Entwicklung eines Datalogging-Systems zur zentralen Datenerfassung und Live-Darstellung für ein Formula Student Fahrzeug

Bachelorarbeit für den Studiengang Fahrzeugelektronik und Elektromobilität

vorgelegt von: Tarik Yagli

Matrikelnummer: 10013523

Betreut durch: Prof. Dr. Klaus Thelen(HRW)

Prof. Dr. Katja Rösler(HRW)

Abgabedatum: 22.06.2025

Bearbeitungszeitraum: 11 Wochen

Erklärung

Hiermit erkläre ich, dass die vorliegende Arbeit mit dem Titel „Entwicklung eines Datalogging-Systems zur zentralen Datenerfassung und Live-Darstellung für ein Formula Student Fahrzeug“ selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt, keine anderen als die angegeben Hilfsmittel benutzt und wörtliche sowie sinngemäße Zitate als solche gekennzeichnet habe.

Inhaltsverzeichnis

Erklärung ii
Vorwort iv
1 Einleitung 1
1.1 Bedeutung und Einsatzbereiche von Dataloggern 1
1.2 Technologischer Wandel und Entwicklungsgeschichte 1
1.3 Aufgabenstellung und Zielsetzung 2
1.4 Aufbau der Arbeit 2
2 Stand der Technik 3
2.1 Komponenten eines modernen Dataloggers 3
2.1.1 Sensoren 3
2.1.2 Mikrocontroller und Prozessoren 4
2.1.3 Speicher 4
2.1.4 Kommunikationseinheit 5
2.2 CAN-Bus im Motorsport und in der Industrie 5
3 Methoden und Lösungsansätze 10
3.1 Auswahl der Hardwareplattform: Raspberry Pi 10
3.2 Anbindung an den CAN-Bus: MCP2515 12
3.3 Grundkonfiguration des Raspberry Pi und softwareseitige Einbindung des MCP2515 18
4 Ergebnisse 19
4.1 Funktionstest der CAN-Kommunikation mit variablen Eingangssignalen 20
4.2 Drahtlose Übertragung über LoRa 21
4.3 Drahtlose Übertragung der Can-Bus Daten 26
4.4 PCB-Design für das Fahrzeug CAN-Modul 27
5 Diskussion der Ergebnisse 31
5.1 Raspberry Pi als Mikrocontroller im CAN-Bus-System 31
5.2 Vorteile von LoRa gegenüber WLAN und Mobilfunk (SIM) 31
5.3 Gesamtsystem im Formula-Student-Kontext 32
5.4 Zusammenfassung 32
5.5 Ausblick 35
Abkürzungsverzeichnis vi
Tabellenverzeichnis vii
Abbildungsverzeichnis viii
Literaturverzeichnis ix
Anhang xi

Vorwort

Die Formula Student ist ein Internationaler Wettbewerb bei dem Universitätsstudenten einen Prototyp-Rennwagen entwickeln und Fahren.[1] Der Wettbewerb wird von der Society of Automotive Engineers (SAE) organisiert, welche ebenfalls eine zentrale Rolle in Standards und Richtlinien für die Automobil-, Luftfahrt- und andere Mobilitätsindustrie spielen. [2]

Die Formula Student findet seit 1978 jährlich mit mehr als 120 teilnehmenden Teams aus aller Welt statt. Die Events zu den Rennen werden derzeit in Ländern wie Deutschland, Schweiz, Tschechien, Ungarn, Italien, Spanien, Frankreich und Polen ausgetragen. [1] Neben diesen Europäischen Ländern treten die Teams auch in Japan, Brasilien und Australien gegeneinander an. Um sich für die Teilnahme an den verschiedenen Formula-Student-Events zu qualifizieren, müssen die Teams zunächst ihr technisches Fachwissen unter Beweis stellen. Hierzu werden Online-Wissensquizze eingesetzt, bei denen die Studierenden Fragen zu den Wettbewerbsregeln, technischem Hintergrund und ingenieurwissenschaftlichen Grundlagen beantworten müssen. [3]

Diese Quizze dienen nicht nur dazu, sicherzustellen, dass die Teams die Anforderungen und Regularien des Wettbewerbs verstehen, sondern auch eine tiefere Auseinandersetzung mit den technischen und organisatorischen Herausforderungen des Projekts fördern. Erst nach erfolgreichem Bestehen dieser Prüfungen erhalten die Teams die Möglichkeit, mit ihren selbst entwickelten Rennfahrzeugen an den jeweiligen Events teilzunehmen. In der Formula Student Germany werden beispielsweise maximal 65 Plätze für die Teilnahme am Event vergeben. [3]

Formula Student Germany

Die Formula Student Germany wird seit 2006 ausgetragen und hat sich ständig weiterentwickelt. In den ersten Jahren nahmen ausschließlich Fahrzeuge mit Verbrennungsmotoren teil. Mit dem Fortschritt der Elektromobilität wurde 2009 die Kategorie für elektrisch angetriebene Fahrzeuge eingeführt. Ein weiterer Meilenstein folgte 2017 mit der Einführung der Formula Student Driverless, indem autonome Rennfahrzeuge gegeneinander antreten. [4]

Motivation

Der Wettbewerb soll Ingenieurstudierenden praxisnahe Erfahrungen vermitteln, insbesondere in den Bereichen Teamarbeit, Zeitmanagement, Projektplanung, sowie in der Konstruktion, Fertigung und den wirtschaftlichen Aspekten der Automobil Entwicklung. Die Wettbewerbssprache ist Englisch, was die internationale Zusammenarbeit fördert. Das Ziel der Formula

Student Germany ist es, das außeruniversitäre Engagement der Studierenden sichtbarer zu machen – insbesondere für deutsche Unternehmen und Hochschulen. Durch den starken Wettbewerb sind die Studierenden gezwungen, auf höchstem professionellem Niveau zu arbeiten und dabei Werkzeuge und Methoden einzusetzen, wie sie auch in der Industrie üblich sind.

Zudem eröffnet der Wettbewerb Möglichkeiten für Unternehmen, mit den Studierenden in Kontakt zu treten, sei es über Sponsoring, Preise oder durch die Entsendung von Jurymitgliedern. Dies stärkt die Verbindungen zwischen jungen Ingenieuren und der Industrie auf direkte und nachhaltige Weise.

1 Einleitung

1.1 Bedeutung und Einsatzbereiche von Dataloggern

Datalogger sind elektronische Systeme zur kontinuierlichen Erfassung und Speicherung physikalischer Messgrößen über definierte Zeiträume hinweg. Sie bilden eine zentrale Grundlage für moderne Formen der Überwachung, Analyse und Prozessoptimierung. In zahlreichen Anwendungsfeldern wie der Industrie, dem Umweltschutz, der Medizintechnik, der wissenschaftlichen Forschung sowie im Transportwesen ermöglichen sie die Aufzeichnung relevanter Betriebs- und Umweltparameter. [5]

Typische Messgrößen, die durch Datalogger erfasst werden, umfassen unter anderem Temperatur, Luftfeuchtigkeit, Druck, Spannung, Strömungsgeschwindigkeit oder auch mechanische Bewegungen. [5]

Durch den automatisierten Betrieb werden Daten zuverlässig und ohne Benutzerinteraktion erfasst. Diese können anschließend zur Qualitätssicherung, Fehleranalyse oder für langfristige Studien ausgewertet werden. [5]

1.2 Technologischer Wandel und Entwicklungsgeschichte

Datalogger gibt es schon sehr lange. Ursprünglich waren es einfache Geräte, die physikalische Größen wie Temperatur mechanisch oder elektromechanisch aufzeichneten. Diese Technik fand vor allem in Industrie und Forschung Anwendung. Nach dem Zweiten Weltkrieg sorgten neue elektronische Bauteile wie Transistoren dafür, dass Datalogger immer kleiner und leistungsfähiger wurden. In den 1960er Jahren entstanden die ersten Geräte, die Messwerte digital speichern konnten. Dabei kamen beispielsweise Kassetten als Speichermedium zum Einsatz. [6]

In den 1980er Jahren verbesserten sich die Möglichkeiten durch den Einsatz von Mikroprozessoren und neuen Speichertechnologien deutlich. Datalogger wurden dadurch unabhängiger von einer dauerhaften Stromversorgung und konnten komplexere Daten verarbeiten.

Heute sind viele Datalogger mehr als nur Geräte zur Aufzeichnung. Sie können Messdaten in Echtzeit analysieren, mit anderen Systemen kommunizieren und sogar Prozesse steuern. In der Industrie werden sie zum Beispiel bei der Überwachung von Anlagen oder der automatischen Steuerung von Maschinen eingesetzt. [6]

1.3 Aufgabenstellung und Zielsetzung

Um den technischen Zustand des Rennwagens bewerten zu können, müssen Daten aus Testfahrten erfasst werden. Sie bilden die Grundlage für detaillierte Analysen des Fahrverhaltens und möglicher Schwachstellen am Fahrzeug.

Um dieses Ziel zu erreichen, sind folgende Teilaufgaben zu bearbeiten:

- **Entwicklung und Erprobung einer Datenerfassungsplatine, die Sensordaten vom CAN-BUS lesen kann.**
- **Integration einer stabilen drahtlosen Schnittstelle zur Datenübertragung an eine Ground Station.**
- **Design und Implementierung eines Interfaces, das die gesammelten Daten in Echtzeit darstellt.**
- **Die Erweiterung von neuen Sensordaten muss möglich sein.**

Das Datalogging-System soll eine Erweiterung des bestehenden elektrischen Systems sein und nutzt den bestehenden CAN-BUS zur Auswertung von Sensoren. Durch Erweiterung des CAN-BUS sollen zukünftig Sensoren an beliebiger Stelle angebracht werden und über das in dieser Arbeit auszuarbeitende Datalogging-System ausgewertet werden können.

1.4 Aufbau der Arbeit

Die vorliegende Arbeit ist in vier Hauptkapitel gegliedert:

Zunächst wird ein Überblick über den aktuellen Stand der Technik im Bereich der Datenerfassungssysteme in den Fahrzeugen geschaffen. Es werden bestehende Lösungen zur Sensordatenerfassung über den CAN-BUS erläutert, einschließlich kommerzieller Datalogger, verwendeter Mikrocontroller, drahtloser Übertragungstechnologien und Visualisierungsmethoden.

Anschließend werden die Ansätze und Methoden vorgestellt, wie die Entwicklung eines Prototyps einer eigenen Datenerfassungsplatine zur Integration in das bestehende elektrische System des Rennfahrzeugs erfolgen kann. Dabei werden sowohl der Aufbau der Hardware als auch die eingesetzten Schaltpläne beschrieben. Zudem wird der verwendete Mikrocontroller, die Programmierumgebung und die CAN-Kommunikation erläutert.

Im Kapitel Ergebnisse wird die entwickelte Datalogging-Platine als Prototyp vorgestellt und getestet. Der Fokus liegt auf den Funktionen des Systems zur Erfassung, Zwischenspeiche-

rung und drahtlosen Übertragung der Sensordaten. Außerdem wird das PCB-Design präsentiert, das den notwendigen Spannungsregler und die CAN-Bus-Schnittstelle enthält.

Abschließend werden die Erkenntnisse aus der Arbeit zusammengefasst. Es wird analysiert, wie effektiv das entwickelte Datalogging-System arbeitet und welche Potenziale es hinsichtlich Erweiterbarkeit, Stabilität und Echtzeitfähigkeit bietet.

2 Stand der Technik

In diesem Kapitel wird der aktuelle Stand der Technik im Bereich der Sensordatenerfassung über den CAN-Bus vorgestellt. Es werden verschiedene Datalogger und die eingesetzten Mikrocontroller vorgestellt. Außerdem wird auf drahtlose Übertragungstechnologien und Visualisierungsmethoden eingegangen, die eine Echtzeitüberwachung und Analyse der Daten ermöglichen. Der Fokus liegt auf den technologischen Lösungen, die derzeit in der Fahrzeugtechnik und im Motorsport eingesetzt werden, um eine präzise und zuverlässige Erfassung der Fahrzeugdaten zu gewährleisten.

2.1 Komponenten eines modernen Dataloggers

Ein Datalogger setzt sich aus mehreren wesentlichen Komponenten zusammen, die miteinander harmonieren, um die gewünschten Messdaten zu erfassen, zu speichern und zu übertragen. Im Nachfolgenden werden die wichtigsten Baugruppen aufgezählt und beschrieben, die zu Datalogger zählen. [7]

2.1.1 Sensoren

Sensoren sind das Herzstück eines jeden Dataloggers, da sie die physikalischen Größen messen, die für eine bestimmte Anwendung relevant sind. Die Auswahl des Sensors hängt von den zu überwachenden Parametern ab. Die häufigsten Sensorarten werden nachfolgende aufgezählt.

Temperatursensoren: hierzu zählen Thermoelemente, Widerstandstemperaturdetektoren und Thermistoren. Sie messen die Temperatur und liefern Daten, die in vielen industriellen und wissenschaftlichen Anwendungen von entscheidender Bedeutung sind. [7]

Luftfeuchtigkeitssensoren: diese Sensoren messen die relative Luftfeuchtigkeit und kommen häufig in Klimaanlagen, Gewächshäusern und Lagerhäusern zum Einsatz. [6]

Drucksensoren: sie messen den Druck in Flüssigkeiten oder Gasen und werden in Bereichen wie der Hydraulik, der Pneumatik und in der Umweltüberwachung eingesetzt. [8]

Bewegungssensoren: diese erfassen Vibrationen oder Bewegungen und werden insbesondere in der Fertigungsindustrie verwendet, um Anomalien in Maschinen oder Prozessen zu erkennen.[6]

Licht- und Strahlungssensoren: sie messen Lichtintensität oder Strahlungsniveaus und kommen etwa in der Solarenergieüberwachung oder in meteorologischen Anwendungen zum Einsatz. [8]

Die Wahl der Sensoren ist entscheidend, da sie direkt die Genauigkeit und Zuverlässigkeit der Messungen beeinflussen.

2.1.2 Mikrocontroller und Prozessoren

Der Mikrocontroller oder Prozessor eines Dataloggers übernimmt die Aufgabe, die Messwerte der Sensoren zu verarbeiten und zu speichern. Mikrocontroller sind heutzutage sehr leistungsfähig und energieeffizient. [9] Damit sie funktionieren, ist zusätzlich Stromversorgung erforderlich. Diese kann je nach Anwendung über Batterien, Akkus, Solarpanels oder Netzteile realisiert werden. Sie führen auch häufig grundlegende Analysen durch, wie z.B. die Berechnung von Mittelwerten oder die Filterung von Rauschen in den Rohdaten. In einigen fortgeschrittenen Dataloggern werden auch komplexe Algorithmen ausgeführt, die eine Vorhersage von Ereignissen oder eine Analyse der Messdaten in Echtzeit ermöglichen.

2.1.3 Speicher

Der Speicher eines Dataloggers dient der Speicherung der aufgezeichneten Messwerte.[9] Es gibt verschiedene Arten von Speichermedien, die je nach Anwendung und Datalogger-Modell zum Einsatz kommen:

Flash-Speicher: sehr verbreitet aufgrund seiner Geschwindigkeit und Robustheit. Der Flash-Speicher ist nichtflüchtig, was bedeutet, dass die Daten auch bei einem Stromausfall erhalten bleiben.[7]

SD-Karten: Diese bieten eine größere Speicherkapazität und sind in vielen Dataloggern verbaut. Sie sind insbesondere dann von Vorteil, wenn große Mengen an Daten über längere Zeiträume gesammelt werden müssen.[7]

Cloud-Speicher: Einige moderne Datalogger ermöglichen die direkte Übertragung von Daten in die Cloud, was ebenfalls eine sofortige Analyse biete.[11]

2.1.4 Kommunikationseinheit

Die Kommunikationseinheit eines Dataloggers sorgt dafür, dass die gesammelten Daten an ein übergeordnetes System, wie einen Computer, Server oder Cloud-Dienst, übertragen werden. Dies ist in Anwendungen wichtig, bei denen eine kontinuierliche Überwachung erforderlich ist. Zu den gängigen Kommunikationstechnologien gehören:[11]

WLAN: weit verbreitet und bietet eine einfache Integration in bestehende Netzwerke.[7]

Bluetooth: für Kurzstreckenkommunikation mit Smartphones oder Tablets geeignet.[7]

Mobilfunk: durch die Nutzung von 4G- oder 5G-Netzen können Datalogger in abgelegene Gebiete übertragen werden, in denen keine festen Internetverbindungen bestehen.

Internet der Dinge (IoT): IoT ist eine Funkbasierte Technologie zur drahtlosen Datenübertragung und Fernüberwachung über zentrale Systeme oder die Cloud.[10]

Edge Computing: Edge Computing bedeutet, dass Daten direkt am Mikrocontroller verarbeitet werden, ohne sie an einen zentralen Server zu senden. Das reduziert Latenz, spart Bandbreite, ermöglicht schnelle Reaktionen und ideal für Echtzeitanwendungen. [11]

2.2 CAN-Bus im Motorsport und in der Industrie

Der Bereich der Datenerfassung in Fahrzeugen hat sich in den letzten Jahren rasant weiterentwickelt, besonders im Kontext von Rennsport und High-Performance-Anwendungen. Heute ist die präzise Erfassung und Analyse von Fahrzeugdaten entscheidend, um die Fahrzeugleistung zu optimieren, die Sicherheit zu verbessern und neue Technologien voranzutreiben. Ein zentrales Element in der Fahrzeugkommunikation stellt dabei der CAN-Bus dar. Dieser standardisierte Kommunikationsbus ermöglicht eine effiziente Datenübertragung zwischen verschiedenen Fahrzeugkomponenten.

Das CAN wurde in den 1980er Jahren von Bosch entwickelt und hat sich seitdem als zuverlässiger Kommunikationsstandard in der Automobilindustrie und im Motorsport etabliert. Es erlaubt den schnellen Austausch von Nachrichten zwischen elektronischen Steuergeräten, Sensoren und Aktuatoren. Besonders hervorzuheben sind seine Robustheit gegen-über elektromagnetischen Störungen, die prioritätsbasierte Nachrichtenübermittlung sowie die integrierte Fehlererkennung auf Protokollebene. [12]

Grundlagen CAN-Bus

Physikalisch ist der CAN-Bus immer als Zwei-Draht-Leitung ausgeführt, meist mit den Leitungen CAN_H und CAN_L, die verdrillt sind, um Störeinflüsse zu minimieren. Im Ruhezustand liegen beide Leitungen bei etwa 2,5 V, sodass keine Differenzspannung zwischen ihnen entsteht – dieser Zustand entspricht einem rezessiven Bit (logische 1). Sendet ein Teilnehmer ein dominantes Bit (logische 0), so wird CAN_H auf etwa 3,5 V angehoben und CAN_L auf etwa 1,5 V abgesenkt. Dadurch entsteht eine Differenzspannung von mindestens 2 V. Dominante Bits haben Vorrang und überschreiben rezessive Bits: Sobald ein Knoten ein dominantes Bit sendet, gilt dieses für alle anderen Knoten als „0“, auch wenn sie „1“ senden möchten. [13]

Feld	**Bit-Länge**	**Beschreibung**
Start of Frame (SOF)	1	Dominantes Bit, markiert Beginn des Frames
Identifier	11	Kennung (ID) der Nachricht, dient zur Priorisierung und Inhaltsbestimmung
RTR (Remote Transmission)	1	Unterscheidung zwischen Daten-Frame (dominant) und Remote-Frame (rezessiv)
IDE (Identifier Extension)	1	0 = Base Frame (11 Bit), 1 = Extended Frame (29 Bit)
r0 (Reserve)	1	Reserviertes Bit, immer rezessiv
DLC (Data Length Code)	4	Gibt an, wie viele Bytes im Datenfeld folgen (0 bis 8 Byte)

Feld	Bit-Länge	Beschreibung
Datenfeld (DATA)	0–64	Nutzdaten, zwischen 0 und 8 Byte (pro Byte 8 Bit)
CRC (Prüfsumme)	15	Zyklische Redundanzprüfung (Generatorpolynom $x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1$)
CRC-Delimiter	1	Rezessives Bit, trennt CRC von ACK-Feld
ACK-Slot	1	Sender legt rezessives Bit, Empfänger bestätigt mit dominantem Bit, falls fehlerfrei empfangen
ACK-Delimiter	1	Rezessives Bit, trennt ACK-Slot von nachfolgendem End-of-Frame
End of Frame (EOF)	7	Sieben rezessive Bits, markieren das Ende des Frames
Intermission (IFS)	3	Abstand von mindestens drei rezessiven Bits zum nächsten Frame

Tabelle 1 Struktur eines klassischen CAN-Datenframes [13]

Der CAN-Bus ist so aufgebaut, dass er auch dann funktionsfähig bleibt, wenn ein einzelner Teilnehmer ausfällt oder sich fehlerhaft verhält. Dies unterscheidet ihn von anderen Bussystemen wie zum Beispiel dem LIN-Bus (Local Interconnect Network), bei dem die Kommunikation zentral von einem sogenannten Master gesteuert wird. Fällt dieser Master aus, ist das gesamte Netzwerk nicht mehr betriebsfähig. Der LIN-Bus wird jedoch auch von Automobilherstellern genutzt, hauptsächlich für weniger sicherheitsrelevante Anwendungen. Beim CAN-Bus hingegen existiert keine zentrale Steuereinheit, denn alle Steuergeräte sind gleichberechtigte Teilnehmer. Dadurch ist das System dezentral, robust und fehlertolerant.[13]

Damit die Signale nicht am Ende der Leitung reflektiert werden, müssen an beiden Enden des Buskabels jeweils ein 120-Ohm-Abschlusswiderstand installiert werden. Diese Widerstände sorgen dafür, dass das Signal sauber gedämpft wird und keine Störungen durch Signalreflexionen entstehen. Ohne diese Widerstände kann es zu Störungen, fehlerhaften Übertragungen oder kompletten Kommunikationsabbrüchen kommen. [13]

Die Datenrate eines klassischen CAN-Busses beträgt 1 Megabit pro Sekunde, was 1.000.000 Bits pro Sekunde entspricht. Ein typischer Standard-CAN-Frame umfasst etwa 128 Bits. Diese

Bitanzahl enthält sämtliche Bestandteile des Frames, darunter das Start-of-Frame-Bit, den Identifier, Steuerbits, das Datenfeld, die Prüfsumme (CRC), das ACK-Feld sowie das End-of-Frame-Signal.[13]
Wenn nun ein Tastbefehl – zum Beispiel zum Herunterfahren eines Fensters – ausgeführt werden soll und dabei ideale Bedingungen herrschen (also keine Verzögerungen durch einen belegten Bus oder andere Störungen auftreten), kann der entsprechende CAN-Frame sehr schnell übertragen werden.

Die Übertragungszeit berechnet sich wie folgt:

$$t = \frac{\text{Anzahl der Bits}}{\text{Datenrate}} = \frac{128\ \text{Bits}}{1000000\ \text{Bits/s}} = 128\ \mu s \qquad \text{(Gl.1)}$$

Das bedeutet, dass der Befehl theoretisch bereits nach 128 Mikrosekunden vollständig auf dem Bus übertragen sein kann.

Im professionellen Motorsport hat sich der CAN-Bus als primäre Datenquelle zur Erfassung fahrzeuginternen Sensorfeedbacks durchgesetzt. Systeme wie der Bosch C 70 Data Logger nutzen zwei CAN-Busse zur Aufzeichnung und Echtzeitanalyse. Diese Daten werden zur Optimierung von Fahrzeugabstimmung, Fahrerverhalten und Leistungsauswertung herangezogen. Kommerzielle Geräte wie das Bosch C 70 zeichnet sich durch Echtzeitfähigkeit und die Fähigkeit aus, eine hohe Anzahl unterschiedlicher Datenströme (Kanäle) gleichzeitig zu erfassen und zu verarbeiten. Dadurch können viele verschiedene Sensorwerte parallel überwacht werden. Diese Systeme erfordern jedoch spezielle Infrastruktur, hohe Kosten und professionelle Betreuung.

Abbildung 1 Bosch Motorsport C70 Datalogger [14]

Der Data Logger C 70 ist ein leistungsfähiges Erfassungsgerät mit vielfältigen Kommunikationsmöglichkeiten. Als zentrale Steuereinheit kommt ein Infineon XC2287M Mikrocontroller zum Einsatz, der speziell für Automotive-Anwendungen ausgelegt ist und eine zuverlässige sowie schnelle Datenverarbeitung ermöglicht.

Der Logger verfügt über zwei CAN-Bus-Schnittstellen (CAN_1 und CAN_2), die eine flexible Integration in das Fahrzeugnetzwerk erlauben. Zusätzlich stehen zwei Ethernet-100BaseT-Anschlüsse für die schnelle kabelgebundene Kommunikation zur Verfügung. Für serielle Kommunikation ist eine RS232-Schnittstelle integriert, die z. B. zur GPS-Datenverarbeitung genutzt werden kann.[14]

Ein USB-Port ermöglicht die Datenspeicherung und -übertragung über einen USB-Stick. Weitere Anschlussmöglichkeiten bestehen über eine USB-Geräteverbindung, beispielsweise für Bosch-Komponenten.

Zur Versorgung externer Sensoren und Module bietet der Logger verschiedene Spannungsausgänge (5 V, 10 V, 12 V) sowie einen abgesicherten GND. [14]

Die Datenaufzeichnung erfolgt intern über 4 GB Speicher. Für die externe Datensicherung kann ebenfalls der USB-Port genutzt werden.[14]

Eine drahtlose Datenübertragung ist im Logger selbst nicht integriert, kann aber extern über Zusatzhardware realisiert werden.

3 Methoden und Lösungsansätze

Im Rahmen dieses Projekts wurde zu Beginn ein Prototyp entwickelt, um CAN-Bus-Daten in einem Fahrzeug zu erfassen und drahtlos bereitzustellen. Als zentrale Plattform diente der Raspberry Pi, der sich durch die kompakte Bauform, ausreichende Rechenleistung und durch sein integriertes WLAN-Modul auszeichnet. Eine wichtige Voraussetzung für den Prototyp war die lokale Aufzeichnung der CAN-Daten. Zusätzlich sollten die Daten in Echtzeit Drahtlos an ein externes Gerät wie ein Notebook oder Smartphone übertragen werden.

Im Vergleich zu klassischen Mikrocontrollern wie dem STM32F103 (72 MHz Taktfrequenz, 64 KB RAM) oder dem ATmega328P (20 MHz, 2 KB RAM) bietet der Raspberry Pi 4 mit seinem 1,5 GHz Quad-Core ARM Cortex-A72 Prozessor und bis zu 8 GB RAM ein Vielfaches an Rechenleistung und Speicherressourcen.[15][16][17] Diese Leistungsreserven ermöglichen es, neben der Erfassung und Verarbeitung von CAN-Daten auch umfangreiches Datenlogging und Visualisierung gleichzeitig durchzuführen.

3.1 Auswahl der Hardwareplattform: Raspberry Pi

Der Raspberry Pi wird aufgrund seiner Vielseitigkeit, Kompaktheit und guten Softwareunterstützung als zentrale Recheneinheit gewählt. Er bietet eine breite Palette an Schnittstellen (GPIO, SPI, I²C, USB, HDMI etc.) sowie ein vollständiges Linux-Betriebssystem mit umfangreichen Netzwerkfunktionen. Besonders vorteilhaft für dieses Projekt ist die Tatsache, dass aktuelle Modelle über integriertes WLAN verfügen, wodurch eine drahtlose Kommunikation ohne zusätzliche Hardwaremodule möglich bleibt.[17] In Abbildung 2 ist ein Raspberry Pi B+ V1.2 dargestellt, auf dem die 40-Pin-GPIO-Leiste, der HDMI-Anschluss, der LAN-Port sowie die 4× USB 2.0 Schnittstellen deutlich zu erkennen sind.
Der vorhandene HDMI-Ausgang ermöglicht eine direkte Anzeigeausgabe. Das ist besonders bei der Entwicklung, Codeanpassung und Fehlersuche hilfreich, da so direkt auf die grafische Benutzeroberfläche oder das Terminal über einen angeschlossenen Monitor zugegriffen werden kann.

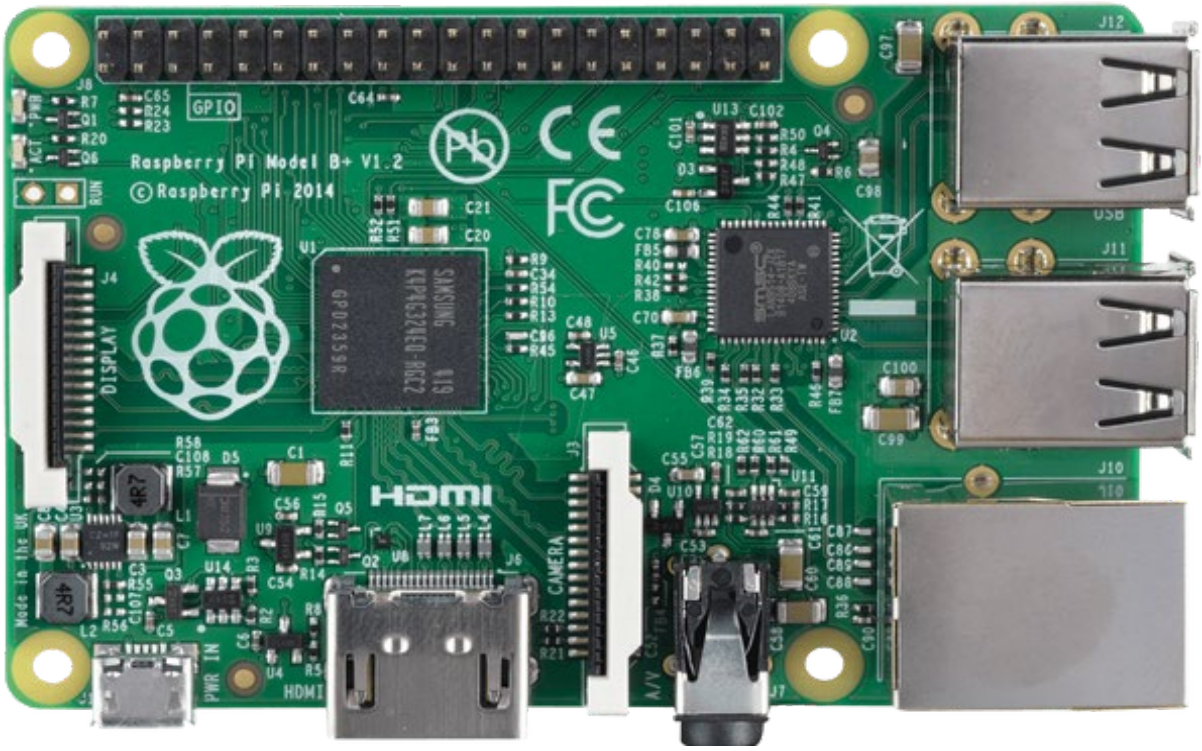

Abbildung 2 Raspberry Pi B+ V1.2 mit 40-Pin-GPIO, HDMI, LAN und 4× USB 2.0 [18]

Ein Nachteil des Raspberry Pi ist jedoch, dass er standartmäßig keine CAN-Bus-Schnittstelle besitzt. CAN-Bus ist ein differenzielles, robustes Kommunikationsprotokoll, das insbesondere in der Automobiltechnik weit verbreitet ist und zur Verbindung von Steuergeräten, Sensoren und Aktoren sorgt. Um den Raspberry Pi CAN-fähig zu machen, musste ein externer CAN-Controller verwendet werden. Die nachfolgende Abbildung zeigt die vollständige GPIO-Belegung des Raspberry Pi. Dabei sind insbesondere die für die SPI-Kommunikation relevanten Pins hervorgehoben, die für die Verbindung mit dem MCP2515 benötigt werden. [19]

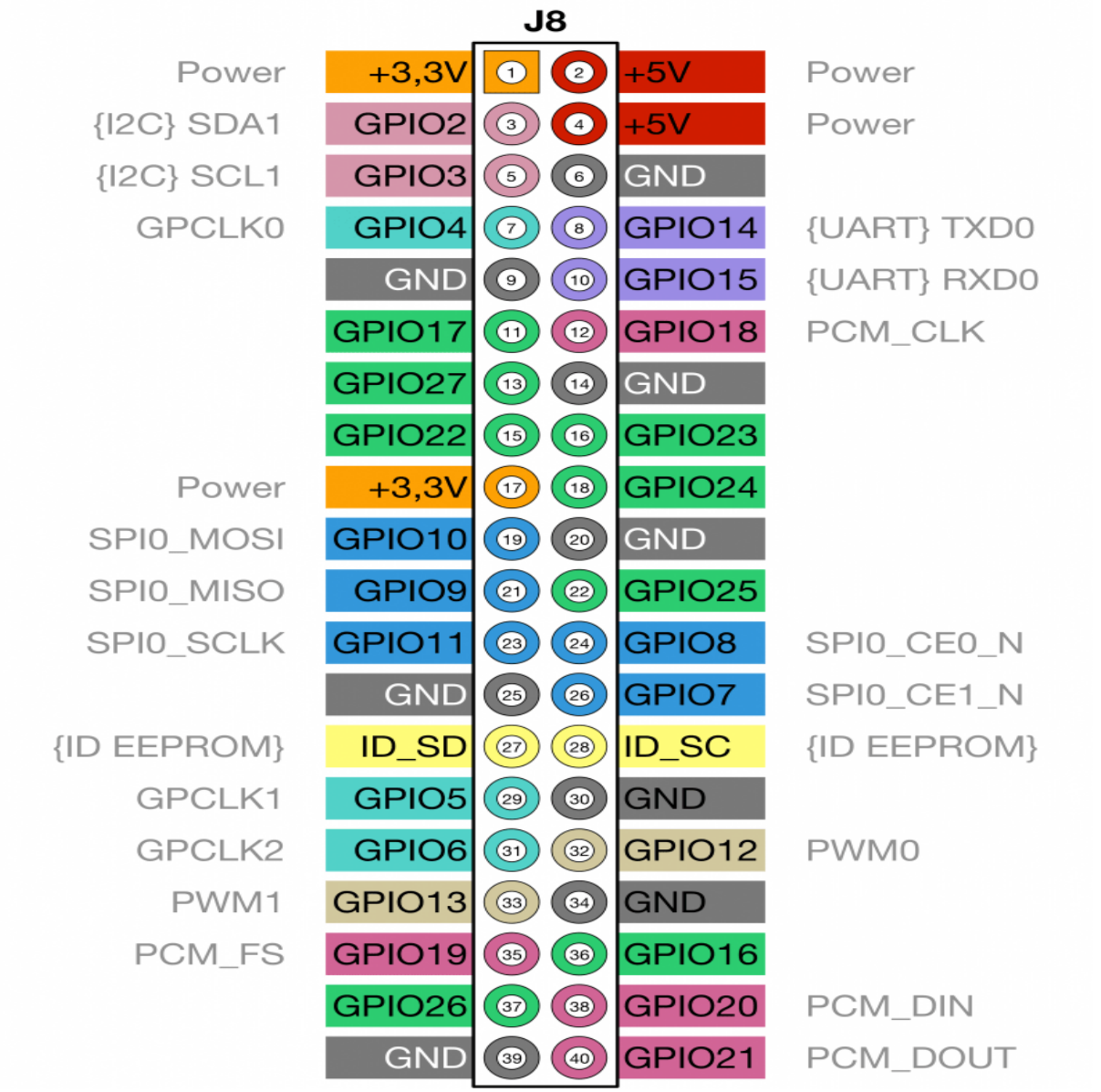

Abbildung 3 GPIO-Pinbelegung Raspberry Pi [19]

3.2 Anbindung an den CAN-Bus: MCP2515

Zur testen der Funktion, kommt in diesem Projekt ein vorgefertigtes CAN-Modul zum Einsatz, auf dem sowohl der MCP2515 als auch der TJA1050 bereits integriert sind. Dabei handelt es sich um eine bewährte und weit verbreitete Kombination, die speziell für Mikrocontroller und Einplatinencomputer wie den Raspberry Pi entwickelt wurde. Der MCP2515 fungiert als eigenständiger CAN-Controller, der die Kommunikation mit dem Raspberry Pi über die SPI-Schnittstelle abwickelt. Dadurch lässt sich das Pi, trotz fehlender CAN-Unterstützung an den CAN-Bus anbinden.

Der integrierte Transceiver TJA1050 übernimmt dabei die Pegelwandlung zwischen den digitalen SPI-Signalen des MCP2515 und den differentiellen Spannungspegeln des Fahrzeugnetzwerks (CAN_H und CAN_L). Diese Pegel sind notwendig, um eine normgerechte physikalische Verbindung zum CAN-Bus herzustellen.

Das Modul bietet somit eine kompakte und zuverlässige Lösung für die CAN-Kommunikation und eignet sich ideal für Anwendungen im Automotive-Bereich, insbesondere für Prototypen oder Analysezwecke. [8] In Abbildung 4 ist das verwendete Modul dargestellt. Es zeigt die zentralen Bauteile sowie die Anschlüsse für CAN_H und CAN_L.

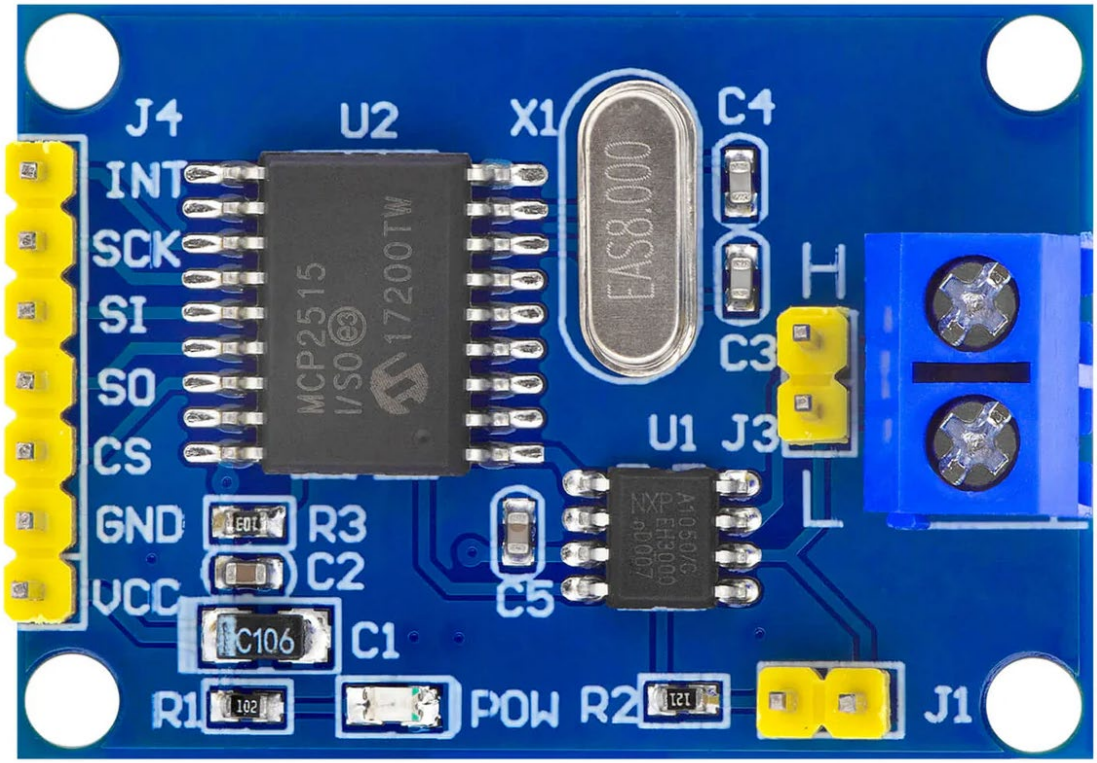

Abbildung 4 CAN-Bus-Modul mit MCP2515 und TJA1050 [20]

Pinbelegung des TJA1050 und Datenfluss

Um den Signalfluss zwischen Fahrzeug, Transceiver und Mikrocontroller nachvollziehen zu können, wird im Folgenden davon ausgegangen, dass zunächst Signale vom Fahrzeug über die CAN-Bus-Leitungen CAN High (CAN_H) und CAN Low (CAN_L) am TJA1050 ankommen. Die folgende Tabelle zeigt die relevanten Pins des TJA1050 sowie deren jeweilige Funktion im Datenfluss.

Pin	Bezeich-nung	Funktion
1	**TXD**	**Transmit Data Input** – Dieser Pin empfängt die von der CAN-Controller-Einheit (z. B. MCP2515) zu sendenden Daten, die an den CAN-Bus weitergeleitet werden. Die Daten werden hier als serielle Bitfolge vom MCP2515 gesendet. TXD ist dabei **der Pin, über den die Nachrichtendaten aus dem MCP2515 an den TJA1050 übertragen werden**, um sie anschließend auf den CAN-Bus zu schicken.
2	**GND**	Masseanschluss
3	**VCC**	Versorgungsspannung 5 V
4	**RXD**	**Receive Data Output** – Dieser Pin empfängt die CAN-Nachrichten, die von einem anderen Gerät auf dem CAN-Bus gesendet wurden. Die empfangenen Daten werden hier von den differentiellen CAN-Pegeln in digitale Logikpegel umgewandelt und an den MCP2515 weitergeleitet, um dort verarbeitet und über SPI an den Raspberry Pi übergeben zu werden. Der **RXD-Pin** liefert dem Raspberry Pi somit die empfangenen CAN-Nachrichten.
5	**Vref**	Referenzausgang (optional, häufig unbeschaltet)
6	**CANL**	**CAN Low** – Die differenzielle Busleitung des CAN-Busses, die zusammen mit **CANH** die physikalische Kommunikationsstrecke zwischen Geräten bildet.
7	**CANH**	**CAN High** – Die differenzielle Busleitung des CAN-Busses, die zusammen mit **CANL** für den Transport der Daten zwischen den CAN-Geräten verantwortlich ist.

Tabelle 2 Pinbelegung des TJA1050 CAN-Transceivers mit Funktionsbeschreibung [21]

Zur Veranschaulichung ist in der folgenden dargestellten Abbildung ein vollständiger CAN-Datenframe im Standardformat (11-Bit-Identifier) zu sehen. Dabei werden sowohl die logisch übertragene Bitfolge als auch die physikalischen Signalverläufe auf CAN_H und CAN_L sowie das daraus resultierende Signal am CAN RX dargestellt.

Der RXD-Pin des TJA1050 entspricht genau dem Signalverlauf des CAN RX. Dieses Signal stellt die rekonstruierte digitale Bitfolge dar, die der Transceiver aus den differenziellen Pegeln auf dem Bus (CAN_H und CAN_L) erkennt. Das bedeutet: Der TJA1050 wertet kontinuierlich

die Pegeldifferenz zwischen CAN_H und CAN_L aus und gibt über RXD den logischen Zustand auf dem Bus weiter:

- Bei einer dominanten Buszustand (CAN_H > CAN_L, z. B. 3.5 V / 1.5 V) gibt der RXD-Pin eine logische 0 aus.
- Bei einem rezessiven Buszustand (CAN_H ≈ CAN_L, z. B. 2.5 V / 2.5 V) liegt eine logische 1 am RXD-Pin an.

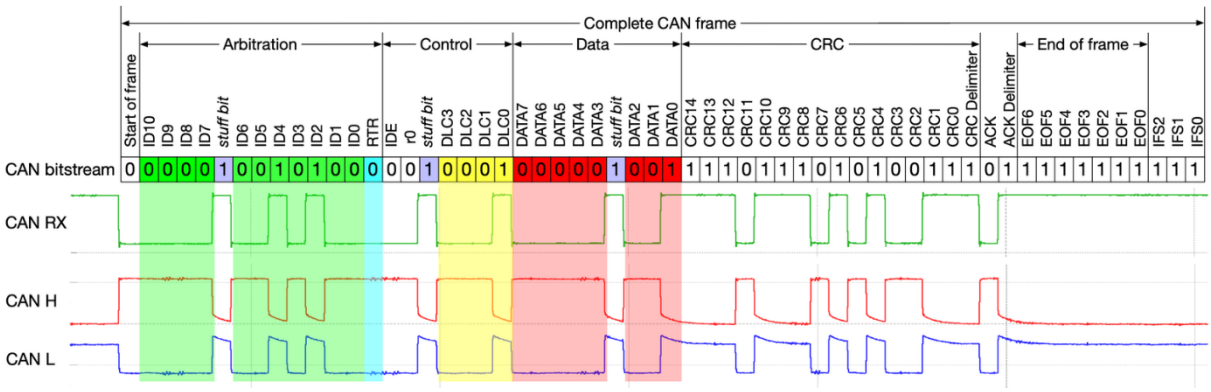

Abbildung 5 Vollständiges CAN-Frame mit der Umwandlung in CAN RX-Signal [22]

Der RXD-Pin liefert also den vom Bus „gelesenen“ Bitstrom zurück und ist für den MCP2515 der Empfangskanal. Im Diagramm zeigt sich dies darin, dass der Verlauf von CAN RX genau dem logisch interpretierten Bitmuster entspricht und der einschließlich automatisch eingefügten Stuff-Bits (z. B. nach fünf aufeinanderfolgenden identischen Bits), die zur Synchronisation dienen.

Mit der vereinfachten Darstellung des RXD-Signals kann der MCP2515 das vollständige Datenpaket entschlüsseln. Dabei ist zu erkennen, dass nach einer Reihe von mehreren rezessiven Einsen (1) ein dominantes Null-Bit (0) folgt. Diese Einsen können sowohl im Leerlauf als auch am Ende eines Pakets auftreten, während das dominante Null-Bit den Beginn eines neuen Pakets signalisiert. Dieses dominante 0-Bit markiert den Start-of-Frame und zeigt somit deutlich den Beginn einer neuen CAN-Nachricht an.

Für den stabilen Betrieb des MCP2515 wird ein Quarzoszillator benötigt, der die notwendige Taktfrequenz bereitstellt. In der Praxis wird hierfür ein 8 MHz Quarz eingesetzt, der an die Pins OSC1 und OSC2 des MCP2515 angeschlossen wird.

Damit der Quarz zuverlässig in Schwingung versetzt wird, sind zwei Keramikkondensatoren erforderlich. Diese werden jeweils von OSC1 und OSC2 nach Masse (GND) geschaltet und bilden gemeinsam mit dem Quarzeinen Schwingkreis.
Diese erforderlichen Kapazitätswerte (auch Lastkapazitäten**)** sind abhängig vom verwendeten Quarztyp sowie von der Leiterplattengeometrie (z. B. Leiterbahnlänge und parasitäre Kapazitäten). Sie werden durch folgende Formel berechnet: [23]

$$c_L = \frac{c_1 c_2}{c_1 + c_2} + c_{stray} \quad \text{(Gl.2)}$$

Dabei gilt:

- **C_L**= vom Quarzhersteller angegebene Lastkapazität (z. B. 18 pF)
- **C_1, C_2** = die zwei Kondensatoren zwischen Quarz und GND
- **C_{stray}**= parasitäre Kapazität des Layouts (typisch 5–10 pF)

In der Praxis wählt man häufig 18 pF bis 22 pF für C_1 und C_2,um den vom Hersteller geforderten Last Wert zu erreichen. Abweichungen von diesen Werten oder das Fehlen der Kondensatoren kann zu unstabiler Takterzeugung, Kommunikationsfehlern oder sogar einem Totalausfall der CAN-Bus-Funktion führen.

Damit sieht der Schaltplan für den für das vorgefertigte Can-Modul mit TJA1050 zusammen mit dem MCP2515I/SO und den dazugehörigen Quarzoszillator wie folgt aus:

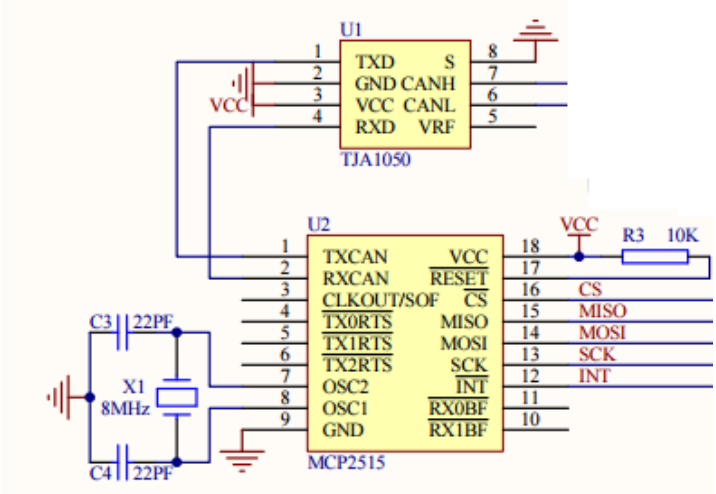

Abbildung 6 Schaltplan des vorgefertigten CAN-Moduls [24]

Abschließend fehlt die Verbindung vom MCP2515 zum Raspberry Pi Mikrocontroller. Die nachfolgende Tabelle zeigt die notwendigen elektrischen Verbindungen im Detail.

Funktion	**MCP2515 Pin**	**Raspberry Pi Pin**	**GPIO-Nummer von Raspberry Pi**	**Anmerkung**
VCC (Stromversorgung)	VCC	Pin 2 oder 4	–	5V Versorgung
GND (Masse)	GND	Pin 6, 9, 14, 20,	–	Masse
SPI MOSI	SI	Pin 19	GPIO 10	Master Out Slave In- Daten vom Pi zum MCP2515
SPI MISO	SO	Pin 21	GPIO 9	Master In Slave Out- Daten vom MCP2515 zum Pi
SPI SCK	SCK	Pin 23	GPIO 11	SPI Clock

Funktion	MCP2515 Pin	Raspberry Pi Pin	GPIO-Nummer von Raspberry Pi	Anmerkung
SPI CS Chip Select	CS	Pin 24	GPIO 8	Kann auch auf anderen GPIO gelegt werden
Interrupt (optional)	INT	Pin22 (z. B.)	GPIO 25 (z. B.)	Muss in Software konfiguriert werden, damit Mikrocontroller den MCP2515 findet

Tabelle 3 GPIO-Zuordnung und Anschlussplan für Raspberry Pi und MCP2515 [25]

Die empfangene Nachricht wird dabei vom MCP2515 über den MISO-Pin an den Raspberry Pi übermittelt.
Somit ergibt sich ein durchgängiger Signalfluss:

Fahrzeugnetzwerk → CAN High/Low → CAN-Transceiver → MCP2515 → SPI → Raspberry Pi

3.3 Grundkonfiguration des Raspberry Pi und softwareseitige Einbindung des MCP2515

Nachdem die Hardware-Voraussetzungen für die CAN-Kommunikation mit dem Raspberry Pi geschaffen wurden, folgt im nächsten Schritt die Einrichtung und Konfiguration der benötigten Softwarekomponenten. Dieser Abschnitt behandelt die Installation der Treiber und Bibliotheken, die Konfiguration der CAN-Schnittstelle sowie die Implementierung erster Kommunikationsroutinen. Ziel ist es, eine stabile und zuverlässige Datenübertragung über den CAN-Bus zu ermöglichen und die empfangenen sowie gesendeten Daten entsprechend verarbeiten zu können. Die folgenden Kapitel beschreiben dabei sowohl die grundlegenden Schritte der Systemeinrichtung als auch Softwarelösungen, die für das Projekt verwendet wurden.
Bevor mit der eigentlichen Softwareinstallation zur CAN-Kommunikation begonnen werden kann, muss der Raspberry Pi zunächst grundlegend eingerichtet werden. Dazu wird als Erstes die microSD-Karte mit einem geeigneten Betriebssystem installiert. Dies kann beispielsweise mithilfe der Raspberry Pi Imagers Software erfolgen, die für die verschiedenen Betriebssysteme verfügbar ist.

Nach dem Installieren wird die SD-Karte in den Raspberry Pi 2B eingesetzt. Für die erste Inbetriebnahme werden ein Monitor, eine Tastatur sowie eine Maus angeschlossen. Zusätzlich wird der Raspberry Pi über WLAN oder ein Ethernet-Kabel mit dem Netzwerk verbunden, um später benötigte Softwarepakete installieren und Updates durchführen zu können.

Sobald der Raspberry Pi hochgefahren ist, erfolgen die ersten grundlegenden Einstellungen wie die Auswahl der Sprache, die Einrichtung des Benutzerkontos, die Netzwerkkonfiguration sowie die Durchführung von Systemupdates.

Nachdem der Raspberry Pi grundlegend eingerichtet wurde, folgt nun die Einrichtung der Software zur CAN-Kommunikation. Alle folgenden Schritte werden über das Terminal (LXTerminal) unter Raspberry Pi OS ausgeführt.

Nach der erfolgreichen Ersteinrichtung wurde die weitere Konfiguration sowie die Entwicklung und Bearbeitung der Software über Visual Studio Code auf einem Laptop durchgeführt. Die Verbindung zum Raspberry Pi erfolgte dabei per Remote-SSH, wodurch sich Dateien und Terminalbefehle direkt auf dem Pi ausführen lassen.

Zur Inbetriebnahme des externen CAN-Controllers MCP2515 auf dem Raspberry Pi wird zunächst die SPI-Schnittstelle über das Tool *raspi-config* aktiviert.[25] Anschließend werden die benötigten CAN-Werkzeuge (*can-utils*) über den Paketmanager installiert. Damit der MCP2515 beim Systemstart erkannt wird, wird ein entsprechender Konfigurationseintrag für Hardware-Erweiterungen in der Datei */boot/firmware/config.txt* ergänzt. Dabei werden auch die Taktfrequenz des Quarzes und der verwendete Interrupt-GPIO angegeben.[25] Nach einem Neustart des Systems wird die CAN-Schnittstelle manuell aktiviert und mit einer Bitrate von 500 kBit/s konfiguriert. Die erfolgreiche Einrichtung wird schließlich durch den Befehl *ifconfig* überprüft, wobei das Interface can0 sichtbar und aktiv geschaltet ist.[25]

4 Ergebnisse

In diesem Kapitel wird das entwickelte Projekt hinsichtlich seiner Funktionalität überprüft. Ziel ist es, die korrekte Inbetriebnahme der Hardware sowie die fehlerfreie Kommunikation über den CAN-Bus sicherzustellen. Durch gezielte Tests und praktische Anwendungen soll die Leistungsfähigkeit und Stabilität der umgesetzten Lösung nachgewiesen werden.

Um die Fahrzeugelektronik realistisch darzustellen, wurde ein Mikrocontroller mit einem CAN-Bus-System eingesetzt, das einen 10 kΩ Potentiometer als analogen Eingang sowie einen standardmäßigen 120 Ω Abschlusswiderstand verwendet. Der MCP2515 CAN-Controller und

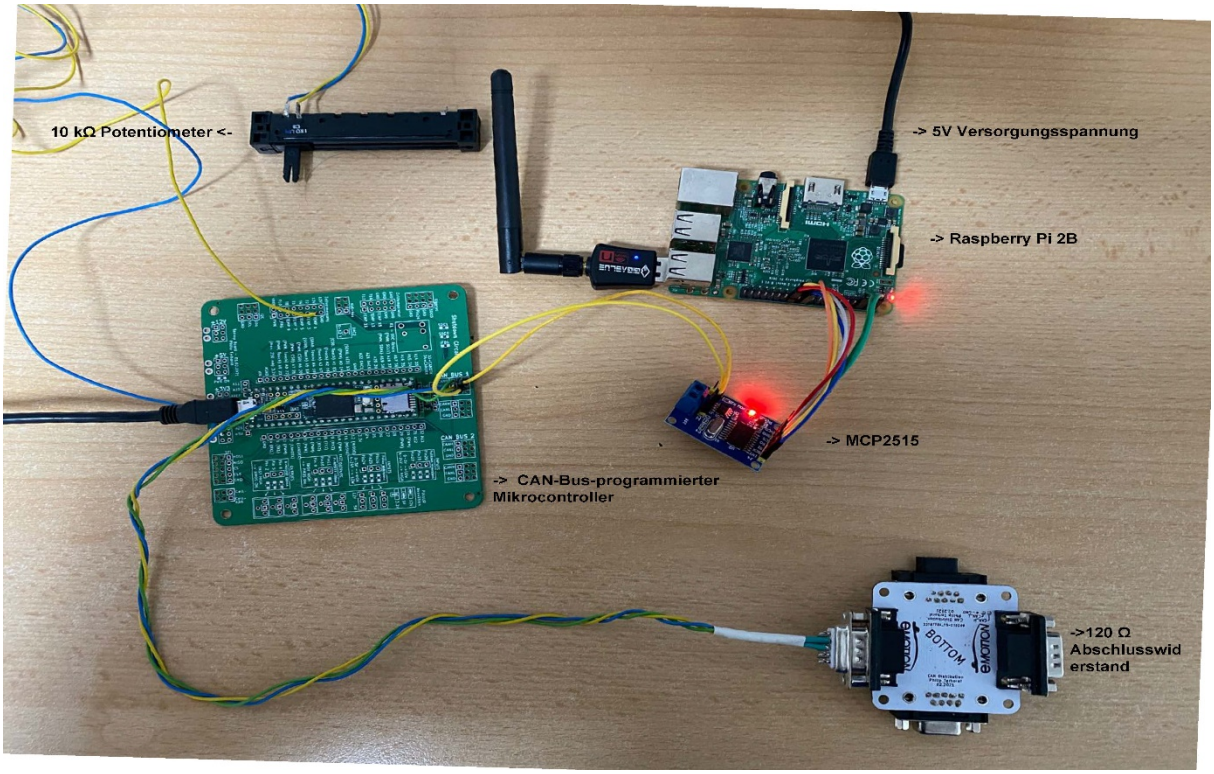

Abbildung 7 Aufbau der Testschaltung mit realistischem CAN-Bus-System

4.1 Funktionstest der CAN-Kommunikation mit variablen Eingangssignalen

Aufbauend auf dem zuvor eingerichteten System wurde die Kommunikation über den CAN-Bus auf ihre Funktionalität hin überprüft. Zur Analyse des Datenverkehrs am CAN-Interface des Raspberry Pi kam das Terminal-Tool candump can0 zum Einsatz. Für den Funktionstest wurde ein am Mikrocontroller angeschlossener 10 kΩ Potentiometer verwendet, um analoge Eingangswerte zu simulieren. Die eingestellten Spannungswerte wurden vom Mikrocontroller erfasst, in digitale Signale umgewandelt und anschließend als CAN-Nachrichten über den Bus gesendet. Durch das manuelle Verändern der Potentiometer Stellung konnten unterschiedliche Werte generiert werden, wodurch sich die übermittelten Dateninhalte sichtbar änderten. Auf diese Weise kann die Funktionstüchtigkeit der gesamten Übertragung von der analogen Erfassung bis zur digitalen Darstellung auf dem Raspberry Pi erfolgreich nachgewiesen werden.

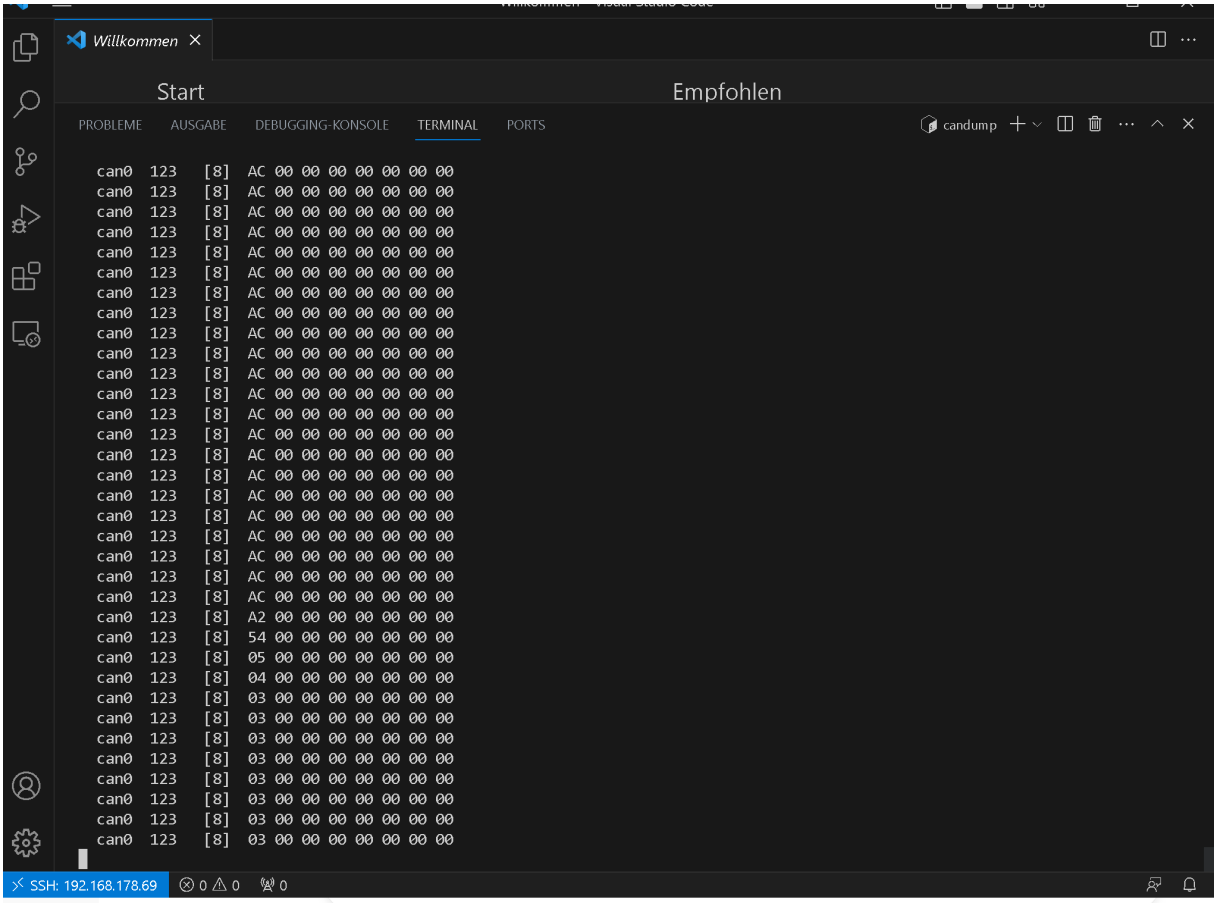

Abbildung 8 Auswertung der CAN-Signale in hexadezimaler Darstellung

Ein besonderes Augenmerk liegt auf dem ersten Datenbyte der CAN-Nachricht. Dieses Byte wurde vom Mikrocontroller entsprechend dem variablen Eingangswert, der durch die Stellung des angeschlossenen 10 kΩ Potentiometers bestimmt wurde, generiert. Durch die manuelle Veränderung der Potentiometer Position änderte sich somit direkt der Spannungswert am variablen Eingang. Dieser wurde digitalisiert und als Teil der CAN-Nachricht übertragen, wobei sich insbesondere der Wert in der ersten Daten-Spalte sichtbar veränderte.

4.2 Drahtlose Übertragung über LoRa

Nachdem die kabelgebundene Kommunikation über den CAN-Bus erfolgreich getestet wurde, sollte im nächsten Schritt eine drahtlose Übertragung der erfassten Daten implementiert werden. Zu Beginn der Projektentwicklung wurde hierfür die Nutzung von WLAN in Erwägung gezogen. Allerdings zeigte sich im Rahmen einer Reichweitenabschätzung, dass die typischen Übertragungsdistanzen WLAN-Module von etwa 500 Metern nicht den Anforderungen des Projekts genügen.

Aus diesem Grund wurde im weiteren Verlauf auf eine geeignete Alternative umentschieden, die eine stabile und energieeffiziente Datenübertragung über größere Entfernungen ermöglicht.
Die Entscheidung fiel schließlich auf das LoRa-Funkprotokoll, das sich durch seine hohe Reichweite, geringe Leistungsaufnahme und robuste Übertragungseigenschaften auszeichnet und damit eine optimale Lösung für den angestrebten Anwendungsfall darstellt.

Das LoRa-Modul SX1262 von Semtech, das im 868 MHz-Frequenzband arbeitet, bietet gegenüber herkömmlichen WLAN-Verbindungen mehrere entscheidende Vorteile, die es für industrielle und IoT-Anwendungen attraktiv machen. [26] Ein zentraler Vorteil liegt in der großen Kommunikationsreichweite. Während WLAN-Netzwerke typischerweise auf wenige bis einige Hundert Meter beschränkt sind, ermöglicht das SX1262 LoRa-Modul Reichweiten von bis zu fünf Kilometern unter optimalen Bedingungen. Diese Reichweite wird durch die Modulationstechnologie von LoRa erreicht, die das Signal sehr robust gegen Störungen macht und somit auch in komplexen und elektromagnetisch belasteten Umgebungen eine zuverlässige Datenübertragung sicherstellt.

Abbildung 9 SX1262 868M LoRa HAT [26]

Ein weiterer wichtiger Aspekt ist der geringe Energieverbrauch. Im Vergleich zu WLAN-Modulen, die häufig mehrere hundert Milliampere Strom verbrauchen, benötigt das SX1262 im Sendebetrieb etwa 133 mA, im Empfangsmodus 11 mA und im Tiefschlaf nur 2 µA.

Das Frequenzband, in dem das SX1262 arbeitet, liegt im Bereich von etwa 850 bis 915 MHz und gehört in vielen Regionen, darunter auch in Europa, zu den sogenannten lizenzfreien oder „gesetzlos“ nutzbaren Frequenzbereichen. Das bedeutet, dass für die Nutzung dieser Frequenzen keine spezielle Funklizenz erforderlich ist, was den Einsatz der Technologie in unterschiedlichsten Anwendungen vereinfacht.[26]

Wie in Tabelle 1 dargestellt, besteht ein Standard-CAN-Frame aus 111 Bit, was etwa 14 Bytes entspricht. Davon entfallen 8 Bytes auf reine Nutzdaten, während die restlichen 6 Bytes Protokollinformationen wie die CAN-ID, Steuerbits, CRC-Prüfsumme und Rahmenkennzeichnungen enthalten. Bei der Übertragung von CAN-Daten über LoRa werden diese Protokollbestandteile ebenfalls berücksichtigt.

Das eingesetzte LoRa-Modul SX1262 erlaubt eine maximale Nutzdatenlänge von 240 Bytes pro Sendeframe. [26] Daraus ergibt sich eine effektive LoRa-Nutzdatenrate von 240 Bytes pro Sekunde. Die Anzahl der übertragbaren CAN-Pakete pro LoRa-Übertragung ergibt sich damit zu:

$$\frac{240\ \text{Bytes}}{14\ \times \frac{\text{Bytes}}{\text{CAN Paket}}} = 17{,}14\ \text{CAN Pakete} \qquad \text{(Gl.3)}$$

Jedes CAN-Paket enthält 8 Bytes reine Nutzdaten, wobei jedes Byte einen einzelnen Sensorwert repräsentieren kann. Daraus ergibt sich die übertragbare Anzahl an Sensorwerten pro LoRa-Nachricht.

$$17{,}14\ Can\ Pakete \times 8\ Byt = 137{,}12\ \ Sensorwerte \qquad \text{(Gl.4)}$$

Das bedeutet, dass bis zu 137 einzelne Sensorwerte innerhalb eines LoRa-Sendevorgangs übertragen werden können.
Zudem verfügt das SX1262-Modul über einen internen Buffer mit 1000 Bytes, [26] der die Zwischenspeicherung mehrerer CAN-Nachrichten erlaubt. Dieser Puffer ermöglicht es, mehrere Nachrichten in einer Warteschlange vorzuhalten und anschließend in mehreren, aufeinander-folgenden LoRa-Frames zu übertragen.

Dadurch kann eine kontinuierliche Datenübertragung realisiert werden, selbst wenn die maximale Frame-Größe überschritten wird.Im praktischen Aufbau der Kommunikationsarchitektur wurde das SX1262 LoRa HAT-Modul direkt auf den Raspberry Pi aufgesteckt. Durch die standardisierte Bauweise ist eine einfache mechanische und elektrische Anbindung möglich, da die GPIO-Leiste des Raspberry Pi exakt zu den Anschlüssen des SX1262 HAT passt.

Für die Spannungsversorgung des LoRa-Moduls wurde keine externe Stromquelle verwendet. Stattdessen erfolgte die Stromversorgung direkt über den 5V-Pin des Raspberry Pi, welcher über die GPIO-Leiste zur Verfügung steht. Hierzu wurde die gelbe Klemme so eingesteckt, dass die Stromversorgung vom Raspberry Pi direkt an das SX1262-Modul weitergeleitet wird (Pi → LoRa). Dies reduziert den Verkabelungsaufwand und minimiert mögliche Fehlerquellen, die durch externe Spannungsversorgungen oder unsaubere Masseverbindungen entstehen könnten.

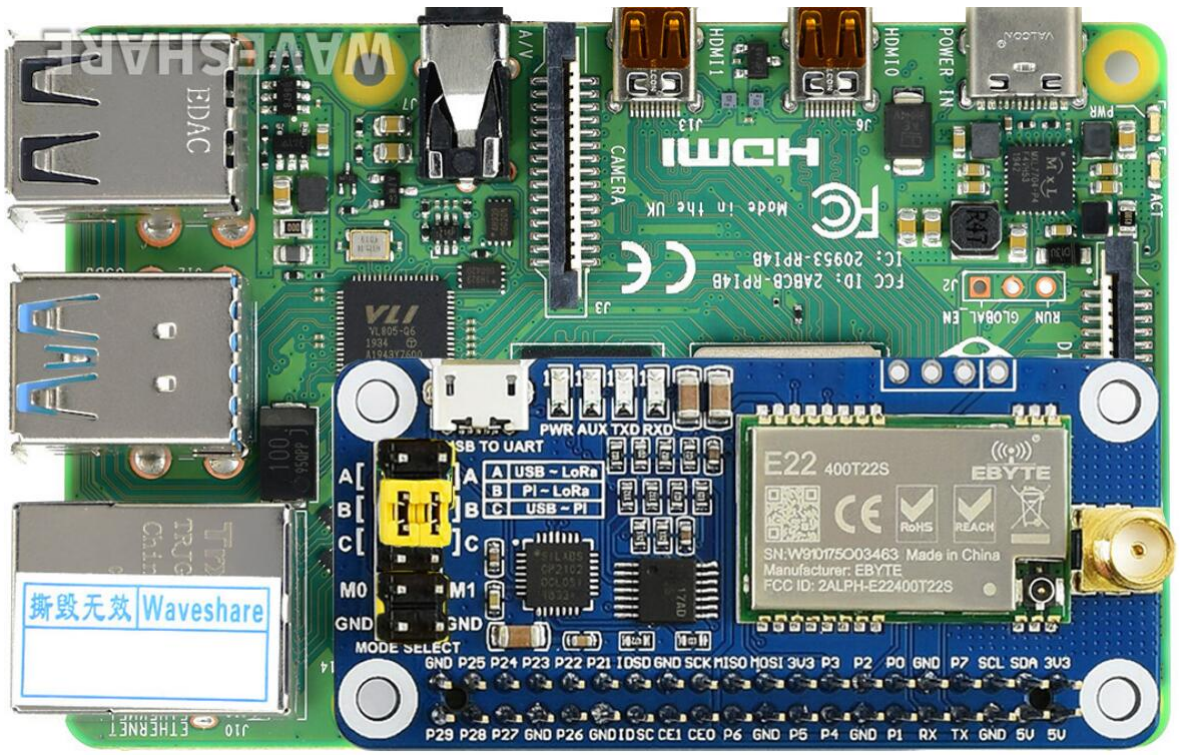

Abbildung 10 Konfiguration der Spannungsversorgung über interne Klemme [26]

Für die Anbindung des LoRa-HATs wurde die Anleitung auf der Waveshare-Wiki-Seite zum SX1262 915 MHz LoRa-HAT herangezogen. Sie beschreibt ausführlich die erforderliche Hard- und Softwarekonfiguration, insbesondere die Anbindung an den Raspberry Pi über die GPIO-Schnittstelle. Dazu gehört u. a. das Setzen der Jumper, die Einrichtung der seriellen Schnittstelle sowie die Installation erforderlicher Python-Module.

Die beiden im Projekt verwendeten Python-Programme (sx126x.py und main.py) stammen ebenfalls aus dieser Anleitung.

- sx126x.py enthält die Treiber- und Steuerlogik zur direkten Kommunikation mit dem SX1262-Modul über UART und GPIO.
- main.py stellt eine Benutzeroberfläche dar, über die Nachrichten gesendet oder empfangen werden können (z. B. manuelle Eingaben oder automatisiertes Senden der CPU-Temperatur).

Diese Programme ermöglichen eine direkt Punkt-zu-Punkt-Kommunikation zwischen zwei LoRa-Geräten herzustellen. Nach der erfolgreichen Einrichtung des ersten Geräts wurde ein zweiter Raspberry Pi mit einem SX1262 868M LoRa-HAT ausgestattet.

Auch auf diesem Gerät wurden die notwendigen Konfigurationsschritte gemäß der Anleitung durchgeführt. Die beiden mit LoRa bestückten Raspberry Pis konnten nun direkt über die LoRa-Schnittstelle miteinander kommunizieren.

Um die Funktionalität der LoRa-Verbindung zu testen, wurde das Python Hauptprogramm (main.py) auf beiden Raspberry Pis ausgeführt. Beide Geräte wurden per SSH über die Remote-SSH-Erweiterung von Visual Studio Code verbunden, sodass eine direkte Terminalverbindung zu jedem Raspberry Pi möglich war. Dadurch konnten die Ein- und Ausgaben auf beiden Seiten parallel beobachtet werden, um sicherzustellen, dass die Kommunikation über LoRa korrekt funktioniert.

Beide Programme (sx126x.py und das Hauptskript main.py) wurden auf beiden Raspberry Pis gestartet. Sobald sie erfolgreich geladen sind, erscheinen auf jedem Gerät die Eingabeaufforderungen:

Press Esc to exit

Press i to send

Press s to send CPU temperature every 10 seconds

Um den Sendemodus zu aktivieren, wählt man auf dem Sender-Pi (IP 192.168.178.69) per Taste **s** die zyklische Übertragung der CPU-Temperatur aus. Sofort wechselt der andere Raspberry Pi (IP 192.168.178.103) in den Empfangsmodus, ohne dass dort eine weitere Eingabe nötig ist. Im Terminalfenster dieses Empfängers sieht man dann fortlaufend Zeilen wie in der Abbildung, die anzeigen

- auf welcher Frequenz gerade empfangen wird (z. B. „868.125 MHz“)
- und den aktuellen CPU-Temperaturwert, den das Gegenstück soeben gesendet hat.

Damit ist eine einfache, bidirektionale LoRa-Kommunikation zwischen den beiden Geräten demonstriert. In der folgenden Abbildung sind die jeweils empfangenen CPU-Temperaturen deutlich zu erkennen. Der empfangende Raspberry Pi listet sie fortlaufend im Terminal auf. So lässt sich nachvollziehen, dass die Datenübertragung per LoRa erfolgreich funktioniert und regelmäßig aktualisiert wird.

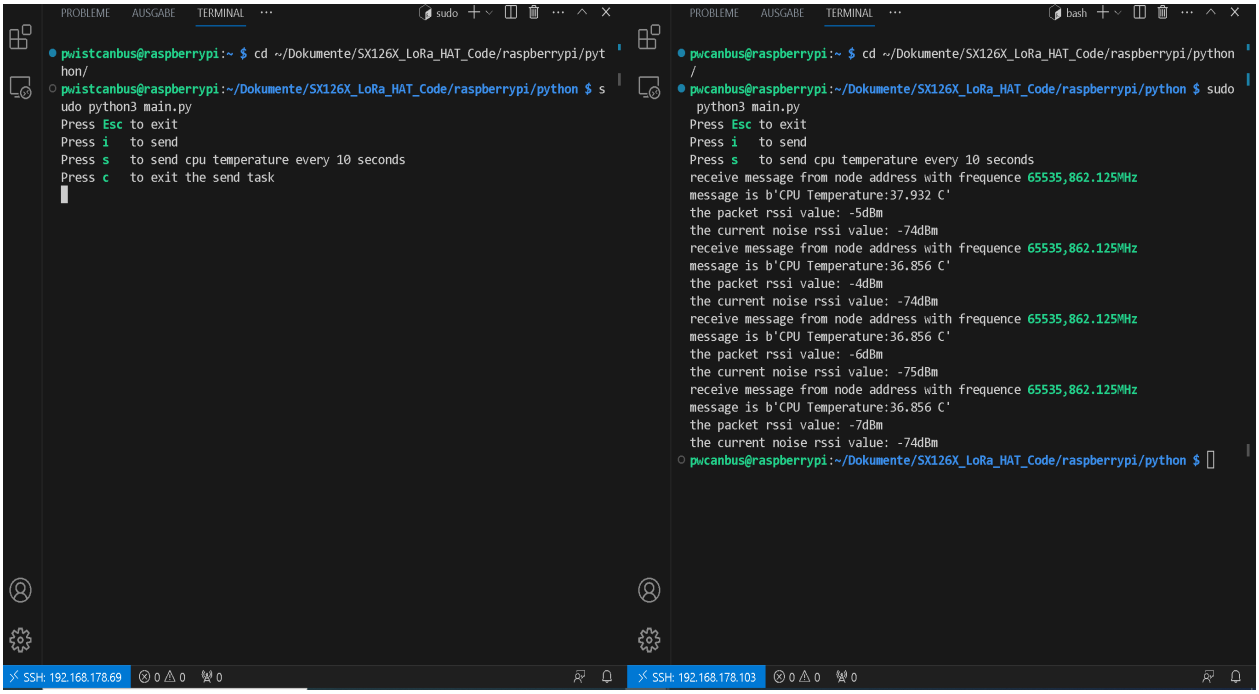

Abbildung 11 Testbetrieb der LoRa Kommunikation mit Referenzcode

4.3 Drahtlose Übertragung der Can-Bus Daten

Nach dem erfolgreichen Test der LoRa-Kommunikation wurde der bestehende Python-Code so erweitert, dass nun statt der CPU-Temperatur CAN-Bus-Daten über das LoRa-Modul gesendet werden. Die neue Funktion *send_can_data()* orientiert sich dabei an der bisherigen Funktion *send_cpu_continue()*, die regelmäßig die Temperatur des Raspberry Pi übertragen hat. Damit die CAN-Daten verarbeitet werden können, wurde oben im Code die Bibliothek *import can* ergänzt. Ohne diese Bibliothek könnten weder das CAN-Interface geöffnet noch

empfangene Nachrichten ausgewertet werden. Die nötigen Timer-Elemente wurden übernommen, damit *send_can_data ()* in festen Abständen ausgeführt wird.

In der Funktion wird die CAN-Schnittstelle mit *can.interface.Bus()* geöffnet. Dann wird mit *bus.recv(1.0)* bis zu einer Sekunde auf eine Nachricht gewartet. Bei empfangener Nachricht werden ID und Daten mithilfe der festgelegten Python Attribute *message.arbitration_id* und *message.data* ausgelesen und formatiert. Zusätzlich stehen auch Informationen wie der Zeitstempel (messa-ge.timestamp) zur Verfügung, falls diese ebenfalls übertragen oder angezeigt werden sollen. [] Diese formatierten Daten werden dann wie schon bei der Temperaturübertragung in ein LoRa-Paket umgewandelt und mit *node.send(data)* versendet. Danach startet der Timer erneut, um die Übertragung regelmäßig fortzusetzen. In der folgenden Abbildung ist zu sehen, dass die ID 291 empfangen wurde und sich die Daten mit dem Verstellen des Potentiometers ändern.

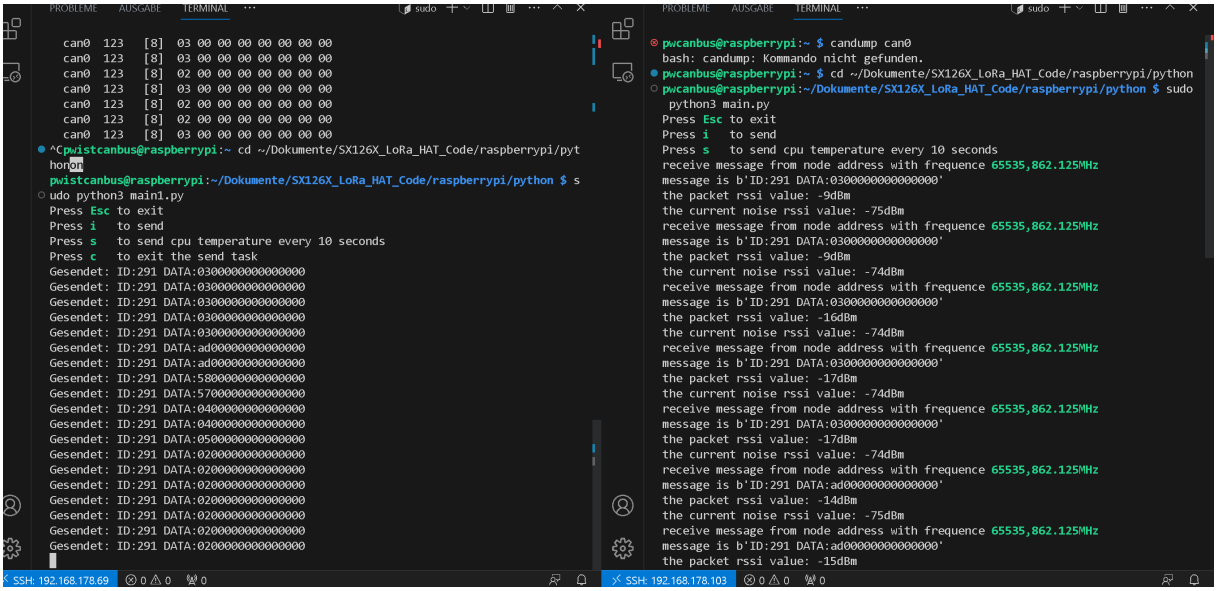

Abbildung 12 Test der CAN-Kommunikation

4.4 PCB-Design für das Fahrzeug CAN-Modul

Im nächsten Schritt soll mit Hilfe von *KiCad* ein eigenes PCB entworfen werden, das es ermöglicht, das Sendemodul dauerhaft im Fahrzeug zu verbauen.
Abbildung 13 zeigt den vollständigen Schaltplan des Systems, auf dem das spätere Leiterplattendesign erstellt wird. Der Plan umfasst die Spannungsversorgung, den CAN-Transceiver (SN65HVD230), den CAN-Controller (MCP2515) sowie die Verbindung zum Raspberry Pi über die SPI-Schnittstelle.

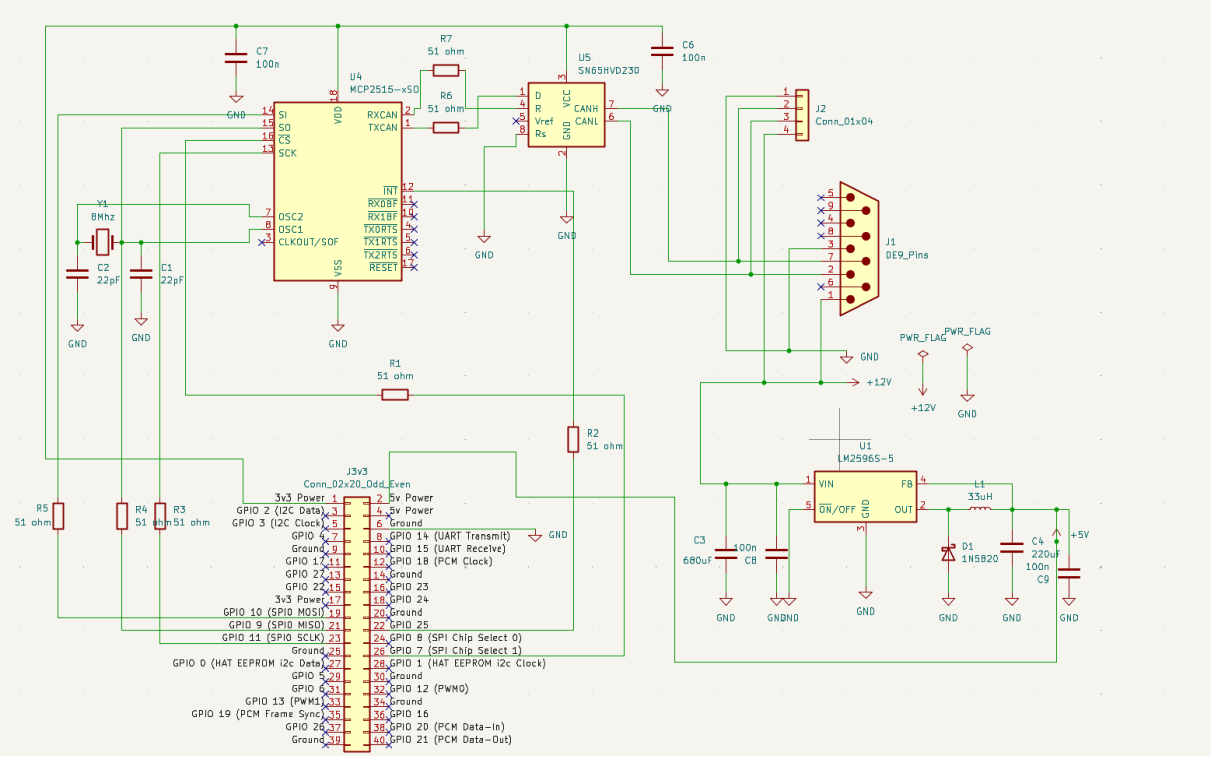

Abbildung 13 vollständiger Schaltplan erstellt mit KiCad

Da im Auto eine 12V Bordspannung anliegt, muss diese gemäß den Anforderungen des verwendeten Mikrocontrollers auf 5V bei bis zu 2 Ampere heruntergeregelt werden.

Für den Betrieb im Fahrzeug sind folgende Anschlüsse notwendig:

– 12V Eingangsspannung

– Masse (GND)

– CAN High (CANH)

– CAN Low (CANL)

Diese Eingänge sollen auf dem PCB entsprechend ausgeführt werden, sodass das Modul direkt an den fahrzeugeigenen CAN-Bus angeschlossen und mit Strom versorgt werden kann.

Um die GPIO-Leitungen des Raspberry Pi zuverlässig vor Spannungsüberschreitungen und Störimpulsen aus dem Fahrzeug-CAN-Bus zu schützen, wurde bewusst der SN65HVD230 CAN-Transceiver ausgewählt. Dieser arbeitet ausschließlich mit einer Versorgungsspannung

von 3,3 V und gibt die empfangenen CAN-Signale ebenfalls als 3,3-V-Logiksignale aus. Dadurch ist sichergestellt, dass keine höheren Spannungspegel an die GPIO-Pins des Raspberry Pi gelangen können.

In Kombination mit dem Transceiver wird der MCP2515-xSO CAN-Controller verwendet, der ebenfalls mit 3,3 V betrieben wird. Durch die gleiche Betriebsspannung und Logikpegel ist eine direkte SPI-Kommunikation mit dem Raspberry Pi problemlos möglich. Die Auswahl dieser beiden Komponenten stellt sicher, dass keine Pegelwandler oder Schutzbeschaltungen notwendig sind, was den Aufbau und zuverlässig macht. Dies ist besonders wichtig, da die GPIOs des Raspberry Pi nicht 5V-tolerant sind und bereits bei kurzzeitiger Überschreitung der maximal zulässigen Spannung beschädigt werden können.

Für die physische Anbindung an den fahrzeugeigenen CAN-Bus ist ein Sub-D 9-poliger Anschluss (DB9) vorgesehen, der sich in der Automobil- und Industrieelektronik bewährt hat und eine besonders robuste und steckerfeste Verbindung gewährleistet. Zusätzlich steht eine 4-polige Klemmleiste zur Verfügung. Über diese lassen sich die Leitungen für 12 V, Masse, CAN_H und CAN_L direkt verdrahten, sodass das Modul auch ohne DB9-Stecker in den CAN-Bus integriert werden kann. Auf diese Weise bietet die Platine maximale Flexibilität bei der Fahrzeuginstallation und sorgt gleichzeitig für optimalen Schutz der empfindlichen Pi-GPIOs.

Signal- und Versorgungselektronik auf dem PCB

Zur Sicherstellung einer stabilen und störungsarmen Signalübertragung auf dem CAN-Bus wurden in den Signalleitungen jeweils 51Ω Widerstände eingesetzt. Diese Widerstände dienen der Impedanz Anpassung und minimieren Signalreflexionen, die andernfalls zu Datenfehlern führen können. Zusätzlich sind an den Ein- und Ausgängen jeweils 100-nF-Kondensatoren platziert, die als Entkopplungs- und Filterelemente wirken. Obwohl diese Kondensatoren nicht explizit im Datenblatt des CAN-Transceivers vorgesehen sind, tragen sie in der Praxis entscheidend zur Unterdrückung hochfrequenter Störimpulse bei, die insbesondere in der Fahrzeugumgebung aufgrund elektromagnetischer Einflüsse auftreten können.

Darüber hinaus wurde beim Leiterplattendesign darauf geachtet, dass die Signalleitungen möglichst ohne scharfe 90°-Winkel geführt werden. Stattdessen kommen diese in 45°-Winkel zum Einsatz, um Signalreflexionen und Störungen weiter zu minimieren.

Der notwendige Abschlusswiderstand von 120 Ω wird nicht auf der Platine integriert. Er ist bereits in der Fahrzeug-CAN-Bus-Infrastruktur zur Vermeidung von Reflexionen und zur Sicherstellung der CAN-Kommunikation vorgesehen. Durch diese Maßnahmen wird eine robuste und störungsfreie Übertragung der CAN-Signale gewährleistet, was die Zuverlässigkeit der Kommunikation im Fahrzeug verbessert.

In der nachfolgenden Abbildung ist das 3D Design für die Aufsteckplatine dargestellt. Die Maße sowie die Position der Befestigungsbohrungen wurden dem offiziellen Datenblatt des Raspberry Pi entnommen. Auf dieser Grundlage wurde ein passender Footprint für die Platine erstellt, der eine Montage auf dem Raspberry Pi ermöglicht. Zum Schluss wurden die Gerber-Dateien erzeugt und für die Fertigung der Platine bereitgestellt.

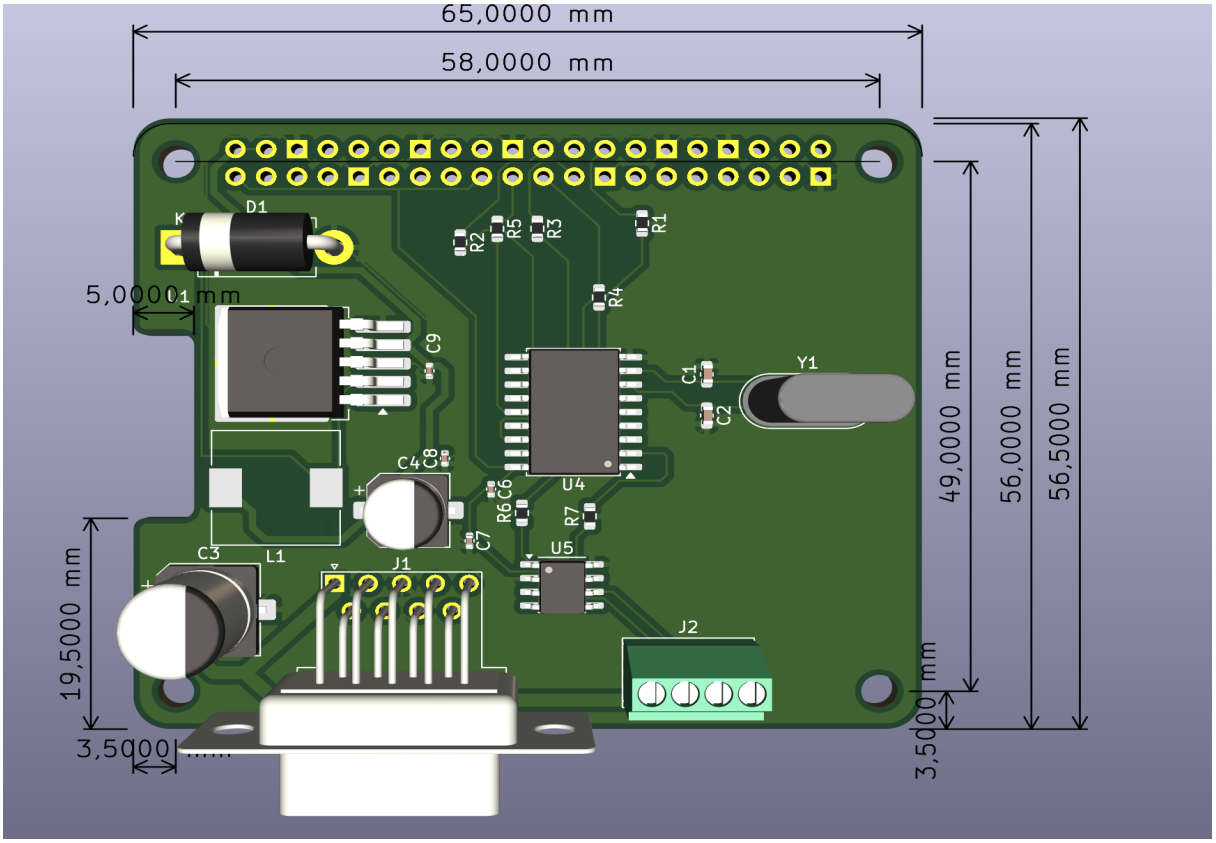

Abbildung 14 3D-Ansicht des Leiterplattenlayouts für die Aufsteckplatine

5 Diskussion der Ergebnisse

Die Umsetzung der CAN-Bus-Kommunikation in Kombination mit einem Raspberry Pi als Mikrocontroller sowie der Einsatz eines LoRa-Moduls zur drahtlosen Datenübertragung zeigt sich im Gesamtkontext als technisch sinnvolle, kosteneffiziente und zukunftsfähige Lösung.

5.1 Raspberry Pi als Mikrocontroller im CAN-Bus-System

Der Einsatz des Raspberry Pi als zentrale Steuereinheit im CAN-Bus-Netzwerk bietet mehrere Vorteile. Durch die Unterstützung gängiger Protokolle wie SPI und I²C sowie die Verfügbarkeit zahlreicher Bibliotheken und Tools unter Linux ist der Raspberry Pi besonders flexibel und einfach zu integrieren. In Verbindung mit dem MCP2515-xSO CAN-Controller und dem SN65HVD230 Transceiver lässt sich eine Kommunikation mit dem fahrzeuginternen CAN-Bus aufbauen.

Ein wesentlicher Vorteil des Raspberry Pi ist seine hohe Rechenleistung im Vergleich zu klassischen Mikrocontrollern (z. B. der STM32- oder ATmega-Familie). Diese Rechenleistung ermöglicht nicht nur eine schnelle Verarbeitung eingehender CAN-Daten, sondern auch deren Analyse, Visualisierung und ggf. Übertragung an weitere Systeme – wie beispielsweise ein LoRa-Gateway oder einen Datenspeicher.

Darüber hinaus profitiert das System vom Linux, das eine einfache Entwicklung, Wartung und Erweiterung erlaubt. Insbesondere im Hochschulumfeld, wie bei der Formula Student, ist dies ein bedeutender Vorteil, da Studierende ohne tiefgehende Embedded-Kenntnisse schnell produktiv arbeiten können.

5.2 Vorteile von LoRa gegenüber WLAN und Mobilfunk (SIM)

Für die drahtlose Datenübertragung wurde in diesem Projekt bewusst auf LoRa gesetzt – und nicht auf WLAN oder Mobilfunktechnologien wie LTE. Der wesentliche Vorteil von LoRa liegt in seiner hohen Reichweite bei gleichzeitig sehr geringem Energieverbrauch. Besonders im Kontext eines Rennfahrzeugs wie bei der Formula Student, das sich über ein weites Gelände bewegen kann, bietet LoRa eine deutlich zuverlässigere Abdeckung als WLAN, das auf die Reichweite eines lokalen Routers beschränkt ist.

Im Vergleich zu Mobilfunklösungen über SIM-Karten hat LoRa weitere Vorteile: SIM-basierte Übertragungen sind mit laufenden Kosten verbunden, benötigen oft stabile Netzabdeckung (was auf abgesperrten Testgeländen oder Rennstrecken nicht immer gegeben ist) und haben höhere Latenzen. LoRa hingegen arbeitet lizenzfrei im ISM-Band, verursacht keine Verbindungskosten und benötigt keine Netzbetreiberinfrastruktur. Die Nutzung von LoRaWAN oder

auch eines einfachen Punkt-zu-Punkt-Setups genügt, um zuverlässig Daten in Echtzeit zu übertragen.

Für die Anforderungen der Formula Student, in der insbesondere Robustheit, Echtzeitfähigkeit und Mobilität im Vordergrund stehen, ist LoRa daher eine besonders geeignete Technologie. Auch das stark eingeschränkte Zeit- und Budgetumfeld bei studentischen Rennteams spricht gegen komplexe Mobilfunklösungen und für den Einsatz leicht integrierbarer LoRa-Module.

5.3 Gesamtsystem im Formula-Student-Kontext

In der Anwendung innerhalb eines Formula-Student-Rennwagens erweist sich die Kombination aus CAN-Bus, Raspberry Pi und LoRa als besonders praxistauglich. Das Fahrzeug nutzt bereits einen CAN-Bus zur Steuerung und Überwachung zentraler Komponenten wie Motorregelung, Batterieüberwachung und Sensorik. Die hier vorgestellte Lösung erlaubt es, diese Daten auszulesen und ohne Eingriff in die bestehende Steuerungseinheit sicher zu übertragen.

Gleichzeitig stellt der Raspberry Pi eine vielseitige Plattform dar, auf der Auswertungsalgorithmen, Datenlogging oder sogar eine einfache Benutzeroberfläche laufen können – ohne dass dafür zusätzliche Hardware nötig ist.

Durch die hohe Modularität kann das entwickelte PCB dauerhaft im Fahrzeug verbleiben, ohne bei Änderungen im Fahrzeugdesign ersetzt werden zu müssen. Die Wahl robuster Anschlüsse (DB9, Klemmleiste) und industriell bewährter Schnittstellen garantiert auch im motorsportlichen Umfeld eine störungsfreie Funktion.

5.4 Zusammenfassung

Im Rahmen des Projekts wurden die definierten Teilaufgaben, wie in Kapitel 1.3 dargestellt, mit unterschiedlichen Graden der Vollständigkeit bearbeitet:

- **Entwicklung und Erprobung der Datenerfassungsplatine**:
  Die Platine wurde erfolgreich entwickelt und aufgebaut. Die Spannungsversorgung über einen 12V-auf-5V-Spannungsregler funktionierte zuverlässig. Allerdings konnte keine stabile Kommunikation zwischen dem Raspberry Pi und dem MCP2515-CAN-Controller hergestellt werden, sodass die Funktionalität der Platine nicht abschließend bestätigt werden konnte.

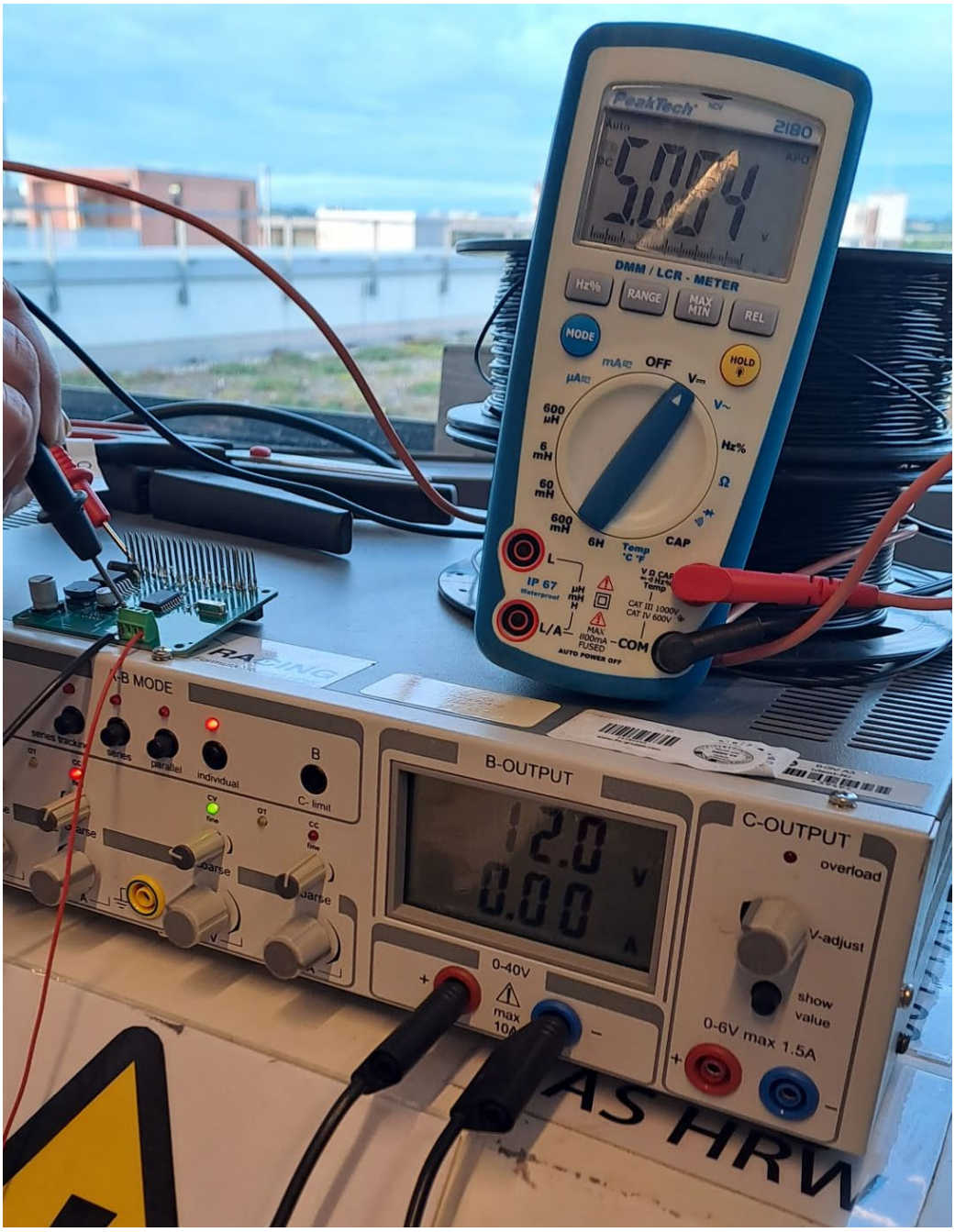

Abbildung 15 Spannungsregelung von 12 Volt auf 5 Volt

- **Integration einer stabilen drahtlosen Schnittstelle**:
  Die drahtlose Kommunikation über LoRa wurde erfolgreich realisiert. Die grundlegende Funktionalität konnte nachgewiesen werden. Eine genaue Reichweitenanalyse wurde jedoch nicht durchgeführt, sodass das Potenzial der Technologie noch nicht vollständig ausgeschöpft wurde.

- **Design und Implementierung eines Echtzeit-Interfaces**:
  Die empfangenen CAN-Daten werden in hexadezimaler Form ausgegeben. Ein

grafisches oder interpretiertes Interface zur benutzerfreundlichen Echtzeitdarstellung wurde nicht umgesetzt.

- **Erweiterbarkeit für neue Sensoren**:
  Die Softwarearchitektur wurde grundsätzlich so gestaltet, dass eine Erweiterung möglich ist. Es wurde jedoch nicht praktisch getestet, wie viele Sensoren sich gleichzeitig auslesen und zuverlässig verarbeiten lassen.

Die Ergebnisse belegen, dass die gewählte Systemarchitektur den Anforderungen gerecht wird. Sie setzt sich aus dem CAN-Bus, einem Raspberry Pi sowie dem LoRa-Funkmodul zusammen. Diese Kombination ermöglicht eine effiziente, zuverlässige und kostengünstige Lösung für die Fahrzeugkommunikation im Rahmen der Formula Student. Gleichzeitig schafft die drahtlose Datenübertragung per LoRa eine stabile Grundlage für die Technische Analyse im studentischen Rennsport.

5.5 Ausblick

Im Rahmen der vorliegenden Bachelorarbeit wurden nur die dargestellten Punkte bearbeitet. Mit der hier eingesetzten LoRa-Kommunikation wurde ein Ansatz aus dem Bereich IoT genutzt, um gesammelte Fahrzeugdaten drahtlos an ein entferntes System zu übertragen. Gerade im Umfeld der Formula Student kann diese Methode bei bestimmten Sensoren besonders sinnvoll sein – etwa zur Überwachung der Motortemperatur während der Testfahrten. So lassen sich Fehlerquellen frühzeitig erkennen oder bestimmte Fahrzustände analysieren.

Es sollten jedoch weitere wichtige Punkte im Rahmen dieses Projekts geforscht werden. Wie bereits in Kapitel 2 beschrieben, ist ein weiterer Schritt in der Entwicklung von Datalogger der Einsatz von Edge Computing. Hierbei werden nicht nur Daten gesammelt, sondern lokal ausgewertet und es kann automatisiert auf kritische Zustände reagiert werden. Dies eröffnet neue Möglichkeiten: beispielsweise könnten bei Überhitzung einzelner Akkuzellen die Akku-Pads automatisch abgeschaltet oder gekühlt werden, um die Funktionstüchtigkeit zu gewährleisten.

Solche Funktionen machen aus einem einfachen Datalogger ein intelligentes Überwachungssystem, das nicht nur Daten bereitstellt, sondern auch aktiv zur Sicherheit und Zuverlässigkeit beiträgt.

Abkürzungsverzeichnis

ACK	Acknowledgement
CAN-BUS	Controller Area Network Bussystem
CRC	Cyclic Redundancy Check
CPU	Central Processing Unit
GPIO	General Purpose Input/Output
HDMI	High-Definition Multimedia Interface
IoT	Internet der Dinge
LAN	Local Area Network
LIN-Bus	Local Interconnect Network
LoRaWAN	Long Range Wide Area Network
LoRa	Long Range
LTE	Long Term Evolution
MCP	Microchannel Plate
PCB-Design	Printed Circuit Board
SAE	Society of Automotive Engineers
SIM-Karten	Subscriber Identity Module Card

Tabellenverzeichnis

Tabelle 1 Struktur eines klassischen CAN-Datenframes [13]..... 7
Tabelle 2 Pinbelegung des TJA1050 CAN-Transceivers mit Funktionsbeschreibung [21] 14
Tabelle 3 GPIO-Zuordnung und Anschlussplan für Raspberry Pi und MCP2515 [25] 18

Abbildungsverzeichnis

Abbildung 1 Bosch Motorsport C70 Datalogger [14] 9
Abbildung 2 Raspberry Pi B+ V1.2 mit 40-Pin-GPIO, HDMI, LAN und 4× USB 2.0 [18] 11
Abbildung 3 GPIO-Pinbelegung Raspberry Pi [19] 12
Abbildung 4 CAN-Bus-Modul mit MCP2515 und TJA1050 [20] 13
Abbildung 5 Vollständiges CAN-Frame mit der Umwandlung in CAN RX-Signal [22] 15
Abbildung 6 Schaltplan des vorgefertigten CAN-Moduls [24] 17
Abbildung 7 Aufbau der Testschaltung mit realistischem CAN-Bus-System 20
Abbildung 8 Auswertung der CAN-Signale in hexadezimaler Darstellung 21
Abbildung 9 SX1262 868M LoRa HAT [26] 22
Abbildung 10 Konfiguration der Spannungsversorgung über interne Klemme [26] 24
Abbildung 11 Testbetrieb der LoRa Kommunikation mit Referenzcode 26
Abbildung 12 Test der CAN-Kommunikation 27
Abbildung 13 vollständiger Schaltplan erstellt mit KiCad 28
Abbildung 14 3D-Ansicht des Leiterplattenlayouts für die Aufsteckplatine 30
Abbildung 15 Spannungsregelung von 12 Volt auf 5 Volt 33

Literaturverzeichnis

[1] University of Pittsburgh. Formula SAE Team – About. 2024.
URL: https://www.engineering.pitt.edu/subsites/student-orgs/fsae/about/

[2] SAE International. *Offizielle Website.* 2025.
URL: https://www.sae.org/

[3] FSG Competition Handbook 2017

[4] Formula Student Germany. Autonomous Driving at Formula Student Germany 2017. 12. August 2016.
URL: https://www.formulastudent.de/pr/news/details/article/autonomous-driving-at-formula-student-germany-2017/

[5] Delphin Technology AG. Was ist ein Datenlogger und wie funktioniert er? 20. November 2024.
URL: https://www.delphin.de/was-ist-ein-datenlogger-und-wie-funktioniert-er/

[6] Delphin Technology AG**.** *Datenlogger.2025*
URL: https://www.delphin.de/loesungen/datenlogger/

[7] B. Merabtane and N. Benabadji, “Design and Implementation of a Multi Connectivity Datalogger using SD Card Storage for Photovoltaic System Monitoring,” in 2023 International Conference on Control, Electronics, Energy and Mechanical Engineering (IC2EM), 2023. IEEE, 2024. DOI: 10.1109/IC2EM59347.2023.10419399.

[8] Poornima H. S**.** *CAN Bus Integration for Enhanced Industrial Control and Safety*. May 2024. URL: https://www.researchgate.net/publication/380791576_CAN_Bus_Integration_for_Enhanced_Industrial_Control_and_Safety.

[9] Sudarshan Pant and Sangdon Lee. “Design and Implementation of a CAN Data Analysis Test Bench based on Raspberry Pi”. In: *Journal of Multimedia Information System*, Vol. 6, No. 4, Dec. 2019, pp. 239–244. DOI: 10.33851/JMIS.2019.6.4.239.

[10] P. Raichl and P. Marcon. “An IoT-Based Datalogger of Environmental Values”. In: *2021 13th International Conference on Measurement*. IEEE, May 2021. DOI: 10.23919/Measurement52780.2021.9446796.

[11] I. Ficili, M. Giacobbe, G. Tricomi, and A. Puliafito. “From Sensors to Data Intelligence: Leveraging IoT, Cloud, and Edge Computing with AI”. In: *Sensors*, vol. 25, no. 6, Mar. 2025. DOI: 10.3390/s25061763.

[12] Assured Systems GmbH. *Enträtseln der Leistungsfähigkeit von Controller Area Network (CAN)-Technologie.* 2023.
URL**:**https://www.assured-systems.com/de/entratseln-der-leistungsfahigkeit-von-controller-area-network-can-technologie-artikel/

[13] Prof. Dr. Lothar U. Kempen. *Nachrichtentechnik–Computernetze II: Schnittstellen und Feldbusse – 07 Schnittstellen und Feldbusse.Dezember* 2022.

[14] Bosch Motorsport. *Data Logger C 70 – Dataheet.* 2022.

[15] STMicroelectronics. *STM32F103C8 – Microcontroller – Datasheet.* 2021.

[16] ATmega328/P – Microcontroller – Datasheet. 2016.

[17] Raspberry Pi Foundation. *Raspberry Pi 4 Model B – Technische Dokumentation.* 2025.

[18] Reichelt Elektronik. Raspberry Pi B – 4x USB 2.0, 40pin GPIO. 2024. URL: https://www.reichelt.de/de/de/shop/produkt/raspberry_pi_b_4x_usb_2_0_40pin_gpio-146194.

[19] Indibit. Raspberry Pi – die GPIO-Schnittstelle: Grundlagen & Belegung. 2023. URL: https://indibit.de/raspberry-pi-die-gpio-schnittstelle-grundlagenbelegung/.

[20] AZ-Delivery. MCP2515 CAN-Bus Modul, Draufsicht. 2024. URL: https://www.az-delivery.de/products/mcp2515-can-bus-modul.

[21] TJA1050, High-speed CAN transceiver. Rev. 04 – 22 April 2008. NXP Semiconductors. 2008

[22] Toby. *38. Performing CAN Frame Analysis (Including Stuff Bit).* Medium, 01.10.2023. URL:https://medium.com/@johnehk86/38-performing-can-frame-analysis-including-stuff-bit-e96079a0fdce

[23] Crystal Oscillator Basics and Crystal Selection for rfPIC™ and PICmicro® Devices. Application Note AN826, Rev. DS00826A. Microchip Technology Inc., 2002.

[24] Robu.in. Schematic of MCP2515 CAN Module TJA1050 Receiver SPI 51. Electronics Hardware.2017.url:https://robu.in/wp-content/uploads/2017/09/CAN2515_Schematic.png

[25] 38-3D. Verwenden des MCP2515 mit dem Raspberry Pi. Electronics Hardware. 2025. url:https://38-3d.co.uk/de/blogs/blog/verwenden-des-mcp2515-mit-dem-raspberry-pi

[26] Waveshare Wiki, zuletzt geändert am 5. Januar 2022. URL: https://www.waveshare.com/wiki/SX1262_868M_LoRa_HAT

Anhang

Referenzcode LoRa :Main.Py mit CPU abfrage

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

#
#    this is an UART-LoRa device and thers is an firmware on Module
#    users can transfer or receive the data directly by UART and dont
#    need to set parameters like coderate,spread factor,etc.
#    |========================================== |
#    |   It does not suport LoRaWAN protocol !!!   |
#    | ==========================================|
#
#    This script is mainly for Raspberry Pi 3B+, 4B, and Zero series
#    Since PC/Laptop does not have GPIO to control HAT, it should be con-
figured by
#    GUI and while setting the jumpers,
#    Please refer to another script pc_main.py
#

import sys
import sx126x
import threading
import time
import select
import termios
import tty
from threading import Timer

old_settings = termios.tcgetattr(sys.stdin)
tty.setcbreak(sys.stdin.fileno())


#
#    Need to disable the serial login shell and have to enable serial in-
terface
#    command `sudo raspi-config`
#    More details: see https://github.com/Mith-
unHub/LoRa/blob/main/Basic%20Instruction.md
#
#    When the LoRaHAT is attached to RPi, the M0 and M1 jumpers of HAT
should be removed.
#


#    The following is to obtain the temprature of the RPi CPU
def get_cpu_temp():
    tempFile = open( "/sys/class/thermal/thermal_zone0/temp" )
    cpu_temp = tempFile.read()
    tempFile.close()
    return float(cpu_temp)/1000

#    serial_num
#        PiZero, Pi3B+, and Pi4B use "/dev/ttyS0"
```

```
#
#    Frequency is [850 to 930], or [410 to 493] MHz
#
#    address is 0 to 65535
#        under the same frequence,if set 65535,the node can receive
#        messages from another node of address is 0 to 65534 and similarly,
#        the address 0 to 65534 of node can receive messages while
#        the another note of address is 65535 sends.
#        otherwise two node must be same the address and frequence
#
#    The tramsmit power is {10, 13, 17, and 22} dBm
#
#    RSSI (receive signal strength indicator) is {True or False}
#        It will print the RSSI value when it receives each message
#

# node = sx126x.sx126x(serial_num =
"/dev/ttyS0",freq=433,addr=0,power=22,rssi=False,air_speed=2400,re-
lay=False)
node = sx126x.sx126x(serial_num =
"/dev/ttyS0",freq=868,addr=0,power=22,rssi=True,air_speed=2400,relay=False)

def send_deal():
    get_rec = ""
    print("")
    print("input a string such as \033[1;32m0,868,Hello World\033[0m,it
will send `Hello World` to lora node device of address 0 with 868M ")
    print("please input and press Enter key:",end='',flush=True)

    while True:
        rec = sys.stdin.read(1)
        if rec != None:
            if rec == '\x0a': break
            get_rec += rec
            sys.stdout.write(rec)
            sys.stdout.flush()

    get_t = get_rec.split(",")

    offset_frequence = int(get_t[1])-(850 if int(get_t[1])>850 else 410)
    #
    # the sending message format
    #
    #         receiving node              receiving node
receiving node           own high 8bit           own low 8bit
own
    #         high 8bit address           low 8bit address
frequency                address                 address
frequency             message payload
    data = bytes([int(get_t[0])>>8]) + bytes([int(get_t[0])&0xff]) +
bytes([offset_frequence]) + bytes([node.addr>>8]) + bytes([node.addr&0xff])
+ bytes([node.offset_freq]) + get_t[2].encode()

    node.send(data)
    print('\x1b[2A',end='\r')
    print(" "*200)
    print(" "*200)
    print(" "*200)
    print('\x1b[3A',end='\r')
```

```
def send_cpu_continue(continue_or_not = True):
    if continue_or_not:
        global timer_task
        global seconds
        #
        # boarcast the cpu temperature at 868.125MHz
        #
        data = bytes([255]) + bytes([255]) + bytes([18]) + bytes([255]) +
bytes([255]) + bytes([12]) + "CPU Temperature:".en-
code()+str(get_cpu_temp()).encode()+" C".encode()
        node.send(data)
        time.sleep(0.2)
        timer_task = Timer(seconds,send_cpu_continue)
        timer_task.start()
    else:
        data = bytes([255]) + bytes([255]) + bytes([18]) + bytes([255]) +
bytes([255]) + bytes([12]) + "CPU Temperature:".en-
code()+str(get_cpu_temp()).encode()+" C".encode()
        node.send(data)
        time.sleep(0.2)
        timer_task.cancel()
        pass

try:
    time.sleep(1)
    print("Press \033[1;32mEsc\033[0m to exit")
    print("Press \033[1;32mi\033[0m   to send")
    print("Press \033[1;32ms\033[0m   to send cpu temperature every 10 sec-
onds")

    # it will send rpi cpu temperature every 10 seconds
    seconds = 10

    while True:

        if select.select([sys.stdin], [], [], 0) == ([sys.stdin], [], []):
            c = sys.stdin.read(1)

            # dectect key Esc
            if c == '\x1b': break
            # dectect key i
            if c == '\x69':
                send_deal()
            # dectect key s
            if c == '\x73':
                print("Press \033[1;32mc\033[0m   to exit the send task")
                timer_task = Timer(seconds,send_cpu_continue)
                timer_task.start()

                while True:
                    if sys.stdin.read(1) == '\x63':
                        timer_task.cancel()
                        print('\x1b[1A',end='\r')
                        print(" "*100)
                        print('\x1b[1A',end='\r')
                        break

            sys.stdout.flush()

        node.receive()
```

```
        # timer,send messages automatically

except:
    termios.tcsetattr(sys.stdin, termios.TCSADRAIN, old_settings)
    # print('\x1b[2A',end='\r')
    # print(" "*100)
    # print(" "*100)
    # print('\x1b[2A',end='\r')

termios.tcsetattr(sys.stdin, termios.TCSADRAIN, old_settings)
# print('\x1b[2A',end='\r')
# print(" "*100)
# print(" "*100)
# print('\x1b[2A',end='\r')
```

Referenzcode sx126x

```
import RPi.GPIO as GPIO
import serial
import time

class sx126x:

    M0 = 22
    M1 = 27
    # if the header is 0xC0, then the LoRa register settings dont lost when
it poweroff, and 0xC2 will be lost.
    # cfg_reg =
[0xC0,0x00,0x09,0x00,0x00,0x00,0x62,0x00,0x17,0x43,0x00,0x00]
    cfg_reg = [0xC2,0x00,0x09,0x00,0x00,0x00,0x62,0x00,0x12,0x43,0x00,0x00]
    get_reg = bytes(12)
    rssi = False
    addr = 65535
    serial_n = ""
    addr_temp = 0

    #
    # start frequence of two lora module
    #
    # E22-400T22S           E22-900T22S
    # 410~493MHz      or    850~930MHz
    start_freq = 850

    #
    # offset between start and end frequence of two lora module
    #
    # E22-400T22S           E22-900T22S
    # 410~493MHz      or    850~930MHz
    offset_freq = 18

    # power = 22
```

```
    # air_speed =2400

    SX126X_UART_BAUDRATE_1200 = 0x00
    SX126X_UART_BAUDRATE_2400 = 0x20
    SX126X_UART_BAUDRATE_4800 = 0x40
    SX126X_UART_BAUDRATE_9600 = 0x60
    SX126X_UART_BAUDRATE_19200 = 0x80
    SX126X_UART_BAUDRATE_38400 = 0xA0
    SX126X_UART_BAUDRATE_57600 = 0xC0
    SX126X_UART_BAUDRATE_115200 = 0xE0

    SX126X_PACKAGE_SIZE_240_BYTE = 0x00
    SX126X_PACKAGE_SIZE_128_BYTE = 0x40
    SX126X_PACKAGE_SIZE_64_BYTE = 0x80
    SX126X_PACKAGE_SIZE_32_BYTE = 0xC0

    SX126X_Power_22dBm = 0x00
    SX126X_Power_17dBm = 0x01
    SX126X_Power_13dBm = 0x02
    SX126X_Power_10dBm = 0x03

    lora_air_speed_dic = {
        1200:0x01,
        2400:0x02,
        4800:0x03,
        9600:0x04,
        19200:0x05,
        38400:0x06,
        62500:0x07
    }

    lora_power_dic = {
        22:0x00,
        17:0x01,
        13:0x02,
        10:0x03
    }

    lora_buffer_size_dic = {
        240:SX126X_PACKAGE_SIZE_240_BYTE,
        128:SX126X_PACKAGE_SIZE_128_BYTE,
        64:SX126X_PACKAGE_SIZE_64_BYTE,
        32:SX126X_PACKAGE_SIZE_32_BYTE
    }

    def __init__(self,serial_num,freq,addr,power,rssi,air_speed=2400,\
                 net_id=0,buffer_size = 240,crypt=0,\
                 relay=False,lbt=False,wor=False):
        self.rssi = rssi
        self.addr = addr
        self.freq = freq
        self.serial_n = serial_num
        self.power = power
        # Initial the GPIO for M0 and M1 Pin
        GPIO.setmode(GPIO.BCM)
        GPIO.setwarnings(False)
        GPIO.setup(self.M0,GPIO.OUT)
        GPIO.setup(self.M1,GPIO.OUT)
        GPIO.output(self.M0,GPIO.LOW)
        GPIO.output(self.M1,GPIO.HIGH)
```

```
        # The hardware UART of Pi3B+,Pi4B is /dev/ttyS0
        self.ser = serial.Serial(serial_num,9600)
        self.ser.flushInput()

self.set(freq,addr,power,rssi,air_speed,net_id,buffer_size,crypt,re-
lay,lbt,wor)

    def set(self,freq,addr,power,rssi,air_speed=2400,\
            net_id=0,buffer_size = 240,crypt=0,\
            relay=False,lbt=False,wor=False):
        self.send_to = addr
        self.addr = addr
        # We should pull up the M1 pin when sets the module
        GPIO.output(self.M0,GPIO.LOW)
        GPIO.output(self.M1,GPIO.HIGH)
        time.sleep(0.1)

        low_addr = addr & 0xff
        high_addr = addr >> 8 & 0xff
        net_id_temp = net_id & 0xff
        if freq > 850:
            freq_temp = freq - 850
            self.start_freq = 850
            self.offset_freq = freq_temp
        elif freq >410:
            freq_temp = freq - 410
            self.start_freq  = 410
            self.offset_freq = freq_temp

        air_speed_temp = self.lora_air_speed_dic.get(air_speed,None)
        # if air_speed_temp != None

        buffer_size_temp = self.lora_buffer_size_dic.get(buffer_size,None)
        # if air_speed_temp != None:

        power_temp = self.lora_power_dic.get(power,None)
        #if power_temp != None:

        if rssi:
            # enable print rssi value
            rssi_temp = 0x80
        else:
            # disable print rssi value
            rssi_temp = 0x00

        # get crypt
        l_crypt = crypt & 0xff
        h_crypt = crypt >> 8 & 0xff

        if relay==False:
            self.cfg_reg[3] = high_addr
            self.cfg_reg[4] = low_addr
            self.cfg_reg[5] = net_id_temp
            self.cfg_reg[6] = self.SX126X_UART_BAUDRATE_9600 +
air_speed_temp
            #
            # it will enable to read noise rssi value when add 0x20 as fol-
low
            #
```

```
            self.cfg_reg[7] = buffer_size_temp + power_temp + 0x20
            self.cfg_reg[8] = freq_temp
            #
            # it will output a packet rssi value following received message
            # when enable eighth bit with 06H register(rssi_temp = 0x80)
            #
            self.cfg_reg[9] = 0x43 + rssi_temp
            self.cfg_reg[10] = h_crypt
            self.cfg_reg[11] = l_crypt
        else:
            self.cfg_reg[3] = 0x01
            self.cfg_reg[4] = 0x02
            self.cfg_reg[5] = 0x03
            self.cfg_reg[6] = self.SX126X_UART_BAUDRATE_9600 +
air_speed_temp
            #
            # it will enable to read noise rssi value when add 0x20 as fol-
low
            #
            self.cfg_reg[7] = buffer_size_temp + power_temp + 0x20
            self.cfg_reg[8] = freq_temp
            #
            # it will output a packet rssi value following received message
            # when enable eighth bit with 06H register(rssi_temp = 0x80)
            #
            self.cfg_reg[9] = 0x03 + rssi_temp
            self.cfg_reg[10] = h_crypt
            self.cfg_reg[11] = l_crypt
        self.ser.flushInput()

        for i in range(2):
            self.ser.write(bytes(self.cfg_reg))
            r_buff = 0
            time.sleep(0.2)
            if self.ser.inWaiting() > 0:
                time.sleep(0.1)
                r_buff = self.ser.read(self.ser.inWaiting())
                if r_buff[0] == 0xC1:
                    pass
                    # print("parameters setting is :",end='')
                    # for i in self.cfg_reg:
                        # print(hex(i),end=' ')

                    # print('\r\n')
                    # print("parameters return is  :",end='')
                    # for i in r_buff:
                        # print(hex(i),end=' ')
                    # print('\r\n')
                else:
                    pass
                    #print("parameters setting fail :",r_buff)
                    break
            else:
                print("setting fail,setting again")
                self.ser.flushInput()
                time.sleep(0.2)
                print('\x1b[1A',end='\r')
                if i == 1:
                    print("setting fail,Press Esc to Exit and run again")
                    # time.sleep(2)
```

```
                    # print('\x1b[1A',end='\r')

        GPIO.output(self.M0,GPIO.LOW)
        GPIO.output(self.M1,GPIO.LOW)
        time.sleep(0.1)

    def get_settings(self):
        # the pin M1 of lora HAT must be high when enter setting mode and
get parameters
        GPIO.output(M1,GPIO.HIGH)
        time.sleep(0.1)

        # send command to get setting parameters
        self.ser.write(bytes([0xC1,0x00,0x09]))
        if self.ser.inWaiting() > 0:
            time.sleep(0.1)
            self.get_reg = self.ser.read(self.ser.inWaiting())

        # check the return characters from hat and print the setting param-
eters
        if self.get_reg[0] == 0xC1 and self.get_reg[2] == 0x09:
            fre_temp = self.get_reg[8]
            addr_temp = self.get_reg[3] + self.get_reg[4]
            air_speed_temp = self.get_reg[6] & 0x03
            power_temp = self.get_reg[7] & 0x03

            print("Frequence is {0}.125MHz.",fre_temp)
            print("Node address is {0}.",addr_temp)
            print("Air speed is {0} bps"+
lora_air_speed_dic.get(None,air_speed_temp))
            print("Power is {0} dBm" + lora_power_dic.get(None,power_temp))
            GPIO.output(M1,GPIO.LOW)

#
# the data format like as following
# "node address,frequence,payload"
# "20,868,Hello World"
    def send(self,data):
        GPIO.output(self.M1,GPIO.LOW)
        GPIO.output(self.M0,GPIO.LOW)
        time.sleep(0.1)

        self.ser.write(data)
        # if self.rssi == True:
            # self.get_channel_rssi()
        time.sleep(0.1)


    def receive(self):
        if self.ser.inWaiting() > 0:
            time.sleep(0.5)
            r_buff = self.ser.read(self.ser.inWaiting())

            print("receive message from node address with fre-
quence\033[1;32m
%d,%d.125MHz\033[0m"%((r_buff[0]<<8)+r_buff[1],r_buff[2]+self.start_freq),e
nd='\r\n',flush = True)
            print("message is "+str(r_buff[3:-1]),end='\r\n')

            # print the rssi
```

```
            if self.rssi:
                # print('\x1b[3A',end='\r')
                print("the packet rssi value: -{0}dBm".format(256-r_buff[-
1:][0]))
                self.get_channel_rssi()
            else:
                pass
                #print('\x1b[2A',end='\r')

    def get_channel_rssi(self):
        GPIO.output(self.M1,GPIO.LOW)
        GPIO.output(self.M0,GPIO.LOW)
        time.sleep(0.1)
        self.ser.flushInput()
        self.ser.write(bytes([0xC0,0xC1,0xC2,0xC3,0x00,0x02]))
        time.sleep(0.5)
        re_temp = bytes(5)
        if self.ser.inWaiting() > 0:
            time.sleep(0.1)
            re_temp = self.ser.read(self.ser.inWaiting())
        if re_temp[0] == 0xC1 and re_temp[1] == 0x00 and re_temp[2] ==
0x02:
            print("the current noise rssi value: -{0}dBm".format(256-
re_temp[3]))
            # print("the last receive packet rssi value: -{0}dBm".for-
mat(256-re_temp[4]))
        else:
            # pass
            print("receive rssi value fail")
            # print("receive rssi value fail: ",re_temp)
```

Angepasster Main code zum senden der CAN Daten:main1.py

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

#
#    this is an UART-LoRa device and thers is an firmware on Module
#    users can transfer or receive the data directly by UART and dont
#    need to set parameters like coderate,spread factor,etc.
#    |========================================== |
#    |   It does not suport LoRaWAN protocol !!!   |
#    | ==========================================|
#
#    This script is mainly for Raspberry Pi 3B+, 4B, and Zero series
#    Since PC/Laptop does not have GPIO to control HAT, it should be con-
figured by
#    GUI and while setting the jumpers,
#    Please refer to another script pc_main.py
#

import sys
import sx126x
import threading
import time
import select
import termios
import tty
import can    # CAN-Bibliothek für Python1
from threading import Timer

old_settings = termios.tcgetattr(sys.stdin)
tty.setcbreak(sys.stdin.fileno())


#
#    Need to disable the serial login shell and have to enable serial in-
terface
#    command `sudo raspi-config`
#    More details: see https://github.com/Mith-
unHub/LoRa/blob/main/Basic%20Instruction.md
#
#    When the LoRaHAT is attached to RPi, the M0 and M1 jumpers of HAT
should be removed.
#


#    The following is to obtain the temprature of the RPi CPU
def get_cpu_temp():
    tempFile = open( "/sys/class/thermal/thermal_zone0/temp" )
    cpu_temp = tempFile.read()
    tempFile.close()
    return float(cpu_temp)/1000

def send_can_data(continue_or_not=True):
    if continue_or_not:
        global timer_task
        global seconds
```

```
        bus = can.interface.Bus(channel='can0', bustype='socketcan')  #
CAN-Schnittstelle öffnen
        message = bus.recv(1.0)  # Warte max. 1 Sekunde auf eine Nachricht

        if message:
            can_data = f"ID:{message.arbitration_id} DATA:{mes-
sage.data.hex()}"  # Formatierung
            data = bytes([255, 255, 18, 255, 255, 12]) + can_data.encode()
# LoRa-Paket

            node.send(data)  # Nachricht per LoRa senden
            print(f"Gesendet: {can_data}")  # Debug-Ausgabe

        timer_task = Timer(seconds, send_can_data)  # Timer für nächste
Übertragung
        timer_task.start()
    else:
        timer_task.cancel()
#    serial_num
#        PiZero, Pi3B+, and Pi4B use "/dev/ttyS0"
#
#     Frequency is [850 to 930], or [410 to 493] MHz
#
#     address is 0 to 65535
#         under the same frequence,if set 65535,the node can receive
#         messages from another node of address is 0 to 65534 and similarly,
#         the address 0 to 65534 of node can receive messages while
#         the another note of address is 65535 sends.
#         otherwise two node must be same the address and frequence
#
#     The tramsmit power is {10, 13, 17, and 22} dBm
#
#     RSSI (receive signal strength indicator) is {True or False}
#         It will print the RSSI value when it receives each message
#

# node = sx126x.sx126x(serial_num =
"/dev/ttyS0",freq=433,addr=0,power=22,rssi=False,air_speed=2400,re-
lay=False)
node = sx126x.sx126x(serial_num =
"/dev/ttyAMA0",freq=868,addr=0,power=22,rssi=True,air_speed=2400,re-
lay=False)

def send_deal():
    get_rec = ""
    print("")
    print("input a string such as \033[1;32m0,868,Hello World\033[0m,it
will send `Hello World` to lora node device of address 0 with 868M ")
    print("please input and press Enter key:",end='',flush=True)

    while True:
        rec = sys.stdin.read(1)
        if rec != None:
            if rec == '\x0a': break
            get_rec += rec
            sys.stdout.write(rec)
            sys.stdout.flush()

    get_t = get_rec.split(",")
```

```
    offset_frequence = int(get_t[1])-(850 if int(get_t[1])>850 else 410)
    #
    # the sending message format
    #
    #         receiving node              receiving node
receiving node           own high 8bit           own low 8bit
own
    #         high 8bit address           low 8bit address
frequency                address                 address
frequency             message payload
    data = bytes([int(get_t[0])>>8]) + bytes([int(get_t[0])&0xff]) +
bytes([offset_frequence]) + bytes([node.addr>>8]) + bytes([node.addr&0xff])
+ bytes([node.offset_freq]) + get_t[2].encode()

    node.send(data)
    print('\x1b[2A',end='\r')
    print(" "*200)
    print(" "*200)
    print(" "*200)
    print('\x1b[3A',end='\r')


# def send_cpu_continue(continue_or_not = True):
 #   if continue_or_not:
  #      global timer_task
   #     global seconds
        #
        # boarcast the cpu temperature at 868.125MHz
        #
#         data = bytes([255]) + bytes([255]) + bytes([18]) + bytes([255]) +
bytes([255]) + bytes([12]) + "CPU Temperature:".en-
code()+str(get_cpu_temp()).encode()+" C".encode()
#         node.send(data)
#         time.sleep(0.2)
#         timer_task = Timer(seconds,send_cpu_continue)
#         timer_task.start()
   # else:
#         data = bytes([255]) + bytes([255]) + bytes([18]) + bytes([255]) +
bytes([255]) + bytes([12]) + "CPU Temperature:".en-
code()+str(get_cpu_temp()).encode()+" C".encode()
#         node.send(data)
#         time.sleep(0.2)
#         timer_task.cancel()
#         pass

try:
    time.sleep(1)
    print("Press \033[1;32mEsc\033[0m to exit")
    print("Press \033[1;32mi\033[0m   to send")
    print("Press \033[1;32ms\033[0m   to send cpu temperature every 10 sec-
onds")

    # it will send rpi cpu temperature every 10 seconds
    seconds = 1

    while True:

        if select.select([sys.stdin], [], [], 0) == ([sys.stdin], [], []):
            c = sys.stdin.read(1)
```

```
            # dectect key Esc
            if c == '\x1b': break
            # dectect key i
            if c == '\x69':
                send_deal()
            # dectect key s
            if c == '\x73':
                print("Press \033[1;32mc\033[0m   to exit the send task")
                timer_task = Timer(seconds,send_can_data)
                timer_task.start()


                while True:
                    if sys.stdin.read(1) == '\x63':
                        timer_task.cancel()
                        print('\x1b[1A',end='\r')
                        print(" "*100)
                        print('\x1b[1A',end='\r')
                        break

            sys.stdout.flush()

        node.receive()

        # timer,send messages automatically

except:
    termios.tcsetattr(sys.stdin, termios.TCSADRAIN, old_settings)
    # print('\x1b[2A',end='\r')
    # print(" "*100)
    # print(" "*100)
    # print('\x1b[2A',end='\r')

termios.tcsetattr(sys.stdin, termios.TCSADRAIN, old_settings)
# print('\x1b[2A',end='\r')
# print(" "*100)
# print(" "*100)
# print('\x1b[2A',end='\r')
```