitungen. Die Abb. 5.4 zeigt zum einen die induzierten Spannungsspitzen auf den CAN-Leitungen, zum anderen die Differenzbildung dieser. Es wird schnell erkennbar, dass die Differenzbildung die Spitzen nahezu ausmerzt. Die Restspitzen haben keine weitere Auswirkungen, da der CAN-Treiber eine interne „Slope-Control“ besitzt, welche Spitzen im Nanosekundenbereich filtert.



(a) Induzierte Spannungsspitzen auf beide CAN-Leitungen



(b) Differenzbildung der CAN-Leitungen aus dem Diagramm (a)

Abbildung 5.4: Gegentaktunterdrückung

Es wurden zudem weitere Szenarien getestet, wie das Eliminieren des zweiten Terminierungswiderstands und das Trennen einzelner CAN-Leitungen. Da das Bussystem auf einem 1:10 Auto in seiner Gesamtlänge recht kurz ist, genügt hier eine Busterminierung. Das Trennen einzelner CAN-Leitungen, sprich das Herbeiführen eines Ein-Draht-Betriebes, ist nicht für CAN-High-Speed spezifiziert und führt an der Stelle der Trennung zu einer Unterbrechung der Datenübertragung, d. h. das Bussystem wird in zwei unabhängig agierende Teilsysteme geteilt, welche weiterhin arbeitsfähig sind, vorausgesetzt beide besitzen ihre eigene Busterminierung.

6 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde ein Backbone für die nächste Generation des oTToCars erschaffen. Dabei wurden vier neue Platinen designt, welche über den neuen multi-master-fähigen CAN-Bus miteinander kommunizieren. Für die auf den Platinen verbauten Mikrocontroller wurden die Firmwares von Grund auf neu geschrieben. Sie wurden mit einem Echtzeitbetriebsystem ausgestattet, welches ein bequemes Arbeiten mit mehreren Tasks erst ermöglicht. Es besteht die Möglichkeit, über das NUC auf einzelne CAN-Busteilnehmer zuzugreifen und diese zu konfigurieren.

Bezüglich des Themas Power-Multiplexer musste ein kleiner Rückschlag verzeichnet werden. Nichtsdestotrotz übernimmt das Power Distribution Board aktuell die Aufgabe der Spannungsversorgung ohne Probleme und wird in seiner nächsten Revision komplett überarbeitet, sodass das Adjektiv „unterbrechungsfrei“ zusätzlich angehängt werden kann. Zudem erhält das Board einen EEPROM, damit Konfigurationen, die über das UAVCAN GUI Tool vorgenommen werden können, auch nach Wegfall der Spannungsversorgung abrufbar sind. Die restlichen Platinen besitzen ihre Grundfunktionalitäten und werden softwareseitig weiter ausgebaut bzw. angepasst, sobald das Auto im Rahmen einer anderen Bachelorarbeit fahrbar wird. Die RF-Hardware wurde bereits erfolgreich auf ihre Grundfunktionalität getestet.

Neben dem Power Distribution Board erfahren die anderen Platinen zukünftig ebenfalls kleinere Überarbeitungen. So soll die CAN-USB-Bridge über das NUC statt über die 5V-Leitung des CAN-Busses versorgt werden. Damit wird sichergestellt, dass das NUC weiterhin CAN-Nachrichten empfangen kann, sobald die 5V-Versorgung des CAN-Busses ausgefallen ist. Des Weiteren bekommen die Platinen mehrere Testpoints an signifikanten Stellen, um das Debuggen auf elektrischer Ebene, vor allem das einer bestimmten Peripherie wie I^2C oder SPI, zu vereinfachen.

Das Zusammenspiel der Hard- und Software schafft eine solide Basis für das zukünftige Arbeiten mit dem Auto.

Literatur

[Alb20] Michael Albrecht. *Entwurf und Evaluierung eines Einzelradantriebes für autonom fahrende 1:10 Automodelle*. Bachelorarbeit. 2020.

[Bie08] Klaus Bierschenk. "Botschaften, Nachrichten oder Frames". In: *Kraftfahrzeugmechatronik*. Troisdorf: Bildungsverlag EINS, 2008, S. 22–30. ISBN: 978-3-427-04858-9.

[Elea] ElectronicsTutorials. *The Schottky Diode*. URL: https://www.electronics-tutorials.ws/diode/schottky-diode.html (besucht am 24. 07. 2020).

[Eleb] ElectronicsTutorials. *The Signal Diode*. URL: https://www.electronics-tutorials.ws/diode/diode_4.html (besucht am 24. 03. 2020).

[Elec] Elektronik Kompendium. *Low-ESR-Elektrolytkondensatoren*. URL: https://www.elektronik-kompendium.de/sites/bau/0810091.htm (besucht am 25. 03. 2020).

[Gay18] Warren Gay. "FreeRTOS". In: *Beginning STM32: Developing with FreeRTOS, libopencm3 and GCC*. Berkeley, CA: Apress, 2018, S. 59–72. ISBN: 978-1-4842-3624-6. DOI: 10.1007/978-1-4842-3624-6_5. URL: https://doi.org/10.1007/978-1-4842-3624-6_5.

[Hün19] Felix Hüning. "Echtzeitbetriebssystem". In: *Embedded Systems für IoT*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2019, S. 89–111. ISBN: 978-3-662-57901-5. DOI: 10.1007/978-3-662-57901-5_8. URL: https://doi.org/10.1007/978-3-662-57901-5_8.

[KUN] KUNBUS GmbH. *RS-485 Schnittstelle*. URL: https://www.kunbus.de/rs-485.html (besucht am 17. 05. 2020).

[KWa] Pavel Kirienko und Hamish Willee. *UAVCAN v0: Basic concepts*. URL: https://legacy.uavcan.org/Specification/2._Basic_concepts/ (besucht am 25. 05. 2020).

[KWb] Pavel Kirienko und Hamish Willee. *UAVCAN v0: Introduction*. URL: https://legacy.uavcan.org/Specification/1._Introduction/ (besucht am 25. 05. 2020).

[KWK] Pavel Kirienko, Hamish Willee und Kjetil Kjeka. *UAVCAN v0: Data structure description language*. URL: https://legacy.uavcan.org/Specification/3._Data_structure_description_language/ (besucht am 25. 05. 2020).

[Max03] Maxim Integrated. *Understanding, Using, and Selecting Hot-Swap Controllers*. AN2736. Dez. 2003.

[MEM] ME-Meßsysteme GmbH. *CAN Bus Grundlagen*. URL: https://www.me-systeme.de/de/technik-zuerst/elektronik/can-bus-grundlagen (besucht am 29. 06. 2020).

[Sel19] Karikalan Selvaraj. *Basics of Ideal Diodes*. SLVAE57. Texas Instruments. Mai 2019.

[Tec] Technische Universität Braunschweig. *Carolo-Cup*. URL: https://wiki.ifr.ing.tu-bs.de/carolocup/carolo-cup (besucht am 12. 05. 2020).

[Tec19a] Technische Universität Braunschweig. *Carolo-Basic-Cup Regulations 2020*. Aug. 2019. URL: https://wiki.ifr.ing.tu-bs.de/carolocup/system/files/Basic-Cup%20Regulations.pdf (besucht am 12. 05. 2020).

[Tec19b] Technische Universität Braunschweig. *Carolo-Master-Cup Regulations 2020*. Aug. 2019. URL: https://wiki.ifr.ing.tu-bs.de/carolocup/system/files/Master-Cup%20Regulations.pdf (besucht am 12. 05. 2020).

[Tri18] Alex Triano. *Basics of Power MUX*. SLVAE51. Texas Instruments Incorporated. Nov. 2018.

[Wan17] K. C. Wang. "Models of Embedded Systems". In: *Embedded and Real-Time Operating Systems*. Cham: Springer International Publishing, 2017, S. 95–111. ISBN: 978-3-319-51517-5. DOI: 10.1007/978-3-319-51517-5_4. URL: https://doi.org/10.1007/978-3-319-51517-5_4.

[Wik19] Wikipedia Mitwirkender. *Balun*. 2019. URL: https://de.wikipedia.org/wiki/Balun (besucht am 10. 06. 2020).

Datenblätter

[Atm15] Atmel Corporation. *AT86RF212B - Complete Datasheet*. 42002E. Datenblatt. Feb. 2015.

[Int20] Intel. *NUC8i3BE/NUC8i5BE/NUC8i7BE*. K15389-007. Technische Spezifikation. Feb. 2020.

[STM18] STMicroelectronics. *STM32™ 32-bit MCU family*. BRSTM320218. Portofolio. Feb. 2018.

[Tex15a] Texas Instruments Incorporated. *TPS2474x 2.5-V to 18-V High Performance Hot Swap and ORing Controller*. SLVSCV6A. Datenblatt. Jan. 2015.

[Tex15b] Texas Instruments Incorporated. *TPS2474x Design Calculator*. SLVRBD9B. 2015.

[Tex18] Texas Instruments Incorporated. *SN65HVD23x 3.3-V CAN Bus Transceivers*. SLOS346O. Datenblatt. Apr. 2018.

Selbstständigkeitserklärung

Hiermit erklären wir, dass wir die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet haben.

Tim Wiesner und Maximilian Grau Magdeburg, 27. Juli 2020