

Semesterprojekt zum Thema

---

# **Aufbau einer HIL/MIL-Simulationsumgebung für ein Frequenzumrichtersystem**

6. Semester Mechatronik

Verfasser: Daniel Hamm (928580), Lucas Röthemeier (928597)

Kontakt: [daniel.hamm@student.fh-kiel.de](mailto:daniel.hamm@student.fh-kiel.de), [lucas.roethemeier@student.fh-kiel.de](mailto:lucas.roethemeier@student.fh-kiel.de)

Dozent: Prof. Dr.-Ing. Jochen Immel

Ausgabetermin des Themas: 14.03.2019

Abgabetermin der Arbeit: 15.08.2019

## Inhaltsverzeichnis

i.	Abbildungsverzeichnis .....	3
ii.	Tabellenverzeichnis .....	4
1	Ziele .....	5
2	Zeitplan .....	6
3	Simulation eines LRC-Glieds .....	7
3.1	Ziel .....	7
3.2	Blockschaltbild .....	7
3.3	Hardware für SPI Kommunikation .....	8
3.4	Anbindung des Digital-Analog-Wandlers .....	8
3.4.1	Verbindungsplan .....	8
3.4.2	Ansteuerung DAC .....	8
3.4.3	MATLAB Simulink DAC .....	8
3.4.5	Umrechnung für DAC .....	9
3.4.6	Steuerwort für DAC .....	9
3.4.7	Ausgabe DAC .....	10
3.4.8	Fazit DAC .....	10
4	Anbindung des Analog-Digital-Wandlers .....	11
4.1	Verbindungsplan .....	11
4.2	Ansteuerung ADC .....	11
4.3	Umrechnung Digitalwert in Spannung .....	12
4.4	Ergebnisse ADC .....	13
4.5	Fazit ADC .....	13
4.6	Gesamtaufbau .....	14
5	Regelung eines Frequenzumrichters mit MATLAB .....	15
5.1	Ziel .....	15
5.2	Hardware .....	15
5.2.1	Verwendete Hardware .....	15
5.2.2	Einrichtung der Hardware .....	17
5.2.3	Belegungsplan SPI-Schnittstelle .....	17
5.3	MATLAB .....	18
5.3.1	Grundlagen .....	18
5.3.2	SPI-Schnittstelle .....	18
5.3.3	Blockschaltbild .....	19
5.3.4	Aufbau in MATLAB .....	20
5.4	Verwendete Software .....	22

5.4.1	Atom Editor .....	22
5.4.2	Ubuntu WSL (Windows Subsystem for Linux) .....	23
5.4.3	STM32 ST-Link Utility.....	24
5.4.4	BLDC Tool.....	25
5.5	Firmware .....	26
5.5.1	Grundlagen .....	26
5.5.2	Aufbau der Custom-App „SPI-Control“ .....	27
5.5.3	Weitere Anpassungen der Firmware.....	37
5.5.4	Aufgetretene Probleme .....	39
6	Fazit .....	40
7	Quellen .....	41

## i. Abbildungsverzeichnis

Abbildung 1:	Blockschaltbild des Simulationsaufbaus .....	7
Abbildung 2:	DAC-Bitmuster .....	8
Abbildung 3:	Signalbildung für DAC .....	9
Abbildung 4:	Signalverlauf DAC.....	10
Abbildung 5:	ADC Bitmuster .....	11
Abbildung 6:	24 Bit Signal Maskieren.....	12
Abbildung 7:	MATLAB-Modell für ADC Auswertung .....	12
Abbildung 8:	Simuliertes Signal und reales Messsignal .....	13
Abbildung 9:	Gesamtaufbau des DAC-/ADC-Systems .....	14
Abbildung 10:	Hardware LRC-Glied .....	14
Abbildung 11:	Umrichter-Platine mit STM32-Board und angeschlossenen Motorphasen.....	15
Abbildung 12:	Rückseite der Frequenzumrichter-Platine mit Treiberbaustein und MOSFETs .....	16
Abbildung 13:	Discovery-Board verbunden mit dem Debug-Eingang des STM32-Boards .....	16
Abbildung 14:	Gesamter Testaufbau des Systems.....	17
Abbildung 15:	Bitmuster SPI-Kommunikation.....	18
Abbildung 16:	Bitmuster SPI-Kommunikation mit ID-Bits.....	18
Abbildung 17:	Aufnahme des Datenpaketes mit dem Oszilloskope .....	19
Abbildung 18:	Blockschaltbild Signalverlauf MATLAB.....	19
Abbildung 19:	MATLAB-Modell Sendeblock.....	20
Abbildung 20:	MATLAB-Modell Empfangsblock.....	20
Abbildung 21:	MATLAB-Modell Signalverarbeitung.....	21
Abbildung 22:	Atom-Editor Übersicht mit Dateistruktur und Quellcode.....	22
Abbildung 23:	WSL Kommandozeile mit Debug-Ausgabe.....	23
Abbildung 24:	ST-Link Utility Übersicht mit verbundenem STM32.....	24
Abbildung 25:	ST-Link Utility Flashvorgang .....	24
Abbildung 26:	BLDC-Tool Übersicht .....	25
Abbildung 27:	BLDC-Tool Motorsteuerung	
Abbildung 28:	BLDC-Tool Serielle Schnittstelle .....	25

## ii. Tabellenverzeichnis

Tabelle 1: Zeitplanung .....	6
Tabelle 2: Verbindungsplan MC4922 mit 68-Pin Terminal Board .....	8
Tabelle 3: Verbindungsplan MC3204 mit 68-Pin Terminal Board .....	11
Tabelle 4: 8-Pin Stiftleiste Belegung STM32-Board (links nach rechts).....	17
Tabelle 5: 68-Pin Terminal Board HIL-System (SPI Channel 2) .....	17

## 1 Ziele

Die Ziele der Projektarbeit lassen sich in zwei große Teilziele unterteilen.

Das erste Ziel ist der Vergleich zwischen einem simulierten System und einem realen System. Hierfür wird ein LRC-Glied mit Hilfe von MATLAB simuliert und ausgewertet, für das reale System ist eine fertige Schaltung gegeben, an welcher die zu vergleichenden Parameter gemessen werden können.

Das zweite Ziel ist die Regelung eines Elektromotors mit Hilfe eines MATLAB-Modells. Für diesen Aufgabenbereich ist bereits Hardware in Form einer Umrichter-Platine und eines Computers mit Kommunikationsschnittstelle gegeben. Auf der gegebenen Umrichter-Platine befindet sich ein STM32, welcher die Regelung eines Motors übernimmt. Die Aufgabe ist es die vorhandene Firmware auf dem STM32 so zu modifizieren, dass eine Kommunikation nach außen mit Hilfe einer SPI-Schnittstelle möglich ist. Die Regelung des Motors erfolgt dann durch das MATLAB-Modell, hierfür wird ein Teil der Firmware des STM32 so verändert, dass der STM32 die Ströme, welche ihm vom MATLAB-Modell vorgegeben werden, steuert.

## 2 Zeitplan

- 0 – Aktueller Zeitpunkt
- 1 – Projektstart
- 2 – HIL-/MIL-System
- 3 – Einarbeitung in die Komponenten
- 4 – Recherche von Bauteilen und Komponenten
- 5 – Inbetriebnahme der vorhandenen RW-Platine
- 6 – Aufbau eines Simulink-Modells für ein LRC-Glied
- 7 – Vergleich HIL mit Modell
- 8 – Umrichter HW
- 9 – Einarbeiten in die vorhandene SW
- 10 – Inbetriebnahme der Umrichterplatine
- 11 – Analyse der Firmware des  $\mu$ C hinsichtlich der SPI-Schnittstelle
- 12 – Anbindung der HIL SPI-Schnittstelle an den  $\mu$ C
- 13 – Anpassung einer Simulink Vorlage für den vorliegenden Motor
- 14 – Inbetriebnahme des gesamten Systems
- 15 – Abschlusspräsentation

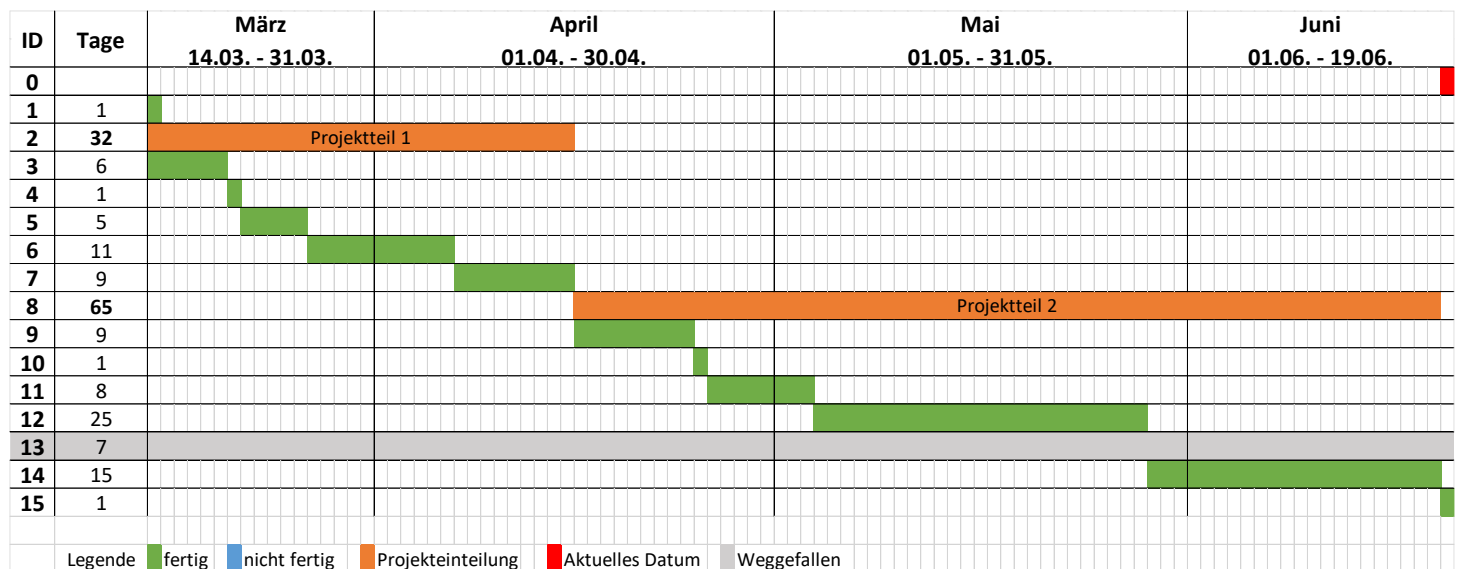


Tabelle 1: Zeitplanung

### 3 Simulation eines LRC-Glieds

#### 3.1 Ziel

Für den Vergleich zwischen einem realen und einem simulierten System werden die beiden Systeme gleichzeitig in Betrieb genommen und die Zielparameter dokumentiert. Zur vorhandenen Hardware gehört ein Windows-PC mit MATLAB. Auf diesem PC läuft ein MATLAB-Modell für das simulierte LRC-Glied. Das Modell wird dann auf einem zweiten Computer berechnet, welcher nur für das Berechnen der Simulation zuständig ist. Beide Computer kommunizieren über eine Ethernet-Schnittstelle. Der zweite Computer verfügt außerdem über ein FPGA Board, auf welchem das kompilierte Matlab Simulink Modell läuft. Für die Kommunikation ist eine SPI Schnittstelle initialisiert. Die Messwerte des realen Systems werden mit Hilfe eines ADCs eingelesen und dann über SPI an den Computer, welcher das System berechnet, gesendet. Außerdem werden die Messwerte des simulierten Systems über einen DAC in Form von Spannung ausgegeben, somit ist auch ein Vergleich der beiden Systeme mit Hilfe eines Oszilloskops möglich.

#### 3.2 Blockschaltbild

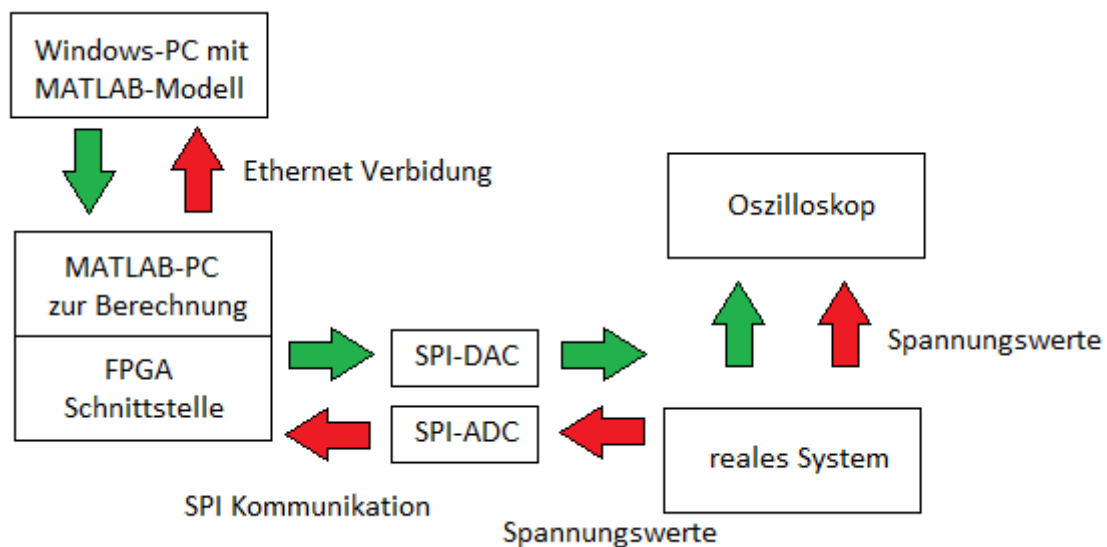


Abbildung 1: Blockschaltbild des Simulationsaufbaus

### 3.3 Hardware für SPI Kommunikation

Für den SPI-DAC wird ein MCP4922 verwendet, bei diesem Baustein handelt es sich um einen Digital-Analog-Wandler welcher eine Auflösung von 12 Bit hat und über SPI kommuniziert. Als SPI-ADC wird ein MCP3204 verwendet, dieser hat auch eine Auflösung von 12 Bit und kommuniziert über SPI.

### 3.4 Anbindung des Digital-Analog-Wandlers

#### 3.4.1 Verbindungsplan

Um externe Hardware wie den MCP4922 zu nutzen wird eine externe Spannungsquelle in Form eines Netzteils benötigt.

MCP4922 Pin-Nummer	Bedeutung	Speedgoat Pin-Nummer	Bedeutung
1	VDD	-	5V anlegen
2	NC	-	-
3	CS	25	SPI_CS
4	SCK	24	SPI_CLK
5	SDI	27	SPI_SDO
6	NC	-	-
7	NC	-	-
8	LDAC	26	GND
9	SHDN	-	5V anlegen
10	V_OUT_B	-	-
11	V_REF_B	-	-
12	VSS	26	GND
13	V_REF_A	-	5V anlegen
14	V_OUT_A	-	-

Tabelle 2: Verbindungsplan MC4922 mit 68-Pin Terminal Board

Beim Anschließen ist es wichtig zu beachten, dass die Masse des Netzteils mit der Masse des Speedgoat-FPGA verbunden wird.

#### 3.4.2 Ansteuerung DAC

Der Digital-Analog-Wandler wird mit einem 16 Bit-Muster über SPI angesteuert. Das 16-Bit Muster lässt sich in 4x4 Bit Datenwörter teilen, wobei das erste Datenwort ein 4 Bit Steuerwort ist. Die darauffolgenden 12 Bit enthalten das gewünschte Spannungssignal in Binärer Form.

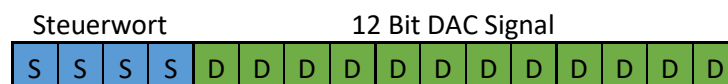


Abbildung 2: DAC-Bitmuster

#### 3.4.3 MATLAB Simulink DAC

Die Einbindung des Digital-Analog-Wandlers im Simulink-Modell erfolgt mit Hilfe einer kleinen Signal Umrechnung und der Kommunikationsschnittstelle des Speedgoat-Systems.

In dem verwendeten Modell für die Simulation der Sprungantwort des LRC-Glieds erhalten wir einen Ausgangs-Wert von 0 bis 1,2. Wobei die 1 als Ausgangswert für 100% der verwendeten Betriebsspannung entspricht. Der maximal Wert von 1,2 wird durch das aufschwingen des LRC-Glieds im Spitzenmoment erreicht.



### 3.4.5 Umrechnung für DAC

Das Signal für den Digital-Analog-Wandler kann maximal einen Wert von 12 gesetzten Bits haben (4095 Dezimal). Da das Signal aus dem System sich zwischen 0 und 1,2 bewegt muss eine Umrechnung entsprechend der Auflösung von 12 Bit erfolgen. Diese Signalumrechnung erfolgt mit Hilfe eines „Gain“-Blocks in MATLAB-Simulink. Bei der Variable Y handelt es sich um den Ausgangswert des zeitdiskreten Systems des LRC-Gliedes.

$$Y * \frac{2^{12}-1}{1,2} = DAC\ Signal \quad (1)$$

### 3.4.6 Steuerwort für DAC

Damit der Digital-Analog-Wandler auch funktioniert, muss dem Signalwert ein Steuerwort mitgegeben werden. In unserem Fall wird das Steuerwort 0x1000 mit dem DAC-Signal zum Wandler gesendet. Hierfür wird in MATLAB-Simulink das erhaltene DAC Signal (aus dem „Gain-Block“) und die Konstante 0x1000 mit einem „Add“-Block verbunden bzw. addiert. Somit hat das Signal für den DAC seine vorgesehene Signallänge von 16 Bits.

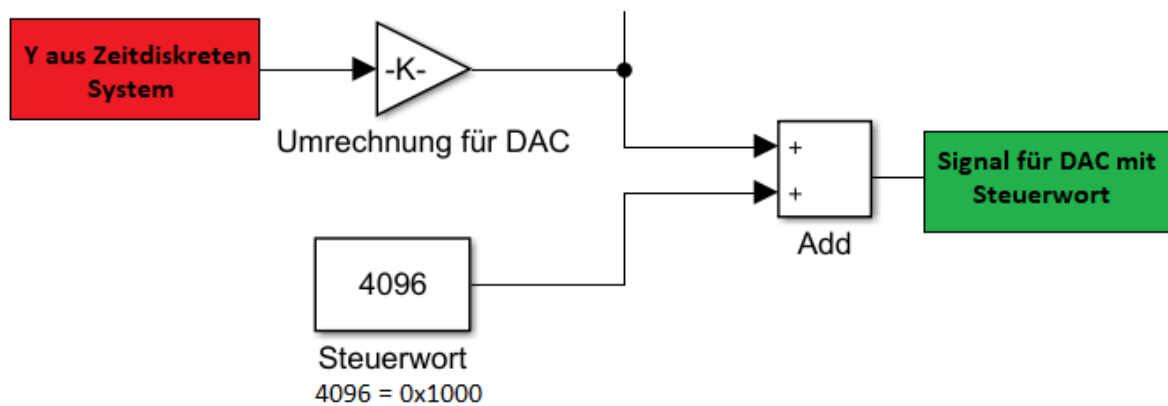


Abbildung 3: Signalbildung für DAC

### 3.4.7 Ausgabe DAC

Der Signalverlauf am DAC wurde dann mit Hilfe eines Oszilloskops aufgenommen und gespeichert.

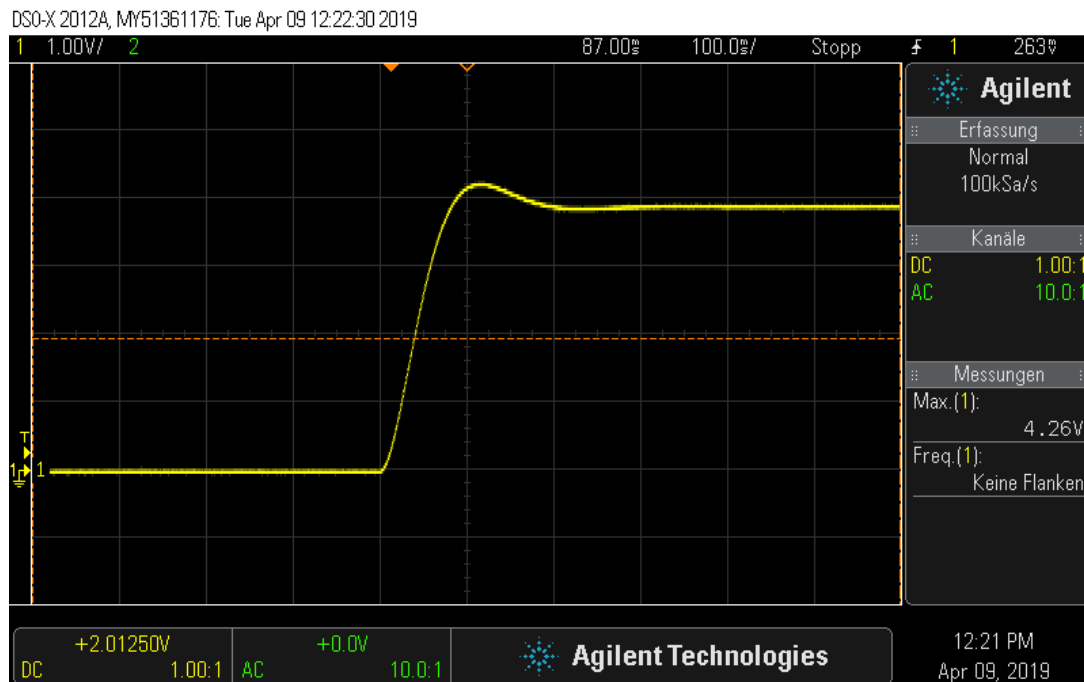


Abbildung 4: Signalverlauf DAC

In der Abbildung 4 ist der Signalverlauf des DAC zu erkennen. Der Verlauf ist identisch mit der realen Sprungantwort des LRC Glieds.

### 3.4.8 Fazit DAC

Der verwendete Digital-Analog-Converter lässt sich ohne Umstände in das HIL-/MIL-System einbinden. Wie in Abbildung 4 zu erkennen ist, entspricht das Signal der gewünschten Sprungantwort. Außerdem sind keine Unregelmäßigkeiten oder unsauberen Sprünge im Signalverlauf des DAC zu erkennen, was darauf hinweist, dass sowohl die Auflösung des DAC als auch die Übertragungsrate mehr als ausreichend sind.

## 4 Anbindung des Analog-Digital-Wandlers

### 4.1 Verbindungsplan

Um externe Hardware wie den MCP3204 zu nutzen wird eine externe Spannungsquelle in Form eines Netzteils benötigt.

MCP3204 Pin-Nummer	Bedeutung	Speedgoat Pin-Nummer	Bedeutung
1	CH0	-	Eingangssignal
2	CH1	-	-
3	CH2	-	-
4	CH3	-	-
5	NC	-	-
6	NC	-	-
7	DGND	26	GND
8	CS	21	SPI_CS
9	D_IN	22	SPI_SDO
10	D_OUT	23	SPI_SDI
11	CLK	20	SPI_CLK
12	AGND	26	GND
13	V_REF	-	5V anlegen
14	VDD	-	5V anlegen

Tabelle 3: Verbindungsplan MCP3204 mit 68-Pin Terminal Board

Beim Anschließen ist es wichtig zu beachten, dass die Masse des Netzteils mit der Masse des Speedgoat-FPGA verbunden wird.

### 4.2 Ansteuerung ADC

Der Analog-Digital-Wandler wird durch ein 24 Bit-Muster angesteuert. Die Übertragung lässt sich in 4 Blöcke unterteilen. So ist der erste Datenblock 5 Bit lang und besteht nur aus Nullen. Der nächste Datenblock ist nur ein Bit lang und enthält das Startbit, welches 1 sein muss. Im dritten Datenblock wird dann die Konfiguration des ADCs, bzw. das Steuerwort übermittelt. Dieses ist 4 Bit lang und es ist in unserem Fall jedes Bit gesetzt, hiermit wird der Eingangskanal gewählt (CH0). Der vierte und damit letzte Datenblock ist 14 Bit lang, hierbei handelt es sich nur um Don't-Care-Bits, die zur Vollständigen Kommunikation notwendig sind.

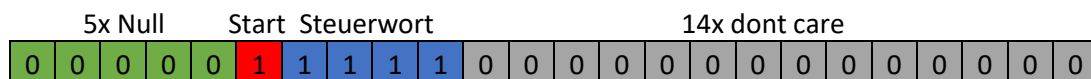


Abbildung 5: ADC Bitmuster

Der Analog-Digital-Wandler hat eine Signal-Auflösung von 12 Bit. Der ADC sendet aber ein Signal mit einer Länge von 24 Bit. In diesem Signal ist das 12 Bit lange Datenwort für den ADC-Wert enthalten. Durch Maskierung der empfangen 24 Bit mit dem entsprechenden Bitmuster erhalten wir das gewünschte Signal. Die Maskierung des Signals wird in MATLAB-Simulink mit einem „Bitwise AND“-Block realisiert.



Abbildung 6: 24 Bit Signal Maskieren

Mit einer 12 Bit Auflösung bewegt sich der empfangene Wert zwischen 0 und 4095. Somit muss das empfangene Signal mit dem Bitmuster 0xFFF logisch UND verknüpft werden (siehe Abbildung 5).

#### 4.3 Umrechnung Digitalwert in Spannung

Damit der erhaltene Signalwert auch richtig interpretiert werden kann muss eine Umrechnung erfolgen, welche den digitalen Zahlenwert in eine Spannung umwandelt, der digitale Zahlenwert wird durch die Variable  $X$  dargestellt. Bei der Variablen  $U_{MAX}$  handelt es sich um die maximale Spannung des Analog-Digital-Wandlers, in diesem Fall ist diese Spannung 5V. Es wird folgende Formel verwendet:

$$X * \frac{U_{MAX}}{2^{12}-1} = X * \frac{5V}{4095} = U_{ADC} \quad (2)$$

Durch die Variable  $U_{MAX}$  wird das Maximum von  $U_{ADC}$  festgelegt.

In MATLAB-Simulink wird folgendes Modell für den ADC aufgebaut.

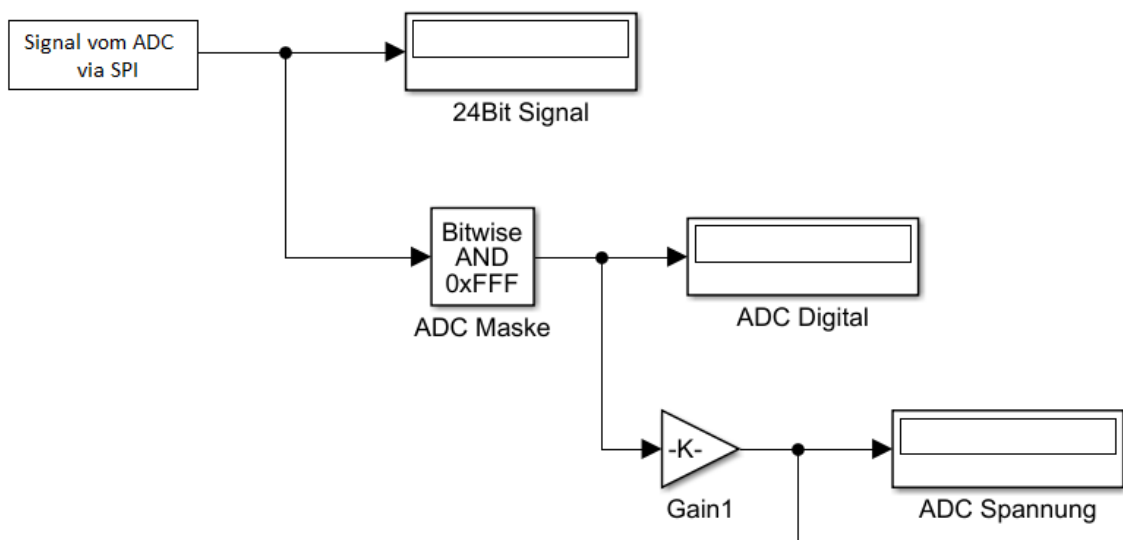


Abbildung 7: MATLAB-Modell für ADC Auswertung

#### 4.4 Ergebnisse ADC

Nach erfolgreicher Anbindung des ADC wurde folgender Signalverlauf dokumentiert.

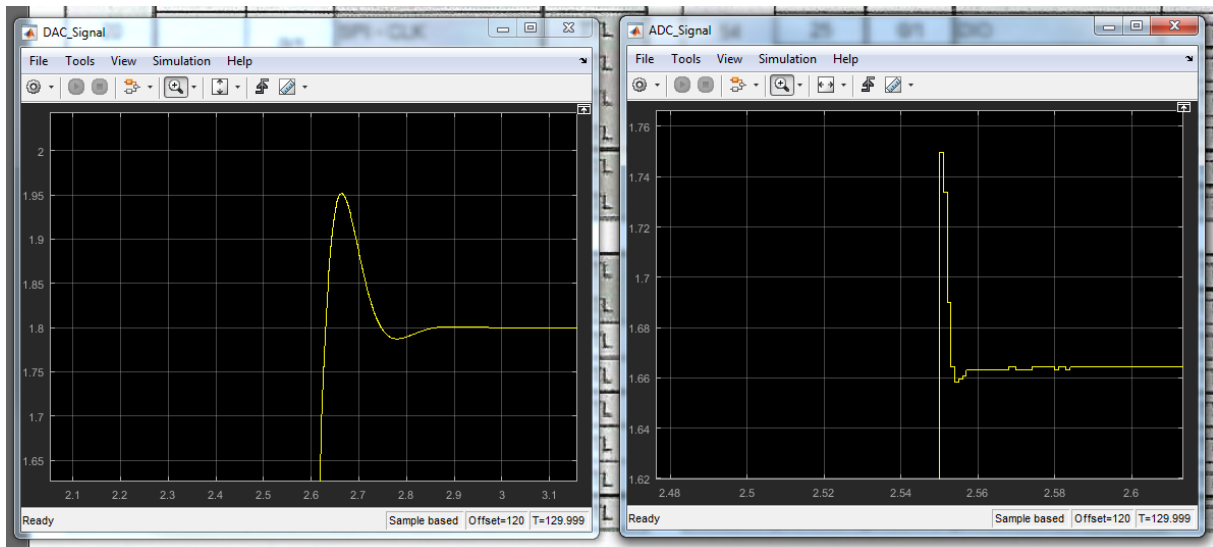


Abbildung 8: Simuliertes Signal und reales Messsignal

Aus Abbildung 8 ist der Signalverlauf der realen Schaltung und des simulierten Signals zu erkennen. Auf der linken Seite der Abbildung befindet sich die simulierte Sprungantwort des LRC-Glieds. Die rechte Seite der Abbildung zeigt die Messdaten des ADC an.

#### 4.5 Fazit ADC

Das reale Signal ließ sich problemlos mit Hilfe des verwendeten ADC einlesen. Leider ist die Auflösung des ADC wahrscheinlich zu groß was zu längeren Wandlungszeiten führt. Auf Grund dieser Wandlungszeiten ändert sich auch die Übertragungsrate bzw. ist diese zu langsam um einen sauberen Signalverlauf abbilden zu können. In Abbildung 8 sind eindeutige Stufenartige Sprünge im Signal zu erkennen, unter Verwendung eines anderen ADC könnte man das Signal sauberer mit kleineren Sprüngen darstellen.

#### 4.6 Gesamtaufbau

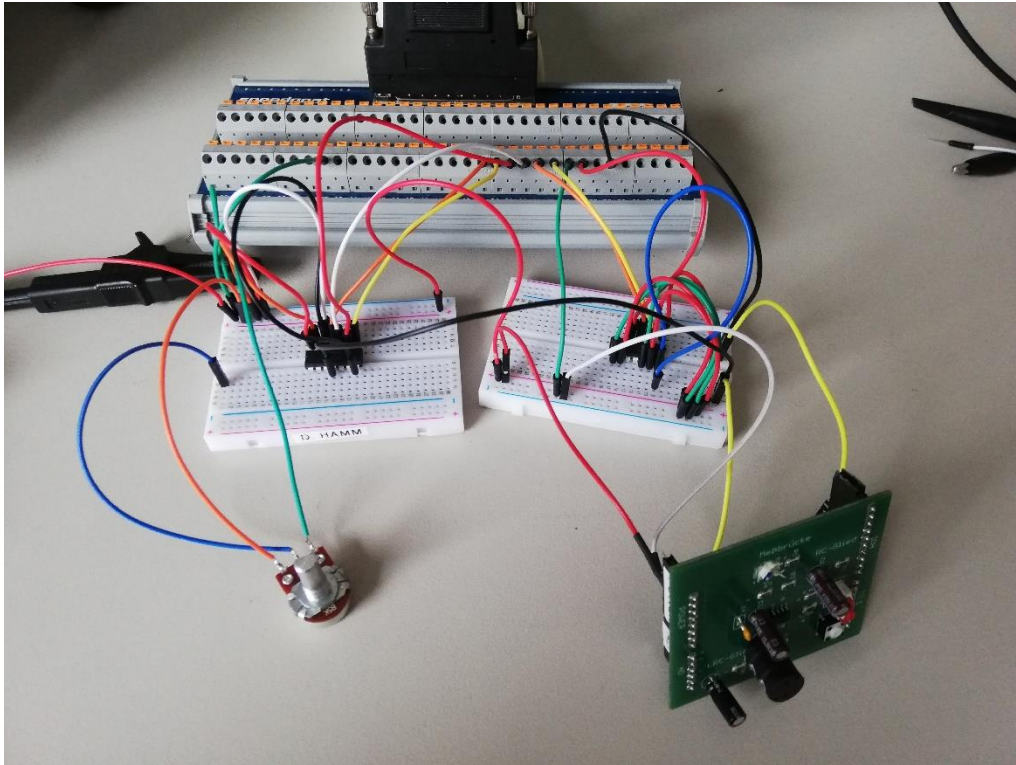


Abbildung 9: Gesamtaufbau des DAC-/ADC-Systems

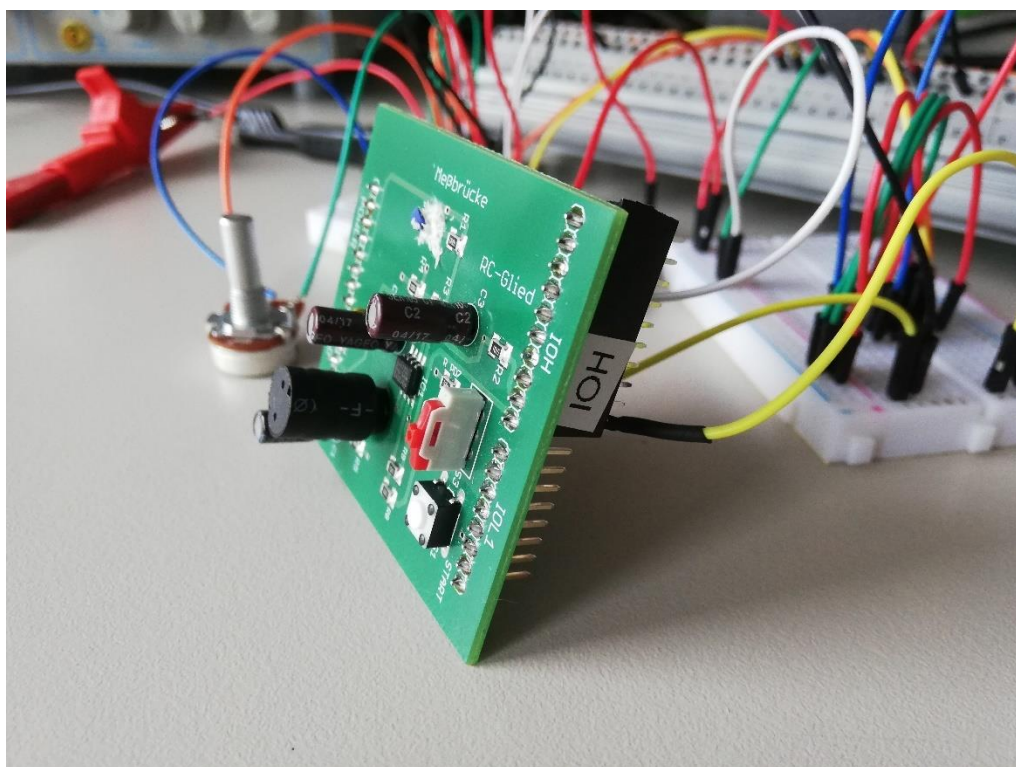


Abbildung 10: Hardware LRC-Glied

## 5 Regelung eines Frequenzumrichters mit MATLAB

### 5.1 Ziel

Das Ziel ist es eine vorhandene Frequenzumrichter-Platine so zu konfigurieren, dass eine Motorregelung von außerhalb der Platine realisiert werden kann. Auf dieser Platine befindet sich ein Mikrocontroller, welcher die Regelung eines Motors übernimmt. Damit das Ziel realisiert werden kann wird die Software auf dem Mikrocontroller umgeschrieben. Es wird eine SPI-Schnittstelle initialisiert, welche eine Kommunikation nach außen ermöglicht. Durch diese SPI-Schnittstelle können dann Parameter für die Motor-Regelung zwischen Frequenzumrichter-Platine und Computer (mit MATLAB) übertragen werden.

### 5.2 Hardware

#### 5.2.1 Verwendete Hardware

Zur Verwendung kommt eine Frequenzumrichter-Platine, die in einem früheren Projekt an der Fachhochschule Kiel entwickelt wurde. Sie basiert auf dem Open-Source ESC (Electronic Speed Controller) von Benjamin Vedder ([www.vedder.se](http://www.vedder.se)). Die Frequenzumrichter-Platine besteht im Wesentlichen aus dem Treiberbaustein DRV8302, mit welchem sich BLDC Motoren ansteuern lassen. Der Treiberbaustein wird von einem STM32F405 angesteuert, auf diesem befindet sich die passende Open-Source-Firmware zur Motorsteuerung, welcher wir in diesem Projekt unter anderem eine SPI-Schnittstelle hinzufügen.

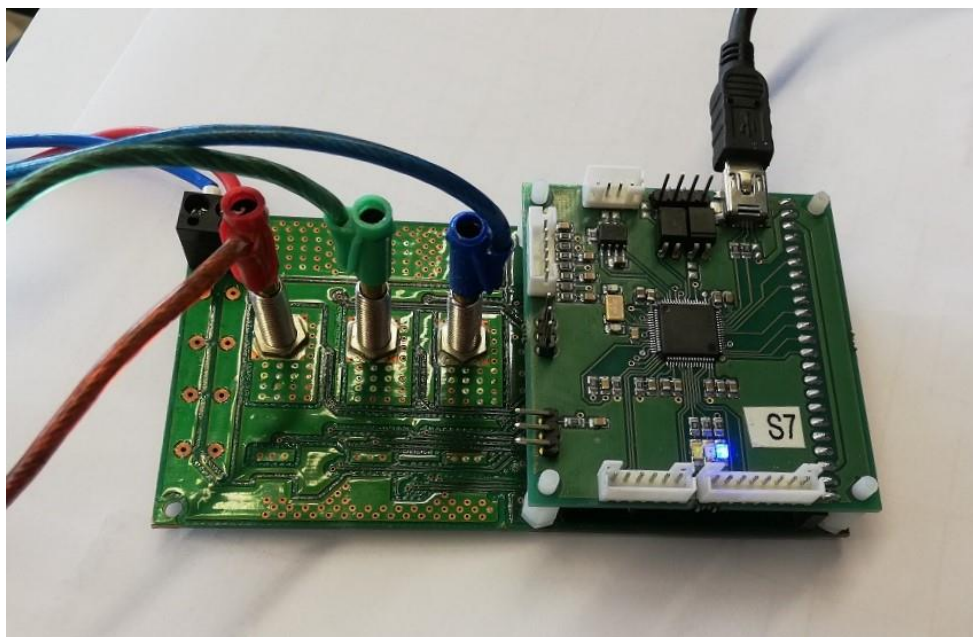


Abbildung 11: Umrichter-Platine mit STM32-Board und angeschlossenen Motorphasen



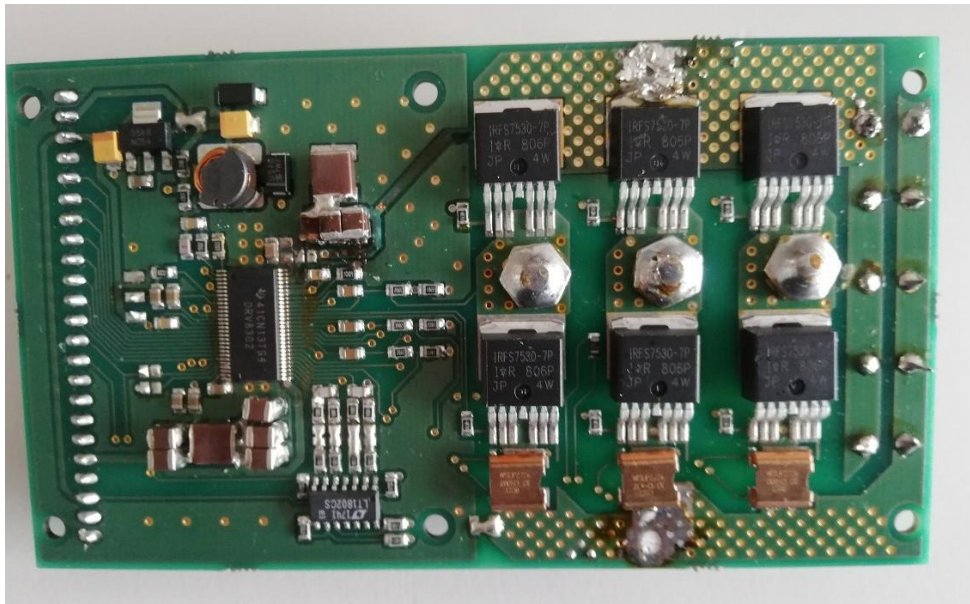


Abbildung 12: Rückseite der Frequenzumrichter-Platine mit Treiberbaustein und MOSFETs

Zum Programmieren des STM32F405 auf der Frequenzumrichter-Platine wird ein STM32F4DISCOVERY Board der Version MB997D verwendet. Auf dem Discovery Board befindet sich ein STM32F407, dieser wird jedoch nicht verwendet, die Platine wird ausschließlich als ST-Link Programmiereinheit genutzt um mit unserem STM32 zu kommunizieren und diesen mit unserer Firmware zu flashen. Sie wird zum einen per USB mit einem PC verbunden, auf dem das Programm ST-Link Utility installiert ist, es dient zum flashen und auslesen des STM32, zum anderen wird das Discovery Board mit dem Debug Eingang unseres STM32 verbunden.

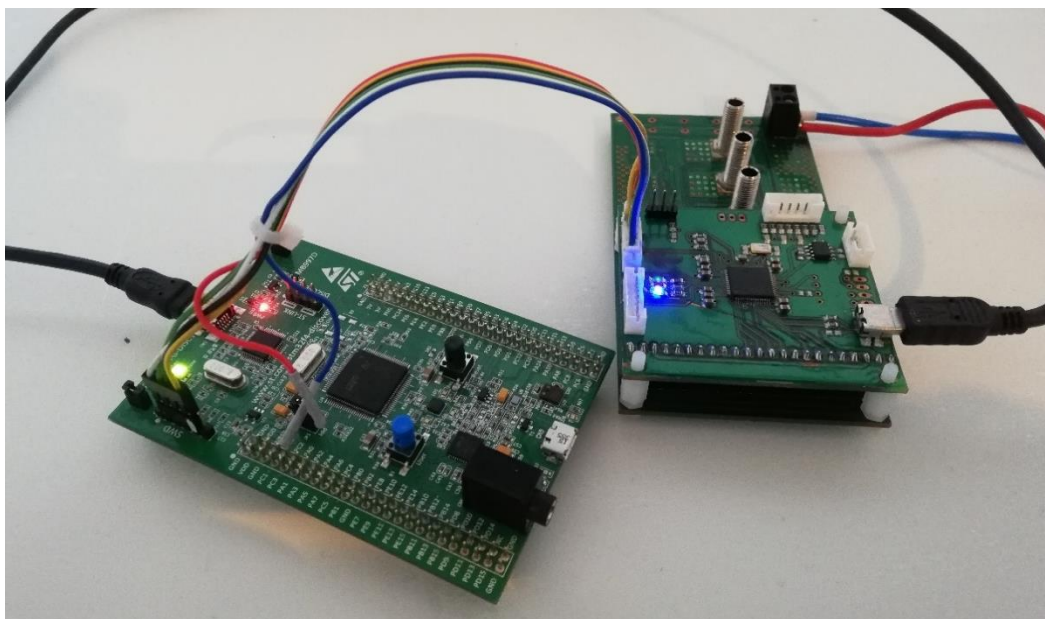


Abbildung 13: Discovery-Board verbunden mit dem Debug-Eingang des STM32-Boards



### 5.2.2 Einrichtung der Hardware

Zur einfachen Veranschaulichung ein Bild des gesamten Testaufbaus:

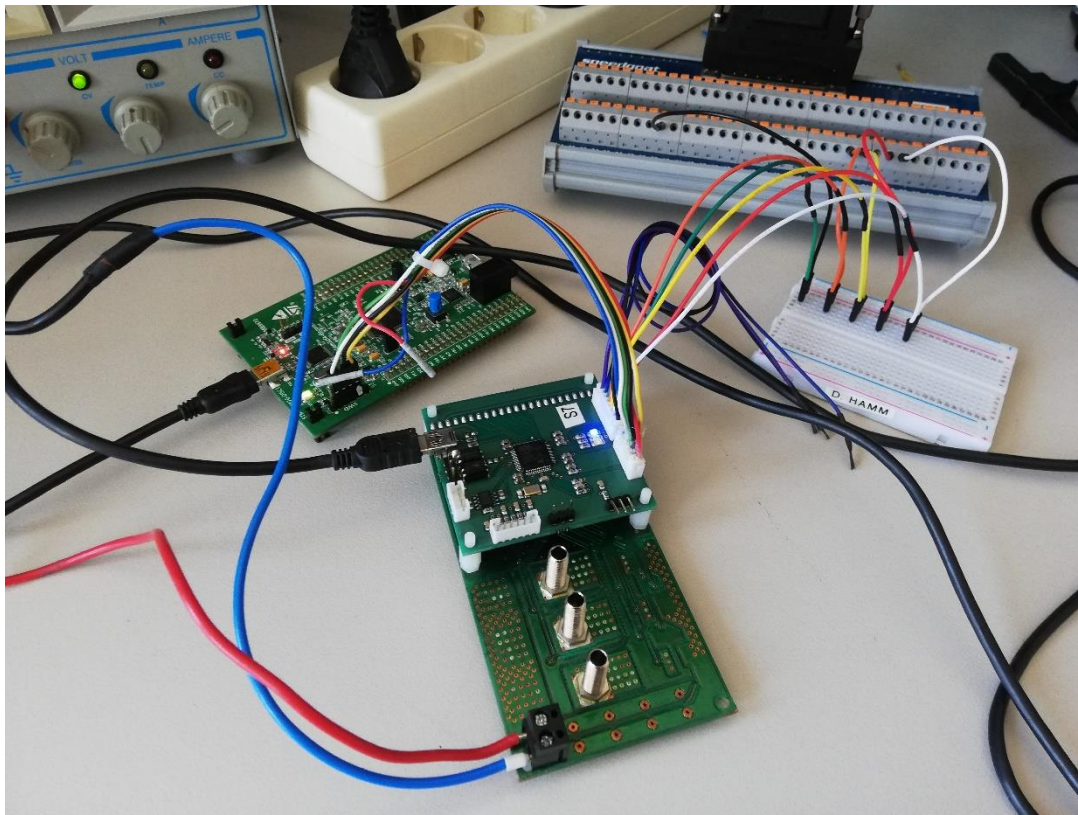


Abbildung 14: Gesamter Testaufbau des Systems

### 5.2.3 Belegungsplan SPI-Schnittstelle

#### STM32-Board:

Pin 1	Pin 2	Pin 3	Pin 4	Pin 5	Pin 6	Pin 7	Pin 8
5V	3V	GND	SPI-SCK	SPI-CS	SPI-MOSI	SPI-MISO	Nicht verbunden
Lila	Blau	Grün	Orange	Gelb	Weiß	Rot	Schwarz

Tabelle 4: 8-Pin Stiftleiste Belegung STM32-Board (links nach rechts)

#### 68-Pin Terminal Board:

Pin 9	Pin 24	Pin 25	Pin 27	Pin 28
GND	CLK	CS	SDO verbinden mit MISO	SDI verbinden mit MOSI
Grün	Orange	Gelb	Rot	Weiß

Tabelle 5: 68-Pin Terminal Board HIL-System (SPI Channel 2)

## 5.3 MATLAB

### 5.3.1 Grundlagen

Die Aufgabe der MATLAB-Seite in diesem Projekt ist es eine funktionierende Kommunikation zu ermöglichen und Parameter zu Senden und zu Empfangen. Im Weiteren soll es später möglich sein die empfangenen Ist-Werte in MATLAB auszuwerten und entsprechende Soll-Werte dem Mikrocontroller zu übergeben.

### 5.3.2 SPI-Schnittstelle

Für die Kommunikation zwischen STM und MATLAB wird eine SPI Schnittstelle auf beiden Seiten initialisiert. Die SPI Kommunikation ermöglicht eine hohe Übertragungsrate und eine frei wählbare Datenwortlänge. Bei der SPI-Kommunikation kommt es bei dem aktuellen Testaufbau jedoch leider vermehrt zur fehlerhaften Kommunikation. Der Grund hierfür sind wahrscheinlich die schlecht bis nicht geschirmten Signalleitungen, sowie die Verwendung eines Breadboards. Auf Grund von unterschiedlichen Interpretationen des Taktsignals auf Sender- und Empfängerseite verschieben sich die Signalwerte um ca. ein Bit nach links oder nach rechts.

#### a) Problemlösung

Das Problem kann auf zwei mögliche Arten gelöst werden, entweder mit einer geschirmten Hardware und einer sicheren Signalverbindung oder eine softwareseitige Anpassung, welche die Verschiebung des Bitmusters erkennt. Zur Lösung des Problems wurde die Software auf dem Mikrocontroller und in MATLAB so angepasst, dass im Fehlerfall die Verschiebung im Bitmuster zu erkennen ist.

#### b) Funktionsweise SPI-Kommunikation

Um die Verschiebung des Bitmusters zu erkennen wurden dem Signal 4 weitere Bits zugewiesen, von denen zwei Bits so genannte Null-Bits sind und die übrigen zwei Start und Stoppbit sind. Sollte sich das Signal nun verschieben, geht eins der Null-Bits verloren und vom wichtigen Inhalt des Signals bleiben die Bits erhalten. Durch die Start- und Stoppbits ist es dann möglich den Anfang der Kommunikation und auch das Ende der Kommunikation (als Redundanz) zu bestimmen.



Abbildung 15: Bitmuster SPI-Kommunikation

Von den übrigen 16 Bits aus dem 20 Bit Signalwort werden dann noch 3 Bits als Identifikationsbits verwendet. Somit ist es möglich mehrere unterschiedliche Parameter zu übertragen und diese dann auch fehlerfrei unterscheiden zu können.



Abbildung 16: Bitmuster SPI-Kommunikation mit ID-Bits

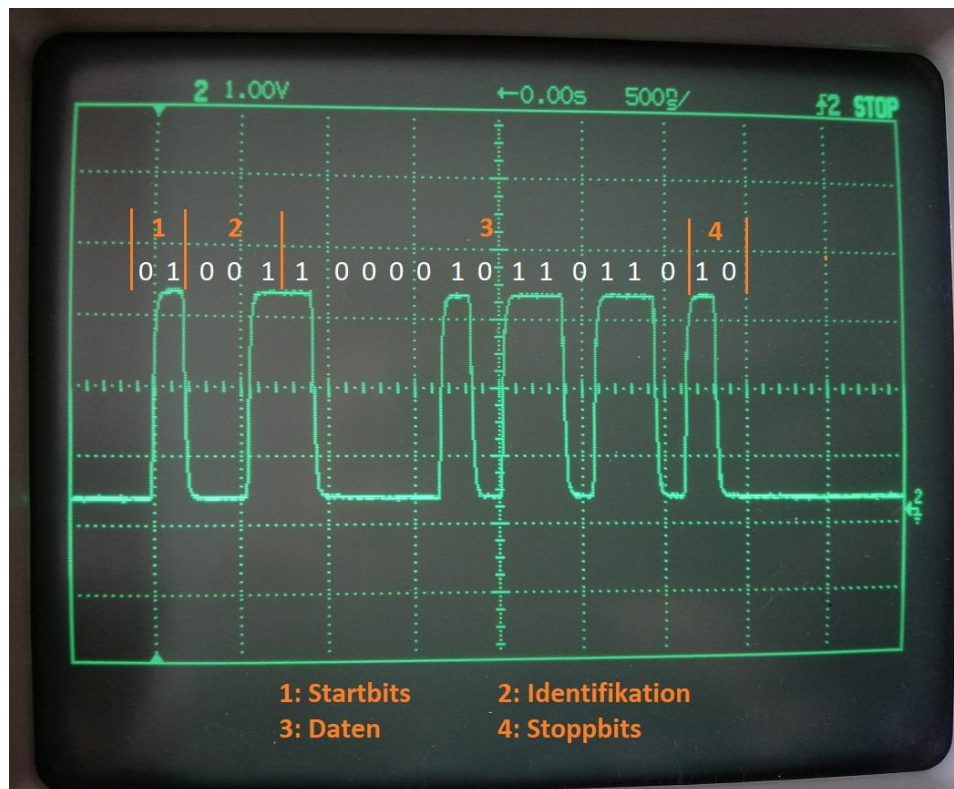


Abbildung 17: Aufnahme des Datenpaketes mit dem Oszilloskope

### 5.3.3 Blockschaftbild

Im folgenden Blockschaftbild wird der vereinfachte Ablauf der Signalverarbeitung und der Signalübermittlung auf der MATLAB Seite präsentiert.

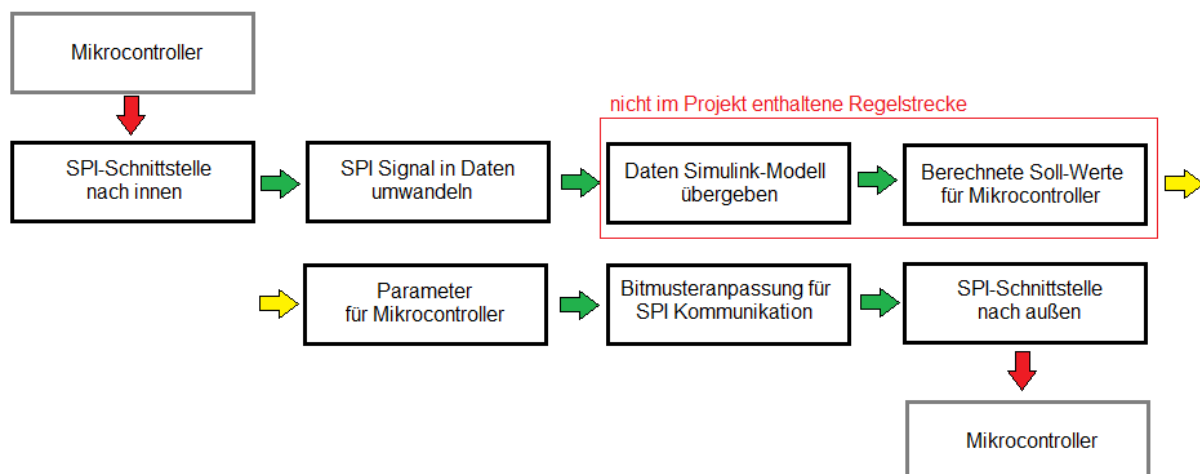


Abbildung 18: Blockschaftbild Signalverlauf MATLAB

### 5.3.4 Aufbau in MATLAB

In den folgenden Abbildungen ist der Aufbau des MATLAB-Modells zu erkennen. Zu beachten ist hierbei, dass der Regelstrecken Block, wie bereits in Abbildung 15 beschrieben wurde, nicht eingebunden ist. In dem Modell kann momentan ein Parameter Übertragen und ein Parameter Empfangen und anschließend dargestellt werden. Bei der aktuellen Signaleingabe handelt es sich um den Soll-Stromwert für den angeschlossenen Motor, der Strom wird hierbei in mA übermittelt. Das bedeutet eine Eingabe von 500 übergibt den Mikrocontroller den Befehl den Motorstrom auf 500mA zu setzen. Maximal kann momentan ein Wert von 4095 auf Grund der verwendeten Länge des Datenwortes eingegeben werden. Durch eine einfache Umrechnung kann man jedoch problemlos die Steuerung bis 10A erweitern, mit einer Genauigkeit von ca. 2mA.

Um das MATLAB-Modell möglichst ausführlich und übersichtlich zu erklären wird dieses in kleinere Blöcke unterteilt.

#### a) Sendeblock

In dem ersten Block geht es um die Soll-Wert Übertragung, wo momentan ein Konstanten-Block angebunden ist, kann eine Regelstrecke ihren entsprechenden Soll-Wert in Richtung Mikrocontroller übermitteln. Momentan kann man hier den Soll-Strom für den angeschlossenen Motor in mA eingeben. Wie bereits im vorherigen Absatz beschrieben können nur Werte von 0 – 4095 eingegeben werden. Der Wert wird dann in dem folgenden Funktionsblock von 16 Bit in ein 20 Bit Signal mit Start-/Stopp- und Nullbits umgewandelt.

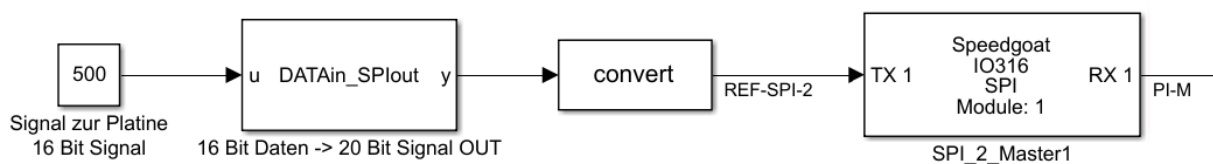


Abbildung 19: MATLAB-Modell Sendeblock

#### b) Empfangsblock

Bei dem zweiten Block wird das empfangene Bitmuster mit Hilfe eines weiteren Funktionsblocks von 20 Bit in 16 Bit umgewandelt. Somit erhalten wir ein Signal welches dann im weiteren Verlauf ohne Probleme als 16 Bit Signal verarbeitet werden kann. Bei Übertragung von mehreren Signalen werden von den 16 Bit die reservierten 3 Bit zur Signaltyp Identifikation hier ausgewertet, in unserem Fall ist dies aber nicht notwendig.

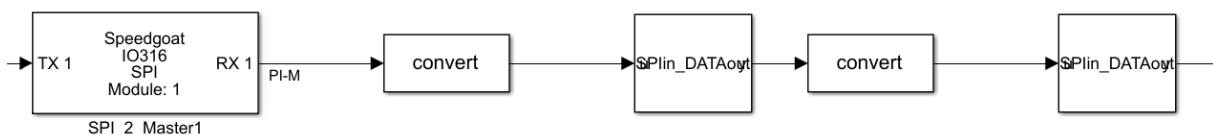


Abbildung 20: MATLAB-Modell Empfangsblock

### c) Signalverarbeitung

In dem letzten Block findet dann die Verarbeitung des empfangenen Signals statt. Da unser Ist-Wert auch negativ sein kann, wurde dieser auf Mikrocontroller Seite um die Hälfte des möglichen Maximalwertes angehoben, dadurch werden nur positive Werte übertragen. Das Signal muss aber in diesem Fall nach fertiger Übertragung wieder auf den realen Wert angepasst werden, hierfür wird ein einfacher Offset Abgleich verwendet bei welchen der Parameterwert um die vorherige Erhöhung wieder reduziert wird. Nach dem „ADD-Block“ würde dann die Übergabe an dem vorliegenden Regelsystem erfolgen, in unserem Fall werden die erhaltenen Werte digital dargestellt um sicherzustellen, dass die Übertragung erfolgreich war.

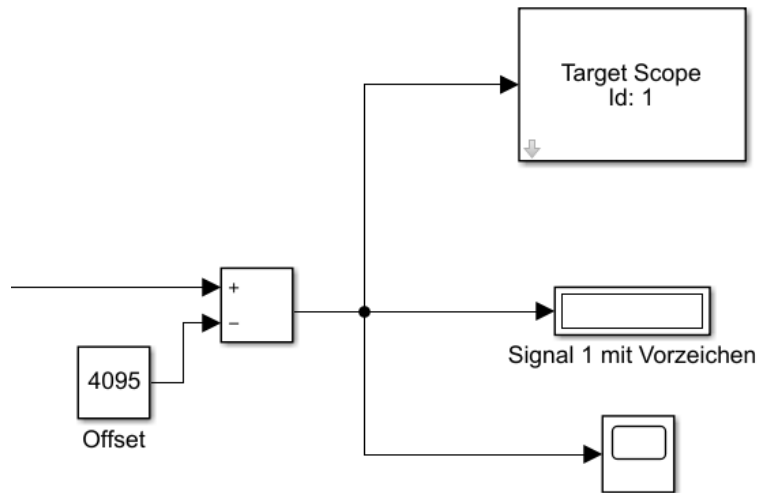


Abbildung 21: MATLAB-Modell Signalverarbeitung

## 5.4 Verwendete Software

### 5.4.1 Atom Editor

Zum Programmieren der Firmware wurde der Open-Source-Texteditor „Atom“ genutzt. Dieser ermöglicht uns eine übersichtliche Projektstruktur, eine einfache Bearbeitung des Quellcodes und eine sehr gute Suchfunktion zur schnellen Analyse aller Quelldateien innerhalb des Projektordners.

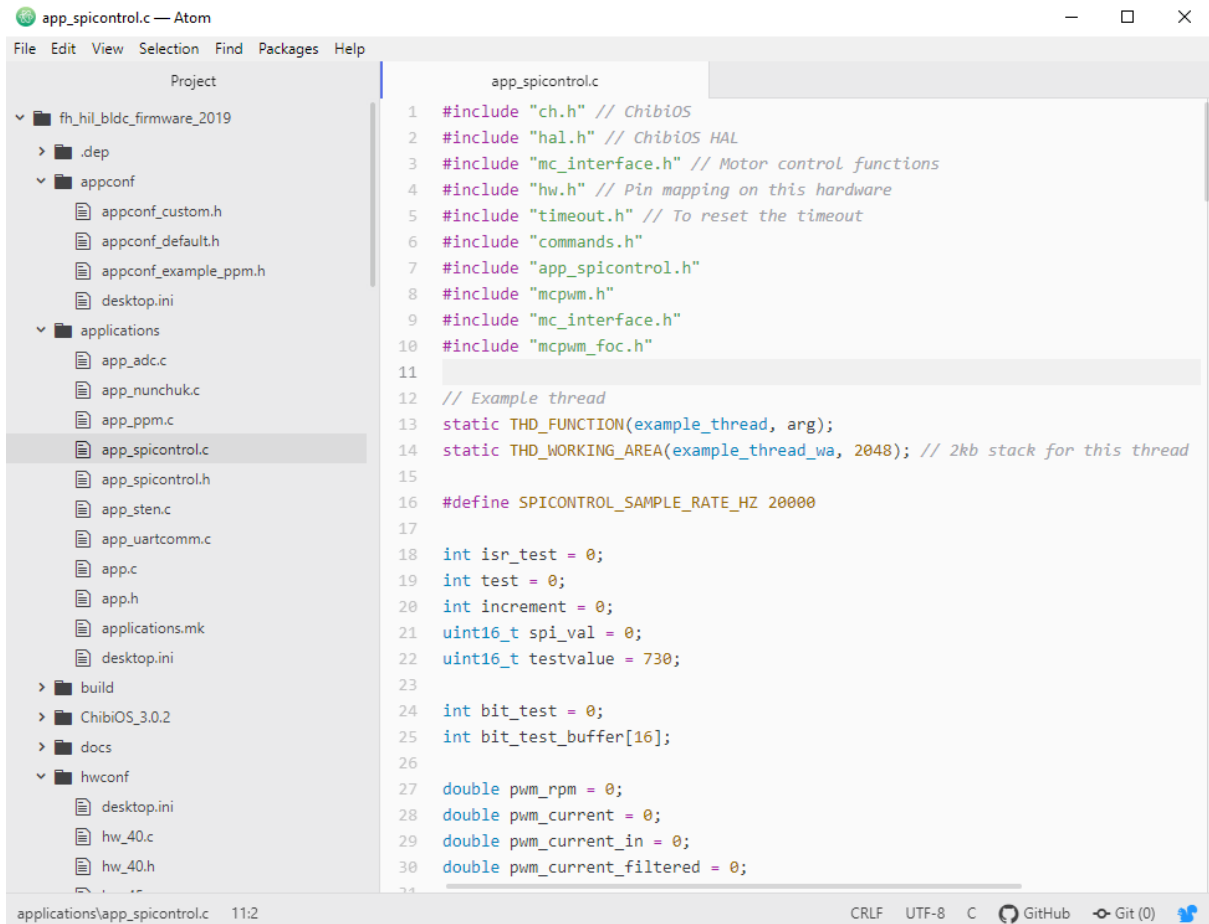
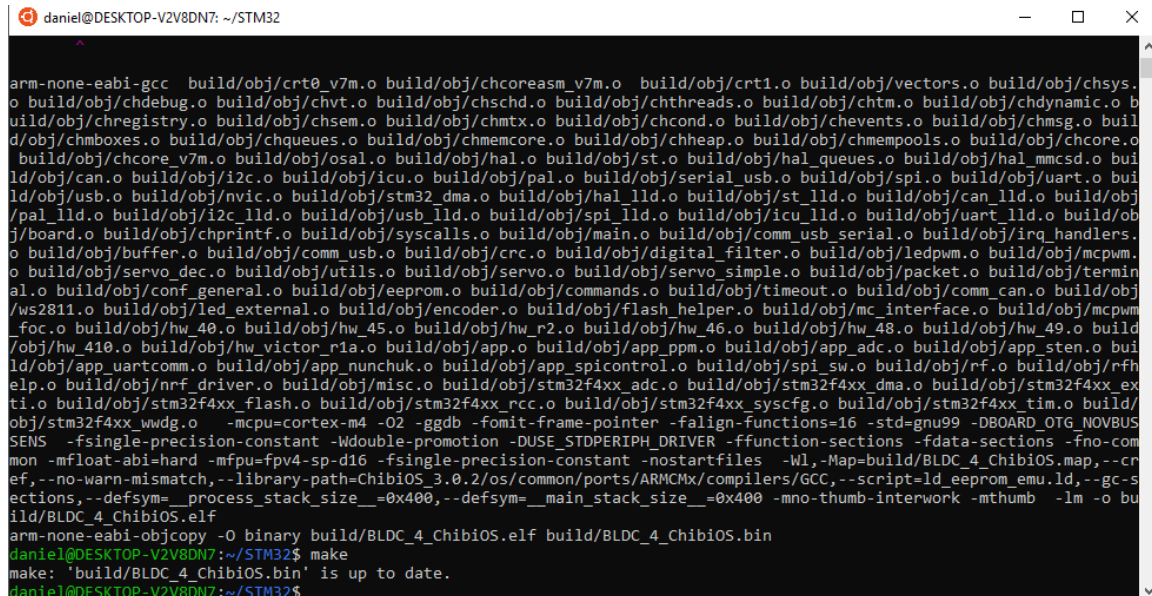


Abbildung 22: Atom-Editor Übersicht mit Dateistruktur und Quellcode



### 5.4.2 Ubuntu WSL (Windows Subsystem for Linux)

Da die vorhandene Open-Source-Firmware darauf ausgelegt ist in einer Linux Entwicklungsumgebung kompiliert zu werden, wurde sich dazu entschieden das „Windows-Subsystem for Linux“ basierend auf der Linux Distribution Ubuntu zu nutzen, welches eine Linux-Kommandozeile in Windows bereitstellt, so ist es möglich die gewohnten Linux Befehle zum kompilieren (z.B. make) unter Windows zu nutzen.



```
daniel@DESKTOP-V2V8DN7: ~/STM32
arm-none-eabi-gcc build/obj/crt0_v7m.o build/obj/chcoreasm_v7m.o build/obj/crt1.o build/obj/vectors.o build/obj/chsys.o
build/obj/chdebug.o build/obj/chvt.o build/obj/chschd.o build/obj/chthreads.o build/obj/ctm.o build/obj/chdynamic.o b
uild/obj/chregistry.o build/obj/chsem.o build/obj/chmtx.o build/obj/chcond.o build/obj/chevents.o build/obj/chmsg.o bui
ld/obj/chmboxes.o build/obj/chqueues.o build/obj/chmemcore.o build/obj/chheap.o build/obj/chmempools.o build/obj/chcore.o
build/obj/chcore_v7m.o build/obj/osal.o build/obj/hal.o build/obj/st.o build/obj/hal_queues.o build/obj/hal_mmcstd.o bui
ld/obj/can.o build/obj/i2c.o build/obj/icu.o build/obj/pal.o build/obj/serial_usb.o build/obj/spi.o build/obj/uart.o bui
ld/obj/usb.o build/obj/nvic.o build/obj/stm32_dma.o build/obj/hal_ll.o build/obj/st_ll.o build/obj/can_ll.o build/obj
/pal_ll.o build/obj/i2c_ll.o build/obj/usb_ll.o build/obj/spi_ll.o build/obj/icu_ll.o build/obj/uart_ll.o build/obj
j/board.o build/obj/chprintf.o build/obj/syscalls.o build/obj/main.o build/obj/comm_usb_serial.o build/obj/irq_handlers.o
build/obj/buffer.o build/obj/comm_usb.o build/obj/crc.o build/obj/digital_filter.o build/obj/ledpwm.o build/obj/mcpwm.o
build/obj/servo_dec.o build/obj/utls.o build/obj/servo.o build/obj/servo_simple.o build/obj/packet.o build/obj/termin
al.o build/obj/conf_general.o build/obj/eeprom.o build/obj/commands.o build/obj/timeout.o build/obj/comm_can.o build/obj
/ws2811.o build/obj/led_external.o build/obj/encoder.o build/obj/flash_helper.o build/obj/mc_interface.o build/obj/mcpwm
.foc.o build/obj/hw_40.o build/obj/hw_45.o build/obj/hw_r2.o build/obj/hw_46.o build/obj/hw_48.o build/obj/hw_49.o build
/obj/hw_410.o build/obj/hw_victor_r1a.o build/obj/app.o build/obj/app_ppm.o build/obj/app_adc.o build/obj/app_sten.o bui
ld/obj/app_uartcomm.o build/obj/app_nunchuk.o build/obj/app_spicontrol.o build/obj/spi_sw.o build/obj/rf.o build/obj/rfh
elp.o build/obj/nrf_driver.o build/obj/misc.o build/obj/stm32f4xx_adc.o build/obj/stm32f4xx_dma.o build/obj/stm32f4xx_ex
ti.o build/obj/stm32f4xx_flash.o build/obj/stm32f4xx_rcc.o build/obj/stm32f4xx_syscfg.o build/obj/stm32f4xx_tim.o build
obj/stm32f4xx_wwdg.o -mcpu=cortex-m4 -O2 -ggdb -fomit-frame-pointer -falign-functions=16 -std=gnu99 -DBOARD_OTG_NOVBUS
SENS -fsingle-precision-constant -Wdouble-promotion -DUSE_STDPERIPH_DRIVER -ffunction-sections -fdata-sections -fno-com
mon -mfloat-abi=hard -mfpu=fpv4-sp-d16 -fsingle-precision-constant -nostartfiles -Wl,-Map=build/BLDC_4_ChibiOS.map,-cr
ef,-no-warn-mismatch,-library-path=ChibiOS_3.0.2/os/common/ports/ARMCortexM/compilers/GCC,--script=ld_eeprom_emu.ld,-gc-s
ections,-defsym=__process_stack_size__=0x400,-defsym=__main_stack_size__=0x400 -mno-thumb-interwork -mthumb -lm -o bu
ild/BLDC_4_ChibiOS.elf
arm-none-eabi-objcopy -O binary build/BLDC_4_ChibiOS.elf build/BLDC_4_ChibiOS.bin
daniel@DESKTOP-V2V8DN7:~/STM32$ make
make: 'build/BLDC_4_ChibiOS.bin' is up to date.
daniel@DESKTOP-V2V8DN7:~/STM32$
```

Abbildung 23: WSL Kommandozeile mit Debug-Ausgabe

#### a) Installation

Aktivierung des Windows-Subsystems for Linux:

1. Öffnen der Windows PowerShell als Administrator
2. Eingabe des Befehls zum Aktivieren des Tools
  - a. `Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-Subsystem-Linux`
3. Windows neu starten
4. Ubuntu aus dem Microsoft Store installieren

Installation der Abhängigkeiten innerhalb der Ubuntu Kommandozeile

1. Ausführen von Updates
  - a. `sudo apt update -y && sudo apt upgrade -y`
2. GCC ARM Toolchain und Compiler installieren
  - a. `sudo add-apt-repository ppa:team-gcc-arm-embedded/ppa`
  - b. `sudo apt update`
  - c. `sudo apt install gcc-arm-embedded make gcc`

Zum kompilieren der Firmware muss nun in das Stammverzeichnis des Projektordners gewechselt werden, hier liegt das Makefile, welches alle Arbeitsschritte zum Übersetzen, Linken etc. ausführt.

1. Wechseln in das Projektverzeichnis
  - a. `cd /mnt/c/Users/daniel/FH/Frequenzumrichter/fh_hil_bldc_firmware_2019`
2. Kompilieren
  - a. `make`

### 5.4.3 STM32 ST-Link Utility

Der STM32 Mikrocontroller auf der Frequenzumrichter-Platine wird mit Hilfe der Software „STM32 ST-Link Utility“ geflasht. Neben dem flashen ist es ebenfalls möglich die Integrität der vorhandenen Firmware auf dem Mikrocontroller zu verifizieren (Vergleich von Prüfsummen) und im Zweifel den gesamten Speicher zu löschen und dann neu zu bespielen.

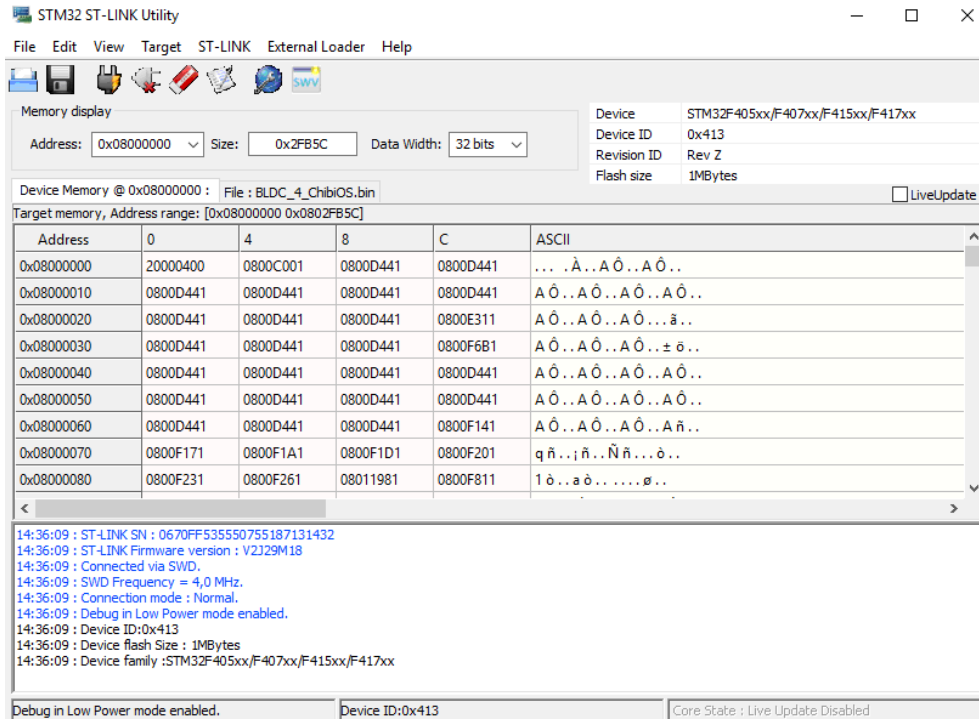


Abbildung 24: ST-Link Utility Übersicht mit verbundenem STM32

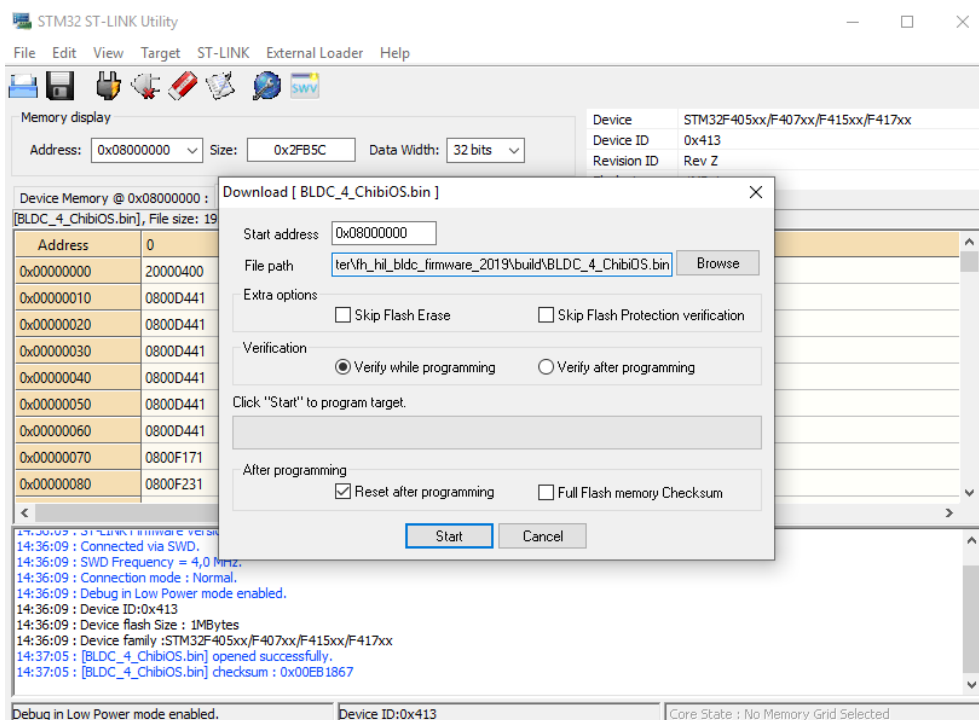


Abbildung 25: ST-Link Utility Flashvorgang



#### 5.4.4 BLDC Tool

Das BLDC Tool ist eine von Benjamin Vedder bereitgestellte Open-Source-Software zur einfachen Motorsteuerung. Die Software stellt via USB eine serielle Verbindung zum Frequenzumrichter her, es lassen sich jegliche Motorparameter einstellen und auswerten. Weiterhin ist eine schnelle und einfache Überwachung der aktuellen Messwerte, wie Motorströme, in einem Live-Diagramm darstellbar. Mit Hilfe eines integrierten Terminals lassen sich Debug-Nachrichten, welche von der Firmware gesendet werden, auswerten. Somit ist eine einfache und schnelle Fehlersuche möglich.

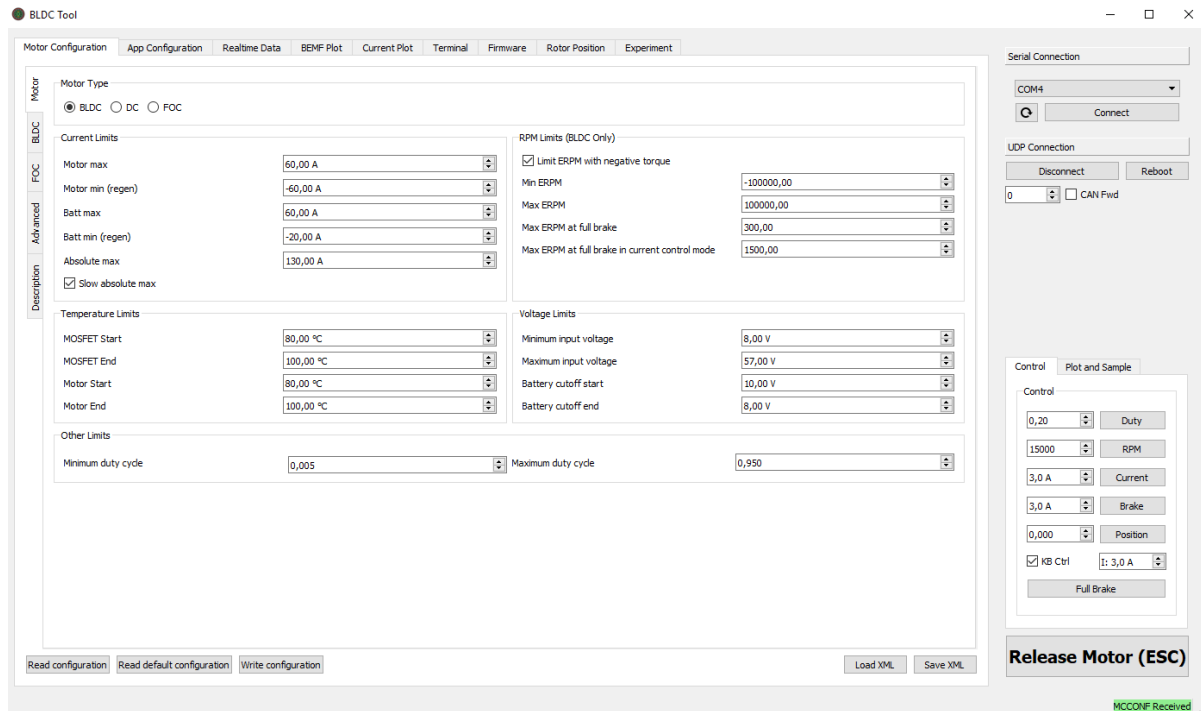


Abbildung 26: BLDC-Tool Übersicht

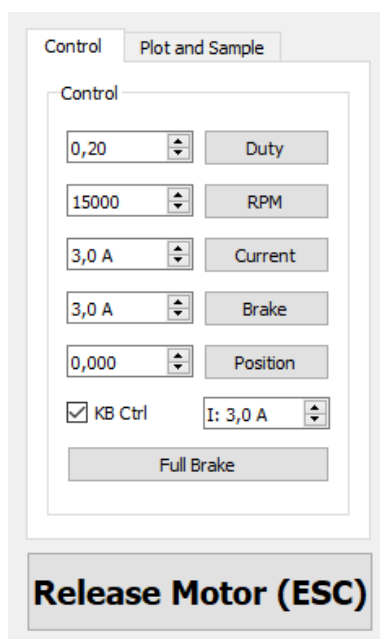


Abbildung 27: BLDC-Tool Motorsteuerung

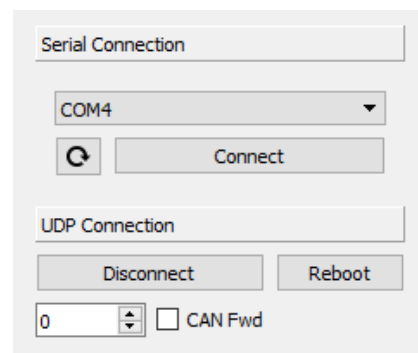


Abbildung 28: BLDC-Tool Serielle Schnittstelle

## 5.5 Firmware

### 5.5.1 Grundlagen

Die Firmware zur Motorsteuerung ist ebenfalls eine unter einer Open-Source-Lizenz bereitgestellte Software von Benjamin Vedder. Sie ist öffentlich zugänglich auf dem Onlinedienst GitHub. Die zum jetzigen Zeitpunkt neuste Version ist die Firmware 3.57, wir verwenden allerdings für unsere Motorsteuerung die etwas ältere Version 2.18, da die uns bereitgestellte Frequenzumrichter-Platine auf dieser Version aufbaut und die entsprechende Pinbelegung hat. Die Basis der Firmware ist das Echtzeit Betriebssystem ChibiOS 3.0.2, welches unter anderem für die ARM Cortex-M4 Architektur des verwendeten Mikrocontrollers STM32F405 zur Verfügung steht.

Unser Ziel ist es eine SPI-Schnittstelle in die vorhandene Softwarestruktur zu implementieren, mit welcher wir anschließend wichtige Live-Daten des Motors und des Frequenzumrichters an das HIL-System übermitteln und dann entsprechend auswerten können.

Wir machen uns eine bereits implementierte Funktion zu Nutze, welche es uns ermöglicht eine sog. „Custom App“ zu erstellen. Eine App besteht aus globalen und lokalen Funktionen zur Motorsteuerung, sowie einem Thread. Mit Hilfe des Threads lassen sich Aufgaben parallel zur restlichen Prozessstruktur der Firmware abarbeiten.

Es gibt bereits einige vorgefertigte Apps, welche über das BLDC-Tool auswählbar sind, so gibt es die Möglichkeit die ADC-App auszuwählen. Nach Auswahl der ADC-App und dem Anschluss eines Potentiometers an dem Frequenzumrichter wäre es so möglich die Drehzahl des Motors durch Drehen an dem Potentiometer zu steuern.

Von Haus aus besteht die Option einen Hall-Encoder über eine SPI-Schnittstelle an den Frequenzumrichter anzubinden. Die Funktionen zur SPI-Kommunikation des Hall-Encoders finden sich in der Datei „encoder.c“ wieder. Wir haben die Funktionen zur SPI-Kommunikation hieraus übernommen, an unsere Gegebenheiten angepasst und in unsere Custom-App implementiert.

### 5.5.2 Aufbau der Custom-App „SPI-Control“

Die hier aufgeführten Funktionen befinden sich in der Datei „app\_spicontrol.c“. Ebenfalls zur Custom-App gehört die Datei „app\_spicontrol.h“, diese wird hier nicht weiter erläutert, da diese nur die Funktionsprototypen der hier folgenden Funktionen enthält.

#### a) Includes und Defines

##### Quelltext

```
#include "ch.h" // ChibiOS
#include "hal.h" // ChibiOS HAL
#include "mc_interface.h" // Motorsteuerungs-Funktionen
#include "hw.h" // Pinbelegung unserer Hardware
#include "timeout.h" // Timeout Reset
#include "commands.h" // printf für BLDC Terminal
#include "app_spicontrol.h" // Funktionsdeklarationen
#include "mcpwm.h" // Funktionen zum Messen von Live-Daten
#include "mcpwm foc.h" // Funktionen zum Messen von Live-Daten
#include "mc_interface.h" // Funktionen zum Motorsteuerung

// Example thread
static THD_FUNCTION(example_thread, arg);
static THD_WORKING_AREA(example_thread_wa, 2048); // 2kb Stack für den Thread

// Einstellung der Samplerate in HZ
#define SPICONTROL_SAMPLE_RATE_HZ 1000

uint32_t spi_val = 0;
uint32_t testvalue = 0;
uint32_t merker[2];

double pwm_rpm = 0;
double pwm_current = 0;
```

##### Aufgabe

Zunächst werden alle wichtigen Header-Dateien in die Custom-App eingebunden:

„ch.h“ und „hal.h“ enthalten wichtige Funktionen des Basisbetriebssystem ChibiOS.

„mc\_interface.h“ enthält Funktionen, die unter anderem über das BLDC Interface aufgerufen werden können, so kann etwa der Motor über eine Funktion gestartet oder gestoppt werden.

„hw.h“ enthält die Pinbelegung der Hardwareversion unserer Frequenzumrichter-Platine.

„timeout.h“ wird für der Timer benötigt.

„commands.h“ enthält den printf-Befehl zur Ausgabe von Nachrichten an das BLDC-Terminal.

„app\_spicontrol.h“ enthält die Funktionsprototypen der Custom-App.

„mcpwm.h“ und „mcpwm\_foc.h“ enthalten Funktionen zum Auslesen von Live-Daten der Frequenzumrichter-Platine.

„mc\_interface.h“ enthält Funktionen zur Motorsteuerung.

Nach dem Einbinden der Header-Dateien, wird dem Thread zum einen ein Name zugewiesen und zum anderen wird dem Stack des Threads ein Speicherplatz von 2kB zugeschrieben. Es folgt die Definition der Samplerate, mit ihr lässt dich die Taktfrequenz des SPI-Signals einstellen.

## b) app\_spicontrol\_init

### Quelltext

```
// SPI Control Initialisierung
void app_spicontrol_init(void) {

    // Definition der Timer Struktur
    TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;

    // Setzen der SPI-Pins als Ein- bzw. Ausgänge
    palSetPadMode(HW_SPI_PORT_MISO, HW_SPI_PIN_MISO, PAL_MODE_INPUT);
    palSetPadMode(HW_SPI_PORT_MOSI, HW_SPI_PIN_MOSI, PAL_MODE_OUTPUT_PUSHPULL);
    palSetPadMode(HW_SPI_PORT_SCK, HW_SPI_PIN_SCK, PAL_MODE_OUTPUT_PUSHPULL |
PAL_STM32_OSPEED_HIGHEST);
    palSetPadMode(HW_SPI_PORT_NSS, HW_SPI_PIN_NSS, PAL_MODE_OUTPUT_PUSHPULL |
PAL_STM32_OSPEED_HIGHEST);

    // Setzen der NSS (CS), SCK und MOSI Pins auf LOW
    palClearPad(HW_SPI_PORT_NSS, HW_SPI_PIN_NSS);
    palClearPad(HW_SPI_PORT_SCK, HW_SPI_PIN_SCK);
    palClearPad(HW_SPI_PORT_MOSI, HW_SPI_PIN_MOSI);

    // Clock enable
    HW_ENC_TIM_CLK_EN();

    // Timer Einstellungen
    TIM_TimeBaseStructure.TIM_Prescaler = 0;
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
    TIM_TimeBaseStructure.TIM_Period = ((168000000 / 2 / SPICONTROL_SAMPLE_RATE_HZ)
- 1);
    TIM_TimeBaseStructure.TIM_ClockDivision = 0;
    TIM_TimeBaseStructure.TIM_RepetitionCounter = 0;
    TIM_TimeBaseInit(HW_ENC_TIM, &TIM_TimeBaseStructure);
    TIM_ITConfig(HW_ENC_TIM, TIM_IT_Update, ENABLE);
    TIM_Cmd(HW_ENC_TIM, ENABLE);

    // Setzt die Priorität des Interrupt Handlers und aktiviert ihn
    nvicEnableVector(HW_ENC_TIM_ISR_CH, 6);

    // Start des Threads
    chThdCreateStatic(example_thread_wa, sizeof(example_thread_wa),
    NORMALPRIO, example_thread, NULL);
}
```

### Aufgabe

Die Funktion „app\_spicontrol\_init“ startet die wichtige Initialisierung unserer Custom-App, sie beschäftigt sich zunächst mit der Definition der Timer-Struktur, welche wichtig für die Taktung des SPI-Signals ist. Weiterhin werden die benötigten Ein- und Ausgänge für die SPI-Kommunikation gesetzt. Nach der Aktivierung des Taktes, werden wichtige Einstellungen der Timer-Struktur gesetzt, so unter anderem die Periode des Timers, welche über die STM32 Taktfrequenz von 168MHz und einer vorher definierten Samplerate berechnet wird. Über die Einstellung der Samplerate kann entsprechend die Frequenz des SPI-Taktes definiert werden. Anschließend muss noch die Priorität des Interrupt Handlers definiert werden, bei falsch eingestellter Priorität der ISR wird der STM32 abstürzen. Zum Schluss wird noch der Thread der Custom-App gestartet.

### c) daten\_to\_spi

#### Quelltext

```
// Übertragung der Messdaten in unsere SPI-Struktur
// 12 Bit auf 20 Bit
// Setzen von Start-, End-, Vorzeichen- und Identifikations-Bit
uint32_t daten_to_spi(int32_t daten, uint32_t typ){
    if(daten<(-4095))daten=-4095;    //Daten dürfen nicht kleiner 4095 sein

    daten+=4095;                    // offset
    daten<<=2;                      // wegen startstopbit
    daten=daten|0x40002;            // setzen von start/stop

    //Setzen des Identifikations-Bits
    typ<<=15;
    daten=daten|typ;

    return daten;
}
```

#### Aufgabe

Mit Hilfe der Funktion „daten\_to\_spi“ wird ein 13 Bit Datenwort, welches z.B. einen Messwert enthalten kann, in ein 20 Bit Signal gewandelt, um zusätzlich ein definiertes Start- und End-Bit zu setzen. Hierzu wird das Datenwort zunächst um 2 Bit nach links verschoben und dann ODER verknüpft. Außerdem werden noch 3 Bits für die Definition des Signals verwendet, die notwendigen Bits werden auch wieder mit einer ODER Verknüpfung im Signal gesetzt.

#### d) *spicontrol\_tim\_isr*

##### Quelltext

```
// Interrupt Service Routine (ISR)
// Senden / Empfangen von SPI-Daten bei Interrupt
void spicontrol_tim_isr(void) {
    uint16_t pos = 0;

    //SPI-NSS (Chip Select) auf LOW
    spicontrol_begin();

    //Gleichzeitiger Transfer und Empfang von SPI-Daten
    spicontrol_transfer(&pos, &testvalue, 1);

    //SPI-NSS (Chip Select) auf HIGH
    spicontrol_end();

    //Übertragung empfangener Daten an globale Variable
    spi_val = pos;
}
```

##### Aufgabe

Die Funktion „spicontrol\_tim\_isr“ stellt die Interrupt Service Routine der Custom-App dar, ihre wesentliche Aufgabe ist die SPI-Kommunikation zu takten. Die ISR ruft zunächst die Funktion „spicontrol\_begin“ auf, welche den Chip Select Pin der SPI-Schnittstelle auf LOW setzt. In dem Zeitfenster, in welchem Chip Select auf LOW gesetzt ist, kann die SPI-Schnittstelle Daten senden und empfangen. Deshalb wird anschließend die Funktion „spicontrol\_transfer“ aufgerufen, diese ist für das Senden und Empfangen der Daten zuständig. Ihr wird zum einen die Adresse der Variablen übergeben, in die die empfangenen Daten geschrieben werden sollen. Weiterhin wird ihr die Adresse übergeben, an der die Daten stehen, die gesendet werden sollen. Nach erfolgreichem Datentransfer wird der Chip Select Pin wieder auf HIGH gesetzt und die SPI-Kommunikation somit beendet. Am Schluss werden die empfangenen Daten noch in eine globale Variable geschrieben.

#### e) spicontrol\_get\_val

##### Quelltext

```
// Funktion zum Auslesen der Empfangen SPI-Daten
uint16_t spicontrol_get_val(void) {
    return spi_val;
}
```

##### Aufgabe

Mit der Funktion „spicontrol\_get\_val“ ist ein einfaches Auslesen der empfangenen Daten möglich.

#### f) spicontrol\_transfer

##### Quelltext

```
// Funktion zum Senden und Empfangen von SPI-Daten
void spicontrol_transfer(uint16_t *in_buf, const uint16_t *out_buf, int length) {
    for (int i = 0; i < length; i++) {
        uint16_t send = out_buf ? out_buf[i] : 0xFFFF;
        uint16_t recieve = 0;

        for (int bit = 0; bit < 20; bit++) {
            palWritePad(HW_SPI_PORT_MOSI, HW_SPI_PIN_MOSI, send >> 19);
            send <<= 1;

            spicontrol_delay();
            palSetPad(HW_SPI_PORT_SCK, HW_SPI_PIN_SCK);
            spicontrol_delay();

            recieve <<= 1;
            if (palReadPad(HW_SPI_PORT_MISO, HW_SPI_PIN_MISO)) {
                recieve |= 1;
            }

            palClearPad(HW_SPI_PORT_SCK, HW_SPI_PIN_SCK);
            spicontrol_delay();
        }

        if (in_buf) {
            in_buf[i] = recieve;
        }
    }
}
```

##### Aufgabe

Die Übertragung der Daten erfolgt in Form einer einfachen „for-Schleife“ welche das Ausgangsbit entsprechend des Signals setzt und dann Taktet. Nach Übertragung vom ersten Bit verschiebt die „for-Schleife“ das Signal um ein Bit und legt das entsprechende neue Signal Bit an den Ausgang und Taktet erneut. Dieser Prozess wiederholt sich entsprechend der Signallänge (in unserem Fall 20mal) so oft bis jedes Bit des Signals übertragen worden ist. Da Senden und Empfangen parallel laufen wird das zu empfangene Signal auch in der „for-Schleife“ gebildet. Hierfür wird die gleiche Art und Weise der Signalverarbeitung wie beim Senden verwendet. Es wird immer das aktuelle Bit am Eingang eingelesen und nach dem Takt wird die Wertigkeit des nächsten Bits um ein Bit erhöht.

## Funktionsablauf

1. MSB vom Signalwert an Ausgang legen
2. Signalwert um ein Bit nach links schieben
3. Kurzes Delay für Übernahme des Bits am Ausgang
4. Pad für CLK-Signal auf HIGH setzen
5. Kurzes Delay für Übernahme des CLK Signals
6. Signalwert für Empfangen um ein Bit nach links schieben
7. Wenn Pin vom Signaleingang gesetzt wird LSB vom Empfangen Signalwert auf 1 gesetzt
8. Pad für CLK-Signal auf LOW setzen
9. Kurzes Delay für Übernahme des CLK Signals
10. Empfangenes Signal in Globaler Variable aktualisieren

Dieser Funktionsablauf wiederholt sich entsprechend der Signallänge bis alle Bits Übertragen bzw. empfangen worden sind.

## *g) spicontrol\_begin*

### Quelltext

```
// Funktion zum Setzen von SPI-NSS (CS) auf LOW
void spicontrol_begin(void) {
    palClearPad(HW_SPI_PORT_NSS, HW_SPI_PIN_NSS);
}
```

## Aufgabe

Die Funktion „spicontrol\_begin“ setzt den SPI-NSS Pin (Chip Select), auch SPI-CS genannt, auf LOW. Sie stellt den Start der SPI-Kommunikation dar. In dem Fenster zwischen LOW und HIGH des Chip Select Pin findet der Datenaustausch statt.



#### *h) spicontrol\_end*

##### Quelltext

```
// Funktion zum Setzen von SPI-NSS (CS) auf HIGH
void spicontrol_end(void) {
    palSetPad(HW_SPI_PORT_NSS, HW_SPI_PIN_NSS);
}
```

##### Aufgabe

Die Funktion „spicontrol\_end“ setzt den SPI-NSS Pin (Chip Select), auch SPI-CS genannt, auf HIGH. Sie stellt das Ende der SPI-Kommunikation dar. Somit ist kein Datenaustausch mehr möglich.

#### *i) spicontrol\_delay*

##### Quelltext

```
// Funktion zum Auslösen eines minimalen Delays
void spicontrol_delay(void) {
    __NOP();
    __NOP();
    __NOP();
    __NOP();
}
```

##### Aufgabe

Die Funktion „spicontrol\_delay“ erzeugt ein minimales Delay. Sie wird in der „spicontrol\_transfer“ Funktion benötigt und war bereits in der Firmware integriert. Die Funktion „\_\_NOP“ ruft eine sog. Nulloperation auf, also ein Befehl, der nichts bewirkt. Der Befehl erzeugt jedoch eine minimale Zeitverzögerung, da der Prozessor diesen Befehl trotzdem abarbeiten muss.

j) *spi\_to\_data*

#### Quelltext

```
// Funktion zur Anwendung unseres Algorithmus auf die empfangenen SPI-Daten
// Überprüfung der Position des Start- und End-Bits
uint32_t spi_to_data(void){
    uint32_t daten=spicontrol_get_val();

    if(daten&0x80000){
        if(daten&0x0004 && (daten&0x0003) == 0){
            daten=daten&0x7FFF8;
            daten>>=3;
            return daten;
        }
    }
    else{
        if(daten&0x40000){
            if(daten&0x0002 && (daten&0x0001) == 0){
                daten=daten&0x3FFFC;
                daten>>=2;
                return daten;
            }
        }
        else{
            if(daten&0x20000){
                if(daten&0x0001 && (daten&0xC0000) == 0){
                    daten=daten&0x1FFFE;
                    daten>>=1;
                    return daten;
                }
            }
        }
    }

    return 4096;
}
```

#### Aufgabe

Die Funktion filtert das empfangene SPI-Signal und sucht den entsprechenden Datenblock. Hierfür wird mit ein paar „If-Abfragen“ überprüft an welcher Stelle im 20 Bit Datenmuster, das gewünschte Datenmuster beginnt. Entsprechend der Position des Startbits und des Stopbits wird dann das empfangene Signal um bis zu drei Bit zur Seite verschoben und alle irrelevanten Bits werden mit Hilfe einer Bitmaske ignoriert. Diese Funktion macht so aus den 20 Bit langen Signal ein verwendbares 16 Bit Signal und gibt dieses dann zurück. Im Fehlerfall wird der Fehlercode 2 von der Funktion zurückgegeben.

#### k) Thread

##### Quelltext

```
// Thread
static THD_FUNCTION(example_thread, arg) {
    (void)arg;

    chRegSetThreadName("APP_EXAMPLE");

    uint32_t newdata=0;

    for(;;) {

        // Auslesen der Live-Daten des Frequenzumrichters
        // Funktionen aus der "mcpwm.c" und "mcpwm_foc.c"

        //FOC Get Current
        //Aktuellen Stromwert des Motors auslesen
        pwm_current = mcpwm_foc_get_tot_current();

        // Empfangene SPI-Daten in Variable speichern
        newdata=spi_to_data();
        if(!(newdata>=4096))
            commands_printf("SPI Receive: %d", newdata);
        commands_printf("SPI Receive: %lf", ((double)newdata/1000.0));

        // Merker, zur Abfrage ob sich der zu setzende Stromwert geändert hat
        if(merker[0]!=newdata){
            merker[0]=newdata;
            merker[1]=1;
        }else{
            merker[1]=0;
        }

        //FOC Set Current
        //Funktion zum Setzen eines neuen Stromwertes
        //Diese wird nur einmalig bei einer Änderung aufgerufen
        if(merker[1]){
            commands_printf("Set Current: %d", merker[1]);
            mc_interface_set_current((float)newdata/1000.0);
        }

        //Debug: Loopen des gesendeten Signals von Matlab
        //pwm_current=(float)newdata/1000.0;

        //Debug: Setzen eines festen Stromwertes
        //pwm_current = -1;

        //Stromsignal in mA umrechnen
        pwm_current*=1000;

        //Aufrufen der Funktion zum Senden der Messdaten an Matlab
        testvalue=daten_to_spi((pwm_current),1);
        commands_printf("pwm_current: %lf", pwm_current);
        commands_printf("SPI Send: %d", testvalue);

        // Kleines Verzögerung, damit der Thread den STM nicht überlastet
        chThdSleepMilliseconds(100);

        // Reset des Timeouts
        timeout_reset();
    }
}
```

## Aufgabe

Der Thread der Custom-App läuft parallel zur restlichen Prozessstruktur der Firmware. Innerhalb des Threads läuft eine Endlosschleife. Der Thread wird zum einen benötigt, um ständig die Messwerte des Frequenzumrichters auszulesen, z.B. Stromwerte, zum anderen werden hier Befehle zum Senden und Empfangen der SPI-Kommunikation aufgerufen. Ebenfalls wird er zur Fehlersuche verwendet, so können Messwerte und Daten, die über die SPI-Schnittstelle empfangen wurden, mit Hilfe des „commands\_printf“ Befehls in dem Terminal des BLDC Tools ausgegeben werden. Sehr wichtig ist der Aufruf der Funktion „chThdSleepMilliseconds“, dieser verzögert die Ausführung des Threads minimal, damit dieser den Prozessor nicht überlastet, sollte die Zeit zu gering oder im schlimmsten Fall gar nicht gesetzt sein, stürzt der STM32 direkt nach dem Starten ab. Der Befehl „timeout\_reset“ ist ein bereits integrierter Befehl der Firmware und wird für die Funktion des Timers benötigt.

Das Auslesen der Motorströme geschieht über die Funktion „mcpwm\_foc\_get\_tot\_current()“, dieser wird über unsere „daten\_to\_spi()“ Funktion verarbeitet und über SPI an den Matlab-PC gesendet.

Die Ansteuerung des Motors geschieht über das Setzen eines neuen Stromwertes, dieser wird über die Funktion „mc\_interface\_set\_current()“ verändert. Der neue Stromwert wird vom Matlab-PC in mA gesendet, dieser wird anschließend in A umgerechnet und entsprechend verarbeitet. Weiterhin wurde ein Merker integriert, mit dessen Hilfe die Funktion zum Setzen des Stromes nur einmalig aufgerufen wird.

### 5.5.3 Weitere Anpassungen der Firmware

Abgesehen von dem Aufbau unserer Custom-App mussten noch weitere Anpassungen der Firmware vorgenommen werden, so mussten einige Definitionen in verschiedenen Dateien angepasst werden. Folgende Dateien wurden angepasst:

#### a) *appconf\_custom.h* und *appconf\_default.h*

Quelltext

```
#define APPCONF_APP_TO_USE          APP_CUSTOM
```

Aufgabe

Hier wird die definiert, welche Standard-App beim Starten der Firmware genutzt werden soll. In diesem Fall unsere Custom-App „APP\_CUSTOM“.

#### b) *conf\_general.h*

Quelltext

```
#define FW_VERSION_MAJOR    2
#define FW_VERSION_MINOR    18

#define HW_VERSION_410
```

Aufgabe

Hier werden die Hardware- und Firmware-Versionen des Systems angegeben. Wir verwenden die Firmware 2.18, welche für die Hardware-Version 410 unserer Frequenzumrichter-Platine optimiert ist.

#### c) *hw\_410.h*

Diese Datei wurde nicht verändert, wird hier aber aufgeführt, da sie wichtige Definitionen für die Frequenzumrichter-Platine enthält, so finden sich hier jegliche Pinbelegungen des Systems, z.B. die der SPI-Schnittstelle:

```
// SPI pins
#define HW_SPI_DEV          SPID1
#define HW_SPI_GPIO_AF      GPIO_AF_SPI1
#define HW_SPI_PORT_NSS     GPIOA
#define HW_SPI_PIN_NSS      4
#define HW_SPI_PORT_SCK     GPIOA
#define HW_SPI_PIN_SCK      5
#define HW_SPI_PORT_MOSI    GPIOA
#define HW_SPI_PIN_MOSI     7
#define HW_SPI_PORT_MISO    GPIOA
#define HW_SPI_PIN_MISO     6
```

#### d) *app.h*

##### Quelltext

```
void app_spicontrol_init(void);  
void spicontrol_tim_isr(void);
```

##### Aufgabe

Enthält die Prototypen zur Initialisierung der Custom-App, sowie der ISR.

#### e) *applications.mk*

##### Quelltext

```
APPSRC = applications/app.c \  
         applications/app_ppm.c \  
         applications/app_adc.c \  
         applications/app_sten.c \  
         applications/app_uartcomm.c \  
         applications/app_nunchuk.c \  
         applications/app_spicontrol.c
```

```
APPINC = applications
```

##### Aufgabe

Hier wird der Speicherort der Custom-App „app\_spicontrol.c“ angegeben.

#### 5.5.4 Aufgetretene Probleme

##### *Die Firmware ist zu neu*

- In diesem Projekt muss zwingend mit der Firmware 2.18 gearbeitet werden. Neuere Firmware führt zu einer Fehlermeldung im BLDC Tool.

##### *BLDC-Tool: Board taucht nicht in der Liste mit den COM-Ports auf*

- USB-Verbindung aus-/einstecken
- Frequenzumrichter-Platine vom Strom trennen und neu verbinden
- USB-Treiber Probleme
- BLDC-Tool neustarten

##### *ST-Link Utility erkennt STM32 nicht*

- USB-Treiber Probleme
- USB-Verbindung aus-/einstecken

##### *STM32 verbindet sich nicht nach neuem flashen*

- Ursache wird hier vermutlich sein, dass der STM32 direkt nach dem Starten abstürzt. Dieses Problem ist meistens aufgetreten, wenn Änderungen an der ISR oder dem Thread vorgenommen worden. Eine Ursache kann hier schon ein printf-Befehl innerhalb der ISR sein, dieser verzögert die Ausführung ggf. zu stark. Eine weitere Ursache kann die falsche Konfiguration des Delays innerhalb des Threads sein, sollte das Delay zu klein gewählt worden sein, oder es ist evtl. gar nicht vorhanden, wird der STM32 überlastet und stürzt ab. Hier reicht schon ein Delay von 2ms um Abstürze zu verhindern.

##### *Probleme mit der SPI-Kommunikation*

- Ursache können hier zu lange, oder nicht abgeschirmte Leitungen sein, ebenfalls führt die Verwendung von Breadboards zu Störungen auf dem SPI-Signal. Hier sollten die Leitungen idealerweise direkt mit dem Terminal-Board des HIL-Systems verbunden werden.

## 6 Fazit

Wir konnten in dieser Projektarbeit einen umfassenden Eindruck über die Arbeit mit einem HIL-/MIL-System und die anschließende Anbindung eines Frequenzumrichtersystems über eine SPI-Schnittstelle lernen. Unsere Projektarbeit wurde in zwei Teilabschnitte aufgeteilt.

Zunächst haben wir mit der Anbindung eines Digital-Analog-Converters und eines Analog-Digital-Converters an das HIL-/MIL-System zum Einlesen einer Sprungantwort eines Hardware-LRC-Gliedes und dem Vergleich mit einer simulierten Sprungantwort beschäftigt. Hier konnten wir schon einige Erfahrung mit dem Umgang des HIL-/MIL-Systems und der integrierten SPI-Schnittstelle sammeln. So musste zunächst die korrekte Anbindung der IC-Bausteine herausgefunden werden, anschließend mussten die Datenpakete, welche an die Converter gesendet werden richtig konfiguriert sein, um dann die analogen Messdaten des ADCs auf einem Scope in Matlab-Simulink auszugeben und mit dem simulierten LRC-Glied zu vergleichen. Zu Problemen hatte hier die falsche Konfiguration des Datenwortes an den ADC geführt, da das benötigte Senden von 5 Nullen vor dem eigentlichen Datenwort nicht ersichtlich war.

Darauf aufbauend folgte der zweite Teil unseres Projektes, das Anbinden der Frequenzumrichterplatine mittels SPI an die Simulationsumgebung. Hier mussten wir uns zunächst in die umfassende Firmware des Mikrocontrollers hereinarbeiten und die passenden Konfigurationen zum Anbinden der SPI-Schnittstelle vornehmen. Nach erfolgreicher Analyse der Firmware konnte eine SPI-Kommunikation aufgebaut werden. Im Zuge der Firmwareanalyse wurden ebenfalls die bereits in der Firmware implementierten Funktionen zum Messen von Live-Daten und zur direkten Ansteuerung von Motorparametern gefunden. Anschließend folgte der Aufbau der Custom-App, in welchem unsere gesamte Kommunikation und die Ansteuerung des Motors abläuft. Zu Problemen haben hier zum einen der schier endlose Quellcode geführt, welcher über mehrere Tage analysiert werden musste, zu anderen hatten wir Probleme mit der Hardware, wodurch es uns zunächst nicht möglich war, die korrekte Funktion der Ansteuerung des Motors zu testen.

Die bereits in der ersten Projektbesprechung angedachte Regelung des Gesamtsystems konnte aus zeitlichen Gründen nicht realisiert werden, dies hätte den Umfang unseres Projektes überstiegen. Weiterhin haben wir im zweiten Teil des Projektes den Aufbau einer zuverlässigen SPI-Schnittstelle unterschätzt, da wir auf zunächst unerklärliche Störungen in unserer SPI-Kommunikation getroffen sind, diese konnten zwar gelöst werden, jedoch hat dies unseren Zeitplan weiter nach hinten verschoben. Die von uns gesteckten Arbeitspakete und der zeitliche Rahmen konnten jedoch weiterhin von uns eingehalten werden.



## 7 Quellen

- <https://www.speedgoat.com/products/simulink-programmable-fpgas-fpga-i-o-modules-io306>
- <https://www.microchip.com/wwwproducts/en/MCP3204>
- <https://www.microchip.com/wwwproducts/en/MCP4922>
- <http://vedder.se/>
- <http://vedder.se/2015/08/vesc-writing-custom-applications/>
- <http://vedder.se/2015/01/vesc-open-source-esc/>
- <https://github.com/vedderb/bldc-hardware>
- <https://github.com/vedderb/bldc>
- <https://www.mathworks.com/help/simulink/ug/incorporate-c-code-using-a-matlab-function-block.html>
- <https://www.mathworks.com/help/simulink/slref/matlabssystem.html>
- Projektdokumentation: Aufbau eines Versuchsstands für die HIL/MIL-Simulation von Mohamed El Balaoui