HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

BACHELORARBEIT

Herr
Yang Zhou

Programmierung einer Vehicle Control Unit (VCU) für einen elektrifizierten Rasentraktor

Mittweida, 2023

Fakultät Ingenieurwissenschaften

BACHELORARBEIT

Programmierung einer Vehicle Control Unit (VCU) für einen elektrifizierten Rasentraktor

Autor:
Herr Yang Zhou

Studiengang:
Mechatronik

Seminargruppe:
Me18wA-b

Erstprüfer**:**
Prof. Dr.-Ing. Lutz Rauchfuß

Zweitprüfer:
M. Sc. Sebastian Nitschke

Einreichung:
Mittweida, 31.05.2023

Verteidigung/Bewertung:
Mittweida, 2023

Faculty Engineering

BACHELOR THESIS

Programming of a Vehicle Control Unit (VCU) for an electrified lawn tractor

author:
Mr. Yang Zhou

course of studies:
Mechatronics

seminar group:
Me18wA-b

first examiner:
Prof. Dr.-Ing. Lutz Rauchfuß

second examiner:
M. Sc. Sebastian Nitschke

submission:
Mittweida, 31.05.2023

defence/ evaluation:
Mittweida, 2023

Bibliografische Beschreibung:

Zhou, Yang:
Programmierung einer Vehicle Control Unit (VCU) für einen elektrifizierten Rasentraktor . - 2023. - 10, 44, 19 S.Mittweida, Hochschule Mittweida, Fakultät Ingenieurwissenschaften, Bachelorarbeit , 2023

Referat:

In der heutigen Zeit sind elektrische Antriebe in vielen Bereichen des Alltags anzutreffen. Elektrofahrzeuge tragen dazu bei, Reisen umweltfreundlicher zu gestalten. Das Ziel dieser Arbeit ist die Elektrifizierung einer Rasenmäher Maschine mit Schwerpunkt auf VCU-Programmierung. In dieser Arbeit wird detailliert beschrieben, wie alle Funktionen, einschließlich Kommunikation, Tests usw., umgesetzt werden. Darüber hinaus werden auch zukünftige Entwicklungen und Fragen zur Gerätesicherheit untersucht.

Fakultät, Ingenieurwissenschaften (INW)
Professur Antriebs- und Regelungstechnik (ART)
Prof. Dr.-Ing. Lutz Rauchfuß

HOCHSCHULE MITTWEIDA
UNIVERSITY OF APPLIED SCIENCES

Bacheloraufgabenstellung für Zhou Yang

Mittweida, 31.01.2023

Thema: Programmierung einer Vehicle Control Unit (VCU) für einen elektrifizierten Rasentraktor

Rasentraktoren ermöglichen die Pflege großer Grundstücke mit vertretbarem Zeiteinsatz, dabei entstehen Abgase und unangenehme Geräusche. Ein handelsüblicher Rasentraktor soll von Benzin- auf Akkubetrieb umgerüstet werden, um diese Nachteile zu kompensieren. Zur Überwachung und Regelung aller Komponenten im Fahrzeug ist eine zentrale Steuereinheit VCU notwendig. Informationen für den Fahrer werden von der VCU über ein Display bereitgestellt und Schalter zur Bedienung des Rasentraktors sind von der VCU abzufragen. Die Kommunikation zwischen den Komponenten soll via CAN-Bus erfolgen.

Arbeitspakete:

1. Programmierung der VCU zur Ansteuerung des Displays via I²C-Bus im Einschaltmodus des Rasentraktors
   - Mit dem Wechsel des Zündschalters von **Stellung 0 (AUS)** auf **Stellung 1 (Zündung EIN)** wird die gesamte Elektronik des Rasentraktors inkl. VCU eingeschaltet und die **Wake-up-Routine** gestartet.
   - **Stellungen des Zündschlosses:**
     0-AUS
     1-Zündung EIN (dauerhaft) ▯ **Wake-up-Routine**
     2-Fahrbetrieb (tastend) ▯ **Fahr-Routine**
   - In der **Wake-up-Routine** muss die VCU das ***Bereitschaftssignal*** (Arduino-PIN 23) von 0 auf **1** setzen, damit werden die Pegel der Bedienschalter über eine elektronische Hardwareverriegelung an die Eingangspins des Arduinos gelegt.
     Bedienschalter:
     - ☐ Fahrersitz 0 (unbesetzt)
     - ☐ **Fahrersitz 1 (besetzt)**
     - ☐ Richtungsschalter: Vorwärts
     - ☐ **Richtungsschalter: Stillstand**
     - ☐ Richtungsschalter: Rückwärts
     - ☐ **Mähwerk: 0 (ausgeschaltet)**
     - ☐ Mähwerk: **1** (Mähmotor ein)
   - Befinden sich die Bedienschalter in den fett gedruckten Positionen, ist die VCU bereit von der **Wake-up-Routine** in die **Fahr-Routine** zu wechseln. Das Display zeigt **„START mit Stellung 2"** und die Position der Bedienschalter an **„Fahrersitz 1"**, **„Stillstand"** und **„Mähwerk 0"** an.
   - Befinden sich die Bedienschalter nicht in den fett gedruckten Positionen, ist die VCU nicht bereit in die **Fahr-Routine** zu wechseln, im Display wird **„nicht Fahrbereit"** und die Position der Bedienschalter angezeigt z.B. **„Fahrersitz 0"** , **„Vorwärts"** und **„Mähwerk 1"**.
   - Die VCU bleibt in der **Wake-up-Routine** und liest weiter die Bedienschalter ein. Erfolgt nun ein Wechsel der Bedienschalter, werden die aktuellen Positionen im Display angezeigt. Werden die fett gedruckten Positionen der Bedienschalter erreicht, ist die VCU bereit von der **Wake-up-**

Routine in die **Fahr-Routine** zu wechseln. Das Display zeigt **„START mit Stellung 2“** und die Position der Bedienschalter an **„Fahrersitz 1“**, **„Stillstand“** und **„Mähwerk 0“** an.

- Schaltet der Fahrer auf die Zündschlossposition **2-Fahrbetrieb (tastend)** wechselt die VCU von der **Wake-up-Routine** in die **Fahr-Routine,** setzt die Selbsthaltung (Arduino-PIN 34) und zeigt im Display **„Fahr-Routine“** und den aktuellen **SOC-Wert** an, der via CAN-Bus eingelesen wurde. Mit der Selbsthaltung schaltet ein TTL-Relais das „Ignition“-Signal am BMS ein und macht damit die Kommunikation via CAN möglich.

2. In der Schalterstellung 2 des Zündschlosses muss die Kommunikation mit dem BMS via CAN hergestellt werden
   - Aus dem definierten Register des tiny-BMS muss die VCU via CAN den aktuellen Wert des SOC auslesen.
   - Das Display zeigt **„Fahr-Routine“** und den aktuellen **SOC-Wert** an.
   - Die Namenskonvention der CAN-Signale soll einen klaren Bezug von Sender und Empfänger haben, dabei wird der Sender zuerst und der Empfänger in zweiter Position durch ein Underline getrennt, das physikalische Signal wird in der 3. Position, nach dem 2. Underline genannt.
     Beispiel zur Namenskonvention:
     1. Position: **BMS** (Sender) Signalquelle
     2. Position: **VCU** (Empfänger) ist Empfänger des Signals
     3. Position: **SOC** (physikalisches Signal) SOC des Akku
     „BMS_VCU_SOC_ist“

Hochschulbetreuer:

Prof. Dr.-Ing. Lutz Rauchfuß

Inhalt

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ABS	Anti-lock Braking System
ACC	Adaptive cruise Control
ASCII	American Standard Code for Information Interchange
BMS	Battery Management System
CAN	Controller Area Network
CANID	Controller Area Network Identifier
CANRX	Controller Area Network Receiver
CANTX	Controller Area Network Transmitter
CRC	Cyclic Redundancy Check
DLC	Data Length Code
ECU	Electronic Control Unit
I2C/IIC	Inter-Integrated Circuit
IDE	Integrated development Environment
LCD	Liquid Crystal Display
LSB	Least Significant Bit
MSB	Most Significant Bit
OSI	Open System Interconnection Model
RAM	Random Access Memory
ROM	Read only Memory
SCL	Serial Clock
SDA	Serial Data
SoC	State of Charge
UART	Universal Asynchronous Receiver/Transmitter

1 Einleitung

Dieses Bachelorarbeit wurde im Rahmen des Sommersemesters 2022 an der Hochschule Mittweida verfasst, bei dem es um die Umrüstung eines Rasentraktor von Benzinantrieb auf einen Elektroantrieb geht.

Im Einleitung Kapitel wurden die Motivation und Zielsetzung dieser Arbeit erörtert. Es wurde auch eine kurze Übersicht über die jeweiligen Kapitel der vorliegenden Arbeit gegeben.

1.1 Motivation

Rasentraktoren ermöglichen die Pflege großer Grundstücke mit vertretbarem Zeiteinsatz, dabei entstehen Abgase und unangenehme Geräusche. Ein handelsüblicher Rasentraktor soll von Benzinbetrieb auf Akkubetrieb umgerüstet werden, um diese Nachteile zu kompensieren.[1] Elektrischer Antrieb ist der Trend der Zukunft und erfüllt die Marktanforderungen an Umweltschutz und Nachhaltigkeit. Elektrischer Antrieb wird daher auf absehbare Zeit eine führende Rolle spielen.

Abbildung 1 Rasentraktor Attila SKD[9]

Herkömmliche Rasentraktoren haben vier Räder, mit zwei angetriebenen Rädern auf der Hinterachse, die nicht lenkbar sind, und zwei lenkbaren Rädern auf der Vorderachse. Beim Abbiegen gewährleistet das Ackermann-Lenksystem, dass die Innenräder der Kurve mehr einschlagen als die Außenräder, um eine reibungslose Lenkung des Fahrzeugs zu ermöglichen. Normalerweise ist das Drehen auf der Stelle nicht möglich, aber durch die elektrische Umrüstung kann der Rasentraktoren durch unterschiedliche Geschwindig-

keiten der Innen- und Außenräder einen kleinen Wendekreis oder sogar das Drehen auf der Stelle erreichen. Dies ist besonders nützlich bei Arbeiten in engen Räumen.

1.2 Zielsetzung

Um alle Komponenten des Fahrzeugs zu überwachen und zu steuern, ist eine Vehicle Control Unit (VCU) erforderlich. Die VCU liefert dem Fahrer über einen LCD Informationen und erhält Signale von Schaltkreis, um die VCU zu steuern. Die VCU wiederum steuert andere Arduinos. Die VCU benötigt Kommunikation mit dem Antriebsmotor, dem Mähwerkmotor und dem Batteriemanagementsystem (BMS). Die Kommunikation zwischen Komponenten erfolgt über den CAN-Bus.

1.3 Kapitelübersicht

Kapitel 1: Motivation und Ziel des Projekts wurden vorgestellt.

Kapitel 2: Die Aufgabenanforderungen dieser Arbeit wurden detailliert erläutert.

Kapitel 3: Die in der Arbeit verwendeten Techniken und theoretischen Grundlagen wurden vorgestellt.

Kapitel 4: Die Implementierung der Software- und Hardwarefunktionen wurde detailliert erklärt.

Kapitel 5: Abschließend wurden die Daten überprüft und verglichen, um sicherzustellen, dass das VCU die festgelegten Aufgabenanforderungen erfüllt.

Kapitel 6: Es wurde eine Zusammenfassung gegeben und ein Ausblick auf mögliche Erweiterungen und Probleme des VCU gegeben.

2 Präzisierung der Aufgabenstellung

Der Kern des gesamten Projekts besteht darin, die VCU mit Arduino zu programmieren. Arduino ist dafür zuständig, Signale von der Steuerplatine zu verarbeiten, das LCD über I2C zu steuern, Informationen von der TinyBMS über CAN-Bus zu lesen und den Antriebsmotor und den Mähwerkmotor mit Arduino zu steuern. Außerdem werden der Lenkwinkel des Lenkrads und die Drehzahl der Vorderräder des Fahrzeugs ebenfalls ausgelesen. Schließlich muss der Arduino auf der Steuerplatine die Fahrzeugsignale korrekt verarbeiten, die Position der Schlüssel bestimmen und prüfen, ob sich ein Fahrer im Fahrersitz befindet usw. Arduino liest Informationen von der TinyBMS über CAN-Bus aus und verarbeitet sie, um die Restlaufzeit zu berechnen. Das LCD zeigt in Echtzeit die Batteriekapazität und die berechnete Restlaufzeit von der TinyBMS sowie Fahrzeug-informationen von der Steuerplatine an. Das Gesamtsystem ist in den Abbildungen 2 und 3 zu sehen, in denen auch die Namen der einzelnen Komponenten aufgeführt sind.

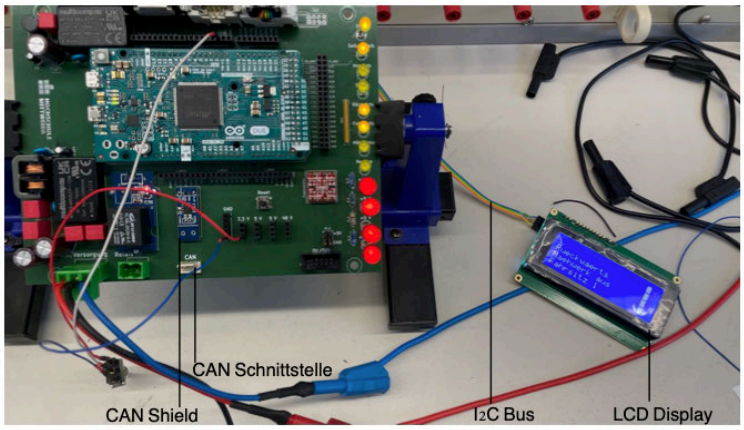

Abbildung 2 LCD verbunden mit Arduino über I2C-Bus

Die Aufgabe kann in einen Hardware- und einen Softwareteil unterteilt werden. Der Hardwareteil muss zuerst abgeschlossen werden, da sichergestellt werden muss, dass die Hardware keine Probleme aufweist, bevor das Programm getestet wird. Wenn es Hardwareprobleme gibt, ist der Programmtest sinnlos und das Debuggen wird schwierig und zeitaufwendig. Aber in diesem Projekt ist es nicht erforderlich, alle Hardware-Tests abzuschließen, bevor mit der Programmierung begonnen wird. I.d.R. wird zunächst ein Hardware-Test durchgeführt, gefolgt von der entsprechenden Programmierung. Dieser Ansatz gewährleistet, dass die Hardware und das Programm aus dem vorherigen Schritt

fehlerfrei sind, was die Effizienz und Zuverlässigkeit verbessert. Gleichzeitig wird die Schwierigkeit der Fehlerbehebung bei der nachfolgenden Programmierung verringert.

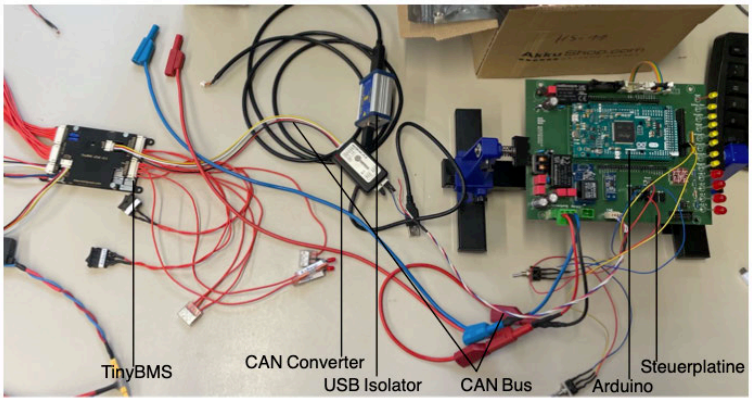

Abbildung 3 TinyBMS und Steuerplatine

Nach erfolgreicher CAN-Kommunikation kann CANoe zur Signalanalyse oder Fehlersuche verwendet werden, einschließlich grafischer Darstellung. Abbildung 4 zeigt ein Beispiel für die grafische Darstellung. Die grafische Darstellung dient der intuitiven Beobachtung aller Daten, wie Geschwindigkeit, Lenkwinkel usw. Da während des Projekts keine Geräte wie Winkelsensoren oder Geschwindigkeitssensoren installiert wurden, dient das grafische Programm lediglich als Referenz für zukünftige Entwicklungen und ist nicht der Schwerpunkt dieser Arbeit. Bei Bedarf kann es auf der bestehenden Grundlage erweitert und verbessert werden, indem Variablen und Funktionen ausgetauscht werden.

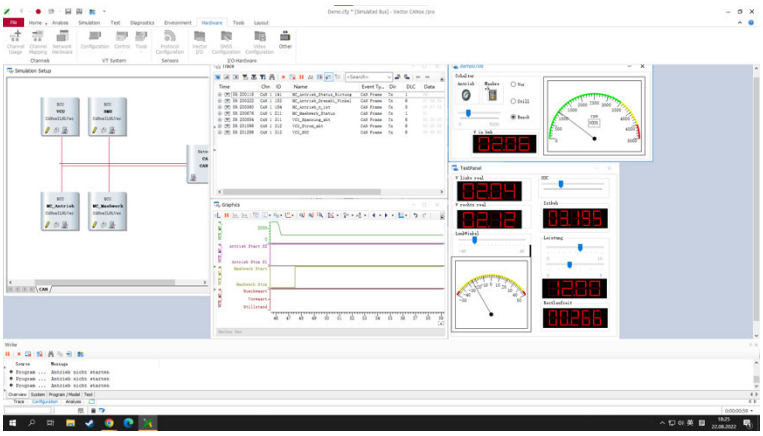

Abbildung 4 CANoe Grafische Darstellung

3 Grundlagen und Stand der Technik

Bevor das Projekt angegangen wird, besteht die vorrangige Aufgabe darin, eine umfassende Untersuchung über den aktuellen Stand der Technologie im relevanten Bereich durchzuführen. Dieser Schritt ist von entscheidender Bedeutung, um Empfehlungen für die Umsetzung des Projekts zu erhalten.

3.1 Arduino Due

Arduino ist eine Open-Source-Hardware- und Softwareplattform zur Entwicklung und Steuerung verschiedener physischer Geräte und interaktiver Systeme. Arduino-Board verwendet Mikrocontroller zur Programmierung und Steuerung von elektronischen Komponenten wie LED-Lichtern, Motoren und Sensoren. Es kann problemlos mit Computern, Netzwerken und anderen Geräten verbunden und gesteuert werden.

Arduino kann seine Funktionen durch zusätzliches Zubehör, auch bekannt als Arduino Shields, erweitern werden. Dieses Shields umfasst Sensoren, Aktuatoren, Displays, Kommunikationsmodule usw., die mit dem Arduino-Board verbunden werden können und durch Programmierung gesteuert und interagiert werden können. Durch Hinzufügen geeigneten Shields können eine Vielzahl von Projekten und Anwendungen realisiert werden, wie z.B. Temperaturüberwachung, Bewegungserkennung, drahtlose Kommunikation usw. Die Auswahl des Shields hängt von den Anforderungen und Zielen des Projekts ab. 2

Gängige Arduino-Boards sind Arduino Uno, Arduino Mega, Arduino Nano, Arduino Leonardo, Arduino Due usw. Abbildung 5 zeigt einen gewöhnlichen Arduino. Weil Arduino Open Source ist, gibt es auch viele inoffizielle Arduinos, aus denen man wählen kann In diesem Projekt wird Arduino Due verwendet. Arduino Due verwendet einen leistungsstärkeren 32-Bit-Atmel SAM3X8E ARM Cortex-M3-Prozessor.2 Im Vergleich zu anderen Arduino-Boards bietet Arduino Due eine schnellere Verarbeitungs-geschwindigkeit und größeren Speicherplatz. Arduino Due arbeitet mit einer Taktfrequenz von 84 MHz und verfügt über 512 KB Flash-Speicher, was es ermöglicht, komplexere Aufgaben und größere Projekte zu bewältigen. Darüber hinaus verfügt Arduino Due über 54 digitale Ein-/Ausgabe-Pins (davon 12 für PWM-Ausgabe), 12. analoge Eingabe-Pins, 2 analoge Ausgabe-Pins, serielle Kommunikationsports und CAN-Kommunikationsports, um problemlos mit anderen Geräten kommunizieren und diese steuern zu können.

Abbildung 5 Arduino-Boards[10]

Arduino Due unterstützt auch die Arduino Programmiersprache und Entwicklungsumgebung Arduino IDE, sodass Benutzer mit vertrauten Tools und Sprachen programmieren und entwickeln können. Wie andere Arduino-Boards profitiert auch Arduino Due von einer großen Community-Unterstützung, einer Vielzahl von Open-Source-Ressourcen, viele Zubehörteile und Bibliotheken zur Verfügung, und es gibt eine aktive Community, die Unterstützung bietet und verschiedene Ideen und Projekte teilt. Aufgrund seiner Benutzerfreundlichkeit und niedrigen Kosten ist Arduino zu einem beliebten Werkzeug für viele Hobbyisten geworden.

3.2 I2C/IIC-Bussystem

Inter-Integrated Circuit wird auch als I2C-Bus oder IIC-Bus bezeichnet. I2C-Bus ist ein serieller Bus, der in den 1980er Jahren von der Firma Philips eingeführt wurde. Es zeichnet sich durch seine geringe Anzahl an Pins, einfache Hardwareimplementierung und hohe Skalierbarkeit aus. I2C-Bus erfolgt im Master-Slave-Modus, unterstützt jedoch auch Multi-Master-Kontrolle, wobei jedes Gerät auf dem Bus senden/empfangen kann. Es kann jedoch nur einen Master zu einem bestimmten Zeitpunkt geben.

I2C ist sowohl ein Bus als auch ein Kommunikationsprotokoll. In der Embedded-Entwicklung können Kommunikationsprotokoll in zwei Schichten unterteilt werden: die physikalische Schicht und die Protokollschicht. Die physikalische Schicht gewährleistet die Übertragung der Daten über das physische Medium. Im Fall der I2C-Kommunikation werden nur zwei bidirektionale Leitungen benötigt: die serielle Datenleitung (SDA) und die serielle Taktleitung (SCL). Die serielle Datenleitung SDA wird zur Übertragung der Daten verwendet.[2] Die serielle Taktleitung SCL wird für die Datenübertragung synchronisiert.

Abbildung 6 zeigt, wie das I2C-Signal übertragen wird, wobei der Master die Nachricht sendet und der Slave sie beantwortet.

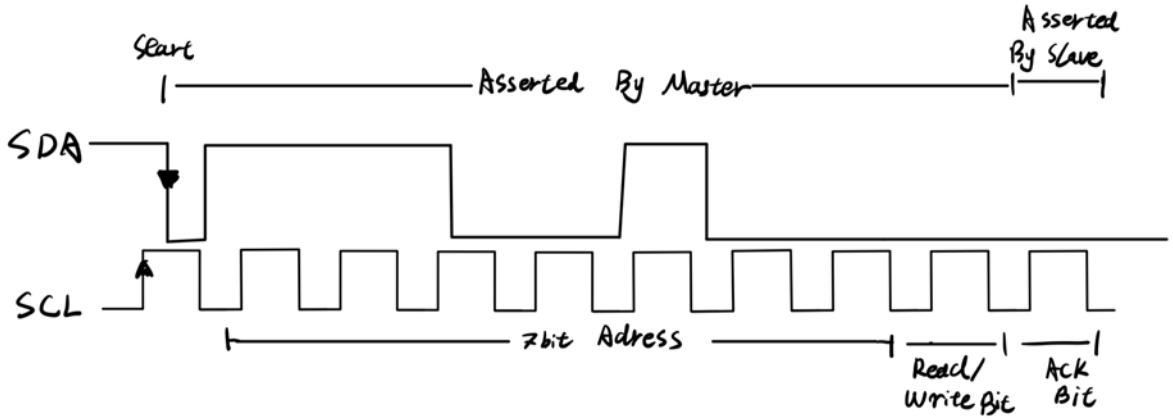

Abbildung 6 I2C-Signal Übertragung

Die Protokollschicht definiert Start- und Stop-Signale, Datenvalidität, Response, Adressen-Broadcast, Arbitrierungssynchronisation, und Standards für die Datenverpackung und Datenentpackung zwischen Sender und Empfänger. Start- und Stop-Signale werden vom Master gesendet. Nachdem das Start-Signal gesendet wurde, befindet sich der Bus im besetzten Zustand. Nachdem das Stop-Signal gesendet wurde, befindet sich der Bus im Leerlaufzustand.

Jedes Gerät, das mit dem I2C-Bus verbunden ist, hat eine eindeutige Adresse, über die es vom Master angesprochen werden kann. Wenn mehrere Master gleichzeitig den Bus nutzen, muss eine Arbitrierung stattfinden, um zu bestimmen, welches Gerät den Bus verwenden darf, da sonst Datenkonflikte auftreten können.

Die Übertragungsgeschwindigkeit von I2C ist i.d.R. relativ niedrig, kann aber je nach Bedarf angepasst werden. Im Standardmodus beträgt die Übertragungsgeschwindigkeit 100 KBit/s, während im Fast-Modus bis zu 400 KBit/s erreicht werden können. Es gibt auch schnellere Modi wie den High-Speed-Modus (1 Mbit/s) [3] und den Ultra-High-Speed-Modus (3,4 Mbit/s). Obwohl der I2C-Hochgeschwindigkeitsmodus das Problem der Übertragungsgeschwindigkeit löst, bringt er gleichzeitig auch eine Reihe von Problemen mit sich, wie beispielsweise die Notwendigkeit einer Aktualisierung des Controllers und höhere Anforderungen an die Übertragungsleitung. Dies hat dazu geführt, dass seine Verbreitung immer noch auf gewisse Hindernisse stößt und eine großflächige Anwendung bisher ausbleibt.

3.3 LCD

Ein Liquid Crystal Display (LCD) ist ein einfacher Bildschirm, der in den 1980er Jahren entwickelt wurde und auch heute noch sehr beliebt ist. Das Funktionsprinzip eines LCDs beruht darauf, dass sich am unteren Ende des LCDs eine LED-Hintergrundbeleuchtung befindet, die bei Stromzufuhr leuchtet. Darüber liegt eine Schicht aus Flüssigkristall, wobei jeder einzelne Pixel von einem kleinen Flüssigkristall Stück gesteuert wird. Durch die Steuerung der Ausgangsspannung für den Flüssigkristall kann die Lichtdurchlässigkeit gesteuert werden. Abbildung 7 zeigt das Funktionsprinzip des LCD. Die Steuerung aller Anzeigepixel auf eine LCD erfolgt durch eine zeilenweise scannen (auch englisch Progressive Scan). Zuerst werden die Pixel der ersten Zeile gesteuert, dann die der nächsten Zeile, und so weiter in einem endlosen Zyklus, um alle Pixel zu kontrollieren. Aufgrund des Phänomens des visuellen Nachbildes im menschlichen Auge und der schnelle Scan des LCDs erscheint das Bild als statisch.

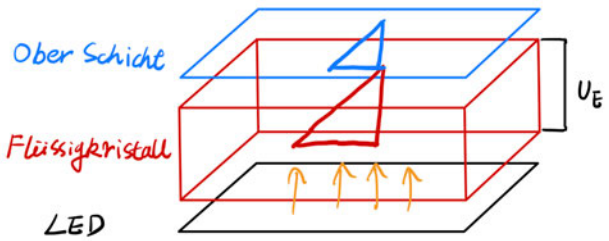

Abbildung 7 LCD Funktionsprinzip

Das in diesem Projekt verwendete LCD-Modul ist ein 20x4-LCD mit einem HD44780u-Controller. Dieses LCD ermöglicht das Schreiben von Zeichen in Weiß auf einem blauen Hintergrund. Der Bildschirm hat 20 horizontale Displaybereiche und 4 vertikale Displaybereiche, weshalb das Modell auch als “2004 LCD" bezeichnet wird. Jeder Displaybereich besteht aus 35 Pixeln (5x7), also ganz LCD insgesamt 2800 Pixel. Da jeder Displaybereich nur 35 Pixel hat, kann er nur einfache Informationen wie Zahlen, Buchstaben und Symbole anzeigen. Es können insgesamt 192 Zeichen angezeigt werden, die im ROM dauerhaft programmiert sind. Alle verfügbaren 192 Zeichen sind in Abbildung 8 dargestellt. Darüber hinaus können benutzerdefinierte Zeichen mithilfe von ASCII-Codes codiert und im Speicher abgelegt werden, um darauf zuzugreifen.

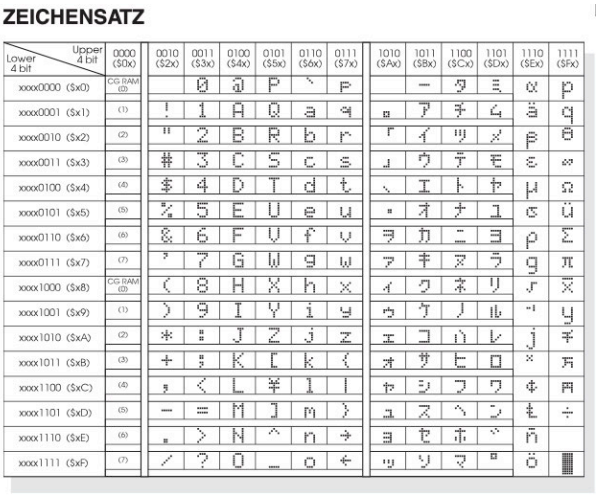

Abbildung 8 Zeichensatz von LCD [11]

Der HD44780u ist der Hauptsteuerchip des LCDs. Er verfügt über 80 Byte Display-RAM und kann perfekt 80 Displaybereiche steuern. Da jedoch die Anzahl der Pins des HD44780-Chips begrenzt ist, wird die Steuerung von mehr als 16 Zeichen durch zusätzliche Treiberchips (HD44100h) unterstützt. Diese helfen dem Hauptsteuerchip, die restlichen Anzeigeaufgaben zu erledigen. Der Hauptsteuerchip sendet Nachrichten seriell an die Treiberchips, die dann ihre Pin-Status steuern können. Um 80 Anzeigezeichen zu steuern, werden 1 HD44780u als Hauptsteuerchip und 4 HD44100h als Treiberchips benötigt. Abbildung 9 zeigt das LCD im Test.

Abbildung 9 LCD im Test

3.4 CAN-Bussystem

Der Controller Area Network (CAN) ist ein Bussystem. Die erste Version des CAN-Bus wurde im Jahr 1983 veröffentlicht. Die aktuellen Versionen umfassen CAN 2.0A, CAN 2.0B und CAN FD. Der ursprüngliche Anwendungsbereich lag in der Automobilindustrie, um das Kabelgewicht zu reduzieren und die Verkabelung zwischen den elektronischen Steuergeräten (ECU) im Fahrzeug zu vereinfachen. Moderne Fahrzeuge sind mit vielen elektronischen Steuergeräten (ECU) ausgestattet, wie z.B. Motorsteuergeräte, Airbagsteuergeräten, ABS, ACC usw. Einige dieser Steuergeräte sind eigenständige Systeme, während anderen Steuergeräten mit anderen Systemen kommunizieren müssen, um Aktuatoren zu steuern oder Rückmeldungen von Sensoren zu empfangen. Dafür wurde das CAN-Bussystem entwickelt, um die verschiedenen Systeme im Fahrzeug miteinander zu verbinden. Die traditionelle Verkabelung ist kostspielig und komplex, während das Controller Area Network einfach durch Software realisiert werden kann. Es ist nicht nur sicher und kosteneffizient, sondern auch äußerst praktisch. Heutzutage wird das CAN-Bussystem nicht nur in der Automobilindustrie, sondern auch in Bereichen wie Automation und medizinischen Geräten weit verbreitet eingesetzt. Abbildung 10 zeigt den grundlegenden strukturellen Aufbau eines CAN-Netzwerks.

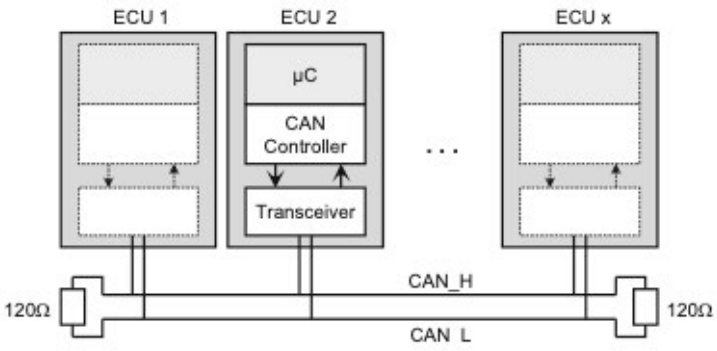

Abbildung 10 Struktur eines CAN-Netzwerkes [4]

3.4.1 Arbitrierung im CAN-Bus

Der CAN-Bus ist ein Nachrichten Bussystem, bei dem Informationen, die von einem Knoten gesendet werden, von allen Knoten gelesen oder von einem bestimmten Knoten empfangen werden können. Auf diese Weise können wichtige Daten an viele Teilnehmer gesendet werden. Jedoch kann es aufgrund der gemeinsamen Nutzung des Busses zu Übertragungskonflikten kommen. Konflikte treten auf, wenn zwei oder mehr Knoten gleichzeitig mit der Übertragung von Nachrichten beginnen und ihre Nachrichtenrahmen auf dem Bus kollidieren können. Um solche Konflikte zu lösen, verwendet der CAN-Bus einen Arbitrierungsprozess. Der Arbitrierungsprozess des CAN-Busses kann wie folgt

zusammengefasst werden: In der Übertragungsvorbereitungsphase überprüft die Knoten, ob der Bus frei ist. Im Arbitrierungsphasen vergleichen die Knoten die Nachrichtenidentifikatoren (CANID), um die Priorität zu bestimmen. D.h. CANID mit niedrigeren Werten haben eine höhere Priorität. Es kann auf Abbildung 11 verwiesen werden.Das Ergebnis der Arbitrierung ermöglicht es dem Knoten mit höchster Priorität, die Nachricht fortzusetzen, während andere Knoten in den Empfangsmodus wechseln oder auf den freien Bus warten. Der Arbitrierungsprozess gewährleistet eine geordnete Übertragung der Nachrichten und löst Konflikte auf.

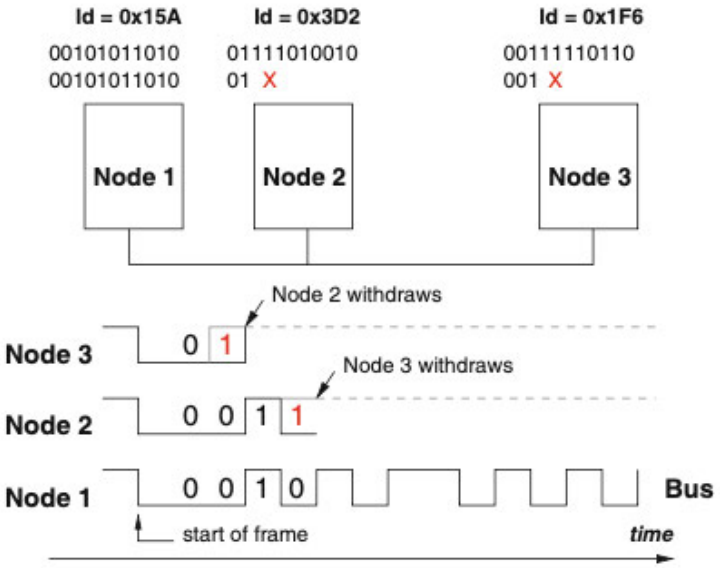

Abbildung 11 Beispiel für einen CAN-Bus Arbitrierungsprozess [5]

3.4.2 Aufbau der CAN-Botschaft

Die Kommunikation über den CAN-Bus erfolgt über eine Zweidrahtleitung in differentieller Form. CAN-Botschaften werden in Form von Botschaften übertragen. Eine CAN-Botschaften besteht im Allgemeinen aus vier Teilen: CANID, Datenfeld, Prüffeld und Rückmeldefeld (Abb. 12 und Abb. 13). Die CANID hat i.d.R 11 Bits, während für eine längere Extend-CANID 29 Bits verwendet werden. Das Datenfeld enthält die Datenlängecode (DLC) und die eigentlichen Daten, die eine maximale Länge von 8 Byte haben können. Das Prüffeld ist eine Zyklus-Redundanzprüfung (CRC), die Fehler in den übertragenen Daten oder im Originaltext erkennt. Das Rückmeldefeld ist für die Bestätigung des erfolgreichen Empfangs der Nachricht und die Überprüfung der Nachrichtenintegrität zuständig.

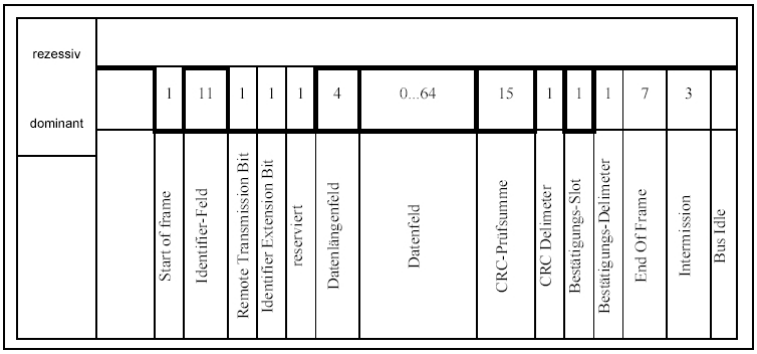

Abbildung 12 Aufbau eines CAN-Botschaft [12]

Abbildung 13 Format eines CAN-Botschaft

3.4.3 CAN-Bus Geschwindigkeit

In den Standards ISO 11898-2 und ISO 11898-3 wird festgelegt, dass alle Übertragungsraten über 250 kbit/s als High-Speed-CAN klassifiziert werden, während Übertragungsraten unter 125 kbit/s als Low-Speed-CAN klassifiziert werden. Die Signalpegel für High-Speed-CAN und Low-Speed-CAN sind unterschiedlich. Die CAN-Bus-Leitung besteht aus zwei differenziellen Leitungen, und die Erkennung von High- und Low-Signalen erfolgt anhand der Potentialdifferenz zwischen den beiden Leitungen. Abbildung 14 zeigt, wie sie die High- und Low-Signalen bestimmen.

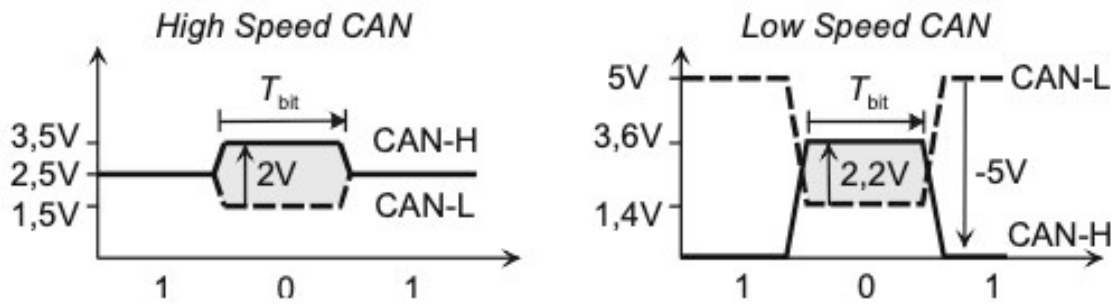

Abbildung 14 High Speed und Low Speed CAN [4]

Die CAN-Übertragungsgeschwindigkeit hängt auch von der Länge der Leitung ab. Die Baudrate und die Buslänge stehen in umgekehrtem Verhältnis zueinander. Je höher die Baudrate ist, desto kürzer darf die Kabellänge sein. Dies kann anhand der folgenden Tabelle und den daraus resultierenden Diagrammen veranschaulicht werden：

Baudrate(KBit/s)	Maximale Netzausdehnung(m)
500	112
300	200
200	310
100	640
50	1300

Tabelle 1 Abhängigkeit zwischen Baudrate und Buslänge [6]

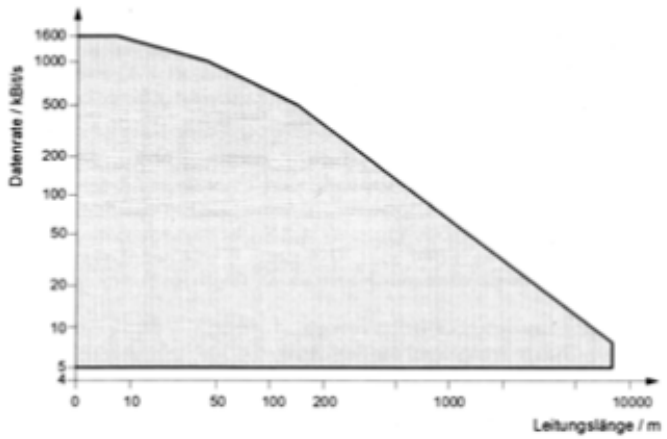

Abbildung 15 Abhängigkeit zwischen Baudrate und Buslänge [6]

Zusätzlich sind an beiden Enden des CAN-Busses zwei CAN-Leitungen über einen 120-Ohm-Widerstand miteinander verbunden. Dies repräsentiert den charakteristischen Wellenwiderstand der verdrillten Doppelleitung und dient dazu, Reflexionen an den Enden der Leitung zu vermeiden (Abb. 10).

3.4.4 CAN-Bus Hardware

In dem Projekt wird das CAN-Netzwerk, bestehend aus Arduino und TinyBMS, zur Steuerung und Analyse mit dem VN1610 CAN-Adapter von Vector Informatik GmbH verwendet. Diese Hardware ermöglicht die echtzeitige Analyse oder Steuerung von zwei CAN-Kanälen und wird über USB mit dem PC verbunden.

Um eine CAN-Kommunikation mit Arduino und TinyBMS zu realisieren, ist ein CAN-UART-Konverter unerlässlich. Zusätzlich befindet sich auf der Steuerplatine eine andere CAN-Shield. Des Weiteren werden an beiden Enden des CAN-Bus jeweils 120-Ohm-Widerstände benötigt. Im rechten schwarzen Draht des Diagramms 16 befindet sich ein 120-Ohm-Widerstand. Die im Projekt verwendeten CAN-UART-Konverter wurden von Energus geliefert.

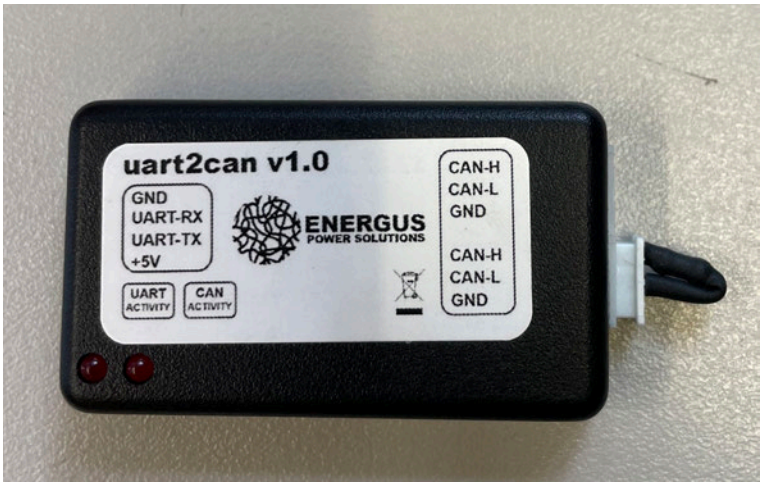

Abbildung 16 CAN-UART-Konverter mit 120 Ohm Widerstand

3.4.5 CAN-Bus Software

In dem Projekt wurde CANoe verwendet, um die CAN-Botschaften zu testen und zu analysieren. Es wurde auch verwendet, um Tests für das TinyBMS durchzuführen. Darüber hinaus wurden und nutzte seine Tools für die Simulation eingesetzt.

3.4.5.1 CANoe

CANoe ist ein weit verbreitetes Entwicklungswerkzeug zur Entwicklung, Test und Analyse von CAN-Bus-Systemen. Es wurde von der Firma Vector Informatik GmbH im Jahr 1996 veröffentlicht, und die aktuelle Version ist CANoe 16. Es ermöglicht das Management und die Steuerung mehrerer CAN-Netzwerke. In diesem Projekt wurde CANoe Version 15 (SP4) verwendet. CANoe bietet eine Vielzahl von Funktionen und Tools, darunter CANalyzer, CAPL, CANape usw. Mit CANoe können CAN-Bus-Systeme einfach entwickelt und debuggt werden.

CANoe kann den CAN-Bus simulieren, wodurch Benutzer CAN-Botschaften generieren und senden, verschiedene Knoten und Kommunikationsverhalten simulieren können, um die Reaktion und Leistung des Systems zu testen. CANoe verfügt über leistungsstarke Nachrichtenanalyse- und Aufzeichnungsfunktionen. Es kann alle übertragenden

Nachrichten auf dem CAN-Bus erfassen und anzeigen. CANoe unterstützt die Simulation und Test von CAN-Bus-Systemen in einer virtuellen Umgebung, und die Ergebnisse können gespeichert und wiedergegeben werden. Benutzer können mit simulierten CAN-Knoten und Signalen die Funktionalität und Leistung des Systems überprüfen, ohne tatsächliche Hardware sowie Schaltungsaufbau zu benötigen. CANoe bietet umfangreiche Diagnose- und Debugging-Funktionen zur Lokalisierung und Behebung von Problemen im CAN-Bus-System. Es kann den Kommunikationsstatus, Fehler und Störungen zwischen den Knoten anzeigen und Echtzeitüberwachungs- und Analysetools bereitstellen. Abbildung 17 zeigt ein CANoe-Programm für Testzwecke, bei dem ein entsprechendes CAN-Botschaft ausgegeben wird, wenn das Licht eingeschaltet oder der Schlüssel gedreht wird.

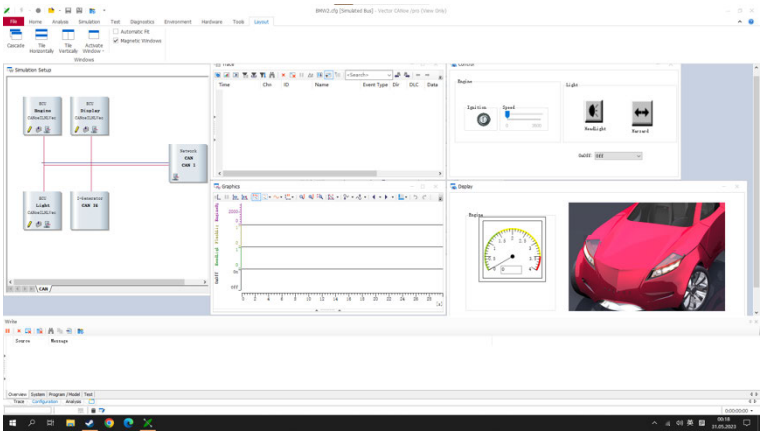

Abbildung 17 Beispiel für die Verwendung CANoe

3.4.5.2 CANdbc++

CANdbc++ ist ein von der Firma Vector Informatik GmbH entwickeltes Datenbankwerkzeug zur Verwaltung und Verarbeitung von Datendefinitionen für das CAN-Bus-System. Es bietet Funktionen zum Erstellen und Bearbeiten von CAN-Datenbanken und ist ein Werkzeug für die Entwicklung und Testung von CAN-Datenbanken.

3.4.5.3 CAPL

CAPL ist eine von Vector Informatik GmbH entwickelte Programmiersprache, und wird speziell für CANoe Knoten Simulation verwendet. Es handelt sich um eine hochrangige Skriptsprache, die verwendet wird, um benutzerdefinierte Funktionen und Logik zum Steuern und Verarbeiten von Daten auf Kommunikationsbussen wie CAN, LIN usw. zu schreiben. Die Ausführung von CAPL-Skripten basiert auf Ereignissen. Das bedeutet, dass das dem jeweiligen Ereignis zugewiesene Programm ausgeführt wird, wenn ein

bestimmtes Ereignis eintritt. Z.B. für Ereignisse sind der Beginn und das Ende einer Messung sowie das Ablaufende eines Timers.

3.5 TinyBMS

Das in diesem Projekt verwendete BMS ist das TinyBMS S516 750A, das von der Firma Energus entwickelt wurde und nun von der Firma Enepaq in Litauen hergestellt wird. Das TinyBMS zeichnet sich durch ein kompaktes Design, umfangreiche Funktionalität, Erweiterbarkeit und Datenkommunikation aus. Das TinyBMS ist klein und eignet sich daher für Anwendungen mit begrenztem Platzangebot. Es bietet eine Vielzahl von Funktionen wie Batterieüberwachung, Ladesteuerung, Entladesteuerung, Temperatur-überwachung, Batterieausgleich und mehr. Das TinyBMS verfügt außerdem über eine intuitive Benutzeroberfläche, die über einen Computer oder ein Mobilgerät gesteuert und angezeigt werden kann. Abbildung 18 zeigt einen Computer-Client mit der Benutzer-schnittstelle des TinyBMS.

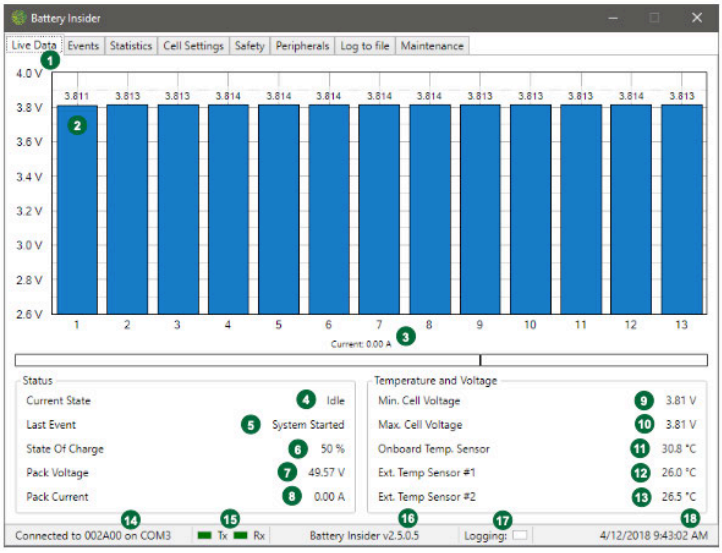

Abbildung 18 Benutzerschnittstelle des TinyBMS [7]

Das TinyBMS kann über CAN-Bus oder UART kommunizieren, um den Batteriezustand auszulesen. Es kann Informationen wie Spannung, Strom, SoC (State of Charge), Batterieereignisse und mehr abrufen. Wenn eine CAN-Kommunikation erforderlich ist, wird ein CAN-UART-Konverter benötigt, da das TinyBMS eigentlich eine UART-Schnittstelle bietet, die in CAN-Botschaft umgewandelt werden muss, bevor sie verwendet werden kann. Abbildung 19 stellt ein TinyBMS dar, das an einen CAN-UART-Konverter angeschlossen ist.

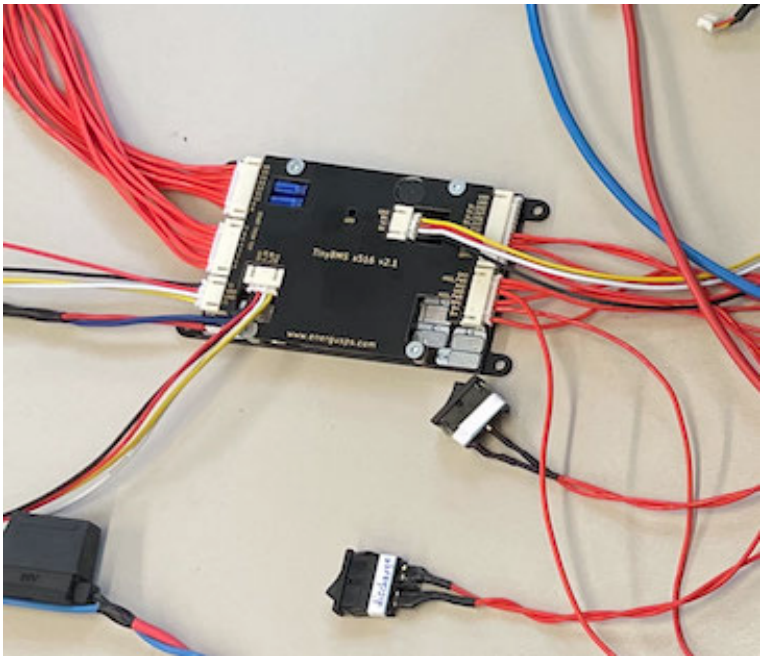

Abbildung 19 TinyBMS

4 Implementierung

Dieser Abschnitt behandelt alle in vorangegangenen Abschnitt erwähnten Funktionen. Es werden detaillierte Informationen über die Implementierung, Parameter und den Zweck des Programms bereitgestellt. Darüber hinaus werden einige Codeausschnitte und die Testergebnisse der Geräte sowie das gesamte System gezeigt.

Die meisten Programme in diesem Projekt wurden mit der Arduino IDE entwickelt, während einige Programme, die mit CANoe zusammenhängen, mit CAPL geschrieben wurden. Die Arduino IDE ist eine Software, mit der Arduino-Programme geschrieben, hochgeladen und debuggt werden können. Es ist die offizielle Entwicklungsumgebung der Arduino-Plattform und bietet eine einfache und benutzerfreundliche Schnittstelle. Die Programmiersprache von Arduino ähnelt der Sprache C. CAPL-Programme wurden anhand von Beispielprogrammen entwickelt, die von CANoe offiziell bereitgestellt wurden.

4.1 Entwicklungsumgebung

Die in diesem Projekt verwendete Version der Arduino IDE ist 1.8.19, die im Dezember 2021 veröffentlicht wurde. Die aktuelle Version der Arduino IDE ist 2.0.1, die im Oktober 2022 aktualisiert wurde. Abbildung 20 veranschaulicht ein Arduino-Beispielprogramm "Blink", das den Client 1.8.19 verwendet. Um sicherzustellen, dass die Programme ordnungsgemäß funktionieren, sind auch die Arduino-Bibliotheken unerlässlich. In diesem Projekt werden die Bibliotheken "due_can", "can_common" und "LiquidCrystal_I2C" verwendet. "due_can" und "can_common" sind Bibliotheken, die für die Kommunikation mit Arduino Due über CAN-Bus relevant sind, und "LiquidCrystal_I2C" eine Bibliothek zur Steuerung der Kommunikation mit einem I2C-LCD verwendet wird.

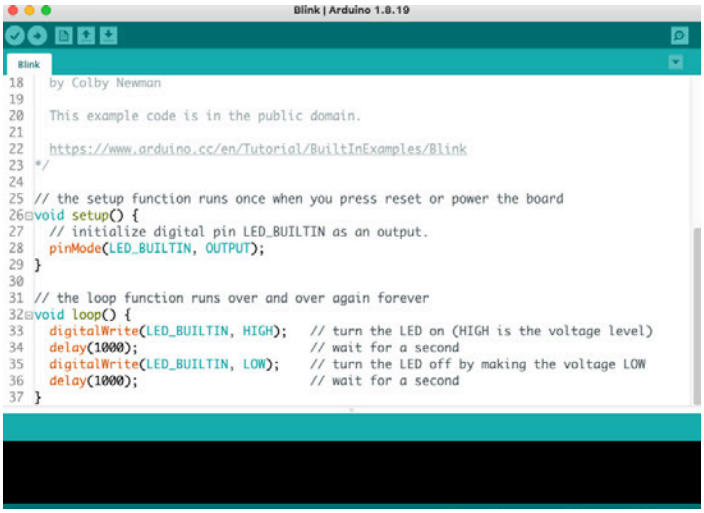

Abbildung 20 Arduino Blink Programm

4.2 Funktionstest der Hardware

In diesem Abschnitt beschreibt den gesamten Prozess der Funktionstest von Hardware. Bei der Verwendung von Jumperkabeln oder Leitungen ist zu beachten, dass die Steuerplatine oder der Arduino vor dem Anschließen der Jumperkabel von der Stromversorgung getrennt werden sollte. Nach dem Anschluss der Jumperkabel sollten die Verbindungen erstmal überprüfen und dann die Stromversorgung der Steuerplatine oder des Arduino wieder einschalten.

Es ist besonders wichtig zu beachten, dass die positive Signalspannung des Arduino 3,3 V beträgt. Eine falsche Signalspannung kann zu Beschädigungen der Arduino-Pins führen. Bei der Verwendung der CAN-UART-Konverter des TinyBMS und der Kommunikation mit dem Computer sollte ein USB-Isolator (Abb. 21) verwendet werden, um eine falsche Spannung zu vermeiden, die den Computer oder das TinyBMS beschädigen könnte.

Abbildung 21 USB-Isolator [13]

4.2.1 LCD Funktionstest

Die erste getestete Hardware ist das LCD. Das LCD kann entweder über mind. 6 Leitungen oder über den I2C-Bus mit dem Arduino kommunizieren. In dieser Aufgabe wurde ein I2C-Bus zum Arduino verwendet, und die Vorteile von I2C-Bus wurden bereits zuvor erwähnt. Durch das Anlöten eines I2C-Backpacks auf der Rückseite des LCDs (Abb. 22) kann es mit nur 4 Leitungen gesteuert werden, wobei 2 Leitungen für die Stromversorgung und die anderen 2 Leitungen für die Übertragung des I2C-Signals verwendet werden. Dies vereinfacht die Verdrahtung erheblich, entlastet den Arduino von Rechenaufgaben und verbessert die Stabilität der Hardware.

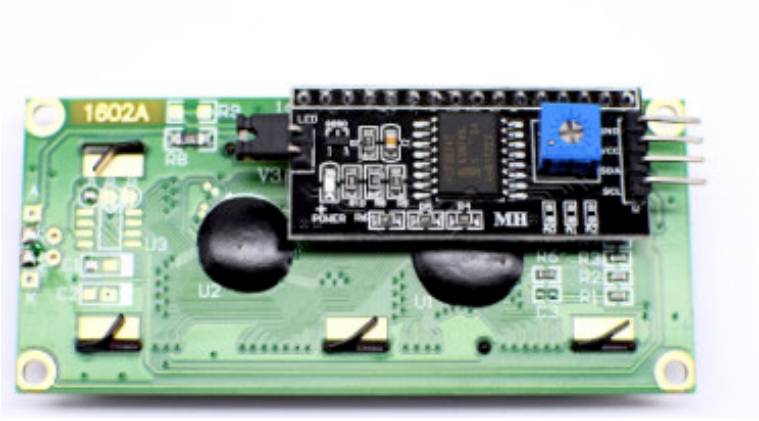

Abbildung 22 Rückseite des LCD mit I2C-Backpack [14]

Die Unversehrtheit der Lötstelle und die Adresse des LCDs werden ebenfalls nach Abschluss des Lötvorgangs geprüft. Wenn die Lötstelle unversehrt ist, werden die Zeichen nicht vollständig oder falsch angezeigt. Die Adresserkennung ist ebenfalls auch notwendig, da unterschiedliche Chips für die I2C-Backpack verwendet werden können. Die Adresserkennung erfolgt mit dem I2C Scanner Code. Dieser Code kann die Geräte auf dem I2C-Bus scannen und die Adresse jedes Geräts erkennen. Durch Ausführen des I2C Scanner Codes können die Adresse des angeschlossenen LCDs ermitteln, um die I2C Kommunikation und Steuerung in Zukunft zu ermöglichen.

Die Helligkeitseinstellung der Hintergrundbeleuchtung des LCDs erfolgt über einen blauen Drehknopf auf dem I2C-Backpack (Abb. 23). Wenn das LCD keine Symbole anzeigen kann, sollten Sie zuerst überprüfen, ob die Spannung +5V beträgt, und versuchen, den Drehknopf für die Helligkeit entsprechend einzustellen.

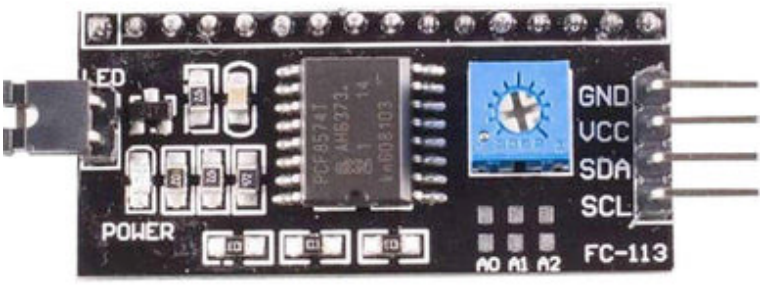

Abbildung 23 I2C-Backpack [15]

4.2.2 Arduino und Steuerplatine Funktionstest

Nach dem Test des LCDs folgt der Test zwischen dem Arduino und der Steuerplatine. Es wird überprüft, ob alle in den Programmen festgelegten Pin-Anschlüsse mit der Steuerplatine übereinstimmen und ob die Funktionen ordnungsgemäß ausgeführt werden können. In diesem Test wurde zuerst auf ein separater Arduino durchgeführt, wobei der bereits auf der Steuerplatine platzierte Arduino nicht verwendet wurde. Um mögliche Schäden an der Steuerplatine aufgrund von Bedienungsfehlern zu vermeiden, wurde dies unternommen.

Die Schritte umfassen das Hochladen des Demo-Programms auf den Arduino und das wiederholte Testen der Schalter, um sicherzustellen, dass der Arduino Informationen korrekt erkennt und die Operationen und Fahrzeugzustände korrekt auf dem LCD angezeigt werden. Nach erfolgreichem Test auf dem separaten Arduino wird das Demo-Programm auf den Arduino der Steuerplatine hochgeladen und erneut getestet. Dieses Demo-Programm dient nur dazu, die Schaltung und die korrekte Herstellung der Steuerplatine zu überprüfen und sendet keine tatsächlichen CAN-Botschaften aus.

4.2.3 VN1610 Funktionstest

Im Projekt wurde die Vector VN1610 als CAN-Hardware verwendet, um die CAN-Botschaften zu senden und zu analysieren, in Kombination mit CANoe. Die VN1610 verfügt über einen USB-Anschluss an einem Ende und einen D-Sub-Anschluss am anderen Ende. Die VN1610 kann gleichzeitig zwei CAN-Botschaften empfangen und senden, jedoch wurde in der Aufgabe nur ein Kanal verwendet. Dieser Kanal wurde über einen D-Sub-Anschluss mit zwei Drähten verbunden. Die gelbe Leitung (Abb 24) steht für CAN-Low und die grüne Leitung für CAN-High.

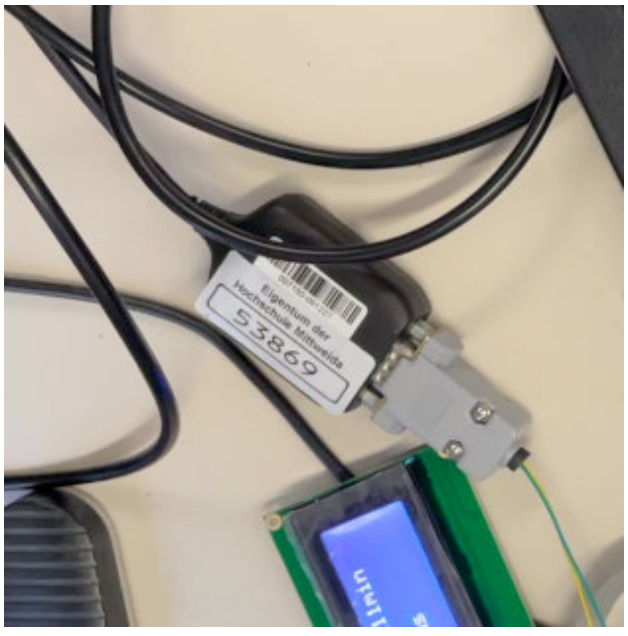

Abbildung 24 CAN Hardware VN1610

Auf der Steuerplatine ist ein CAN-Bus Shield für die Implementierung der Arduino CAN-Funktionalität vorhanden. Es bietet einen CAN-Controller und die entsprechende Schaltung, um die Kommunikation zwischen dem Arduino und dem CAN-Bus zu ermöglichen. Durch die Verbindung dieser beiden Leitungen mit den entsprechenden CAN-Shield auf der Steuerplatine können die CAN-Botschaften gelesen werden. Ein direkter CAN-Anschluss an den Arduino ist nicht möglich, da der Arduino nicht über CANHigh oder CANLow verfügt, sondern nur über CANTX (Senden) und CANRX (Empfangen).

4.2.4 TinyBMS und CAN Konverter Funktionstest

Die letzte Hardware-Testphase umfasst die CAN-Kommunikation zwischen der Steuerplatine, dem Arduino und dem TinyBMS. Die Kommunikation zwischen Arduino und TinyBMS erfolgt über CAN-Bus, wobei auf der Steuerplatine eine CAN-Shield (Abb. 25) für den Empfang und Versand von CAN-Botschaften zuständig ist. Die CAN-Botschaften werden über ein Kabel und einen CAN-UART-Konverter in UART-Signale umgewandelt und an das TinyBMS übertragen. Da die Schnittstellen auf dem TinyBMS nur UART-Signale akzeptieren und senden können, ist der CAN-UART-Konverter notwendig. Der Datentransfer erfolgt über eine Zwei-Draht-Leitung in differentieller Weise. Alle CAN-Botschaften für das TinyBMS sind in der Kommunikationsanleitung des TinyBMS festgelegt. In diesem Projekt wird eine CAN-Bus-Bitrate von 500 KBit/s und eine serielle Übertragungsrate von 115200 Bit/s verwendet. Diese Einstellungen wurden gemäß den technischen Anforderungen von TinyBMS vorgenommen. Falsche Einstellungen können zu Kommunikationsfehlern führen.

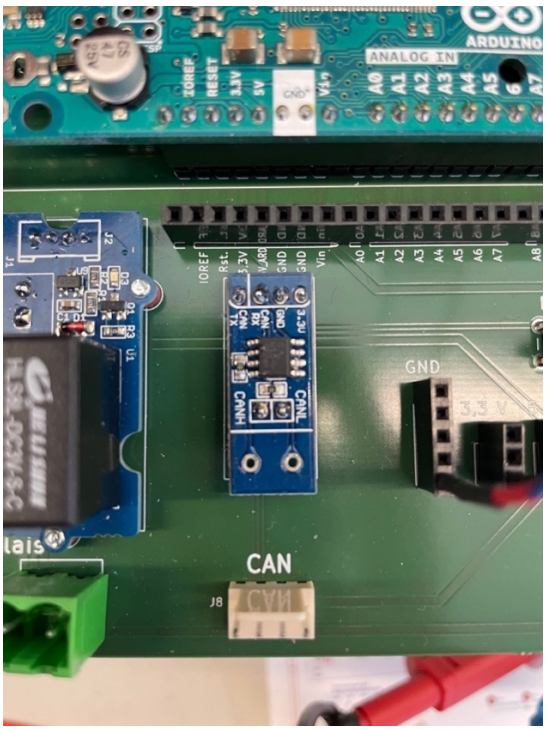

Abbildung 25 CAN Shield und CAN Schnittstelle auf der Steuerplatine

4.3 Arduino Programmierung

Um die Funktionen der Hardware zu testen, ist es erforderlich, für jeder Hardware ein entsprechendes Testprogramm zu erstellen. Da der Arduino in dieser Aufgabe eine viele Code erfordert, besteht die Lösungsidee darin, das Ziel in kleinere Teile aufzuteilen. Durch den Test der einzelnen Unterprogramme kann die Zuverlässigkeit überprüft und gleichzeitig die Hardwarefunktionen getestet werden. Schließlich werden alle Programme zusammengefügt, um das endgültige Programm zu erstellen.

4.3.1 LCD Programm

Nach der Überprüfung der LCD-Adresse und die Unversehrtheit der Lötverbindungen beginnen mit der Programmierung für die Aufgabe. Dieses Programm beinhaltet nur den Test von Grafiken und hat keine direkte Verbindung zum tatsächlichen Schaltkreis oder der Steuerplatine.

Unabhängig davon, ob vorwärts oder rückwärts fahren, sollte ein Pfeil (Abb. 26) nach vorne oder hinten angezeigt werden. Die visuelle Darstellung eines Pfeils ist deutlich anschaulicher als eine textliche Beschreibung der Fahrtrichtung. Obwohl die Höchstgeschwindigkeit des Rasentraktor auf 10 km/h begrenzt ist, ist es auf einem holprigen Fahrzeug einfacher, einen Pfeil zu erkennen als Text. Zudem hat das LCD eine Anzeigefläche von 20x4, und durch die ausschließliche Verwendung von Text wird das Display nicht vollständig ausgenutzt.

Die Gestaltung des Pfeils berücksichtigt daher die bestmögliche Nutzung der vorhandenen LCD-Zeichen (ASCII Zeichen), während eigene Zeichen möglichst wiederverwendet werden sollten. Nach wiederholten Experimenten und Vergleichen wurde die aktuelle Symbolkombination für den Pfeil ausgewählt. 0xFF repräsentiert das im LCD integrierte ASCII-Zeichen, das eine vollständig schwarze Darstellung ermöglicht. Die eigenen Zeichen können durch einfache Berechnungen erstellt werden, um den gewünschten Zeichenstring zu erhalten. Diese Zeichenstrings müssen erste im Arduino gespeichert und benannt werden, um sie bei Bedarf direkt abrufen zu können.

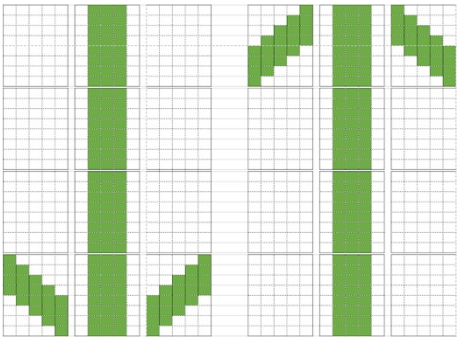

Abbildung 26 Pfeilzeichen

Der Batteriestatus kann direkt numerisch angezeigt werden, jedoch wird in der Aufgabe zusätzlich eine grafische Darstellung verwendet, um schnell den Ladezustand der Batterie (voll, halbvoll, weniger als halbvoll oder fast leer) erkennen zu können. Die Batteriegrafiken sind auch eigens erstellt und müssen im Voraus berechnet, im Arduino gespeichert und benannt werden, um sie bei Bedarf direkt abrufen zu können. Die Testergebnisse des LCDs sind in Abbildung 27 dargestellt.

Abbildung 27 LCD Testergebnis

4.3.2 Schaltung Test Programm

In diesem Programm werden nun testen, ob der geschriebene Code den angegebenen Anforderungen entspricht und unter den angegebenen Schalterbedingungen das richtige Feedback liefert. Z.B. sollte Arduino bei Auswahl des Vorwärtsgangs die Schaltkreisinformationen korrekt lesen und auf dem LCD anzeigen sowie später über den CAN-Bus senden. Es ist entscheidend, festzustellen, ob jemand auf dem Sitz sitzt. Wenn der Rasentraktor ohne Fahrer startet, kann dies zu Schäden führen. Das Ablaufschema des Programms ist in Abbildung 28 dargestellt.

Die Vorbereitung für die Fahrt kann nur erfolgen, wenn der Sitzschalter geschlossen ist. Der Status des Rasenmähers muss nur als "Ein" oder "Aus" angezeigt werden und erfordert keine zusätzliche grafische Darstellung. Es ist zu beachten, dass an dieser Stelle "Maehwerk" verwendet werden sollte anstelle des deutschen Zeichens "ä", da dies auf dem LCD möglicherweise zu falschen Zeichen führen könnte. Die Testergebnisse sind in Abbildung 29 dargestellt.

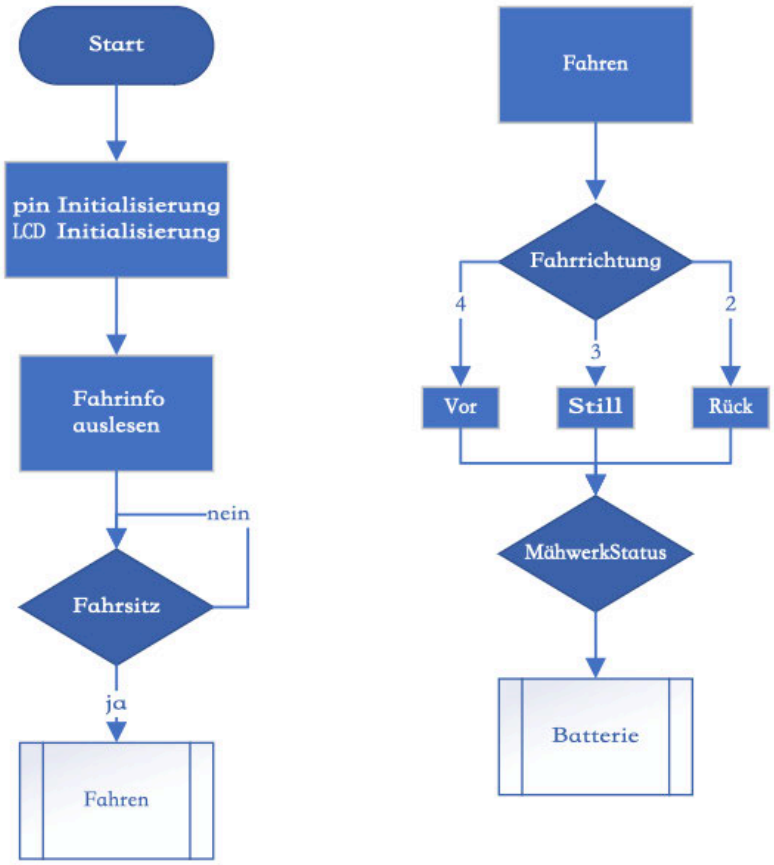

Abbildung 28 Ablaufplan des Testprogramms für die Schaltung

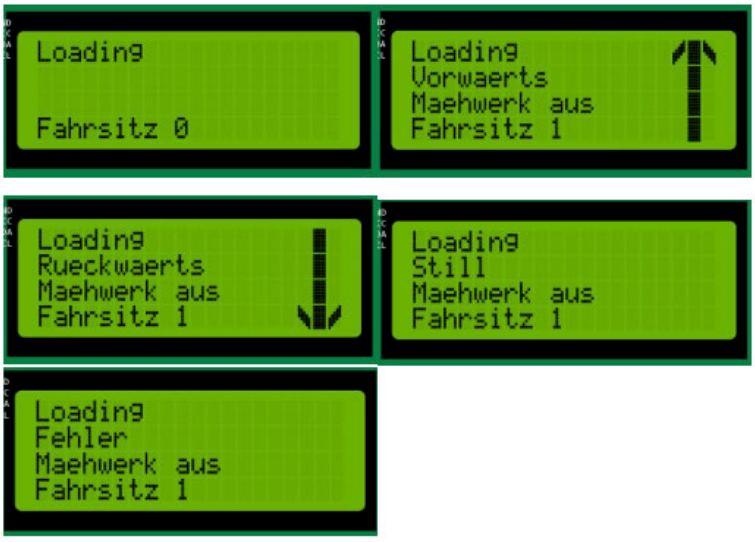

Abbildung 29 Testergebnis des Programms für die Schaltung

Die Bildanzeige auf dem LCD sollte den Fahrzeugstatus (Vorwärts, Rückwärts oder Stillstand), den Batteriestand, die Restlaufzeit und den Status des Rasenmähers umfassen. Der Batteriestand muss über CAN-Bus von der TinyBMS gelesen werden. Da der CAN-Bus-Test noch nicht durchgeführt wurde, kann der tatsächliche Batteriestand nicht bestimmt werden und die Restlaufzeit kann ebenfalls auch nicht berechnet werden. Um die Anzeige des Batteriebildes zu testen, wird mit einer For-Schleife ein virtueller Batteriestand von 1 bis 100 erstellt. Die Restlaufzeit wird nur als einfaches Beispiel auf dem LCD angezeigt und die Berechnung erfolgt später. Wenn der Batteriestand unter 10% fällt, blinkt der LCD-Bildschirm, um anzuzeigen, dass die Batterie fast leer ist. Die Testergebnisse sind in Abbildung 30 dargestellt.

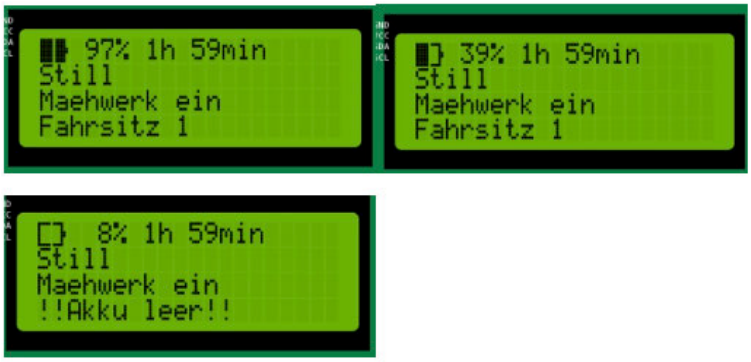

Abbildung 30 Batteriestand getestet von 100 bis 0

4.3.3 Selbsthaltung Programm

Die Selbsthaltung ist ein wichtiger Teil des Programms, da der Zündschlüssel nur einmal in der Position "S2" gedreht wird. Daher muss ein Statuswert gespeichert werden. Pin 34 auf der Steuerplatine ist der Selbsthaltungspin. Das Fahrzeug kann nur in den Fahrzyklus (S2 Fahrroutine) eintreten, wenn Pin 34 auf "HIGH" ist. Andernfalls kann es nur in den Vorbereitungszyklus (S1 Wake-Up Routine) eintreten. Das Ablaufschema des Programms ist in Abbildung 31 dargestellt.

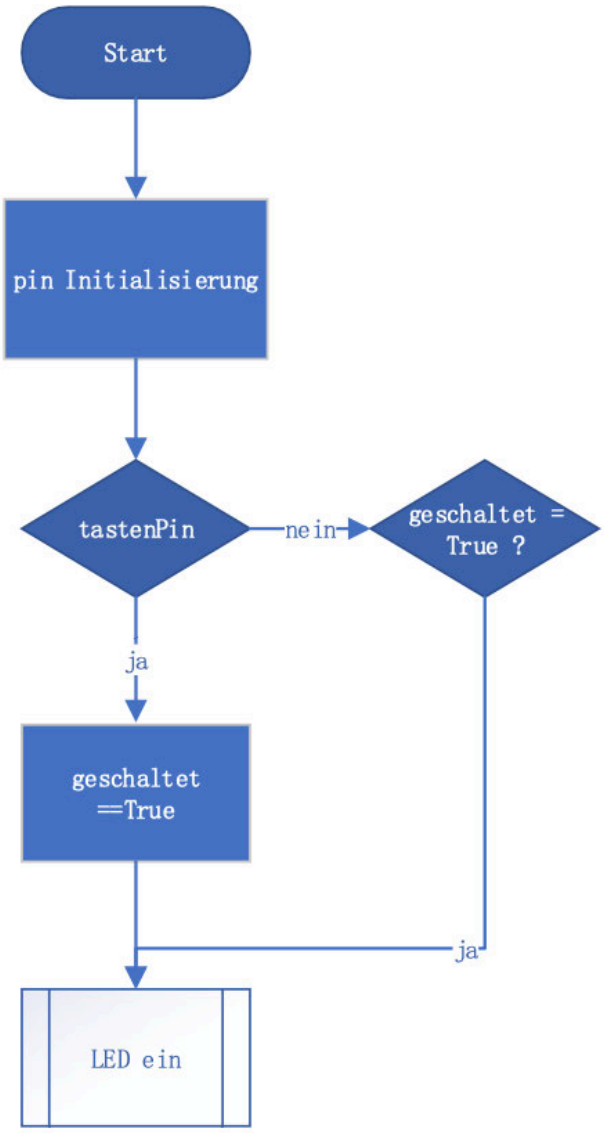

Abbildung 31 Ablaufplan des Testprogramms für die Selbsthaltung

4.3.4 CAN-Bus Kommunikationsprogramm

CAN-Kommunikationsprogramm wird verwendet, um zu testen, ob zwischen zwei CAN-Bus-Komponenten eine Kommunikation aufgebaut werden kann. Hier wurde zunächst ein Empfangsprogramm und ein Sendeprogramm für Arduino erstellt, um die Kommunikation zwischen den beiden Arduino zu testen (Abb. 32). Der eine Arduino dient als Empfänger, und der andere Arduino als Sender agiert. Nach erfolgreichem Empfang können die gesendeten CAN-Botschaften im Arduino Serial-Monitor beobachtet werden.

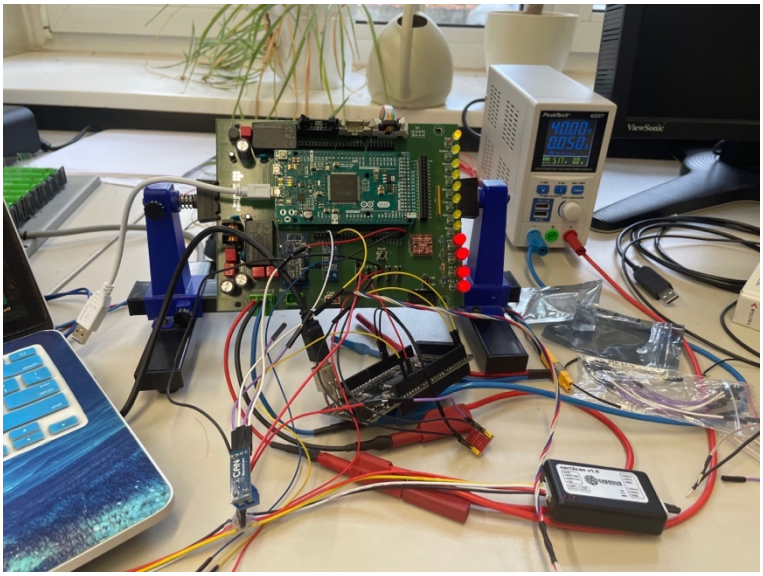

Abbildung 32 CAN Kommunikationstest zwischen 2 Arduino Due

4.3.5 TinyBMS Auswertungsprogramm

Nach Abschluss aller bisherigen Tests kann der letzte Test der TinyBMS CAN-Kommunikation durchgeführt werden. Die Kommunikation zwischen Arduino und TinyBMS erfolgt gemäß den in der TinyBMS-Kommunikationsprotokolle angegebenen IDs und Botschaften. Das Programm liest CAN-Converter-Knoten-ID, Spannung, Strom und SoC (State of Charge), die für spätere Berechnungen nützlich sind. Die Knoten-ID des CAN-Konverter kann geändert werden und erfordert besondere Aufmerksamkeit. Das Ablaufschema des Programms ist in Abbildung 33 dargestellt.

Nach dem Lesen der erforderlichen Werte werden Berechnungen durchgeführt und die benötigten Werte an das Hauptprogramm zurückgegeben. Allerdings ist zu beachten, dass die Daten für Spannung und Strom im Float-Format (*float*) und die Daten für den SoC im Integer-Format (*int*) vorliegen. Wenn die Batterie geladen wird, ist der Stromwert positiv, während er im Entladezustand negativ ist.

Die Formel zur Berechnung der Restlaufzeit lautet wie folgt:

$$Restlaufzeit = \frac{SollkWh * SoC}{Spannung * Strom}$$

Ein Teil des Codes aus dem TinyBMS-Test ist in Abbildung 34 dargestellt.

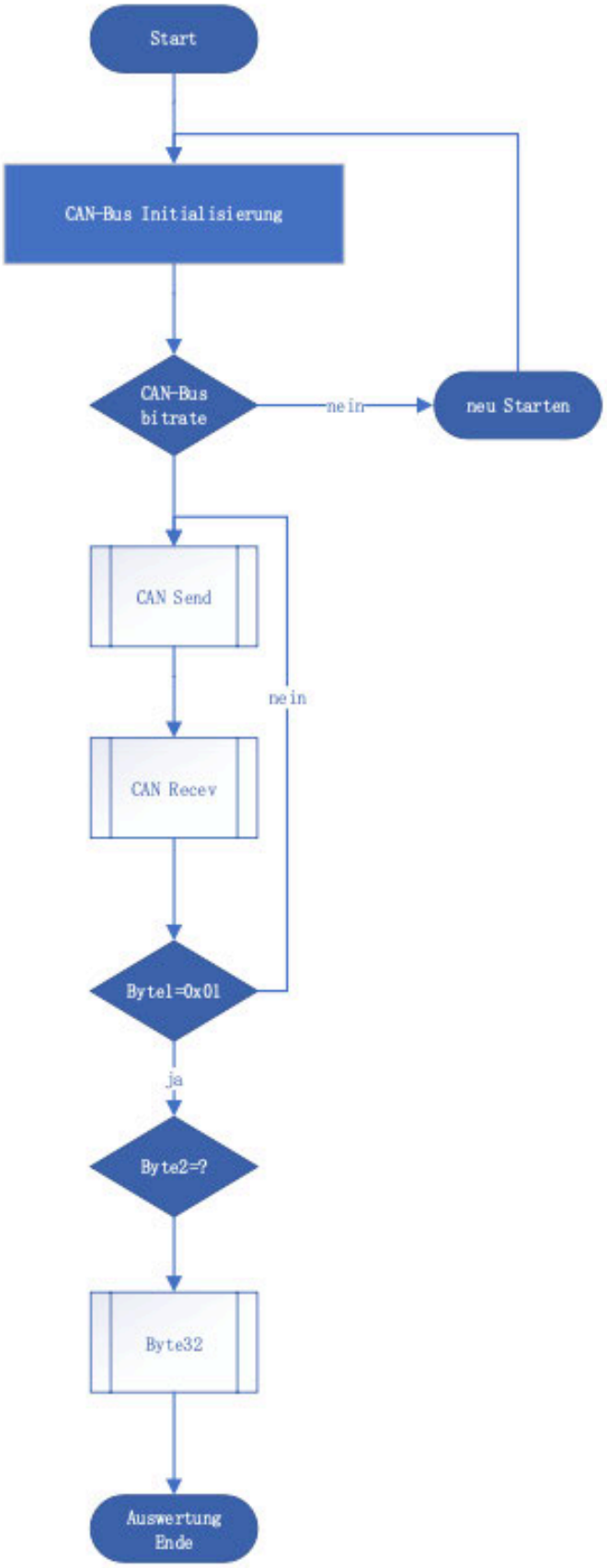

Abbildung 33 Ablaufplan des Testprogramms für die TinyBMS

```
51 void loop()
52 {
53   //ID auslesen
54   Serial.println("ReadID");
55   CAN_Send(0x218, 1, txRead);
56   CAN_BMS_Recev();
57
58   //SOC auslesen
59   Serial.println("SOC");
60   CAN_Send(0x218, 1, txSOC);
61   CAN_BMS_Recev();
62
63   //Spannung auslesen
64   Serial.println("Spannung");
65   CAN_Send(0x218, 1, txVolt);
66   CAN_BMS_Recev();
67
68   //Strom auslesen
69   Serial.println("Strom");
70   CAN_Send(0x218, 1, txCurr);
71   CAN_BMS_Recev();
72
73   //Restlaufzeit rechnen
74   float SollkWh = IstkWh * SOC_akt / 100;
75   //P_akt = Volt_akt * Curr_akt; nun es gibt keine echte Strom, also P= 0
76   P_akt = startLeistung;
77   float Restzeit = SollkWh / P_akt;
78
79   //Ausgabe BMS Auswertung
80   Serial.print("Leistung : ");
81   Serial.println(P_akt);
82   Serial.print("SollkWh : ");
83   Serial.println(SollkWh);
84   Serial.print("Restlaufzeit : ");
85   Serial.println(Restzeit);
86   Serial.println("Ende des Empfangs");
87 }
```

Abbildung 34 Teil des Codes im TinyBMS-Test.

Das TinyBMS und der Arduino verwenden entgegengesetzte LSB- und MSB-Anordnungen, werden die vom Arduino gelesenen Daten im Vergleich zur tatsächlichen Situation invertiert. So wird ein weiteres Programm benötigt, um sie zu invertieren. Der Code für dieses Unterprogramm ist in der folgenden Abbildung 35 dargestellt.

```
188 uint32_t Byte32 (uint8_t Bytea, uint8_t Byteb, uint8_t Bytec, uint8_t Byted)
189 {
190   //Bildung von 4 x 8bit Daten zu 32bit Daten
191   uint32_t Bytecan;
192   Bytecan = Bytea; //1 Byte speichern
193   Bytecan <<= 8; //Verschiebung 8 Bits nach links, beginnend bei 0x100 statt 0x01
194   Bytecan = Bytecan | Byteb;//OR-Operation, d.h. 0x100 wird mit 0x71 verbunden, also 0x171
195   Bytecan <<= 8;
196   Bytecan = Bytecan | Bytec;
197   Bytecan <<= 8;
198   Bytecan = Bytecan | Byted;
199   return Bytecan;//Erhalten die zusammengeführten 32Bit Daten
200 }
```

Abbildung 35 Teil des Codes in der Umkehrung der LSB und MSB

4.4 Abschlussprogramm

Wie bereits erwähnt, enthält das Abschlussprogramm viele der vorherigen Unterprogramme. Es handelt sich jedoch nicht einfach um eine einfache Kombination, sondern um eine sorgfältige Aufteilung und erneuert Neukombination dieser Unterprogramme. Es sind auch einige Änderungen an den Unterprogrammen erforderlich. Auch der Ablauf des Programms, die logische Beurteilung, wurde geändert. Z.B. einige Variablen sind allgemein und nicht auf ein bestimmtes Unterprogramm beschränkt, während einige Unterprogramme teilweise Werte zurückgeben müssen. Es ist auch wichtig, auf das Datenformat zu achten. In den CAN-Botschaften, die von TinyBMS zurückgegeben werden, sind einige Werte als Ganzzahlen (*int*) und andere als Gleitkommazahlen (*float*) formatiert. Außerdem müssen die empfangenen CAN-Botschaften sortiert werden. Alle CAN-Botschaften, die sich auf das TinyBMS gasenden werden, verwenden dieselbe CANID, und die einzige Möglichkeit, sie zu unterscheiden, besteht darin, ein zweites Byte zu klassifizieren.

```
321  if (0x01 == Byte1) { //falls 1.Byte nicht 0x01, CANSignal ist falsch
322    Serial.println(" CAN Auslesen erfolgreich!");
323  }
324  else {
325    Serial.println(" CAN Auslesen Fehler! Bitte erneut machen");
326    switch (Byte2) ///Das zweite Byte kann als Identifizierung verwendet werden
327    {
328      case 0x1A://SOC
329        Serial.print("SOC Auslesen Fehler");
330        break;
331      case 0x14://akt Spannung
332        Serial.print("Spannung Auslesen Fehler");
333        break;
334      case 0x15://akt Strom
335        Serial.print("Strom Auslesen Fehler");
336        break;
337      default:
338        Serial.println("unbekannt CAN-Botschaft,bitte erneut starten!");
339        break;
340    }
341    return;/*Falsche CAN-Signale werden nicht akzeptiert und werden in der nächsten
342    Runde neu berechnet.*/
343  }
```

Abbildung 36 Teil des Codes in die TinyBMS CAN-Botschaften Sortierung

Wenn ein Fehler in den gelesenen Daten auftritt, wird dieser einfach ignoriert und die nächsten Daten werden gelesenen. Die fehlerhaften Daten werden im nächsten Zyklus gelesenen.

Es ist auch zu beachten, dass das CAN-Sende-Programm in Zukunft möglicherweise für die zukünftige Antrieb- und Mähwerkmotorsteuerung wiederverwendet werden kann, aber das CAN-Empfangsprogramm für TinyBMS nicht wiederverwendet werden kann. Aufgrund der Besonderheiten der Tiny-BMS CAN-Kommunikation sind die ID- und Datenformate bereits festgelegt, und dieses Unterprogramm ist nun speziell für TinyBMS entwickelt. Wenn dieses CAN-Empfangs-programm verwenden möchten, müssen einige Änderungen vornehmen. Das genaue Format der CAN-Botschaften für die zukünftige Antrieb- und Mähwerkmotorsteuerung können noch nicht bestimmt werden. Das

Programm für Datei Inversion ist auch nicht immer dasselbe, in TinyBMS müssen alle 4 Daten umgedreht werden. Das gesamte Ablaufschema des Programms ist in Abbildung 37 und 38 dargestellt

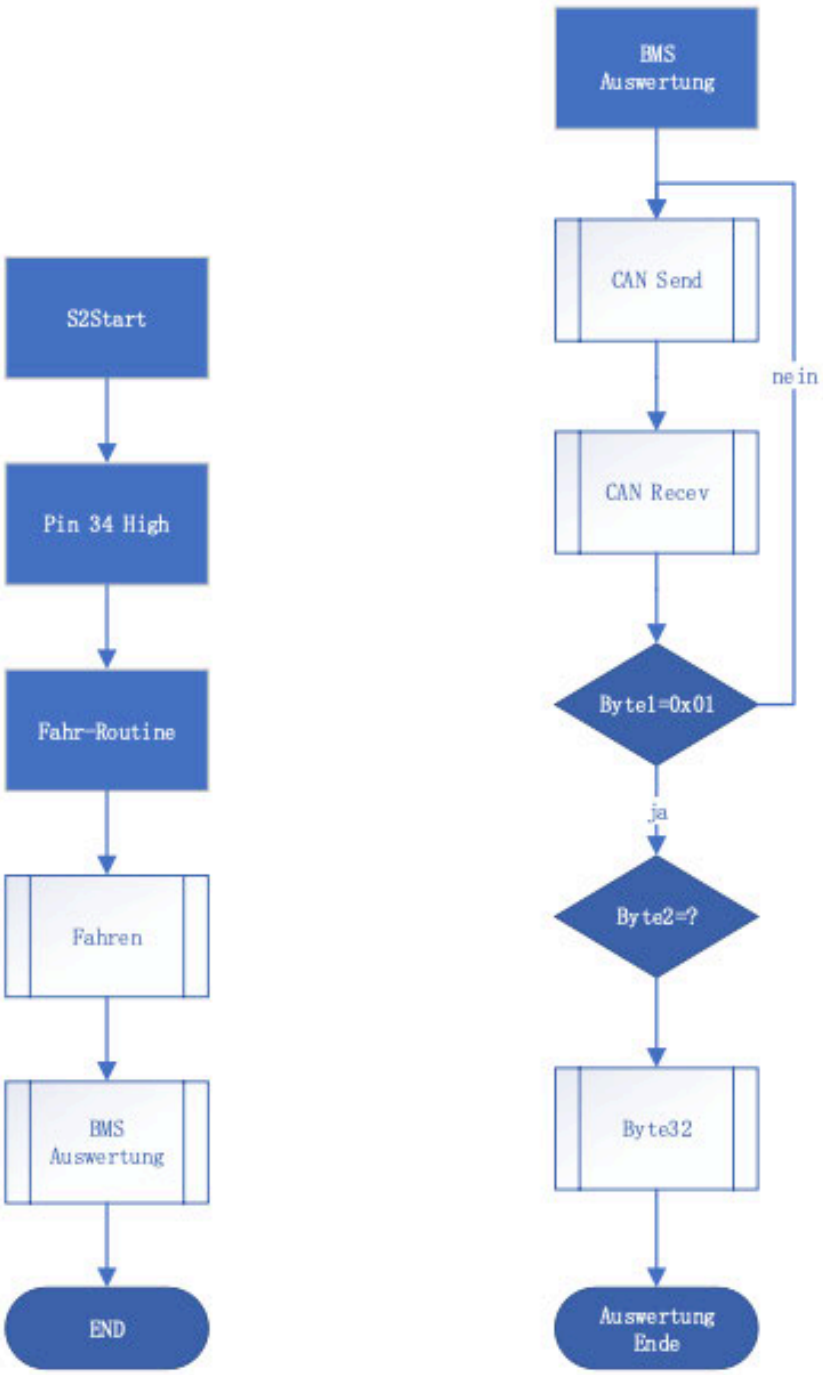

Abbildung 37 Unterprogramm S2Start und TinyBMS Auswertung

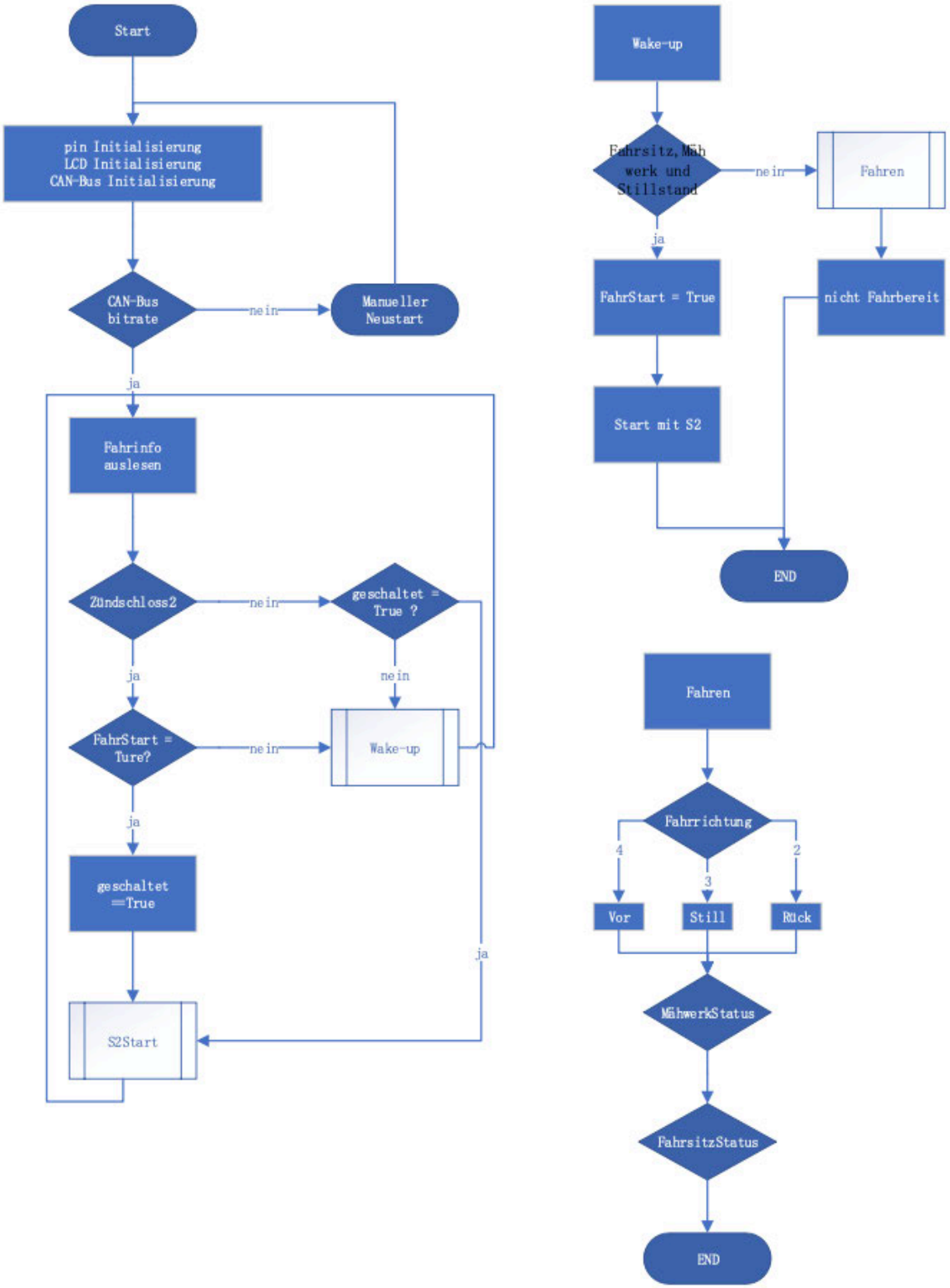

Abbildung 38 Das Ablaufschema des gesamtprogramms

Generell läuft das Programm folgendermaßen ab: Beim Einstecken des Schlüssels und Drehen in die Position S1 startet das VCU die Wake-up-Routine. Dabei werden kontinuierlich Fahrzeuginformationen ausgelesen, bis folgende Bedingungen erfüllt sind: Der Fahrersitz ist besetzt, das Fahrzeug steht im Stillstand und der Mähwerk ist ausgeschaltet. In diesem Fall wird auf dem LCD "START S2" angezeigt. Andernfalls wird "nicht Fahrbereit" angezeigt. Alle drei Bedingungen müssen erfüllt sein. Die Ergebnisse sind in Abbildung 39 dargestellt.

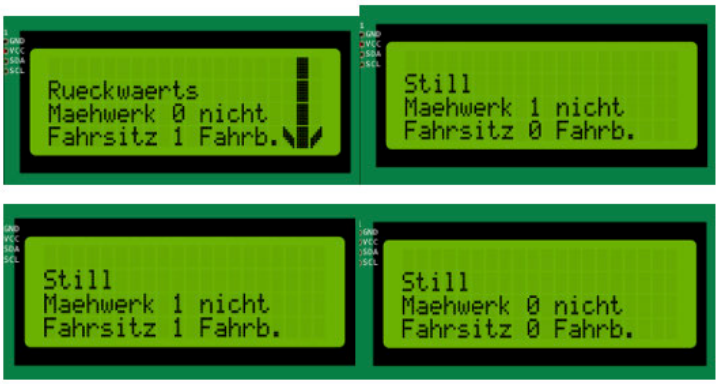

Abbildung 39 Ergebnisse von Wake-up-Routine

Wenn auf dem LCD "START S2" angezeigt wird und der Schlüssel einmal in die Position S2 gedreht wird, wechselt das VCU in die Fahrroutine und beginnt mit dem Auslesen der TInyBMS-Informationen. In der Fahrroutine reagiert das VCU entsprechend auf alle Schalter und zeigt die entsprechenden Informationen auf dem LCD an. Die Ergebnisse sind in Abbildung 40 dargestellt. Die endgültigen Testergebnisse sind in Abbildung 41 dargestellt. Das TinyBMS und die Steuerplatine wurden erfolgreich gelesen.

Abbildung 40 Ergebnisse von Fahr-Routine

Abbildung 41 Endgültigen Testergebnisse

4.5 CANoe Programmierung und Nutzung

Bevor ein Programm in CANoe schreiben, müssen mögliche Variablen in Programm identifiziert und kategorisiert werden. Dabei sollte auch die Namensgebung, Datentypen und Längen festlegen. In diesem Prozess ist eine DBC-Datei eine Voraussetzung, jedoch kann sie je nach den Anforderungen der späteren Entwicklung geändert werden. Alle erforderlichen Dosensignale sind in Abbildung 42 enthalten. Abbildung 43 zeigt die wie Abbildung 42 vorbereitete dbc-Datei.

	Gerät	Position	Funktion	Signalübertraung	Sender	Empfänger	Reaktion	CAN-Signal	Wert
Von VCU	Richtungswahlschalter	rechts am Fahrersitz	Vorwärts Stillstand Rückwärts	3,3V-Signal (Ein) " "	VCU VCU VCU	MC Antrieb MC Antrieb MC Antrieb	VCU sendet via CAN das Richtungssignal an MC Antrieb	VCU_MC_Antrieb_Vorwärts VCU_MC_Antrieb_Stillstand VCU_MC_Antrieb_Rückwärts	1 1 1
	Schlüsselschalter	Amaturenbrett	0-AUS 1-Zündung EIN 2-Fahrbetrieb	0,0V-Signal in Stellung 0 3,3V-Signal 3,3V-Signal	VCU	MC Antrieb	Fahrzeug im Sleep-Modus Fahrzeug wake up Hauptrelais geschlossen	VCU_MC_Antrieb_Stopp VCU_MC_Antrieb_Stopp VCU_MC_Antrieb_Start	1
	Lenkwinkel-Resolver	Lenksäule unter Amaturenbrett	Lenkwinkel	0 - 2,5...5V links - mitte...rechts Auswertung erfolgt im IC	VCU	MC Antrieb	VCU legt den Lenkwinkel für MC Antrieb auf den CAN	VCU_MC_Antrieb_PHI_Lenk	-45° ..0 ..45°
	Gaspedal	Fußbremse	Sollwertvorgabe für Antriebe	analog 0...5V	VCU	MC Antrieb	VCU legt den n-Sollwert für MC Antrieb auf den CAN	VCU_MC_Antrieb_n_soll	0 - 1500 1/min
	Drehzahlmesser Vorderrad links und rechts	linkes und rechtes Vorderrad	links-vorn und rechts-vorn	x Impulse pro Umdrehung	VCU	MC Antrieb	VCU legt den n-Istwert_links und den n-Istwert_rechts für MC Antrieb auf den CAN	VCU_MC_Antrieb_n_links_ist VCU_MC_Antrieb_n_rechts_ist	0 - 1500 1/min mit iG multiplizieren
	Sitzkontakt	unter dem Fahrersitz	beim Aussteigen des Fahrers müssen alle Antriebe abgeschalten werden Sicherheitsfunktion!!!	5V Endlagenschalter	VCU	MC Antrieb MC Mähwerk	VCU sendet via CAN an alle MC das Stopp-Signal	VCU_MC_Antrieb_Stopp VCU_MC_Mähwerk_Stopp	0
	Mähschalter	Amaturenbrett	AUS EIN	0,0V-Signal 3,3V-Signal	VCU	MC Mähwerk	VCU sendet an MC Mähwerk die Botschaft 1 oder Null	VCU_MC_Mähwerk_Stopp VCU_MC_Mähwerk_Start	0/1
	Display I2c	Amaturenbrett	Anzeige 4Zeilen 20Zeichen 2 Seiten	I²C	VCU	Display		VCU_Display_Zeile_1_SOC VCU_Display_Zeile_2_I_Antrieb VCU_Display_Zeile_3_I_Mähwerk VCU_Display_Zeile_4_Restlaufzeit	
Von BMS	Drehzahlmesser Vorderrad links und rechts	linkes und rechtes Vorderrad	links-vorn und rechts-vorn	x Impulse pro Umdrehung	BMS	MC Antrieb	BMS legt den n-Istwert_links und den n-Istwert_rechts für MC Antrieb auf den CAN	BMS_MC_Antrieb_n_links_ist BMS_MC_Antrieb_n_rechts_ist	0 - 1500 1/min mit iG multiplizieren
	BMS SOC	hinter	State of Charge		BMS	VCU		BMS_VCU_SOC	
	BMS Restlaufzeit	hinter	Restlaufzeit		BMS	VCU		BMS_VCU_Rest_laufzeit	

Abbildung 42 CAN Botschaften

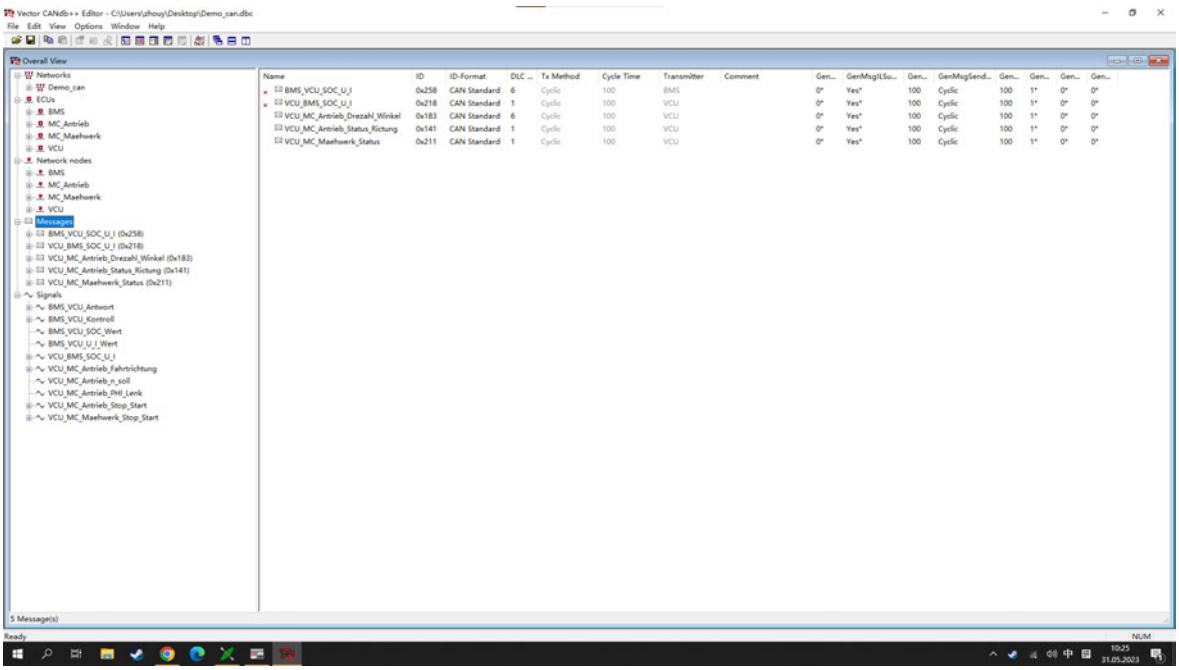

Abbildung 43 dbc-Datei

CAPL wird hauptsächlich verwendet, um Funktionsgleichungen zu simulieren oder in Beziehung zu setzen (Abb. 44), und wird für Tests verwendet, wenn das Gerät noch nicht vollständig in Betrieb genommen wurde.

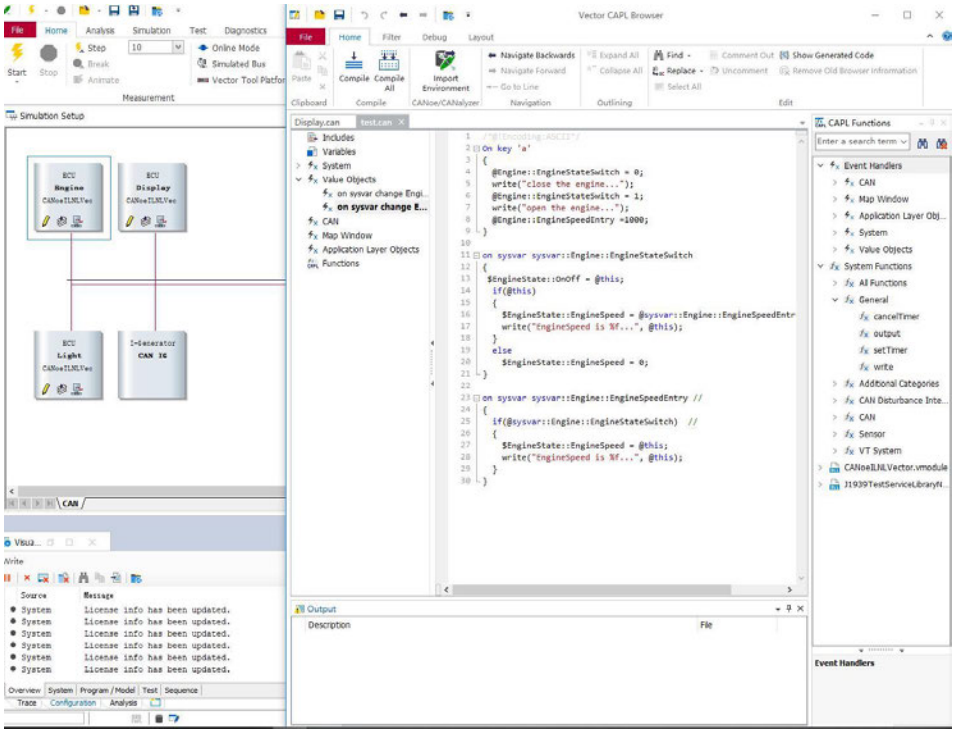

Abbildung 44 CAPL Programm

Die grafischen Fenster von CANoe können im Panel Designer (Abb. 45) gefunden werden, der eine breite Palette von Werkzeugen bietet, darunter Schalter, LED-Anzeigen und vieles mehr.

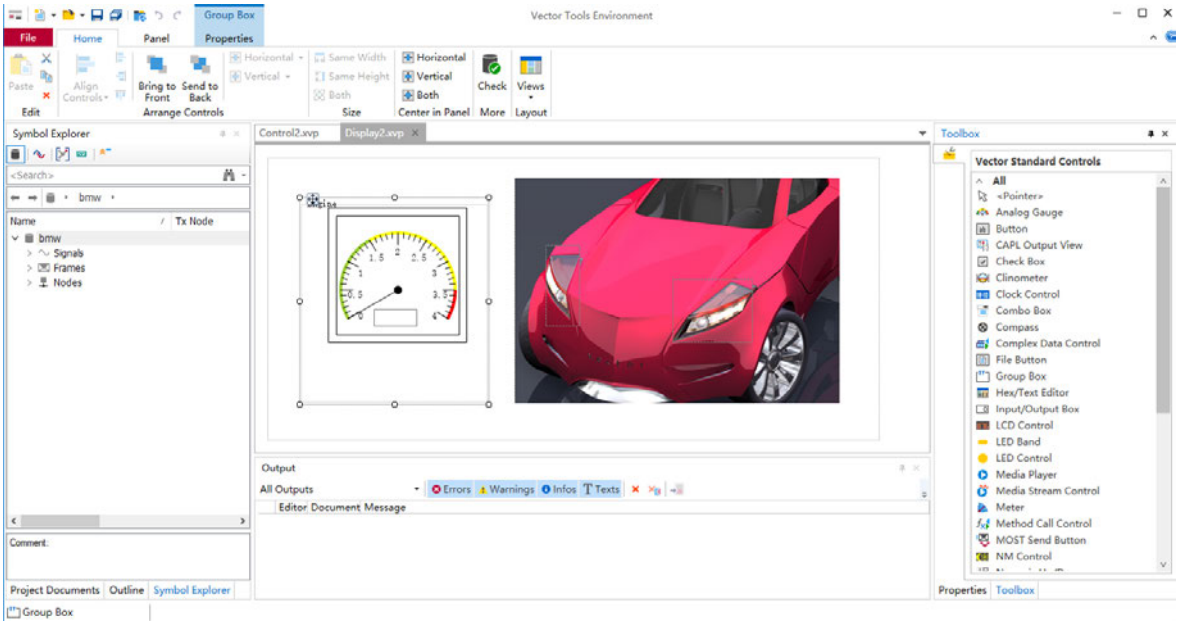

Abbildung 45 Panel Designer

5 Softwareverifikation

Für den Arduino ist es wichtig, dass die empfangenen CAN-Botschaften genau mit den Daten übereinstimmen, die von CANoe überwacht werden und die vom TinyBMS-Client erhalten werden. Andernfalls kann nicht davon ausgegangen werden, dass die Daten erfolgreich empfangen wurden. Achten besonders auf die Daten für Spannung, Strom und SoC. Deren CAN-Botschaften-IDs sind jeweils 0x14, 0x15 und 0x1A. Alle Originaldaten befinden sich im Anhang. Hier wurden nur die nützlichen Teile übernommen, Duplikate entfernt und die Daten in Tabellen formatiert. Während des Entladevorgangs nehmen sowohl SoC als auch Spannung ab, der Strom bleibt jedoch zunächst konstant, was mit den Ergebnissen eines anderen Praktikums vollkommen übereinstimmt. Die Tabellen 2, 3 und 4 sind alle aus TinyBMS ausgelesenen Werte.

0x1A	SOC	int
HEX	DEC	Wert in %
0512f050	85127248	85.127248
0512e7d0	85125072	85.125072
04b51f6c	78978924	78.978924
04b4fec9	78970569	78.970569
04b4de26	78962214	78.962214
04b4bd83	78953859	78.953859
04b49ce0	78945504	78.945504

Tabelle 2 Ausgelesene SoC-Werte

0x14 Spannung	float
HEX	Wert
42826354	65.194
42758c4a	61.387
4275872b	61.382
4275851f	61.38
42758000	61.375

Tabelle 3 Ausgelesene Spannung-Werte

0x15 Strom	float
HEX	Wert
bf82f1aa	-1.023

Tabelle 4 Ausgelesene Strom-Werte

Abbildung 46 und 47 zeigen Daten, die mit CANoe und Arduino ausgelesen wurden. Es ist zu beachten, dass die Daten für 011A in beiden Fällen identisch sind, daher kann man davon ausgehen, dass Arduino erfolgreich das CAN-Botschaften von TinyBMS gelesen hat.

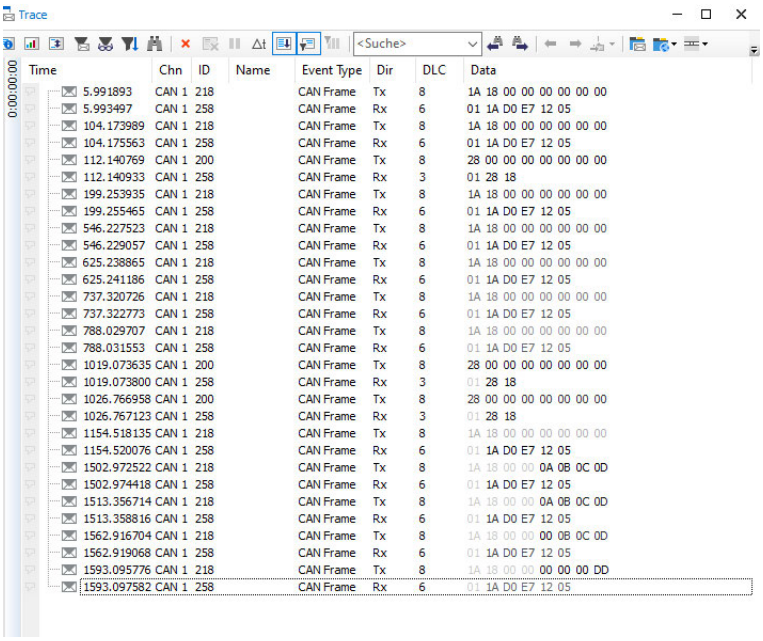

Abbildung 46 Auslesen von CANoe

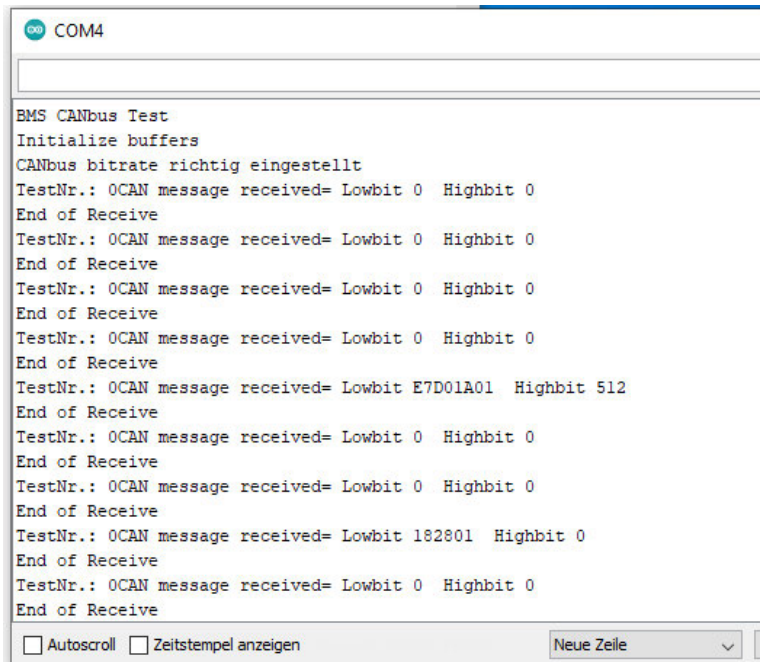

Abbildung 47 Auslesen von Arduino

Abbildung 48 zeigt einen von Frau Liu durchgeführten Test bezüglich TinyBMS, bei dem die Daten mit den von CANoe erfassten Daten übereinstimmen. Das heißt, das angestrebte Ziel wurde erfolgreich erreicht.

Abbildung 48 Spannung-Strom-Diagramm [8]

6 Zusammenfassung und Ausblick

Im Rahmen dieser Bachelorarbeit wurden alle Funktionen des VCU gemäß den Anforderungen erfolgreich umgesetzt, einschließlich der CAN-Kommunikation mit dem TinyBMS und der I2C-Kommunikation mit dem LCD. Das VCU ist nun in der Lage, die verbleibende Arbeitszeit zu berechnen und auf dem LCD genau anzuzeigen. Die spezifischen Ergebnisse dieser Bachelorarbeit werden in den folgenden Abschnitten diskutiert. Darüber hinaus wurden weitere Schritte für das Projekt in Betracht gezogen.

6.1 Zusammenfassung

Um die CAN-Kommunikation mit dem TinyBMS gemäß den Aufgabenanforderungen zu realisieren, wurden nach mehreren Anpassungsversuchen Hardwareprobleme behoben. Das TinyBMS funktioniert wie erwartet, weist jedoch möglicherweise noch einige Schwachstellen auf. Zunächst wurde im VCU-Programm keine spezielle Selbsttestfunktion für die Schaltung implementiert, da dies für die Aufgabenstellung nicht unbedingt erforderlich ist. Im Vergleich zu einem Auto gibt es weniger Sensoren, um Fehler zu erkennen, und es gibt auch nicht so viele ECU wie in einem Auto. Eine relativ einfache Schaltung ermöglicht eine schnelle Lokalisierung von Problemen. Die Fehlererkennungsfunktion des CAN-Busses ist nicht unbedingt erforderlich, da der CAN-Bus selbst über Fehlerkorrekturmechanismen verfügt und fehlerhafte CAN-Knoten automatisch blockiert und neu startet. Das CAN-Botschaft des TinyBMS wird alle 2 Sekunden abgerufen, daher können Einmalfehler schnell korrigiert werden, indem sie einfach in der Schleife zur Datenabfrage übersprungen werden. Bei Bedarf kann in zukünftigen Entwicklungen ein entsprechendes Fehlerprotokollprogramm für die Analyse von CAN-Bus-Fehlern implementiert werden.

Zum Thema Zuverlässigkeit wurde ein kontinuierlicher 5-stündiger Test im Labor durchgeführt, was etwa doppelt so lange ist wie die erwartete Betriebsdauer. Während dieser Zeit traten keine Verzögerungen oder Neustarts auf, und das TinyBMS konnte erfolgreich Daten lesen. Die LCD-Funktionalität funktionierte ebenfalls einwandfrei. Dies deutet darauf hin, dass das System eine hohe Zuverlässigkeit aufweist, es bleibt jedoch ungewiss, ob es im Betrieb weiterhin so zuverlässig bleibt. Es ist erforderlich, weitere realen Daten zu sammeln und zu analysieren, um die Gesamtzuverlässigkeit des Systems zu gewährleisten.

In Bezug auf die Programmoptimierung: Wenn im Programm zu viele if-Schleifen verwendet werden, um Schaltungssignale zu überprüfen, wird die Rechenleistung des Arduino beeinträchtigt. Daher wurden in der Richtungsbestimmung und bei der Auslesung der TinyBMS-Botschaften switch-Schleifen verwendet. Bei der Richtungsbestimmung wurde eine Methode verwendet, bei der die Werte zunächst gewichtet und dann addiert wurden. Im Projekt wurden die Werte mit 2, 3 und 4 multipliziert anstelle von 1, 2 und 3, da die Kombination 2, 3 und 4 zu einer stabileren Datenauswertung führte. Wenn ein bestimmtes Programm zu einem festgelegten Zeitpunkt ausgeführt werden soll, sollte ein Timer verwendet werden anstelle der delay-Funktion, da die delay-Funktion dazu führen kann, dass andere Programme verzögert werden. Nach Abschluss der Entwicklung können alle seriellen Ausgaben entfernt werden, um Rechenressourcen zu sparen.

Die Gestaltung des DBC sollte auch anhand zukünftiger Entwicklungen angepasst werden. Beispielsweise sollte die CAN-ID des Drehzahlsignals nicht größer sein als die des BMS, während einige unwichtige Informationen eine höhere CAN-ID haben können. Es gibt auch Optimierungsbereiche für die Nachrichtenpriorität, wie Verzögerung, Lastrate, CPU-Auslastung und sekundäre Aspekte wie Flexibilität, Wiederverwendbarkeit und Robustheit usw. Die Gestaltung der CAN-ID sollte auch die Priorität berücksichtigen, jedoch wurde dies in dieser Bachelorarbeit nicht im Detail diskutiert, da es nicht der Schwerpunkt war.

6.2 Ausblick

Die Genauigkeit der Restlaufzeit kann weiter verbessert werden. Die von TinyBMS ausgelesenen Daten sind Momentanwerte, was bedeutet, dass die Berechnung der Restlaufzeit nicht sehr präzise ist. Es wird empfohlen, die Genauigkeit der Zeitprognose zu verbessern, z. B. durch die Sammlung und Analyse von Echtzeitdaten, um eine genauere Schätzung der Restlaufzeit zu ermöglichen. Dies kann durch den Einsatz zusätzlicher Sensoren oder fortschrittlicher Leistungsberechnungsalgorithmen erfolgen. Es wird empfohlen, kontinuierlich zu überwachen und zu aktualisieren, um eine zuverlässige Anzeige der Restlaufzeit sicherzustellen.

Im Projekt kann auch die Restladezeit berechnet werden. Die Formel zur Berechnung der Restladezeit lautet wie folgt:

$$Restladezeit = \frac{SollkWh * (100 - SoC)}{Spannung * Strom}$$

Die Restladezeit kann in der S1-Phase berechnet werden. Zunächst werden über den CAN-Bus Daten von TinyBMS ausgelesen, die den Batteriestand und die geschätzte Restladezeit auf dem LCD anzeigen. Wenn der Akku während des Wake-up-Routine aufgeladen wird, wird die Restladezeit auf dem LCD angezeigt. Gleichzeitig wird ein

kleines Batteriesymbol blinken. Es ist zu beachten, dass die Restladezeit nicht besonders präzise ist, da der Ladestrom zu Beginn größer ist als am Ende des Ladevorgangs.

Aufgrund der fehlenden Internetverbindungskomponenten im Arduino ist Netzwerkangriffe praktisch unmöglich. Die mögliche Sicherheitslücke besteht in physischen Schnittstelleninjektionen. Um mögliche physische Schnittstelleninjektionen zu vermeiden, sollten nach Abschluss der Entwicklung die physischen Schnittstellen, insbesondere die Programmierschnittstelle des Arduino, deaktiviert werden. Darüber hinaus sollte auch die Sicherheit des CAN-Busses berücksichtigt werden. Wenn externe Botschaften die Antriebsmotoren oder den Rasenmäher steuern könnten, besteht die Möglichkeit von Schäden an Personen oder Gegenständen. Bei der zukünftigen Gestaltung der CAN-Botschaften kann eine angemessene Verschlüsselung in Betracht gezogen werden, z. B. durch Einfügen einer Prüfsumme in eine feste Byte-Position, um zu überprüfen, ob das CAN-Botschaften vom eigenen Arduino stammt. Diese Zahl könnte auf der Grundlage der Laufzeit des Arduino berechnet werden. Im Allgemeinen werden bei gleichzeitigem Starten der drei Arduinos theoretisch die Laufzeiten der drei Arduinos identisch sein. Unter Berücksichtigung der Übertragungszeit des CAN-Busses und der internen Berechnungszeit des Arduinos sollte ein Fehler in die berechnete Zahl eingeführt werden.

Wenn der Fehler innerhalb des zulässigen Bereichs liegt, kann das CAN-Botschaft als von einem Bord-Arduino stammend betrachtet werden. Eine alternative Methode besteht darin, das gesendete Signal mit einer einfachen Verschlüsselung zu ändern, z. B. die Verwendung der Caesar-Verschlüsselung, bei der bestimmte Zahlenpaare vertauscht werden, wie F und 1 oder A und 3. Nach dem Empfang muss das Signal erneut entschlüsselt und neu angeordnet werden, um den richtigen Wert zu erhalten. Im Vergleich zur Verwendung der Arduino-Laufzeit erhöhen diese Method den Rechenaufwand. Insgesamt zielt die Verschlüsselung darauf ab, potenzielle CAN-Bus-Injektionsbedrohungen zu verhindern.

Literatur

[1] Rauchfuss, L., Bacheloraufgabe (2023).

[2] Mankar, J., Darode, C., Trivedi, K., Kanoje, M., & Shahare, P. (2014). Review of I2C protocol. International Journal of Research in Advent Technology, 2(1).

[3] Leens, F. (2009). An introduction to I 2 C and SPI protocols. IEEE Instrumentation & Measurement Magazine, 12(1), 8-13.

[4] Zimmermann, Schmidgall, Ingenieur, Dr., Schmidgall, Ralf, & Dr. (2014). Bussysteme in der Fahrzeugtechnik Protokolle, Standards und Softwarearchitektur (5., aktualisierte und erw. Aufl. 2014 ed., ATZ/MTZ-Fachbuch). S. 59

[5] Di Natale, M., Zeng, H., Giusto, P., & Ghosal, A. (2012). Understanding and Using the Controller Area Network Communication Protocol Theory and Practice (1st ed. 2012. ed.). S.19

[6] Etschberger, K. (1994). CAN Controller-Area-Network ; Grundlagen, Protokolle, Bausteine, Anwendungen ; mit 15 Tabellen. S.75

[7] Energus Power Solutions Ltd., TinyBMS Communication Protocols (2018)

[8] Liu, X. (2022)

Internetquellenverzeichnis

[9] Rasentraktor, Modell Attila SKD

URL: https://www.etesia.de/produkte/gestrueppmaeher/attila-skd

verfügbar am: 30.05.2023, 10 Uhr

[10] Arduino Office Website

URL: https://www.arduino.cc/en/hardware

verfügbar am: 30.05.2023, 10 Uhr

[11] Display Visions,LCD Datasheet
URL: https://www.displayvisions.us/products/dotmatrix/tables.html#c2405
verfügbar am: 30.05.2023, 10 Uhr

[12] CAN Bus Wikipedia
URL: http://de.academic.ru/dic.nsf/dewiki/223005
verfügbar am: 30.05.2023, 10 Uhr

[13] USB-Isolator
URL: https://www.reichelt.de/de/
verfügbar am: 30.05.2023, 10 Uhr

[14] LCD 2004
URL: https://www.komputer.de/
verfügbar am: 30.05.2023, 10 Uhr

[15] I2C Backpack
URL: https://www.az-delivery.de/
verfügbar am: 30.05.2023, 10 Uhr

Anlagen

Anlagen 1, Arduino Hauptprogramm

```
//Final Hauptprogramm Update 30/05/2023

//Erforderliche Bibliotheken
#include <Wire.h> // Wire Bibliothek einbinden, I2C Kommunikation
#include "variant.h"//CANBus Bibliothek
#include <due_can.h>//CANBus Bibliothek fuer DUE arduino
#include <LiquidCrystal_I2C.h> //LCD_I2C Bibliothek einbinden
LiquidCrystal_I2C lcd(0x27, 20, 4); /*Hier wird festgelegt um was für einen Display es sich
  handelt.In diesem Fall eines mit 20 Zeichen in 4 Zeilen und der HEX-Adresse 0x27. */

//Pin mit Steuerplatine verbinden
int FahrsitzPin = 27;
int VorwaertsPin = 24;
int StillstandPin = 25;
int RueckwaertsPin = 26;
int MaehwerkPin = 28;
int Zuendschloss2 = 2;
int SelbsthaltungPin = 34;
int BereitPin = 23;

//Statusvariable definieren
int FahrState = 0;
int FahrsitzState = 0;
int RichtungStateV = 0;
int RichtungStateS = 0;
int RichtungStateR = 0;
int RichtungState = 0;
int MaehwerkState = 0;
bool geschaltet = false;
bool FahrStart = false;

//Grafische Darstellung der Batterie
unsigned char Batt_R_Voll[8] = {0x1e, 0x1e, 0x1e, 0x1f, 0x1f, 0x1e, 0x1e, 0x1e};
unsigned char Batt_R_Fast_Leer[8] = {0x1e, 0x1a, 0x1a, 0x1b, 0x1b, 0x1a, 0x1a, 0x1e};
unsigned char Batt_R_Leer[8] = {0x1e, 0x02, 0x02, 0x03, 0x03, 0x02, 0x02, 0x1e};
unsigned char Batt_L_Fast_Leer[8] = {0x1f, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x1f};
unsigned char Batt_L_Leer[8] = {0x1f, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x1f};


//Grafische Darstellung des Pfeils
unsigned char Pfeil_seit_1[8] = {0x10, 0x18, 0x1c, 0x1e, 0x0f, 0x07, 0x03, 0x01};
unsigned char Pfeil_seit_2[8] = {0x01, 0x03, 0x07, 0x0f, 0x1e, 0x1c, 0x18, 0x10};

//CAN Botschaften
unsigned char txVolt[1] = {0x14};
unsigned char txCurr[1] = {0x15};
unsigned char txSOC[1] = {0x1A};

//BMS Parameter
float SollkWh = 15.8;
int SOC_akt;
float Volt_akt;
float Curr_akt;
float Curr_akt_neg;// Positiver Strom beim Laden, negativer Strom beim Entladen
float P_akt;
float IstkWh ;
float Restzeit;

// Timer
static unsigned long lastTime = 0;

void setup() {
  //Fahrpin Initialisierung
  pinMode(FahrsitzPin, INPUT);
  pinMode(VorwaertsPin, INPUT);
  pinMode(StillstandPin, INPUT);
  pinMode(RueckwaertsPin, INPUT);
  pinMode(MaehwerkPin, INPUT);
  pinMode(Zuendschloss2, INPUT);
  pinMode(SelbsthaltungPin, OUTPUT);
  pinMode(BereitPin, OUTPUT);
```

```
72
73    //LCD Initialisierung
74    lcd.clear();
75    lcd.init(); //Im Setup wird der LCD gestartet
76    lcd.backlight(); //Hintergrundbeleuchtung einschalten
77    lcd.createChar(2, Pfeil_seit_2);
78    lcd.createChar(3, Pfeil_seit_1);
79    lcd.createChar(4, Batt_R_Fast_Leer);
80    lcd.createChar(5, Batt_R_Voll);
81    lcd.createChar(6, Batt_R_Leer);
82    lcd.createChar(7, Batt_L_Leer);
83    lcd.createChar(8, Batt_L_Fast_Leer);
84
85    Serial.begin(115200);//Initailisierung der seriellen Kommunikation 11520kbps
86
87    //CAN Initialisierung,  CAN0-Kannal ist Pin CANRX und CANTX
88    Can0.begin(CAN_BPS_500K);// CANbus Bitrate sychron,starten CANbus 500 kbps
89    if (Can0.begin(CAN_BPS_500K)) { //CANrate checken
90      Serial.println("CANbus Bitrate richtig eingestellt");
91      digitalWrite(BereitPin, HIGH);    // Wakeup pin23 auf 1 setzen
92    }
93    else {
94      Serial.println("CAN Initialisierung (sync) Fehler");
95      Serial.println("Bitte Arduino neustart");
96      while (1);//Bei falsche CANrate muss neu starten.
97    }
98  }
99
100 void loop() {
101   //Fahrinfo auslesen
102   FahrsitzState = digitalRead(FahrsitzPin);
103   RichtungStateV = digitalRead(VorwaertsPin);
104   RichtungStateS = digitalRead(StillstandPin);
105   RichtungStateR = digitalRead(RueckwaertsPin);
106   MaehwerkState = digitalRead(MaehwerkPin);
107   if (digitalRead(Zuendschloss2) == HIGH) { // Abfrage ob der S2 Taster gedrückt ist
108     if (FahrStart == true) {
109       geschaltet = true;// sollte er aus sein, setzen wir "geschaltet" auf true
110       digitalWrite(SelbsthaltungPin, HIGH);// TinyBMS Ignition
111       S2Start();//Fahr Routine
112     }
113     else {
114       Wakeup();// Fahrbedingungen nicht erreicht, Wakeup Routine
115     }
116   }
117   else {
118     if (geschaltet == true) {// "geschaltet" ist true, dann Selbsthaltung bleiben
119       digitalWrite(SelbsthaltungPin, HIGH);// TinyBMS Ignition
120       S2Start();
121     }
122     else {
123       Wakeup();
124     }
125   }
126 }
127
128 void S2Start() {
129   //Fahr Routine
130   lcd.setCursor(11, 2);
131   lcd.print("Fahr-");
132   lcd.setCursor(11, 3);
133   lcd.print("Rout. ");
134   Fahren();
135   if ((millis() - lastTime > 2000)) { //jeder 2s einmal messen
136     lastTime = millis();
137     BMS();//BMS auslesen
138     BMS_LCD();//Auf LCD zeigen
139   }
140 }
141
```

```
142 void Wakeup() {
143   //Wakeup Routine
144   //Fahrbedingungen auslesen:Fahrsitz, Maehwerk und Stillstand
145   FahrState = FahrsitzState * 4 + RichtungStateS * 3 + MaehwerkState * 2;
146   if (FahrState == 7) {// Fahrsitz 1, Stillstand, Maehwerk 0
147     //Ready, START S2
148     FahrStart = true;
149     lcd.setCursor(11, 2);
150     lcd.print("START");
151     lcd.setCursor(11, 3);
152     lcd.print(" S 2  ");
153     lcd.setCursor(0, 1);
154     lcd.print("Still      ");
155     lcd.setCursor(0, 2);
156     lcd.print("Maehwerk 0");
157     lcd.setCursor(0, 3);
158     lcd.print("Fahrsitz 1");
159   }
160   else {
161     //Wakeup Routine
162     Fahren();
163     FahrStart = false;
164     lcd.setCursor(11, 2);
165     lcd.print("nicht");
166     lcd.setCursor(11, 3);
167     lcd.print("Fahrb.");
168   }
169 }
170
171 void Fahren() {
172   //Die Richtung der Situation beurteilen
173   RichtungState = RichtungStateV * 4 + RichtungStateS * 3 + RichtungStateR * 2;
174   switch (RichtungState) {
175     case 4 ://Vorwaertsfahren
176       lcd.setCursor(0, 1);
177       lcd.print("Vorwaerts  ");
178       Pfeilkoeper();//Richtungspfeile
179       lcd.setCursor(19, 0);
180       lcd.write(char (3));
181       lcd.setCursor(17, 0);
182       lcd.write(char (2));
183       lcd.setCursor(19, 3);
184       lcd.write(0xfe);
185       lcd.setCursor(17, 3);
186       lcd.write(0xfe);
187       break;
188     case 3://Stillstand
189       lcd.setCursor(0, 1);
190       lcd.print("Still      ");
191       for (int i = 0; i < 4; i++) {//Richtungspfeile
192         lcd.setCursor(18, i);
193         lcd.write(0xfe);
194       }
195       lcd.setCursor(19, 0);
196       lcd.write(0xfe);
197       lcd.setCursor(17, 0);
198       lcd.write(0xfe);
199       lcd.setCursor(19, 3);
200       lcd.write(0xfe);
201       lcd.setCursor(17, 3);
202       lcd.write(0xfe);
203       break;
```

```
204     case 2 ://Rueckwaertsfahren
205       lcd.setCursor(0, 1);
206       lcd.print("Rueckwaerts");
207       Pfeilkoeper();//Richtungspfeile
208       lcd.setCursor(19, 3);
209       lcd.write(char (2));
210       lcd.setCursor(17, 3);
211       lcd.write(char (3));
212       lcd.setCursor(19, 0);
213       lcd.write(0xfe);
214       lcd.setCursor(17, 0);
215       lcd.write(0xfe);
216       break;
217     default://wenn keine Vor/Rueckwaerts/Stillsignal kommt, zeigen Fehler
218       lcd.setCursor(0, 1);
219       lcd.print("Fehler     ");
220       for (int i = 0; i < 4; i++) {//Richtungspfeile loeschen
221         lcd.setCursor(18, i);
222         lcd.write(0xfe);
223       }
224       lcd.setCursor(19, 0);
225       lcd.write(0xfe);
226       lcd.setCursor(17, 0);
227       lcd.write(0xfe);
228       lcd.setCursor(19, 3);
229       lcd.write(0xfe);
230       lcd.setCursor(17, 3);
231       lcd.write(0xfe);//Löschen die Richtungspfeile auf der LCD
232       break;
233   }
234   lcd.setCursor(0, 2);
235   if (MaehwerkState == LOW) {
236     lcd.print("Maehwerk 0");
237   }
238   else {
239     lcd.print("Maehwerk 1");
240   }
241
242   lcd.setCursor(0, 3);
243   if (FahrsitzState == LOW) {  //Wenn niemand besitzt, darf Rasentraktor nicht gestartet werden.
244     lcd.print("Fahrsitz 0");
245   }
246   else {
247     lcd.print("Fahrsitz 1");
248   }
249 }
250
251 void Pfeilkoeper() {
252   //Richtungspfeile
253   for (int i = 0; i < 4; i++) {
254     lcd.setCursor(18, i);
255     lcd.write(0xff);
256   }
257 }
258
259 void BMS() {
260   //CAN Botschaften senden
261   CAN_Send(0x218, 1, txSOC);
262   CAN_BMS_Recev();
263   Serial.println("SOC");
264   Serial.println(SOC_akt);
265   CAN_Send(0x218, 1, txVolt);
266   CAN_BMS_Recev();
267   Serial.println("Spannung");
268   Serial.println(Volt_akt);
269   CAN_Send(0x218, 1, txCurr);
270   CAN_BMS_Recev();
271   Serial.println("Strom");
272   Serial.println(Curr_akt);
```

```
  //Restlaufzeit rechnen
  P_akt = Volt_akt * Curr_akt;
  IstkWh = SollkWh * SOC_akt / 100;
  Restzeit = IstkWh / P_akt;
  Serial.print("Leistung : ");
  Serial.println(P_akt);
  Serial.print("IstkWh : ");
  Serial.println(IstkWh);
  Serial.print("Restlaufzeit : ");
  Serial.println(Restzeit);
  Serial.println("End of Receive");
}

void CAN_Send(int ID, int laenge, unsigned char *Botschaft) {
  CAN_FRAME outgoing;
  outgoing.extended = 0;
  outgoing.id = ID;
  outgoing.length = laenge;
  for (int i = 0; i < laenge; i++) {
    outgoing.data.byte[i] = Botschaft[i];
  }
  Can0.sendFrame(outgoing);
  Serial.println("CANbus senden erfoglrich");
}

void CAN_BMS_Recev() {
  Can0.watchFor(0x258);//fuer BMS nur CANID mit 0x258 pass
  uint8_t Byte1 ; // fuer richtige CAN Antwort muss erste Byte = 1 sein
  uint8_t Byte2;
  uint8_t Byte3;
  uint8_t Byte4;
  uint8_t Byte5;
  uint8_t Byte6;//Alle CAN-Signale von der BMS haben maximal 6 Byte.
  uint32_t  Bytecan = 0;

  //CAN Botschaften empfangen
  CAN_FRAME incoming;
  Can0.read(incoming);

  //Speichern die empfangenen CAN Daten in Bytes
  Byte1 = incoming.data.byte[0];
  Byte2 = incoming.data.byte[1];
  Byte3 = incoming.data.byte[2];
  Byte4 = incoming.data.byte[3];
  Byte5 = incoming.data.byte[4];
  Byte6 = incoming.data.byte[5];

  //Ausgabe des empfangenen Signals
  Serial.print("CAN message ID = ");
  Serial.print(incoming.id, HEX);
  Serial.print("  byte1: ");
  Serial.print(incoming.data.byte[0], HEX);
  Serial.print("  byte2: ");
  Serial.print(incoming.data.byte[1], HEX);
  Serial.print("  byte3: ");
  Serial.print(incoming.data.byte[2], HEX);
  Serial.print("  byte4: ");
  Serial.println(incoming.data.byte[3], HEX);
  Serial.print("  byte5: ");
  Serial.print(incoming.data.byte[4], HEX);
  Serial.print("  byte6: ");
  Serial.print(incoming.data.byte[5], HEX);
```

```
336
337   if (0x01 == Byte1) { //falls 1.Byte nicht 0x01, CANSignal ist falsch
338     Serial.println(" CAN Auslesen erfolgreich!");
339   }
340   else {
341    Serial.println(" CAN Auslesen Fehler! Bitte erneut machen");
342     switch (Byte2) ///Das zweite Byte kann als Identifizierung verwendet werden
343     {
344       case 0x1A://SOC
345         Serial.print("SOC Auslesen Fehler");
346         break;
347       case 0x14://akt Spannung
348         Serial.print("Spannung Auslesen Fehler");
349         break;
350       case 0x15://akt Strom
351         Serial.print("Strom Auslesen Fehler");
352         break;
353       default:
354         Serial.println("unbekannt CAN-Botschaft,bitte erneut starten!");
355         break;
356     }
357     return;/*Falsche CAN-Signale werden nicht akzeptiert und werden in der nächsten
358     Runde neu berechnet.*/
359   }
360
361   switch (Byte2) ///Das zweite Byte kann als Identifizierung verwendet werden
362   {
363     case 0x1A://SOC
364       Bytecan = Byte32 (Byte6, Byte5, Byte4, Byte3);
365       SOC_akt = Bytecan / 1000000;//0-100 SOC
366       Serial.print("SOC");
367       Serial.println(SOC_akt);
368       break;
369     case 0x14://akt Spannung
370       Bytecan = Byte32 (Byte6, Byte5, Byte4, Byte3);
371       Volt_akt = *(float*)&Bytecan;
372       Serial.print("Spannung");
373       Serial.println(Volt_akt);
374       break;
375     case 0x15://akt Strom
376       Bytecan = Byte32 (Byte6, Byte5, Byte4, Byte3);
377       Curr_akt_neg = *(float*)&Bytecan;//bei fahren wurde immer negative stromwert
378       Curr_akt = abs(Curr_akt_neg);
379       Serial.print("Strom");
380       Serial.println(Curr_akt);
381       break;
382     default:
383       Serial.println("unbekannt CAN-Botschaft,bitte erneut starten!");
384       break;
385   }
386 }
387
388 void BMS_LCD() {
389   //Restlaufzeit auf LCD-Display zeigen
390   int RestzeitStu = Restzeit;
391   lcd.setCursor(10, 0);
392   lcd.print(RestzeitStu);
393   int RestzeitMin = (Restzeit - RestzeitStu) * 60;
394   if ( RestzeitMin >= 10) {
395     lcd.setCursor(12, 0);
396     lcd.print(RestzeitMin);
397   }
398   else {
399     lcd.setCursor(13, 0);
400     lcd.print(RestzeitMin);
401   }
402   lcd.setCursor(11, 0);
403   lcd.print("h");
404   lcd.setCursor(14, 0);
405   lcd.print("min");
```

```
406
407   //Grapische Darstellung der Batterie
408   if (SOC_akt > 99) {
409     delay(100);
410     lcd.setCursor(3, 0);
411     lcd.print(SOC_akt);
412     lcd.setCursor(6, 0);
413     lcd.print("%");
414   }
415   else {
416     if (SOC_akt > 9) {
417       delay(100);
418       lcd.setCursor(3, 0);
419       lcd.write(0xfe);
420       lcd.setCursor(4, 0);
421       lcd.print(SOC_akt);
422       lcd.setCursor(6, 0);
423       lcd.print("%");
424     }
425     else {
426       delay(100);
427       lcd.setCursor(3, 0);
428       lcd.write(0xfe);
429       lcd.setCursor(4, 0);
430       lcd.write(0xfe);
431       lcd.setCursor(5, 0);
432       lcd.print(SOC_akt);
433       lcd.setCursor(6, 0);
434       lcd.print("%");
435       lcd.noBacklight();
436       delay(100);
437       lcd.backlight();
438     }
439   }
440
441   //Grapische Darstellung der Batterie
442   if (SOC_akt > 99) {
443     lcd.setCursor(0, 0);
444     lcd.write(0xFF);
445     lcd.setCursor(1, 0);
446     lcd.write(char (5));
447   }
448   else {
449     if (SOC_akt > 79) {
450       lcd.setCursor(0, 0);
451       lcd.write(0xFF);
452       lcd.setCursor(1, 0);
453       lcd.write(char (4));
454     }
455     else {
456       if (SOC_akt > 59) {
457         lcd.setCursor(0, 0);
458         lcd.write(0xFF);
459         lcd.setCursor(1, 0);
460         lcd.write(char (6));
461       }
462       else {
463         if (SOC_akt > 29) {
464           lcd.setCursor(0, 0);
465           lcd.write(char(8));
466           lcd.setCursor(1, 0);
467           lcd.write(char (6));
468         }
469         else {
470           lcd.setCursor(0, 0);
471           lcd.write(char(7));
472           lcd.setCursor(1, 0);
473           lcd.write(char (6));
474         }
475       }
476     }
477   }
478 }
```

```
479
480 uint32_t Byte32 (uint8_t Bytea, uint8_t Byteb, uint8_t Bytec, uint8_t Byted) {
481   //Bildung von 4 x 8bit Daten zu 32bit Daten
482   uint32_t Bytecan;
483   Bytecan = Bytea; //1 Byte speichern
484   Bytecan <<= 8; //Verschiebung 8 Bits nach links, beginnend bei 0x100 statt 0x01
485   Bytecan = Bytecan | Byteb;//OR-Operation, d.h. 0x100 wird mit 0x71 verbunden, also 0x171
486   Bytecan <<= 8;
487   Bytecan = Bytecan | Bytec;
488   Bytecan <<= 8;
489   Bytecan = Bytecan | Byted;
490   return Bytecan;//Erhalten die zusammengeführten 32Bit Daten
491 }
```

Anlagen 2, Arduino Testprogramm

CAN_Empf_Test §

```
////CAN Empfangen Test Programm, ob der Arduino Nachrichten über den CAN Bus empfangen kann
//Erforderliche Bibliotheken
#include "variant.h"
#include <due_can.h>

void setup()
{
  Serial.begin(115200);//Initailisierung der seriellen Kommunikation mit 11520kbps
  Serial.println("CAN Empfangen Test");

  // CAN0-Kannal ist Pin CANRX und CANTX
  Can0.begin(CAN_BPS_500K);//CANbus Bitrate sychron,starten CANbus mit 500 kbps
  while (!Serial);//NUR bei Serial Monitor Ready straten, bei Hauptprogramm weg!!!
  if (Can0.begin(CAN_BPS_500K))
  {
    Serial.println("CANbus Bitrate richtig eingestellt");
  }
  else
  {
    Serial.println("CAN Initialisierung (sync) Fehler");
    Serial.println("Bitte Arduino neustart");
    while (1);//Bei falsche CANrate muss neu starten.
  }
  Can0.watchFor();//Alle CAN-Signal Pass
}
void loop() {
  CAN_FRAME incoming;
  Can0.read(incoming);$
  //Can0.sendFrame(incoming);
  Serial.print("Empfangene CAN-Nachricht = ");
  Serial.print(incoming.data.low, HEX);
  Serial.println(incoming.data.high, HEX);
  Serial.println("\nEnde des Empfangs");
}
```

CAN_Send_Test

```
//CAN Senden Test Programm, ob der Arduino Nachrichten über den CAN Bus senden kann
// Erforderliche Bibliotheken
#include "variant.h"
#include <due_can.h>

void setup()
{
  Serial.begin(115200);//Initailisierung der seriellen Kommunikation mit 11520kbps
  Serial.println("CAN Senden Test");

  // CAN0-Kannal ist Pin CANRX und CANTX
  Can0.begin(CAN_BPS_500K);//CANbus Bitrate sychron,starten CANbus mit 500 kbps
  //while (!Serial);//NUR bei Serial Monitor Ready straten, bei Hauptprogramm weg!!!
  if (Can0.begin(CAN_BPS_500K))
  {
    Serial.println("CANbus Bitrate richtig eingestellt");
  }
  else
  {
    Serial.println("CAN Initialisierung (sync) Fehler");
    Serial.println("Bitte Arduino neustart");
    while (1);//Bei falsche CANrate muss neu starten.
  }
  Can0.watchFor();//Alle CAN-Signal Pass
}

void loop()
{
  CAN_FRAME outgoing;
  outgoing.id = 0x200;
  outgoing.data.low = 0x11112222;
  outgoing.data.high = 0x33334444;
  Can0.sendFrame(outgoing);
  Serial.println("CANbus senden erfoglrich");
}
```

BMS_auswerten

```
//BMS auslesen und auswerten
#include "variant.h"
#include <due_can.h>

//BMS CAN-Botschaften
unsigned char txRead[1];
//unsigned char txWrite[2];
unsigned char txVolt[1];
unsigned char txCurr[1];
unsigned char txSOC[1];

float IstkWh = 15.8;
int SOC_akt;
int CANID;
float Volt_akt;
float Curr_akt;
float P_akt;
int startLeistung = 9; /*Derzeit kann das bms keine echte Spannung und Stromstärke
  messen, da das bms nicht im Rasentraktor installiert ist. Der angegebene Wert ist
  nur eine Schätzung sowie ein Testwert. 2x3 + 5 / 2~=9*/

void setup()
{
  Serial.begin(115200);//Initailisierung der seriellen Kommunikation mit 11520kbps
  Serial.println("BMS Auswerten");

  // CAN0-Kannal ist Pin CANRX und CANTX
  Can0.begin(CAN_BPS_500K);// CANbus Bitrate sychron,starten CANbus mit 500 kbps
  while (!Serial);//NUR bei Serial Monitor Ready straten, bei Hauptprogramm  weg!!!
  if (Can0.begin(CAN_BPS_500K))//CANrate checken
  {
    Serial.println("CANbus Bitrate richtig eingestellt");
  }
  else
  {
    Serial.println("CAN Initialisierung (sync) Fehler");
    Serial.println("Bitte Arduino neustart");
    while (1);//Bei falsche CANrate muss neu starten.
  }
  Can0.watchFor(0x258);//nur CANID mit 0x258 pass

  //Botschaften Inhalten
  txRead[0] = 0x28;
  /*txWrite[0] = 0x29;
    txWrite[1] = 0x18;*/
  txVolt[0] = 0x14;
  txCurr[0] = 0x15;
  txSOC[0] = 0x1A;
}

void loop()
{
  //ID auslesen
  Serial.println("ReadID");
  CAN_Send(0x218, 1, txRead);
  CAN_BMS_Recev();

  //SOC auslesen
  Serial.println("SOC");
  CAN_Send(0x218, 1, txSOC);
  CAN_BMS_Recev();

  //Spannung auslesen
  Serial.println("Spannung");
  CAN_Send(0x218, 1, txVolt);
  CAN_BMS_Recev();

  //Strom auslesen
  Serial.println("Strom");
  CAN_Send(0x218, 1, txCurr);
  CAN_BMS_Recev();

  //Restlaufzeit rechnen
  float SollkWh = IstkWh * SOC_akt / 100;
  //P_akt = Volt_akt * Curr_akt; nun es gibt keine echte Strom, also P= 0
  P_akt = startLeistung;
  float Restzeit = SollkWh / P_akt;

```

```
  //Ausgabe BMS Auswertung
  Serial.print("Leistung : ");
  Serial.println(P_akt);
  Serial.print("SollkWh : ");
  Serial.println(SollkWh);
  Serial.print("Restlaufzeit : ");
  Serial.println(Restzeit);
  Serial.println("Ende des Empfangs");
}

void CAN_Send(int ID, int laenge, unsigned char *Botschaft)
{
  //CAN Sendprogramm
  CAN_FRAME outgoing;
  outgoing.id = ID;
  outgoing.length = laenge;
  for (int i = 0; i < laenge; i++)
  {
    outgoing.data.byte[i] = Botschaft[i];
  }
  Can0.sendFrame(outgoing);
  Serial.println("CANbus senden erfoglrich");
}

void CAN_BMS_Recev()//NUR fuer BMS
{
  //CAN Empfangsprogramm
  uint8_t Byte1 ; // fuer richtige CAN Antwort muss erste Byte = 1 sein
  uint8_t Byte2;
  uint8_t Byte3;
  uint8_t Byte4;
  uint8_t Byte5;
  uint8_t Byte6;//Alle CAN-Signale von der BMS haben maximal 6 Byte.
  uint32_t  Bytecan = 0;

  //CAN Botschaften empfangen
  CAN_FRAME incoming;
  Can0.read(incoming);

  //Speichern die empfangenen CAN Daten in Bytes
  Byte1 = incoming.data.byte[0];
  Byte2 = incoming.data.byte[1];
  Byte3 = incoming.data.byte[2];
  Byte4 = incoming.data.byte[3];
  Byte5 = incoming.data.byte[4];
  Byte6 = incoming.data.byte[5];

  //Ausgabe des empfangenen Signals
  Serial.print("CAN message ID = ");
  Serial.print(incoming.id, HEX);
  Serial.print("  byte1: ");
  Serial.print(incoming.data.byte[0], HEX);
  Serial.print("  byte2: ");
  Serial.print(incoming.data.byte[1], HEX);
  Serial.print("  byte3: ");
  Serial.print(incoming.data.byte[2], HEX);
  Serial.print("  byte4: ");
  Serial.println(incoming.data.byte[3], HEX);
  Serial.print("  byte5: ");
  Serial.print(incoming.data.byte[4], HEX);
  Serial.print("  byte6: ");
  Serial.print(incoming.data.byte[5], HEX);
  //Serial.print("  byte7: ");
  //Serial.print(incoming.data.byte[6], HEX);
  //Serial.print("  byte8: ");
  //Serial.println(incoming.data.byte[7], HEX);

  if (0x01 == Byte1) //falls 1.Byte nicht 0x01, CANSignal ist falsch
  {
    Serial.println(" CAN Auslesen erfolgreich!");
  }
  else
```

```
  else
  {
    Serial.println(" CAN Auslesen Fehler! Bitte erneut machen");
    return;/*Falsche CAN-Signale werden nicht akzeptiert und werden in der nächsten
    Runde neu berechnet.*/
  }
  switch (Byte2)//Das zweite Byte kann als Identifizierung verwendet werden
  {
    case 0x1A://SOC
      Bytecan = Byte32 (Byte3, Byte4, Byte5, Byte6);
      SOC_akt = Bytecan / 1000000;//0-100 SOC
      Serial.print("SOC");
      Serial.println(SOC_akt);
      break;
    case 0x14://akt Spannung
      Bytecan = Byte32 (Byte3, Byte4, Byte5, Byte6);
      Volt_akt = *(float*)&Bytecan;
      Serial.print("Spannung");
      Serial.println(Volt_akt);
      break;
    case 0x15://akt Strom
      Bytecan = Byte32 (Byte3, Byte4, Byte5, Byte6);
      Curr_akt = *(float*)&Bytecan;
      Serial.print("Strom");
      Serial.println(Curr_akt);
      break;
    case 0x28://Read ID
      Bytecan = Byte32 (Byte3, Byte4, Byte5, Byte6);
      CANID = Bytecan;
      Serial.print("ID");
      Serial.println(CANID);
      break;
    default:
      Serial.println("unbekannt CAN-Botschaft,bitte erneut starten!");
      break;
  }
}

uint32_t Byte32 (uint8_t Bytea, uint8_t Byteb, uint8_t Bytec, uint8_t Byted)
{
  //Bildung von 4 x 8bit Daten zu 32bit Daten
  uint32_t Bytecan;
  Bytecan = Bytea; //1 Byte speichern
  Bytecan <<= 8; //Verschiebung 8 Bits nach links, beginnend bei 0x100 statt 0x01
  Bytecan = Bytecan | Byteb;//OR-Operation, d.h. 0x100 wird mit 0x71 verbunden, also 0x171
  Bytecan <<= 8;
  Bytecan = Bytecan | Bytec;
  Bytecan <<= 8;
  Bytecan = Bytecan | Byted;
  return Bytecan;//Erhalten die zusammengeführten 32Bit Daten
}
```

I2C_Scanner

```
// I2C Scanner
// Written by Nick Gammon
// Date: 20th April 2011
#include <Wire.h>
void setup() {
Serial.begin (115200);
// Leonardo: wait for serial port to connect
while (!Serial)
{
}
Serial.println ();
Serial.println ("I2C scanner. Scanning ...");
byte count = 0;
Wire.begin();
for (byte i = 8; i < 120; i++)
{
Wire.beginTransmission (i);
if (Wire.endTransmission () == 0)
{
Serial.print ("Found address: ");
Serial.print (i, DEC);
Serial.print (" (0x");
Serial.print (i, HEX);
Serial.println (")");
count++;
delay (1); // maybe unneeded?
} // end of good response
} // end of for loop
Serial.println ("Done.");
Serial.print ("Found ");
Serial.print (count, DEC);
Serial.println (" device(s).");
} // end of setup
void loop() {}
```

Display_Test

```
// LCD Display Test Programm
#include <Wire.h> // Wire Bibliothek einbinden
#include <LiquidCrystal_I2C.h> //LCD_I2C Bibliothek einbinden
LiquidCrystal_I2C lcd(0x27, 20, 4); /*Hier wird festgelegt um was für einen Display es sich
  handelt.In diesem Fall eines mit 20 Zeichen in 4 Zeilen und der HEX-Adresse 0x27. */

//Grafische Darstellung der Batterie
unsigned char Batt_R_Voll[8] = {0x1e, 0x1e, 0x1e, 0x1f, 0x1f, 0x1e, 0x1e, 0x1e};
unsigned char Batt_R_Fast_Leer[8] = {0x1e, 0x1a, 0x1a, 0x1b, 0x1b, 0x1a, 0x1a, 0x1e};
unsigned char Batt_R_Leer[8] = {0x1e, 0x02, 0x02, 0x03, 0x03, 0x02, 0x02, 0x1e};
unsigned char Batt_L_Fast_Leer[8] = {0x1f, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x1f};
unsigned char Batt_L_Leer[8] = {0x1f, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x1f};

//Grafische Darstellung des Pfeils
unsigned char Pfeil_seit_1[8] = {0x10, 0x18, 0x1c, 0x1e, 0x0f, 0x07, 0x03, 0x01};
unsigned char Pfeil_seit_2[8] = {0x01, 0x03, 0x07, 0x0f, 0x1e, 0x1c, 0x18, 0x10};

void setup()
{
  lcd.clear();
  lcd.init(); //Im Setup wird der LCD gestartet
  lcd.backlight(); //Hintergrundbeleuchtung einschalten

  //Speichern die gewünschten Grafische Darstellung im Programm.
  lcd.createChar(2, Pfeil_seit_2);
  lcd.createChar(3, Pfeil_seit_1);
  lcd.createChar(4, Batt_R_Fast_Leer);
  lcd.createChar(5, Batt_R_Voll);
  lcd.createChar(6, Batt_R_Leer);
  lcd.createChar(7, Batt_L_Leer);
  lcd.createChar(8, Batt_L_Fast_Leer);
}

void loop()
{
  lcd.setCursor(3, 0);
  lcd.print("LCD I2C Test");
  lcd.setCursor(3, 2);
  lcd.print("HS Mittweida");
  lcd.setCursor(3, 3);
  lcd.print("Projekt");

  //Pfeil koeper auf LCD zeigen
  lcd.setCursor(18, 0);
  lcd.write(0xff);
  lcd.setCursor(18, 1);
  lcd.write(0xff);
  lcd.setCursor(18, 2);
  lcd.write(0xff);
  lcd.setCursor(18, 3);
  lcd.write(0xff);

  //Rueckwaertsfahren Pfeil
  lcd.setCursor(17, 3);
  lcd.write(char (3));
  lcd.setCursor(19, 3);
  lcd.write(char (2));

  //Vorwaertsfahren Pfeil
  lcd.setCursor(19, 0);
  lcd.write(char (3));
  lcd.setCursor(17, 0);
  lcd.write(char (2));

  //voll Akku
  lcd.setCursor(0, 0);
  lcd.write(0xFF);
  lcd.setCursor(1, 0);
  lcd.write(char (5));

  //halb Akku
  lcd.setCursor(0, 1);
  lcd.write(0xFF);
  lcd.setCursor(1, 1);
  lcd.write(char (6));

  //fast leer Akku
  lcd.setCursor(0, 2);
  lcd.write(char(8));
  lcd.setCursor(1, 2);
  lcd.write(char (6));

  //ganz leer Akku
  lcd.setCursor(0, 3);
  lcd.write(char(7));
  lcd.setCursor(1, 3);
  lcd.write(char (6));
}
```

Schaltung_Test

```
//Demo Test Programme fuer Steuerplatine und Schaltungen 08.08.2023
#include <Wire.h> // Wire Bibliothek einbinden
#include <LiquidCrystal_I2C.h> // LCD_I2C Bibliothek einbinden
LiquidCrystal_I2C lcd(0x27, 20, 4); /*Hier wird festgelegt um was für einen Display es sich
  handelt.In diesem Fall eines mit 20 Zeichen in 4 Zeilen und der HEX-Adresse 0x27. */

//Pin mit Steuerplatine verbinden
int FahrsitzPin = 27;
int VorwaertsPin = 24;
int StillstandPin = 25;
int RueckwaertsPin = 26;
int MaehwerkPin = 28;
int SelbsthaltungPin = 23;

//Statusvariable definieren
int FahrsitzState = 0;
int RichtungStateV = 0;
int RichtungStateS = 0;
int RichtungStateR = 0;
int RichtungState = 0;
int MaehwerkState = 0;

//Grafische Darstellung der Batterie
unsigned char Batt_R_Voll[8] = {0x1e, 0x1e, 0x1e, 0x1f, 0x1f, 0x1e, 0x1e, 0x1e};
unsigned char Batt_R_Fast_Leer[8] = {0x1e, 0x1a, 0x1a, 0x1b, 0x1b, 0x1a, 0x1a, 0x1e};
unsigned char Batt_R_Leer[8] = {0x1e, 0x02, 0x02, 0x03, 0x03, 0x02, 0x02, 0x1e};
unsigned char Batt_L_Fast_Leer[8] = {0x1f, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x1f};
unsigned char Batt_L_Leer[8] = {0x1f, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x1f};

//Grafische Darstellung des Pfeils
unsigned char Pfeil_seit_1[8] = {0x10, 0x18, 0x1c, 0x1e, 0x0f, 0x07, 0x03, 0x01};
unsigned char Pfeil_seit_2[8] = {0x01, 0x03, 0x07, 0x0f, 0x1e, 0x1c, 0x18, 0x10};

void setup() {
  //Fahrpin Initialisierung
  pinMode(FahrsitzPin, INPUT);
  pinMode(VorwaertsPin, INPUT);
  pinMode(StillstandPin, INPUT);
  pinMode(RueckwaertsPin, INPUT);
  pinMode(MaehwerkPin, INPUT);
  pinMode(SelbsthaltungPin, OUTPUT);

  //LCD Initialisierung
  lcd.clear();
  lcd.init(); //Im Setup wird der LCD gestartet
  lcd.backlight(); //Hintergrundbeleuchtung einschalten
  lcd.setCursor(0, 0);
  lcd.print("Loading");

  //Speichern die gewünschten Grafische Darstellung im Programm.
  lcd.createChar(2, Pfeil_seit_2);
  lcd.createChar(3, Pfeil_seit_1);
  lcd.createChar(4, Batt_R_Fast_Leer);
  lcd.createChar(5, Batt_R_Voll);
  lcd.createChar(6, Batt_R_Leer);
  lcd.createChar(7, Batt_L_Leer);
  lcd.createChar(8, Batt_L_Fast_Leer);
}

void loop() {
  //Steuerplatine/Schaltungen Status auslesen
  FahrsitzState = digitalRead(FahrsitzPin);
  RichtungStateV = digitalRead(VorwaertsPin);
  RichtungStateS = digitalRead(StillstandPin);
  RichtungStateR = digitalRead(RueckwaertsPin);
  MaehwerkState = digitalRead(MaehwerkPin);
  lcd.setCursor(0, 3);
  if (FahrsitzState == LOW)
  {
    //Wenn niemand besitzt, darf Rasentraktor nicht gestartet werden.
    lcd.print("Fahrsitz 0   ");
  }
  else {
    lcd.print("Fahrsitz 1   ");
    Fahren();// Der Rasentraktor darf nun gestartet werden.
  }
}
```

```
79 void Fahren() {
80   //Fahrtrichtung beurteilen
81   RichtungState = RichtungStateV * 4 + RichtungStateS * 3 + RichtungStateR * 2;
82
83   switch (RichtungState)
84   {
85     case 4 ://Vorwaerts fahren und Richtung auf LCD zeigen
86       lcd.setCursor(0, 1);
87       lcd.print("Vorwaerts  ");
88       Pfeilkoeper();
89       lcd.setCursor(19, 0);
90       lcd.write(char (3));
91       lcd.setCursor(17, 0);
92       lcd.write(char (2));
93       lcd.setCursor(19, 3);
94       lcd.write(0xfe);
95       lcd.setCursor(17, 3);
96       lcd.write(0xfe);
97       break;
98     case 3:
99       lcd.setCursor(0, 1);
00       lcd.print("Still        ");
01       for (int i = 0; i < 4; i++)
02       {
03         lcd.setCursor(18, i);
04         lcd.write(0xfe);
05       }
06       lcd.setCursor(19, 0);
07       lcd.write(0xfe);
08       lcd.setCursor(17, 0);
09       lcd.write(0xfe);
10       lcd.setCursor(19, 3);
11       lcd.write(0xfe);
12       lcd.setCursor(17, 3);
13       lcd.write(0xfe);
14       break;
15     case 2 :
16       lcd.setCursor(0, 1);
17       lcd.print("Rueckwaerts  ");
18       Pfeilkoeper();
19       lcd.setCursor(19, 3);
20       lcd.write(char (2));
21       lcd.setCursor(17, 3);
22       lcd.write(char (3));
23       lcd.setCursor(19, 0);
24       lcd.write(0xfe);
25       lcd.setCursor(17, 0);
26       lcd.write(0xfe);
27       break;
28     default:
29       lcd.setCursor(0, 1);
30       lcd.print("Fehler     ");
31       for (int i = 0; i < 4; i++)
32       {
33         lcd.setCursor(18, i);
34         lcd.write(0xfe);
35       }
36       lcd.setCursor(19, 0);
37       lcd.write(0xfe);
38       lcd.setCursor(17, 0);
39       lcd.write(0xfe);
40       lcd.setCursor(19, 3);
41       lcd.write(0xfe);
42       lcd.setCursor(17, 3);
43       lcd.write(0xfe);
44       break;
45   }
```

```
146   //Ob der Maehwerk eingeschaltet ist
147   lcd.setCursor(0, 2);
148   if (MaehwerkState == LOW)
149   {
150     lcd.print("Maehwerk aus");
151   }
152   else
153   {
154     lcd.print("Maehwerk ein");
155     Batterie();
156   }
157 }
158
159 void Batterie() {
160   //Achtung! NUR Beispiele Test Programme fuer Grafische Darstellung der Batterie!
161   lcd.setCursor(7, 0);
162   lcd.print("1h 59min");//Beispiele Restlaufzeit
163
164   //Beispiele Batterie Kapatität/SOC von 100 bis 0
165   for (int battCap = 100; battCap > 0; battCap--)
166   {
167     if (battCap > 99)
168     {
169       delay(150); lcd.setCursor(2, 0); lcd.print(battCap);
170       lcd.setCursor(5, 0); lcd.print("% ");
171     }
172     else
173     {
174       delay(150); lcd.setCursor(2, 0); lcd.write(0xfe);
175       if (battCap > 9)
176       {
177         delay(150); lcd.setCursor(3, 0); lcd.print(battCap);
178         lcd.setCursor(5, 0); lcd.print("%");
179       }
180       else
181       {
182         delay(150); lcd.setCursor(3, 0); lcd.write(0xfe);
183         lcd.setCursor(4, 0); lcd.print(battCap); lcd.setCursor(5, 0); lcd.print("%");
184         lcd.setCursor(0, 3); lcd.print("!!Akku leer!!");
185         //Wenn der Akku leer ist, soll das LCD blinken.
186         lcd.noBacklight(); delay(150); lcd.backlight(); delay(150);
187       }
188     }
189
190     switch (battCap)// Anzeige des Batteriebildes in Prozent
191     {
192       case 100:
193         lcd.setCursor(0, 0); lcd.write(0xFF); lcd.setCursor(1, 0); lcd.write(char (5));
194         break;
195       case 80:
196         lcd.setCursor(0, 0); lcd.write(0xFF); lcd.setCursor(1, 0); lcd.write(char (4));
197         break;
198       case 60:
199         lcd.setCursor(0, 0); lcd.write(0xFF); lcd.setCursor(1, 0); lcd.write(char (6));
200         break;
201       case 30:
202         lcd.setCursor(0, 0); lcd.write(char(8)); lcd.setCursor(1, 0); lcd.write(char (6));
203         break;
204       case 10:
205         lcd.setCursor(0, 0); lcd.write(char(7)); lcd.setCursor(1, 0); lcd.write(char (6));
206         break;
207     }
208   }
209 }
210
211 void Pfeilkoeper() {
212   for (int i = 0; i < 4; i++)
213   {
214     lcd.setCursor(18, i);
215     lcd.write(0xff);
216   }
217 }
```

Selbsthaltung_0608

```
//Seblsthaltung
bool geschaltet = false;      // es dient zur Speicherung des Schaltzustandes
int ledPin = 13;
int tasterPin = 2;

void setup()
{
  pinMode(ledPin, OUTPUT);
  pinMode(tasterPin, INPUT);
}

void loop() {
  if (digitalRead(tasterPin) == HIGH)  // Abfrage ob der Taster gedrückt ist
  {
    if (geschaltet == false) {
      geschaltet = true;              /*Abfrage ob der logischer Schalter aus ist,falls
      es aus ist, setzen geschaltet auf true*/
      digitalWrite(ledPin, HIGH);// schalten die LED ein
    }
    else
    {
      if (geschaltet == true) {
        digitalWrite(ledPin, HIGH);// schalten die LED ein
      }
    }
  }
```

Anlagen 3, TinyBMS Kommunikationsprotokoll

2.1.6. Read battery pack voltage (*Reg:36*)

Request to *BMS*																	
CAN identifier 11 bits											8 data bytes*						
10	9	8	7	6	5	4	3	2	1	0	Byte 1	Byte 2*	Byte 3*	Byte 4*	Byte 5*	Byte 6*	Byte 7*	Byte 8*
0	1	0	0	0	Node ID (0x01...0x3F)						0x14	0x00	0x00	0x00	0x00	0x00	0x00	0x00
					Default node ID - 0x01													

* - Last command bytes with zeros can be ignored

Response from *BMS* [OK]																	
CAN identifier 11 bits											6 data bytes						
10	9	8	7	6	5	4	3	2	1	0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	
0	1	0	0	1	Node ID (0x01...0x3F)						0x01	0x14	DATA:LSB	DATA	DATA	DATA:MSB	
					Default node ID - 0x01								[FLOAT]				

Response from *BMS* [ERROR]													
CAN identifier 11 bits											3 data bytes		
10	9	8	7	6	5	4	3	2	1	0	Byte 1	Byte 2	Byte 3
0	1	0	0	1	Node ID (0x01...0x3F)						0x00	0x14	ERROR
					Default node ID - 0x01								

ERROR – Response error code

2.1.7. Read battery pack current (*Reg:38*)

Request to *BMS*																	
CAN identifier 11 bits											8 data bytes*						
10	9	8	7	6	5	4	3	2	1	0	Byte 1	Byte 2*	Byte 3*	Byte 4*	Byte 5*	Byte 6*	Byte 7*	Byte 8*
0	1	0	0	0	Node ID (0x01...0x3F)						0x15	0x00	0x00	0x00	0x00	0x00	0x00	0x00
					Default node ID - 0x01													

* - Last command bytes with zeros can be ignored

Response from *BMS* [OK]																	
CAN identifier 11 bits											6 data bytes						
10	9	8	7	6	5	4	3	2	1	0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	
0	1	0	0	1	Node ID (0x01...0x3F)						0x01	0x15	DATA:LSB	DATA	DATA	DATA:MSB	
					Default node ID - 0x01								[FLOAT]				

Response from *BMS* [ERROR]													
CAN identifier 11 bits											3 data bytes		
10	9	8	7	6	5	4	3	2	1	0	Byte 1	Byte 2	Byte 3
0	1	0	0	1	Node ID (0x01...0x3F)						0x00	0x15	ERROR
					Default node ID - 0x01								

ERROR – Response error code

2.1.8. Read battery pack max. cell voltage (*Reg:41*)

Request to *BMS*																	
CAN identifier 11 bits											8 data bytes*						
10	9	8	7	6	5	4	3	2	1	0	Byte 1	Byte 2*	Byte 3*	Byte 4*	Byte 5*	Byte 6*	Byte 7*	Byte 8*
0	1	0	0	0	Node ID (0x01...0x3F)						0x16	0x00	0x00	0x00	0x00	0x00	0x00	0x00
					Default node ID - 0x01													

* - Last command bytes with zeros can be ignored

Response from *BMS* [OK]														
CAN identifier 11 bits											4 data bytes			
10	9	8	7	6	5	4	3	2	1	0	Byte 1	Byte 2	Byte 3	Byte 4
0	1	0	0	1	Node ID (0x01...0x3F)						0x01	0x16	DATA:LSB	DATA:MSB
					Default node ID - 0x01								[UINT_16]	

* - Last command bytes with zeros can be ignored

Response from *BMS* [OK]																
CAN identifier 11 bits											6 data bytes					
10	9	8	7	6	5	4	3	2	1	0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
0	1	0	0	1	Node ID (0x01...0x3F)						0x01	0x19	DATA:LSB	DATA	DATA	DATA:MSB
					Default node ID - 0x01								[UINT32]			

Response from *BMS* [ERROR]													
CAN identifier 11 bits											3 data bytes		
10	9	8	7	6	5	4	3	2	1	0	Byte 1	Byte 2	Byte 3
0	1	0	0	1	Node ID (0x01...0x3F)						0x00	0x19	ERROR
					Default node ID - 0x01								

ERROR – Response error code

2.1.12. Read *Tiny BMS* estimated *SOC* value (*Reg:46*)

Request to *BMS*																	
CAN identifier 11 bits											8 data bytes*						
10	9	8	7	6	5	4	3	2	1	0	Byte 1	Byte 2*	Byte 3*	Byte 4*	Byte 5*	Byte 6*	Byte 7*	Byte 8*
0	1	0	0	0	Node ID (0x01...0x3F)						0x1A	0x00	0x00	0x00	0x00	0x00	0x00	0x00
					Default node ID - 0x01													

* - Last command bytes with zeros can be ignored

Response from *BMS* [OK]																
CAN identifier 11 bits											6 data bytes					
10	9	8	7	6	5	4	3	2	1	0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
0	1	0	0	1	Node ID (0x01...0x3F)						0x01	0x1A	DATA:LSB	DATA	DATA	DATA:MSB
					Default node ID - 0x01								[UINT32]			

Response from *BMS* [ERROR]													
CAN identifier 11 bits											3 data bytes		
10	9	8	7	6	5	4	3	2	1	0	Byte 1	Byte 2	Byte 3
0	1	0	0	1	Node ID (0x01...0x3F)						0x00	0x1A	ERROR
					Default node ID - 0x01								

ERROR – Response error code

2.1.13. Read *Tiny BMS* device temperatures (*Reg:48*, *Reg:42*, *Reg:43*)

Request to *BMS*																	
CAN identifier 11 bits											8 data bytes*						
10	9	8	7	6	5	4	3	2	1	0	Byte 1	Byte 2*	Byte 3*	Byte 4*	Byte 5*	Byte 6*	Byte 7*	Byte 8*
0	1	0	0	0	Node ID (0x01...0x3F)						0x1B	0x00	0x00	0x00	0x00	0x00	0x00	0x00
					Default node ID - 0x01													

* - Last command bytes with zeros can be ignored

Response from *BMS* [OK] – MSG 1																
CAN identifier 11 bits											6 data bytes					
10	9	8	7	6	5	4	3	2	1	0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
0	1	0	0	1	Node ID (0x01...0x3F)						0x01	0x1B	PL	DATA1:LSB	DATA1:MSB	0x00
					Default node ID - 0x01									[INT_16]		

Response from *BMS* [OK] – MSG 2																
CAN identifier 11 bits											6 data bytes					
10	9	8	7	6	5	4	3	2	1	0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
0	1	0	0	1	Node ID (0x01...0x3F)						0x01	0x1B	PL	DATA2:LSB	DATA2:MSB	0x01
					Default node ID - 0x01									[INT_16]		

Response from *BMS* [OK] – MSG 3																
CAN identifier 11 bits											6 data bytes					
10	9	8	7	6	5	4	3	2	1	0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
0	1	0	0	1	Node ID (0x01...0x3F)						0x01	0x1B	PL	DATA3:LSB	DATA3:MSB	0x02
					Default node ID - 0x01									[INT_16]		

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Nürnberg,30.05.2023

30.05.2023

Yang Zhou