Fachbereich: Informatik und Elektrotechnik

Embedded Systems / Internet of Things AG

Projekt: Plattform Autonomes Fahrzeug

Leiter der Veranstaltung: Prof. Dr. Ralf Patz
Prof. Dr.-Ing. Wolfram Acker

Projektteilnehmer:

Bosse Hauschildt	(935105)
Jakob Mohr	(935189)
Eric Ozdoba	(935138)
Tobias Paulikat	(935204)

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Zielsetzung

Das Projekt „Autonomes Fahrzeug“ beschäftigt sich mit der Realisierung eines steuerbaren Fahrzeuges im Format eines Modellautos. Der Schwerpunkt liegt auf der Kommunikation von vier verbauten Schrittmotoren über ein CAN-Bus System mittels Mikrocontrollern. Für die Bedienoberfläche soll ein Raspberry Pi mit grafischer Benutzeroberfläche und Touch-Eingabe verwendet werden. Ziel ist es, präzise Eingabeaufforderung umzusetzen. Erhöhte Geschwindigkeiten und unkontrolliertes Verhalten sind nicht gewünscht. Des Weiteren ist die Entwicklung eigener Platinen geplant. Das Fahrzeug soll modular für Erweiterungen ausgelegt werden. In Zukunft könnten Kameras und Sensoren implementiert werden. Die Herausforderung besteht zunächst darin, die Kommunikation aller vorhandenen Schnittstellen zu ermöglichen.

2. Aufbau des Modellautos

Der Aufbau des Modellautos ist einfach und in erster Linie stabil gestaltet. Die Hauptkomponente bildet eine einfache PVC-Platte mit Aluminiumprofilen zur Verstärkung. Die Schrittmotoren sind mit Lochbändern unter der Platte aufgehängt. Die Räder besitzen einen Durchmesser von 17cm und sind mit sondergefertigten Felgenmitnehmern befestigt. Der große Durchmesser dient zur nötigen Bodenfreiheit. Die Größe der Plattform wurde so gewählt, dass die einzelnen Platinen der Schrittmotoren Platz finden. Der Raspberry Pi inkl. Touch-Display und die Hauptplatine sollen über Abstandshalter zentral befestigt werden. Des Weiteren kann ein Akku zur Spannungsversorgung verbaut werden.

3. Schrittmotoren

Die zweipoligen Schrittmotoren verfügen über ein Haltemoment von 1,68 Nm mit einer Nennspannung von 2,8 V und einem Nennstrom von 1,68 A. Für die präzise und kraftvolle Anwendung sind die Schrittmotoren mit einem 5:1 Getriebe übersetzt. Die Ansteuerung der Motoren erfolgt über die H-Brücke L298N und dem IC L297. Sie besitzen jeweils zwei Spulen, die über die H-Brücke gepolt werden können. Durch gezielte Umpolung der H-Brücke, werden sie in Voll- oder Halbschritten in Bewegung gesetzt und besitzen eine 4-Pin Buchse als Anschluss. Alle Schrittmotoren bekommen ihre Befehle über den CAN-Bus.

4. Mikrocontroller

Der Mikrocontroller soll sowohl als Kommunikationsschnittstelle als auch zur Steuerung der Motoren genutzt werden. Verwendet wurde der dsPIC33EP512GP502, da dieser den von MPLab zur Verfügung gestellten Code-Configurator (MCC) unterstützt und die Pinbelegung nur geringe Abweichungen zu dem dsPIC33FJ128GP802 aufweist.

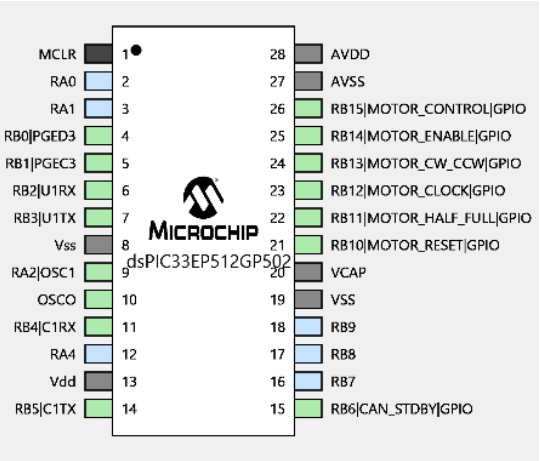

Abbildung 1 Pinout dsPIC33EP512GP502

Mit Hilfe des Code-Configurator ist es möglich einzelne Module des Mikrocontrollers übersichtlich in einer grafischen Oberfläche zu konfigurieren. Dieser generiert eine fertige Dateihierarchie, welche vorgefertigte Funktionen beinhaltet. Diese Funktionen können dann vom Nutzer aufgerufen werden, um bestimmte Aktionen der Module auszuführen.
Genutzt wurde der Konfigurator für die Module CAN-Bus inklusive DMA, UART, GPIO's sowie des Oszillators.

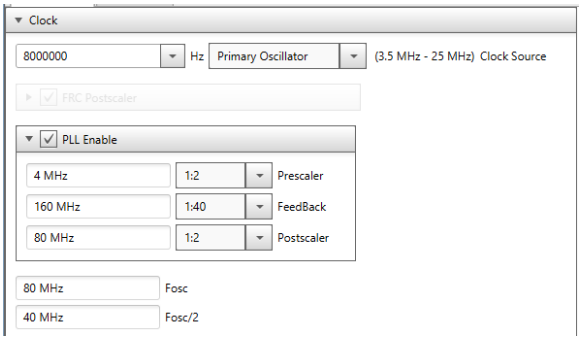

Abbildung 2 MCC Clock-Einstellungen

Da für den CAN-Bus eine hohe Taktfrequenz benötigt wird, wurde der Oszillator auf 80 MHz per PLL getaktet.

Der entworfene Programmcode ist so konzipiert, dass nur ein Projekt existiert, in dem über Define-Befehle zwischen den spezifischen Aufgaben des Controllers unterschieden werden kann.

Es existieren folgende Funktionsmöglichkeiten:

- Motor linke Seite

- Motor rechte Seite

- Raspberry Schnittstelle

4.1 Motorsteuerung

Zur Ansteuerung eines Motors wird ein Rechtecksignal benötigt, welches über einen Timer des Mikrocontrollers erzeugt wird. Die Drehzahl des Motors wird dabei über die Periode des Timers geregelt. Um ein Anfahrverhalten zu erzeugen, wird bis zu einer gewissen Drehzahl des Motors, die Periode des Timers verändert. Gleiches gilt beim Anhalten.
Um ein präzises Navigieren des Fahrzeuges zu ermöglichen, wird jedem Motor eine Restschrittzahl übermittelt, die dieser bis zu seinem Stillstand noch drehen muss.

Zusätzlich zu den Restschritten erhalten die Motoren bestimmte Befehle:

- F: Forward (Vorwärts)
- B: Backward (Rückwärts)
- S: Stop (Anhalten)

```
// Auswertung der Restschritte und Steuerung des Periodenregisters -> Geschwindigkeit
void schrittAuswertung() {
    // Bei Richtungswechsel abbremsen und wieder anfahren
    if(changeDirection != 0) {
        if(PR1 < MOTOR_MIN_SPEED)
            PR1 += MOTOR_ACCELERATION;
        else {
            PR1 = MOTOR_MIN_SPEED;
            changeDirection = 0;
        }
    }
    // Zum anhalten abbremsen dann stoppen
    else if(anhaltenFlag != 0) {
        if(PR1 < MOTOR_MIN_SPEED)
            PR1 += MOTOR_ACCELERATION;
        else {
            stopMotor();
            PR1 = MOTOR_MIN_SPEED;
        }
    }
    // wenn Schritte aufgebraucht stoppen
    else if(restSchritte <= 0) {
        stopMotor();
        PR1 = MOTOR_MIN_SPEED;
        anhaltenFlag = 0;
    }
    // abbremsen wenn nur noch genug Restschritte zum gerade eben anhalten
    else if(restSchritte <= ((MOTOR_MIN_SPEED - PR1) / MOTOR_ACCELERATION) && PR1 < MOTOR_MIN_SPEED) {
        PR1 += MOTOR_ACCELERATION;
    }
    // beschleunigen solange noch nicht full speed
    else if(PR1 > MOTOR_MAX_SPEED) {
        PR1 -= MOTOR_ACCELERATION;
    }
    // Restschritte reduzieren
    restSchritte--;
```

Abbildung 3 Schrittauswertung

Da der Motortreiber ebenfalls einige Einstellungen benötigt, um den Motor richtig zu betreiben, werden diese über verschiedene GPIO‘s am Mikrocontroller konfiguriert:

- Motor enable
- Motor reset
- Halb oder Vollschritt
- Rotationsrichtung (Clockwise/Counterclockwise)
- Chopping (Auswirkung der Stromregelung)

```
    // Initialisiere Steuerpins (GPIOs)
    MOTOR_ENABLE_SetHigh();     // Enable L297, Pull Low to disable
    MOTOR_RESET_SetHigh();      // Pull Low to Reset L297 to HOME Position
    MOTOR_HALF_FULL_SetLow();   // Pull Low for FULL-STEP, High for HALF-STEP
    // Vorwärtsdefinition
#ifdef MOTOR_LINKS
    MOTOR_CW_CCW_SetHigh();     // Pull Low for CounterClockWise, High for ClockWise Rotation
#endif
#ifdef MOTOR_RECHTS
    MOTOR_CW_CCW_SetLow();      // Pull Low for CounterClockWise, High for ClockWise Rotation
#endif
    MOTOR_CONTROL_SetHigh();    // Pull High for Chopping on ABCD, Low for INH1/2
}
```

Abbildung 4 GPIO-Einstellungen

4.2 CAN-Bus System

Zur Steuerung der einzelnen Motoren wird ein CAN-Bus verwendet. Über diesen werden Nachrichten mit Steuerungsbefehlen von einem, zentralen an einem Raspberry angeschlossenen, Mikrocontroller gesendet. Die gesendeten Nachrichten werden von weiteren an dem CAN-Bus teilnehmende Mikrocontroller empfangen und verarbeitet.

Der im Mikrocontroller verwendete CAN-Bus (Control-Area-Network) wurde für eine schnelle Datenübertragung auf eine Geschwindigkeit von 1Mbps konfiguriert. Zur Einteilung der verschickten Nachrichten filtern die Mikrocontroller die empfangenen Nachrichten nach sogenannten Messageid’s.

```
    /* Configure the filters */
    C1RXF0SIDbits.SID = 0x1;
#ifdef MOTOR_LINKS
    C1RXF1SIDbits.SID = 0x2;
#endif
#ifdef MOTOR_RECHTS
    C1RXF2SIDbits.SID = 0x3;
#endif
```

Abbildung 5 CAN-Filter-Einstellungen

Zur Sicherheit des Fahrzeugs wurde eine allgemeine Nachrichtenid (0x1) verwendet um einen einheitlichen Notstopp unabhängig von der behinhalteten Nachricht auszulösen.

Das CAN-Module benötigt einen konfigurierten Direct Memory Access (DMA). Dieses Modul ermöglicht direkten Zugriff auf den Speicher ohne den Prozessor zu

unterbrechen.
Im Hauptspeicher des Mikrocontrollers, ist ein Buffer konfiguriert, der bis zu 31 Nachrichten nach dem First in First out -Prinzip beinhalten kann.

Durch den MCC werden zwei Funktionen bereit gestellt um den CAN-Bus Receiver beziehungsweise den Transmitter zu aktivieren und zu deaktivieren. Um Nachrichten senden und empfangen zu können, wird eine bestimmte Abfolge von Daten benötigt. Der MCC liefert eine Struktur aus der diese Abfolge in Form eines Messageobjekts erzeugt werden kann.

```
typedef union {
    uint8_t msgfields;
    struct{
        uint8_t idType:1;       // 1 bit (Standard Frame or Extended Frame)
        uint8_t frameType:1;    // 1 bit (Data Frame or RTR Frame)
        uint8_t dlc:4;          // 4 bit (No of data bytes a message frame contains)
        uint8_t reserved:2;     // 2 bit (Reserved bit)
    };
} CAN_MSG_FIELD;

typedef struct
{
    uint32_t msgId;          // 29 bit (SID: 11bit, EID:18bit)
    CAN_MSG_FIELD field;     // CAN TX/RX Message Object Control
    uint8_t *data;           // Pointer to message data
} CAN_MSG_OBJ;
```

Abbildung 6 Struktur CAN-Message-Objekt

Um die gewünschte Funktion zu erreichen werden lediglich die Messageid und die Daten der Nachrichten aus dem Messageobjekt benötigt.

Des Weiteren wurden für die übersendeten Nachrichten Standardwerte für den Id-Typ, Frame-Typ, sowie der Datenlänge eingesetzt. Da die Gesamtanzahl der CAN Nachrichten im aktuellen Zustand sehr klein ist, wurde die Datenlänge auf 8 Byte festgelegt.

Der Ablauf zum Empfangen von Nachrichten gliedert sich wie folgt:

- Interrupt wenn Nachrichten im Buffer vorhanden
- Überprüfung von der Anzahl restlicher Nachrichten im Buffer
- Erzeugung des Messageobjektes
- Einlesen der Nachricht in das Messageobjekt aus dem Buffer
- Auswerten der empfangenen Nachricht

```
void CanRxBufferInterruptHandler() {
    receivedMessageFlag = 1;
}

void CanMessageAuswertung() {
    // Überprüfe ob wirklich Nachrichten im Buffer liegen
    if(CAN1_ReceivedMessageCountGet() == 0) {
        receivedMessageFlag = 0;
        return;
    }

    // Speicherplatz für CAN_MSG_OBJ Daten
    message.data = test;

    // Hole Datensatz aus Buffer
    if(CAN1_Receive(&message) == true) {
```

Abbildung 7 CAN-Message-Empfang

sowie zum Senden von Nachrichten:

- Erzeugung des Messageobjektes
- Zuweisen der Daten und Einstellungen zum Messageobjekt
- Übergeben des Messageobjektes an das CAN-Modul

```
message.msgId = 0x1;
message.field.idType = CAN_FRAME_STD;
message.field.frameType = CAN_FRAME_DATA;
message.field.dlc = CAN_DLC_8;
message.data = data;
// starte CAN Transmission
CAN1_Transmit(CAN_PRIORITY_NONE,&message);
while(!C1INTFbits.TBIF);
C1INTFbits.TBIF = 0;
```

Abbildung 8 CAN-Message-Senden

4.3 Universal Asynchronous Receiver/Transmitter

Um die Daten durch eine Benutzereingabe über einen Raspberry Pi an die Mikrocontroller und somit an die Motoren zu schicken, wird eine UART Verbindung benötigt.

Dieses Modul wurde ebenfalls per MCC Konfiguriert und wird nur auf dem Mikrocontroller aufgesetzt, welcher eine direkte Verbindung zum Raspberry Pi besitzt. Die UART Konfiguration baut sich wie folgt auf:

- Baud Rate 115200 bit/s
- Kein Parity-Bit
- 8 Daten Bits
- 1 Stop Bit

Um empfangene Daten auszuwerten, erwartet der Mikrocontroller einen bestimmten Protokollaufbau, angelehnt an das NMEA-Protokoll:

“ ${Steuerbefehl}{Schrittzahl}* “, Beispiel: “ $F5000* “

Steuerbefehle stellen dabei die Bewegungsrichtung dar. Diese unterteilen sich in:

- F: Forward (Vorwärts)
- B: Backward (Rückwärts)
- S: Stop (Anhalten)
- L: Left (Linksdrehung)
- R: Right (Rechtsdrehung)
- E: Emergency-Brake (Notstopp)

Die übertragene Schrittzahl gibt an, wie viele Schritt das Fahrzeug insgesamt zurücklegen soll.

Bevor die Nachrichten über den CAN-Bus an die weiteren Mikrocontroller versendet werden, wird der Steuerbefehl ausgewertet. So müssen sich beispielsweise die Räder

auf der gegenüberliegenden Seite jeweils entgegengesetzt drehen, wenn eine Links- oder Rechtsdrehung erfolgen soll.

```c
// setze UART String zusammen und werte diesen aus
void UartAuswertung() {
    static char teststring[20];
    static int i = 0;
    static int keep = 0;

    // einlesen des nächsten Characters
    teststring[i] = UART1_Read();
    // Startzeichen eines Befehlsstrings
    if(teststring[i] == '$')
        keep = 1;
    // Endzeichen eines Befehlsstrings
    else if(teststring[i] == '*') {
        // Zerlege String in Befehlszeichen und Schrittanzahl
        char befehl = teststring[1];
        int schritte = atoi(teststring + 2);
        // starte CAN Transmission
        sendCAN(befehl, schritte);
        // leere String für den nöchsten Befehlsstring
        keep = 0;
        i = 0;
    }
    // String zu lang evtl Zeichenfehler
    if(i >= 19) {
        keep = 0;
        i = 0;
    }
    // wenn Befehlsstring erkannt zähle index hoch
    if(keep)
        i++;
}
```

```c
// sende per UART empfangene Daten auf den CAN Bus
void sendCAN(char befehl, int schritte) {
    CAN_MSG_OBJ message;
    uint8_t data[8] = {0, 0, 0, 0, 0, 0, 0, 0};
    char befehlLinks;
    char befehlRechts;

    // erstelle spezifische Befehle für die Motorseiten
    if(befehl == 'L') {
        befehlLinks = 'B';
        befehlRechts = 'F';
    }
    else if(befehl == 'R') {
        befehlLinks = 'F';
        befehlRechts = 'B';
    }
    else if(befehl == 'E') {
        // Sende eine leere Nachricht mit der ID 0x01 -> Notstop!
        message.msgId = 0x1;
        message.field.idType = CAN_FRAME_STD;
        message.field.frameType = CAN_FRAME_DATA;
        message.field.dlc = CAN_DLC_8;
        message.data = data;
        // starte CAN Transmission
        CAN1_Transmit(CAN_PRIORITY_NONE,&message);
        while(!C1INTFbits.TBIF);
        C1INTFbits.TBIF = 0;
        return;
    }
    else {
        befehlLinks = befehl;
        befehlRechts = befehl;
    }

    // Teile Schritte auf 3 mal 8 Byte auf
    data[1] = schritte >> 8;
    schritte &= 0x0000FF;
    data[2] = schritte;
```

Abbildung 9 UART zu CAN Übersetzung

5. Raspberry Pi

5.1 Funktion und Allgemeines

Als zentraler Steuerungscomputer des Modellautos wird ein Raspberry Pi verwendet. Im aktuellen Zustand des Projektes verfügt dieser über folgende Funktionen:

- Bereitstellen einer grafischen Oberfläche zur Bedienung
- Entwicklungsoberfläche

Die grafische Benutzeroberfläche basiert auf dem Anwendungsframework „Qt“ und wird auf einem FHD AMOLED Display der Marke „Waveshare“ dargestellt.

5.2 Hardware

Computer:	Raspberry Pi 4
Display:	WAVESHARE, 5.5inch HDMI AMOLED, touch

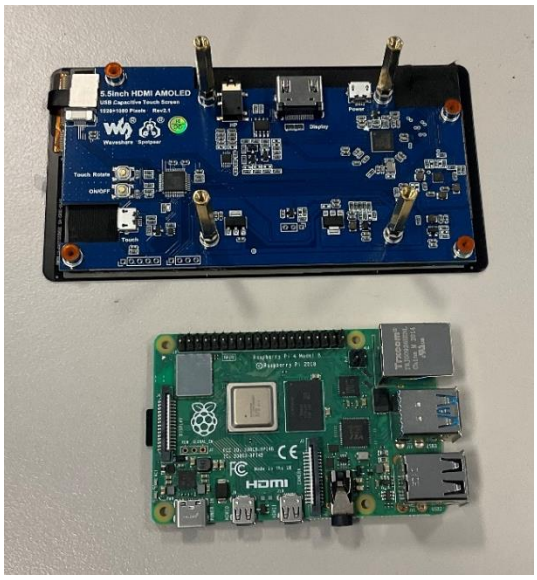

Abbildung 10 Touchdisplay (hinten) und Raspberry Pi 4

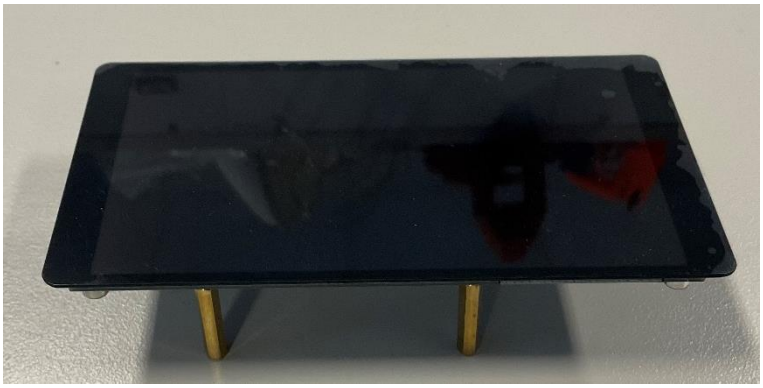

Abbildung 11 Touchdisplay (vorne)

5.3 Software

Der Raspberry Pi bietet eine freie Arbeitsoberfläche durch das installierte „Raspberry Pi OS“ und kann von jeder, aktuell bearbeitenden Projektgruppe frei verändert werden. Zurzeit besteht eine kleine, grundsätzliche Ordnerstruktur. Unter dem Pfad */home/pi* befinden sich, neben den vorinstallierten Ordnern, eigens angelegte Ordner:

- */app* Ordner für Applikationen
- */app_backup* Ordner zur Sicherung des letzten Standes
- */dev* Ordner für Test-Applikationen und Test-Code

Unter dem Pfad */home/pi/app/Display* ist die aktuell einzige Applikation zu finden. Es handelt sich dabei um die grafische Benutzeroberfläche zur Steuerung des Modellautos. Diese lässt sich durch die Befehlskette im Befehlsfenster ausführen:

```
pi@raspberrypi:~/app_Backup/Displayohneslider $ cd /home/pi/app/Display
pi@raspberrypi:~/app/Display $ ./Display
```

Abbildung 12 GUI Startbefehl

Es erscheint die folgende Benutzeroberfläche:

Abbildung 13 GUI

1. Mit der Schaltfläche „START“ lässt sich eine einstellbare Motoransteuerung abspielen.
2. Über den „Slider“ unter „START“ lassen sich die Anzahl der Schritte (die Dauer) der eingestellten Bewegung einstellen.
3. Das „Drop-Down-Menu“ lässt entscheiden, ob es sich um eine Drehung nach links/rechts oder eine Bewegung vorwärts/rückwärts handeln soll (siehe Abbildung unten).
4. Der Schalter „DEV“ lässt vom Vollbildmodus in den Fenstermodus und zurück wechseln.
5. Mit „QUIT“ wird das Programm geschlossen.

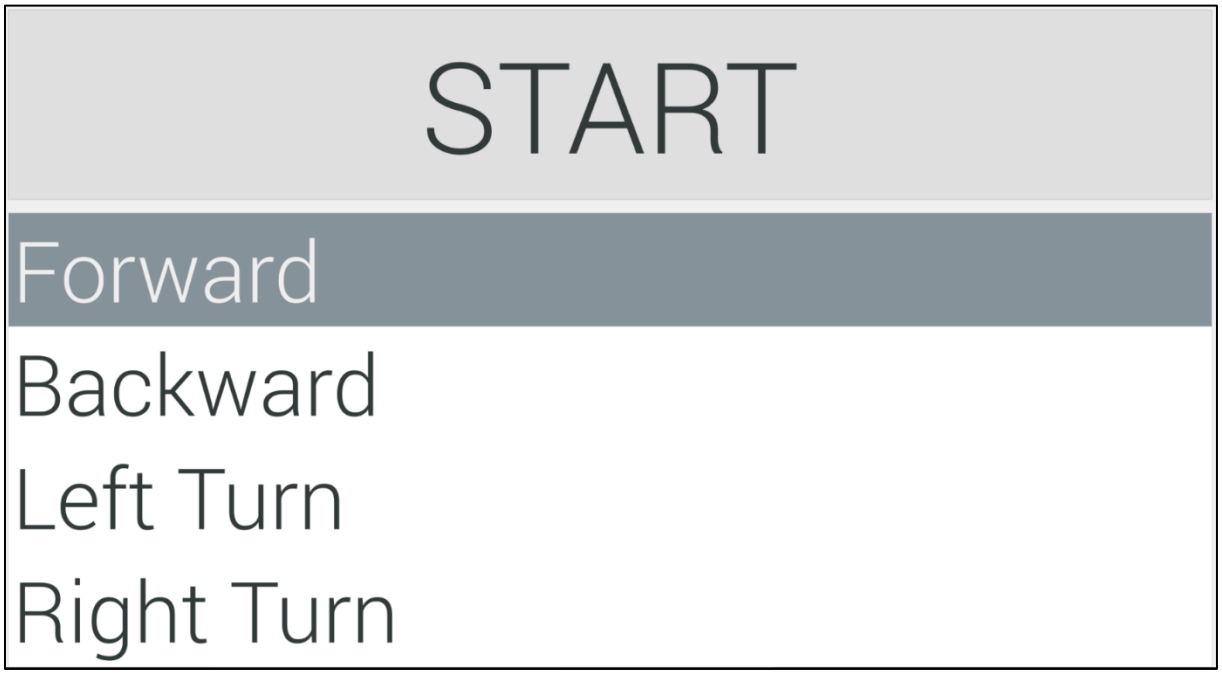

Abbildung 14 GUI Dropdown-Menu

Im aktuellen Zustand ist das Menü funktionsfähig, jedoch sind sowohl die Auswahl der Bewegungsart im Dropdown-Menü als auch die einstellbare Schrittanzahl mit dem Slider ohne Funktion im Hintergrund.

Der Code besteht aus einer Hauptdatei (*main.cpp*), zwei Dateien bezüglich des Fensters (*Window.cpp* und *Window.h*) und zwei Dateien, um ein Sender-Objekt zu definieren, mit dem die UART-Schnittstelle freigeschaltet und darüber gesendet wird (*Sender.cpp* und *Sender.h*). Im nachfolgenden sind die Dateien mit ihren Kurzbeschreibungen aus dem Programmcode und einigen Ergänzungen erläutert.

Main.cpp

Beschreibung im Code:

„In der main Funktion wird eine qt-übliche Initialiserung der Applikation vorgenommen, ein Objekt vom Typ Window erstellt und dieses aufgerufen.“

```
int main(int argc, char *argv[])
{
   QApplication a(argc, argv);
   Window *wnd=new Window();
   wnd->show();
   QSize mysize = wnd->frameSize();
   return a.exec();
}
```

Abbildung 15 qt main Aufruf

In dem Hauptprogramm werden nur die notwendigsten Aufrufe getätigt. Der Rest der Benutzeroberfläche wurde ausgelagert.

Window.h

Beschreibung im Code:

„Dieser Header-File dient dazu, die Klasse ‚Window‘ zu definieren. Objekte der Klasse ‚Window‘ besitzen diverse Bedienelemente in Form von qt-Widgets und Methoden, die bei Bedienung der einzelnen Widgets ausgeloest werden.“

```
/* Class */
class Window : public QMainWindow
{
    Q_OBJECT
    public:
        QPushButton *startBtn;
        QPushButton *quitBtn;
        QPushButton *devBtn;
        QSlider *sld;
        QWidget *wdg;
        QComboBox *modeDropdown;
        QVBoxLayout *mainLayout;
        Sender *mainSender;
        Window();
        void send();
        std::string order;

    public slots:
        void OnSliderMoved(int i);
        void StartRun();
        void Devmode();
        void Usermode();
};
```

Abbildung 16 qt QMainWindow Klasse

Window.cpp

Beschreibung im Code:

„Hier wird der Standardkonstruktor fuer ein Objekt vom Typ ‚Window‘ definiert und dessen Methoden ausformuliert.“

Der Code besteht hauptsächlich aus Einstellungen der einzelnen qt-Widgets. Die Funktion der Methode zum starten der Applikation wird in der Sender.cpp Datei weiter beschrieben, da dort nur die Methode *send()* des „Senders“ aufgerufen wird.

```
//Send String via UART
void Window::send()
{
    mainSender->send("$F5000*");
}
```

Abbildung 17 qt Methode send()

Sender.h

Beschreibung im Code:

„Dieser Header-File dient dazu die Klasse ‚Sender‘ zu definieren. Objekte der Klasse ‚Sender‘ koennen den UART Kanaldes Raspberry Pi’s oeffnen und ueber diesen einen ‚string‘ senden.“

```
/* Class */
class Sender
{
  public:
    int uart0_filestream = -1;
    Sender();
    void send(const char daten[20]);
};
```

Abbildung 18 qt Sender Klasse

Sender.cpp

Beschreibung im Code:

„Hier wird der Standardkonstruktor fuer ein Objekt vom Typ ‚Sender' definiert und die Methode ‚send' ausformuliert. Dieser Code basiert auf Beispielen aus Internetrecherchen."

```
/* Constructors */
Sender::Sender()
{
    uart0_filestream = open("/dev/ttyS0", O_RDWR | O_NOCTTY | O_NDELAY);
    if (uart0_filestream == -1) {
        printf("[ERROR] UART open()\n");
    }

    /* UART Options */
    struct termios options;

    tcgetattr(uart0_filestream, &options);

    options.c_cflag = B115200 | CS8 | CLOCAL | CREAD;
    options.c_iflag = IGNPAR;
    options.c_oflag = 0;
    options.c_lflag = 0;

    tcflush(uart0_filestream, TCIFLUSH);

    tcsetattr(uart0_filestream, TCSANOW, &options);
}

/* Methods */
void Sender::send(const char daten[20])
{
    uart0_filestream = open("/dev/ttyS0", O_RDWR | O_NOCTTY | O_NDELAY);

    if (uart0_filestream != -1) {
        int out = write(uart0_filestream, daten,strlen(daten)+1);
            if (out < 0) {
                printf("[ERROR] UART TX\n");
            } else {
                //printf("[STATUS: TX %i Bytes] %s\n", out, BUF_TX);
            }
    }
    close(uart0_filestream);
}
```

Abbildung 19 Sender Konstruktor und Methode send()

Die Einstellungen und Funktionen basieren auf folgender Quelle und sind leicht abgeändert bzw. vereinfacht:

https://www.einplatinencomputer.com/raspberry-pi-uart-senden-und-empfangen-in-c/

6. Schaltplan und Platine

Zur Implementierung der in Kapitel 1 genannten Signalkette wurde eine Platine entwickelt, die den Mikrocontroller inklusive CAN-Transmitter und Motorsteuerungs-ICs beheimatet. Dazu wurde das Programm KiCAD genutzt. Aus den Datenblättern zu den einzelnen Bauteilen wurden die notwendigen konzentrierten Bauelemente zur Beschaltung gesucht und ebenfalls auf dieser Platine primär in SMD Bauform aufgebracht.

Es ergibt sich folgendes Layout:

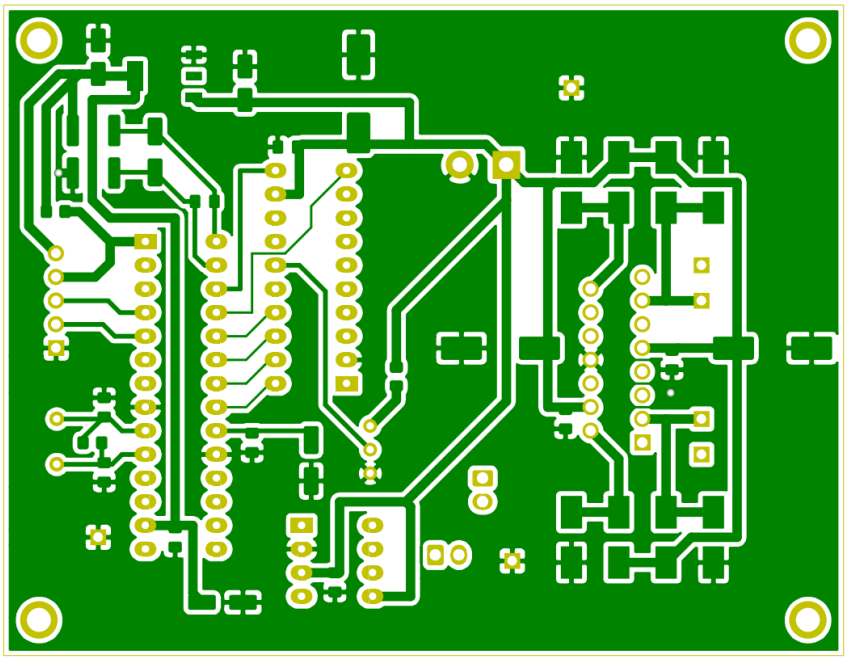

Abbildung 20 Platinen Layout Rückseite

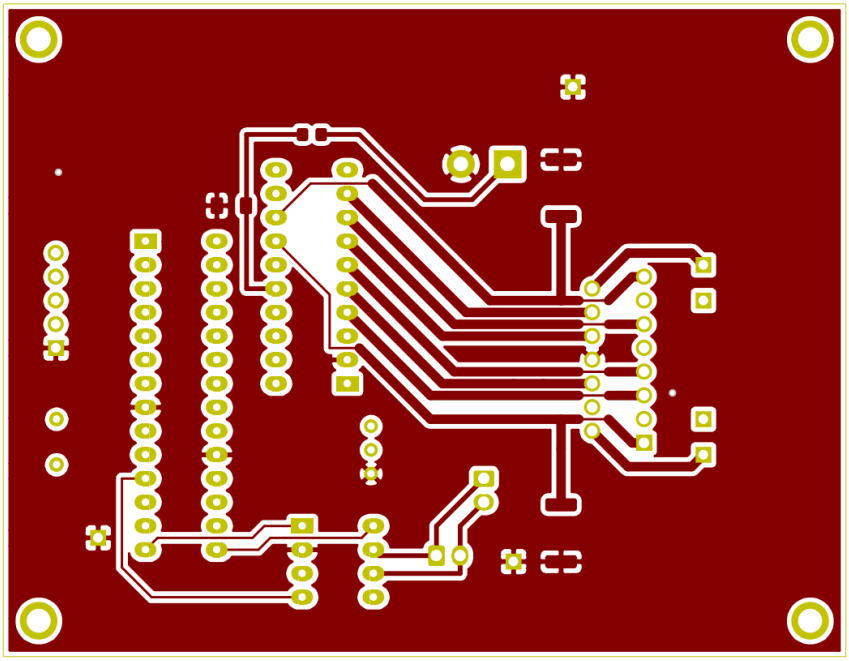

Abbildung 21 Platinen Layout Vorderseite

Dieses Layout wurde dann auf einem FR4 Substrat realisiert:

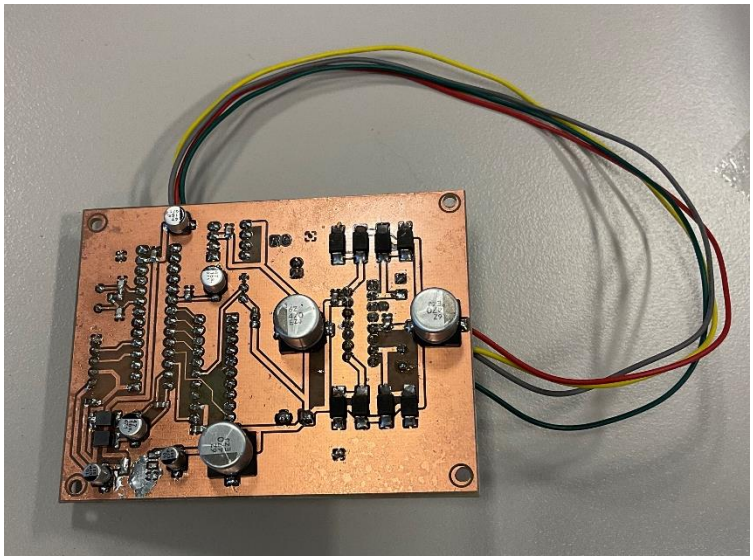

Abbildung 22 Platine Rückseite

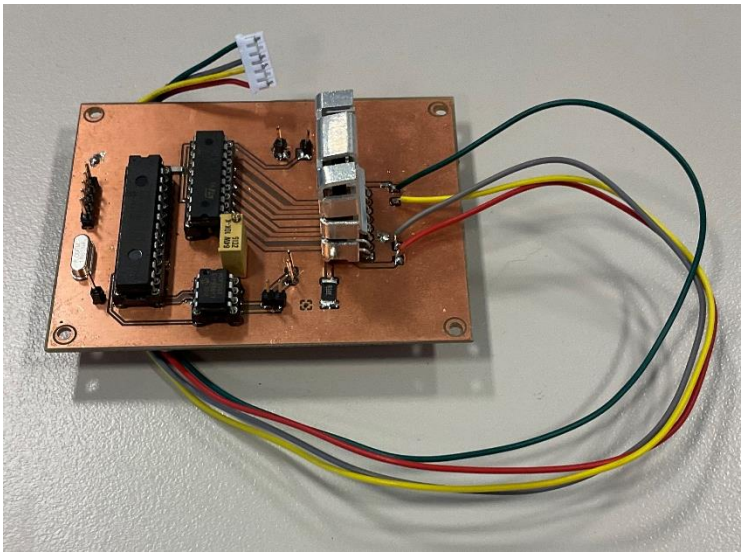

Abbildung 23 Platine Vorderseite

7. Ausblick

Im aktuellen Entwicklungsstand ist das Fahrzeug nicht fahrtüchtig. Die Signalkette von der grafischen Benutzeroberfläche bis zur Ansteuerung eines Motors über einen Mikrocontroller ist funktionsfähig und bietet eine Grundlage für den weiteren Ausbau zum fahrtüchtigen Fahrzeug.

7.1 Bestehende Fehler

Aktuell besteht ein Problem der Spannungsversorgung auf der entworfenen Platine, wodurch der Mikrocontroller nicht ordnungsgemäß startet. Die bisherige Fehlersuche ergab, dass bei Verwendung des Spannungsreglers ein Ripple auf der 3,3V Versorgung liegt. Dieser Fehler kann provisorisch behoben werden, indem der Spannungsregler entfernt wird, die Spannungsversorgungen somit getrennt werden und anstelle dessen, die Versorgung des Mikrocontrollers über die Programmierschnittstelle gespeist wird.

7.2 Offene Themen

Die grafische Benutzeroberfläche ist derzeit in der Lage einen vorher definierten String über die UART Schnittstelle des Raspberry Pi's an einen Mikrocontroller zu senden. Die bestehenden Menüoptionen (Slider und Dropdown-Menü) existieren und sind bedienbar. Allerdings verfügen diese über keine bestehende Funktion, um das Bewegungsmuster des Fahrzeuges einstellen zu können.

Um den Raspberry Pi mit möglichst wenig Verdrahtung in den CAN-Bus zu integrieren, muss eine weitere Platine für die UART auf CAN-Bus Umsetzung entworfen werden.

Zur Behebung des genannten Fehlers auf der Platine, muss dessen Design bzw. Hardwaredimensionierung überarbeitet werden.

Zur mechanischen Stabilitätsverbesserung der Plattform, sollte die Aufhängung der Schrittmotoren gegebenenfalls Maßgefertigt werden.

7.3 Erweiterungsmöglichkeiten

Durch Abschließen der offenen Themen, wird eine stabile Entwicklungsplattform für weitere Teilprojekte aus dem Themengebiet des autonomen Fahrens geboten. Der CAN-Bus bietet dazu eine stabile Datenverbindung, die die Implementierung von Sensoren oder weiteren Aktoren modular ermöglicht.