Fachhochschule Dortmund

Fachbereich Elektrotechnik

Studiengang Elektrotechnik

Bachelorarbeit

Realisierung einer UART zur CAN Kommunikationsbrücke mit Hilfe eines STM32 ARM Microcontrollers

Realization of a UART to CAN communication bridge using a STM32 ARM microcontroller

von:

Fouzya Samdouni

Betreuer:

Prof. Dr.-Ing.

Michael Karagounis

Zweitprüfer:

Dipl.-Ing.

Rolf Paulus

Dortmund; den 23.10.23

Abstract

This bachelor's thesis focused on the implementation of a UART-to-CAN communication bridge using an STM32L476RG ARM microcontroller. The project aimed to enable communication between two PCs and two microcontrollers via UART and CAN. The work included the physical connection of the components, programming the microcontrollers using STMCubeIDE and CubeMX, and creating a user interface with Qt Creator. The CAN protocol unit in the STM32L476RG microcontroller played a central role in the communication after successful configuration. By monitoring signals on the oscilloscope proper data transmission has been ensured. The thesis presents the configuration of the UART and CAN interfaces and the implementation of communication protocols to exchange messages between the PCs and microcontrollers. The results demonstrate the successful realization of the communication bridge, with reliable data transmission.

This work, therefore, makes a valuable contribution to the development of communication solutions in embedded systems, demonstrating how microcontrollers can effectively realize communication bridges. This project opens avenues for future applications where networking microcontrollers and PCs is necessary.

Abstrakt

Diese Bachelorarbeit beschäftigte sich mit der Realisierung einer UART-zu-CAN-Kommunikationsbrücke mithilfe eines STM32L476RG ARM Mikrocontrollers. Ziel des Projekts war es, die Kommunikation zwischen zwei PCs und zwei Mikrocontrollern über UART und CAN zu ermöglichen. Die Arbeit umfasste die physische Verbindung der Komponenten, die Programmierung der Mikrocontroller mit Hilfe von STMCubeIDE und CubeMX sowie die Erstellung einer Benutzeroberfläche mit Qt Creator. Die CAN-Protokolleinheit im STM32L476RG Mikrocontroller spielte eine zentrale Rolle in diesem Projekt. Sie ermöglichte die CAN-Kommunikation nach erfolgreicher Konfiguration. Durch die Überwachung der Signale über ein Oszilloskop konnte die ordnungsgemäße Datenübertragung festgestellt werden. Die Arbeit präsentiert die Konfiguration der UART- und CAN-Schnittstellen sowie die Implementierung von Kommunikationsprotokollen, um Nachrichten zwischen den PCs und Mikrocontrollern auszutauschen. Die Ergebnisse zeigen, dass die Kommunikationsbrücke erfolgreich realisiert wurde und die Datenübertragung zuverlässig funktioniert.

Diese Arbeit liefert somit einen wertvollen Beitrag zur Entwicklung von Kommunikationslösungen in eingebetteten Systemen und zeigt, wie Mikrocontroller effektiv für die Realisierung von Kommunikationsbrücken eingesetzt werden können. Dieses Projekt eröffnet Möglichkeiten für zukünftige Anwendungen, bei denen die Vernetzung von Mikrocontrollern und PCs erforderlich ist.

Vorwort und Danksagung

Die vorliegende Bachelorarbeit ist das Ergebnis meiner Bemühungen während meines Studiums an der Fachhochschule Dortmund zur Erlangung des akademischen Grades „Bachelor of Engineering“ im Bachelorstudiengang Elektrotechnik (mit der Studienrichtung „Automatisierungs- und Antriebssysteme“).

Zuallererst möchte ich meinen Betreuern, Herrn Prof. Dr. Michael Karagounis und Herrn Dipl.-Ing Rolf Paulus, herzlich danken. Ihre fachliche Kompetenz, ihre Geduld und ihre konstruktive Kritik haben mir geholfen, diese Arbeit auf ein höheres Level zu heben.

Ein großer Dank gebührt auch meiner Familie, insbesondere meiner Mutter und meinem Mann, sowie meinen Freunden. Ihre unerschütterliche Unterstützung und ihre aufmunternden Worte haben mir die nötige Motivation gegeben, dieses Projekt erfolgreich abzuschließen.

Inhaltsverzeichnis

Kapitel 1

Einleitung

In einer zunehmend vernetzten Welt gewinnt die Kommunikation zwischen elektronischen Geräten und Mikrocontrollern eine immer größere Bedeutung. In diesem Kontext spielt die Entwicklung effizienter Kommunikationsbrücken eine entscheidende Rolle, um den reibungslosen Datenaustausch zwischen verschiedenen Plattformen zu ermöglichen.

Die vorliegende Arbeit behandelt die Realisierung einer UART-zu-CAN-Kommunikationsbrücke unter Verwendung des STM32L476RG ARM-Mikrocontrollers. Die zunehmende Komplexität von Kommunikationssystemen erfordert effiziente Methoden zur Integration verschiedener Technologien. Die Kombination von Universal Asynchronous Receiver-Transmitter (UART) und Controller Area Network (CAN) bietet eine robuste Lösung für die Punkt-zu-Punkt-Kommunikation zwischen Mikrocontrollern und PCs in industriellen Anwendungen.

Die UART-Schnittstelle ermöglicht eine einfache serielle Kommunikation, während das CAN-Protokoll eine zuverlässige und störungsresistente Netzwerkkommunikation ermöglicht. Die Implementierung dieser Kommunikationsbrücke erfordert die präzise Programmierung des STM32L476RG Mikrocontrollers, die Konfiguration von UART- und CAN-Schnittstellen sowie die Verwendung externer Transceiver für die physische Schicht des CAN-Bus.

Diese Arbeit beleuchtet die Hardware- und Softwarekomponenten, die Programmierung des Mikrocontrollers, den verwendeten Versuchsaufbau und die Analyse der erzielten Ergebnisse. Die gewonnenen Erkenntnisse bieten nicht nur Einblicke in die praktische Umsetzung von Kommunikationsbrücken, sondern tragen auch zur Weiterentwicklung von Lösungen für komplexe industrielle Kommunikationsanforderungen bei.

Kapitel 2

Beschreibung der Hardware

Im Rahmen dieser Arbeit soll ein ARM-Mikrocontroller über die serielle Schnittstelle eines PCs angesteuert werden. Prozessoren mit ARM-Architektur sind kostengünstig und zeichnen sich durch eine geringe Leistungsaufnahme aus. Sie eignen sich daher besonders für den Einsatz in batteriebetriebenen Geräten und eingebetteten Systemen. Sie werden häufig für mobile Endgeräte – vor allem Smartphones und Tabletcomputer – verwendet. Andere Anwendungsbeispiele sind Geräte im Internet der Dinge (IoT) oder Spielkonsolen. Konkret wird ein Arm-Prozessor aus der Familie STM32 benutzt. Im Folgenden wird ein Überblick auf die Prozessorfamilie und die zugehörigen Mikrocontroller-Boards gegeben und anschließend das verwendete System im Detail erläutert.

2.1 Prozessorfamilie STM32

Bei der Prozessorfamilie STM32, welche von der Firma STMicroelectronics angeboten wird, handelt es sich um 32-Bit-Prozessoren. Sie basieren auf Prozessorkernen des Typs ARM Cortex® M, welche für niedrigen Energieverbrauch optimiert sind. Die Mikrocontroller enthalten neben Prozessorkern(en) ein statisches RAM (SRAM), einen Flash-Speicher, eine Debugging-Schnittstelle sowie diverse Peripheriegeräte.[1]

Abbildung 2.1 gibt eine Übersicht über die Nucleo-Boards, welche den STM32-Prozessor nutzen. Dies werden in drei Größen angeboten, die nach der Anzahl der Pins wie folgt bezeichnet werden:

- Nucleo-32
- Nucleo-64
- Nucleo-144

Diese Boards werden mit unterschiedlichen Mikrocontrollern angeboten. Konzeptionell gliedern sich die Produktlinien in vier Produktserien:

- Mainstream-Serie (dunkelblau)
- Ultra-Low-Power-Serie (gelb)
- High-Performance-Serie (türkisgrün)
- Wireless-Serie (hellgrau)

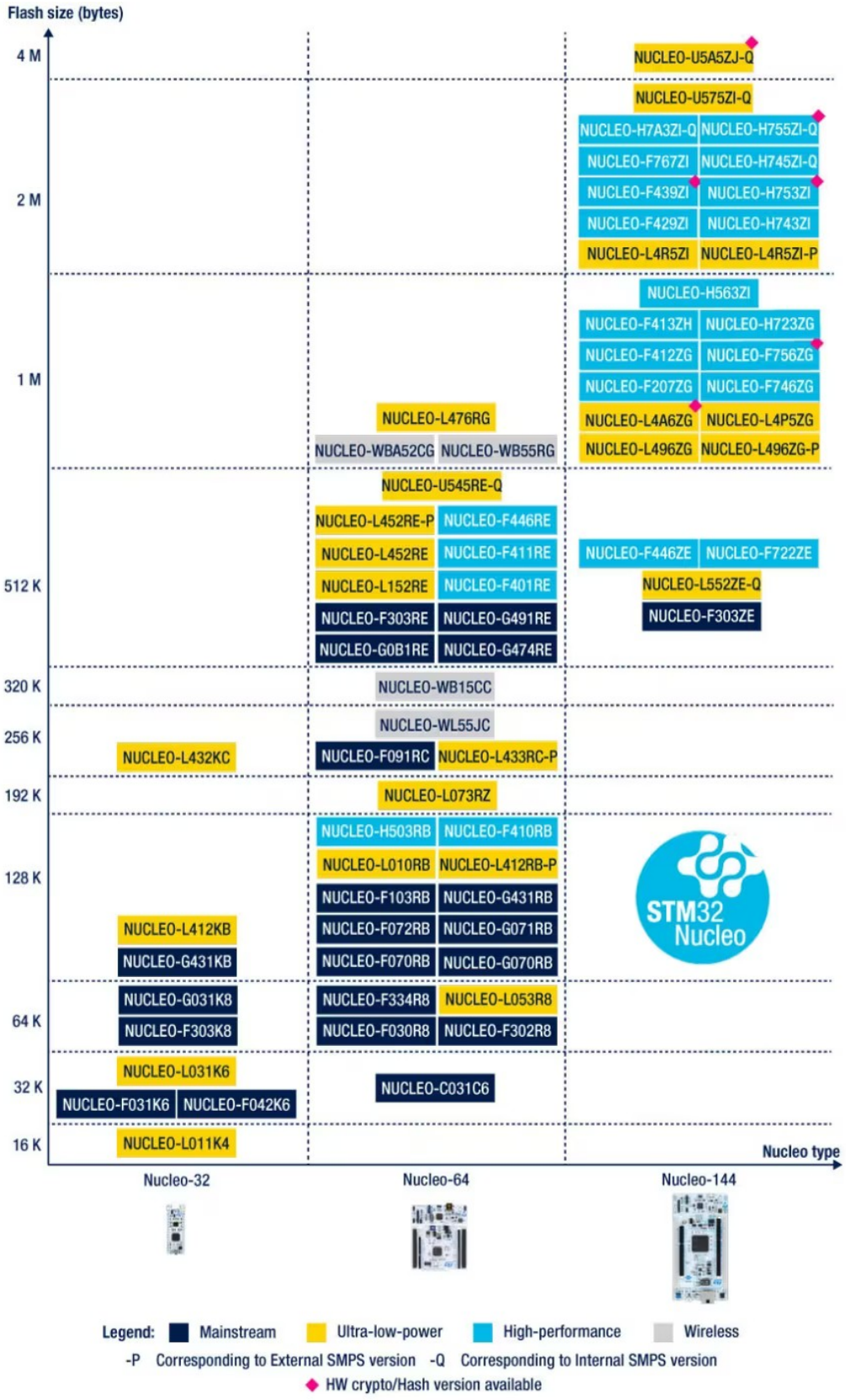


Abbildung 2.1: Übersicht angebotener Nucleo-STM32-Boards [2]

Die *Mainstream*-Serie bietet eine skalierbare Reihe von Allzweck-Mikrocontrollern für vielfältige Anwendungen. Der Fokus bei der Entwicklung dieser Serie richtet sich auf kostengünstige, robuste und langlebige Systeme. Diese Serie umfasst die folgenden Produktlinien: [3]

- CO: ARM® Cortex® M0+ (48 MHz)
- FO: ARM® Cortex® M0 (48 MHz)
- GO: ARM® Cortex® M0+ (64 MHz)
- F1: ARM® Cortex® M3 (72 MHz)
- F3: ARM® Cortex® M4 (72 MHz)
- G4: ARM® Cortex® M4 (170 MHz)

Die Ultra-Low-Power-Serie bietet Systeme, deren Design auf eine besonders niedrige Leistungsaufnahme hin optimiert ist. Dazu gehören die Produktlinien:[4]

- LO: ARM® Cortex® M0+ (32 MHz)
- L4: ARM® Cortex® M4 (80 MHz)
- L4+: ARM® Cortex® M4 (120 MHz)
- L5: ARM® Cortex® M33 (110 MHz)
- U5: ARM® Cortex® M33 (160 MHz)

Die Systeme der *High-Performance*-Serie sind für eine hohe Leistung bei der Codeausführung, Datenübertragung und Datenverarbeitung konzipiert. Die Systeme verfügen über besonders schnelle Prozessoren und große Speicher. Dazu gehören die Produktlinien:[5]

- F2: ARM® Cortex® M3 (120 MHz)
- F4: ARM® Cortex® M4 (180 MHz)
- H5: ARM® Cortex® M33 (250 MHz)
- F7: ARM® Cortex® M7 (214 MHz)
- H7: ARM® Cortex® M7 (550 MHz)

Die *Wireless-Serie*, mit der Kennung W umfasst Systemen mit integrierter Antenne, welche entweder Bluetooth bzw. Bluetooth Low Energy (WB bzw. WBA) oder LoRa® unterstützen. Sie nutzen eine Dual-Core Konzept mit einem ARM® Cortex® M4 als Hauptprozessor und einem ARM® Cortex® M0+ für den Funkbetrieb. Ein Ausnahme bildet die Produktlinie WBA welche sich auf einen ARM® Cortex® M33 stützt [6]–[9].

Für die vorliegende Arbeit wird ein Board aus der Ultra-Low-Power-Serie L4: ARM® Cortex® M4 (80 MHz) verwendet. Dieses System wird im folgenden Abschnitt beschrieben.

2.2 Cortex-Familie

Die Wahl des richtigen Mikrocontrollers für die Entwicklung effizienter Programme und Projekte hängt von vielen Faktoren ab.

Darunter sind unter Anderem zu nennen Kosten, Geschwindigkeit, Leistungsaufnahme, Größe, Anzahl der digitalen und analogen Ein- und Ausgangsports, Programm- und Datenspeichergrößen, Interrupt-Unterstützung, Spezielle Busunterstützung (z.B. USB, CAN), Erleichterung der Systementwicklung (z.B. Programmierung) usw.

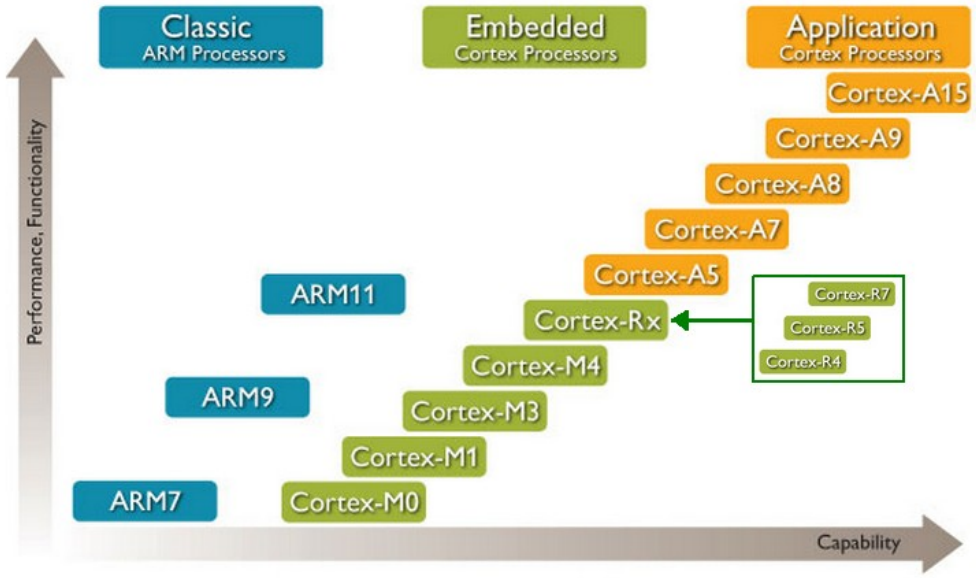


Abbildung 2.2: Überischt auf ARM-Prozessoren[10]

Die Cortex-Familie besteht aus 3 Familien Abbildung 2.2:

• **Cortex-A** sind die leistungsfähigsten ARM-Prozessoren, die für den Einsatz in mobilen Anwendungen wie Mobiltelefonen, Tabletts und GPS-Geräten entwickelt wurden. Diese Prozessoren unterstützen erweiterte Funktionen für Betriebssysteme wie Android, iOS, Linux und Windows. Darüber hinaus wird die erweiterte Speicherverwaltung durch einen virtuellen Speicher unterstützt.

• **Cortex-R-Serie** ist ein Echtzeit-Prozessor mit höherer Leistung als der Cortex-M.
Davon sind einige Mitglieder mit hohen Taktraten von mehr als 1 GHz ausgelegt.
Diese Prozessoren werden häufig in Festplattencontrollern, Netzwerkgeräten und Automobilanwendungen eingesetzt.

• **Cortex-M**: Diese Prozessoren sind speziell für den Mikrocontrollermarkt entwickelt worden. Sie zeichnen sich durch geringen Stromverbrauch, niedrige Kosten und einfache Bedienung aus.

2.3 Nucleo-Board STM32L476RG

In diesem Projekt wird der Mikrocontroller STM32L476RG verwendet. Er basiert auf der Cortex-M4-Architektur, welche Prozessoren mittlerer Leistung adressiert, die in Mikrocontroller-Anwendungen eingesetzt werden. Der verwendete Cortex-M4 Kern unterstützt auch DSP und Gleitkommaarithmetik durch spezielle Befehlserweiterungen.

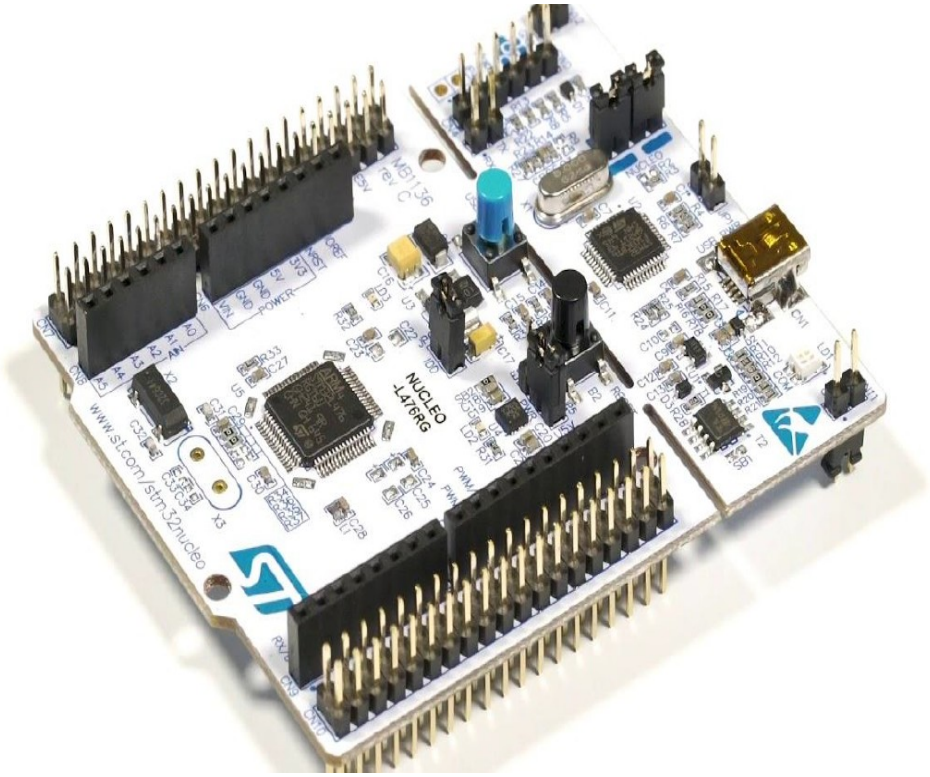


Abbildung 2.3: Nucleo-Board STM32L476RG

2.3.1 Eigenschaften

Das Nucleo Board STM32L476RG ist ein Entwicklungsboard von STMicroelectronics. Das Board kann mit einer Taktrate von bis zu 80 MHz betrieben werden und ist eine ausgezeichnete Wahl für die Entwicklung von Anwendungen im Bereich industrieller Automatisierung, Medizintechnik, Robotik, Sensornetzwerke und viele andere Bereiche. Das Board ist einfach zu verwenden und bietet eine leistungsstarke Plattform für die schnelle Entwicklung von Embedded-Systemen. Es ist auch ein großartiges Board für Studenten und Hobbyisten, um ihre Programmier- und Elektronikkenntnisse zu erweitern.

2.3.2 Zweiteilige Platine

Das Board hat die Maße 70mm x 82,5mm. Wie in Abbildung 2.4 dargestellt, besteht die Platine aus zwei Teilen und zwar dem kleineren ST-LINK-Teil mit einem Mini USB-Anschluss zum Anschluss an einen PC und dem MCU-Teil.

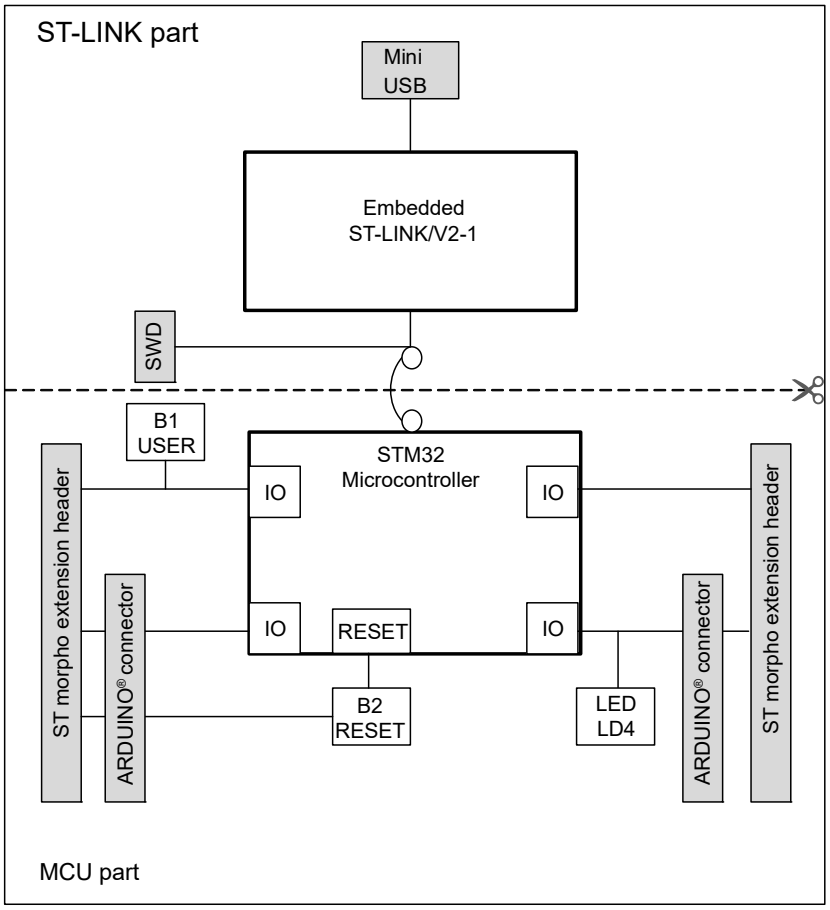


Abbildung 2.4: Blockschaltbild des STM32 Nucleo-64 Board [11]

2.3.3 Bauteile

Die Bauteile auf der Oberseite der Platine sind in Abbildung 2.5 dargestellt.

Einige der wichtigsten Komponenten auf dem Board sind:

• **CN1**: Mini USB-Buchse (5V)

• **CN2**: ST-LINK/Nucleo Auswahlschalter

• **B1**: Benutzer-Taster der mit dem STM32 Ein-/Ausgabe Pin PC_13 (Pin 23 am Stecker CN7) verbunden ist.

• **B2**: Reset-Taster, mit der die MCU zurückgesetzt wird.

• **LD1**: rot/grüne Kommunikations-LED. Die Standardfarbe dieser LED ist rot und wird grün, wenn eine Kommunikation zwischen dem PC und dem ST-LINK stattfindet.

• **LD2**: grüne Benutzer-LED die mit Pin 11 des Steckverbinders CN10(STM32 Ein-/Ausgabe Pin PA_5) auf der Nucleo-L476RG Platine verbunden ist.

• **LD3**: rote Power-LED, die anzeigt, dass +5V Strom auf dem MCU-Board verfügbar ist.

• **Arduino-Steckverbinder**

• **ST Morpho Steckverbinder**

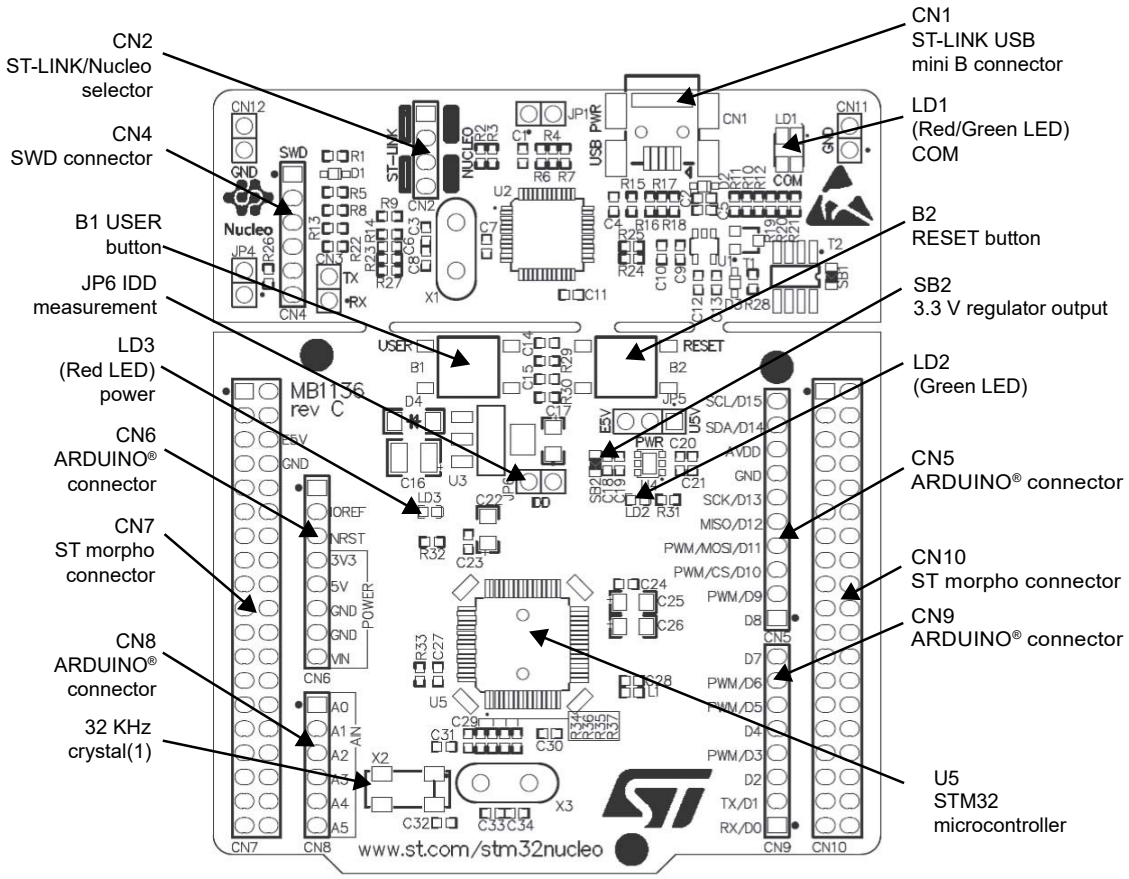


Abbildung 2.5: Layout der Oberseite des STM32 Nucleo-64 Board [11]

Das Board verfügt über eine Vielzahl von Funktionen, einschließlich eines ARM Cortex-M4 Prozessors mit 1 MB Flash-Speicher und 128 KB RAM, zahlreichen Peripheriegeräten wie GPIOs, Timer, UART, SPI und I2C Schnittstellen, einem integrierten ST-Link Debugger/Programmierer, sowie einem USB-Anschluss und einer Stromversorgung.

2.3.4 Blockdiagramm des Prozessors STM32L476xx

Abbildung 2.6 zeigt das Blockdiagramm des Prozessors STM32L476xx. Die 80MHz ARM Cortex-M4 CPU ist in der Mitte oben in der Abbildung dargestellt. Die digitalen Schnittstellen befinden sich oben links, die GPIOs unten links, die LCD-Schnittstelle oben rechts, Timer in der Mitte rechts, analoge Schnittstellen und parallele Schnittstellen unten rechts, und die DMA-Kanäle und der Speicher sind im mittleren Teil der Abbildung dargestellt.

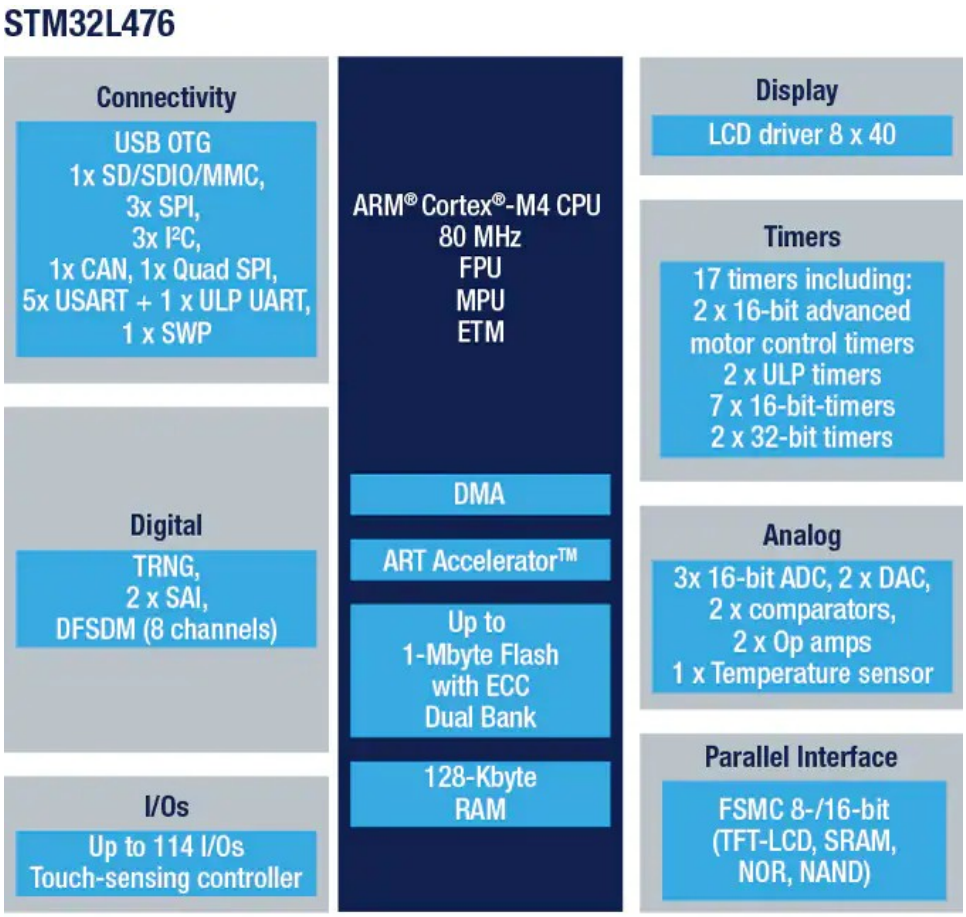


Abbildung 2.6: Blockdiagramm des STM32L476xx Prozessors [12]

2.3.5 Pinbelegung

Abbildung 2.7 zeigt die Pinbelegung des STM32L476RX. Die Pinbelegung ist wie folgt (einige Pins werden mit anderen Funktionen geteilt):

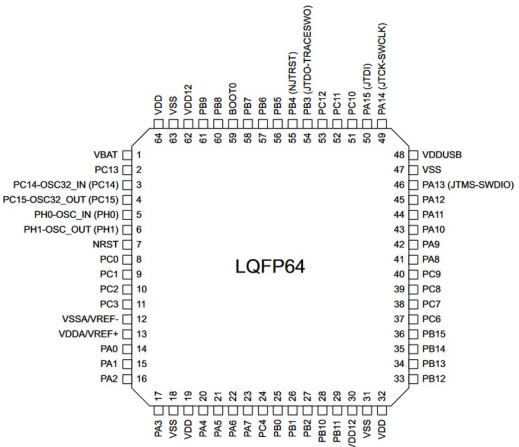


Abbildung 2.7: Pinbelegung des STM32L476RX [12]

- PA0 - PA15 - GPIO Port A Pins
- PB0 - PB15 - GPIO Port B Pins
- PC0 - PC15 - GPIO Port C Pins
- PD2 - GPIO Port D Pin
- PH0 - PH1 - GPIO Port H Pins
- VDD, VSS - Strom- und Erdungspins
- VSSA, VDDA - Referenzspannungspins
- NRST - Reset-Pin
- VBAT - Externer Batterieanschluss
- BOOT0 - Boot0 Pin
- VDD12 - Externer SMPS-Pin für die Stromversorgung

2.3.6 Allzweck-Ein- und Ausgänge (GPIOs)

Der Mikrocontroller STM32L476Rx hat 64 Pins und 51 davon können als Allzweck-Ein- oder Ausgänge (GPIOs) verwendet werden.

Die GPIOs sind in 5 Ports angeordnet, wobei die Ports A, B und C 16 Bit lang sind, Port H 2 Bit lang ist und Port D nur 1 Bit lang ist.[13]

Das Board ist auch mit der STM32CubeIDE Software von STMicroelectronics kompatibel, die eine benutzerfreundliche grafische Schnittstelle bietet, um schnell und einfach Code zu generieren und das Board zu konfigurieren.

2.4 CAN-Protokolleinheit im STM32L476RG Mikrocontroller

Die CAN-Protokolleinheit im STM32L476RG Mikrocontroller ist eine spezialisierte Hardwarekomponente, die entwickelt wurde, um das Controller Area Network (CAN) Protokoll in diesem Mikrocontroller zu unterstützen. Dieser Mikrocontroller gehört zur STM32-Familie von STMicroelectronics und ist für anspruchsvolle Anwendungen in den Bereichen Energieeffizienz und Echtzeitverarbeitung konzipiert.

Die CAN-Protokolleinheit im STM32L476RG bietet eine robuste und zuverlässige CAN-Kommunikation. Sie ermöglicht die einfache Integration in CAN-basierte Netzwerke und bietet Funktionen zur Übertragung von Nachrichten in Echtzeit. Diese Einheit unterstützt sowohl den Standard-CAN-Modus (CAN 2.0A/B) als auch den erweiterten CAN-Modus (CAN FD), was eine breite Palette von Anwendungsfällen abdeckt.

Zu den Funktionen der CAN-Protokolleinheit gehören Hardware-Filter, welche die Nachrichtenfilterung erleichtern, sowie leistungsstarke Fehlererkennungs- und Fehlerkorrekturmechanismen, um die Datenintegrität sicherzustellen. Darüber hinaus ermöglicht sie die Verwendung von bis zu drei unabhängigen CAN-Bussen, was die Flexibilität und Skalierbarkeit erhöht.

Die CAN-Protokolleinheit im Mikrocontroller spielt eine entscheidende Rolle für Anwendungen, die eine zuverlässige und effiziente Kommunikation in CAN-Netzwerken erfordern. Diese Anwendungen finden sich beispielsweise in der Automobilindustrie, der industriellen Automatisierung und anderen anspruchsvollen Branchen. Die Einheit trägt dazu bei, die Implementierung von CAN-basierten Lösungen zu vereinfachen und die Leistungsfähigkeit dieses Mikrocontrollers in diesen Anwendungen zu maximieren.

2.5 CAN-Transceiver Platine

Die Transceiver Bausteine befinden sich auf einer zusätzlichen Platine, welche in Abbildung 2.8 zu sehen ist und im Rahmen einer anderen Bachelorarbeit entwickelt wurde [14]. Diese Platine wird über die Arduino kompatiblen Stecker auf das Nucleo Boards aufgebracht. Dadurch wird die Aufsteckplatine durch das Nucleo-Board mit Spannung versorgt und auch die Verbindung zur CAN Protokolleinheit des Mikrocontrollers etabliert. Der Anschluss an den CAN Bus erfolgt über D-SUB Stecker, welcher ebenfalls auf dieser zusätzlichen Platine vorhanden sind.

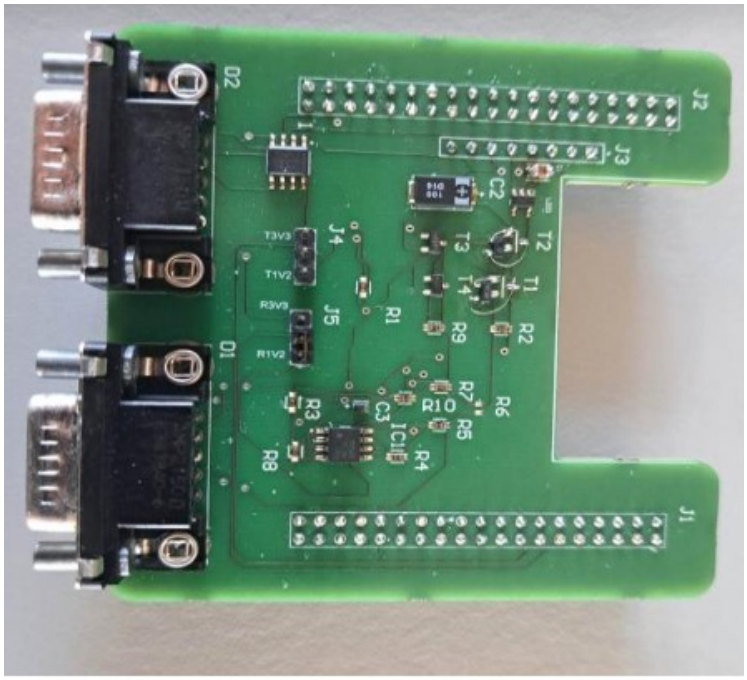


Abbildung 2.8: CAN-Transceiver [14]

Kapitel 3

Serielle Kommunikation zwischen Mikrocontroller und PC über UART

3.1 Einführung in die UART-Schnittstelle

UART (Universal Asynchronous Receiver/Transmitter) ist ein asynchrones serielles Übertragungsprotokoll, das Daten zwischen zwei Geräten über eine einzige Verbindung überträgt. Es wird häufig in einfachen Systemen wie Mikrocontrollern, Sensoren, GPS-Modulen und anderen ähnlichen Geräten verwendet.

Es ist auch das am weitesten verbreitete serielle Protokoll für die serielle Vollduplex-Kommunikation. UART funktioniert so, dass ausgehende Daten in einen seriellen Binärstrom konvertiert werden und empfangene Daten mit Hilfe einer Seriell-Parallel-Wandlung in die parallele Form umgewandelt werden. Diese Daten werden mit einem fixen Rahmen und einer definierten Baudrate übertragen.

Für die Kommunikation im Rahmen der UART-Schnittstelle werden zwei Datenleitungen benutzt:

- Ein Tx-Anschluss (oder auch TxD für Transmit Data), der für das Senden von Daten dient und als Ausgang-Pin bezeichnet wird.
- Ein Rx-Anschluss (oder auch RxD für Receive Data), der für das Empfangen von Daten dient und als Eingang-Pin bezeichnet wird.

Wichtig ist dabei aber ein gemeinsamer Massenanschluss, da es sonst im schlimmsten Fall zur Beschädigung des Mikrocontrollers führen kann, wenn die beiden Kommunikationspartnern auf unterschiedlichen Masse-Potenzialen liegen.

Bei der Verbindung der Anschlüsse der beiden Kommunikationspartnern handelt es sich um eine Kreuzverbindung, in dem der Tx-Ausgangs des ersten Systems mit dem Rx-Eingang des zweiten angeschlossen wird. Dadurch entsteht ein bidirektionaler Bus, der eine Vollduplex-Kommunikation zulässt, wobei die Teilnehmer gleichzeitig Daten senden und empfangen können (siehe Abbildung 3.1).

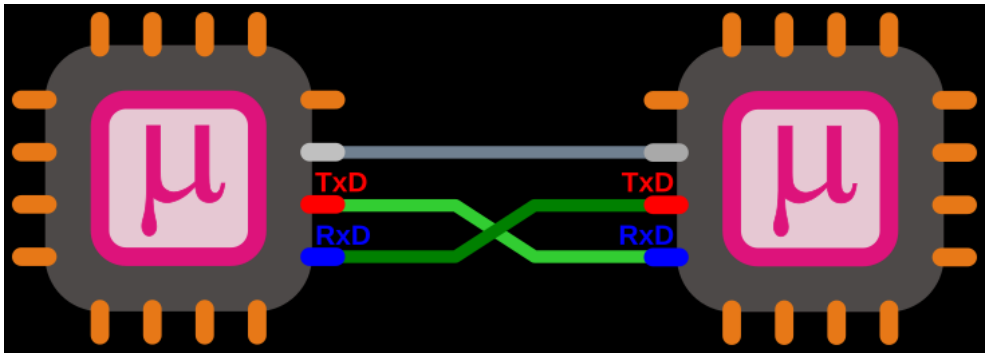


Abbildung 3.1: Aufbau der UART Schnittstelle [15]

Bei der UART-Datenübertragung existiert kein Taktsignal, mit dem sich Sender und Empfänger synchronisieren können. Aus diesem Grund muss die Übertragungsgeschwindigkeit oder Baudrate beim Sender sowie beim Empfänger identisch sein. Die Baudrate darf maximal um 10% von den beiden Geräten abweichen.

Der UART-Rahmen (UART-Frame) besteht aus einem Start-Bit, 5 bis 9 Datenbits, einem optionalen Paritätsbit und einem oder zwei Stop-Bits (Siehe Abbildung 3.2).

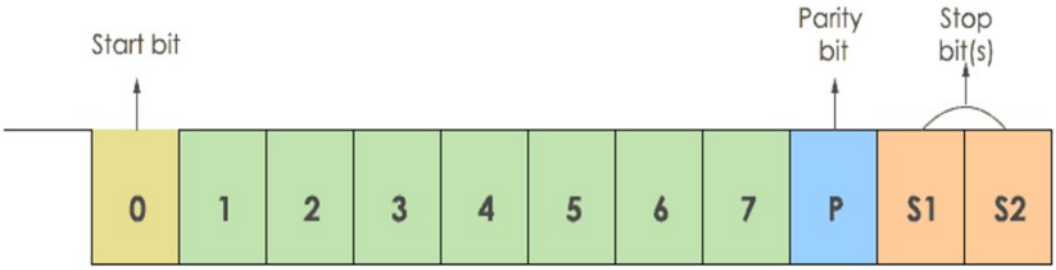


Abbildung 3.2: Beispiel eines UART-Frames [16]

- **Das Startbit:**
  Damit sich die Kommunikationspartnern synchronisieren können, wird das Startbit im UART-Protokoll benötigt. Dieses Bit wird auf logisch „0“ gesetzt und signalisiert den Start eines UART- Frames.
- **Datenbits:**
- Datenfelder sind 5 bis 8 Datenbits lang, und folgen direkt dem Startbit. Wenn kein Paritätsbit verwendet wird, kann das Datenfeld auch 9 Bit lang sein. Die Datenbits werden im „Little-Endian“ Format übertragen. Im „Little-Endian“ Format beginnt die Versendung mit dem „Least Significant Bit“ als Erstes und das „Most Significant Bits“als Letztes.

- **Paritätsbit**
  Das Paritätsbit ist ein optionales Bit und dient zur Fehlererkennung. Es wird zwischen drei Paritätsarten unterschieden:

  ➢ Odd-Parity(O):
  Wenn die Anzahl der logisch 1-Bits ungerade ist, wird das Paritätsbit auf eine logisch „0“ gesetzt.
  ➢ Even-Parity(E):
  Wenn die Anzahl der logisch 1-Bits gerade ist, wird das Paritätsbit auf eine logisch „0“ gesetzt .
  ➢ No Parity(N):
  Das Paritätsbit wird nicht verwendet.

- **Stop-Bits:**
  Durch ein oder zwei Stop-Bits wird das Ende des UART-Frames signalisiert. Stop-Bits haben den Zustand logisch „1“.

In dieser Arbeit wird UART als Peripheriegerät im „STM32L476RG“ Mikrocontroller genutzt, um eine serielle Datenübertragung zwischen einem PC und dem Mikrocontroller über eine USB-Verbindung zu etablieren. Der UART Datenstrom wird dabei automatisiert über einen externen Baustein in einen USB Datenstrom gewandelt und an den PC weitergeleitet.

3.2 Schritte der UART-Übertragung

1. Der sendende UART empfängt Daten parallel vom Datenbus.

2. Der sendende UART fügt das Startbit, das Paritätsbit und das/die Stopbit(s) zum Datenrahmen hinzu.

3. Das gesamte Paket wird seriell vom sendenden UART an den empfangenden UART gesendet. Der empfangende UART tastet die Datenleitung mit der vorkonfigurierten Baudrate ab.

4. Der empfangende UART verwirft das Startbit, das Paritätsbit und das Stopbit aus dem Datenrahmen.

5. Der empfangende UART wandelt die seriellen Daten wieder in parallele Daten um und überträgt sie an den Datenbus auf der Empfangsseite.

3.3 Software-Methoden für die UART-Kommunikation

Für die softwareseitige Verarbeitung des Datenaustauschs über UART bieten STM32 Mikrocontrollern drei Varianten an:

- Polling-Mode
- Interrupt-Mode
- DMA-Mode

3.3.1 Polling-Mode

Die Polling-Methode ist eine Funktion, die von der Hauptroutine aufgerufen wird und die CPU blockiert, so dass sie mit der Ausführung der Hauptaufgabe nicht fortfahren kann, bis eine bestimmte Anzahl von UART-Datenbytes empfangen werden. Nach dem Empfang der erforderlichen Datenmenge wird die Funktion beendet und die CPU setzt die Ausführung des Hauptcodes fort. Andernfalls, wenn z.B. die UART-Peripherie aus irgendeinem Grund nicht die erwartete Datenmenge empfangen hat, blockiert die Funktion die CPU für eine bestimmte Zeitspanne "Time-out". Nach dieser Zeitspanne wird die Funktion beendet und die Kontrolle an die CPU zurückgegeben, um den Hauptcode fortzusetzen.

```
HAL_StatusTypeDef HAL_UART_Receive(UART_HandleTypeDef *huart, uint8_t *pData,
                                   uint16_t Size, uint32_t Timeout);
```

Quellcode 3.1: Beispiel zur Empfangsfunktion in Polling-Mode

Im Quellcode 3.1 ist ein Beispiel einer UART-Funktion zu sehen, die für den Daten Empfang im Polling-Mode geeignet ist.

Dabei werden im Empfangspuffer „pData“ eine bestimmte Anzahl „Size“ von Bytes erwartet. Wenn der Empfang erfolgt ist, kann die CPU die Hauptcodeausführung wieder aufnehmen. Wenn diese Datenmenge nicht empfangen wird, wird die CPU für eine bestimmte Zeit „Timeout“ blockiert, bis diese Funktion in den Hauptkontext zurückkehrt. Polling auf Grund der Blockade der CPU kein effizienter Weg, um serielle Daten zu empfangen, auch wenn die Methode prinzipiell funktioniert.

3.3.2 Interrupt-Mode

Die Verwendung von Interrupt-Signalen ist ein effizienterer Weg, um den UART-Datenempfang abzuwickeln. Die CPU initialisiert die UART-Empfangshardware so, dass sie ein Interrupt-Signal auslöst, sobald ein neues Datenbyte empfangen wird. Im ISR-Code (Interrupt Service Routine) werden die empfangenen Daten in einem Puffer zur weiteren Verarbeitung gespeichert. Dabei ist die ISR, welche auch Interrupt-Handler genannt wird, ein Softwareprozess, welcher durch eine Unterbrechungsanforderung von einem Hardwaregerät aufgerufen wird. Er bearbeitet die Anforderung und sendet sie an die CPU, wodurch der aktive Prozess unterbrochen wird. Wenn die ISR abgeschlossen ist, wird der unterbrochene Prozess wieder aufgenommen.

Diese Art der Handhabung des UART-Datenempfangs ist eine nicht-blockierende Methode. Die CPU leitet den Empfangsprozess ein und fährt mit der Ausführung des Hauptcodes fort. Wenn ein Interrupt empfangen wird, friert sie den Hauptkontext ein und schaltet zum ISR-Handler, um das empfangene Byte in einem Puffer zu speichern, und schaltet zurück zum Hauptkontext und setzt den Hauptcode fort.

```
HAL_StatusTypeDef HAL_UART_Receive_IT(UART_HandleTypeDef *huart,
                                      uint8_t *pData, uint6_t Size)
```

Quellcode 3.2: Beispiel zur Empfangsfunktion in Interrupt-Mode

Im Quellcode 3.2 ist ein Beispiel zu einer UART-Funktion zu sehen, die für den Daten-Empfang im Interrupt-Mode geeignet ist. Diese Funktion initialisiert den UART-Empfangsprozess im Interrupt-Mode, und nach Abschluss wird eine Callback-Funktion aufgerufen, um der Anwendung und dem Prozessor mitzuteilen, dass der empfangene Datenpuffer jetzt bereit ist.

3.3.3 DMA-Mode

Der DMA-Mode steht für „Direct Memory Access“ und wird verwendet, um die empfangenen seriellen UART-Daten vom UART-Peripheriegerät direkt in den Speicher zu leiten. Diese Methode gilt als die effizienteste Art, eine solche Aufgabe zu erledigen, da sie keinerlei Eingreifen der CPU erfordert. Diese Funktion muss nur eingerichtet und im Hauptanwendungscode ausgeführt werden. Der DMA Controller benachrichtigt die CPU nach Abschluss des Transfers der Daten in den Datenpuffer, der sich an der vorprogrammierten Stelle im Hauptspeicher befindet.

Kapitel 4

Beschreibung der Software

In diesen Kapiteln werden die Software-Entwicklungswerkzeuge STM32CubeIDE und STM32CubeMX beschrieben, die für die Programmierung des Mikrocontrollers auf dem Nucleo-Board verwendet wurden. Anschließend wird die für die Erstellung einer grafischen Benutzeroberfläche verwendete Programmierumgebung Qt Creator vorgestellt.

4.1 Nucleo Software Entwicklungswerkzeuge

In diesem Kapitel geht es um Softwareentwicklungswerkzeuge, welche auch Integrierte Entwicklungsumgebung, IDE oder Toolchain genannt werden, mit denen Software für die ARM Prozessoren auf den Nucleo-Boards erstellt wird.

Es stehen mehrere Softwareentwicklungswerkzeuge in Form von integrierten Entwicklungsumgebungen (IDEs) für die Programmierung zur Verfügung. Im Allgemeinen bieten die meisten IDEs einen integrierten Texteditor, Compiler, Debugger und ein Programm-Upload-Tool, mit dem der entwickelte Programmcode in den Programmspeicher des jeweiligen Mikrocontrollers hochgeladen werden kann. Der Compiler ist in der Regel auf die Verarbeitung von C/C++ Programmen ausgelegt.

In diesem Projekt wird die gesamte Toolchain im Ordner /STM32Toolchain installiert.

STM32CubeIDE und STM32CubeMX können kostenlos von den folgenden Website heruntergeladen werden:

- https://www.st.com/en/development-tools/stm32cubeide.html
- https://www.st.com/en/development-tools/stm32cubemx.html

4.1.1 STM32CubeIDE

STM32CubeIDE wurde von STMicroelectronics entwickelt und ist eine integrierte Entwicklungsumgebung für die Programmierung von STM32-Mikrocontrollern, die plattformübergreifend auf Windows, Linux und macOS verwendet werden kann.

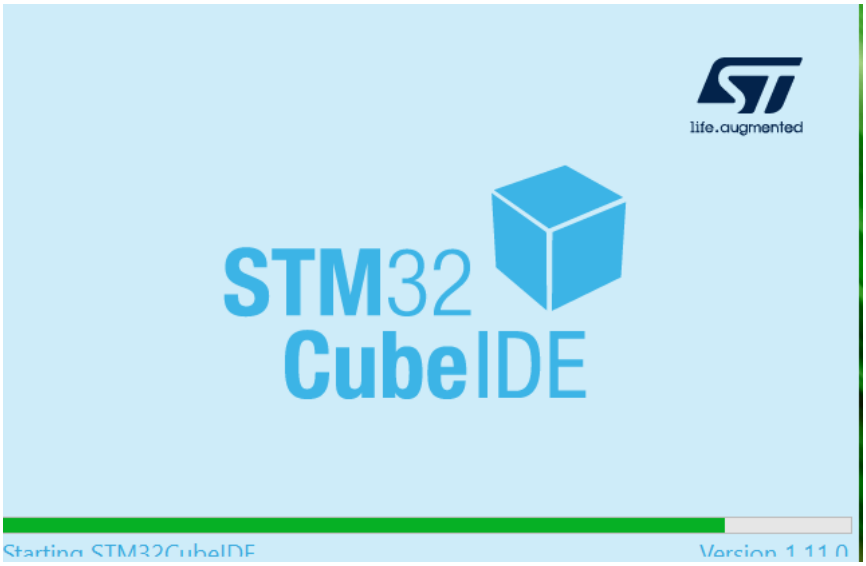


Abbildung 4.1: STM32CubeIDE

STM32CubeIDE enthält eine Reihe von Werkzeugen, die für die Programmierung von STM32-Mikrocontrollern benötigt werden, wie z. B. Compiler, Debugger und Code-Generatoren. Die Entwicklungsumgebung bietet auch eine grafische Benutzeroberfläche (GUI) für die Konfiguration von Peripheriegeräten und für das Generieren von Code aus einer grafischen Darstellung.

STM32CubeIDE basiert auf der Open-Source-IDE Eclipse und stellt eine Vielzahl von Funktionen zur Verfügung, welche die Programmierung von STM32-Mikrocontrollern erleichtern, wie z.B. Code-Generierung und Debugging. STM32CubeIDE ist auch mit anderen STMicroelectronics-Tools und Bibliotheken kompatibel wie z.B. STM32CubeMX einer Software für die Konfiguration von STM32-Mikrocontrollern.

4.1.2 STM32CubeMX

CubeMX ist eine grafische Konfigurationssoftware von STMicroelectronics, die bei der Entwicklung von Embedded-Systemen eingesetzt wird. Die Software bietet eine intuitive Benutzeroberfläche, die es Entwicklern ermöglicht, schnell und einfach die Konfiguration für verschiedene STM32-Mikrocontroller zu erstellen.

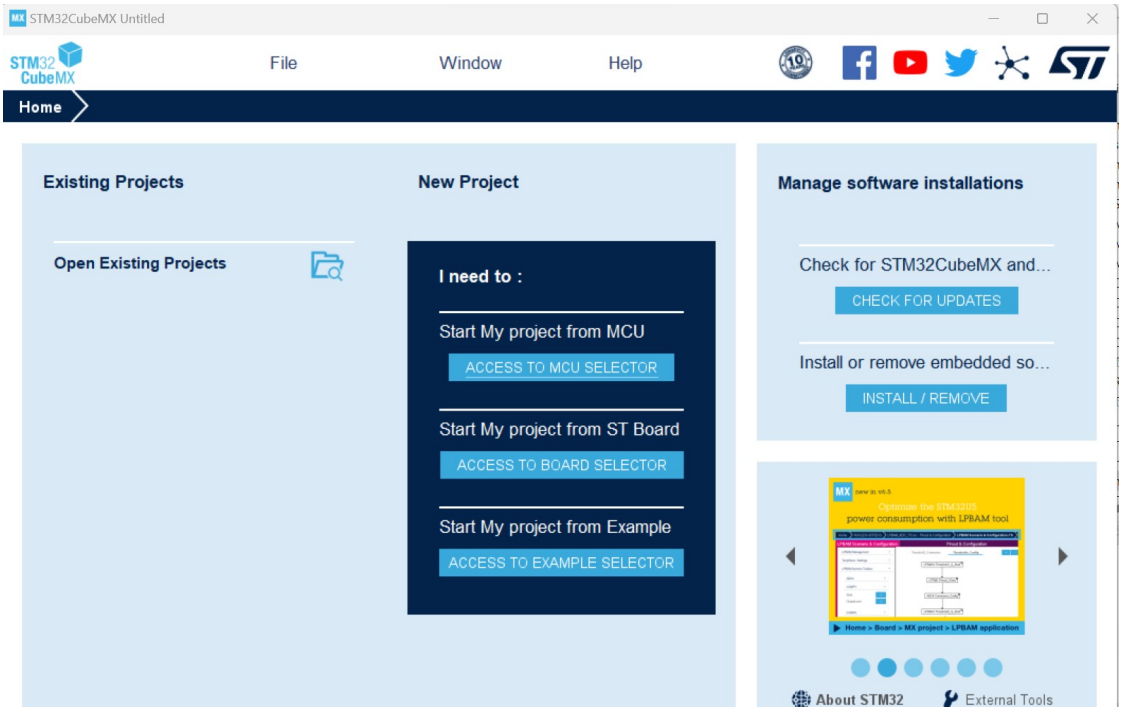


Abbildung 4.2: Startseite von STM32CubeMX

Zu den Funktionen und Tools, die CubeMX für die effiziente Konfiguration von Projekten bietet, gehören:[17]

- Pin-Konfiguration: CubeMX erlaubt es dem Entwickler, die Pin-Belegung des Mikrocontrollers zu definieren und die Pin-Konfiguration für jede Peripherie auszuwählen.
- Clock-Konfiguration: CubeMX erleichtert die Einstellung der System- und Peripherie-Clocks, einschließlich der internen und externen Oszillatoren.
- Peripheral-Konfiguration: CubeMX ermöglicht es dem Entwickler, schnell und einfach die Konfiguration für verschiedene Peripheriegeräte wie ADCs, DACs, Timern und UARTs zu erstellen.
- Projekt-Management: CubeMX erlaubt es Entwicklern, ihre Projekte zu verwalten und zu speichern, so dass sie später einfach wieder aufgenommen werden können.
- Code-Generierung: CubeMX generiert automatisch den Initialisierungscode für die Konfiguration des Mikrocontrollers und der Peripheriegeräte. Dadurch können Entwickler Zeit sparen und sich auf die eigentliche Anwendungslogik konzentrieren.
- Unterstützung für externe Libraries: CubeMX unterstützt die Integration von externen Libraries wie CMSIS, FreeRTOS und HAL.

Insgesamt ist CubeMX eine nützliche Software für Embedded-Entwickler, die eine schnelle und effektive Konfiguration von Mikrocontrollern und Peripheriegeräten benötigen.

4.1.3 QT Creator (GUI)

Qt Creator ist eine integrierte Entwicklungsumgebung (IDE) (Abbildung 4.3) für die Programmiersprache C++ und die Qt-Anwendungsrahmenbibliothek. Es wird von der Firma The Qt Company entwickelt und bietet eine benutzerfreundliche Oberfläche für die Entwicklung von Desktop- und mobilen Anwendungen.

Qt Creator bietet eine Reihe von Funktionen, darunter eine grafische Benutzeroberfläche zum Entwerfen von Benutzeroberflächen (GUIs), einen integrierten Compiler und Debugger sowie eine Code-Vervollständigungsfunktion, die es Entwicklern ermöglicht, schnell und einfach Code zu schreiben.

Qt Creator ist sowohl für Windows als auch für macOS und Linux verfügbar und wird von vielen Entwicklern aufgrund seiner Benutzerfreundlichkeit und Flexibilität geschätzt.

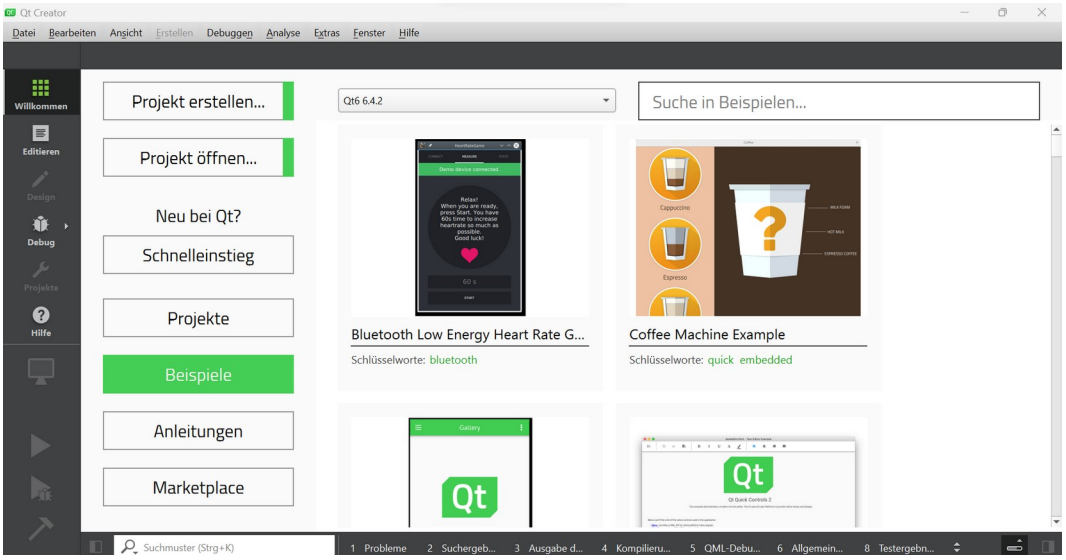


Abbildung 4.3: Qt Creator – Willkommensbildschirm

Auf der linken Seite sieht man den Moduswähler. Mit dem kann man in einen anderen Qt Creator-Modus wechseln.Die wichtigen Modus für uns sind:

• **Willkommensmodus:** Modus zum Öffnen von Projekten, Tutorials und Beispielen.

• **Editiermodus:** Modus zum Bearbeiten von Projekt- und Quelldateien.

• **Designmodus:** Modus zum Entwerfen und Entwickeln von Benutzeroberflächen. Dieser Modus ist für UI-Dateien verfügbar.

Im Design-Fenster kann man das Aussehen der Benutzeroberflächenformulare durch Ziehen und Ablegen von Oberflächenelementen wie z.B. Buttons auf dem Formular gestalten.

Die Namen und Eigenschaften aller Objekte, die Teil des Benutzeroberflächenformulars sind, können nach Bedarf angepasst werden.

Kapitel 5

Controller Area Network

5.1 Einführung in das CAN-Protokoll

Um eine fehlerfreie Kommunikation zwischen verschiedenen Systemen oder Geräten zu gewährleisten, müssen Protokolle definiert werden. CAN (Controller Area Network) ist ein synchrones serielles Übertragungsprotokoll, das in der Automobilindustrie und in anderen industriellen Anwendungen weit verbreitet ist. Es ermöglicht die Übertragung von Daten zwischen verschiedenen Steuergeräten innerhalb eines Fahrzeugs oder industriellen Systemen. CAN ist ein robustes Protokoll mit hoher Geschwindigkeit und Fehlertoleranz, das aufgrund seiner Zuverlässigkeit und hohen Datenrate in vielen Anwendungen bevorzugt wird.

Controller Area Network Protokoll wird im ISO 11898-Standards spezifiziert. Es ist ein nachrichtenorientiertes Protokoll, das hauptsächlich für die elektrische Multiplexverkabelung in der Automobilindustrie entwickelt wurde. Aufgrund seiner hohen Geschwindigkeit, niedrigen Kosten und erheblichen Störfestigkeit fand der CAN-Bus jedoch Anwendung als Feldbus in anderen Automatisierungsumgebungen.

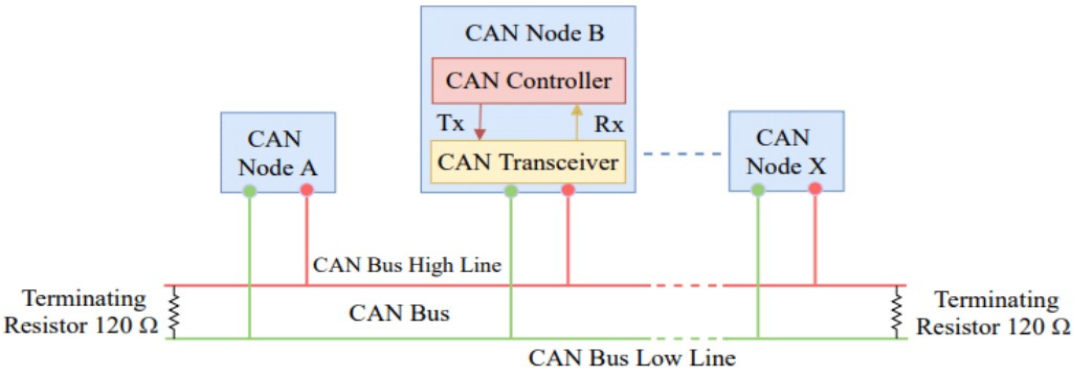


Abbildung 5.1:Controller Area Network (CAN) [18]

Wie in Abbildung 5.1 dargestellt, sind in einem CAN-Netzwerk alle Knoten über einen konventionellen Zweidrahtbus miteinander verbunden, wobei die Drähte aus einem verdrillten Paar bestehen, das eine charakteristische Nennimpedanz von 120 Ω hat, wobei aber auch eine Impedanz von 108 Ω bis 132 Ω zulässig ist. Die physische CAN-Übertragung entspricht den standardisierten Protokollen in ISO 11898-2 und ISO 11898-3, welches auch als High-Speed-CAN bzw. Low-Speed-, fehlertolerantes- oder Fault-Tolerant-CAN bezeichnet wird.

Um eine hohe elektrische Störsicherheit zu erreichen, wird ein Differenzsignal verwendet, bei dem sich ein einzelnes Bit auf zwei Leitungen gleichzeitig mit einer Potentialänderung in entgegengesetzter Richtung manifestiert. Eine invertierte Übertragung des logischen Signals erfolgt folglich auf einer zweiten Leitung. Störungen, die in die Leitung einkoppeln, wirken sich in derselben Richtung auf die Leitung aus. Der Pegelunterschied bleibt bei diesen Fehlern jedoch bestehen, da die beiden Differenzleitungen immer entgegengesetzte Pegel haben. In diesem Kontext wird dies als Gleichtaktunterdrückung (engl. Common Mode Rejection Ratio, CMRR) bezeichnet, wobei die CAN-High- und CAN-Low-Leitungen das invertierte und das nicht invertierte serielle Datensignal enthalten (siehe Abbildung 5.2).

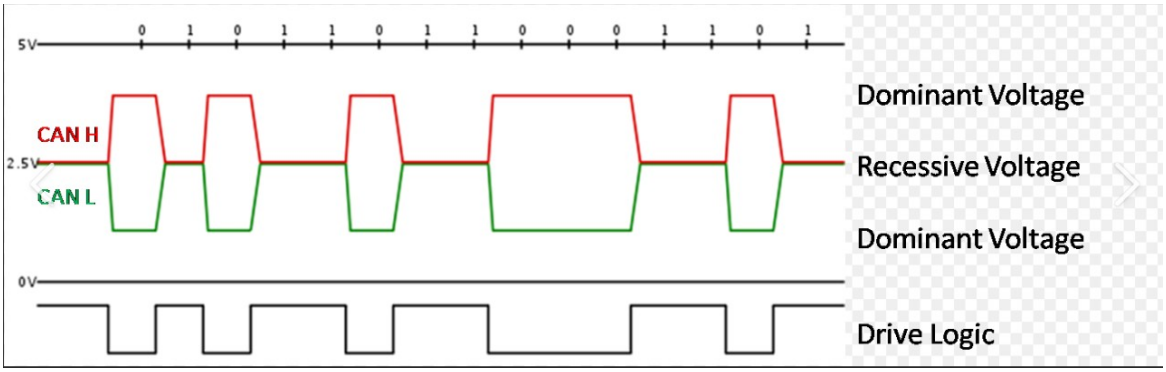


Abbildung 5.2: CAN-High und CAN-Low Signal [19]

Der Zustand mit zwei verschiedenen Pegeln von CAN H und CAN L wird als dominanter Zustand und mit gleichen Pegeln als rezessiver Zustand genannt, wobei die Pegeldifferenz beim dominanten Zustand nominal 2 Volt und bei dem rezessiven 0 Volt beträgt. Der dominante Zustand entspricht einer logischen Null. Dementsprechend überschreibt ein Knoten, der eine logische Null auf den Bus legt, den Zustand einer logischen Eins eines anderen Knotens.

Um die Störsicherheit zu erhöhen, wird die Non-Return to Zero (NRZ) Kodierungstechnik mit Hilfe von „Bit-Stuffing“ eingesetzt. Bei dieser Technik wird nach jeweils 5 Bit derselben Polarität (dominant oder rezessiv) ein Bit mit entgegengesetzter Polarität eingefügt, um einen Verlust der Synchronisation zwischen dem Sender und den CAN-Netzwerkteilnehmern zu verhindern. Die Stuff-Bits werden dann automatisch vom Empfänger gelöscht, damit die gesendete Bit-Sequenz und die zum Hostcontroller weitergeleitete Sequenz identisch bleibt.

5.2 Prinzip des Datenverkehrs in CAN-Netzwerk

Der CAN-Bus ist ein Broadcast-Bustyp, was bedeutet, dass alle Knoten alle übertragenen Nachrichten empfangen können und somit den gesamten Datenverkehr aufnehmen. Bei der Datenübertragung wird daher kein bestimmter Knoten adressiert, sondern der Inhalt einer Nachricht wird durch einen eindeutigen Identifier (ID) festgelegt, der auch die Priorität der entsprechenden Nachricht bestimmt. Indem jeder Teilnehmer im CAN-Netzwerk den ID überprüft, kann er feststellen, ob die empfangene Nachricht für ihn relevant ist oder nicht.

5.3 Aufbau einer CAN-Nachricht

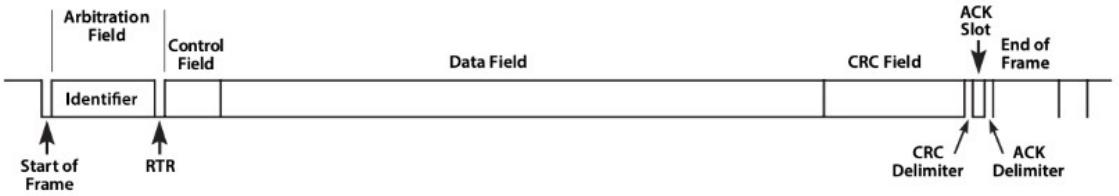
Eine CAN-Nachricht muss eine bestimmte Form besitzen. Diese Form wird Rahmen oder Frame auf Englisch genannt. Es wird zwischen 4 Rahmentypen unterschieden:

- Data-Frame oder Datenrahmen: ist das eigentliche Frame für die Daten-Übertragung.
- Remote Frame: ist ein Frame, das die Übertragung einer bestimmten Kennung anfordert.
- Error-Frame oder Fehlerframe: wird bei Festlegung einer Übertragungsfehler gesendet.
- Overload-Frame oder Überlastungsframe: dient zum Einfügen einer Verzögerung zwischen Daten und Remote Frames.

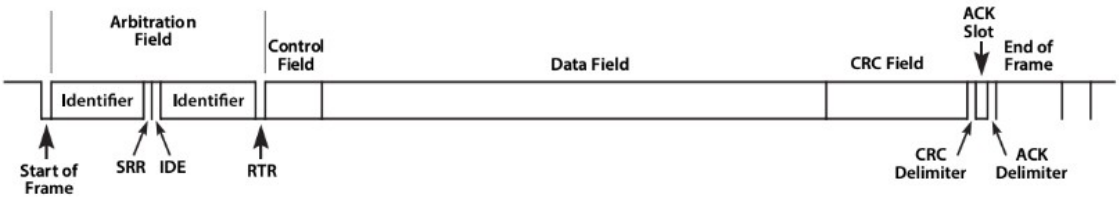
5.3.1 Datenrahmen

Der Datenrahmen kann zwei Nachrichtenformate verwenden. (siehe Abbildung 5.3):

- Ein Basisrahmenformat mit 11-Bit-Identifier, welches nach ISO 11898-1 bzw. CAN 2.0 A und CAN 2.0 B beschrieben wird.
- Ein erweitertes Rahmenformat mit 29-Bit-Identifier, welches nach ISO 11898-1 und durch CAN 2.0 B beschrieben wird.



Ein CAN 2.0A ("Standard CAN") Datenrahmen.



Ein CAN 2.0B ("extended CAN") Datenrahmen.

Abbildung 5.3: Aufbau einer CAN-Nachricht[20]

Wie es der Abbildung 5.3 zu entnehmen ist, besteht der Datenrahmen aus sieben Kennfeldern:

- Startfeld („Start of Frame“):
  Dieses Feld stellt den Start der Rahmenübertragung dar und steht als dominanter Bit „0“ fest. Das SOF dient zur Synchronisation der empfangenden Teilnehmern mit den sendenden Teilnehmern des Busses.
- Arbitrationsfeld:
  Dieses Feld besteht bei CAN 2.0 A aus einem 11-Bit-Identifier und einem RTR-Bit (Remote Transmission Request-Bit), das bei Datenrahmen dominant gesetzt wird. Bei CAN 2.0 B besteht dieses Feld aus einem 29-Bit-Identifier und dem RTR-Bit. Das Arbitrationsfeld bestimmt die Priorität der Nachricht, wenn zwei oder mehr Knoten um den Bus konkurrieren, dabei hat die Nachricht mit dem niedrigsten numerischen Wert im Identifier-Feld die höchste Priorität.
  Für ein Datenrahmen wir das RTR-Bit als dominant „0“ und bei einer Remote-Frame als rezessiv „1“ gesetzt und das sowohl beim Basis- als auch beim Erweiterten Rahmenformat.
- Control Feld:
  Das Control Feld besteht aus einem Identifier Extension-Bit (IDE-Bit), ein dominantes reserviertes Bit „r0“ und vier weiteren Bits „DLC“, die den Datenlängencode enthalten. Das IDE-Bit wird bei einem Identifier mit 11-Bit Länge dominant“0“ und bei einem Identifier mit 29-Bit Länge rezessiv „1“ gesetzt.

- Datenfeld:
  Dieses Feld kann 0 bis 8 Byte an zu übertragende Daten enthalten.
- CRC-Feld:
  CRC steht für „Cyclic Redundancy Check“. Dieses Feld enthält eine 15-Bit-Prüfsumme sowie ein Bit für die Endmarkierung „CRC Delimiter“. Die 15-Bit-Prüfsumme wird für die meisten Teile der Nachricht berechnet und zur Fehlererkennung verwendet.
- Acknowledge-Feld:
  Wird auch ACK-Slot genannt und besteht aus 2 Bits. Jeder CAN-Bus Teilnehmer schickt am Ende der Übertragung eine Rückmeldung in der Form eines Bits, um mitzuteilen, ob der Daten-Empfang korrekt war oder nicht. Dieses Acknowledge-Bit ist bei einem erfolgreichen Daten-Empfang dominant „0“. Mit anderen Worten enthält das ACK-Feld eine Bestätigung von anderen Teilnehmern bei einem korrekten Empfang der Nachricht.
- Ende Feld (end-of-Frame):
  Dieses Feld besteht aus sieben rezessiven Bits und kennzeichnet das Ende des Datenrahmens. Die folgende Abbildung biete eine erweiterte Übersicht über den Kennfeldern bei einem Basis- und erweiterten Datenrahmen.

- Basis Frame nach ISO11898-1 (früher als CAN 2.0A bezeichnet)

Start	Indentifier	RTR	IDE	r0	DLC	Data	CRC	ACK	EOF+FS
1 Bit	11 Bit	1 Bit	1 Bit	1 Bit	4 Bit	0...8 Byte	16 Bit	2 Bit	10 Bit

- Extended Frame nach ISO11898-1 (früher als CAN 2.0B bezeichnet)

Start	Indentifier	SRR	IDE	Indentifier	RTR	r1	r0	DLC	Data	CRC	ACK	EOF+FS
1 Bit	11 Bit	1 Bit	1 Bit	11 Bit	1 Bit	1 Bit	1 Bit	4 Bit	0...8 Byte	16 Bit	2 Bit	10 Bit

Abbildung 5.4: Zusammenfassende Übersicht der Kennfelder bei Basis- und Extended CAN Datenrahmen [21]

5.3.2 Das Remote-Frame

Das Remote-Frame ist genau wie das Datenrahmen bis auf zwei wesentliche Unterschiede:

- Das RTR-Bit bei dem Remote-Frame ist rezessiv „1“.
- Das Remote-Frame enthält kein Datenfeld.

Bei dem Remote Transmission Request (RTR) Frame, wird die Nachricht angefordert, sodass der Empfänger mit dem angeforderten Identifier, den zu diesem Identifier gehörenden Datenrahmen liefert. Das Remote-Frame wird in der Praxis selten genutzt und oft nur für die Implementierung eines Anforderungs-Antwort-Kommunikations-Typs für das Busverkehrsmanagement verwendet. In Abbildung 5.5 ist ein Beispiel zu einem Remote-Frame dargestellt.

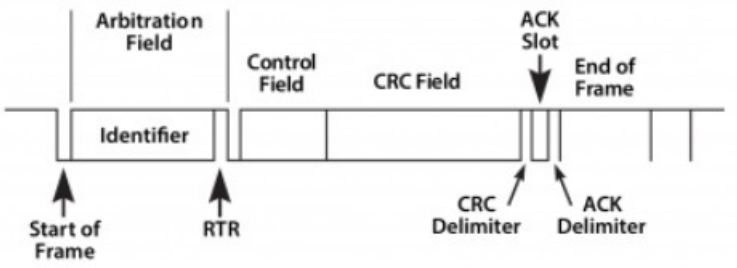


Abbildung 5.5: Remote-Frame [20]

5.3.3 Das Error-Frame:

Ein Fehlerrahmen wird dann übertragen, wenn ein Knoten einen Fehler erkennt, um alle anderen Knoten darauf aufmerksam zu machen und ebenfalls einen Fehler erkennen zu lassen, Die anderen Konten reagieren dann auf die Fehlererkennung ebenfalls mit der Versendung von Fehler-Frames. Dadurch versucht der Sender automatisch die fehlerhaft erkannte Nachricht erneut zu übertragen. Ein Fehler-Frame besteht aus einem Error-Flag, das 6 Bits des gleichen Werts besitzt und einem Error-Delimiter, der aus 8 rezessiven Bits besteht. Im Bus-Active Fall besteht der Error Flag aus dominanten und im Bus-Pasive-Fall aus rezessiven Bits. Das Fehlertrennzeichen bietet den anderen Knoten die Möglichkeit, ihre Fehlerflags auf dem Bus zu senden, nachdem sie das erste Fehlerflag erkannt haben.

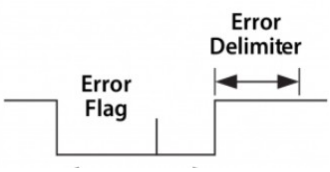


Abbildung 5.6: CAN Fehlerrahmen[20]

5.3.4 Das Overload-Frame:

Das Overload-Frame ist dem Error-Frame sehr ähnlich und wird von einem Knoten übertragen, der zu beschäftigt ist. Der Overload-Frame besteht aus einem Overload-Flag mit 6 dominanten Bits und einem Overload-Delimiter mit 8 rezessiven Bits.

5.4 CAN-Bit-Timing

Das Timing der CAN-Bits spielt eine entscheidende Rolle bei der Erleichterung der Kommunikation innerhalb des CAN-Busses und stellt sicher, dass alle Knoten synchron arbeiten. Um die Synchronisation im CAN-Netzwerk zu erreichen, arbeiten alle Teilnehmer mit derselben nominalen Bit-Zeit, da kein unabhängiges Taktsignal verwendet wird.

Da ein Knoten sowohl die Daten, die er empfängt, als auch die anderer Knoten gleichzeitig wahrnehmen muss, ist es unerlässlich, dass sich diese Knoten während der Arbitrierung selbst synchronisieren. Diese Synchronisation ist auch wichtig, um Fehler zu vermeiden, die aufgrund von Diskrepanzen im Oszillator-Timing zwischen Knoten auftreten können.

Der Beginn der harten Synchronisation erfolgt nach einer Startbitperiode, insbesondere beim ersten Übergang von rezessiv zu dominant. Darüber hinaus findet bei jedem nachfolgenden Übergang von rezessiv zu dominant eine Resynchronisierung statt, bei der die Bit-Zeit entweder verlängert oder verkürzt wird. Idealerweise sollte dieser Übergang in einem Vielfachen der nominalen Bit-Zeit erfolgen. Ist dies jedoch nicht der Fall und werden die Erwartungen des CAN-Controllers nicht erfüllt, wird die nominale Bit-Zeit entsprechend angepasst. Eine nominale Bit-Zeit ist in Abbildung 5.7 dargestellt.

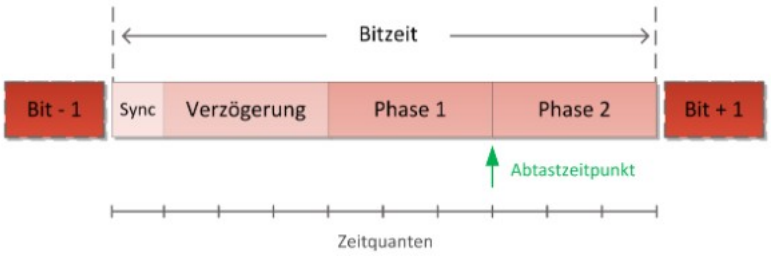


Abbildung 5.7: CAN Bit-Timing[22]

Um die Anpassung der Bit-Zeit zu ermöglichen, wird jedes Bit in Zeitscheiben unterteilt, die als Zeitquanten Tq (Time-Quantum) bezeichnet werden, diese Zeit-Quanten befinden sich in vier Segmenten: das Synchronisationssegment, das Propagation-Segment (Verzögerung), das Phase Buffer Segment 1 und das Phase Buffer Segment 2, dabei variiert die Anzahl der Quanten in dem Segment je nach Bitrate und Netzwerkbedingungen und die Anzahl der Quanten, die in jedem Bit unterteilt sind, je nach Controller.

Das Synchronisationssegment, das immer ein Quantum lang ist, wird für die Synchronisation der Knoten im CAN-Bus verwendet. Das Ausbreitungssegment wird benötigt, um die Verspätung der Übertragung in den Busleitungen auszugleichen. Wenn sich ein Übergang verspätet oder verzögert, wird dem Mikrocontroller veranlasst, die Zeitdifferenz zu kompensieren, in dem er das Phasensegment 1 verlängert oder das Phasensegment 2 verkürzt, um die Synchronisation zu erreichen. Dieser Prozess (Resynchronisation) wird kontinuierlich bei jedem rezessiven zu dominantem Übergang durchgeführt, um sicherzustellen, dass der Sender und Empfänger synchron bleiben.

Zur Bestimmung der Zeit-Quanten, um welche die Phasensegmente 1 und 2 verlängert oder verkürzt werden, kommt die sogenannte Synchronisationssprungweite (Synchronization Jump Width oder SJW) zum Tragen, die einen Wert von 1 bis 4 Zeit-Quanten annehmen kann.

Wenn ein Bit-Übergang nach dem Synchronisationssegment auftritt, wird dies als positiver Phasenfehler bezeichnet. Und wenn dieser Phasenfehler kleiner als das SJW ist, wird er kompensiert, in dem das Phase Buffer Segment 1 diesem Fehler entsprechend verlängert wird, da die SJW der maximalen möglichen Bitzeitänderung entspricht.

Wenn ein Bit-Übergang vor dem Synchronisationssegment auftritt, wird dies als negativer Phasenfehler bekannt. Und wenn dieser Phasenfehler ebenfalls kleiner als das SJW ist, wird er kompensiert, indem das Phase Buffer Segment 2 diesem Fehler entsprechend verkürzt wird, da das SJW die maximale mögliche Bitzeitänderung darstellt.

Wird ein Phasenfehler erkannt, der im Betrag größer oder gleich dem Synchronisationssprungweite SJW ist, wird die Bitzeit um den Betrag des SJW verkürzt bzw. verlängert.

Kapitel 6

Konfiguration und Verarbeitung der CAN-Kommunikation

In diesem Kapitel wird die Konfiguration des Mikrocontrollers für die CAN Kommunikation beschrieben. Außerdem wird der Code erklärt mit dem CAN-Nachrichten vom Mikrocontroller gesendet und empfangen werden können.

6.1 Aktivierung der CAN-Peripheriegeräts mit STM32CubeMx

Bei der Konfiguration des Mikrocontrollers mit STM32CubeMx muss zunächst das CAN-Peripheriegerät aktiviert werden. Dazu wird in STM32CubeMx in der Ansicht „Pinout“ der Punkt „Connectivity“ ausgewählt und CAN1 durch Anklicken aktiviert. Infolgedessen werden die Pins 11 und 12 auf der rechten Seite als CAN RX und TX Port zugeordnet und grün markiert (siehe Abbildung 6.1)

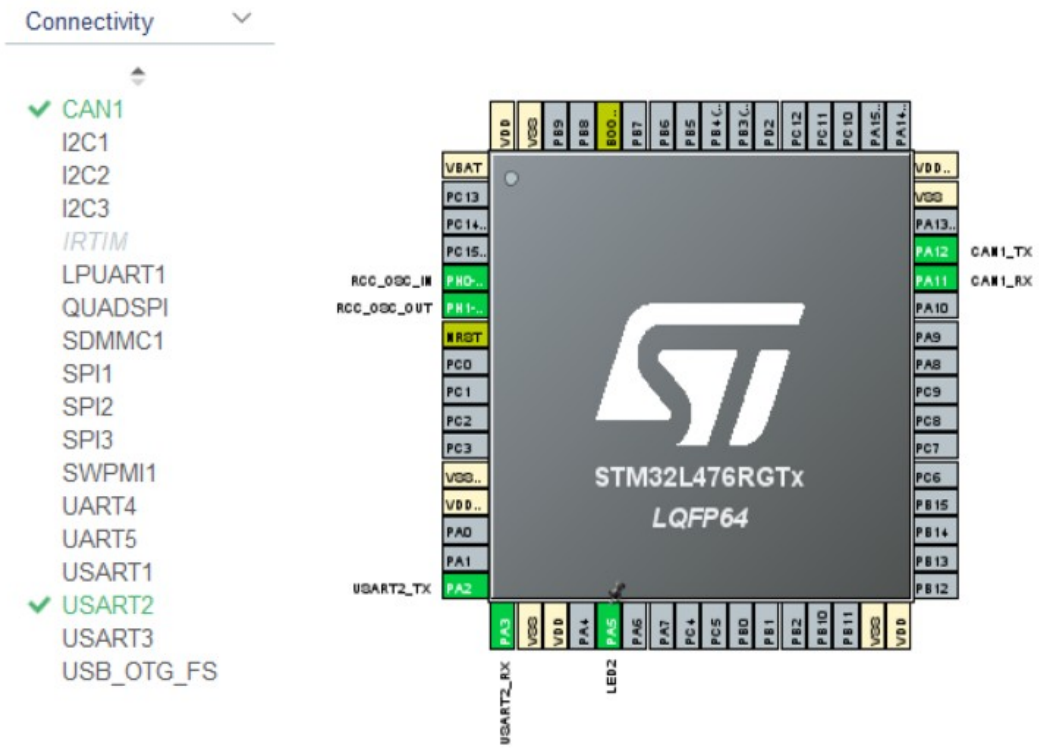


Abbildung 6.1: Aktivierung der CAN Protokolleinheit mit STM32CubeMx

Anschließend kann der C Code für die Geräteaktivierung generiert und mit STM32CubeIDE angepasst werden. In den folgenden Abschnitten werden zunächst einige Datenstrukturen und Funktionen der Programmierschnittstelle, die für diese Arbeit relevant sind (engl. Application Programming Interface, API) vorgestellt und diskutiert.

6.2 Strukturen der CAN-Firmware-Treiberregister [23]

In diesem Abschnitt werden die relevanten Parameter für die Konfiguration des CAN- Controllers vorgestellt. Diese werden anhand der Dokumentation der Hardwareabstraktionsschicht (HAL) des STM 32- ermittelt und erläutert. Die in den folgenden Abschnitten dokumentierten Strukturen werden in der Header-Datei stm32l4xx_hal_can.h des CAN HAL-Moduls definiert.

6.2.1 CAN_InitTypeDef

Die Struktur CAN_InitTypeDef definiert die Werte für grundlegende Parameter. Der Quellcode 6.1 zeigt den zugehörigen Ausschnitt der Header-Datei stm32l4xx_hal_can.h:

```
typedef struct
{
 uint32_t Prescaler;                 /*!< Specifies the length of a time
                                          quantum. This parameter must be a
                                          number between Min_Data = 1 and
                                          Max_Data = 1024. */
 uint32_t Mode;                      /*!< Specifies the CAN operating mode. This
                                          parameter can be a value of @ref
                                          CAN_operating_mode */
 uint32_t SyncJumpWidth;             /*!< Specifies the maximum number of time
                                          quanta the CAN hardware is allowed to
                                          lengthen or shorten a bit to perform
                                          resynchronization. This parameter can
                                          be a value of @ref
                                          CAN_synchronisation_jump_width */
 uint32_t TimeSeg1;                  /*!< Specifies the number of time quanta in
                                          Bit Segment 1. This parameter can be a
                                          value of @ref
                                          CAN_time_quantum_in_bit_segment_1 */
 uint32_t TimeSeg2;                  /*!< Specifies the number of time quanta in
                                          Bit Segment 2. This parameter can be a
                                          value of @ref
                                          CAN_time_quantum_in_bit_segment_2 */
 FunctionalState TimeTriggeredMode;  /*!< Enable or disable the time triggered
                                          communication mode. This parameter can
                                          be set to ENABLE or DISABLE. */
 FunctionalState AutoBusOff;         /*!< Enable or disable the automatic bus-
                                          off management.This parameter can be
                                          set to ENABLE or DISABLE. */
 FunctionalState AutoWakeUp;         /*!< Enable or disable the automatic wake-
                                          up mode. This parameter can be set to
                                          ENABLE or DISABLE. */
 FunctionalState AutoRetransmission; /*!< Enable or disable the non-automatic
                                          retransmission mode. This parameter
                                          can be set to ENABLE or DISABLE. */
 FunctionalState ReceiveFifoLocked;  /*!< Enable or disable the Receive FIFO
                                          Locked mode. This parameter can be set
                                          to ENABLE or DISABLE. */
 FunctionalState TransmitFifoPriority;/*!< Enable or disable the transmit FIFO
                                          priority. This parameter can be set to
                                          ENABLE or DISABLE. */
} CAN_InitTypeDef;
```

Quellcode 6.1: Definition von CAN_InitTypeDef in der stm32l4xx_hal_can.h

Die Bedeutung der einzelnen Parameter der CAN_InitTypeDef und die Werte, die sie annehmen können, werden im Folgenden aufgelistet.

- uint32_t CAN_InitTypeDef::Prescaler
  Dieser Wert gibt die Länge eines Zeitquantums an. Dieser Parameter muss eine Zahl zwischen Min_Data = 1 und Max_Data = 1024 sein.
- uint32_t CAN_InitTypeDef::Mode
  Dieser Parameter legt den CAN Operating Mode fest. Mögliche Werte sind:
  - CAN_MODE_NORMAL
  - CAN_MODE_LOOPBACK
  - CAN_MODE_SILENT
  - CAN_MODE_SILENT_LOOPBACK
- uint32_t CAN_InitTypeDef::SyncJumpWidth
  Dierser Paramter gibt die maximale Anzahl von Zeitquanten an, um welche die CAN-Hardware ein Bit verlängern oder verkürzen darf, um eine Resynchronisierung durchzuführen. Dieser Parameter kann ein Wert von CAN_synchronisation_jump_width sein:
  - CAN_SJW_1TQ: ein Zeitquantum
  - CAN_SJW_2TQ: zwei Zeitquanten
  - CAN_SJW_3TQ: drei Zeitquanten
  - CAN_SJW_4TQ: vier Zeitquanten
- uint32_t CAN_InitTypeDef::TimeSeg1:
  Dieser Parameter definiert die Anzahl der Zeitquanten im Bitsegment 1. Er kann einen Wert von CAN_time_quantum_in_bit_segment_1 mit CAN_BS1_*n*TQ annehmen, wobei *n* der Anzahl der Zeitquanten entspricht und Werte zwischen 1 und 16 annehmen kann.
- uint32_t CAN_InitTypeDef::TimeSeg2
  Gibt die Anzahl der Zeitquanten im Bitsegment 2 an. Dieser Parameter kann einen Wert von CAN_time_quantum_in_bit_segment_2 mit CAN_BS2_*n*TQ annehmen, wobei *n* der Anzahl der Zeiztqaunten entspricht und Werte zwischen 1und 8 annehmen kann.
- FunctionalState CAN_InitTypeDef::TimeTriggeredMode
  Aktivierung oder Deaktivierung des zeitgesteuerten Kommunikationsmodus. Dieser Parameter kann auf ENABLE oder DISABLE gesetzt werden.
- FunctionalState CAN_InitTypeDef::AutoBusOff
  Aktivierung oder Deaktivierung der automatischen Bus-Off-Verwaltung. Dieser Parameter kann auf ENABLE oder DISABLE gesetzt werden.
- FunctionalState CAN_InitTypeDef::AutoWakeUp
  Aktivierung oder Deaktivierung des automatischen Weckmodus. Dieser Parameter kann auf ENABLE oder DISABLE gesetzt werden.

- FunctionalState CAN_InitTypeDef::AutoRetransmission
  Aktivierung oder Deaktivierung der automatischen Wiederholung von CAN Botschaften. Dieser Parameter kann auf ENABLE oder DISABLE gesetzt werden.
- FunctionalState CAN_InitTypeDef::ReceiveFifoLocked
  Aktivierung oder Deaktivierung des Receive FIFO Locked Modus. Wenn dieser Modus aktiviert wird, werden alte CAN Botschaften nicht von neuen überschrieben. Dieser Parameter kann auf ENABLE oder DISABLE gesetzt werden.
- FunctionalState CAN_InitTypeDef::TransmitFifoPriority
  Aktivierung oder Deaktivierung des Transmit-FIFO-Priority. Wenn dieses Bit gesetzt wird, wird der Sender Buffer in einer First-in First-Out Wiese ausgelesen. Neue Botschaften können alte Botschaften nicht überschreiben. Dieser Parameter kann auf ENABLE oder DISABLE gesetzt werden.

6.2.2 CAN_FilterTypeDef

Mit den Parametern von CAN_FilterTypeDef (siehe Quellcode 6.2) werden Filtereigenschaften festgelegt.

```
typedef struct
{
 uint32_t FilterIdHigh;          /*!< Specifies the filter identification number
                                      (MSBs for a 32-bit configuration, first one
                                      for a 16-bit configuration). This parameter
                                      must be a number between Min_Data = 0x0000
                                      and Max_Data = 0xFFFF. */
 uint32_t FilterIdLow;           /*!< Specifies the filter identification number
                                      (LSBs for a 32-bit configuration, second
                                      one for a 16-bit configuration). This para-
                                      meter must be a number between Min_Data =
                                      0x0000 and Max_Data = 0xFFFF. */
 uint32_t FilterMaskIdHigh;      /*!< Specifies the filter mask number or identi-
                                      fication number, according to the mode
                                      (MSBs for a 32-bit configuration, first one
                                      for a 16-bit configuration). This parameter
                                      must be a number between Min_Data = 0x0000
                                      and Max_Data = 0xFFFF. */
 uint32_t FilterMaskIdLow;       /*!< Specifies the filter mask number or identi-
                                      fication number, according to the mode
                                      (LSBs for a 32-bit configuration, second
                                      one for a 16-bit configuration). This para-
                                      meter must be a number between Min_Data =
                                      0x0000 and Max_Data = 0xFFFF. */
 uint32_t FilterFIFOAssignment;  /*!< Specifies the FIFO (0 or 1U) which will be
                                      assigned to the filter. This parameter can
                                      be a value of @ref CAN_filter_FIFO */
 uint32_t FilterBank;            /*!< Specifies the filter bank which will be in-
                                      itialized. For single CAN instance(14 dedi-
                                      cated filter banks), this parameter must be
                                      a number between Min_Data = 0 and Max_Data
                                      = 13. For dual CAN instances(28 filter
                                      banks shared), this parameter must be a
                                      number between Min_Data = 0 and Max_Data =
```

```
                                        27. */
  uint32_t FilterMode;              /*!< Specifies the filter mode to be initiali-
                                        zed. This parameter can be a value of @ref
                                        CAN_filter_mode */
  uint32_t FilterScale;             /*!< Specifies the filter scale. This parameter
                                        can be a value of @ref CAN_filter_scale */
  uint32_t FilterActivation;        /*!< Enable or disable the filter. This parame-
                                        ter can be a value of @ref CAN_filter_acti-
                                        vation */
  uint32_t SlaveStartFilterBank;    /*!< Select the start filter bank for the slave
                                        CAN instance. For single CAN instances,
                                        this parameter is meaningless. For dual CAN
                                        instances, all filter banks with lower in-
                                        dex are assigned to master CAN instance,
                                        whereas all filter banks with greater index
                                        are assigned to slave CAN instance. This
                                        parameter must be a number between Min_Data
                                        = 0 and Max_Data = 27. */
} CAN_FilterTypeDef;
```

Quellcode 6.2: Defintion CAN_FilterTypeDef in der stm32l4xx_hal_can.h

Die Struktur enthält die folgenden Parameter

- uint32_t CAN_FilterTypeDef::FilterIdHigh:
  Dabei handelt es sich um die Filteridentifikationsnummer (MSB für eine 32-Bit-Konfiguration, LSB für eine 16-Bit-Konfiguration). Dieser Parameter ist dementsprechend eine Zahl zwischen Min_Data = 0x0000 und Max_Data = 0xFFFF sein.
- uint32_t CAN_FilterTypeDef::FilterIdLow:
  Dabei handelt es sich um die Filteridentifikationsnummer (LSBs für eine 32-Bit-Konfiguration, MSB für eine 16-Bit-Konfiguration). Dieser Parameter muss daher eine Zahl zwischen Min_Data = 0x0000 und Max_Data = 0xFFFF sein.
- uint32_t CAN_FilterTypeDef::FilterMaskIdHigh:
  Dieser Parameter gibt die Filtermaskennummer bzw. -identifikationsnummer entsprechend dem Modus an (MSBs für eine 32-Bit-Version Konfiguration, LSB für eine 16-Bit-Konfiguration). Dieser Parameter muss eine Zahl zwischen Min_Data = 0x0000 und Max_Data = 0xFFFF sein.
- uint32_t CAN_FilterTypeDef::FilterMaskIdLow:
  Dieser Parameter gibt die Filtermaskennummer bzw. -identifikationsnummer entsprechend des Modus an (LSBs für eine 32-Bit-Version Konfiguration, zweite für eine 16-Bit-Konfiguration). Der Parameter muss eine Zahl zwischen Min_Data = 0x0000 und Max_Data = 0xFFFF sein.
- uint32_t CAN_FilterTypeDef::FilterFIFOAssignment:
  Gibt den FIFO (0 oder 1U) an, welcher dem Filter zugewiesen wird. Dieser Parameter kann folgende Werte von CAN_filter_FIFO annehmen:
  - CAN_FILTER_FIFO0: Filter Zuweisung FIFO 0
  - CAN_FILTER_FIFO1: Filter Zuweisung FIFO 1

- uint32_t CAN_FilterTypeDef::FilterBank:
  Dieser Parameter legt die Filterbank fest, welche initialisiert wird. Für eine einzelne CAN-Instanz, die über 14 dedizierte Filterbänke verfügt, muss der Parameter eine Zahl zwischen Min_Data = 0 und Max_Data = 13 sein. Für duale CAN-Instanzen, die sich 28 Filterbänke teilen, muss er eine Zahl sein, die zwischen Min_Data = 0 und Max_Data = 27 liegt.
- uint32_t CAN_FilterTypeDef::FilterMode:
  Hier wird der zu initialisierende Filtermodus festgelegt. Dieser Parameter kann ein Wert von CAN_filter_mode annehmen. Mögliche Ausprägungen sind
  - CAN_FILTERMODE_IDMASK: Identifier mask mode
  - CAN_FILTERMODE_IDLIST: Identifier list mode
- uint32_t CAN_FilterTypeDef::FilterScale
  Gibt die Filterskala an. Dieser Parameter kann ein Wert von CAN_filter_scale sein:
  - CAN_FILTERSCALE_16BIT: Two 16-bit filters
  - CAN_FILTERSCALE_32BIT: One 32-bit filter
- uint32_t CAN_FilterTypeDef::FilterActivation:
  Dieser Parameter legt den Aktivierungszustand des Filters fest und kann einen Wert von CAN_filter_activation haben, nämlich:
  - CAN_FILTER_DISABLE: Filter ist deaktiviert
  - CAN_FILTER_ENABLE: Filter ist aktiviert
- uint32_t CAN_FilterTypeDef::SlaveStartFilterBank:
  Dieser Parameter definiert die Aufteilung der Filterbänke zwischen Master und Slave bei dualen CAN-Instanzen durch die Angabe eines Startindex. Filterdatenbänke mir einem kleineren Index werden Master-CAN-Instanz zugewiesen, während alle Filterbänke mit gleichem und größerem Index der Slave-CAN-Instanz zugeordnet sind. Dieser Parameter muss eine Zahl zwischen Min_Data = 0 und Max_Data = 27 sein. Für einzelne CAN-Instanzen ist dieser Parameter ohne Bedeutung.

6.2.3 CAN_TxHeaderTypeDef

Mit Hilfe der CAN_TxHeaderTypeDef wird der Header von CAN-Tx-Nachrichten definiert. Quellcode 6.3 zeigt den zugehörigen Ausschnitt aus der Hader-Datei stm32l4xx_hal_can.h.

```
typedef struct
{
 uint32_t StdId;     /*!< Specifies the standard identifier.
                          This parameter must be a number between Min_Data = 0
                          and Max_Data = 0x7FF. */
 uint32_t ExtId;     /*!< Specifies the extended identifier.
                          This parameter must be a number between Min_Data = 0
                          and Max_Data = 0x1FFFFFFF. */
 uint32_t IDE;       /*!< Specifies the type of identifier for the message that
```

```
                          will be transmitted. This parameter can be a value of
                          @ref CAN_identifier_type */
  uint32_t RTR;       /*!< Specifies the type of frame for the message that will
                          be transmitted. This parameter can be a value of @ref
                          CAN_remote_transmission_request */

  uint32_t DLC;       /*!< Specifies the length of the frame that will be
                          transmitted. This parameter must be a number between
                          Min_Data = 0 and Max_Data = 8. */
  FunctionalState TransmitGlobalTime; /*!< Specifies whether the timestamp counter
                          value captured on start of frame transmission, is sent
                          in DATA6 and DATA7 replacing pData[6] and pData[7].This
                          parameter can be set to ENABLE or DISABLE.*/
} CAN_TxHeaderTypeDef;
```

Quellcode 6.3: Definition CAN_TxHeaderTypeDef in der stm32l4xx_hal_can.h

Im einzelnen enthält die CAN_TxHeaderTypeDef die folgenden Parameter:

- uint32_t CAN_TxHeaderTypeDef::StdId:
  Dieser Parameter repräsentiert den Standardbezeichner. Er muss eine Zahl zwischen Min_Data = 0 und Max_Data =0x7FF sein.
- uint32_t CAN_TxHeaderTypeDef::ExtId:
  Dieser Parameter enthält den erweiterten Bezeichner, welcher eine Zahl zwischen Min_Data = 0 und Max_Data =0x1FFFFFFF sein muss.
- uint32_t CAN_TxHeaderTypeDef::IDE
  Hiermit wird der Typ des Bezeichners für die Nachricht angegeben, die übertragen wird. Dieser Parameter kann ein Wert von CAN_identifier_type sein, d. h.:
  - CAN_ID_STD: Standardbezeichner
  - CAN_ID_EXT: erweiterter Bezeichner
- uint32_t CAN_TxHeaderTypeDef::RTR
  Dieser Paramter gibt den Frametyp für die Nachricht an, die übertragen wird. Hier kann ein Wert von CAN_remote_transmission_request zugewiesen werden:
  - CAN_RTR_DATA: Data frame
  - CAN_RTR_REMOTE: Remote frame
- uint32_t CAN_TxHeaderTypeDef::DLC
  Hiermit wird die Länge des Frames, der übertragen wird, festgelegt. Dieser Parameter muss eine Zahl zwischen Min_Data = 0 und Max_Data = 8 sein.
- FunctionalState CAN_TxHeaderTypeDef::TransmitGlobalTime
  Gibt an, ob der Zählerwert des Zeitstempels, der bei Beginn der Frame-Übertragung erfasst wird, in DATA6 und DATA7 gesendet wird und damit pData[6] und pData[7] ersetzt. Dieser Parameter kann auf ENABLE oder DISABLE gesetzt werden. Anmerkung:
  - Der zeitgesteuerte Kommunikationsmodus muss aktiviert sein.
  - DLC muss als 8 Bytes programmiert werden, damit diese 2 Bytes gesendet werden.

6.2.4 CAN_RxHeaderTypeDef

Datenstrukturen vom Typ CAN_RxHeaderTypeDef enthalten den Headers von CAN-Rx-Nachrichten. Der Aufbau kann dem Quellcode 6.4 entnommen werden.

```
typedef struct
{
 uint32_t StdId;     /*!< Specifies the standard identifier.
                          This parameter must be a number between Min_Data = 0
                          and Max_Data = 0x7FF. */
 uint32_t ExtId;     /*!< Specifies the extended identifier.
                          This parameter must be a number between Min_Data = 0
                          and Max_Data = 0x1FFFFFFF. */
 uint32_t IDE;       /*!< Specifies the type of identifier for the message that
                          will be transmitted. This parameter can be a value of
                          @ref CAN_identifier_type */
 uint32_t RTR;       /*!< Specifies the type of frame for the message that will
                          be transmitted. This parameter can be a value of @ref
                          CAN_remote_transmission_request */
 uint32_t DLC;       /*!< Specifies the length of the frame that will be
                          transmitted. This parameter must be a number between
                          Min_Data = 0 and Max_Data = 8. */
 uint32_t Timestamp;  /*!< Specifies the timestamp counter value captured on
                          start of frame reception.  This parameter must be a
                          number between Min_Data = 0 and Max_Data = 0xFFFF. */
 uint32_t FilterMatchIndex; /*!< Specifies the index of matching acceptance
                          filter element. This parameter must be a number between
                          Min_Data = 0 and Max_Data = 0xFF. */
} CAN_RxHeaderTypeDef;
```

Quellcode 6.4: Definition CAN_RxHeaderTypeDef in der stm32l4xx_hal_can.h

Die Struktur stimmt weitgehend mit der Struktur des Tx-Headers überein. Die Parameter des CAN_RxHeaderTypeDef sind die folgenden:

- uint32_t CAN_RxHeaderTypeDef::StdId:
  Dieser Parameter gibt den Standardbezeichner an und muss eine Zahl zwischen Min_Data = 0 und Max_Data =0x7FF sein.
- uint32_t CAN_RxHeaderTypeDef::ExtId:
  Dieser Parameter gibt den erweiterten Bezeichner an und muss eine Zahl zwischen Min_Data = 0 und Max_Data =0x1FFFFFFF sein.
- uint32_t CAN_RxHeaderTypeDef::IDE:
  Durch diesen Parameter wird der Typ des Bezeichners für die Nachricht angegeben. Er kann ein Wert von CAN_identifier_type sein, dieses sind:
  - CAN_ID_STD: Standardbezeichner
  - CAN_ID_EXT: erweiterter Bezeichner
- uint32_t CAN_RxHeaderTypeDef::RTR:
  Hiermit wird der Frametyp für die übertragene Nachricht definiert. Dieser Parameter kann ein Wert von CAN_remote_transmission_request sein:
  - CAN_RTR_DATA: Data frame

  - CAN_RTR_REMOTE: Remote frame
- uint32_t CAN_RxHeaderTypeDef::DLC:
  Diesser Parameter gibt die Länge des übetragenen Frames an. Er muss eine Zahl zwischen Min_Data = 0 und Max_Data = 8 sein.
- uint32_t CAN_RxHeaderTypeDef::Timesmap:
  Dieser Parameter definiert den Zählerwert des Zeitstempels, der zu Beginn des Empfangs des Frames erfasst wird. Dieser Parameter muss eine Zahl zwischen Min_Data = 0 und Max_Data = 0xFFFF.
  Anmerkung:: Der zeitgesteuerte Kommunikationsmodus muss aktiviert sein.
- uint32_t CAN_RxHeaderTypeDef::FilterMatchIndex.
  Hiermit wird der Index des Matching Akzeptanzfilterelements spezifiziert. Bei der Akzeptanzfilterung werden nur Filterelemente mit einem kleineren Index als dem Matchingindex verwendet. Dieser Parameter muss eine Zahl zwischen Min_Data = 0 und Max_Data = 0xFF

6.2.5 CAN_HandleTypeDef

Mit Hilfe des CAN_HandleTypeDef wird in der Header-Datei stm32l4xx_hal_can.h die Struktur der CAN-Handles definiert:(siehe Quellcode 6.5)

```
typedef struct __CAN_HandleTypeDef
{
 CAN_TypeDef                 *Instance;    /*!< Register base address */
 CAN_InitTypeDef             Init;         /*!< CAN required parameters */
 __IO HAL_CAN_StateTypeDef   State;        /*!< CAN communication state */
 __IO uint32_t               ErrorCode;    /*!< CAN Error code.
                                                This parameter can be a value of
                                                @ref CAN_Error_Code */

#if USE_HAL_CAN_REGISTER_CALLBACKS == 1
 void (* TxMailbox0CompleteCallback)(struct __CAN_HandleTypeDef *hcan);
                                       /*!< CAN Tx Mailbox 0 complete callback */
 void (* TxMailbox1CompleteCallback)(struct __CAN_HandleTypeDef *hcan);
                                       /*!< CAN Tx Mailbox 1 complete callback */
 void (* TxMailbox2CompleteCallback)(struct __CAN_HandleTypeDef *hcan);
                                       /*!< CAN Tx Mailbox 2 complete callback */
 void (* TxMailbox0AbortCallback)(struct __CAN_HandleTypeDef *hcan);
                                       /*!< CAN Tx Mailbox 0 abort callback */
 void (* TxMailbox1AbortCallback)(struct __CAN_HandleTypeDef *hcan);
                                       /*!< CAN Tx Mailbox 1 abort callback  */
 void (* TxMailbox2AbortCallback)(struct __CAN_HandleTypeDef *hcan);
                                       /*!< CAN Tx Mailbox 2 abort callback */
 void (* RxFifo0MsgPendingCallback)(struct __CAN_HandleTypeDef *hcan);
                                       /*!< CAN Rx FIFO 0 msg pending callback */
 void (* RxFifo0FullCallback)(struct __CAN_HandleTypeDef *hcan);
                                       /*!< CAN Rx FIFO 0 full callback */
 void (* RxFifo1MsgPendingCallback)(struct __CAN_HandleTypeDef *hcan);
                                       /*!< CAN Rx FIFO 1 msg pending callback */
 void (* RxFifo1FullCallback)(struct __CAN_HandleTypeDef *hcan);
                                       /*!< CAN Rx FIFO 1 full callback */
 void (* SleepCallback)(struct __CAN_HandleTypeDef *hcan);
```

```
                                        /*!< CAN Sleep callback */
  void (* WakeUpFromRxMsgCallback)(struct __CAN_HandleTypeDef *hcan);
                                        /*!< CAN Wake Up from Rx msg callback */
  void (* ErrorCallback)(struct __CAN_HandleTypeDef *hcan);
                                        /*!< CAN Error callback */
  void (* MspInitCallback)(struct __CAN_HandleTypeDef *hcan);
                                        /*!< CAN Msp Init callback */
  void (* MspDeInitCallback)(struct __CAN_HandleTypeDef *hcan);
                                        /*!< CAN Msp DeInit callback */
#endif /* (USE_HAL_CAN_REGISTER_CALLBACKS) */
} CAN_HandleTypeDef;
```

Quellcode 6.5: Definition CAN_HandleTypeDef in der stm32l4xx_hal_can.h

Die Struktur umfasst folgende Parameter

- CAN_TypeDef __CAN_HandleTypeDef::Instance:
  Dieser Parameter legt die Basisadresse des Registers fest.
- CAN_InitTypeDef __CAN_HandleTypeDef::Init:
  Hiermit werden die erforderlichen CAN-Parameter angegeben.
- IO HAL_CAN_StateTypeDef __CAN_HandleTypeDef::State
  Dieseer Parameter repräsentiert den Zustand der CAN-Kommunikation.
- IO uint32_t __CAN_HandleTypeDef::ErrorCode:
  Mit diesem Parameter wird ein CAN-Fehlercode angegeben. Dieser Parameter kann ein Wert von CAN_Error_Code (siehe Tabelle 6.1) sein.

Tabelle 6.1: CAN_Error_Code, die in der stm32l4xx_hal_can.h festgelegt sind [24]

Error-Code	Wert	Bedeutung
HAL_CAN_ERROR_NONE	(0x00000000U)	No error
HAL_CAN_ERROR_EWG	(0x00000001U)	Protocol error warning
HAL_CAN_ERROR_EPV	(0x00000002U)	Error passive
HAL_CAN_ERROR_BOF	(0x00000004U)	Bus off error
HAL_CAN_ERROR_STF	(0x00000008U)	Stuff error
HAL_CAN_ERROR_FOR	(0x00000010U)	Form error
HAL_CAN_ERROR_ACK	(0x00000020U)	Acknowledgment error
HAL_CAN_ERROR_BR	(0x00000040U)	Bit recessive error
HAL_CAN_ERROR_BD	(0x00000080U)	Bit dominant error
HAL_CAN_ERROR_CRC	(0x00000100U)	CRC error
HAL_CAN_ERROR_RX_FOV0	(0x00000200U)	Rx FIFO0 overrun error
HAL_CAN_ERROR_RX_FOV1	(0x00000400U)	Rx FIFO1 overrun error
HAL_CAN_ERROR_TX_ALST0	(0x00000800U)	Tx0 transmit failure due to arbitration lost
HAL_CAN_ERROR_TX_TERR0	(0x00001000U)	Tx0 transmit failure due to transmit error
HAL_CAN_ERROR_TX_ALST1	(0x00002000U)	Tx1 transmit failure due to arbitration lost

Error-Code	Wert	Bedeutung
HAL_CAN_ERROR_TX_TERR1	(0x00004000U)	Tx1 transmit failure due to transmit error
HAL_CAN_ERROR_TX_ALST2	(0x00008000U)	Tx2 transmit failure due to arbitrary lost
HAL_CAN_ERROR_TX_TERR2	0x00010000U)	Tx2 transmit failure due to transmit error
HAL_CAN_ERROR_TIMEOUT	(0x00020000U)	Timeour error
HAL_CAN_ERROR_NOT_INITIALIZED	(0x00040000U)	Peripheral not initialized
HAL_CAN_ERROR_NOT_READY	(0x00080000U)	Peripheral not ready
HAL_CAN_ERROR_NOT_STARTED	(0x00100000U)	Peripheral not started
HAL_CAN_ERROR_PARAM	(0x00200000U)	Paramter error

6.3 Einige Funktionen der CAN-Library

6.3.1 Initialisierungs- und Deinitialisierungsfunktionen

Dieser Abschnitt enthält Funktionen, die Folgendes ermöglichen:

- HAL_CAN_Init : Initialisierung und Konfiguration der CAN-Schnittstelle.
- HAL_CAN_DeInit : Deinitialisierung der CAN-Schnittstelle.
- HAL_CAN_MspInit: Initialisierung der CAN MSP (Microcontroller Support Package).
- HAL_CAN_MspDeInit : Deinitialisierung der CAN MSP.

6.3.2 Konfigurations- und Kontrollfunktionen

Dieser Abschnitt enthält Funktionen, für die Konfiguration und Steuerung der CAN Protokolleinheit.

- HAL_CAN_ConfigFilter : Konfiguration des CAN-Empfangsfilters
- HAL_CAN_Start : Start des CAN-Moduls.
- HAL_CAN_Stop : Stop des CAN-Moduls.
- HAL_CAN_RequestSleep : Anforderung zum Übergang in den Schlafmodus.
- HAL_CAN_WakeUp : Aufwachen aus dem Schlafmodus.
- HAL_CAN_IsSleepActive : Überprüfen, ob der Schlafmodus aktiv ist.
- HAL_CAN_AddTxMessage: Eine Nachricht zu den Tx-Postfächern hinzufügen und die entsprechende Übertragungsanforderung aktivieren.
- HAL_CAN_AbortTxRequest : Übertragungsanforderung abbrechen.
- HAL_CAN_GetTxMailboxesFreeLevel : Rückgabe der Anzahl der Level freier Tx Postfächer.
- HAL_CAN_IsTxMessagePending : Überprüfen, ob eine Übertragungsanforderung für das ausgewählte Tx-Postfach aussteht.

- HAL_CAN_GetRxMessage : Einen CAN-Rahmen aus dem Rx FIFO auslesen.
- HAL_CAN_GetRxFifoFillLevel : Rückgabe des aktuellen Rx FIFO Füllstands.

6.3.3 Interrupt Funktionen

Dieser Abschnitt enthält Funktionen, die mit denen das Interrupt Verhalten der Protokolleinheit konfiguriert werden kann.

- HAL_CAN_ActivateNotification: Interrupts aktivieren.
- HAL_CAN_DeactivateNotification: Interrupts deaktivieren.
- HAL_CAN_IRQHandler : Verarbeitet CAN-Interrupt-Anforderung.

6.3.4 Periphere Zustands-und Fehlerfunktionen

Dieser Abschnitt enthält Funktionen, für die Fehlerfallbehandlung.

- HAL_CAN_GetState() : Gibt den CAN-Status zurück.
- HAL_CAN_GetError() : Gibt ggf. CAN-Fehlercodes zurück.
- HAL_CAN_ResetError() : Setzt die CAN-Fehlercodes zurück, falls vorhanden.

6.4 Programmierung der CAN Datakommunikation

Die Hauptfunktion des Programms ist die Implementierung einer UART-zu-CAN-Kommunikationsbrücke

Das Ziel ist eine Nachricht mit CAN Konfiguration Daten über die UART-Schnittstelle zu empfangen, die Konfigurationsdaten auszulesen und die Nachricht dem CAN-Controller zu übergeben, der diese Nachricht über den CAN Bus versendet.

Das Programm ermöglicht die bidirektionale Kommunikation zwischen einem PC und dem Mikrocontroller über eine UART-Schnittstelle sowie die Weiterleitung von Daten zwischen der UART- und der CAN-Schnittstelle. Hier ist eine Übersicht über die Funktionalität der Programmkomponenten.

Zuerst werden die vom STM32CubeMx automatisch generierten Initialisierungen dargestellt:

➢ Initialisierung der Peripherie:

```
/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_CAN1_Init(void);
static void MX_USART2_UART_Init(void);
```

Quellcode 6.6: Initialisierte Peripherien

- Initialisierung der UART-Schnittstelle:

Die Konfiguration der UART-Schnittstelle erfolgt hauptsächlich in der Funktion `MX_USART2_UART_Init`. Hierbei handelt es sich um den Teil des Codes, in dem die UART-Hardware des Mikrocontrollers initialisiert und verschiedene Parameter für die UART-Kommunikation festgelegt werden. Im Folgenden ist den Code-Ausschnitt, welcher sich mit der Konfiguration der UART-Schnittstelle befasst:

```
static void MX_USART2_UART_Init(void)
{
 /* USER CODE BEGIN USART2_Init 0 */
 /* USER CODE END USART2_Init 0 */
 /* USER CODE BEGIN USART2_Init 1 */
 /* USER CODE END USART2_Init 1 */
 huart2.Instance = USART2;                                          1
 huart2.Init.BaudRate = 115200;                                     2
 huart2.Init.WordLength = UART_WORDLENGTH_8B;                       3
 huart2.Init.StopBits = UART_STOPBITS_1;                            4
 huart2.Init.Parity = UART_PARITY_NONE;                             5
 huart2.Init.Mode = UART_MODE_TX_RX;                                6
 huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;                       7
 huart2.Init.OverSampling = UART_OVERSAMPLING_16;                   8
 huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;          9
 huart2.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;      10
 if (HAL_UART_Init(&huart2) != HAL_OK)                              11
 {
  Error_Handler();
 }
 /* USER CODE BEGIN USART2_Init 2 */
```

Quellcode 6.7: Initialisierte Konfiguration der UART-Schnittstelle

Hier ist eine Erläuterung der einzelnen Konfigurationselemente:

1. huart2.Instance = USART2: Dieser Code weist der UART-Konfigurationsstruktur (huart2) die Verwendung von USART2 zu. Dieser Schritt stellt sicher, dass die Konfiguration auf die richtige Hardware-Schnittstelle angewendet wird.

2. huart2.Init.BaudRate = 115200: Hier wird die Baudrate für die UART-Kommunikation festgelegt. Die Baudrate bestimmt, wie viele Bits pro Sekunde übertragen werden. In diesem Fall wird eine Baudrate von 115200 Bits pro Sekunde verwendet.

3. huart2.Init.WordLength = UART_WORDLENGTH_8B: Dieser Code legt die Wortlänge für die Kommunikation fest. UART_WORDLENGTH_8B steht für 8 Datenbits pro Zeichen, was ein gebräuchliches Format ist.

4. huart2.Init.StopBits = UART_STOPBITS_1: Hier wird die Anzahl der Stopbits pro Zeichen festgelegt. UART_STOPBITS_1 bedeutet, dass ein Stopbit pro Zeichen verwendet wird. Dies ist ebenfalls eine gängige Konfiguration.

5. huart2.Init.Parity = UART_PARITY_NONE: Dieser Code legt fest, ob Paritätsbits für Fehlererkennung und -korrektur verwendet werden. In diesem Fall wird keine Paritätsbit-Überprüfung aktiviert.
6. huart2.Init.Mode = UART_MODE_TX_RX: Hier wird der Modus für die UART-Kommunikation festgelegt. UART_MODE_TX_RX bedeutet, dass die Schnittstelle sowohl für den Empfang (RX) als auch für die Übertragung (TX) von Daten konfiguriert wird.
7. huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE: Dieser Code legt fest, ob Hardware-Flusssteuerung aktiviert ist. Hier wird keine Hardware-Flusssteuerung verwendet (UART_HWCONTROL_NONE).
8. huart2.Init.OverSampling = UART_OVERSAMPLING_16: Dieser Parameter bestimmt die Oversampling-Rate. Hier wird eine Oversampling-Rate von 16 verwendet, was für die meisten Anwendungen geeignet ist.
9. huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE: Dieser Code deaktiviert die Ein-Bit-Abtastung, die in einigen Konfigurationen verwendet werden kann.
10. huart2.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT: Hier werden erweiterte Funktionen der UART-Schnittstelle deaktiviert. Die erweiterten Funktionen werden in diesem Fall nicht verwendet.
11. Schließlich wird die UART-Schnittstelle mithilfe von HAL_UART_Init(&huart2) initialisiert. Wenn die Initialisierung erfolgreich ist, wird die Schnittstelle für die Kommunikation bereit sein. Andernfalls wird die Funktion Error_Handler() aufgerufen, um einen Fehler zu behandeln.

Zusammengefasst konfiguriert diese Funktion die UART-Schnittstelle für die serielle Kommunikation mit einer bestimmten Baudrate, Wortlänge, Stopbits, Parität und anderen Parametern, je nach den Anforderungen der Anwendung.

➢ Initialisierung der CAN-Schnittstelle:

Die Funktion `MX_CAN1_Init` initialisiert die CAN (Controller Area Network)-Schnittstelle für den CAN1-Peripheriekanal auf einem STM32-Mikrocontroller.

```
static void MX_CAN1_Init(void)
{
 /* USER CODE BEGIN CAN1_Init 0 */
 /* USER CODE END CAN1_Init 0 */
 /* USER CODE BEGIN CAN1_Init 1 */
 /* USER CODE END CAN1_Init 1 */
 hcan1.Instance = CAN1;
 hcan1.Init.Prescaler = 10;
 hcan1.Init.Mode = CAN_MODE_NORMAL; //CAN_MODE_NORMAL CAN_MODE_LOOPBACK
 hcan1.Init.SyncJumpWidth = CAN_SJW_3TQ;
 hcan1.Init.TimeSeg1 = CAN_BS1_4TQ;
 hcan1.Init.TimeSeg2 = CAN_BS2_5TQ;
 hcan1.Init.TimeTriggeredMode = DISABLE;
 hcan1.Init.AutoBusOff = DISABLE;
 hcan1.Init.AutoWakeUp = DISABLE;
 hcan1.Init.AutoRetransmission = DISABLE;
 hcan1.Init.ReceiveFifoLocked = DISABLE;
 hcan1.Init.TransmitFifoPriority = DISABLE;
 if (HAL_CAN_Init(&hcan1) != HAL_OK)
 {
   Error_Handler();
  }
 /* USER CODE BEGIN CAN1_Init 2 */
 /* USER CODE END CAN1_Init 2 */
}
```

Quellcode 6.8: Konfiguration vom CAN_InitTypeDeff

Hier ist eine Erläuterung der einzelnen Konfigurationsparameter:

1. **hcan1.Instance:** Hier wird die Peripherieinstanz festgelegt, die auf CAN1 verweist. CAN1 ist einer der CAN-Controller auf dem STM32-Mikrocontroller.

2. **hcan1.Init.Prescaler:** Der Prescaler ist ein wichtiger Parameter, der die Baudrate des CAN-Bus bestimmt. Ein niedrigerer Wert bedeutet eine höhere Baudrate. In diesem Fall beträgt der Prescaler 10, was die Baudrate des CAN-Bus festlegt.

3. **hcan1.Init.Mode:** Hier wird der Modus der CAN-Schnittstelle festgelegt. Mit `CAN_MODE_NORMAL` ist die CAN-Schnittstelle im normalen Modus, bei dem sie Daten auf dem CAN-Bus sendet und empfängt. Alternativ kann `CAN_MODE_LOOPBACK` verwendet werden, um eine Schleifenrückkopplung zu aktivieren, bei der gesendete Daten auch empfangen werden, ohne den physischen Bus zu nutzen.

4. **hcan1.Init.SyncJumpWidth:** Dieser Parameter bestimmt die Länge der Synchronisierungssequenz (Sync Jump Width) in Time Quanta (TQ). SJW ist ein Timing-Parame-

ter, der die maximale Abweichung zwischen den Synchronisationszeitpunkten für Sender und Empfänger definiert. Hier beträgt die SJW 3 Taktzyklen.

5. **hcan1.Init.TimeSeg1:** Dieser Parameter legt die Anzahl der Taktzyklen für den ersten Time Segment 1 fest. TSeg1 ist die Zeit, in der der CAN-Controller die dominante (Low) Phase des CAN-Signals beibehält. In diesem Fall beträgt TSeg1 4 Taktzyklen.
6. **hcan1.Init.TimeSeg2:** Dieser Parameter legt die Länge des TimeSeg2 in TQ fest. TSeg2 ist die Zeit, in der der CAN-Controller die rezessive (High) Phase des CAN-Signals beibehält. Hier beträgt TSeg2 5 Taktzyklen.
7. **hcan1.Init.TimeTriggeredMode:** Mit diesem Parameter kann der time-triggered Modus aktiviert oder deaktiviert werden. In diesem Fall ist er deaktiviert (DISABLE).
8. **hcan1.Init.AutoBusOff:** Dieser Parameter legt fest, ob der Bus-Off-Modus (bei einem Busfehler) automatisch aktiviert wird. Hier ist er deaktiviert (DISABLE), was bedeutet, dass im Fehlerfall der Bus-Off-Modus nicht automatisch aktiviert wird.
9. **hcan1.Init.AutoWakeUp:** Mit diesem Parameter kann die automatische Aufweckfunktion aktiviert oder deaktiviert werden. Hier ist sie deaktiviert (DISABLE).
10. **hcan1.Init.AutoRetransmission:** Dieser Parameter legt fest, ob die automatische Wiederholung von Nachrichten aktiviert ist. Mit DISABLE ist die automatische Wiederholung deaktiviert.
11. **hcan1.Init.ReceiveFifoLocked:** Dieser Parameter legt fest, ob das Empfangs-FIFO gesperrt ist. Hier ist es deaktiviert (`DISABLE`).
12. **hcan1.Init.TransmitFifoPriority:** Hier wird die Priorität der Übertragungsfifo konfiguriert. In diesem Fall ist sie deaktiviert (`DISABLE`), was bedeutet, dass keine Priorität für die Übertragung festgelegt ist.

Schließlich wird mit HAL_CAN_Init(&hcan1) die CAN-Konfiguration mit den oben festgelegten Parametern initialisiert. Falls bei der Initialisierung ein Fehler auftritt, wird die Funktion Error_Handler() aufgerufen, um den Fehler zu behandeln.

- CAN Filter Configuration:

Im folgenden Programmabschnitt 6.9 wird eine Instanz der Struktur CAN_FilterTypeDef namens canfilterconfig erstellt und anschließend werden die diversen Filterparameter gesetzt. Dies sind die Parameter, die zur Konfiguration der CAN-Filter verwendet werden, darunter die Filteraktivierung, die Filterbank, die FIFO-Zuordnung, die Filter- und Masken-ID, der Filtermodus und die Skalierung des Filters. Die Struktur wird dann an die Funktion HAL_CAN_ConfigFilter übergeben, um den Filter im CAN-Controller zu konfigurieren.

```
  /* USER CODE BEGIN USART2_Init 2 */
  // CAN Filter Configuration
     CAN_FilterTypeDef canfilterconfig;

     canfilterconfig.FilterActivation=CAN_FILTER_ENABLE;//CAN_FILTER_DI-
SABLE or CAN_FILTER_ENABLE
     canfilterconfig.FilterBank=10;
     canfilterconfig.FilterFIFOAssignment=CAN_RX_FIFO0;
     canfilterconfig.FilterIdHigh=0;//0x100<<5
     canfilterconfig.FilterIdLow=0x0000;
     canfilterconfig.FilterMaskIdHigh=0;//0x100<<5
     canfilterconfig.FilterMaskIdLow=0x0000;
     canfilterconfig.FilterMode=CAN_FILTERMODE_IDMASK;
     canfilterconfig.FilterScale=CAN_FILTERSCALE_32BIT;
     canfilterconfig.SlaveStartFilterBank=0;

     //calling the filter configuration function
     HAL_CAN_ConfigFilter(&hcan1, &canfilterconfig);
  /* USER CODE END USART2_Init 2 */
}
```

Quellcode 6.9: CAN-Filterkonfiguration

➢ Initialisierungen der CAN Filter, der CAN Protokolleinheit und des Interruptverhaltens

```
/* USER CODE END 2 */
        HAL_CAN_ConfigFilter(&hcan1,&canfil); //Initialisierung CAN Filter
        HAL_CAN_Start(&hcan1); //Initialisierung CAN Bus
        HAL_CAN_ActivateNotification(&hcan1,CAN_IT_RX_FIFO0_MSG_PENDING);// Initialisierung CAN Bus Rx Interr
```

Quellcode 6.10: Initialisierung zur Erstellung der CAN_Kommunikation

Hierbei werden die folgenden Funktion auf die beschriebene Weise verwendet.

- HAL_CAN_ConfigFilter:

Diese Funktion wird zur Konfiguration des CAN-Empfangsfilters entsprechend den Parametern , welche in der CAN_FilterInitStruct Datenstruktur gesetzt worden sind, verwendet
&hcan1 ist der Zeiger auf eine CAN_HandleTypeDef-Struktur, welche die Konfigurationsinformationen für die angegebenen CAN Protokolleinheit enthält.
&canfilter ist der Zeiger auf eine CAN_FilterTypeDef-Struktur, welche die Filterkonfigurationsinformationen enthält.

- HAL_CAN_Start: Dient zum Starten des CAN-Moduls.

&hcan1 ist hierbei der Zeiger auf eine CAN_HandleTypeDef-Struktur, welche die Konfigurationsinformationen für die angegebenen CAN Protokolleinheit enthält.

- HAL_CAN_ActivateNotification: Dient zur Aktivierung des Interrupts.

&hcan1 ist hierbei der Zeiger auf eine CAN_HandleTypeDef-Struktur, welche die Konfigurationsinformationen für den angegebenen CAN enthält.
CAN_IT_RX_FIFO0_MSG_PENDING gibt an, welche Interrupts aktiviert werden. Hier wird nur dieser spezifische CAN-Interrupt aktiviert, um den Empfang von CAN-Nachrichten im CAN_RX_FIFO0 zu verarbeiten.

- Code zum UART Daten Empfang in Interrupt Mode:

Dieser Code-Abschnitt 6.11 behandelt die Verarbeitung von empfangenen UART-Daten. Die Funktion beginnt damit, das empfangene Datenbyte UART_RxData[0] zu überprüfen, um festzustellen, ob es sich um Daten für eine CAN-Übertragung oder um Konfigurationsdaten für die CAN-Schnittstelle handelt.

- Wenn `UART_RxData[0]` gleich 2 ist, handelt es sich um Daten für eine CAN-Übertragung. Es werden die CAN-Header-Informationen entsprechend festgelegt, ein Datenpuffer vorbereitet und die Daten über die CAN-Schnittstelle gesendet.
- Wenn `UART_RxData[0]` gleich 1 ist, handelt es sich um Konfigurationsdaten für die CAN-Schnittstelle. Verschiedene Parameter wie der Prescaler, der Betriebsmodus und die Zeitsegmente werden entsprechend eingestellt.

Unabhängig davon, ob es sich um Daten oder Konfigurationsanforderungen handelt, wird durch eine entsprechende Funktion der UART-Receive-Interrupt erneut aktiviert, um auf weitere UART-Daten reagieren zu können Dies ermöglicht die kontinuierliche Überwachung der UART-Schnittstelle für eingehende Daten oder Konfigurationsänderungen.

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
         // Hier beginnt der Code, der in der UART Receive Callback-Funktion
ausgeführt wird.
    if (UART_RxData[0] == 2) {
        // Wenn das erste Byte der empfangenen Daten gleich 2 ist, handelt es
sich um Daten, die über CAN gesendet werden sollen.
        // Hier werden die Header-Informationen für die CAN-Nachricht festgelegt.
         TxHeader.DLC = UART_RxData[2];
         TxHeader.IDE = (UART_RxData[13] == 1) ? CAN_ID_STD : CAN_ID_EXT;
         if (TxHeader.IDE == CAN_ID_STD) {
             TxHeader.StdId = UART_RxData[1];
         } else {
             TxHeader.ExtId = UART_RxData[1];
         }
         TxHeader.RTR =(UART_RxData[11] == 1)?CAN_RTR_DATA:CAN_RTR_REMOTE;
         TxHeader.TransmitGlobalTime = (UART_RxData[12] == 1) ? ENABLE :
DISABLE;
        // Hier wird ein Datenpuffer für die zu übertragenden Daten erstellt
und mit den empfangenen Daten gefüllt.
         uint8_t *Data = malloc(UART_RxData[2] * sizeof(uint8_t));
         for (int i = 0; i < UART_RxData[2]; i++) {
             Data[i] = UART_RxData[i + 3];
         }
        // Die vorbereiteten Daten werden über die CAN-Schnittstelle gesendet.
         HAL_CAN_AddTxMessage(&hcan1, &TxHeader, Data, &TxMailbox);
    } else if (UART_RxData[0] == 1) {
         // Wenn das erste Byte der empfangenen Daten gleich 1 ist, handelt es
sich um eine Konfigurationsanforderung für die CAN-Schnittstelle.
        // Hier werden verschiedene CAN-Konfigurationsparameter basierend auf
den empfangenen Daten eingestellt.
         hcan1.Init.Prescaler = UART_RxData[1];
         CAN_Mode(UART_RxData[2]);
         CAN_SJW(UART_RxData[3]);
         CAN_TSeg1(UART_RxData[4]);
         CAN_TSeg2(UART_RxData[5]);
    }
    // Hier wird die UART-Receive-Interrupt-Funktion erneut aktiviert, um auf
weitere UART-Daten zu warten.
    HAL_UART_Receive_IT(&huart2, UART_RxData, sizeof(UART_RxData));
    // Hier endet die Callback-Funktion.
}
```

Quellcode 6.11: UART Daten Empfang im Interrupt Mode

- Empfang von Daten über die CAN-Schnittstelle und deren Übertragung über die UART-Schnittstelle:

```
void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan)
{
    uint8_t *CAN_RxData = malloc((UART_RxData[1]) * sizeof(uint8_t));
    HAL_CAN_GetRxMessage(hcan, CAN_RX_FIFO0, &RxHeader, CAN_RxData);
    CAN_RxData[3] = 0xa1;
    HAL_UART_Transmit(&huart2, CAN_RxData, 8, 1000);//sizeof(CAN_RxData)
    HAL_UART_Receive_IT(&huart2, UART_RxData, sizeof(UART_RxData));
}
```

Quellcode 6.12: Empfang und Übertragung von Daten

In dieser Callback-Funktion wird der folgende Ablauf durchgeführt:

1. HAL_CAN_RxFifo0MsgPendingCallback wird aufgerufen, wenn eine CAN-Nachricht im CAN_RX_FIFO0 empfangen wird.
2. Es wird ein dynamisch Speicherplatz reserviert (allokiert), um ein Array namens CAN_RxData zu erstellen
3. Mit HAL_CAN_GetRxMessage wird die empfangene CAN-Nachricht in CAN_RxData kopiert.
4. Die Zeile CAN_RxData[3] = 0xa1; ändert die Daten in CAN_RxData. In diesem Fall wird das vierte Byte der empfangenen CAN-Nachricht auf den Wert 0xa1 gesetzt. Das Setzen des vierten Bytes auf `0xa1` dient als Prüfwert , um zu überprüfen, ob die gesendeten Nachrichten erfolgreich empfangen wurden.
5. HAL_UART_Transmit wird verwendet, um die modifizierten CAN-Daten über die UART-Schnittstelle zu senden.
6. HAL_UART_Receive_IT(&huart2, UART_RxData, sizeof(UART_RxData)) Nachdem die Daten übertragen wurden, wird der Empfangsmodus für die UART-Schnittstelle erneut im Interrupt-Modus gestartet. Dies ermöglicht der UART-Schnittstelle, weitere Daten zu empfangen und in UART_RxData zu speichern, um sie für zukünftige Verarbeitungsschritte bereitzuhalten.

Zusammengefasst nimmt dieser Abschnitt empfangene CAN-Nachrichten entgegen, bearbeitet sie, indem er den Wert eines Datenbytes ändert, und leitet die modifizierten Daten über die UART-Schnittstelle an ein anderes Gerät oder einen anderen Mikrocontroller weiter.

Kapitel 7

Benutzeroberfläche für das Senden und Empfangen von CAN-Nachrichten

In diesem Kapitel wird die Gestaltung und Programmierung einer grafischen Benutzeroberfläche beschrieben. Mit dieser kann der Nutzer am PC eine CAN-Nachricht eingeben, konfigurieren und die Versendung über die USB Schnittstelle auslösen. Gleichzeitig können Nachrichten, die über die USB Schnittstelle empfangen werden, angezeigt werden.

Zunächst wird eine graphische Benutzerfläche (GUI) im Qt-Creator entworfen und anschließend deren Bedienelemente programmiert.

7.1 Aufbau und Funktion der graphischen Benutzeroberfläche

Die folgende gestaltete GUI, welche in Abbildung 7.1 zu sehen ist, gliedert sich in sechs Bereiche. Diese sind nach der Reihenfolge ihrer Nutzung für das Senden einer Nachricht nummeriert. Im Einzelnen haben die Bedienelemente der Bereiche folgende Funktion.

Bereich 1: Konfiguration der seriellen Schnittstellen
Bereich 2: Konfiguration der CAN Rahmen
Bereich 3: Konfiguration des CAN-Busses
Bereich 4: Eingabe und Konfiguration der zu sendenden CAN-Nachricht
Bereich 5: Steuerung der Oberfläche
Bereich 6: Anzeige gesendeter und empfangener Nachrichten

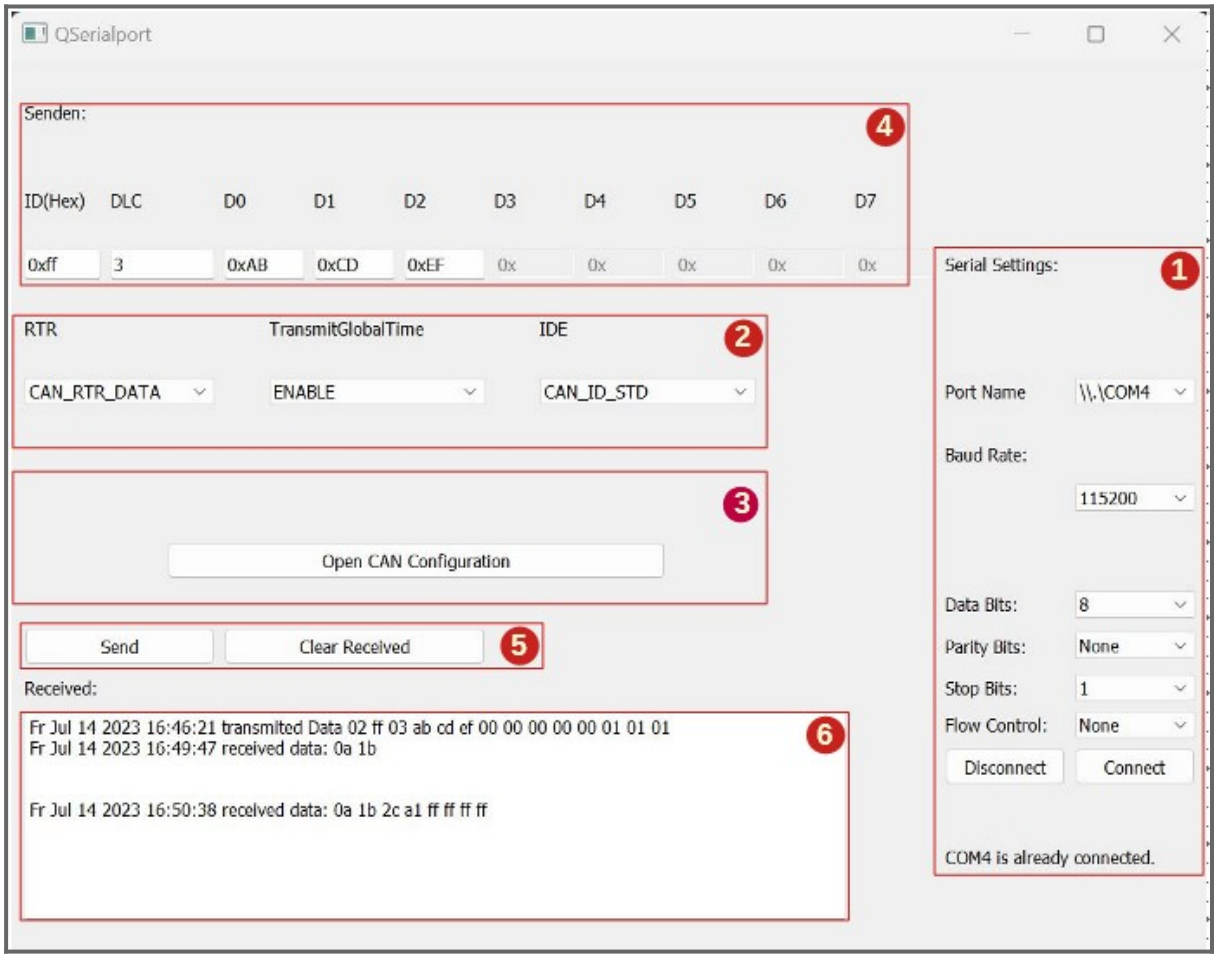


Abbildung 7.1: Design der Benutzeroberfläche

- **Der Bereich 1** enthält insgesamt 6 Bedienelemente für die Konfiguration der seriellen-COM Schnittstelle. Dies sind sämtlich als Drop-Down-Listen ausgeführt, d. h. der Nutzer hat die Möglichkeit aus einer ausklappbaren Liste mit vorgegeben Werte eine Auswahl zu treffen.

In dargestellten Beispiel wird die folgenden Konfiguration für die zu sendende Nachricht verwendet: ID: 0FF, DLC:3, D0:AB, D1:CD, D2:EF

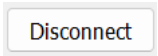
Tabelle 7.1: Beschreibung der Bedienelemente (Combobox) im Bereich 1

Feldbzeichnung	Funktion	Auswahlwerte
Port-Name	Auswahl des Ports für die COM-Schnittstelle	Je nach Anschluss
Baud-Rate	Festlegung der Symbolrate in Baud	9600, 19200, 38400, 115200
Data-Bits	Festlegung der Anzahl an Data-Bits	5, 6, 7, 8
Parity Bits	Festlegung der Parity-Bits	None, Even, Odd, Mark, Space
Stop-Bits	Festlegung der Länge des Stop-Bits	1, 1.5, 2
Flow Control	Festlegung der Konfiguration für Flow Control	None , RTS/CTS , XON/XOFF

In der UART-Konfiguration wird eine Baudrate von 115200 verwendet. Dies entspricht einer etablierten Standardgeschwindigkeit in der seriellen Datenkommunikation, wobei 115200 Baud im Wesentlichen bedeutet, dass 115200 Bits pro Sekunde übertragen werden. Außerdem ist die Datenwortlänge auf 8 Bit festgelegt. Das bedeutet, dass jedes gesendete oder empfangene Datenpaket aus 8 Bit besteht, eine Größe, die im allgemeinen Sprachgebrauch als Byte bezeichnet wird.

Schließlich wird die Anzahl der Stopbits auf eins festgelegt. Stopbits werden zur Synchronisation des Datenstroms verwendet. Sie signalisieren das Ende der Übertragung eines Bytes, indem nach jedem gesendeten Byte ein einzelnes Stopbit gesendet wird. Auf diese Weise kann die Empfangsseite ein Byte vom nächsten unterscheiden, was die Integrität der übertragenen Daten sicherstellt.

Der Bereich 1 enthält auch die 2 Bedienelemente (Push Button):

	Connect: Nach Festlegung der seriellen Einstellungen, dient der Schaltknopf zur Etablierung der Verbindung des Mikrocontrollers mit dem PC, um den Prozess der Übertragung und Empfang der Daten zu beginnen.
	Disconnect: Dient zur Trennung der Verbindung des Mikrocontrollers mit dem PC, d.h. können kein Empfang oder Versendung von Daten erfolgen.

- **Der Bereich 2** enthält insgesamt 3 Bedienelemente für die Konfiguration der CAN Rahmen. Diese Bedienelemente sind sämtlich als Drop-Down-Listen ausgeführt.

Feldbzeichnung	Funktion	Auswahlwerte
RTR	Gibt den Frametyp der Nachricht an, die übertragen wird	CAN_RTR_DATA CAN_RTR_REMOTE
TransmitGlobalTime	Aktivierung oder Deaktivierung des zeitgesteuerten Kommunikationsmodus	Enable/Disable
IDE	Gibt der Standard oder Extended Identifier an	CAN_ID_STD CAN_ID_EXT

- **Bereich 3**: Nach Klick auf Open Can Configuration öffnet das in Abbildung 7.2 dargestellte Fenster. In diesem Fenster werden neben den seriellen UART-Einstellungen auch Konfigurationsparameter des CAN-Controllers in Form von Combo-Boxen dargestellt, die vom Benutzer ausgewählt werden können.

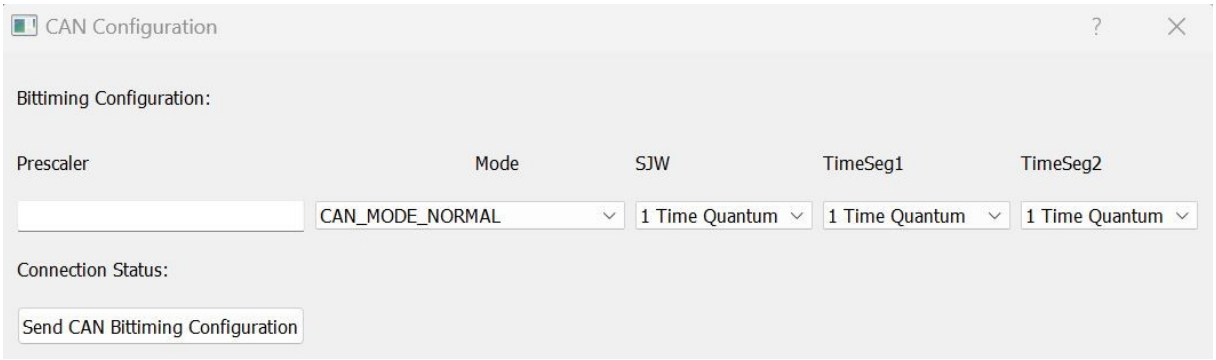


Abbildung 7.2: CAN Konfigurationsparameter

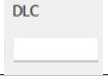
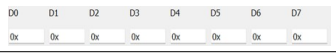
Tabelle 7.2: Beschreibung der Bedienelemente (Combobox) im Bereich 3

Bedienelement	**Funktion**	**Werte**
Presacler	Gibt die Länge der Zeitquanten an	Anzahl Zwischen 1 und 1024
Mode	Dieser Parameter kann einen der 4 Konstanten der Definition CAN_Operating_Mode annehmen	CAN_MODE_NORMAL CAN_MODE_LOOPBACK CAN_MODE_SILENT CAN_MODE_SILENT_LOOPBACK
SJW-Wert	Legt die maximale Synchronisations-sprungweite fest	Von 1 TimeQuantum bis 4 TimeQuantum
TimeSeg1	Gibt die Anzahl der Zeit-Quanten der entsprechenden Bitzeitsegmente an an	Von 1 TimeQuantum bis 10 TimeQuantum
TimeSeg2		Von 1 TimeQuantum bis 8 TimeQuantum

- **Bereich 4:** Eingabe der Nachrichten

Diese Steuerelemente ermöglicht den Benutzern, CAN-Nachrichtenparameter wie ID, Datenlänge und Datenbytes einzugeben.

Tabelle 7.3: Eingabe der Nachrichten

Steuerelement	**Funktion und Werte**
	Dieses Eingabefeld erwartet eine hexadezimale CAN-Bus-Nachrichten-ID
	Gibt die Anzahl der Datenbytes in einer CAN-Nachricht an . Dieser Parameter muss eine Zahl zwischen Min_Data = 1 und Max_Data = 8 sein.
	Diese Eingabefelder sind für die Datenbytes einer CAN-Nachricht vorgesehen `.0x[0-9a-fA-F][0-9a-fA-F]`

- **Bereich 5:** Senden der eingegebenen Nachrichten und Löschen von empfangenen Nachrichten

Tabelle 7.4: Steuerung der Oberfläche

Steuerelement	Elementtyp	Funktion und Werte
Send	Push Button	Send_command: Dient zum Senden der eingegebenen Nachricht.
Clear Received	Push Button	Clear_Console: Dient zum Löschen der empfangenen Nachrichten in dem Empfangsbereich.

- **Bereich 6**:Anzeige gesendeter und empfangener Nachrichten.

Im Feld "Received" werden die übertragenen und empfangenen Daten durch den Mikrocontroller aufgelistet, wobei auch das Datum und die Uhrzeit angegeben werden.
In dargestellten Beispiel wird die folgenden Nachrichten im Bereich 6 angezeigt:

empfangene Nachrichten: 0FF, DLC:3, D0:AB, D1:CD, D2:EF

empfangene Nachrichten: ID: 0FF, DLC:2, D0:0A, D1:1B, D2:A1

7.2 Programmierung der GUI im Qt Creator

Nach der Anpassung der Namen und Eigenschaften aller Objekte auf der graphischen Oberfläche wird mit der Erstellung des Codes im Editiermodus begonnen.

Um die serielle Kommunikation zu realisieren, muss die Bibliothek für serielle Schnittstellen „QSerialPort“ eingebunden werden, in dem die folgende Codezeile in die *.pro-Konfigurationsdatei des Projekts eingefügt wird (siehe Quellcode 7.1).

```
QT       += core gui serialport
```

Quellcode 7.1: Verwendung der QSerialPort Bibliothek

Durch die Verwendung dieser Klasse können die Grundfunktionen der seriellen Schnittstelle genutzt werden. Dazu gehören das Öffnen, das Schreiben, das Lesen und das Schließen der seriellen Schnittstelle. Um die Funktionen der Bibliothek verwenden zu können, müssen die folgenden beiden Dateien inkludiert werden.

```
#include <QtSerialPort/QSerialPort>
#include <QtSerialPort/QSerialPortInfo>
```

Quellcode 7.2: Header Dateien in der Mainwindow.cpp

Im folgenden Quellcode wird ein Objekt der Klasse „QSerialPort“ für die serielle Kommunikation über eine serielle Schnittstelle erstellt und der Instanzvariable „m_serial“ in der aktuellen Klasse zugewiesen. Dies ermöglicht es dem Programm, auf die serielle Schnittstelle zuzugreifen und damit zu kommunizieren.

```
m_serial(new QSerialPort(this))
```

Quellcode 7.3:Erstellung eines Objekts für eine serielle Schnittstelle

Im Programm wir das QSerialPort-Signal readyRead()verwendet, das anzeigt, dass neue Daten empfangen wurden und somit verfügbar sind. Um die Verbindung zwischen einem Signal und einem Slot zu realisieren, wird die folgende Qt Connect-Funktion aufgerufen:

```
connect(this->m_serial, &QSerialPort::readyRead, this, &MainWindow::readData);
```

Quellcode 7.4: Syntax einer Qt Connect-Funktion

Der folgende Quellcode wird verwendet, um Felder in der GUI zur Eingabe der CAN ID, des Längen-Code DLC,und der acht Nutzdaten Byte zu erstellen. Die Validierungsfunktionen stellen sicher, dass nur gültige Eingaben akzeptiert werden, was die Datenintegrität und die korrekte Funktion des Programms gewährleistet.

- [0-9a-fA-F][0-9a-fA-F]: Durch diese Ausdrücke wird festgelegt, dass zwei aufeinander folgende hexadezimale Zahlen erwartet werden, die ein Byte definieren.

```
//ui->comboBox_port_name->event
//allows only Hex in the LineEdit

ui->lineEdit_ID->setValidator(new QregExpValidator(QRegExp("0x[0-9a-fA-F]
                                [0-9a-fA-F][0-9a-fA-F]"),this));
ui->lineEdit_DLC->setValidator(new QRegExpValidator(QRegExp("[1-8]"),this));
ui->lineEdit_D0->setValidator(new QregExpValidator(QRegExp("0x[0-9a-fA-F]
                                [0-9a-fA-F]"),this));
ui->lineEdit_D1->setValidator(new QregExpValidator(QRegExp("0x[0-9a-fA-F]
                                [0-9a-fA-F]"),this));
ui->lineEdit_D2->setValidator(new QregExpValidator(QRegExp("0x[0-9a-fA-F]
                                [0-9a-fA-F]"),this));
ui->lineEdit_D3->setValidator(new QregExpValidator(QRegExp("0x[0-9a-fA-F]
                                [0-9a-fA-F]"),this));
ui->lineEdit_D4->setValidator(new QregExpValidator(QRegExp("0x[0-9a-fA-F]
                                [0-9a-fA-F]"),this));
ui->lineEdit_D5->setValidator(new QregExpValidator(QRegExp("0x[0-9a-fA-F]
                                [0-9a-fA-F]"),this));
ui->lineEdit_D6->setValidator(new QregExpValidator(QRegExp("0x[0-9a-fA-F]
                                [0-9a-fA-F]"),this));
ui->lineEdit_D7->setValidator(new QregExpValidator(QRegExp("0x[0-9a-fA-F]
                                [0-9a-fA-F]"),this));
```

Quellcode 7.5: Erstellung von ID,DLC und die 8 Byte

Im folgenden wird eine Combo-Box erstellt, mit der dem Nutzer eine Auswahl von Baudraten für die Kommunikation über die serielle Schnittstelle anzubieten.

- Baudrate:

Im Codeabschnitt wird die Baudrate "115200" als Defaulteinstellung festgelegt. Dies erfolgt mittels der Methode setCurrentIndex(3) des comboBox_baud_rate, wodurch der Eintrag mit dem Index 3 ausgewählt wird.

```
//ui->comboBox_port_name->addItem(QStringLiteral("COM_Test"));
ui->comboBox_baud_rate->addItem(QStringLiteral("9600"));
ui->comboBox_baud_rate->itemData(ui->comboBox_baud_rate->count()-1, 0) =
                                                    QSerialPort::Baud9600;
ui->comboBox_baud_rate->addItem(QStringLiteral("19200"));
ui->comboBox_baud_rate->itemData(ui->comboBox_baud_rate->count()-1, 0) =
                                                    QSerialPort::Baud19200;
ui->comboBox_baud_rate->addItem(QStringLiteral("38400"));
ui->comboBox_baud_rate->itemData(ui->comboBox_baud_rate->count()-1, 0) =
                                                    QSerialPort::Baud38400;
ui->comboBox_baud_rate->addItem(QStringLiteral("115200"));
ui->comboBox_baud_rate->itemData(ui->comboBox_baud_rate->count()-1, 0) =
                                                    QSerialPort::Baud115200;
ui->comboBox_baud_rate->setCurrentIndex(3);
```

Quellcode 7.6: Verschiedene Baudrate-Varianten und Setzung einer Default-Einstellung

- Data-Bits:

Im folgenden wird eine Combo-Box erstellt, mit welcher der Nutzer die Anzahl der Bits einer UART Botschaft festlegen kann. Die Standard (Defaulteinstellung) für die Anzahl der Datenbits ist in diesem Fall auf 8 Bits festgelegt.

```
ui->comboBox_data_bits->addItem(QStringLiteral("5"));
ui->comboBox_data_bits->itemData(ui->comboBox_data_bits->count()-1, 0) =
                                                    QSerialPort::Data5;
ui->comboBox_data_bits->addItem(QStringLiteral("6"));
ui->comboBox_data_bits->itemData(ui->comboBox_data_bits->count()-1, 0) =
                                                    QSerialPort::Data6;
ui->comboBox_data_bits->addItem(QStringLiteral("7"));
ui->comboBox_data_bits->itemData(ui->comboBox_data_bits->count()-1, 0) =
                                                    QSerialPort::Data7;
ui->comboBox_data_bits->addItem(QStringLiteral("8"));
ui->comboBox_data_bits->itemData(ui->comboBox_data_bits->count()-1, 0) =
                                                    QSerialPort::Data8;
ui->comboBox_data_bits->setCurrentIndex(3);
```

Quellcode 7.7: Verschiedene Data-Bits-Varianten und Setzung einer Default Einstellung

Die Einstellungen in der GUI zur seriellen Schnittstelle werden wie im folgenden Codeausschnitt ausgelesen und in der Datenstruktur m_currentSettings geschrieben.

```
void MainWindow::updateSettings()
{
    this->m_currentSettings.name = ui->comboBox_port_name->itemData(ui-
>comboBox_port_name->currentIndex(), 0).toString();

    m_currentSettings.baudRate = ui->comboBox_baud_rate->itemData(ui-
>comboBox_baud_rate->currentIndex(), 0).toInt();
    m_currentSettings.stringBaudRate = QString::number(m_currentSettings.baudRa-
te);

    m_currentSettings.dataBits = static_cast<QSerialPort::DataBits>(
                ui->comboBox_data_bits->itemData(ui->comboBox_data_bits->curren-
tIndex(), 0).toInt());
    m_currentSettings.stringDataBits =  ui->comboBox_data_bits->itemData(ui->com-
boBox_data_bits->currentIndex(), 0).toString();

    m_currentSettings.parity = static_cast<QSerialPort::Parity>(
                ui->comboBox_parity_bit->itemData((ui->comboBox_parity_bit->cur-
rentIndex(),0)).toInt());
    m_currentSettings.stringParity = ui->comboBox_parity_bit->itemData(ui->combo-
Box_parity_bit->currentIndex(), 0).toString();

    m_currentSettings.stopBits = static_cast<QSerialPort::StopBits>(
                ui->comboBox_stop_bit->itemData((ui->comboBox_stop_bit->curren-
tIndex(),0)).toInt());
    m_currentSettings.stringStopBits =  ui->comboBox_stop_bit->itemData(ui->com-
boBox_stop_bit->currentIndex(), 0).toString();

    m_currentSettings.flowControl = static_cast<QSerialPort::FlowControl>(
                ui->comboBox_flow_control->itemData((ui->comboBox_flow_control-
>currentIndex(),0)).toInt());
    m_currentSettings.stringFlowControl = ui->comboBox_flow_control->itemData(ui-
>comboBox_flow_control->currentIndex(), 0).toString();
}
```

Quellcode 7.8: Festlegung der Einstellungen der seriellen Schnittstelle

Durch Klick auf PushButton_Connect wird der openSerialPort()-Slot aufgerufen. In diesem Slot werden die GUI Einstellungen mit Hilfe der Funktion „updateSettings“ gelesen und die serielle Schnittstelle entsprechend geöffnet und initialisiert. Bei Erfolg wird in der Statusleiste eine Meldung angezeigt, dass das Öffnen mit der angegebenen Konfiguration erfolgreich war „Opened Serial Port!“; andernfalls wird eine Messagebox mit dem entsprechenden Fehlercode und der Meldung „Could not open serial port…“ (siehe Quellcode7.9) angezeigt.

```
void MainWindow::openSerialPort()
{
    closeSerialPort();
       updateSettings();
       this->m_serial->setPortName(this->m_currentSettings.name);
       this->m_serial->setBaudRate(this->m_currentSettings.baudRate);
       this->m_serial->setDataBits(this->m_currentSettings.dataBits);
       this->m_serial->setParity(this->m_currentSettings.parity);
       this->m_serial->setStopBits(this->m_currentSettings.stopBits);
       this->m_serial->setFlowControl(this->m_currentSettings.flowControl);

       qDebug() << "port name: " << this->m_currentSettings.name;
       qDebug() << "baud rate: " << this->m_currentSettings.baudRate;
       qDebug() << "data bits: " << this->m_currentSettings.dataBits;
       qDebug() << "parity: " <<  this->m_currentSettings.parity;
       qDebug() << "stop bits: " << this->m_currentSettings.stopBits;
       qDebug() << "flow control: " << this->m_currentSettings.flowControl;

       if(!m_serial->open(QIODevice::ReadWrite))
       {
           QMessageBox::critical(this, tr("Error"), m_serial->errorString());
           qDebug() << "Could not open serial port...";

       }
       else
       {
           m_serial->flush();
           qDebug() << "Opened Serial port!";
       }
}
```

Quellcode 7.9: openSerialPort() Slot

Bei Klick auf die Schaltfläche pushButton_disconnect wird der closeSerialPort() Slot aufgerufen, was zur Schließung der seriellen Schnittstelle führt (siehe Quellcode7.10).

```
void MainWindow::closeSerialPort()
{
    if(m_serial->isOpen())
    {
        m_serial->close();
        ui->label_connection_status_value->setText("Disconnected");
        qDebug() << "closeSerialPort(): Closed opened connection...";
    }
    else
    {
        qDebug() << "closeSerialPort(): No connection to close...";
    }
}
```

Quellcode 7.10: closeSerialPort() Slot

Bei Klick auf die puschButton_Send Schaltfläche, wird der on_pushButton_send_command_clicked()-Slot aufgerufen, welcher die Zeichen verarbeitet, die in das "Line Edit Command" Feld der GUI eingegeben wurden. Wenn eine valide Eingabe vorliegt, wird die eingegebene Nachricht aus der Variable„data“ mit Hilfe des plainTextEdit_sent_commands angezeigt, an-

derenfalls wird in „data“ die Meldung „FAILED TO SEND“ gespeichert und mit Hilfe des plainTextEdit_sent_commands angezeigt. (siehe Quellcode 7.11).

```
void MainWindow::on_pushButton_send_command_clicked()
{
    QString data = ui->lineEdit_command->text() + "\n";
    if(m_serial->write(data.toLocal8Bit()) < 0)
    {
        data += " - FAILED TO SEND";
        ui->plainTextEdit_sent_commands->insertPlainText(data);
    }
    else
    {
        ui->plainTextEdit_sent_commands->insertPlainText(data);
    }
    qDebug() << "Writing the following data: " << data;
}
```

Quellcode 7.11: on_pushButton_send_command_clicked() Slot

Wenn die serielle Schnittstelle neue Daten empfängt, wird das Signal readyRead() ausgelöst, welches mit dem MainWindow::readData() Slot verbunden ist. Dieser Slot liest die Daten von der seriellen Schnittstelle und zeigt sie im plainTextEdit_sent_commands Feld an (siehe Quellcode7.12).

```
void MainWindow::readData()
{
    const QByteArray data = m_serial->readAll();
    ui->plainTextEdit_sent_commands->insertPlainText(QString::fromLocal8Bit(data.data()));
}
```

Quellcode 7.12: readData() Slot

Wenn Die Schaltfläche pushButton_clear_console angeklickt wird, wird der on_pushButton_clear_console_clicked() Slot aufgerufen, was dazu führt, dass die empfangenen Nachrichten im plainTextEdit_sent_commands Feld gelöscht werden (siehe Quellcode 7.13).

```
void MainWindow::on_pushButton_clear_console_clicked()
{
    ui->plainTextEdit_sent_commands->clear();
}
```

Quellcode 7.13: on_pushButton_clear_console_clicked() Slot

Kapitel 8

Versuchsaufbau & Ergebnisse

Die Inbetriebnahme der UART-zu-CAN-Kommunikationsbrücke mithilfe des STM32L476RG ARM-Mikrocontrollers erfordert einen sorgfältigen Aufbau und die sorgfältige Integration der Hardware und Software Komponenten des Systems, um eine reibungslose und zuverlässige Datenübertragung zwischen verschiedenen Kommunikationssystemen zu ermöglichen.

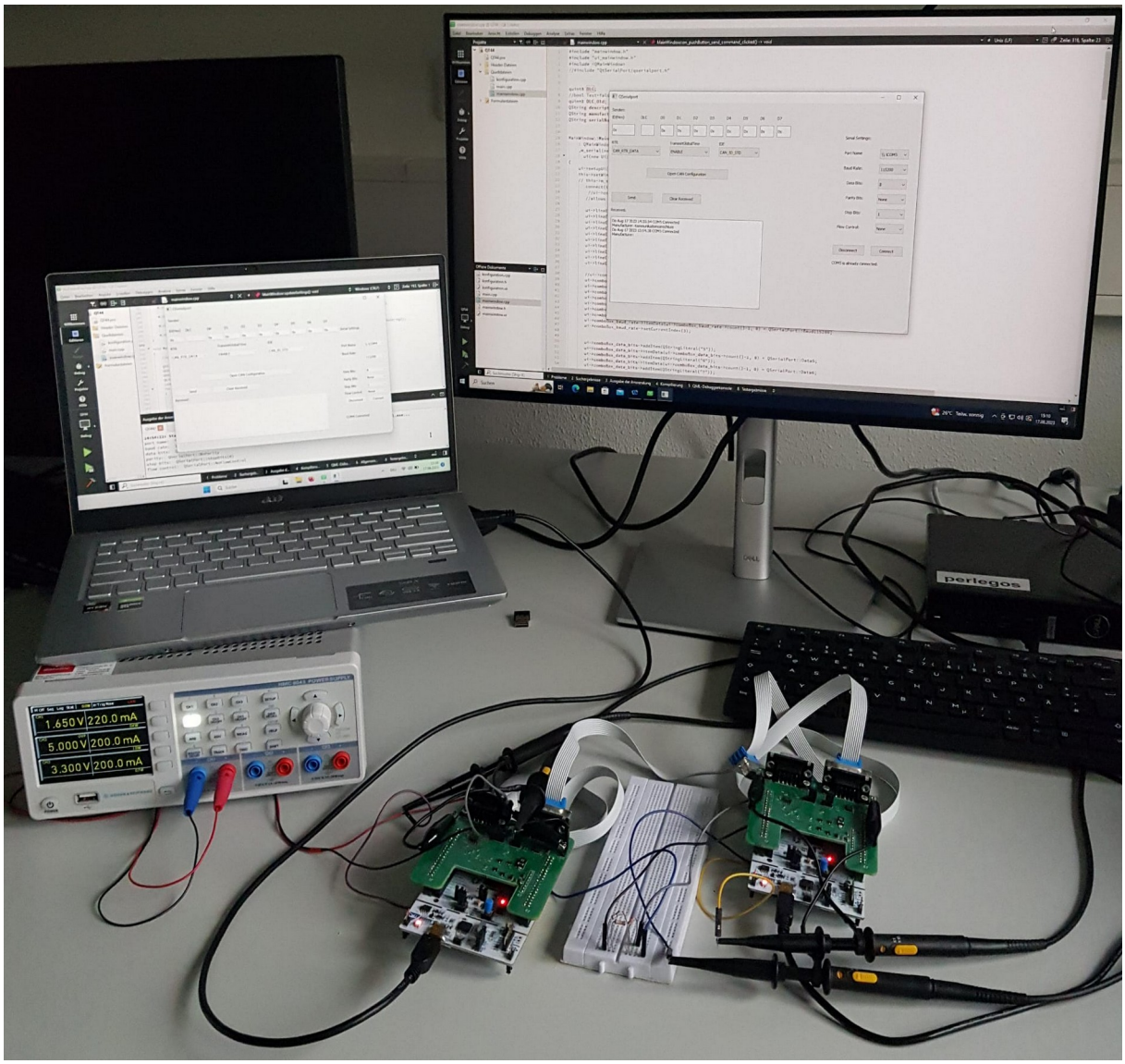


Abbildung 8.1: Aufbau der gesamten Versuch

Der folgende Abschnitt beschreibt den detaillierten Versuchsaufbau.

8.1 Versuchsaufbau

➢ **Komponentenliste**:

- 2x STM32L476RG ARM-Mikrocontroller
- 2x PCs
- 2x CAN-Transceiver (für die physische CAN-Kommunikation)[14]
- 1x 1,6 A Stromversorgung
- 1x Oszilloskop
- Verbindungskabel, Breadboard, Jumperkabel und Netzwerkkabel
- STM32CubeIDE und CubeMX für Mikrocontroller-Programmierung
- Qt Creator für die Benutzeroberflächenprogrammierung

➢ **Physische Verbindungen**:

- Die Aufsteckmodule, welche den CAN Physical Layer beherbergen, werden auf beide Mikrocontroller augesteckt. Dadurch wird unter Anderem die Verbindung der RX und TX Signale des CAN Controllers im Mikrocontroller mit dem Physical Layer etabliert.
- Die Stromversorgung (1,6 A) wird an die erforderlichen Pins des STM32L476RG Mikrocontrollers angeschlossen, um die Stromversorgung sicherzustellen.
- Die USB Anschlüsse der STM32L476RG Mikrocontroller müssen mit den USB Ports der verwenden PCs verbunden werden, um die serielle Kommunikation zwischen ihnen zu ermöglichen.
- Die CAN-High- und CAN-Low-Pins beider Aufsteckmodule werden über DSUB Stecker und Flachbandkabel miteinander verbunden. Zusätzlich muss der Abschlusswiderstand des Busses zwischen CAN-High und CAN-Low angeschlossen werden und die Busspannung für den rezessiven Buszustand über zwei hochohmige Widerstände an CAN-High bzw. CAN-Low eingekoppelt werden.
- Das Oszilloskop wird an die relevanten Pins der CAN-Bus-Leitungen angeschlossen, um die Signalqualität und -integrität während des Versuchs zu überwachen.

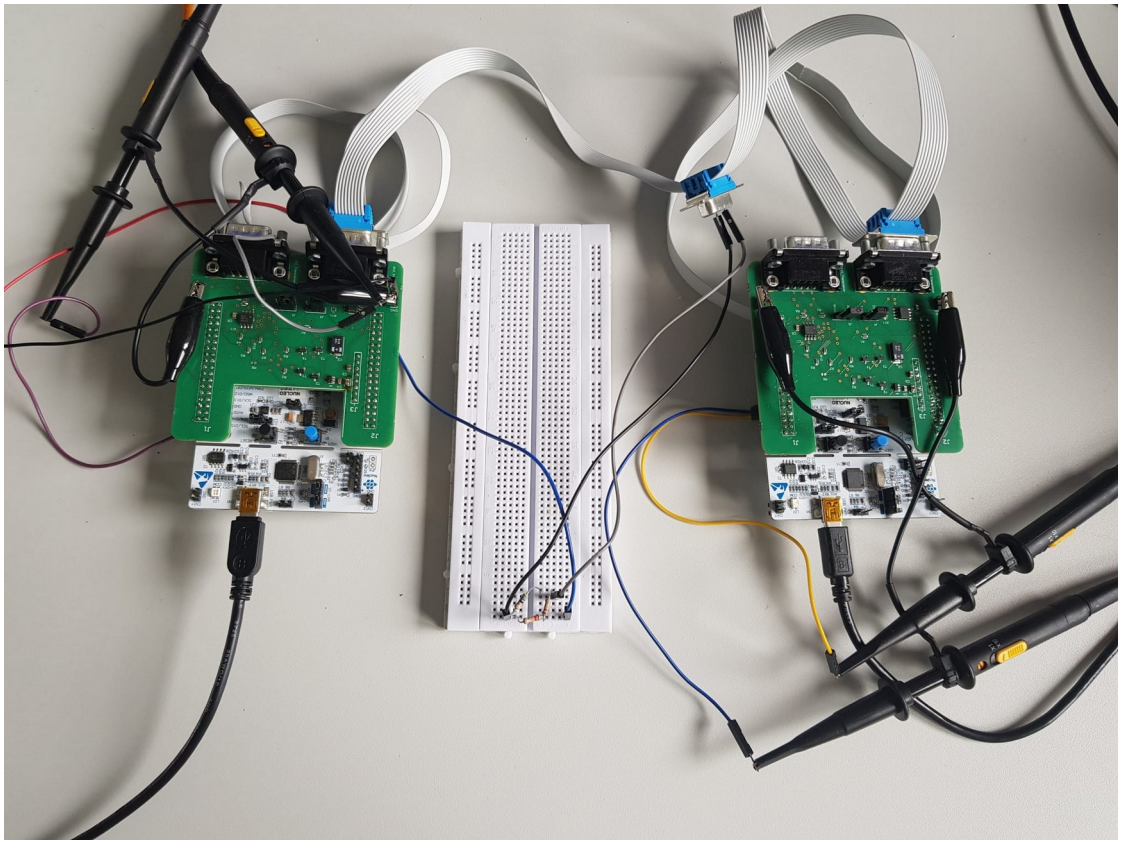


Abbildung 8.2: Physikalische Verbindung zwischen den beiden Mikrocontrollern

➢ **Programmierung**:

- Die Programmierung der Mikrocontroller erfolgt mithilfe der STM32CubeIDE und CubeMX. Die UART- und CAN-Schnittstellen werden gemäß den Versuchsanforderungen konfiguriert.
- Die Mithilfe von Qt Creator erstellte Benutzeroberfläche wird kompiliert und ausgeführt.

➢ **Durchführung des Versuchs**:

- Alle Komponenten, einschließlich der Mikrocontroller, PCs, Stromversorgung und des Oszilloskops, werden eingeschaltet.
- Die Kommunikation zwischen den PCs und den Mikrocontrollern wird mithilfe der implementierten Kommunikationsprotokolle (UART und CAN) gestartet.
- Die Signale auf dem Oszilloskop werden überwacht, um sicherzustellen, dass die Daten ordnungsgemäß übertragen werden.

- Die Benutzeroberfläche wird verwendet, um Nachrichten zwischen den PCs und den Mikrocontrollern auszutauschen und sicherzustellen, dass die Kommunikation erfolgreich verläuft.

➢ **Funktionsweise:**

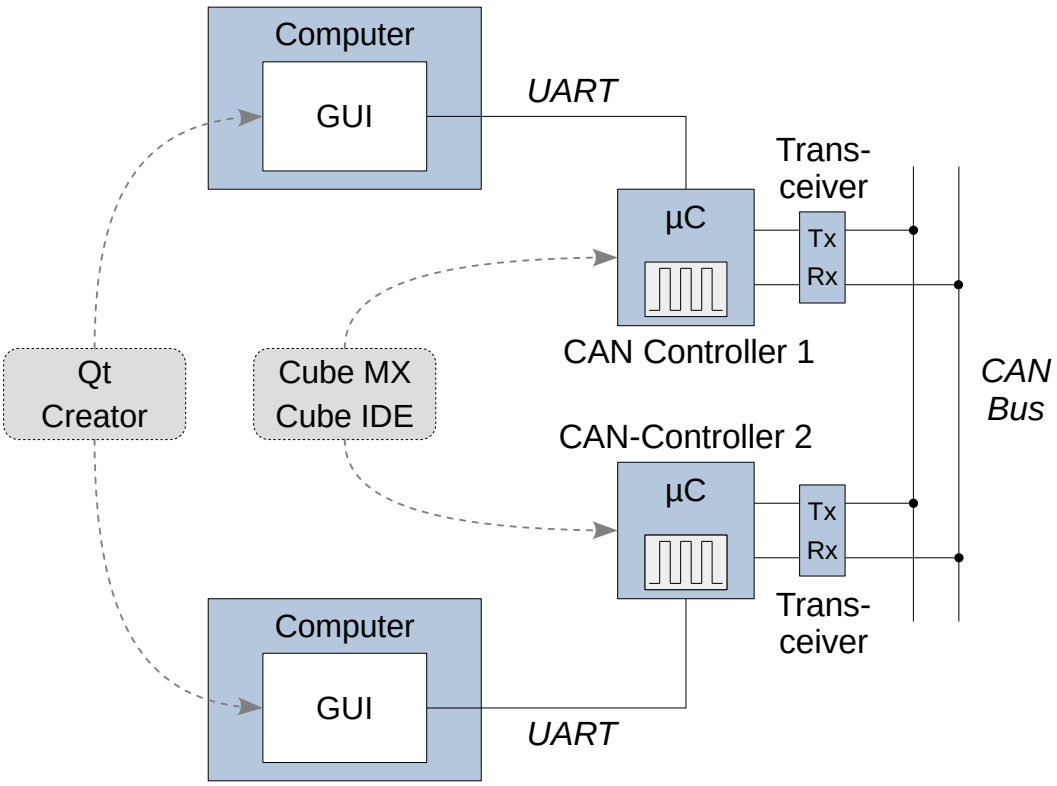


Abbildung 8.3: Schematische Darstellung des Aufbaus

1. Der erste PC sendet Daten über die USB/UART-Verbindung an den ersten Mikrocontroller. Die UART-Schnittstelle des Mikrocontrollers empfängt die Daten und speichert sie.
2. Der erste Mikrocontroller verarbeitet die empfangenen UART-Daten und konvertiert sie in CAN-Nachrichten. Diese Nachrichten werden dann über den CAN-Bus an den zweiten Mikrocontroller gesendet.
3. Der zweite Mikrocontroller empfängt die CAN-Nachrichten über den CAN-Bus und extrahiert die darin enthaltenen Daten.
4. Die extrahierten Daten werden vom zweiten Mikrocontroller über die USB/UART-Schnittstelle an den zweiten PC gesendet.
5. Der zweite PC empfängt die UART-Daten vom zweiten Mikrocontroller und kann sie auf seiner Benutzeroberfläche anzeigen oder weiterverarbeiten.

8.2 Nachrichtenüberwachung mit dem Oszilloskop

Die Darstellung der Nachrichten auf dem Oszilloskop ermöglicht eine visuelle Überprüfung der Signalqualität und der Datenübertragung zwischen den Systemen.

Die im Bereich 4 der Abbildung 7.1 eingegebenen (gesendeten)Nachrichten (ID: 0FF, DLC:3, D0: AB, D1: CD, D2: EF) wurden von Rechner 1 an Rechner 2 gesendet und können mit dem Oszilloskop ausgelesen werden.

Hier sind die Details dieser Nachrichten:

- **ID: 0FF** (Identifikationsnummer)
- **DLC: 3** (Data Length Code, Anzahl der Datenbytes)
- **D0: AB** (Datenbyte 0)
- **D1: CD** (Datenbyte 1)
- **D2: EF** (Datenbyte 2)

Abbildung 8.4: Auslesen der übertragenen Nachrichten mit dem Oszilloskop

Die Signale zeigen den klaren Empfang der Nachrichten durch Rechner 2, was die erfolgreiche Kommunikation über die UART-CAN-Kommunikationsbrücke bestätigt.

Abbildung 8.5: Auslesen der Empfangenen Nachrichten mit dem Oszilloskop

Auf dem Oszilloskop werden die empfangenen Nachrichten (Received Data) (ID: 0FF, DLC:2, D0: 0A, D1: 1B) erfolgreich übertragen werden, die im Bereich 4 der Abbildung 7.1 (von Rechner 2 an Rechner 1) angezeigt werden.

Die Oszilloskop-Darstellungen in den Abbildungen 8.4 und 8.5 bestätigen die effektive und zuverlässige Datenübertragung, was ein entscheidendes Kriterium für die erfolgreiche Implementierung der UART-zu-CAN-Kommunikationsbrücke darstellt.

Die Kommunikation über die UART-Schnittstelle wurde erfolgreich in beiden Richtungen verifiziert, wie auf dem Oszilloskop zu sehen ist. Dies bestätigt die einwandfreie Funktionalität der Kommunikation zwischen dem STM32L476RG Mikrocontroller und anderen angeschlossenen Geräten, wodurch Daten zuverlässig empfangen und gesendet werden können

Kapitel 9

Zusammenfassung und Ausblick

Das Ziel dieser Arbeit bestand darin, eine Verbindung zwischen zwei STM32L476RG-Boards und zwei PCs über einen CAN BUS herzustellen. Zunächst wurde die vollständige Toolchain für die STM32-Plattform installiert- Die Toolchain umfasst eine IDE mit integriertem Quelleneditor, einen Compiler für die ARM Cortex-M-Plattform, einen Debugger für die Firmware sowie die ST-Link-Schnittstelle für die Kommunikation mit dem Hardware-Debugger. Danach wurde der Code erstellt, mit dem eine serielle Kommunikation zwischen dem Mikrocontroller und dem PC über die UART-Schnittstelle ermöglicht wird. Hierfür wurde ein C-Code im STM32CubeMX generiert, welcher der Hardwarekonfiguration für STM32CubeIDE entspricht. Schließlich wurde mithilfe des Qt Creators eine grafische Benutzeroberfläche erstellt.

CAN-Daten können vom Benutzer am PC eingegeben werden und über die serielle Schnittstelle an den Mikrocontroller übertragen werden. Der Mikrocontroller versendet die Daten über den CAN Bus, welche durch den empfangenden Mikrocontroller verarbeitet und über die UART Schnittstelle an den PC übertragen werden, der mit dem empfangenden Mikrocontroller verbunden ist. Der PC stellt die empfangenen Daten in der GUI dar.

Die resultierende Anwendung ermöglicht eine effiziente Kommunikation zwischen dem Benutzer und dem Mikrocontroller und bietet eine reibungslose und intuitive Benutzer-erfahrung.

Die gesetzten Ziele des Projekts wurden erfolgreich erreicht. Die gesamte Toolchain wurde sorgfältig installiert und getestet.

Alles in allem ist dieses Projekt ein gutes Beispiel, um einen Einblick in die Verwendung von Mikrocontrollern und ihre Peripherie zu erhalten.

Abkürzungsverzeichnis

ARM	Advanced RISC Machines
CAN	Controller Area Network
CPU	Central Processing Unit
DLC	Data Length Code
DMA	Direct Memory Access
GPIO	General Purpose Input/Output
HAL	Hardware Abstraction Layer
IDE	Integrated Development Environment
ISO	International Standards Organisation
LED	Light Emitting Diode
MacOS	Macintosh Operating System
MCU	Mikrocontroller
PC	Personal Computer
STM	STMicroelectronics
UART	Universal Asynchronous Receiver Transmitter
USART	Universal Synchronous/Asynchronous Receiver Transmitter
USB	Universal Serial Bus
ACK	Acknowledgment
RTR	Remote Transmission Request
Tq	Time Quantum
SJW	ynchronization Jump Width
NRZ	Non-Return to Zero
IDE	Integrated Development Environment
HAL	Hardware Abstraction Layer
API	Application Programming Interface
MSP	Microcontroller Support Package

Literaturverzeichnis

[1] STMicroelectronics, „STM32 Microcontrollers (MCUs)“. Zugegriffen: 28. April 2023. [Online]. Verfügbar unter: https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html

[2] STMicroelectronics, „STM32 Nucleo Boards“. Zugegriffen: 28. April 2023. [Online]. Verfügbar unter: https://www.st.com/en/evaluation-tools/stm32-nucleo-boards.html

[3] STMicroelectronics, „STM32 Mainstream Microcontrollers (MCUs)“. Zugegriffen: 3. Mai 2023. [Online]. Verfügbar unter: https://www.st.com/en/microcontrollers-microprocessors/stm32-mainstream-mcus.html

[4] STMicroelectronics, „STM32 Ultra Low Power Microcontrollers (MCUs)“. Zugegriffen: 3. Mai 2023. [Online]. Verfügbar unter: https://www.st.com/en/microcontrollers-microprocessors/stm32-ultra-low-power-mcus.html

[5] STMicroelectronics, „STM32 High Performance Microcontrollers (MCUs)“. Zugegriffen: 3. Mai 2023. [Online]. Verfügbar unter: https://www.st.com/en/microcontrollers-microprocessors/stm32-high-performance-mcus.html

[6] STMicroelectronics, „NUCLEO-WBA52CG - STM32 Nucleo-64 development board with STM32WBA52CG MCU“. Zugegriffen: 3. Mai 2023. [Online]. Verfügbar unter: https://www.st.com/en/evaluation-tools/nucleo-wba52cg.html

[7] „NUCLEO-WB15CC - STM32 Nucleo-64 development board with STM32WB15CC, supports Arduino, ST Morpho connectivity“. Zugegriffen: 3. Mai 2023. [Online]. Verfügbar unter: https://www.st.com/en/evaluation-tools/nucleo-wb15cc.html

[8] STMicroelectronics, „NUCLEO-WB55RG - STM32 Nucleo-64 development board with STM32WB55RG MCU, supports Arduino, ST Zio and morpho connectivity“. Zugegriffen: 3. Mai 2023. [Online]. Verfügbar unter: https://www.st.com/en/evaluation-tools/nucleo-wb55rg.html

[9] STMicroelectronics, „NUCLEO-WL55JC - STM32 Nucleo-64 development board with STM32WL55JC MCU, SMPS, supports Arduino and morpho connectivity“. Zugegriffen: 3. Mai 2023. [Online]. Verfügbar unter: https://www.st.com/en/evaluation-tools/nucleo-wl55jc.html

[10] „CortexFamily“. Zugegriffen: 30. Juli 2023. [Online]. Verfügbar unter: http://www.emcu.it/CortexFamily/CortexFamily.html

[11] STMicroelectronics, „UM1724 User manual“. August 2020. Zugegriffen: 28. April 2023. [Online]. Verfügbar unter: https://www.st.com/content/ccc/resource/technical/do-

cument/user_manual/98/2e/fa/4b/e0/82/43/b7/DM00105823.pdf/files/DM00105823.pdf/jcr:content/translations/en.DM00105823.pdf

[12] „STM32L476RG - Ultra-low-power with FPU Arm Cortex-M4 MCU 80 MHz with 1 Mbyte of Flash memory, LCD, USB OTG, DFSDM - STMicroelectronics“. Zugegriffen: 19. August 2023. [Online]. Verfügbar unter: https://www.st.com/en/microcontrollers-microprocessors/stm32l476rg.html

[13] „CAN-Bus“. Zugegriffen: 19. Juli 2023. [Online]. Verfügbar unter: https://www.sprut.de/electronic/interfaces/can/can.htm

[14] Ihssen Boukhriss, „Entwurf eines Aufsteckmoduls für ein STM32 Nucleo Mikrocontroller Board mit einem 3,3V und 1,2V CAN-Transceiver“. Juli 2023.

[15] J. Weissgärber, „Die UART Schnittstelle“. Zugegriffen: 28. April 2023. [Online]. Verfügbar unter: http://www.mathe-mit-methode.com/schlaufuchs_web/elektrotechnik/mikrocontroller_lernmaterial/microcontroller_allgemein/mikrocontroller_ext_hardware/mikrocontroller_uart.html

[16] „UART-Frame“. Zugegriffen: 28. April 2023. [Online]. Verfügbar unter: https://iwo-fr.org/de/uart-kommunikationsprotokoll-wie-funktioniert-es/

[17] „STM32CubeMX - STM32Cube initialization code generator - STMicroelectronics“. Zugegriffen: 17. August 2023. [Online]. Verfügbar unter: https://www.st.com/en/development-tools/stm32cubemx.html

[18] „Figure 1. The CAN bus interface.“, ResearchGate. Zugegriffen: 20. Juli 2023. [Online]. Verfügbar unter: https://www.researchgate.net/figure/The-CAN-bus-interface_fig1_357556599

[19] „High-speed CAN and low-speed, fault-tolerant CAN - Company News“. Zugegriffen: 16. August 2023. [Online]. Verfügbar unter: http://www1.gcanbox.com/fsd/gsxw/high-speed-CAN,low-speed-CAN.html

[20] „CAN Bus Protocol Tutorial | Kvaser“. Zugegriffen: 17. August 2023. [Online]. Verfügbar unter: https://www.kvaser.com/can-protocol-tutorial/

[21] „CAN Bus Grundlagen - ME-Systeme“. Zugegriffen: 17. August 2023. [Online]. Verfügbar unter: https://www.me-systeme.de/de/technik-zuerst/elektronik/can-bus-grundlagen

[22] „2017_Vernetzung_mit_CAN_FD.pdf“. Zugegriffen: 20. Juli 2023. [Online]. Verfügbar unter: https://www.goepel.com/fileadmin/fachartikel/ats/de/2017_Vernetzung_mit_CAN_FD.pdf

[23] „STM32F439xx HAL User Manual: stm32f4xx_hal_can.h Source File“. Zugegriffen: 23. Juli 2023. [Online]. Verfügbar unter: http://www.disca.upv.es/aperles/arm_cortex_m3/llibre/st/STM32F439xx_User_Manual/stm32f4xx__hal__can_8h_source.html

[24] „STM32F439xx HAL User Manual: CAN Error Code“. Zugegriffen: 21. August 2023. [Online]. Verfügbar unter: http://www.disca.upv.es/aperles/arm_cortex_m3/llibre/st/STM32F439xx_User_Manual/group__can__error__code.html

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Quellcodes

Eidesstattliche Versicherung

„Hiermit versichere ich an Eidesstatt, dass ich die vorliegende Arbeit selbständig und ohne die Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht.“

_____ _____

Ort, Datum Fouzya Samdouni