Vertiefung Microcomputertechnik

CAN-TO-GO-SYSTEM

Von Michael Graml und Leonard Kreil

Betruer/Prüfer: Prof. Dr. Stefan Krämer

Abgabe: 12.02.2024

Zusammenfassung / Abstract

Das Projekt "CAN-TO-GO-SYSTEM", entwickelt von Michael Graml und Leonard Kreil, stellt eine fortschrittliche Lösung für die Überwachung und Diagnose von CAN-Bussystemen dar. Es zielt darauf ab, Anwendern bei der Einrichtung und Fehlersuche in CAN-Netzwerken zu assistieren. Das System zeichnet sich durch eine Kombination von Hardware- und Softwarekomponenten aus, die eine effiziente Analyse und Interaktion mit dem CAN-Netzwerk ermöglichen.

Die Hardware besteht aus einem sorgfältig Leiterplattenlayout, das einen ESP32-Mikrocontroller, ein Display, verschiedene Anschlüsse und LEDs integriert. Der ESP32 spielt eine zentrale Rolle bei der Verarbeitung der CAN-Nachrichten und der Interaktion mit anderen Hardwarekomponenten. Das System bietet außerdem eine Benutzeroberfläche über ein kleines Display und eine ergänzende App sowie Web-Integration für eine detailliertere Ansicht und Analyse der CAN-Nachrichten.

Die Softwarearchitektur basiert auf dem ESP32, der eine zentrale Rolle bei der Verarbeitung und Darstellung der CAN-Nachrichten übernimmt. Zusätzlich wird eine benutzerfreundliche GUI über das Flutter-Framework bereitgestellt, die eine flexible Interaktion auf verschiedenen Geräten ermöglicht. Die Implementierung der Software folgt der Clean Architecture und den SOLID-Prinzipien, um eine effiziente und erweiterbare Lösung zu gewährleisten.

Abstract

The "CAN-TO-GO-SYSTEM"project, developed by Michael Graml and Leonard Kreil, represents an advanced solution for monitoring and diagnosing CAN bus systems. It aims to assist users in setting up and troubleshooting CAN networks. The system is characterized by a combination of hardware and software components that enable efficient analysis and interaction with the CAN network.

The hardware consists of a meticulously designed PCB layout, incorporating an ESP32 microcontroller, a display, various connectors, and LEDs. The ESP32 plays a central role in processing CAN messages and interacting with other hardware components. Additionally, the system provides a user interface through a small display and complementary app and web integration for more detailed viewing and analysis of CAN messages.

The software architecture is based on the ESP32, which takes a central role in processing and displaying CAN messages. A user-friendly GUI is also provided through the Flutter framework, allowing flexible interaction across different devices. The software implementation follows Clean Architecture and SOLID principles to ensure an efficient and expandable solution.

Inhaltsverzeichnis

1 Einleitung

Die Arbeit wurde gemeinschaftlich von Michael Graml und Leonard Kreil durchgeführt. Sofern nicht weiter gekennzeichnet ist davon auszugehen das alle Arbeiten gemeinschaftlich bearbeitet wurden.

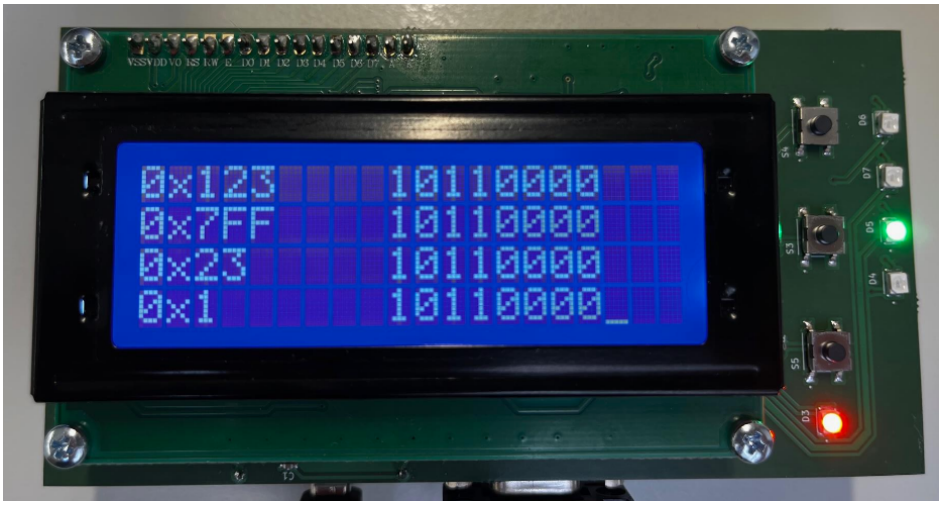

Abbildung 1.1: CAN-TO-GO System

1.1 CAN

Controller Area Network (CAN) ist ein standardisiertes, robustes Fahrzeugbussystem, das für die Kommunikation zwischen verschiedenen Steuergeräten (ECUs) in Fahrzeugen und anderen industriellen Anwendungen konzipiert wurde. Es ermöglicht den zuverlässigen Datenaustausch mit hoher Fehlertoleranz und geringer Latenz, was besonders in Echtzeitanwendungen wichtig ist. CAN-Busse verwenden ein Nachrichtenbasiertes Protokoll, bei dem jede Nachricht eine eindeutige ID hat, die ihre Priorität bestimmt. Dieses System ist besonders nützlich bei der Diagnose von Netzwerkproblemen, da die Priorisierung von Nachrichten und Fehlererkennungsmechanismen die Fehlersuche erleichtern. Die physikalischen Aspekte des CAN-Busses, wie Abschlusswiderstände und die Verkabelung (CANL/CANH), sind ebenfalls entscheidend für die Netzwerkintegrität. Die Anpassungsfähigkeit hinsichtlich der Baudrate und die Unterstützung verschiedener Netzwerkstrukturen machen CAN vielseitig einsetzbar.

1.2 Can-Datentelegram

Ein CAN-Datentelegramm beginnt mit einem Startbit zur Synchronisation der kommunizierenden Geräte. Der Identifier gibt die Nachrichtenpriorität an und dient der Busarbitrierung, wobei das RTR-Bit zwischen einem Daten- und einem Datenanforderungstelegramm unterscheidet. IDE ist die Identifier Extension, die

Start	Identifier	RTR	IDE	r0	DLC	DATA	CRC	ACK	EOF+IFS
1 Bit	11 Bit	1 Bit	1 Bit	1 Bit	4 Bit	0...8 Byte	15 Bit	2 Bit	10 Bit

Abbildung 1.2: Standard Datentelegram [7]

Start	Identifier	SRR	IDE	Identifier	RTR	r1	r0	DLC	DATA	CRC	ACK	EOF+IFS
1 Bit	11 Bit	1 Bit	1 Bit	18 Bit	1 Bit	1 Bit	1 Bit	4 Bit	0...8 Byte	15 Bit	2 Bit	10 Bit

Abbildung 1.3: Extended Datentelegram [7]

anzeigt, ob ein Standard- (11-Bit-Identifier siehe Abbildung 1.2) oder ein Extended-Frame (29-Bit-Identifier siehe Abbildung 1.3) verwendet wird. DATA enthält die eigentlichen Nutzdaten. CRC ist eine Prüfsumme zur Fehlererkennung. ACK signalisiert den korrekten Empfang der Nachricht. EOF und IFS kennzeichnen das Ende des Datentelegramms und den erforderlichen Abstand bis zum nächsten Frame. Im Extended Frame ersetzt das SRR-Bit das RTR-Bit des Standard Frames, und die DLC-Information gibt die Länge der Daten an.

1.3 Anfordderungsanalyse

Das "Can to GoSystem wird entwickelt, um Anwendern bei Problemen beim Aufbau eines CAN-Busses zu assistieren und zu diagnostizieren, wo genau die Schwierigkeiten liegen. Es dient als funktionssicherer CAN-Teilnehmer, der in der Lage ist, CAN-Nachrichten zuverlässig zu lesen und zu interpretieren. Kernmerkmale umfassen ein Anschlusskästchen mit SUB-D-Stecker und Status-LEDs, die den Betriebszustand des CAN-Busses anzeigen. Ein optionaler Abschlusswiderstand, der nach Bedarf zugeschaltet werden kann, sowie die Anpassungsfähigkeit der Baudrate gehören ebenfalls zu den wesentlichen Anforderungen. Darüber hinaus wird das System die CAN-Nachrichten und die zugehörigen Sender-IDs sichtbar machen.

1.4 Endprodukt

1.4.1 Überblick

Das Endprodukt des "Can to GoSystems bildet eine fortschrittliche Lösung für die Überwachung und Diagnose von CAN-Bussystemen. Es integriert die erforderlichen Funktionen aus der Anforderungsanalyse mit zusätzlichen Merkmalen, die seine Benutzerfreundlichkeit und Vielseitigkeit erhöhen.

1.4.2 Hauptmerkmale

- **Anschluss und Konnektivität:** Ausgestattet mit einem Anschlusskästchen und SUB-D-Stecker, verfügt das System über Status-LEDs, die den Betriebszustand des CAN-Busses anzeigen. Diese visuellen Indikatoren erleichtern das schnelle Erkennen von Verbindungsproblemen.
- **Funktionalität und Anpassungsfähigkeit:** Ein optionaler Abschlusswiderstand kann nach Bedarf hinzugeschaltet werden. Die Anpassbarkeit der Baudrate gewährleistet eine breite Kompatibilität mit verschiedenen CAN-Bus-Konfigurationen.

- **Display und Anzeige:** Das Gerät umfasst ein kleines Display, das CAN-Nachrichten anzeigt. Dieses Display bietet eine direkte, wenn auch begrenzte, Einsicht in die CAN-Kommunikation.
- **Energieversorgung:** Es enthält keinen eigenen Akku, kann aber durch eine externe Powerbank mit Energie versorgt werden, was eine flexible Nutzung ermöglicht.

1.4.3 Zusätzliche Funktionen

- **App- und Web-Integration:** Ergänzend zum kleinen Display des Geräts ermöglichen eine App und eine Website die detaillierte Ansicht von CAN-Nachrichten, inklusive Zeitstempeln. Diese digitalen Plattformen erlauben eine umfassende Analyse da sie eine größere Benutzeroberfläche als das Display bieten.

2 Hardwareaufbau und Leiterplattenentwurf

Der Hardwareaufbau dieses Projekts basiert auf einer sorgfältig gestalteten Leiterplatte, die alle erforderlichen Hardwarekomponenten integriert. Der Prozess des Leiterplattenentwurfs wurde mittels KiCad, einem Open-Source-Tool für PCB-Design, durchgeführt.

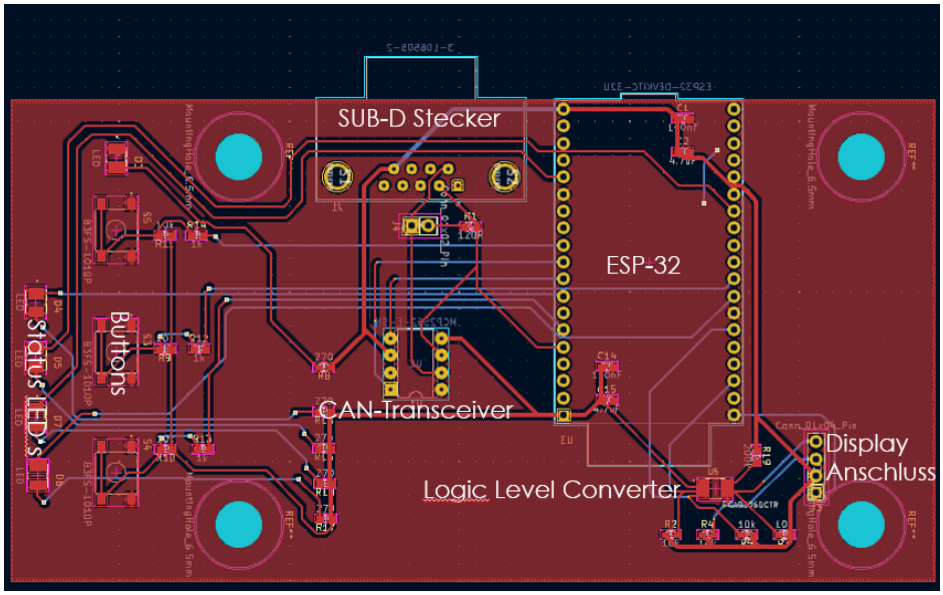

Abbildung 2.1: Leiterplattenentwurf

2.1 Leiterplattenentwurf

2.1.1 Entwurfsprozess

- **Komponentenauswahl und -platzierung:** Zunächst wurden die benötigten elektronischen Bauteile ausgewählt und in den Schaltplan von KiCad integriert. Jedem Bauteil wurde ein entsprechender Footprint zugewiesen. Diese Footprints sind entscheidend, da sie die physische Größe und Platzierung der Komponenten auf der Leiterplatte bestimmen.

- **Layoutgestaltung:** Im Anschluss wurden die Bauteile sorgfältig auf der Leiterplatte platziert. Besonderes Augenmerk lag dabei auf der logischen Gruppierung der Komponenten und der Minimierung der Leitungslängen, um Effizienz und Signalintegrität zu gewährleisten.

2.1.2 Herstellung und Bestückung

- **Leiterplattenbestellung:** Die Leiterplatte wurde bei Aisler in Auftrag gegeben. Die Entscheidung, die Bauteile separat zu bestellen, wurde aus Kostengründen getroffen.

- **Komponentenbeschaffung:** Die einzelnen elektronischen Bauteile wurden von Mouser Electronics bezogen.
- **Bestückung der Leiterplatte:** Die Montage der Bauteile auf der Leiterplatte erfolgte mittels Lötpaste. Bei diesem Prozess wurde besonders auf Präzision und Sauberkeit geachtet, um Kurzschlüsse oder Fehlverbindungen zu vermeiden, die die Funktionalität der Schaltung beeinträchtigen könnten.

Die detaillierte Kostenaufstellung der einzelnen Bauteile sowie der Leiterplatte selbst ist in Tabelle 2.1 dargestellt. Diese Übersicht bietet eine klare Kostentransparenz und erleichtert die Nachvollziehbarkeit des Projekts.

2.2 ESP32

Der ESP32 ist das Herzstück unseres eingebetteten Systems und übernimmt die zentrale Verarbeitung und Koordination aller Aufgaben. Als hochintegrierter Mikrocontroller verbindet er verschiedene Elemente des Systems, wie die CAN-Schnittstelle, das Display, Status-LEDs und Taster, zu einem funktionierenden Ganzen.

Im Rahmen der CAN-Kommunikation liest der ESP32 die über den CAN-Bus gesendeten Nachrichten und verarbeitet diese. Außerdem steuert ESP32 das Display über das I2C-Protokoll, wobei ein Logikpegelwandler für die Anpassung der unterschiedlichen Spannungsniveaus zwischen dem 3,3-Volt-System des ESP32 und dem 5-Volt-Display verwendet wird.

Die Taster sind über Pull-up-Widerstände an den ESP32 angeschlossen und ermöglichen es den Benutzern, durch einfaches Drücken die Konfigurationseinstellungen vorzunehmen. Der ESP32 interpretiert diese Eingaben und führt die entsprechenden Aktionen aus, wie beispielsweise das Wechseln der Baudrate.

Durch seine Vielseitigkeit und leistungsstarke Verarbeitungskapazität bildet der ESP32 das Rückgrat des Systems.

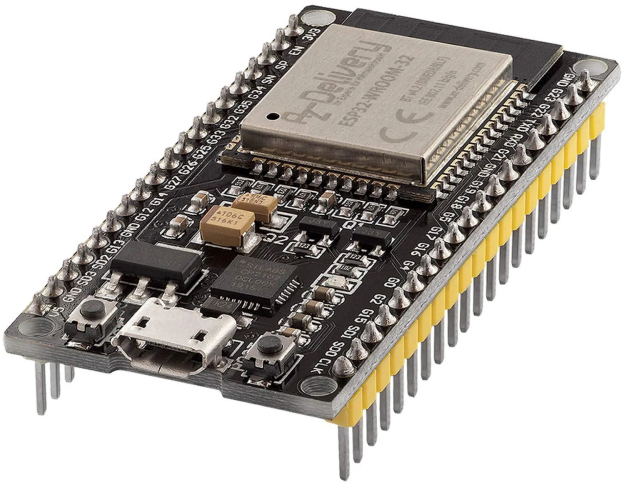

Abbildung 2.2: ESP32 [2]

2.3 Display

Das Display dient in unserem eingebetteten System als primäre Benutzerschnittstelle, indem es die Baudrateneinstellungen im Konfigurationsmodus sowie die CAN-Nachrichten und ihre Identifier im Operationsmodus anschaulich visualisiert. In unserem System erfolgt die Kommunikation zwischen dem ESP32-Mikrocontroller und dem Display über das I2C-Protokoll (Inter-Integrated Circuit), einem weit verbreiteten Kommunikationsstandard, der für seine Einfachheit und Effizienz in der synchronen seriellen Datenübertragung bekannt ist.

I2C nutzt nur zwei Leitungen – eine für das serielle Daten-Signal (SDA) und eine für das serielle Clock-Signal (SCL) – und ermöglicht es dem Mikrocontroller, mehrere Geräte über einen einzigen Bus zu steuern, wobei jedes Gerät über eine einzigartige Adresse identifiziert wird. Dieser Bus ermöglicht eine bidirektionale Kommunikation, was bedeutet, dass der ESP32 Daten an das Display senden und gleichzeitig Statusinformationen vom Display empfangen kann.

Da der ESP32 mit einer Logikspannung von 3,3 Volt arbeitet, während das Display für eine Betriebsspannung von 5 Volt ausgelegt ist, wurde ein Logikpegelwandler verwendet, um die Signale zwischen diesen beiden Spannungsebenen zu übersetzen (siehe Abschnitt ...). Zusätzlich wird für die physische Verbindung des Displays ein serieller I2C/TWI-Schnittstellenadapter verwendet, der bereits im Lieferumfang des Kits enthalten ist.

Abbildung 2.3: Display mit Displayanschluss [1]

2.4 SUB-D Stecker

Der SUB-D-Stecker dient der Einbindung von CAN-Nachrichten in das betreffende System. Über diese Schnittstelle lässt sich das 'CAN to go'-System mit dem zu testenden CAN-Netzwerk verbinden. Die Konfiguration der Pin-Belegung ist spezifisch angeordnet: CAN-Low ist an Pin 2, CAN-High an Pin 6, der Masseanschluss (GND) an Pin 3 und die Versorgungsspannung von +5V an Pin 9 angebunden. Abhängig von den systemseitigen Erfordernissen kann ein 120-Ohm-Terminierungswiderstand zwischen CAN-Low und CAN-High integriert werden, um die Signalintegrität zu gewährleisten. Die Schaltung dieses Terminierungswiderstandes erfolgt über den Anschluss J4, der in der grafischen Darstellung blau umkreist ist (siehe Abbildung 2.4).

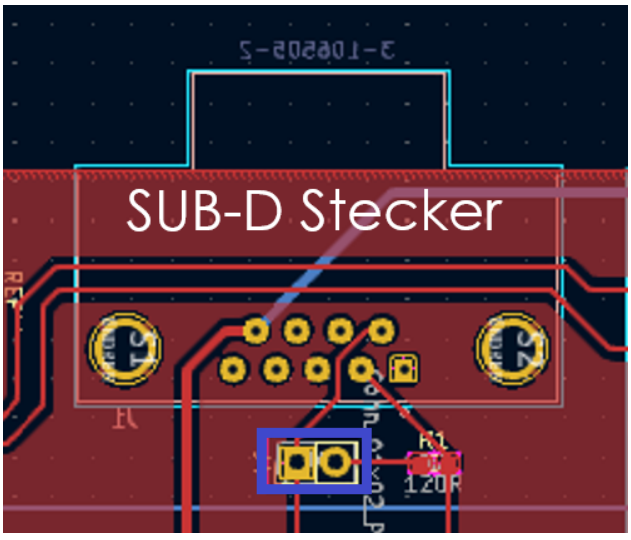

Abbildung 2.4: SUB-D Stecker und Terminierungswiderstand

2.5 Can Transciever

Der MCP2562-E-P Transceiver spielt eine zentrale Rolle im Hardwareaufbau für die CAN-Kommunikation, indem er die Signale aus dem CAN-Netzwerk in digitale Signale umwandelt, die vom ESP32-Mikrocontroller verarbeitet werden können. Der Transceiver agiert als Bindeglied zwischen den differenziellen Signalen des CAN-Busses und den digitalen I/O-Pins des ESP32. Er wandelt die auf dem CAN-Bus empfangenen Hoch- und Niedrigzustände (CAN_H und CAN_L) in logische Pegel um, die der Mikrocontroller verstehen kann. Dies ermöglicht eine reibungslose und zuverlässige Kommunikation zwischen den CAN-Geräten und dem Mikrocontroller.

Zur Implementierung dieser Funktion müssen die CAN_H und CAN_L Leitungen des MCP2562-E-P an die entsprechenden Leitungen des CAN-Busses angeschlossen werden. Die Tx und Rx Pins des Transceivers werden dann mit den entsprechenden digitalen I/O-Pins des ESP32 verbunden, um die Kommunikation zu ermöglichen. Die genaue Pinbelegung und Konfiguration ist in dem Datenblatt des Transceivers zu finden [6].

2.6 Status LEDS

In unserem eingebetteten System fungieren Leuchtdioden (LEDs) als visuelle Indikatoren für verschiedene Zustände des Gerätes. Ihre Hauptfunktion besteht darin, den Benutzer über den aktuellen CAN Status zu informieren. Die LEDs sind über 270-Ohm-Widerstände an den Mikrocontroller angeschlossen, um den Strom zu begrenzen, der durch die LEDs fließt. Dies schützt die LEDs vor Überstrom, der sie beschädigen könnte, und stellt sicher, dass sie innerhalb ihrer spezifizierten Stromgrenzen betrieben werden, was zu einer längeren Lebensdauer und zuverlässigeren Leistung führt.

- **Orange LED (D3):** Die Beleuchtung dieser LED zeigt an, dass das System mit Strom versorgt wird, und dient somit als Power-On-Statusanzeige.
- **Rote LED (D4):** Diese LED beginnt zu leuchten, wenn die Nachrichten auf dem angeschlossenen CAN-Bus nicht interpretiert werden können. Dies dient als Indikator für Störungen oder Fehler im CAN-System.
- **Grüne LED (D5):** Die grüne LED signalisiert, dass die Nachrichten des CAN-Bus korrekt interpretiert werden und das System ordnungsgemäß funktioniert.
- **Blaue LED (D6):** Diese LED beginnt zu blinken wenn eine CAN Nachricht gesendet wird.
- **Blaue LED (D7):** Diese LED beginnt zu blinken wenn eine CAN Nachricht empfangen wird.

Die Ansteuerung der LEDs erfolgt direkt durch den ESP32-Mikrocontroller, der die Fähigkeit besitzt, seine GPIO-Pins (General Purpose Input/Output) als Ausgänge zu konfigurieren. Durch Programmierung des Mikrocontrollers kann jeder dieser Pins einen High- oder Low-Zustand annehmen, wobei High bedeutet, dass der Pin eine Spannung ausgibt, die ausreicht, um die LED zum Leuchten zu bringen. Der Low-Zustand hingegen unterbricht den Stromfluss, sodass die LED erlischt.

2.7 Buttons

In unserem System werden Taster (Buttons) als Eingabemittel für Benutzerinteraktionen verwendet. Sie ermöglichen es dem Benutzer, Steuersignale an den Mikrocontroller zu senden, der diese dann entsprechend der programmierten Logik interpretiert. Die Taster sind einfache, aber effektive Komponenten in der Mensch-Maschine-Interaktion innerhalb eingebetteter Systeme.

Die drei Taster sind jeweils über einen Pull-up-Widerstand an eine Versorgungsspannung (+3V3) angeschlossen. Dies sorgt dafür, dass das Signal am entsprechenden Eingangspin des ESP32 standardmäßig auf einem hohen Logikniveau (High) liegt. Wird der Taster gedrückt, schließt sich der Stromkreis, und der Eingangspin wird auf das niedrige Logikniveau (Ground) gezogen.Die zusätzlichen Widerstände (R12, R13, R14) dienen als Strombegrenzungswiderstände und schützen den Mikrocontroller vor hohen Strömen, die beim Schließen des Tasterkreises entstehen könnten. Diese Anordnung wird als Pull-up-Konfiguration bezeichnet und ist eine gängige Methode, um den Zustand eines Eingangspins zu definieren, wenn kein Signal anliegt.

Im Konfigurationsmodus unseres Systems ermöglichen es die Taster dem Benutzer, die Einstellungen für die Baudrate zu navigieren und zu bestätigen. Jeder Taster hat eine spezifische Rolle:

- **Taster S3:** Dieser Taster ermöglicht es dem Benutzer, auf dem Display nach oben zu navigieren.
- **Taster S4:** Der Taster S4 bietet dem Benutzer die Möglichkeit, Einstellungen zu bestätigen, wie beispielsweise die Auswahl der Baudrate.
- **Taster S5:** Sollte der Benutzer im Display nach unten navigieren müssen, wird Taster S5 verwendet.

2.8 Logic Level Converter

Im Kontext eingebetteter Systeme spielt die Signalintegrität zwischen verschiedenen elektronischen Komponenten eine entscheidende Rolle. Verschiedene Bauteile operieren oft mit unterschiedlichen Logikpegeln, was bedeutet, dass die Spannung, die ein High-Signal (logische "1") repräsentiert, zwischen diesen Komponenten variieren kann. Um eine korrekte und sichere Kommunikation zwischen solchen Bauteilen zu gewährleisten, werden Logikpegelwandler eingesetzt.

In unserem System haben wir den Logikpegelwandler PCA9306DCTR implementiert, um eine bidirektionale Schnittstelle zwischen dem 3,3-Volt-Logikpegel des ESP32-Mikrocontrollers und dem 5-Volt-System des Displays zu schaffen. Dieser Schritt ist notwendig, da eine direkte Verbindung ohne entsprechende Anpassung der Spannungsniveaus zu einem für das Display nicht detektierbaren Signal führen würde.

Der PCA9306DCTR zeichnet sich durch eine Schaltung aus, die mit Transistoren und Pull-up-Widerständen ausgestattet ist. Diese Komponenten sind präzise konfiguriert, um eine nahtlose Umwandlung von Signalen zu ermöglichen: Ein High-Signal wird von der niedrigeren Spannung des ESP32 (3,3 V) auf die höhere Spannung des Displays (5 V) angehoben und umgekehrt. Diese Funktionsweise stellt sicher, dass es zu keiner Signalverzerrung oder Beeinträchtigung der Signalintegrität kommt. Der PCA9306DCTR ermöglicht so eine reibungslose und sichere Interaktion zwischen dem ESP32 und dem Display, was für die Zuverlässigkeit und Robustheit des Gesamtsystems unerlässlich ist.

2.9 Bypass Kondensatoren

Die Stabilität und Zuverlässigkeit der Stromversorgung sind für die Funktionalität des Systems von entscheidender Bedeutung. Um dies zu gewährleisten, werden Bypass- oder Abblockkondensatoren an den Spannungsausgängen des ESP32 eingesetzt. Diese Kondensatoren spielen eine zentrale Rolle bei der Minimierung von Spannungsschwankungen und elektrischem Rauschen, die die Leistung und das Verhalten des Systems negativ beeinflussen können.

Stabilisierung der Versorgungsspannung
Bypass-Kondensatoren dienen als lokale Energiereserven, die direkt an den Spannungsausgängen des ESP32 positioniert sind. Sie bieten eine schnelle Reaktionsquelle für Energie, um die Versorgungsspannung konstant zu halten, wenn der Mikrocontroller zwischen energieintensiven Prozessen wechselt.

Rauschunterdrückung
Elektronische Schaltungen sind anfällig für Rauschen und transiente Störungen, die von anderen Teilen der Schaltung oder externen Quellen stammen können. Die Bypass-Kondensatoren filtern effektiv das hochfrequente Rauschen heraus, indem sie es zur Masse ableiten, und verhindern so, dass es die Leistung des Mikrocontrollers beeinträchtigt.

Reaktion auf Lastwechsel
Der ESP32 kann plötzliche Lastwechsel erfahren, beispielsweise beim Übergang von einem niedrigen Stromverbrauchszustand zu einer hohen Rechenleistung. Bypass-Kondensatoren helfen, solche Lastwechsel zu bewältigen, indem sie sofort Energie liefern oder aufnehmen, um die Versorgungsspannung zu stabilisieren.

Dimensionierung der Bypass-Kondensatoren
Zwei Arten von Bypass-Kondensatoren werden typischerweise verwendet:

- **Kondensatoren mit 100nF:** Diese sind für die Filterung hochfrequenter Störungen ausgelegt und sind ein Standard in vielen Schaltkreisen, die integrierte Schaltungen enthalten.
- **Kondensatoren mit 4,7µF:** Sie bieten eine größere Kapazität für die Energieaufnahme und sind besonders effektiv, um die Spannung bei größeren Lastwechseln zu stabilisieren.

Diese Kondensatoren ergänzen sich gegenseitig, wobei der 100nF-Kondensator schnelle, kurzzeitige Schwankungen abfängt, während der 4,7µF-Kondensator für länger anhaltende Stabilität sorgt. Die Kombination beider Typen stellt sicher, dass der ESP32 unter allen Betriebsbedingungen effizient und fehlerfrei arbeiten kann.

2.10 Stückliste

Name	Bauteilbezeichnung	Spezifikationen	Menge
ESP32	ESP-32 Dev Kit C V4		1
Widerstand		270 Ohm	5
Widerstand		120 Ohm	1
Widerstand		1k Ohm	3
Widerstand		10k Ohm	7
Widerstand		200k Ohm	1
Kondensator		4.7 µF	2
Kondensator		100nF	2
LED			5
SUB-D Stecker	3-106505-2		1
Button	B3FS-1010P		3
CAN Transceiver	MCP2562-E-P		1
Logic Level Converter	PCA9306DCTR		1
Display	WayinTop 20x4 2004 LCD		1
Display Anschluss	TWI IIC I2C LCD		1
Leiterplatte			1

Tabelle 2.1: Stückliste der verwendeten Bauteile

3 Software

Die Software des Systems setzt sich aus zwei Hauptkomponenten zusammen. Zur Verarbeitung der eingehenden CAN-Nachrichten wird eine ESP32 verwendet, der auf dem espidf-Framework basiert. Dieser Microcontroller übernimmt nicht nur das Empfangen der Nachrichten, sondern zeigt diese auch auf einem LCD-Display an und stellt sie über eine REST-API zur Verfügung. Um eine benutzerfreundlichere Oberfläche zu bieten, wurde zusätzlich eine GUI entwickelt. Diese basiert auf dem von Google entwickelten Framework Flutter. Dank dieser GUI können die CAN-Nachrichten komfortabel über eine Windows-Anwendung, einen Browser oder einer App ausgelesen werden. Das Architekturbild 3.1 stellt die grundsätzliche Architektur des Systems dar. Auf die einzelnen Komponenten wird im folgendem genauer eingegangen.

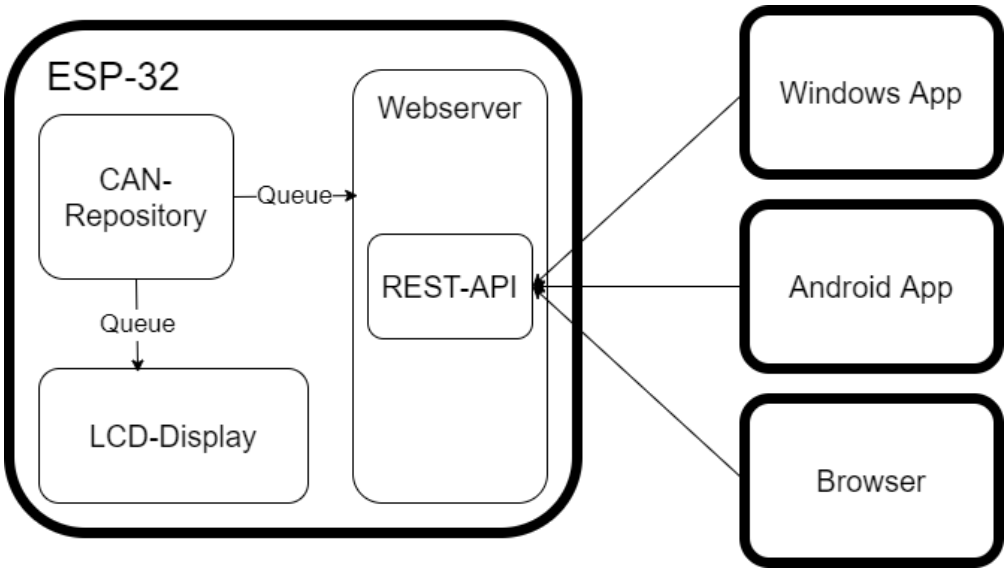

Abbildung 3.1: Architekturbild

3.1 ESP32

3.1.1 Architektur

Projektstruktur

Bei der Projektstruktur 3.2 wurde sich an der Clean Architecture von Robert C. Martin orientiert [5]. Da diese für Objekt Orientierte Programmiersprachen ausgelegt ist, wurde die Struktur an unsere Bedürfnisse angepasst. Der Benefit der Projektstruktur ist, dass die Business Logiken, welche in den Controllern implementiert sind, von der Verarbeitung der Daten gekapselt werden. Jeder Controller ist ein FreeRTOS Task, welche die gleiche Grundstruktur, eine FSM, besitzt. Auf die FSM wird in 3.1.1 genauer eingegangen.

```
    while (1)
    {
        switch (fsm_controller_current_state)
        {
        case STARTING:
            break;
        case CONFIGURATION:
            break;
        case OPERATION:
            break;
        }
        vTaskDelay(10);
    }
```

Der Bereich *data* ist für die Verarbeitung der Daten zuständig. Hier werden die Rohdaten in geeignete Strukturen transformiert, die für die Controller leicht zu verarbeiten sind. Die Strukturen sind in *models* definiert.

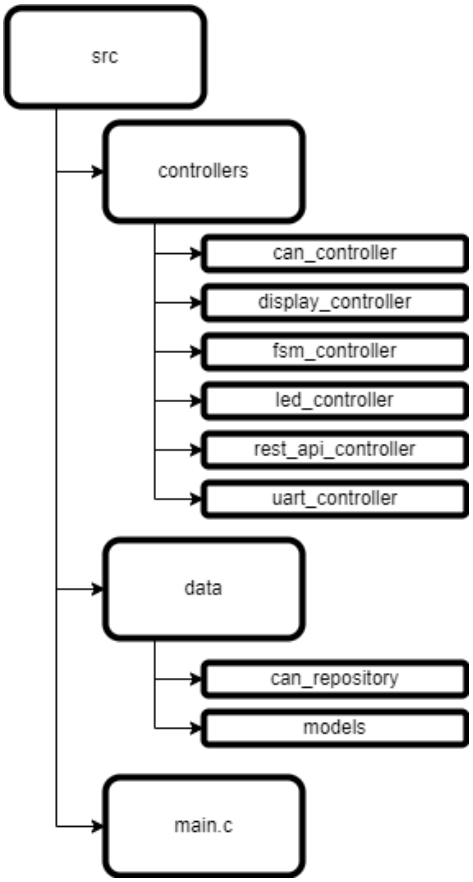

Abbildung 3.2: Projektstruktur ESP

Abbildung 3.3: FSM

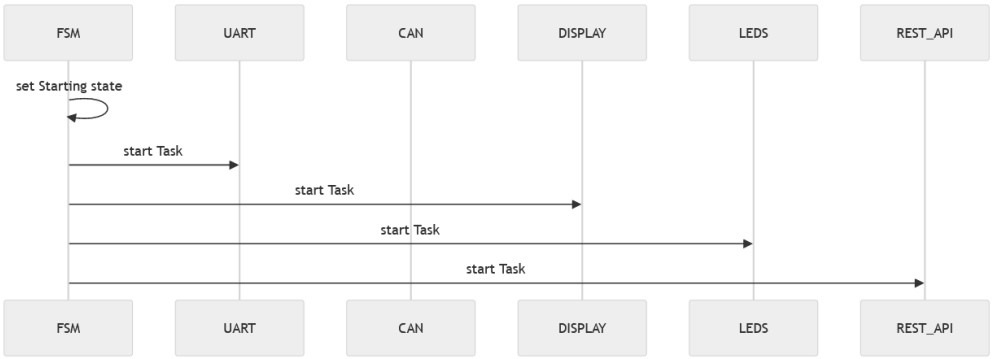

Abbildung 3.4: State STARTING

Finite State Machine

Um das Verhalten des Systems klar und strukturiert zu unterteilen, wurde eine Final State Machine (FSM) eingesetzt. Diese besteht aus den drei States *STARTING*, *CONFIGURATION* und *OPERATION* 3.3. Der State *STARTING* 3.4 ist für den Systemstart verantwortlich. Hierbei werden die einzelnen Controller als FreeRTOS Tasks gestartet. Daraufhin wird in den *CONFIGURATION* 3.5 State übergegangen. Hier wird das System mit den nötigen Einstellungen konfiguriert. So wird unter anderem die Baudrate des CAN Netzwerks vom User gesetzt und an den CAN Controller übertragen. Wenn alle Konfigurationen abgeschlossen sind, kann in den *OPERATION* 3.6 State gewechselt werden. In diesem Zustand wird das CAN Netzwerk laufend auf Funktionalität geprüft. Außerdem wird auf ankommende CAN Nachrichten gelauscht. Das Resultat der Funktionalitätsprüfung wird anschließend dem LED Controller mitgeteilt, sodass dieser die richtigen LEDs schalten kann. Wenn eine CAN Nachricht angekommen ist, wird diese vom CAN Repository verarbeitet und

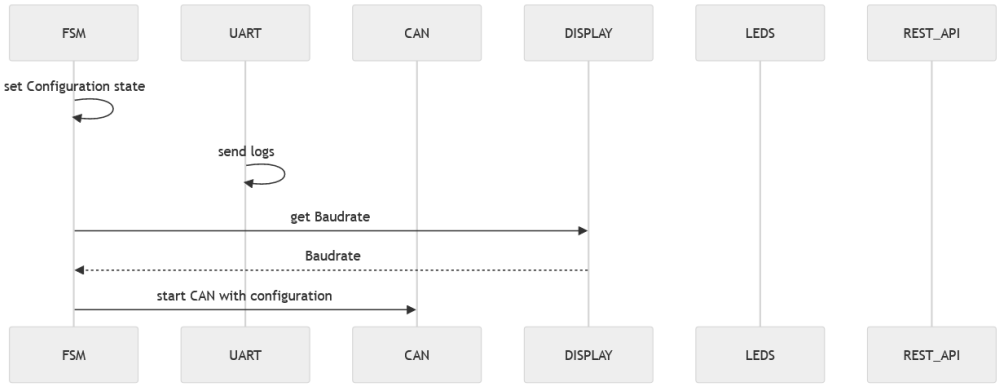

Abbildung 3.5: State CONFIGURATION

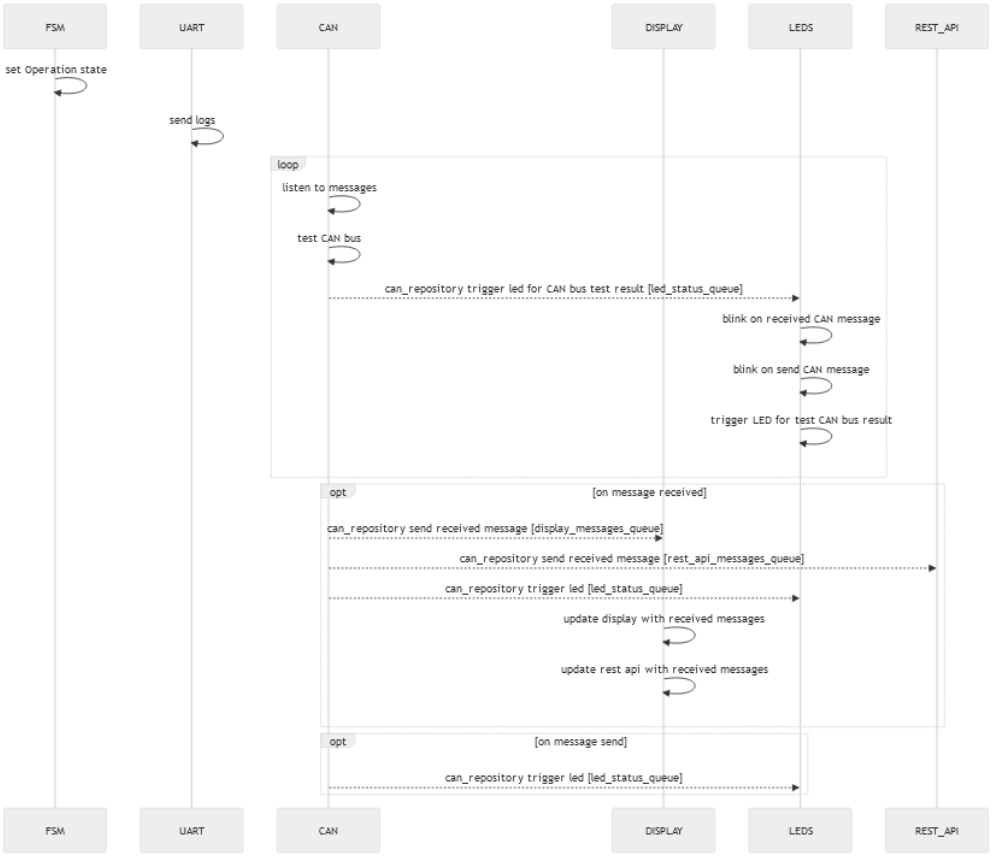

Abbildung 3.6: State OPERATION

auf die jeweiligen Queues aufgeteilt. Anschließend steht die Nachricht dem LED Controller, dem Display Controller und dem REST API Controller zur Verfügung. Jetzt können von den Controllern die richtigen LEDs geschalten, sowie das Display und die REST API mit der neuen Nachricht aktualisiert werden.

3.1.2 Wichtige Komponenten

CAN Repository

Ein zentraler Baustein der Software ist das CAN Repository mit den definierten Datenmodellen. Laut Robert C. Martins Clean Architecture [4] sollten für eine saubere Software Architektur Interface Adapter implementiert werden. Die Interface Adapter sind für die Konvertierung der Daten zwischen Data Source, in unserem Fall dem CAN Controller, und den UseCases, bei uns LEDs, Display und REST API, zuständig. Durch die vorhergehende Datenkonvertierung können sich die UseCases ihrer einzigen Aufgabe, nämlich die Anzeige der Daten widmen. Mit dieser Implementierung ist auch das Single Responsibility Principle der SOLID Prinzipien erfüllt [3]. Dementsprechend wurde für jeden Controller (LED, DISPLAY und REST_API) ein entsprechendes Datenmodell definiert. Diese Datenmodelle enthalten außschließlich Informationen, die für die Anzeige der Daten mit dem jeweiligen Controller nötig sind.

```
1     #LED
2     struct Led
3     {
```

```
4           enum Leds gpio;
5           enum LedValue value;
6       };
7
8       #DISPLAY
9       struct MessageItem
10      {
11          int id;
12          char text[21];
13      };
14
15      #REST_API
16      struct CanMessage
17      {
18          int64_t micros;
19          twai_message_t message;
20      };
```

Für die Konvertierung der Rohdaten des CAN Controllers, in die jeweiligen Datenmodelle wurde im CAN Repository, je eine Funktion implementiert. Danach wird das konvertierte Datenmodell in die jeweilige Queue geschrieben.

```
1  static struct MessageItem
       twai_message_to_display_message_item(twai_message_t twai_message)
2  {
3      struct MessageItem message_item;
4
5      message_item.id = twai_message.identifier;
6
7      const int index_data_start = 9;
8      sprintf(message_item.text, "0x%1X", twai_message.identifier);
9      const int needed_whitespaces = index_data_start -
           strlen(message_item.text);
10
11     for (int i = 0; i < needed_whitespaces; i++)
12     {
13         strcat(message_item.text, " ");
14     }
15
16     int arrayLength = sizeof(twai_message.data) /
           sizeof(twai_message.data[0]);
17
18     for (int i = 0; i < arrayLength; i++)
19     {
20         char item[10];
21         snprintf(item, sizeof(item), "%d", twai_message.data[i]);
22         strcat(message_item.text, item);
23     }
24
25     return message_item;
26 }
```

```
27
28 static struct CanMessage twai_message_to_can_message(twai_message_t
       twai_message)
29 {
30     struct CanMessage can_message;
31
32     int64_t uptime_micros = esp_timer_get_time();
33     can_message.micros = uptime_micros;
34     can_message.message = twai_message;
35
36     return can_message;
37 }
38
39 extern void can_repository_distribute_received_message(twai_message_t
       received_twai_message)
40 {
41     struct MessageItem received_message_item =
           twai_message_to_display_message_item(received_twai_message);
42     xQueueSend(display_messages_queue, &received_message_item, 1);
43
44     struct CanMessage received_can_message =
           twai_message_to_can_message(received_twai_message);
45     xQueueSend(rest_api_messages_queue, &received_can_message, 1);
46
47     struct Led led_blue_receive = {LED_BLUE_RECEIVE, LED_ON};
48     xQueueSend(led_status_queue, &led_blue_receive, 1);
49 }
```

Durch diese Implementierung unter Beachtung der Clean Architecture und SOLID Prinzipien ist die Data Source von den UseCases gekapselt. Somit ist eine leichte Wartbarkeit und Erweiterbarkeit der Software gewährleistet.

Display

Das Display verfügt aktuell über zwei verschiedene Anzeigemöglichkeiten. Zum einen kann ein Menü mit beliebigen Menüeinträgen erstellt werden, zum anderen können die aktuellen CAN Messages angezeigt werden.
Um die Software bei Bedarf mit weiteren Menüs einfach erweitern zu können wurde das *menu_presentation* Modul so gestaltet das ein beliebiges Menü erstellt werden kann. So kann ein Array bestehend aus *MenuItem* Elementen und eine Funktion *void (*func_enter)(int)* übergeben werden, welche bei betätigen des *ENTER* Buttons ausgeführt wird.

```
1 struct MenuItem
2 {
3     u_int8_t id;
4     u_int8_t is_selected;
5     char text[50];
6     void *value;
7 };
```

```
extern void menu_presentation_show(i2c_lcd1602_info_t *lcd_info, struct
    MenuItem menu[], int menu_size, void (*func_enter)(int), enum Button
    button_pressed, int initial_show)
{
    char selected_string[52] = "> ";

    if (button_pressed >= 0 || initial_show)
    {
        handle_button_pressed(button_pressed, menu, menu_size,
            func_enter);

        i2c_lcd1602_reset(lcd_info);
        for (int i = 0; i < menu_size; i++)
        {
            i2c_lcd1602_move_cursor(lcd_info, 0, i);

            if (menu[i].is_selected)
            {
                strcat(selected_string, menu[i].text);
                i2c_lcd1602_write_string(lcd_info, selected_string);
            }
            else
            {
                i2c_lcd1602_write_string(lcd_info, menu[i].text);
            }
        }
    }
}
```

In der aktuellen Implementierung ist ein Menü zur Auswahl der Baudrate implementiert.

```
menu_presentation_show(lcd_info, baudrate_menu, baudrate_menu_size,
    baudrate_menu_send_baudrate, button_pressed, false);
```

Für die Anzeige der CAN Nachrichten auf dem Display wurde ein Algorithmus entwickelt, der sicherstellt, dass sich das Display nur aktualisiert wenn nötig 3.7. Mit dieser Implementierung wird ein Flackern des Displays vermieden. Ereignisse, die ein neuladen des Displays auslösen sind, der Empfang einer neuen CAN ID oder eine Änderung des Data Segments einer bereits enthaltenen CAN ID. Außerdem ist es möglich mit den Buttons nach oben und unten zu scrollen, um mehr als vier CAN IDs auf dem vier zeiligen Display anzeigen zu können. Auch hier wird das Display aktualisiert.

REST API

Um die aktuellen Nachrichten von extern leicht abfragen zu können wurde eine REST API zur Verfügung gestellt. Um auf die REST API zuzugreifen, öffnet der ESP32 den WLAN Access Point *CAN-TO-GO*. Außerdem wird ein Webserver gestartet. Der Webserver stellt die Uri */can* zur Verfügung, auf welche über *http://192.168.4.1/can* zugegriffen werden kann.

```
    static httpd_uri_t get_can = {
        .uri = "/can",
```

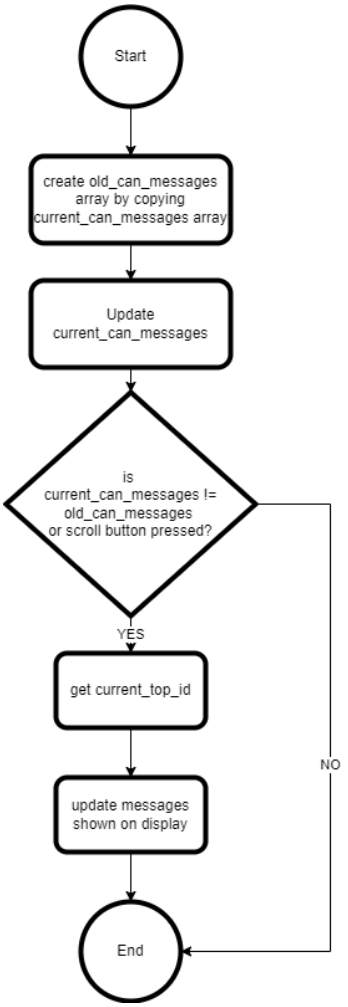

Abbildung 3.7: Algorithmus um CAN Messages anzuzeigen

```
        .method = HTTP_GET,
        .handler = can_get_handler,
        .user_ctx = NULL};
```

Auf dieser Seite werden die aktuellesten 100 CAN Nachrichten im JSON Format zur Verfügung gestellt.

```
{"messages" : [{"micros": 223754230, "id": 291, "ext": 0, "dlc": 8,
   "data": [0,0,1,1,0,0,0,0]},{"micros": 223864208, "id": 2047,
   "ext": 0, "dlc": 8, "data": [0,0,1,1,0,0,0,0]},{"micros":
   223964351, "id": 35, "ext": 0, "dlc": 8, "data":
   [0,0,1,1,0,0,0,0]},...]}
```

3.2 User Interface

Da das Display für den Nutzer nur eine geringe Größe und eine eingeschränkte Funktionalität bietet, wurde ein externes User Interface entwickelt. Um mit einer Code Base das User Interface auf einer Vielzahl von Geräten bereitstellen zu können wurde als Framework *Flutter* gewählt. Flutter ist ein Open-Source-Framework von Google, dass es erlaubt mit nur einer Code Basis das Projekt als Windows Applikation, als Web Applikation und als App auf mobilen Endgeräten zu compilieren. Als Programmiersprache wird *Dart* eingesetzt.

3.2.1 Funktionalitäten

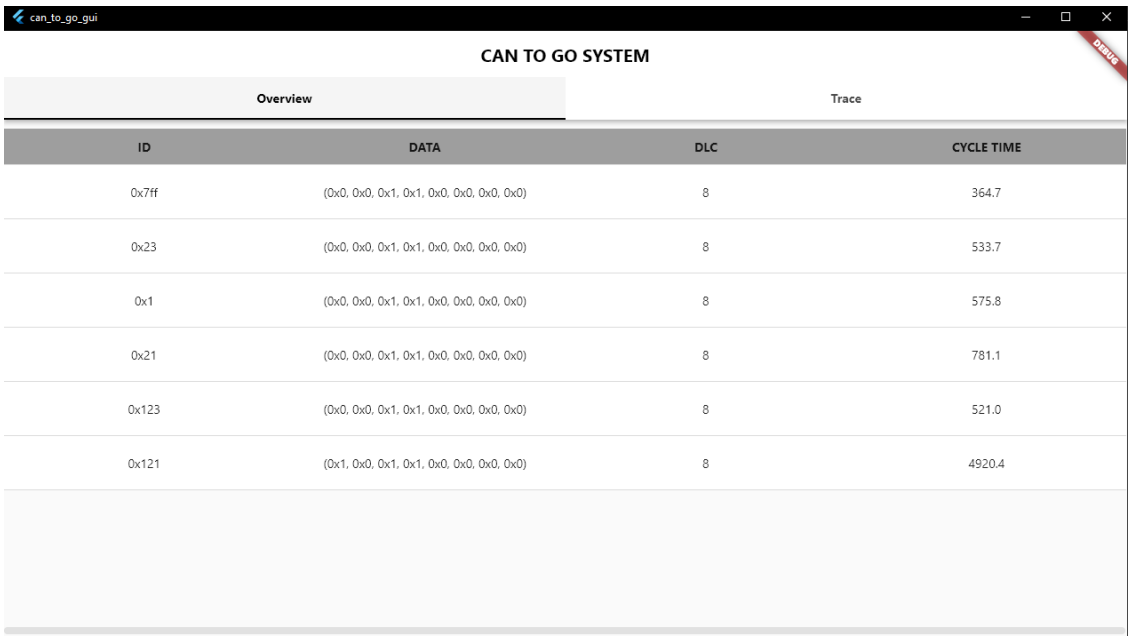

Abbildung 3.8: Overview Tab

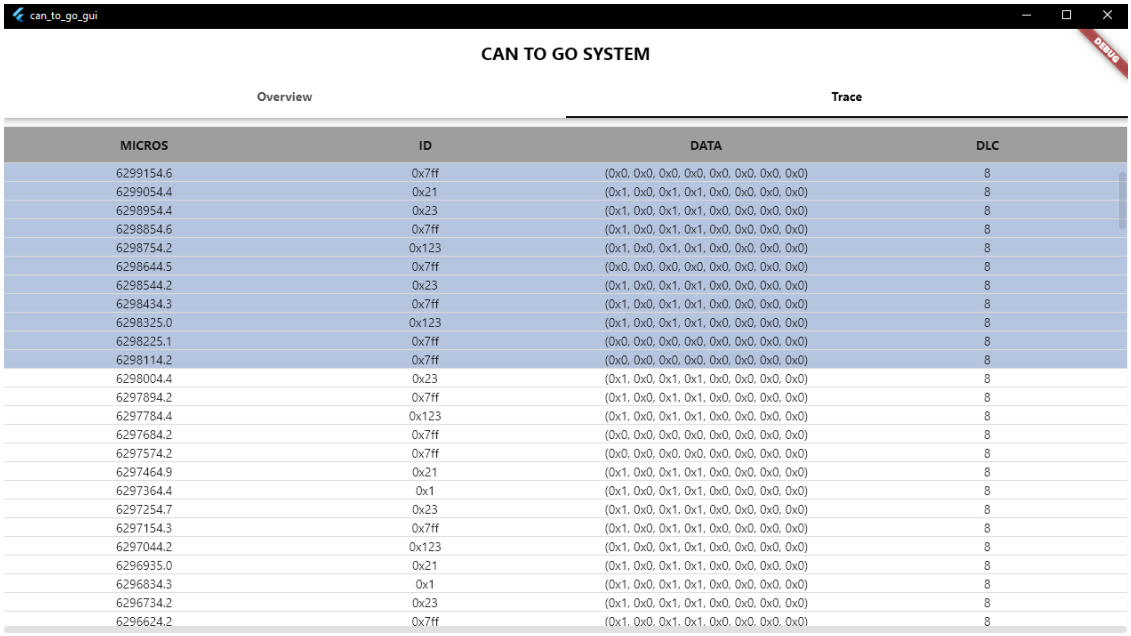

Abbildung 3.9: Trace Tab

Das User Interface besitzt zwei verschiedene Anzeigen für den User. Zum einen gibt es ein *Overview* Tab 3.8, welches alle CAN IDs anzeigt, die empfangen wurden. Das Data Segment repräsentiert dabei immer den aktuellen Stand der jeweiligen ID. Die Cycle Time gibt darüber Auskunft, in welchem Zyklus die jeweilige CAN ID empfangen wird.
Außerdem gibt es die Möglichkeit über den *Trace* Tab 3.9, die letzten angekommen CAN Nachrichten zu sehen. Dabei werden die Nachrichten, die mit dem letzten Batch neu hinzugefügt wurden in einer anderen Farbe markiert, um für den User eine bessere Übersicht zu gewährleisten.

3.2.2 Projektstruktur

Bei der Projektstruktur für die GUI 3.10 wurde sich ebenfalls an der Clean Architecture von Robert C. Martin orientiert [5]. Über die *datasources* werden die Rohdaten von der bereitgestellten REST API gelesen und als eine Liste von *CanMessageModel* zurückgegeben.

```
1    class EspRestApiRemoteDataSource {
2        final String _url = "http://192.168.4.1/can";
3
4        Future<List<CanMessageModel>> getCanMessages() async {
5          final http.Response response = await http.get(Uri.parse(_url));
6
7          List<dynamic> canMessagesJson =
               json.decode(response.body)["messages"];
8
9          return canMessagesJson
10             .map((messageJson) => CanMessageModel.fromJson(messageJson))
11             .toList();
12       }
```

```
13      }
```

Die Repositories sind dafür zuständig die Rohdaten in die Liste des jeweiligen Tabs zu transformieren. Das *CanOverviewRepository* erstellt eine Liste mit den empfangenen CAN IDs zu berechnet die Cylce Time.

```
1     Future<List<CanMessageModel>> getCanOverviewModels(bool refresh)
          async {
2       if (refresh) {
3         canMessagesCache = [];
4         canMessagesUniqueIdCache = [];
5       }
6       final currentCanModelsBatch =
7           (await espRestApiRemoteDataSource.getCanMessages()).reversed;
8
9       for (CanMessageModel canMessage in currentCanModelsBatch) {
10        if (!canMessagesCache.contains(canMessage)) {
11          canMessagesCache.insert(0, canMessage);
12        }
13        if (canMessagesCache.length == 1001) {
14          canMessagesCache.removeLast();
15        }
16      }
17
18      List<CanMessageModel> currentUniqueMessages = [];
19      for (CanMessageModel canMessage in canMessagesCache) {
20        bool doesMessageWithIdExistInList =
21            currentUniqueMessages.any((model) => model.id ==
                  canMessage.id);
22        if (!doesMessageWithIdExistInList) {
23          currentUniqueMessages.add(canMessage);
24        }
25      }
26
27      for (CanMessageModel canMessage in currentUniqueMessages) {
28        int index = canMessagesUniqueIdCache
29            .indexWhere((item) => item.id == canMessage.id);
30
31        if (canMessage.micros != 0) {
32          if (index != -1) {
33            canMessage.cycleTime =
34                getCycleTimeForId(canMessage.id,
                      canMessagesCache.toList());
35            canMessagesUniqueIdCache[index] = canMessage;
36          } else {
37            canMessagesUniqueIdCache.add(canMessage);
38          }
39        }
40      }
41
42      return canMessagesUniqueIdCache;
```

```
43      }
```

Das *CanTraceRepository* erstellt eine Liste mit den letzten 200 empfangen Nachrichten und gibt zurück wie viele Nachrichten neu hinzugefügt wurden.

```
1     Future<CanTraceResult> getCanTraceModels(bool refresh) async {
2         if (refresh) {
3           canMessagesTrace = [];
4         }
5
6         final currentCanModelsBatch =
7             await espRestApiRemoteDataSource.getCanMessages();
8
9         int addedCount = 0;
10
11        for (CanMessageModel canMessage in currentCanModelsBatch) {
12          if (!canMessagesTrace.contains(canMessage)) {
13            canMessagesTrace.insert(0, canMessage);
14            addedCount++;
15          }
16          if (canMessagesTrace.length == 201) {
17            canMessagesTrace.removeLast();
18          }
19        }
20
21        return CanTraceResult(canMessagesTrace, addedCount);
22      }
```

Im *presentation* Layer werden die *pages* und die *widgets* erstellt, die auf den jeweiligen *pages* angezeigt werden. Da sowohl der *Overview* Tab als auch der *Trace* Tab aus einer Tabelle von CAN Nachrichten bestehen, wurde ein Widget *CanTable* definiert, welches über Parameter auf das jeweilige Tab angepasst werden kann.

```
1     class CanTable extends StatelessWidget {
2         final List<String> headers;
3         final List<CanMessageModel> canMessages;
4         final bool isOverviewTab;
5         final double rowHeight;
6         final int? addedCount;
7     ...
8     }
```

Im *Overview* Tab und im *Trace* Tab wird die zuvor definierte Tabelle dann über den *BuildContext* erstellt.

```
1     # Overview Tab
2     Widget build(BuildContext context) {
3         final List<String> headers = [
4           "ID",
5           "DATA",
6           "DLC",
```

```
          "CYCLE TIME",
        ];

        return CanTable(
          headers: headers,
          rowHeight: 60.0,
          isOverviewTab: true,
          canMessages: canMessagesUniqueId,
        );
      }
```

```
    # Trace Tab
    Widget build(BuildContext context) {
        final List<String> headers = [
          "MICROS",
          "ID",
          "DATA",
          "DLC",
        ];

        return CanTable(
          headers: headers,
          rowHeight: 20.0,
          isOverviewTab: false,
          canMessages: canMessages,
          addedCount: addedCount,
        );
      }
```

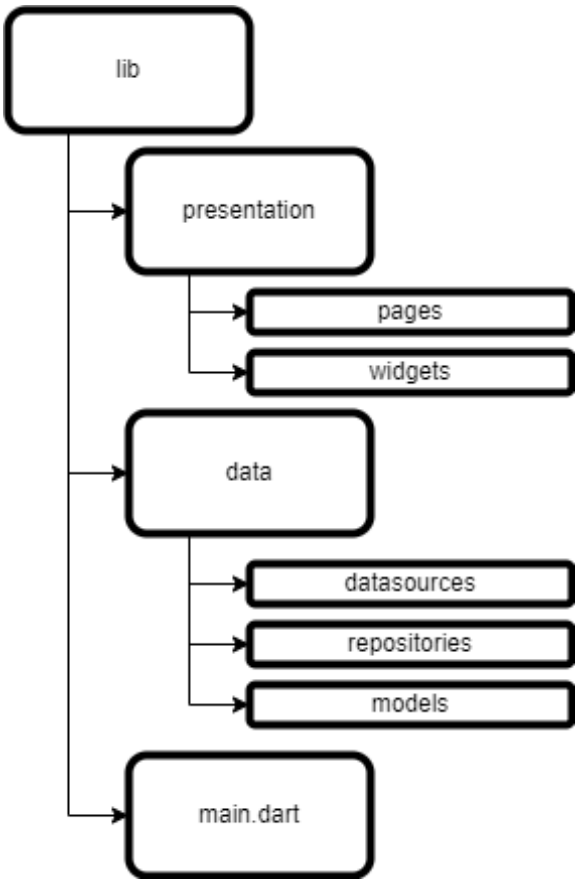

Abbildung 3.10: Projektstruktur GUI

Literatur

[1] amazon. URL: https://www.amazon.de/gp/product/B07TTNBBSC/ref=ppx_yo_dt_b_search_asin_title?ie=UTF8&th=1 (besucht am 30. 12. 2023).

[2] AZ-Delivery. URL: https://www.az-delivery.de/products/esp32-developmentboard (besucht am 27. 12. 2023).

[3] Robert C. Martin. *Solid Relevance*. Okt. 2020. URL: https://blog.cleancoder.com/uncle-bob/2020/10/18/Solid-Relevance.html.

[4] Robert C. Martin. *The Clean Architecture*. Aug. 2012. URL: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html.

[5] Robert C. Martin u. a. *Clean Architecture: a craftsman's guide to software structure and design*. eng. Robert C. Martin series. Boston Columbus Indianapolis New York San Francisco Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto Delhi Mexico City São Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo: Prentice Hall, 2018. ISBN: 978-0-13-449416-6.

[6] Microchip. 2014. URL: https://www.mouser.de/datasheet/2/268/20005167C-1512552.pdf (besucht am 26. 12. 2023).

[7] mikrocontroller.net. URL: https://www.mikrocontroller.net/attachment/6819/canbus.pdf (besucht am 27. 12. 2023).

Abbildungsverzeichnis