

1



OSTBAYERISCHE  
TECHNISCHE HOCHSCHULE  
REGENSBURG

## Projektdokumentation

# MotorXP

*Ansteuerung eines Brushless DC Motors mit einer grafischen Oberfläche und MATLAB  
Simulation*

Vorlesung DT - Wintersemester 2016/17

Christian Brunner, Andreas Kölbl,  
Ricardo Krause, Bernd Krupinski,  
Andreas Lackner, Michael Schleinkofer,  
Franz Welker

# Contents

<b>1</b>	<b>Projektstart</b>	<b>1</b>
1.1	Projektauftrag . . . . .	1
1.2	Projektplan . . . . .	2
1.3	Versionsverwaltung . . . . .	2
1.4	Kommunikation . . . . .	3
1.5	Dokumentenmanagement . . . . .	3
<b>2</b>	<b>Arbeitspaket Kommunikation</b>	<b>4</b>
2.1	Analysephase . . . . .	4
2.1.1	Hardware . . . . .	4
2.1.2	Software . . . . .	5
2.2	Entwurfsphase . . . . .	6
2.2.1	Nachrichtendefinition . . . . .	6
2.2.2	Schnittstellendefinition . . . . .	7
2.3	Implementierung . . . . .	8
2.3.1	PC Bibliothek . . . . .	8
2.3.2	Controller Funktionen . . . . .	10
2.3.3	Parametrierung der DAVE Apps . . . . .	12
2.4	Integration . . . . .	14
2.5	Ausblick . . . . .	15
<b>3</b>	<b>Arbeitspaket Sensorik</b>	<b>16</b>
3.1	Überblick . . . . .	16
3.2	Entwicklungsumgebung . . . . .	16
3.2.1	DAVE 4 . . . . .	17
3.2.2	POSIF . . . . .	19
3.3	Sensor Interface . . . . .	20
3.3.1	Modulstruktur . . . . .	20
3.3.2	Anwendung . . . . .	20
3.4	Hall Sensoren . . . . .	21
3.4.1	Einlesen der Hall-Sensoren . . . . .	22
3.4.2	Ermittlung der Hall-Pattern . . . . .	22
3.5	Inkrementalgeber . . . . .	23
3.5.1	Implementierung . . . . .	23
3.6	Temperatursensor . . . . .	24
3.6.1	Einlesen des Temperaturwerts . . . . .	24
3.7	Ausblick . . . . .	26

<b>4</b>	<b>Arbeitspaket Regelung</b>	<b>27</b>
4.1	Analysephase . . . . .	27
4.1.1	Hardware . . . . .	27
4.1.2	Software . . . . .	27
4.2	Entwurfsphase . . . . .	27
4.3	Implementierung . . . . .	28
4.4	Integration . . . . .	28
4.5	Ausblick . . . . .	28
<b>5</b>	<b>Arbeitspaket GUI</b>	<b>29</b>
5.1	Analysephase . . . . .	29
5.2	Entwurfsphase . . . . .	29
5.3	Implementierung . . . . .	31
5.4	Ausblick . . . . .	33
<b>6</b>	<b>Arbeitspaket Simulation</b>	<b>36</b>
6.1	Analysephase I - Verwendung existierender Bausteine . . . . .	36
6.2	Entwurfsphase und Implementierung I - Anpass-Arbeiten . . . . .	38
6.3	Analysephase II - Alternativ-Modell . . . . .	38
6.4	Entwurfsphase II - eigene Motor-Modellierung . . . . .	38
6.5	Implementierung II - neues Motormodell . . . . .	40
6.6	Bewertung des neuen Modells . . . . .	41
6.7	Bewertung der beiden Modelle . . . . .	42
6.8	Ausblick . . . . .	43
<b>A</b>	<b>Codeausschnitt aus der C#-Bibliothek mit ComPort-Settings</b>	<b>47</b>
<b>B</b>	<b>Codeausschnitt aus der C#-Bibliothek zum Senden der Bytes</b>	<b>48</b>
<b>C</b>	<b>Codeausschnitt aus der C#-Bibliothek mit dem SoF-Automaten</b>	<b>49</b>

### **Abstract**

Diese Dokumentation entstand im Rahmen der Vorlesung "Datenverarbeitung in der Technik" im Wintersemester 2016/17. Zu Beginn erfolgt die Projektdefinition mit Aufteilung der einzelnen Arbeitspakete. Anschließend werden die einzelnen Arbeitspakete näher in den einzelnen Projektschritten erklärt.

# Chapter 1

## Projektstart

Dieses Kapitel wurde von Herr Krause erstellt. Es beschreibt die Startphase des Projektes und zählt auf welche Tätigkeiten und Entscheidungen vor beginn des eigentlichen Projektes durchgeführt wurden. Es werden die Punkte Projektauftrag, Projektplan, Versionsverwaltung, Projektkommunikation und Dokumentenmanagement beschrieben.

### 1.1 Projektauftrag

Das Grundlegende Dokument für jedes Projekt ist der Projektauftrag, dieser Auftrag beinhaltet alle relevanten Parameter und Eckdaten des Projektes.

**Projekttitel:** Der Arbeitstitel des Projektes.

**Projektnummer:** Eine in der Regel einzigartige Nummer zur Identifikation eines Projektes.

**Projektart:** Unterscheidung von verschiedenen möglichen Projekttypen.

**Projektleiter:** Bindeglied zwischen Kunden und Entwicklern mit überwachender und steuernder Tätigkeit.

**Projektauftraggeber:** Der Auftraggeber des Projektes.

**Projektkunden:** Können von Auftraggeber abweichen und auch mehrere sein.

**Projektdauer:** Legt den zeitlichen Rahmen des Projektes fest.

**Ausgangssituation/Problembeschreibung:** Dies beschreibt den aktuellen Zustand und das Problem, welches mit dem Projekt gelöst werden soll.

**Projektgesamtziel:** Zusammengefasste Beschreibung des Projektes mit dem zu erreichenden Ziel.

**Projektteilziele und -ergebnisse:** Detaillierte Auflistung der einzelnen Teilziele, mit messbaren zu erreichenden Ergebnissen.

**Nicht-Ziele / Nicht-Inhalte:** Abgrenzung des Projektes.

**Meilensteine:** Eckpunkte des Projektes mit Ziel Datum, welche erreicht werden müssen um das Projekt erfolgreich zum Abschluss bringen zu können.

**Randbedingungen und -projektkontext:** Nebenbedingungen welche meist nicht direkt von Projektteam beeinflussbar, aber Notwendig zum Erreichen des Projektzieles sind.

**Projektklassifizierung:** Parameter, welche das Projekt als Ziffer beschreiben und somit eine Vergleichbarkeit zwischen anderen Projekten schaffen.

**Projektorganisation:** Beinhaltet das Projektteam, weitere beteiligte Personen.

**Projektressourcen:** Alle für das Projekt zur Verfügung stehenden Ressourcen, beinhalten Personal und Material.

**Projektbudget:** Das zur Verfügung stehende Budget.

**Wirtschaftlicher oder sonstiger Nutzen:** Dieser Parameter ist meist wichtig für die Entscheider eines Projektes und soll den Gewinn durch das Projekt aufzeigen.

**Projektrisiken und -unsicherheiten:** Zeigt alle Risiken auf, welche zu einem Scheitern des Projekts führen könnten.

**Projektentscheidung:** Das Projekt muss vor Beginn von einer Entscheidungsbefugten Person freigegeben werden.

**Sonstige relevante Informationen:** Weitere wichtige Information.

**Anlagen:** Weitere Dokumente, welche das Projekt betreffen.

Der Projektantrag wurde von Herr Krause aufgesetzt, die Inhalte wurden gemeinsam erarbeitet und mit den Projektbeteiligten Personen abgestimmt. Der ausgearbeitete Antrag befindet sich im Anhang

## 1.2 Projektplan

Nach dem der Projektantrag erstellt wurde und die Ziele des Projektes bekannt waren, wurde ein detaillierter Projektplan erstellt. Dieser Plan wurde mit dem Open Source Tool *OpenProj*<sup>1</sup> erstellt und befindet sich ebenfalls im Anhang.

Der Projektplan beinhaltet alle erforderlichen Tätigkeiten und ermöglichte eine Abschätzung, ob das Projekt mit den gegebenen Ressourcen in der gegebenen Zeit durchführbar ist.

## 1.3 Versionsverwaltung

Als weiterer fundamentaler Pfeiler dieses Projektes wurde ein Versionsverwaltungssystem eingerichtet, welches dem Team ermöglichte, ohne größere Probleme gemeinsam an dem Projekt zu arbeiten, ohne Gefahr zu laufen, die erstellten Artefakte<sup>2</sup> zu beschädigen.

Die Wahl fiel auf das kostenlose und Open Source laufende Tool *GIT*<sup>3</sup>, welches als verteiltes Versionsverwaltungssystem für die Software Komponenten und Dokumentation eingesetzt wird.

Das System GIT wurde gewählt, da es schon mehreren Mitgliedern des Entwicklungsteams bekannt war und eine Vertiefung mit dem Umgang dieses Tools gewünscht wurde.

Durch den zusätzlichen Einsatz von *Github*<sup>4</sup>, welches ein Web-Interface für das eingesetzte GIT bietet, wurde es ebenfalls ermöglicht, untereinander die Software Artefakte zu testen und direkt über Issues<sup>5</sup>, Probleme und Änderungswünsche zu berichten. // Die grobe Erstellung des Projektes auf Github übernahm der Herr Krause, die interne Struktur der einzelnen Arbeitspakete wurde von dem Herr Kölbl erledigt.

---

<sup>1</sup>Link zum Projekt <http://www.serena.com/index.php/en/products/pod-update/>

<sup>2</sup>Arbeitsergebnis in einem Projekt

<sup>3</sup>Link zum Projekt <https://git-scm.com/>

<sup>4</sup>Link zum Projekt <https://github.com/>

<sup>5</sup>engl. für Problem oder Angelegenheit

## 1.4 Kommunikation

Für die Kommunikation der Projektgruppe wurde das ebenfalls Open Source laufende Tool *Slack*<sup>6</sup> eingesetzt. Dieses Tool ermöglichte es gemeinsam und Themen orientiert zu kommunizieren und erleichterte durch Schnittstellen zur Versions- und Dokumentenverwaltung die erforderlichen Workflows. Es wurde für jedes Arbeitspaket ein eigener Kommunikationsbereich eingerichtet und Mitglieder welche in einen bestimmten Bereich involviert oder interessiert waren, konnten diesen Bereichen beitreten und miteinander diskutieren.

Das Kommunikationstool wurde vom Herrn Krause eingerichtet.

## 1.5 Dokumentenmanagement

Im Team wurde entschieden, dass der Großteil der Dokumente für das Projekt nicht in die Versionsverwaltung gehört, deshalb wurde für die Verwaltung dieser Dokumente der Online Speicherdienst *Dropbox*<sup>7</sup> gewählt. Dieses System wird bereits von den Studenten eingesetzt und bietet Schnittstellen zu aktuellen Textbearbeitungsprogrammen wie Microsoft Word und ebenfalls eine Integration zu Github und Slack ist möglich.

Es wurde eine grobe Ordnerstruktur<sup>1.1</sup> für die Dokumente festgelegt, welche sich an den Phasen des Projektes orientiert. Erstellte oder beschaffte Dokumente wurde in diese übersichtliche Struktur eingeordnet.

Für Dokumente, welche gemeinsam bearbeitet werden mussten Templates erstellt, welche jeder sep-








Name	Änderungsdatum	Typ	Größe
 0100_AktuelleVersion	04.01.2017 18:00	Dateiordner	
 0200_Projektplan	26.11.2016 22:01	Dateiordner	
 0300_Anforderungen	04.01.2017 18:02	Dateiordner	
 0400_Projekt_Vorbereitung	04.01.2017 18:01	Dateiordner	
 0500_Material_Sammlung	04.01.2017 09:48	Dateiordner	
 0600_Präsentationen	04.01.2017 18:01	Dateiordner	
 0700_Dokumentation	04.01.2017 09:48	Dateiordner	

Bild 1.1: Ordner Struktur Projekt

arat ausfüllen konnte und welche nach Bearbeitung zusammen geführt wurden, dies ermöglichte einen einheitlichen Stil der Dokumente.

<sup>6</sup>Link zum Projekt <https://slack.com/>

<sup>7</sup>Link zum Anbieter <https://www.dropbox.com/home>



## Chapter 2

# Arbeitspaket Kommunikation

Für die Analyse, den Entwurf, die Implementierung dieses Arbeitspaketes übernahm Herr Schleinkofer die Verantwortung. Während der Integration des Arbeitspaketes war Herr Schleinkofer unterstützend eingebunden.

Da die Projektspezifikation einen Datenaustausch zwischen  $\mu$ -Controller und PC enthält, muss diese auch im Projekt realisiert werden. Dabei kann sich auch am ISO/OSI-Schichtenmodell für Netzwerkkommunikation orientiert werden.

## 2.1 Analysephase

### 2.1.1 Hardware

Beginnend bei ISO/OSI-Layer 1, muss zunächst die physikalische Kommunikation in diesem Projekt betrachtet werden. Dabei stellt sich die Frage, welche Möglichkeiten der Controller zur Datenübertragung an ein externes Gerät bietet. In dessen Handbuch ist aufgeführt, dass dort ein Universal Serial Interface Channel (USIC) Baustein zur Verfügung steht. Dies würde eine serielle Kommunikationsverbindung mit dem Universal Asynchronous Receiver Transmitter (UART) ermöglichen. Weitere Möglichkeiten sind auf dem Evaluation Board kaum gegeben. Dieses ist zwar für Ethernet vorbereitet, jedoch müssen dafür ein zusätzlicher Chip und eine RJ45-Buchse nachgerüstet werden. Eine selbst definierte und implementierte parallele Schnittstelle mit den GPIO-Ports wäre theoretisch auch möglich, jedoch in der Umsetzung zu komplex und zeitintensiv.

Die gemachten Überlegungen haben zur Folge, dass die Kommunikation zwischen Testplatz und PC über die serielle Schnittstelle abgewickelt wird. Nun muss entschieden werden, welches Gerät am Computer verwendet werden soll. Möglich wäre eine Kommunikation über ein serielles Kabel, welches an eine EIA-232 Schnittstelle des PC's angeschlossen wird. Dies ist jedoch mit einigen Problemen behaftet. Zunächst verfügen moderne Desktop Computer nur noch in wenigen Fällen über eine dedizierte EIA-232 Schnittstelle. Dies kann jedoch durch Verwendung eines USB-zu-Seriell-Adapters umgangen werden. Am  $\mu$ -Controller erfolgt dann der Anschluss an die GPIO-Pins. Allerdings muss dafür ein Adapter angefertigt werden, welcher auf der einen Seite einen D-Sub 9 Stecker und auf der anderen Seite mindestens die Pins 2 (Data Transceive), 3 (Data Receive) und 5 (Ground) als Stifte, passend zur Buchsenleiste des Evaluation Boards, weiterführt. Allerdings werden die Daten an der EIA-232 Schnittstelle mit bis zu 9 Volt übertragen. Dies stellt ein weiteres Problem dar, da an den Pins des Controllers nur 3,3 oder 5 Volt ausgegeben werden können. Auch kann ein Eingangssignal mit 9 Volt zu Schäden am  $\mu$ -Controller führen. Um dieses Problem zu umgehen, wäre es möglich, einen USB-zu-Seriell-Adapter mit integriertem Wandler-Chip zu verwenden.

Eine weitere Möglichkeit zur seriellen Kommunikation zeigt ein Beispielprojekt von Infineon für das Evaluation Board auf. In diesem erfolgt die Kommunikation zwischen PC und Controller über das USB-Kabel, welches am Debugging-Chip des Boards angeschlossen wird. Auch in diesem Projekt erfolgt der Datenaustausch mit Nutzung des USIC-Bausteins. Allerdings werden hier die Sende- und Empfangsleitungen an den XMC4200 Debug IC (im folgenden auch Debug-Chip genannt) auf der Platine weitergeleitet. Für den PC ist darüber hinaus ein Windows-Treiber verfügbar, welcher einen "virtuellen Com-Port" zur Kommunikation mit dem Evaluation Board einrichtet und über den nun Daten im Rahmen des Beispielprojektes ausgetauscht werden können. Auf die Software des PC's hat diese Vorgehensweise keinen Einfluss, da ein virtueller Com-Port in der gleichen Weise verwendet wird, wie ein Hardware-Gerät. Ein weiterer Vorteil dieser Lösung ist, dass eine zusätzliche Stromversorgung des  $\mu$ -Controllers entfällt, da dies über das USB-Kabel der Kommunikation erfolgt.

Die Aufgaben der zweiten Schicht des ISO/OSI-Modells umfassen unter Anderem auch die Sicherung der Datenintegrität während der Übertragung. Dies geschieht im Fall des Ethernet-Protokolls mittels eines Cyclic Redundancy Check Verfahrens. Und auch andere Kommunikationsprotokolle nutzen diese Art der Integritätsprüfung um etwaigen Störungen während der Übertragung entgegenzuwirken.

Weitere Funktionen aus höheren Schichten des ISO/OSI-Modells scheinen in der gegebenen Zeit nicht umsetzbar oder werden schlichtweg nicht benötigt. Eine Adressierung der Kommunikationspartner ist in diesem Fall aufgrund der auf genau zwei beschränkten Anzahl an Teilnehmern genauso wenig notwendig wie eine Sitzungsverwaltung ähnlich derer in OSI-Schicht 5. Eine Flusskontrolle oder Empfangsbestätigung ist im Falle zu übertragender Sensordaten auch nicht sinnvoll. Diese Sensordaten sollen nach den Anforderungen regelmäßig vom  $\mu$ -Controller an den PC gesendet werden. Bei höheren Drehgeschwindigkeiten des Motors würden diese nur unnötig Rechenzeit auf der Senderseite verbrauchen. Bei der Übertragung von Regelungsparametern allerdings könnte eine Empfangsbestätigung durch den  $\mu$ -Controller durchaus nützlich sein. Allerdings ist hierzu wie bereits erwähnt der Zeitrahmen zu eng gesteckt.

### 2.1.2 Software

Des weiteren muss noch Analysiert werden, wie die Sensor- und Regelungsdaten für die Kommunikation aufbereitet werden. Hierzu kommen verschiedene Serialisierungsverfahren in Betracht. Zunächst besteht die Möglichkeit die Daten als String zu formatieren und die einzelnen Zeichen als char-Daten seriell zu übertragen. Dies ist eine einfache Lösung, jedoch für zukünftige Änderungen nur schlecht erweiterbar. Es muss dazu für jeden Sensor die komplette Decodierungs-Routine des Strings auf beiden Kommunikationspartnern angepasst werden. Sollten nun in Zukunft beispielsweise mehr Experimentierplätze mit unterschiedlichen Sensoren existieren, muss darauf geachtet werden, dass zu jedem auch ein PC verwendet wird, dessen GUI auch den String mit den vorliegenden Sensorwerten auswerten kann.

Alternativ dazu kann ein Serialisierungsformat mit dem Namen "Protocol Buffers" (kurz: Protobuf) verwendet werden. Dieses wurde durch die Firma Google entwickelt und ist für mehrere Programmiersprachen verfügbar. Dies ist wichtig, da die Software auf dem  $\mu$ -Controller in C, der Teil für den PC aber in C# geschrieben wird. Der Vorteil dieser Lösung besteht darin, dass etwaige neu hinzukommende Sensoren einfach in die Protobuf-Nachrichten aufgenommen werden können. Kann eine Gegenstelle diesen Teil einer Nachricht nicht verarbeiten, so wird dieser einfach ignoriert.

**Code 2.1:** Beispieldefinition einer Protocol Buffers Nachricht

```
1 message Person {
```

```

2   required string name = 1;
3   required int32 id = 2;
4   optional string email = 3;
5 }

```

Dies liegt in der Art, wie eine Nachricht in Protobuf aufgebaut ist. Zunächst wird diese in einer Beschreibungssprache definiert. Im Code 2.1 ist ersichtlich, dass eine "Message" aus mehreren Feldern und deren zugehörigen Identifiern ("= 1" für das Feld "name") besteht. Wird nun eine Nachricht um neue Felder erweitert, so kann ein Programm, welches jedoch noch die alte Beschreibung nutzt, eine neuartige Nachricht wieder deserialisieren. Die neuen Felder jedoch werden hierbei aufgrund des unbekannten Identifiers ignoriert. Protobuf verfügt darüber hinaus noch über einen Codegenerator. Das bedeutet: Der Entwickler definiert in der gezeigten Beschreibungssprache eine Nachricht. Anhand dieser Beschreibung wird nun durch den Generator zum Beispiel C#-Code zur De-/Serialisierung erstellt. Zusätzlich wird dabei eine Datenstruktur definiert, die vom Entwickler im Programmcode mit den zu übertragenden Daten ausgefüllt werden und einfach in einen Bytestream verwandelt und wieder zurückverwandelt werden kann.

Zwar gibt es keine "offizielle" Implementierung von Protobuf für C, jedoch existiert ein Projekt mit dem Namen "nanopb" welches die grundlegenden Funktionen von Protocol Buffers in C speziell für Embedded Systems umsetzt.

## 2.2 Entwurfsphase

Der Ablauf der Kommunikation sollte relativ einfach gehalten werden, um in der vorgegebenen Zeit umsetzbar zu sein. Daher senden die Kommunikationspartner nur einfache Protobuf-Nachrichten. Der PC sendet Regelungsparameter und Ziel an den Controller. Dieser wiederum sendet regelmäßig die Daten der angeschlossenen Sensoren an die GUI auf dem PC. Um die Kommunikation für die angrenzenden Arbeitspakete so einfach wie möglich zu gestalten, soll die Schnittstelle nur aus wenigen Funktionen bzw. Methoden bestehen.

### 2.2.1 Nachrichtendefinition

Im System gibt es zwei Arten von Nachrichten. Die erste Art enthält die gesammelten Daten aller Sensoren auf dem Controller und wird von diesem an den PC verschickt. Die zweite Art enthält die Regelungsparameter und die Größe, welche geregelt werden soll. Diese Parameter können an der GUI eingestellt und an den  $\mu$ -Controller gesendet werden. Der grundlegende Aufbau beider Nachrichten auf Byte-Ebene ist gleich und wird in Bild 2.1 dargestellt. Den Beginn der Nachricht markiert ein Byte, welches die Anzahl aller Bytes der Nachricht (nachfolgend Frame genannt) enthält. Anschließend folgen die bereits von Protobuf serialisierten Daten als Reihe von Bytes. Den Abschluss des Frames bilden zwei Bytes, welche die CRC Prüfsumme enthalten. Bei der Prüfsumme handelt es sich um die 16-Bit große CRC-CCITT, welche unter anderem auch beim seriellen Datenübertragungsprotokoll HDLC verwendet wird. (hdl07, Absatz 1)



**Bild 2.1:** Schematischer Aufbau eines Frames

### Sensordaten Nachricht

Bei der Absprache mit den für die GUI verantwortlichen Entwicklern, kam folgende Anforderung hinzu: "Die Nachrichten sollen mit einem Zeitstempel oder einer fortlaufenden Nummer ausgestattet sein, um diese später unter Umständen nochmals sortieren zu können." Um auch für zukünftige Erweiterungen von Sensoren gerüstet zu sein, soll die Anzahl der Sensordaten in der Nachricht variabel gestaltet sein. Zudem wird jedem Sensorwert die ID des Sensors, welcher den Wert produzierte, zugeordnet. Um das alles zu erfüllen gilt für die Protobuf Nachricht folgende Beschreibung:

**Code 2.2:** Beschreibung der Sensordaten Nachricht

```

1 //defining an entry of the data table
2 message DataEntry{
3     uint32 SensorId = 1;
4     double Data = 2;
5 }
6 //defining the real message
7 message SensorMsg{
8     //Upcounting Nr
9     uint64 SequenceNr = 1;
10    //all Data
11    repeated DataEntry DataTable = 2;
12 }
```

### Regelungsparameter Nachricht

Das Team, welches die Regelung umgesetzt hat, stellt eine API bereit, welche neben den Parametern der P-, I- und D-Glieder eines Reglers auch die Regelgröße (z.B. Geschwindigkeit oder Drehmoment) und den Zielwert entgegennimmt. Dazu müssen diese Werte auch in der Protobuf Nachricht wie folgt berücksichtigt werden:

**Code 2.3:** Beschreibung der Parameter Nachricht

```

1 //defining the parameter message
2 message RegParams{
3     uint32 target = 1;
4     float paraP = 2;
5     float paraI = 3;
6     float paraD = 4;
7     float tgtVal = 5;
8 }
```

Im Codebeispiel 2.3 wird die Regelgröße mittels dem "target" Feld übermittelt. Dabei muss jedoch beachtet werden, dass bei beiden Kommunikationspartnern die Werte der gleichen Größe zugeordnet werden.

## 2.2.2 Schnittstellendefinition

### PC Bibliothek

Für den Teil, welcher die Kommunikation auf Seiten der GUI übernimmt, soll eine Bibliothek auf C#-Basis implementiert werden. Um nun Regelungsparameter an den Controller senden zu können, muss ein Datenobjekt beim Aufruf der Sende-Methode übergeben werden. Für den Empfang von Sensordaten ist es von Vorteil, das Hollywood-Prinzip anzuwenden. So soll die Bibliothek nicht regelmäßig

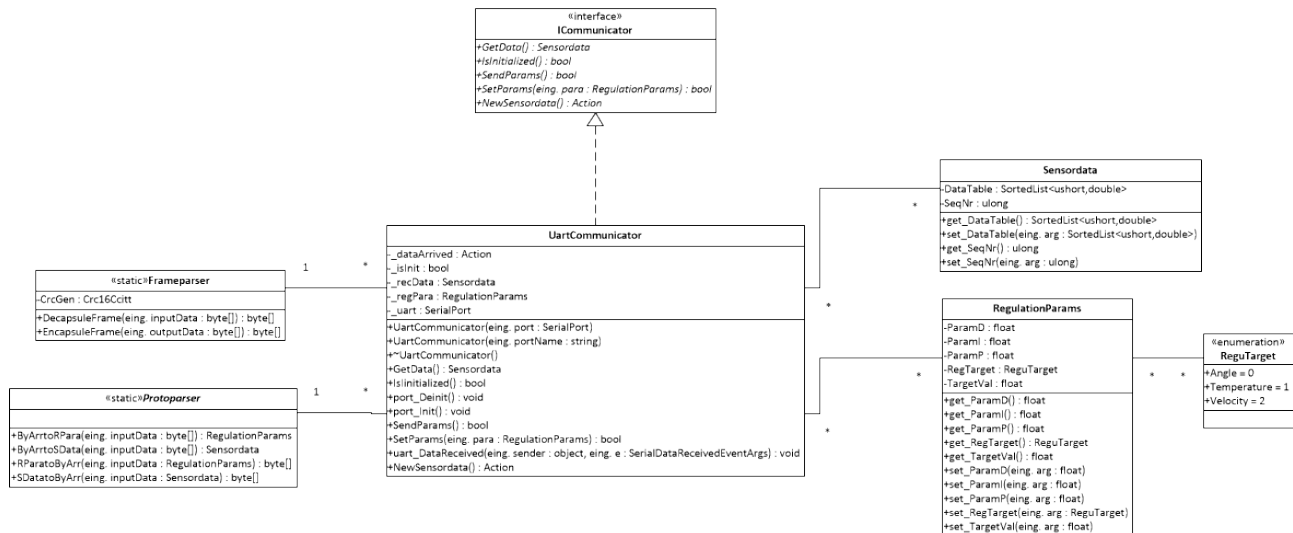


Bild 2.2: wesentliche Klassen der implementierten Bibliothek für die GUI

nach neuen Daten abgefragt werden. Vielmehr löst deren API ein Event aus, welches die Verfügbarkeit neuer Werte anzeigt. Da sich bereits darauf festgelegt wurde, einen virtuellen Com-Port zur Kommunikation zu verwenden, soll dieser nun auch bei der Initialisierung der API als String oder gleich als ComPort-Objekt mit angegeben werden.

## Controller Funktionen

Ähnlich zur PC Bibliothek sollen die Funktionen auf dem  $\mu$ -Controller aufgebaut werden. Es soll zunächst eine Funktion zur Initialisierung der Kommunikation bereitgestellt werden. Darüber hinaus soll es eine Funktion zum Setzen und eine weitere zum Versenden der gesammelten Sensordaten geben. Während der Analysephase wurde ersichtlich, dass auf dem Controller ein DMA-Baustein (Direct Memory Access) genutzt werden kann, um die Nachrichten-Bytes vom und zum UART-Modul zu übertragen. Dies erleichtert die Handhabung der Kommunikationsfunktionen dahingehend, dass nun beim erfolgreichen Empfang von Regelungsparametern ein globales Flag gesetzt werden kann, welches dies gegenüber der Regelungsschleife anzeigt. Eine weitere Aktion bezüglich des Datenempfangs von Seiten der Regelung ist somit nicht erforderlich. Die so empfangenen Regelungsparameter sollen nun nach Erhalt in global sichtbare Variablen abgelegt werden. Dies geschieht auf Wunsch des für die Regelung verantwortlichen Teams.

## 2.3 Implementierung

Da beide Endpunkte gänzlich unterschiedliche Eigenschaften und Ressourcen besitzen, gibt es zwei separate Softwareprojekte. Die Implementierung der Bibliothek erfolgt wie bereits erwähnt in C# und unter Visual Studio 2013. Für die Implementierung des Kommunikationsmoduls auf dem Controller wird die Programmiersprache C unter DAVE 4 genutzt.

### 2.3.1 PC Bibliothek

Wie aus Bild 2.2 ersichtlich ist, dient das Interface "ICommunicator" als API zur GUI hin und die Klasse UartCommunicator ist die Hauptklasse. Diese versammelt alle Funktionalität zum Senden und Empfangen von Nachrichten in sich. Bei der Erstellung eines Objektes dieser Klasse wird auch sofort die Initialisierungsmethode aufgerufen. In dieser wird das SerialPort-Objekt, welches zuvor dem Konstruktor angezeigt wurde, parametrisiert und geöffnet. Die gesetzten Parameter sind in den Zeilen 5

bis 12 im Anhang A ersichtlich. Da zur Übertragung die USB-Schnittstelle mit nur zwei Signalleitungen genutzt wird, ist es nicht möglich für die Kommunikation einen Handshake auszuwählen bzw eine Hardware-gesteuerte Flusskontrolle anzugeben. Dies wäre nur bei einer Verwendung der EIA-232 Schnittstelle möglich, da hier zusätzliche Leitungen für diese Zwecke existieren. Aus diesem Grund sind die Parameter "Handshake", "DtrEnable" und "RtsEnable" auf None bzw. false gesetzt. Alle weiteren Parameter wurden so gewählt, dass diese bereits bekannten Einstellungen für andere serielle Kommunikationen entsprechen.

Die Eigenschaft "ReceivedBytesThreshold" wurde auf 10 gesetzt. Dies bedeutet, dass nach 10 empfangenen Bytes die Methode aufgerufen wird, welche auf dem "DataReceived"-Ereignis des SerialPort-Objektes registriert wurde. Dies ist die private Methode des UartCommunicators. In dieser ist die Logik für den Empfang von Nachrichten implementiert.

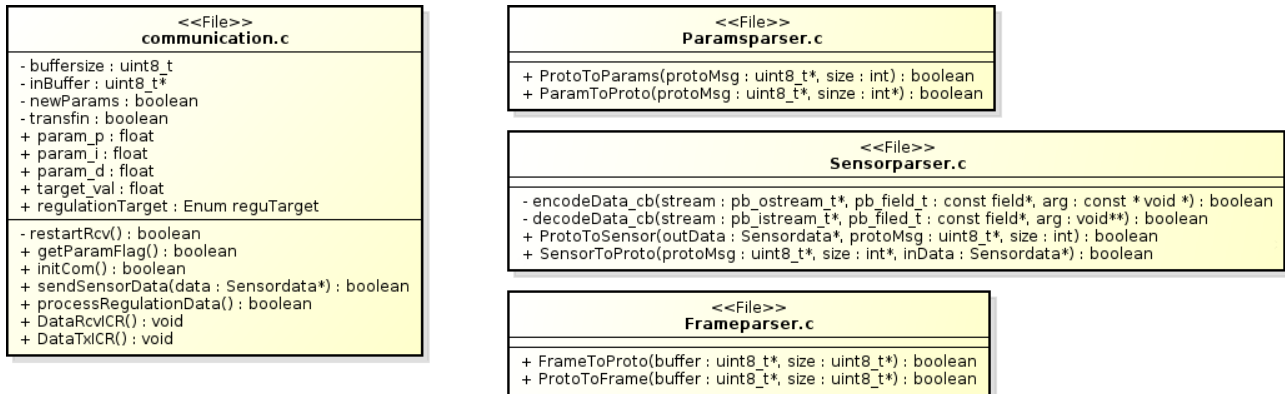
Bei einem Empfang von Daten auf der seriellen Schnittstelle, wird zunächst das erste Byte im Eingangspuffer gelesen. Da dies die Framelänge darstellt (vgl. Bild 2.1), kann nun daraus geschlossen werden, wie lang die empfangene Nachricht tatsächlich ist. Anschließend wird die nun bekannte Anzahl an Bytes in den Bearbeitungspuffer gelesen. Auf diesem Bearbeitungspuffer wird nun zunächst der Frame mit der statischen Methode "DecapsuleFrame" entpackt und folgend die Protobuf-Nachricht in ein Sensordata-Objekt umgewandelt. Am Ende der Empfangsprozedur wird das Event NewSensordata ausgelöst, welche anzeigt, dass neue Sensordaten vom UartCommunicator-Objekt abgeholt werden können. Dies geschieht mit der GetData-Methode.

Um nun Regelungsparameter versenden zu können, muss die GUI dem UartCommunicator-Objekt zunächst mit der SetParams-Methode ein Datenobjekt übergeben. Dieses Datenobjekt der Klasse RegulationParams kann nun mit der Methode "SendParams" an den Controller gesendet werden. Hier ist der Ablauf in umgekehrter Reihenfolge zu durchlaufen. Das Datenobjekt wird nun von einer statischen Methode in eine Protobuf-Nachricht serialisiert und in einen Bearbeitungspuffer abgelegt. Dieser Bearbeitungspuffer wird im Anschluss mit einem Frame-Header und einem Frame-Tail ausgestattet. Abschließend wird der Bearbeitungspuffer versendet. In Zeile 9 im Anhang B ist zu erkennen, dass nach jedem gesendeten Byte des Bearbeitungspuffers 30 Millisekunden gewartet wird. Dies ist notwendig, da ein kontinuierliches Senden der Datenbytes den DMA-Baustein auf dem Mikrocontroller überlastet und dort nur noch unverwertbare Daten vorhanden wären.

Für die Umwandlung der Daten in Protobuf-Nachrichten und umgekehrt, existiert eine separate Protoparser-Klasse. Diese stellt wie im Klassendiagramm ersichtlich statische Methoden zur Verfügung. Um aus einem Bytearray mit Protobuf-Daten ein Sensordaten-Objekt zu erzeugen, nutzt die Methode "ByArrtoSData" zunächst ein SensorMsg-Objekt aus dem durch Protobuf generierten Code. So wird ein Protobuf-Objekt erzeugt, aus welchem im Anschluss die Daten für das Sensordaten-Objekt extrahiert werden.

In die "Gegenrichtung" muss in umgekehrter Reihenfolge vorgegangen werden. Zunächst wird ein Objekt aus dem von Protobuf generierten Code mit den Daten eines Sensordaten-Objektes gefüllt. Dieses Protobuf-Objekt wird nun mit der Methode "ToByteArray" in ein Array aus 8-Bit-Zahlen umgewandelt, welches wieder zurückgegeben wird. Die gleiche Vorgehensweise gilt auch für die Methoden "RParatoByArr" bzw. "ByArrtoRPara" welche die Regelungsparameter in eine Protobuf-Nachricht um- bzw. zurückwandeln.

Für die Handhabung der Nachrichten-Frames stehen in der Klasse "Frameparser" zwei statische Methoden zur Verfügung. Die Methode "DecapsuleFrame" nimmt zunächst das übergebene Byte-Array mit dem Frame und berechnet ab Element 2 die CRC-Summe. Da am Ende des Frames bereits

Bild 2.3: Implementierte Funktionen des Kommunikationsmoduls auf dem  $\mu$ -Controller

ein 16 Bit CRC-Wert steht, entspricht das Ergebnis der aktuellen CRC-Erzeugung bei einem korrekten Frame 0. Ist dies der Fall, so werden die Bytes der Payload in einem neuen Byte-Array zurückgegeben.

Um nun ein Byte-Array mit einer Protobuf-Nachricht mit einem Frame auszustatten, existiert die statische Methode "EncapsuleFrame". Über das übergebene Array wird in dieser Methode die 16 Bit lange CRC-Summe berechnet. In ein neues Byte-Array wird nun an erster Stelle die Länge des alten Arrays + 3 als neue Framelänge abgelegt. Im Anschluss an das erste Element folgt nun das bereits übergebene Array. Abschließend werden beide Bytes des CRC-Wertes in das neue Array eingefügt, bevor dieses wieder zurückgegeben wird.

Für die Berechnung der CRC-CCITT in C# wurde eine Klasse verwendet, welche unter sanity-free.org (crc06) zur Verfügung steht. Als Initialwert bei der Instanziierung wurde "NonZero1" verwendet. Es ist darauf zu achten, dass dieser Wert mit dem Initialwert auf dem  $\mu$ -Controller übereinstimmen muss.

### 2.3.2 Controller Funktionen

Die Implementierung des Kommunikationsmoduls auf dem  $\mu$ -Controller erfolgte unter Verwendung des Programmes DAVE 4 von Infineon. Bild 2.3 Damit ist es relativ einfach, die auf dem Chip vorhandenen Ressourcen zu parametrieren und zu nutzen. Im Rahmen der Kommunikation werden hauptsächlich zwei sogenannte Apps verwendet. Eine davon dient zur Parametrierung des UART-Bausteins, die andere zur Parametrierung der CRC-Erstellung.

Als Hauptobjekt der API dient die Datei communication.c. Hier wird zunächst die Funktion "initCom" zur Verfügung gestellt. Damit werden die globalen Variablen zur Übergabe der Regelungsparameter auf einen definierten Anfangszustand gebracht und der Empfang von Nachrichten auf dem UART-Baustein angestoßen. Dazu wird die Funktion "UART\_Receive" genutzt, welche vom Programm DAVE bei Verwendung der UART-App automatisch generiert werden kann. Allerdings wird hier nur ein Byte gelesen. Denn nach der Definition des Frameaufbaus, resultiert daraus die Framelänge. Ist dieses Byte erfolgreich gelesen worden, so wird die Callback-Funktion "DataRcvICR" durch einen Interrupt aufgerufen.

In dieser Callback-Funktion wird nun ein Bearbeitungspuffer passend für den Empfangenen Frame erstellt und in diesem die noch fehlenden Bytes eines nach dem anderen abgelegt. Wurde nun der gesamte Frame erfolgreich empfangen, so wird ein Flag gesetzt. Falls nicht, wird die Kommunikation neu initialisiert.

Darüber hinaus existiert eine Funktion, welche bei ihrem Aufruf das Flag aus der Callback-Funktion zurückgibt und so anzeigt, ob eine neue Nachricht mit Regelungsparametern vorliegt. Wird hier true zurückgegeben, kann die Regelung die Funktion "processRegulationData" aufrufen. Damit wird die Verarbeitung der empfangenen Datenbytes und das erneute Empfangen von Nachrichten angestoßen.

Um nun die Daten der Sensoren zu versenden, kann die Funktion "sendSensorData" genutzt werden. Dieser muss ein Pointer auf eine Struktur des Typs "Sensordata" mit übergeben werden. Diese Funktion nun wandelt die übergebenen Daten zunächst in eine Protobuf-Nachricht und dann in einen Frame um. Dann wird dieser Frame mit der von DAVE generierten Funktion "UART\_Transmit" übertragen. Da in der Entwurfsphase beschlossen wurde, dass die Übertragung der Daten vom und zum UART-Baustein mittels DMA geschieht, gibt es für den Interrupt eines erfolgreichen Versendens der Daten eine Callback-Funktion. In dieser Funktion wird ein statisches Flag gesetzt, welches verhindert, dass der DMA-Baustein erneut Daten senden muss, obwohl die "alte" Nachricht noch nicht komplett versendet worden ist.

Für die Umwandlung der Sensordaten in eine Protobuf-Message, steht die Funktion "SensorToProto" in der Datei Sensorparser.c zur Verfügung. In dieser wird zunächst eine von nanopb generierte Struktur soweit wie möglich ausgefüllt. Um jedoch das wiederholte Feld für die sogenannte "DataTable" auszufüllen, muss eine Callback-Funktion definiert werden. Dies ist die in der gleichen Datei befindliche "encodeData\_cb". Mit dem Aufruf der ebenfalls durch nanopb generierte Funktion "pb\_encode" wird nun die Message-Struktur in ein Byte-Array umgewandelt.

In der Callback-Funktion muss für jeden Eintrag in der "DataTable" aus der Protobuf-Message die Funktion "pb\_encode\_submessage" aufgerufen werden. Funktionen zum decodieren der Sensordaten-Nachricht wurden in der gleichen Datei implementiert, werden hier jedoch nicht weiter ausgeführt, da diese im Rahmen des Projektes nicht genutzt werden.

Um nun empfangene Nachrichten mit Regelungsparametern dekodieren zu können, wird die in der Datei Paramparser.c implementierte Funktion "ProtoToParams" genutzt. Aufgrund des einfacheren Aufbaus der Parameter-Message, müssen hier nach dem Aufruf der generierten Funktion "pb\_decode" nur die Parameterwerte aus einer temporären Struktur in die zuvor genannten, globalen Variablen übertragen werden. Auch in dieser Datei existiert wieder eine Funktion, welche Regelungsparameter in eine Protobuf-Nachricht umwandelt. Auch diese wurde nur der Vollständigkeit wegen implementiert und wird im Projekt nicht genutzt.

Zum Ein- und Entpacken der Messages in Frames, können die Funktionen "ProtoToFrame" bzw. "FrameToProto" aus der Datei Frameparser.c genutzt werden. Bei der Erstellung des Frames wird der übergebene Puffer auf die richtige Größe gebracht. Weiter werden am Anfang die Anzahl der Bytes des gesamten Frames und am Ende zwei Bytes mit dem CRC-Wert abgelegt. Um diesen Wert zu erhalten werden die durch DAVE generierten Funktionen "CRC\_SW\_CalculateCRC" sowie "CRC\_SW\_GetCRCResult" genutzt. Bei der Nutzung der Funktion "ProtoToFrame" ist darauf zu achten, dass das erste Element des übergebenen Puffers keine Daten der Payload enthält, da hier die Framelänge abgelegt wird.

In der Funktion "FrameToProto" werden auch die genannten Funktionen zur Generierung des CRC-Wertes genutzt. Hier wird jedoch das Ergebnis auf 0 überprüft. Wie bereits geschildert, bedeutet beim Entkapseln des Frames ein CRC-Wert gleich 0, dass kein Fehler während der Übertragung aufgetreten ist. Ist dies der Fall, so wird der Bearbeitungspuffer um 2 Elemente verkleinert. Das dritte nun überschüssige Byte des Puffers wird nicht gelöscht, da ein dadurch notwendiges Verrücken



der Payload im Puffer zu ressourcenintensiv ist. Allerdings wird die Größe der Payload durch die Funktion "FrameToProto" richtig zurückgegeben.

Außerdem existieren noch zwei Header-Dateien `com_types.h` und `SensorIDs.h`. In der Ersteren befinden sich die Definitionen für die Sensordata-Struktur, welche die Regelung zur Übergabe der Sensorwerte nutzt, sowie der Aufzählung der möglichen Regelgröße. In der Letzteren wurden die ID's der einzelnen Sensoren für die Zuordnung der Werte abgelegt.

### 2.3.3 Parametrierung der DAVE Apps

In den Bildern 2.4 bis 2.9 werden alle in den DAVE Apps getroffenen Einstellungen aufgeführt. Die notwendigen Apps "UART [4.1.8]" und "CRC\_SW [4.0.6]" werden über den Menüpunkt "DAVE -> Add New APP ..." hinzugefügt.

General Settings

Advanced Settings

Interrupt Settings

Pin Settings

Operation mode:

Full Duplex

Desired speed [baud]:

9600

Actual speed [baud]:

9600

Data bits:

8

Stop bit:

1 Stop Bit

Parity selection:

Even Parity

**Bild 2.4:** Grundlegende Einstellungen der UART-App: Diese müssen mit den Einstellungen der PC-Bibliothek übereinstimmen.

General Settings

Advanced Settings

Interrupt Settings

Pin Settings

Transmit

Interrupt Priority

Preemption priority

63

☒ End of transmit callback:

DataTxICR

Receive

Interrupt Priority

Preemption priority

63

☒ End of receive callback:

DataRcvICR

**Bild 2.6:** Interrupt Einstellungen der UART-App: Hier werden die Callback-Funktionen angegeben.

General Settings

CRC Configuration

☒ Initial CRC value:

0xFFFF

CRC width[8-bit to 32-bit]:

16

Polynomial:

4129

dec

Algorithm Control

☐ Input data bitwise reflection

☐ Output CRC bitwise reflection

☐ Output CRC inversion

**Bild 2.8:** Einstellungen der CRC-App: Auch hier müssen die Einstellungen mit denen der PC-Bibliothek übereinstimmen.

General Settings

Advanced Settings

Interrupt Settings

Pin Settings

Protocol Handling

Transmit mode:

DMA

Receive mode:

DMA

Timing Settings

Oversampling:

16

FIFO Settings

☒ Enable transmit FIFO

Size:

16

☒ Enable receive FIFO

Size:

4

A total of 64 FIFO entries are available to be configured as transmit and receive buffers. These 64 entries are additionally shared between the two channels of a USART module

**Bild 2.5:** Weiterführende Einstellungen der UART-App: Beide Richtungen werden per DMA gehandhabt.

General Settings

Advanced Settings

Interrupt Settings

Pin Settings

☒ Enable advanced pin characteristics

Transmit

Mode:

Push Pull

Driver strength:

Don't Care

Receive

Mode:

Tristate

**Bild 2.7:** GPIO Einstellungen der UART-App

Filter: USART0

APP Instance Name	APP Pin Name	Pin Number (Port)
UART_0	Receive Pin	#108 ( P1.4 )
	Transmit Pin	#107 ( P1.5 )

**Bild 2.9:** Zuordnung der GPIO-Pins unter "DAVE -> Manual Pin Allocator"; Die Pins P1.4 und P1.5 sind mit dem Debug-Chip verbunden. Siehe (boa16, Table 5)



Bild 2.10: Inhalt eines falsch interpretierten Bearbeitungspuffers



Bild 2.11: Verbessertes Nachrichtendesign mit Start of Frame

## 2.4 Integration

Die Integration der beiden Module in die Regelung erfolgte hauptsächlich durch Herrn Krupinski. Auf Seiten der GUI übernahm diese Aufgabe Herr Krause. Herr Schleinkofer war in beiden Fällen nur beratend tätig. Seine Hauptaufgabe während dieser Phase bestand in der Behebung der auftretenden Fehler. Bei einem ersten Zusammenfügen von Regelung und GUI wurde ersichtlich, dass das Senden der Nachrichten mit den Sensorwerten nur unzureichend robust war. Die Ursache bestand hauptsächlich im Design der Nachrichten und des Kommunikationsablaufs. Sind im Eingangspuffer des SerialPort-Objekts noch Rest-Daten, so interpretiert die Kommunikationsbibliothek diese fälschlicherweise als Beginn einer neuen Nachricht. Das erste Byte wird daher als Framelänge interpretiert. Ein neuer Frame wird dann nur unvollständig in den Bearbeitungspuffer übernommen (siehe Bild 2.10). So befindet sich dort am Anfang die Hälfte der alten Nachricht und am Ende der Anfang der neuen. Dies führt zu einem Verwerfen des Inhalts im Bearbeitungspuffer. Dieses Verhalten wiederholt sich nun im weiteren Programmverlauf, sodass nun keine Sensordaten mehr empfangen werden können. Sender und Empfänger sind nicht mehr im "gleichen Takt".

Um dies zu verhindern, wird nun zu Beginn jedes Frames vom  $\mu$ -Controller ein 16 Bit langer Start of Frame Delimiter (kurz: SoF) mit übertragen (siehe Bild 2.11). Es ist bei der Wahl des SoF darauf zu achten, dass diese Bitreihenfolge möglichst einzigartig in der Kommunikation ist. Aus Bekanntheitsgründen und mangels Vergleichswerten aus der vorliegenden Kommunikation wurde dabei auf einen Teil des Musters eines Ethernet-Frames gewählt. Das erste Byte des SOF ist 0x55 und das zweite 0xD5. Es wären aber auch andere Werte denkbar. Die Bibliothek prüft nun vor dem Verarbeiten eines Frames, ob am Anfang des SerialPort-Eingangspuffers nun diese zwei Bytes vorhanden sind. Dies geschieht mittels eines einfach implementierten Zustandsautomaten. In einer Schleife wird nun immer das erste Byte des Eingangspuffers gelesen und verworfen. Folgendes Bild verdeutlicht den im Anhang C implementierten Automaten.

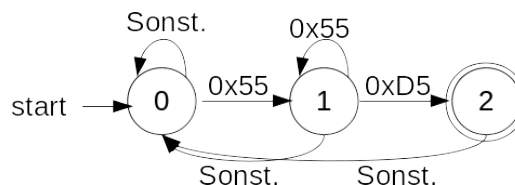


Bild 2.12: Zustandsautomat zur Erkennung des Start of Frame

## 2.5 Ausblick

Leider war bereits ab der Hälfte der Bearbeitungszeit absehbar, dass die anderen Arbeitspakete zeitmäßig nicht mehr auf den XE16x portiert werden können. Deshalb wurde auch das Kommunikations-Modul nicht mehr für den bisher in der Vorlesung Embedded Systems verwendeten Controller umgesetzt. Stattdessen war Herr Schleinkofer in anderen Arbeitspaketen (v.a. der Simulation) unterstützend tätig. Die Umsetzung des Projektes auf den zweiten Chip ist im Nachgang dieses Projektes noch möglich.

Einige Funktionen für eine zuverlässige Kommunikation konnten aufgrund der beschränkten Zeit ebenfalls nicht mehr umgesetzt werden. So ist es von Vorteil, wenn auch die Nachrichten mit den Regelungsparametern einen Start of Frame vorangestellt bekommen. Dies würde wie auch auf der GUI-Seite verhindern, dass eine unvollständig empfangene Nachricht das Erkennen eines Framebeginns verhindert.

Darüber hinaus wäre noch vorstellbar, für die Übertragung der Regelungsparameter eine Empfangsbestätigung zu implementieren. Dies würde einem eventuell notwendigen mehrfachen Sendeversuch durch den Benutzer vorbeugen.

Weiterhin wurde bei einem Review des erstellten Codes ersichtlich, dass die DAVE-App zur Erzeugung des CRC-Wertes keinen Gebrauch der Flexible CRC Engine auf dem  $\mu$ -Controller macht. Aufgrund des Zeitdrucks war eine detailliertere Einarbeitung in die Parametrierung der Hardware nicht mehr möglich.

## Chapter 3

# Arbeitspaket Sensorik

Die Analyse, Implementierung und Integration, sowie der Aufbau der benötigten Schaltungen für die Sensorik des Experimentierplatzes, war Aufgabe von Herrn Lackner mit Unterstützung von Herrn Brunner für Hardware-relevante Problemstellungen.

### 3.1 Überblick

Um den Betrieb des Motorexperimentierplatzes zu ermöglichen und sinnvolle Anwendungen zu finden, müssen eine Reihe von relevanten Kenngrößen, durch den Einsatz von Sensorik, gemessen und diversen Modulen, bzw. den Benutzern des Experimentierplatzes zur Verfügung gestellt werden.

Die zu erfassenden Daten, die in der Analysephase des Projekts festgelegt wurden, ergeben sich hauptsächlich aus den bereits verbauten und uns so zugänglichen Sensoren. Gemessen werden folgende Werte:

- **Drehwinkel**  
Aktueller Winkel der Motorwelle, relativ zu einem Festgelegten Nullpunkt.
- **Drehgeschwindigkeit**  
Umdrehungen der Welle pro Zeiteinheit.
- **Motor-Temperatur**  
Temperatur des Motors, da dieser sich im Betrieb erhitzt.
- **Hall Pattern**  
Stellt eine alternative Möglichkeit da, den Drehwinkel zu ermitteln. Das Pattern wird hauptsächlich zur Kommutierung eingesetzt, da sich daraus sowohl der passende Zeitpunkt als auch das korrekte Motor-Pattern ableiten lassen.

Für die Ermittlung definierten Werte stehen auf den Experimentierstand drei **Hall-Sensoren**, ein **Inkrementalgeber** und ein **Temperatursensor** zur Verfügung, die alle im Aufbau fest verbaut sind.

### 3.2 Entwicklungsumgebung

Wie auch die anderen Module, die direkt mit dem aufgebauten Motor interagieren, befindet sich die gesamte Software für die Sensorik auf dem Steuerungsmikrocontroller. Diese Aufgabe erfüllt momentan ein **XMC4800** von Infineon, aufgebaut auf einem Relax Kit V1. Da die Interaktion mit den Sensoren Hardware-nahe Programmierung erfordert, ist sämtlicher Code in C geschrieben. Als IDE für die Entwicklung dient DAVE 4 von Infineon.

### 3.2.1 DAVE 4

DAVE ist eine von Infineon bereitgestellte Entwicklungsumgebung für das Firmeneigene Controller Portfolio. Die Installationsdatei kann nach Registrierung bei Infineon direkt heruntergeladen werden. Während des Installationsprozesses werden ebenfalls automatisch die benötigten Treiber für den Debug-Chip auf dem Entwicklungsboard mit installiert.

#### Projekt anlegen

DAVE bietet Standardmäßig mehrere verschiedene Projekttypen an, die eine unterschiedliche Menge von bereits fertigem Code mit sich bringen. Bei einem **Empty Project** wird nur die HAL Bibliothek und die für den Controller benötigten Startup Dateien bereitgestellt. Das **Simple Main** Projekt erstellt zusätzlich noch eine main.c Datei, mit einer leeren Main-Funktion. Entscheidet man sich für ein **Easy Start** Projekt, erhält man Beispielcode der exemplarisch eine AD Wandlung vornimmt und einige Ausgänge setzt.

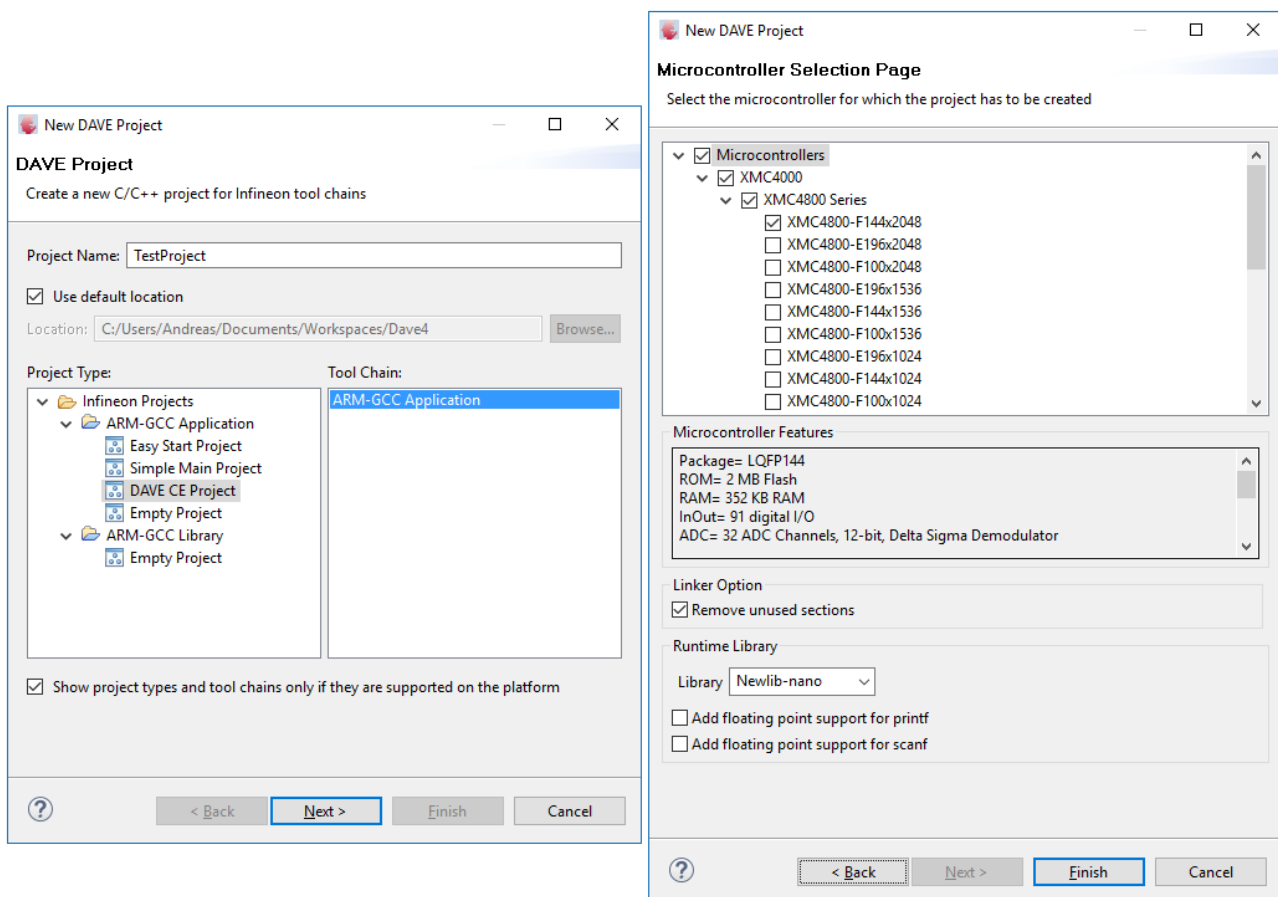


Bild 3.1: Projektdialog DAVE

Einen Sonderfall bei den zur Auswahl stehenden Projektvorlagen stellen **DAVE CE** Projekte da. Diese Art von Projekt, ermöglicht die Verwendung von grafischen Oberflächen für die Konfiguration der Hardwareabstraktion. Dies umfasst unter anderem die Verbindung einzelner Komponenten untereinander oder mit den Pins des Mikrocontrollers, sowie grafische Konfigurationsformulare für die Komponenten selbst (z.B. ADC, CCU, POSIF, DIO usw.).

Nach Wahl des Projektnamens und des gewünschten Projekttyps, muss auf der zweiten Seite des

Dialogs der verwendete Mikrocontroller ausgewählt werden, um die korrekte Hardwarebibliothek bereitstellen zu können. Ein exemplarisches Beispiel für die Projekterstellung befindet sich in Abbildung 3.1.

### Projekt erstellen und herunterladen

Um ein zu testendes Projekt auf den Controller herunterzuladen, muss es erst **kompiliert** werden. Gestartet wird der Buildprozess über den in Abbildung 3.2 markierten Button in der Menüleiste.

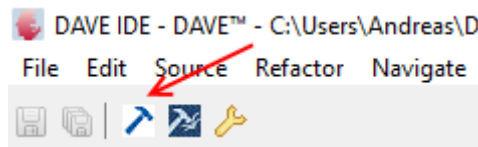


Bild 3.2: Buildprozess starten

Im Anschluss an einen erfolgreichen Erstellungsprozess wird eine **Debug- bzw. Run-Konfiguration** angelegt. Diese umfasst alle für das Flashen benötigten Parameter wie Ort der zu ladenden Binärdatei sowieso GDB Konfigurationen.

Es ist wichtig, dass das Projekt vor dem Anlegen der Konfiguration einmal erstellt wird, da sonst wichtige Parameter nicht automatisch eingetragen werden.

Das exemplarische Anlegen einer Debug-Konfiguration wird in Abbildung 3.3 gezeigt. Dafür wird im Fenster *Debug Configurations* durch Doppelklick auf den eingesetzten Debug-Chip automatisch eine fertige Konfiguration angelegt.

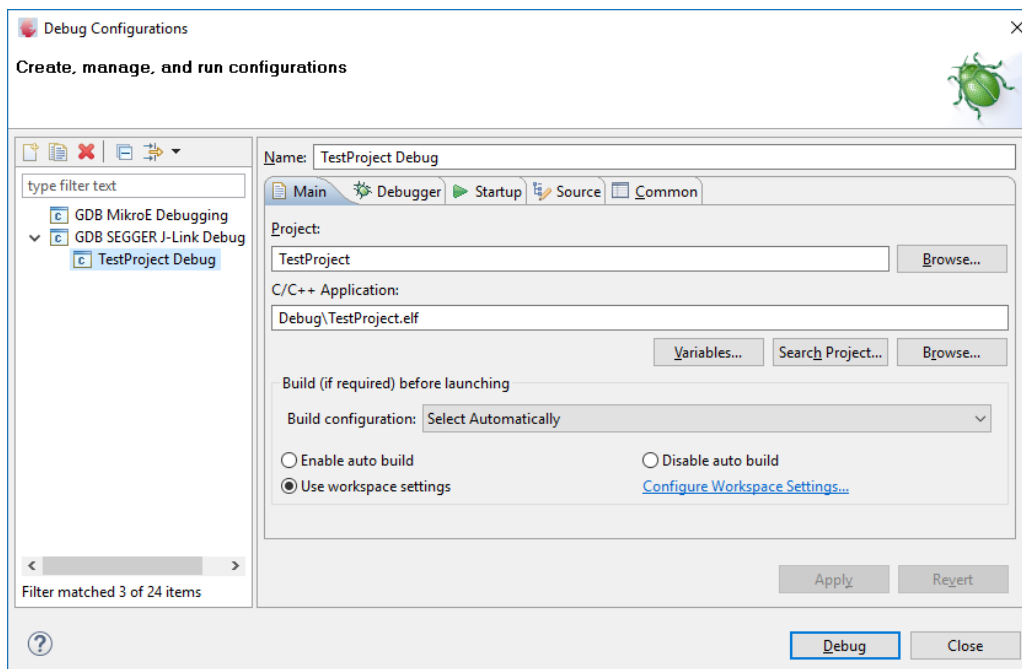


Bild 3.3: Debug-Konfiguration erstellen

**Wichtiger Hinweis!** Es kann vorkommen, dass aus bisher unbekannten Gründen falsche GDB Startup-Skripte generiert werden, die beim Versuch das Projekt zu Debuggen, den sofortigen Abbruch des Vorgangs zur Folge haben. Es muss sichergestellt werden dass in der Debug-Konfiguration, im

Tab *Startup*, unter dem Punkt *Run/Restart Commands* keine Anweisungen eingetragen sind!

Nach dem eine korrekte Konfiguration erstellt ist, kann das Programm über den Debug-Button in der Menüleiste auf den Controller geladen werden. DAVE wechselt im Anschluss automatisch in die Debug-Ansicht und unterbricht die Ausführung bei der ersten Anweisung in der Main-Funktion.

### 3.2.2 POSIF

Das POSIF Modul der XMC Controller ist eigens auf den Betrieb von Motoranwendungen zugeschnitten. Dabei übernimmt es, in Kooperation mit einigen Caputure Compare Units, den kompletten Workflow vom einlesen verschiedener Sensoren für die Bestimmung der Kommutierungszeitpunkte bis hin zur Ansteuerung des Motors, bzw. dem setzen der Ausgänge um die Spulen anzusteuern. Ein allgemeiner Aufbau befindet sich in Abbildung 3.4.

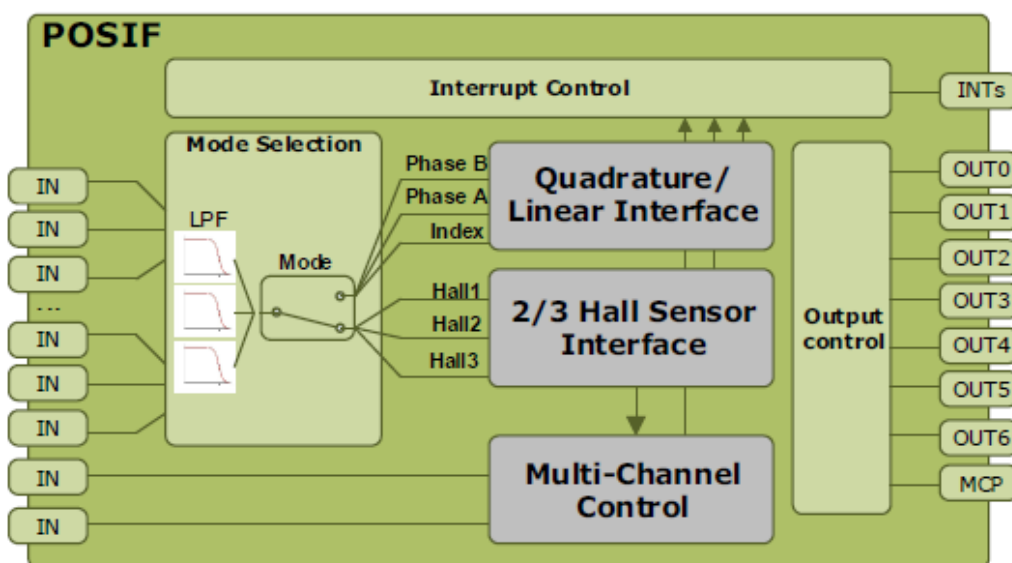


Bild 3.4: POSIF Übersicht

Im wesentlichen verfügt das POSIF Modul über zwei verschiedene Betriebsmodi zum einlesen des Zustands, die Optional mit dem Modul für die Motoransteuerung kombiniert werden können.

- **Hall Modus**

In diesem Betriebsmodus werden dem POSIF Modul die Signalpegel der Hall-Sensoren zugeführt. Wird eine Änderung detektiert, wird das gemessene Pattern, automatisch mit dem erwarteten verglichen und es wird entweder ein Interrupt für ein *Correct-Hall-Event* oder ein *Wrong-Hall-Event* ausgelöst. In der Interruptroutine muss im Code das Register für das aktuelle und erwartete Hall-Pattern aktualisiert werden, um den reibungslosen Betrieb zu ermöglichen.

- **Quadrature Decoder Modus**

Dieser Modus verwendet als Informationsquelle einen Inkrementalgeber, der eine Indexleitung, sowie zwei Phasen A und B bereitstellt (alternativ kann auch ein Encoder mit nur *Direction-* und *Count-Leitung* verwendet werden). Unter Zuhilfenahme einer kompletten CCU lassen sich sowohl Umdrehungsgeschwindigkeit als auch Drehwinkel berechnen.



### 3.3 Sensor Interface

Sämtliche Funktionen die das Auslesen der definierten Sensorwerte ermöglichen sollen, sind in ein eigenes Softwaremodul gekapselt, dass die gewünschte Funktionalität über festgelegte Schnittstellen anbietet.

**Code 3.1:** Sensor Interface

```

1 void Sensor_Init();
2
3 void Sensor_StartAll(void);
4
5 void Sensor_StopAll(void);
6
7 void Sensor_SetDirection(MotorDirection_t direction);
8
9 Std_ReturnType Sensor_RegisterHallCallback(HallCallbackType callback);
10
11 Std_ReturnType Sensor_GetCurrentHallPattern(HallPattern_t* pattern);
12
13 Std_ReturnType Sensor_GetVelocity(double* velocity);
14
15 Std_ReturnType Sensor_GetAngle(double* angle);
16
17 Std_ReturnType Sensor_GetTemperature(int* temperature);

```

#### 3.3.1 Modulstruktur

Das angebotene Interface über die Sensor.h Datei, fungiert nur als Schnittstelle zu den Implementierungen der einzelnen Sensoren. Die Aufteilung in Submodule leitet sich dabei aus den verschiedenen Sensortypen her.

```

Sensor
├── Sensor_Types.h
├── Sensor_Hall.h
├── Sensor_QuadratureDecoder.h
└── Sensor_Temperature.h

```

Alle Submodule bringen zum jetzigen Zeitpunkt ihre Abhängigkeiten<sup>1</sup> schon mit, um den Integrationsaufwand in das fertige Projekt so gering wie möglich zu halten. Dies kann sich zu einem späteren Zeitpunkt noch ändern, wenn die POSIF Konfiguration für das einlesen der Hall-Sensoren mit der Motorsteuerung zusammengezogen wird, da beide Module auf die selbe Hardwarekomponente zugreifen.

#### 3.3.2 Anwendung

Um Sensorwerte über das Softwaremodul einzulesen muss beim Startvorgang der Steuerung einmal die *Sensor\_Init()* Funktion aufgerufen werden um alle eingesetzten Hardwarekomponenten zu initialisieren. Zu diesem Zeitpunkt sollte auch ein Callback für die Hall-Events registriert werden, um über

<sup>1</sup>Mit Abhängigkeiten sind in erster Linie Konfigurationen der Hardwareelemente wie CCU, ADC oder POSIF gemeint.

eintretende Änderungen der Motorwelle informiert zu werden. Registrieren lässt sich eine Callback-Funktion über `Sensor_RegisterHallCallback(<func_ptr>)`.

Vor dem Start muss dem Sensormodul ebenfalls die gewünschte Drehrichtung des Motors, über die Funktion `Sensor_SetDirection(<motor_dir>)` mitgeteilt werden, um das korrekte Hall-Pattern ermitteln zu können.

Der Einlesevorgang für die Sensorwerte muss im Anschluss über die Funktion `Sensor_StartAll()` gestartet werden, und lässt sich über `Sensor_StopAll()` anhalten.

Während des Regelbetriebs lassen sich die einzelnen Werte über die restliche API auslesen.

- **Sensor\_GetCurrentHallPattern(HallPattern\_t\*)** Gibt das Muster zurück, dass sich auch den aktuelle Zustand der digitalen Hall-Sensoren ergibt. HallPattern\_t\* beschreibt dabei einen Pointer auf die Hall-Pattern Struktur die durch den Funktionsaufruf mit Werten gefüllt wird.
- **Sensor\_GetVelocity(double\*)** Gibt die aktuelle Umdrehungsgeschwindigkeit der Motorwelle in Umdrehungen pro Sekunde zurück.
- **Sensor\_GetAngle(double\*)** Gibt den aktuellen Drehwinkel in Grad, relativ zu einem festgelegten Nullpunkt zurück.
- **Sensor\_GetTemperature(int\*)** Gibt die aktuelle Temperatur in Grad Celsius im Motor zurück.

### 3.4 Hall Sensoren

Ein Hall-Sensor misst magnetische Flussdichte, durch die Ausnutzung des Hall-Effekts. In unserem Versuchsaufbau sind drei dieser Sensoren, jeweils im Winkel um  $120^\circ$  versetzt, im Motor verbaut (Abbildung 3.5). Es handelt sich dabei um digitale Sensoren, was bedeutet dass der gemessene Sensor-Wert entweder logisch 1 oder 0 ist.

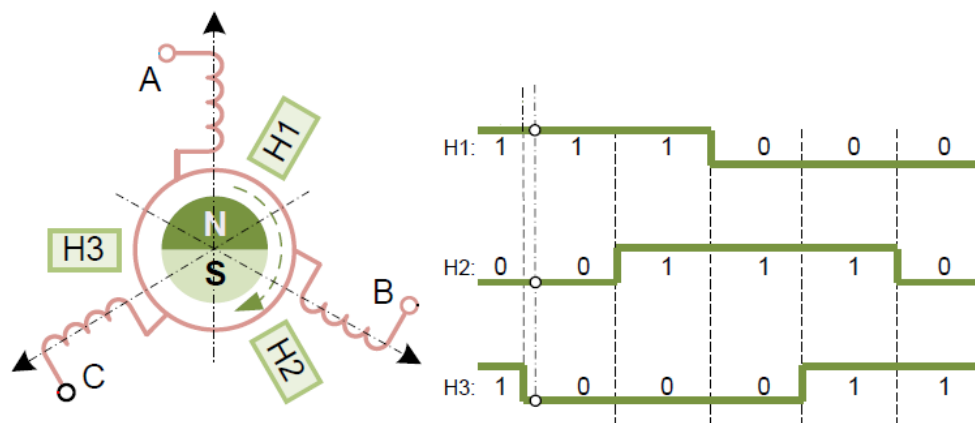


Bild 3.5: Hall Sensoren und Pattern

Der Wert ändert sich dabei in Abhängigkeit davon, ob sich gerade ein magnetischer Nord- bzw. Südpol über den Sensor befindet. Alle drei Sensorzustände zusammen genommen ergeben ein **Hall-Pattern**, was im rechten Teil von Abbildung 3.5 noch einmal verdeutlicht wird. Das Hall-Pattern ist relevant für zwei Dinge. Zum einen kann Anhand des gemessenen und des erwarteten Pattern festgestellt werden, ob beim Drehvorgang Fehler aufgetreten sind (z.B. ob der Motor sich in die falsche Richtung dreht) und zum anderen lässt sich aus dem Zeitpunkt des Auftretens einer Änderung des Musters, eine passende Gelegenheit für die Kommutierung ableiten. Bei gewissen Regelstrategien

lässt sich außerdem aus dem Hall-Pattern ein Motor-Pattern ableiten, welches an die Spulen angelegt wird um eine Drehung des Motors zu erzeugen.

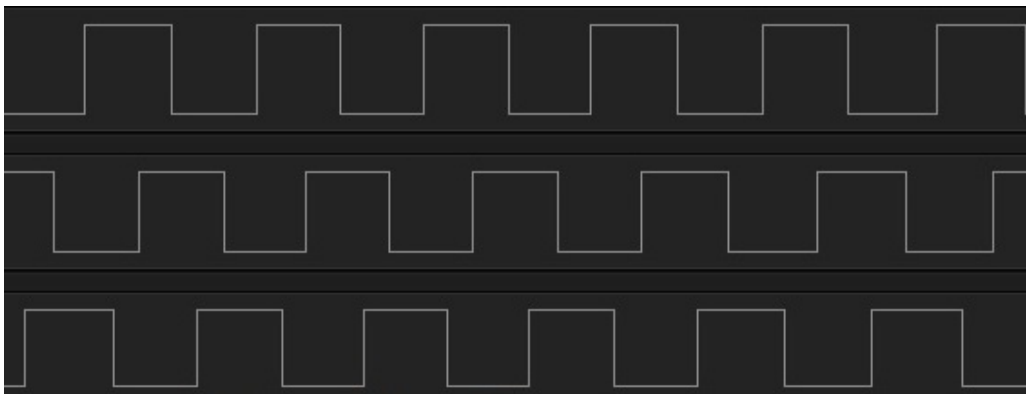
### 3.4.1 Einlesen der Hall-Sensoren

Die für uns effizienteste Möglichkeit das Hall-Pattern einzulesen und seine Korrektheit zu prüfen ist die Verwendung des in Kapitel 3.2.2 beschriebenen POSIF Moduls (POSition InterFace), dass die Verwendeten XMC Mikrocontroller mitbringen.

In der verwendeten Konfiguration benötigt der Hall-Modus zwei CCU Slices. Wird vom POSIF Modul die Änderung der Eingänge detektiert, startet es das erste CCU Slice als Debounce Timer, um auszuschließen dass die Statusänderung ihren Ursprung in einem fehlerhaften Signal hat. Anschließend erfolgt der Vergleich, mit dem im Register abgelegten erwarteten Pattern, was entweder die Generierung eines Correct-Hall-Events oder Wrong-Hall-Events zur Folge hat. Im Fall eines CHE, löst die Software das registrierte Hall-Callback aus, um die Motorsteuerung über das eingetretene Ereignis zu benachrichtigen. Ebenfalls wird im Anschluss daran, das Register für das nächste erwartete Hall-Pattern aktualisiert. Das zweite CCU Slice kann verwendet werden, um auf Basis der auftretenden Hall-Events die aktuelle Umdrehungsgeschwindigkeit bzw. den Winkel der Motorwelle zu berechnen.

### 3.4.2 Ermittlung der Hall-Pattern

Der reibungslose Betrieb des Hall Moduls im Sensor Interface setzt voraus, dass das Register des POSIF Moduls immer mit dem korrekten, erwarteten Hall-Pattern befüllt wird. Dieses Muster unterscheidet sich je nach verwendetem Motor, und musste in unserem Fall extra ausgemessen werden. Dazu habe ich die Hall-Sensoren mit Spannung versorgt und an jede der drei Leitungen ein Oszilloskop gehängt. Durch händisches Drehen des Motors wird die gewünschte Abfolge sichtbar (abgebildet in Abbildung 3.6).



**Bild 3.6:** Eingelesenes Hall-Pattern

Auf Basis dieser Messung lässt sich durch festlegen eines beliebigen Startpunkts das Hall-Pattern für die gewählte Drehrichtung ableiten. Ab dem gewählten Startpunkt führt jede Änderung des Zustands zu einem neuen Pattern. Die Analyse hat für unseren Motor 6 verschiedene Pattern ergeben, die sich pro Umdrehung 7-Mal wiederholen. Dies ergibt eine Gesamtzahl von 42 Hall-Events pro Umdrehung. Das gleiche Verfahren wurde zu Überprüfungszwecken auch mit der zweiten Drehrichtung durchgeführt, um Fehler zu vermeiden.

Um einen effizienten Wechsel der Pattern zu garantieren, wird eine Lookup-Table eingesetzt, die das aktive Hall-Pattern als Index nutzt, um das nächste erwartete Pattern zu finden. Die genaue Struktur der Tabelle ist in Abbildung 3.7 aufgeschlüsselt.

					CCW			CW		
	H3	H2	H1		Index	EHP	CHP	Index	EHP	CHP
1	1	0	1	5	0	-	-	0	-	-
2	0	0	1	1	1	3	1	1	5	1
3	0	1	1	3	2	6	2	2	3	2
4	0	1	0	2	3	2	3	3	1	3
5	1	1	0	6	4	5	4	4	6	4
6	1	0	0	4	5	1	5	5	4	5
					6	4	6	6	2	6

Bild 3.7: Hall-Pattern Lookup-Tables

### 3.5 Inkrementalgeber

Für die Berechnung der Umdrehungsgeschwindigkeit und des aktuellen Winkels steht ein Inkrementalgeber zur Verfügung, der über eine Indexleitung Z und zwei Phasen A und B verfügt. Der Sensor selbst ist mit der Motorwelle gekoppelt und verändert seine Leitungspegel mit den Drehbewegungen der Welle.

Die Umdrehungsgeschwindigkeit lässt sich über die Indexleitung errechnen, die pro vollständiger Umdrehung der Welle an einem definierten Punkt für kurze Zeit einen High-Pegel annimmt. Der Winkel sowie die Drehrichtung lassen sich über die Phasen A und B ermitteln. Wie in Abbildung 3.8 gezeigt, wechseln sowohl A als auch B immer zwischen einem High- und Low-Zustand. Da nicht beide Phasen gleichzeitig den Pegelwechsel vollziehen, sondern je nach Drehrichtung eine Phase vor der anderen, lässt sich daraus die Drehrichtung ableiten. Den aktuellen Winkel erhält man durch Mitzählen, da pro Umdrehung immer die gleiche definierte Anzahl an Flankenwechsel durchgeführt werden. Der absolute Winkel, relativ zum Nullpunkt den die Indexleitung vorgibt, erhält man durch die Formel 3.1.

$$angle = (edge_{count} / total_{edges}) * 360 \quad (3.1)$$

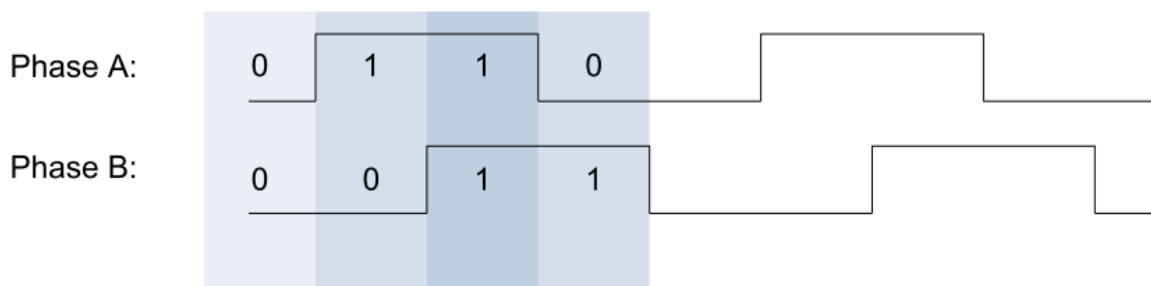


Bild 3.8: Inkrementalgeber Phasen

#### 3.5.1 Implementierung

Für das Einlesen und die Verarbeitung der Leitungssignale des Inkrementalgebers wird das im Kapitel 3.2.2 beschriebene POSIF Modul im Quadrature Decoder Modus herangezogen. Durch zuführen der Phasen und der Indexleitung errechnet die verwendete POSIF Instanz sechs verschiedene Ausgänge.

- Auftreten des Indexsignals
- Fertige Umdrehung

- Drehrichtung
- Quadrature Clock
- Period Clock
- Synchronisierter Start

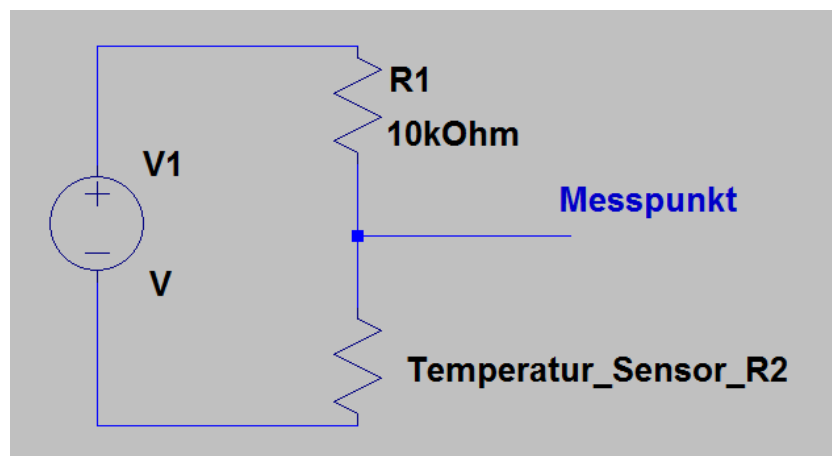
In Kombination mit passend konfigurierten CCU Slices lassen sich Anhand der internen Ausgänge des POSIF Moduls die angestrebten Kenngrößen ermitteln.

## 3.6 Temperatursensor

Der eingesetzte Sensor für die Ermittlung der Temperatur im Motor verwendet einen NTC-Widerstand (Negative Temperature Coefficient) für die Temperaturermittlung. Bei dieser Bauform, sinkt der Widerstand des Sensors bei steigender Temperatur. Die bei uns verbaute Variante deckt einen Temperaturbereich von  $-40^{\circ}$  bis  $110^{\circ}$  Celsius ab und verfügt über einen Basiswiderstand von 10kOhm bei einer Temperatur von  $25^{\circ}$  Celsius.

### 3.6.1 Einlesen des Temperaturwerts

Für die Ermittlung des aktuellen Temperaturwerts ist es notwendig den Widerstandswert des Sensors einzulesen und zu digitalisieren. Da sich der Widerstand nicht direkt messen lässt wird ein Spannungsteiler eingesetzt, dessen Spannung am Messpunkt sich proportional zum Widerstand verhält. Diese Messspannung kann am Controller über einen AD-Wandler eingelesen werden.



**Bild 3.9:** Spannungsteiler zum einlesen des NTC-Widerstands

Der genaue Schaltungsaufbau lässt sich aus Abbildung 3.9 entnehmen. Als fixer Widerstand werden 10kOhm verwendet. Dieser Wert spielt später bei der Errechnung des Widerstands des Temperatursensors T1 eine Rolle. Nach anlegen einer Versorgungsspannung lässt sich die variable Spannung am Messpunkt abgreifen.

Um aus der gemessenen Spannung den Widerstand zu errechnen, muss zunächst von der Formel für  $U_2$  des Spannungsteiler ausgegangen werden.

$$U_2 = \frac{U_{ges}}{R_1 + R_2} * R_2 \quad (3.2)$$

Bekannt sind in dieser Formel die Werte für  $U_{ges}$ , die gemessene Spannung  $U_2$  und der Widerstand  $R_1$ . Durch Äquivalenzumformung lässt sich die Formel auf den gewünschten Wert  $R_2$  umstellen.

$$R_2 = \frac{U_2 * R_1}{U_{ges} - U_2} \quad (3.3)$$

Ist der Widerstand bekannt, lässt sich aus dem Datenblatt des Sensors die dazu passende Temperatur ermitteln.

### Analog Digital Wandlung

Die ADC Einheit des XMC Controllers verfügt über vier unabhängige Konvertierungsgruppen mit bis zu 8 analogen Eingängen und einer maximalen Auflösung von 12 Bit. Standard Referenzspannung sind dabei 3.3 Volt.

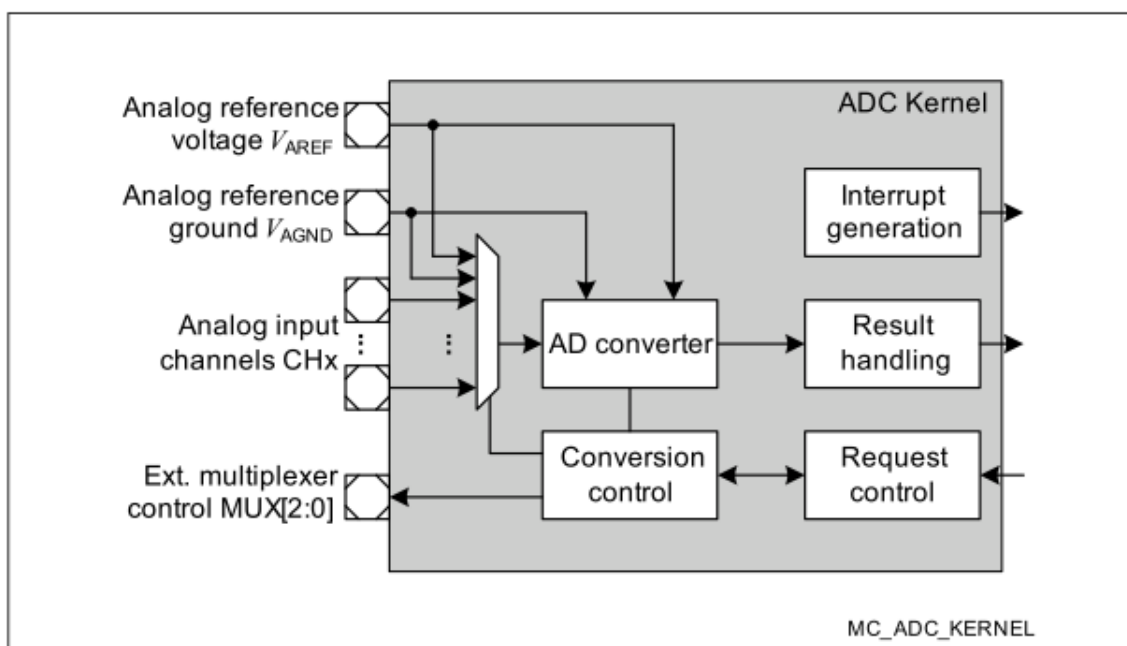


Bild 3.10: ADC Einheit der XMC Controller

Für die Wandlung stehen drei verschiedene Modi zur Verfügung:

- **Fixed Channel Conversion**  
Ein einzelner festgelegter Kanal wird konvertiert.
- **Auto Scan Conversion**  
Alle verfügbare Kanäle werden in einer konfigurierbaren Sequenz automatisch und linear gewandelt.
- **Channel Sequence Conversion**  
Konvertierung von 8 willkürlich gewählten Kanälen.

Jeder dieser Modi kann entweder einmalig auf explizite Anfrage, oder kontinuierlich eine Konvertierung durchführen. Als Quelle der Anfragen stehen ebenfalls mehrere Einheiten zur Verfügung (zwei Group-Request-Sources und eine Background-Request-Source). Da von mehreren Quellen Anfragen gleichzeitig eintreffen können, wird die Reihenfolge der Abarbeitung von einem Arbiter festgelegt. Der genaue Aufbau einer einzelnen ADC Einheit lässt sich der Abbildung 3.10 entnehmen.

In unserem konkreten Beispiel wird eine Fixed-Channel-Konvertierung durchgeführt, da nur ein einzelner Kanal eingelesen werden muss. Das absetzen der Anfrage wird dabei von der Background-Request-Source übernommen. Da die Konvertierung einige Mikrosekunden Zeit benötigt, wird nach Abschluss des Vorgangs ein Interrupt ausgelöst.

### Implementierung

Die Ermittlung des aktuellen Temperaturwerts wird nur auf Anfrage durch die Sensor API ausgeführt. Die im Codesnippet 3.2 aufgeführte Funktion, startet die Konvertierung und wartet durch Busy-Waiting auf deren Abschluss. Zu dem gemessenen Spannungswert wird wie bei 3.6.1 aufgeführt, der zugehörige Widerstandswert berechnet. Der passenden Temperaturwert wird durch die Suche in einer sortierten Tabelle ermittelt.

**Code 3.2:** Temperaturberechnung

```

1  int Sensor_Temperature_Calculate(Sensor_TemperatureType sensor)
2  {
3      int i = 0;
4      double adcVoltage = 0;
5      double res = 0;
6
7      /* Start measurement */
8      MeasurementRunning = 1;
9      Sensor_Temperature_ADC_StartConversion();
10
11     /* Wait for measurement to finish*/
12     while(MeasurementRunning == 1);
13
14     /* Calculate temperature */
15     adcVoltage = ConvertToVoltage(SensorVoltages[sensor]);
16     res = (adcVoltage * R1) / (SENSOR_REF_VOLTAGE - adcVoltage);
17
18     while(TemperatureLookupTable[i++].resistance > res &&
19           i < NTC_LOOKUP_ENTRIES);
20
21     return TemperatureLookupTable[i].temperature;
22 }
```

## 3.7 Ausblick

Momentan befindet sich der Code für das Einlesen der Sensorwerte vom Inkrementalgeber noch in Entwicklung. Bei der aktuellen Bearbeitungslage ist eine vollständige Fertigstellung nicht absehbar. Basierend auf der Ideensammlung aus der Analysephase wäre eine mögliche Erweiterung dieses Arbeitspakets die Portierung auf andere Controller. Durch das mögliche Fehlen des POSIF Moduls auf anderen Plattformen wird dadurch eine Neuimplementierung für das Einlesen der Hall-Sensoren sowie des Inkrementalgebers notwendig.

Eine weitere Aufgabe wäre die Kommutierung nicht von den Hall-Events abhängig zu machen, sondern als Basis den Inkrementalgeber zu verwenden. Dies ermöglicht die Verwendung neuer Kommutierungsstrategien und stellt mit Sicherheit eine interessante Erweiterung des Funktionsumfangs da.

## Chapter 4

# Arbeitspaket Regelung

Für die Analyse, Entwurf, Implementierung und Integration dieses Arbeitspaketes übernahm Herr Krupinski die Verantwortung.

Die Regelung stellt im wesentlichen “Glue-Code” für die Kommunikation, Motorsteuerung und Sensorik bereit. Dabei ist der Kern eine Implementierung eines PID(proportional–integral–derivative)-Regler.

### 4.1 Analysephase

#### 4.1.1 Hardware

Für die Regelung ist es möglich die Programmierung nahezu Hardware unabhängig zu realisieren. Dafür müssen die  $\mu$ -Controller abhängigen Komponenten (Kommunikation, Motorsteuerung und Sensorik) über ein Interface abstrahiert werden. Die Regelung “kennt” und kommuniziert nur gegen diese Interfaces, was es ermöglicht die spezifischen Implementierungen jederzeit austauschen zu können, ohne die Regelung erneut ändern zu müssen.

Da für den I- und D-Anteil der Regelung allerdings eine relative Zeit benötigt wird, ist es nicht komplett möglich  $\mu$ -Controller unabhängig zu entwickeln. Um diese relative Zeit messen zu können, müssen Timer-Komponenten der Hardware genutzt werden. Die einzige Möglichkeit dies zu vermeiden, wäre eine Annahme zu treffen, wie viel Zeit seit der letzten Iteration vergangen ist.

#### 4.1.2 Software

Der Motor soll auf verschiedene Werte hin gesteuert werden können. Zum Beispiel auf eine bestimmte Drehzahl oder Drehmoment. Grundsätzlich ist dies davon abhängig, welche Werte der Controller über die Sensorik lesen und welche Werte über die Motorsteuerung direkt, oder indirekt beeinflusst werden können. Deshalb muss es möglich sein ohne großen Aufwand die Regelung auf mehrere Werte zu erweitern, beziehungsweise grundsätzlich dafür entworfen sein. Zusätzlich müssen diese Parameter zur Laufzeit, über die Kommunikations-Schnittstelle, verändert werden können. Konkret die Einflussfaktoren der P-, I- und D-Anteile, Zielwert und zu regelnden Wert.

### 4.2 Entwurfsphase

Hauptteil des Entwurfs bestand aus der konkreten Definition der Interfaces zwischen Regelung und Kommunikation/Motorsteuerung/Sensorik.



### **4.3 Implementierung**

### **4.4 Integration**

### **4.5 Ausblick**

## Chapter 5

# Arbeitspaket GUI

Für die Analyse, den Entwurf, die Implementierung dieses Arbeitspaketes übernahm Herr Krause die Verantwortung, wobei er bei der Implementierung vom Herrn Krupinski unterstützt wurde. Während der Integration des Arbeitspaketes waren die Herren Krupinski und Schleinkhofer unterstützend eingebunden.

In der Projektspezifikation wurde die Anforderung der Visualisierung der Motor-Charakteristiken und der Steuerung des Motors mit aufgenommen. Zu diesem Zweck wurde eine Benutzeroberfläche in der Programmiersprache C# mit Hilfe des Grafik-Framework<sup>1</sup> *WPF*<sup>2</sup> von Microsoft erstellt.

### 5.1 Analysephase

Das Arbeitspaket GUI wurde mit einer Analysephase begonnen. In dieser Phase wurden die möglichen Funktionalen und Nicht-Funktionalen Anforderungen zusammengetragen und aufgelistet. Zu dem Zweck Anforderungen zu finden, wurden Stakeholder<sup>3</sup> befragt, anschließend wurden die Ergebnisse der Befragung validiert und es wurde eine Priorisierung für die umzusetzenden Projekteinhalte durchgeführt.

Die wichtigsten drei Anforderungen jeder Gruppe, basierend auf der Analyse:

#### **Funktionale Anforderungen:**

- Anzeige von Sensordaten in der GUI
- Regelung möglicher Parameter des Motors
- Einstellung der PID Regelung in der GUI

#### **Nicht-Funktionale Anforderungen:**

- Die Software soll Modular aufgebaut und erweiterbar sein
- Es soll das aktuelle Metro-Design verwendet werden

Als weiteres Ergebnis der Analyse ergab sich eine zwingende Abhängigkeit zu dem Arbeitspaket Kommunikation, wo eine enge Zusammenarbeit in der Definition der Schnittstelle bestand.

### 5.2 Entwurfsphase

Mit den Ergebnissen der Analysephase, wurde ein Entwurf der Benutzeroberfläche erstellt. Der Entwurf wurde auf der einen Seite für die Code Struktur durchgeführt, auf der anderen Seite wurden ebenfalls bereits Designentwürfe des Layouts erstellt.

---

<sup>1</sup>Rahmenstruktur in der Software

<sup>2</sup>Windows Presentation Foundation

<sup>3</sup>Projektbeteiligte, welche nicht an der Entwicklung beteiligt sind

Als grundlegende Architektur wurde ein drei Schichtenmodell<sup>5.1</sup> verwendet, welches den Anforderungen der Analysephase gerecht wird und allen Entwicklern des Projektes bekannt ist. In diesem gewählten Modell kommunizieren nur benachbarte Schichten miteinander. Und jede Schicht hat eine separate Aufgabe.

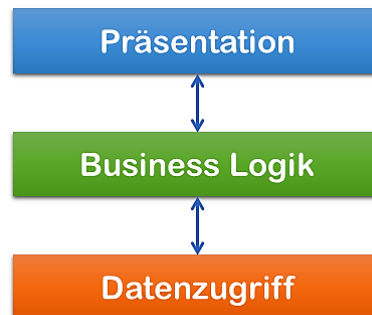


Bild 5.1: Grober Architektur Entwurf

Es wurden mehrere Entwurfsmuster verwendet, welche zur Softwarequalität beitragen und dem Entwickler einzelne Lösungsgerüste für wiederkehrende Probleme liefern. Für die oberen beiden Schichten wurde das MVVM<sup>4</sup> Entwurfsmuster<sup>5.2</sup> gewählt. Dieses Muster trennt die Oberfläche von der zugrundeliegenden Business-Logik, dies ermöglicht es die Oberfläche unabhängig und austauschbar von der Businesslogik zu entwickeln.

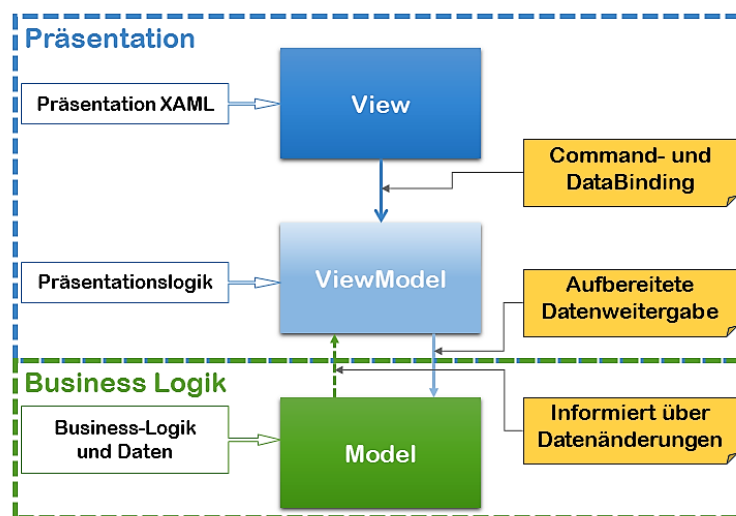


Bild 5.2: MVVM-Konzept

Weitere eingesetzte Entwurfsmuster sind das Kommandomuster, welches alle ausgeführten Events auf der Benutzeroberfläche auffängt und im ViewModel behandelt und verarbeitet, das Repository<sup>5</sup>-Muster, welches die Daten Verwaltung übernimmt und flexible auf Änderungen im Kommunikationsmodul reagiert und das "Umkehrung der Kontrolle"-Muster, für dessen Zweck ein Abhängigkeits-Container zu Beginn des Systemstarts erstellt und gefüllt wird und anschließend den Zugriff auf die im Container befindlichen Objekte von jedem Punkt der Anwendung her ermöglicht.

<sup>4</sup>kurzw. für Model-View-ViewModel

<sup>5</sup>engl. Haufen

Wie bereits in der Analysephase erwähnt, wurden die Datenstrukturen dem Kommunikationsmodul angepasst siehe Abbildung 5.1.

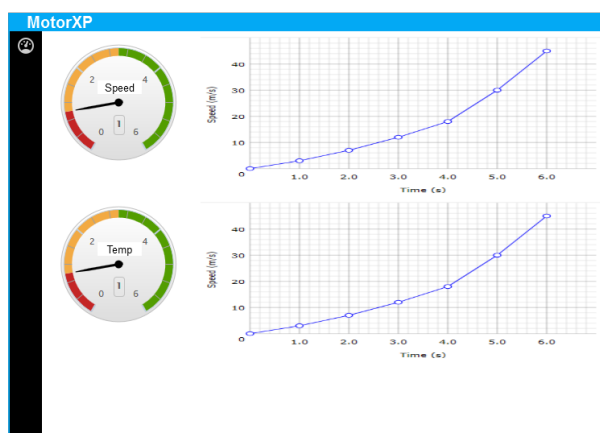
**Code 5.1:** Beschreibung der Sensordatenstruktur

```

1 public class SensorData
2 {
3     public ulong Timestamp { get; set; }
4     public SortedList<ushort, double> DataTable { get; set; }
5 }

```

Im zweiten Teil des Entwurfes wurde ein erstes einfaches Layout<sup>5.3</sup> erstellt. Für dieses wurde das Werkzeug Pencil<sup>6</sup> eingesetzt. Es ist einfach zu bedienen und lieferte schnell ein Ergebnis, welches im Projektteam besprochen werden konnte.



**Bild 5.3:** Erster Layout Entwurf der GUI

## 5.3 Implementierung

In diesem Abschnitt werden die Ergebnisse der Entwurfsphase umgesetzt und die Benutzeroberfläche erstellt. Das Programm wurde mit dem Visual Studio 2015 in der Programmiersprache C# und den .NET Framework WPF erstellt. Diese Phase wurde durch den Herrn Krupinski unterstützt.

Zu Beginn der Implementierung wurde die Frage aufgeworfen, welche Technologie für die Oberfläche zum Einsatz gebracht werden sollten. Zur Auswahl stand, es mit neuesten Multi-Plattform fähigen Web-Technologien umzusetzen und auf das .NET ASP-Framework<sup>7</sup> zu setzen oder eine reine Desktop Applikation mit dem WPF-Framework umzusetzen. Aufgrund des relativ knappen Zeitplanes für das Projekt und der mehrheitlichen Erfahrungen im WPF-Framework, wurde sich für dieses entschieden.

Als Grundlage für den Start des Projektes wurde das MVVM-Light<sup>8</sup> Toolkits verwendet. Es beinhaltet grundlegende Funktionalitäten und Templates, welche den Entwickler entlasten. Das Projekttemplate aus diesem Toolkit bringt ebenfalls die notwendige Struktur für das MVVM-Entwurfsmuster mit und ergänzt es um einen IoC-Container.

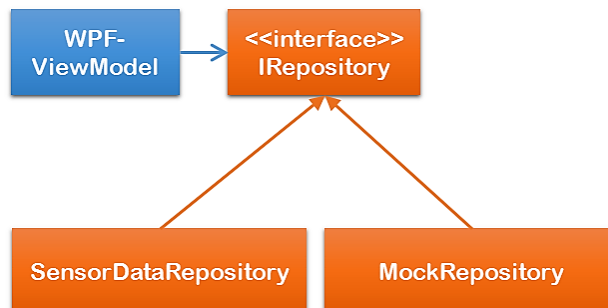
Der Datenzugriff wurde wie ebenfalls im Entwurf beschrieben mit dem Repository-Muster siehe Abbildung 5.4 realisiert. Dies ermöglichte zu Beginn des Projektes und für Test gegen eine Attrappe

<sup>6</sup>Link zum Projekt <http://pencil.evolus.vn/>

<sup>7</sup>Web Applikations Framework von Microsoft

<sup>8</sup>Link zum Projekt <https://mvvmlight.codeplex.com/>

des Kommunikationsmoduls zu entwickeln. Diese Attrappe täuschte das fehlende Modul vor und kann problemlos durch das fertig implementierte Modul ersetzt werden. Durch die vorangegangene Schnittstellen Definition mussten für die Integration des Kommunikationsmoduls keine Änderungen an der internen Systemlogik der GUI mehr vorgenommen werden.



**Bild 5.4:** Repository Muster in der GUI

Ein weiteres Problem, welches es zu lösen gab, waren die Steuerelemente für die Anzeige. Es folgte eine kurze Evaluation und Testphase für die möglichen Tachometer und Linienchart Anzeigen, leider mit dem Ergebnis, dass die fertigen Produkte entweder Kostenpflicht waren oder schlecht dokumentiert und nicht genügend Anpassbar für unseren Zweck waren.

Darauf hin erstellte der Herr Krupinski diese beiden aufwändigen Steuerelemente als Benutzerdefinierte WPF-Steuerelemente selbst, welche uns die geforderten Funktionalitäten und die notwendige Veränderbarkeit mitbrachten.

### Gauge Control

Das entwickelte "Gauge"<sup>9</sup> Control in Abbildung 5.5 besteht aus einer Skala(1) mit einem minimal und maximal Wert, einen Zeiger(2) und einem Label(3), welches den Aktuellen Wert anzeigt.



**Bild 5.5:** Tachometer Anzeige der GUI

### LineChart Control

In Abbildung 5.6 ist das "LineChart"<sup>10</sup> Control abgebildet. Es besteht aus einem anzeige Raster(1), einen Graphen(2) zur Visualisierung der Werte einer Anzeige für die Y-Achse(3) mit einstellbaren Werten und einer Anzeige für den ersten und letzten Wert der X-Achse(4).

<sup>9</sup>engl. Tachometer

<sup>10</sup>engl. Liniendiagramm

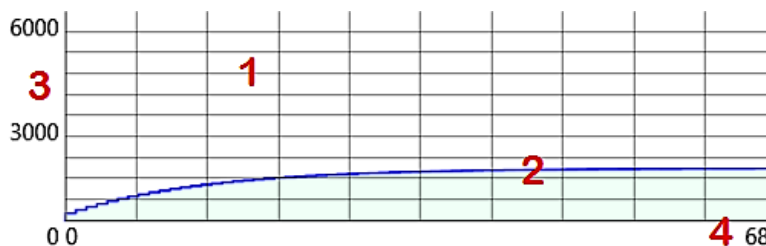


Bild 5.6: Liniendiagramm Anzeige der GUI

Diese Steuerelemente bieten dem Entwickler diverse Einstellmöglichkeiten in Form und Farbgebung und sind beliebig erweiterbar für kommende Anforderungen.

Die beiden Benutzer definierten Steuerelemente wurden zu einem eigenen Control zusammengefasst und mit weiteren Steuerelementen ergänzt. In Abbildung 5.7 ist exemplarisch ein komplettes Anzeigeelement für einen Sensorwert abgebildet.

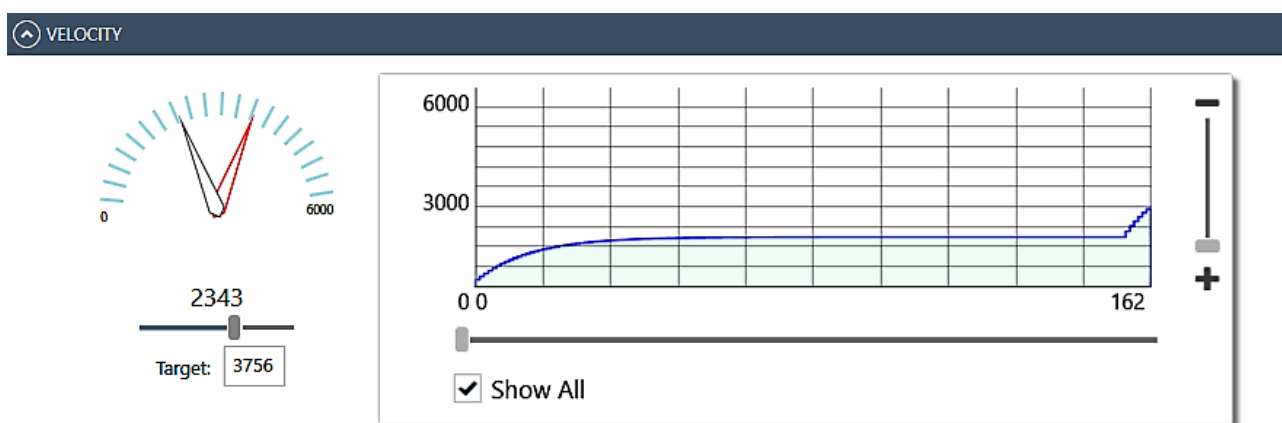


Bild 5.7: Datenanzeige Control

Die Abbildung 5.8 zeigt fertige die GUI zum Ende des Projektes. Sie ist aktuell in der Lage die definierten Sensorwerte anzuzeigen, die einzelnen Anzeigeelemente 5.7 bieten die Möglichkeiten im Liniendiagramm zu Zoomen und das Wertefenster der X-Achse festzulegen und zu verschieben. Für einstellbare Größen, wie zum Beispiel die Drehgeschwindigkeit kann ein Zielwert eingegeben werden, welcher durch eine zweite rote Nadel im Tachometer angezeigt wird. Im oberen Bereich der GUI kann man die Regelparameter ebenfalls verändern und an den Controller senden.

Zum Abschluss der Implementierungsphase wurden Unit-Tests geschrieben, wodurch noch einige Fehler aus der Codebasis aufgespürt werden konnten. Diese Test helfen ebenfalls zukünftigen Entwickler an dem Projekt, da unbedachte Änderungen welche zu Fehlverhalten der Applikation führen können schnell aufgedeckt werden.

## 5.4 Ausblick

Dieser Abschnitt beinhaltet mögliche Erweiterungen und nicht vollständig umgesetzte Projektteile, welche eine weitere Gruppe als Arbeitspaket aufnehmen kann.

**Multi-Plattformfähigkeit** Durch die Separierung der einzelnen Codeteile in eigenständige Projekte und die strikte Umsetzung eines Domain Driven Designs kann man die WPF GUI austauschbar machen und durch zum Beispiel eine Web-API ersetzen, welche anschließend mit einem Web-

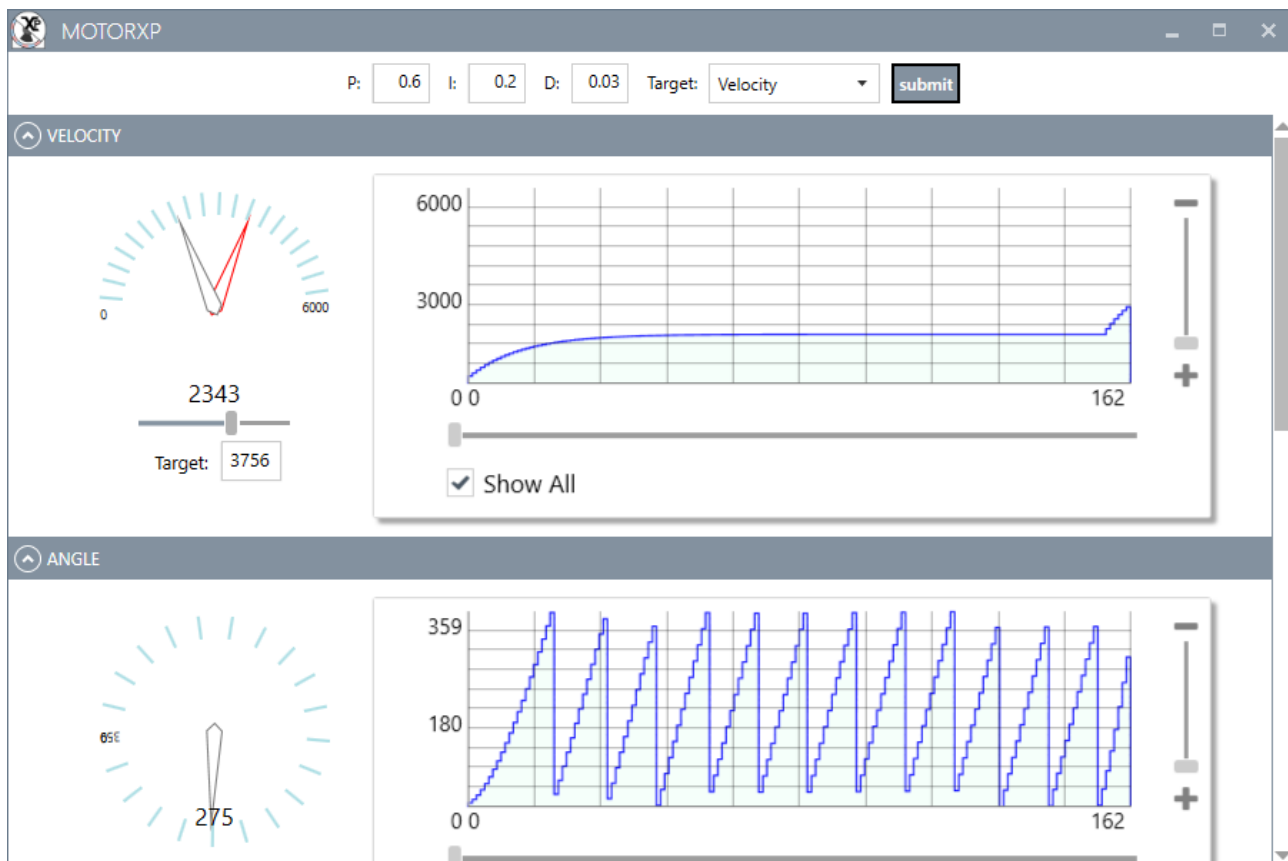


Bild 5.8: Ansicht der GUI zum Projektende

Frontend z.B. ASP.NET oder Angular2 kommuniziert. Auch eine Smartphone Applikation mit zum Beispiel Xamarin wäre möglich.

**MouseHover in LineChart** Diese Funktionalität konnte leider nicht Fertiggestellt werden, es wird noch nicht der aktuelle Wert unter der Maus angezeigt.

**Änderung des Layoutes** Durch eine Änderung des Layoutes wäre es möglich alle Sensorwerte auf einen Bildschirm Darzustellen, ohne die Notwendigkeit von Scrollbars zu haben. Empfehlung dafür wäre ein Wrappanel mit Horizontaler Ausrichtung.

**Hall Pattern Anzeige** Die Aktuelle Anzeige des Hall Pattern ist in einem extra Control unter dem ItemsTemplate für die Anzeigeelemente, dies verhindert Aktuell die einfache Umsetzung der vorher beschriebenen Änderung des Layouts. Hier müsste das Hall Pattern mit in das ItemsTemplate implementiert werden.

**Auswahl des Com-Ports** Eine wichtige Erweiterung wäre eine Einstellmöglichkeit für den Com-Port, da dieser aktuell Hardcoded ist und ggf. im Quelltext angepasst werden muss. Eine Notlösung wäre eine Auslagerung der Einstellung in eine Konfigurations-Datei. Aber die Bessere Lösung wäre eine ComboBox in der GUI zur Auswahl aus verfügbaren Ports.

**Mehrere Inputs** Die GUI könnte noch für mehrere Inputs erweitert werden um es zu ermöglichen mehrere Messstationen oder die Simulation nebeneinander Vergleichend laufen zu lassen.

**Benutzer Handbuch** Dem knappen Zeitrahmen geschuldet wurde noch kein Benutzerhandbuch erstellt. Dies wäre Sinnvoll nach der Änderung des Layouts zu erstellen, um aktuelle Bilder verwenden zu können.

**Weitere Tests** Sinnvoll für mehr stabilität wären weitere Tests, einmal für die CustomControls (Gauge und LineChart) Unittests, für die Oberfläche allgemein Coded-UI Tests und ein automatisierter Integrationstest für das Kommunikationsmodul.

**Import/Export** Eine Im-/Exportfunktion um Testläufe zu sichern und wiederholt zu laden.

**RingSpeicher** Implementierung eines Ringspeichers, welcher eine Maximale Anzahl an Elementen fasst und überschüssige Daten auf der Festplatte oder ähnlich Speichert, um ein Überlaufen des Arbeitsspeichers zu verhindern.



## Chapter 6

# Arbeitspaket Simulation

Für Analyse, Entwurf und Implementierung dieses Arbeitspakets übernahm Herr Welker die Verantwortung.

### 6.1 Analysephase I - Verwendung existierender Bausteine

Gemäß Projektspezifikation wurde gewünscht, dass die zu entwickelnde Simulation in Echtzeit laufen soll (1 Sekunde Simulation soll einer Sekunde in der Realität entsprechen), so dass die Simulationsergebnisse später ggf. mit dem realen Versuchsaufbau verglichen oder gar verbunden werden können.

Zunächst wurde hierzu der Einsatz eines *Rapid Prototyping System* von dSPACE erwogen. Das Problem hierbei ist jedoch die Beschaffung (Hardware) und der nach bereits früher erhaltenen Aussagen extrem hohe Preis.

Eine weitere Alternative wäre *Simulink Desktop Real-Time*. Hier wird zwar keine kostspielige externe Hardware benötigt, aber auch die Anschaffungskosten von 2000 Euro wären für den Simulationsteil des Projekts wesentlich zu hoch.

Aufgrund dieser Randbedingungen wurde von einer Simulation in Echtzeit zunächst wieder Abstand genommen.

Ein zusätzliches Ziel der Simulation sollte eine Kommunikations-Verbindung mit der GUI sein. Hierbei wäre es wünschenswert, wenn die Kommunikation äquivalent zur echten Hardware funktionieren würde, so dass mit ein- und derselben GUI sowohl Simulation als realer Versuchsaufbau angesprochen werden können.

Da im realen Aufbau die Kommunikation zwischen  $\mu$ -Controller und PC mithilfe der seriellen Schnittstelle umgesetzt werden würde, sollte auch in der Simulation die Kommunikations-Verbindung mittels serieller Schnittstelle realisiert werden.

Simulink stellt hierfür eigene Blöcke bereit, mit deren Hilfe es möglich ist, Modelle mit externen Geräten kommunizieren zu lassen.

Da in der Konzept-Phase nicht eindeutig ausgeschlossen werden konnte, dass Simulation und GUI ggf. auf demselben PC laufen würden, wurde eine Möglichkeit gesucht, dies mithilfe von virtuellen COM-Ports zu realisieren.

Hierzu wurde das Programm *Virtual Serial Port Driver* ausgewählt, mit dessen Hilfe es möglich ist, virtuelle COM-Ports zu erzeugen und diese virtuell zu verbinden. Damit wird es möglich, Simulation und GUI auf demselben PC laufen zu lassen und dieselben Schnittstellen zu verwenden wie bei Kommunikation mit externer realer Hardware.

Da die Simulation alle real verfügbaren Sensoren sowie das Verhalten des Motors möglichst gut abbilden soll, wurde weiterhin recherchiert, wie sich der reale Aufbau des Motor-Experimentierplatzes am besten in Simulink umsetzen ließ. Im Verlauf dieser Recherchen wurde festgestellt, dass es in Simulink bereits mehrere Modelle für einen solchen Aufbau (BLDC Motor mit Sensorik) gibt, die anhand ihrer Anpassbarkeit hinsichtlich der Motorparameter evaluiert wurden.

Dabei stellte sich heraus, dass das Modell *power\_pmmotor* am besten hinsichtlich der Motorparameter eingestellt werden kann.

Dieses Modell ist in Abbildung 6.1

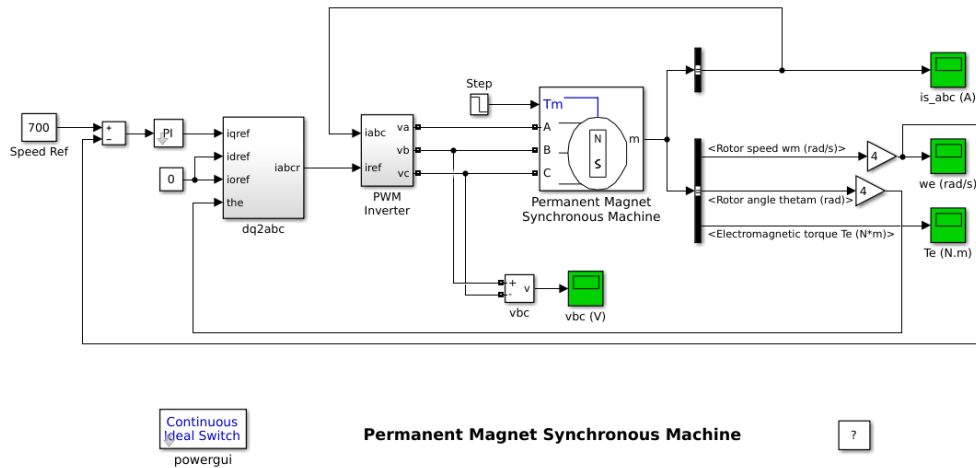


Bild 6.1: *power\_pmmotor* Simulink Modell

zu sehen. Hier ist zu erkennen, dass in diesem Modell sowohl der Motor als auch die Sensorik (Hall-Sensoren) bereits umgesetzt sind.

Das Modell bietet Einstellmöglichkeiten für folgende (in der Motorspezifikation angegebenen) Parameter des echten Motors sowie der Regelung:

- Anzahl Polpaare
- Anzahl der Spulen
- Spulen Ansteuerungsform (sinusförmig)
- Rotor Typ (rund)
- Parameter für Modus (Generator- oder Motor-Betrieb)
- Spulen-Widerstand ( $0,33\Omega$ )
- Drehmomentkonstante ( $0,065Nm/A$ )
- (Regelungsparameter): Motor-Regelung auf Drehzahl-Sollwert
- P- und I-Anteil des Drehzahl-Reglers

Das Modell berechnet zusätzlich zur Drehzahl-Regelung das Motor-Drehmoment, die Werte der Hallsensoren sowie die Ströme in den Spulen. Des weiteren führt das Modell auch die Kommutierung durch, welche benötigt wird um das Drehmoment auf einem konstanten Level zu halten.

Im ersten Simulations-Modell wird also Simulink mit dem Modell *power\_pmmotor* verwendet und per *Virtual Serial Port Driver* an eine GUI angebunden (Implementierung des Kommunikations-Protokolls im Simulations-Kontext fehlt noch, der externe Kommunikationspartner wurde im Rahmen weiterer Teilgebiete dieses Projekts erstellt).

## 6.2 Entwurfsphase und Implementierung I - Anpass-Arbeiten

Es wurde damit begonnen, das Modell an den echten/geplanten Versuchsaufbau anzupassen:

- Um sich möglichst viele Freiheiten bei der Drehzahlregelung offen halten zu können, wurde geplant, einen PID-Regler zu verwenden.
- Weiterhin wurden die zusätzlich benötigten Simulink-Blöcke für die serielle Kommunikation implementiert und passend parametrisiert. Da das Modell mit einem *Solver* mit variabler Schrittgröße berechnet wird (hierbei berechnet der Lösungsalgorithmus von Simulink selbst, zu welchen Zeitpunkten die nächste Berechnung durchgeführt werden muss), konnte die Kommunikation nicht direkt umgesetzt werden, da die Kommunikationsblöcke nur mit *Solvern* funktionieren die eine feste Schrittgröße (festes Abtastraster) verwenden. Um dieses Problem zu lösen wurde hierfür ein *Sample and Hold* Block eingesetzt. Dieser Block behält den Wert am Eingang so lange gültig, bis ein neuer Wert am Eingang anliegt, gibt am Ausgang jedoch zu definierten Zeiten einen Wert aus. Hiermit war es möglich zyklisch Daten aus dem Simulink Modell heraus zu senden und zu empfangen.
- Anschließend wurde äquivalent zum realen Versuchs-Aufbau begonnen Protobuff zu implementieren, da die GUI-Schnittstelle mit diesem Protokoll kommuniziert. Diese Arbeiten wurde dann zugunsten eines alternativen Motormodells (s.u.) allerdings auf Eis gelegt.

## 6.3 Analysephase II - Alternativ-Modell

Da ohne den echten Aufbau nicht weiter getestet werden konnte, ob das in der ersten Entwurfsphase ausgewählte Matlab Modell den Aufbau bereits ausreichend genau beschreiben kann und zum anderen die Komplexität dieses Modells sehr hoch ist, wurde beschlossen, ein neues, eigenes und nach Möglichkeit einfacheres Alternativ-Modell zu erstellen.

## 6.4 Entwurfsphase II - eigene Motor-Modellierung

Das Alternativ-Modell für den Motor sollte folgende Form haben - siehe Abbildung 6.2:

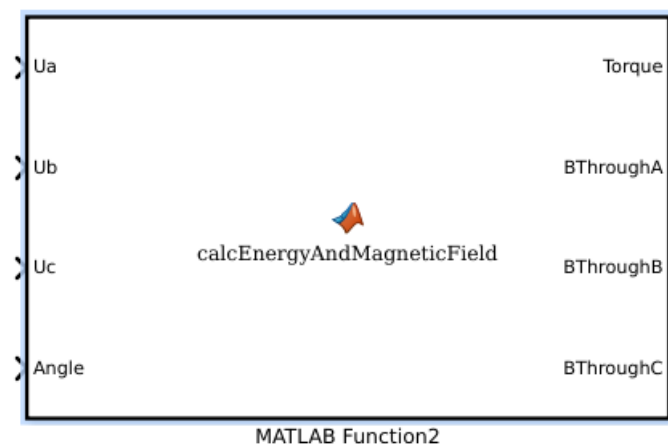


Bild 6.2: Eigenschaften neuer Simulink Motor Block

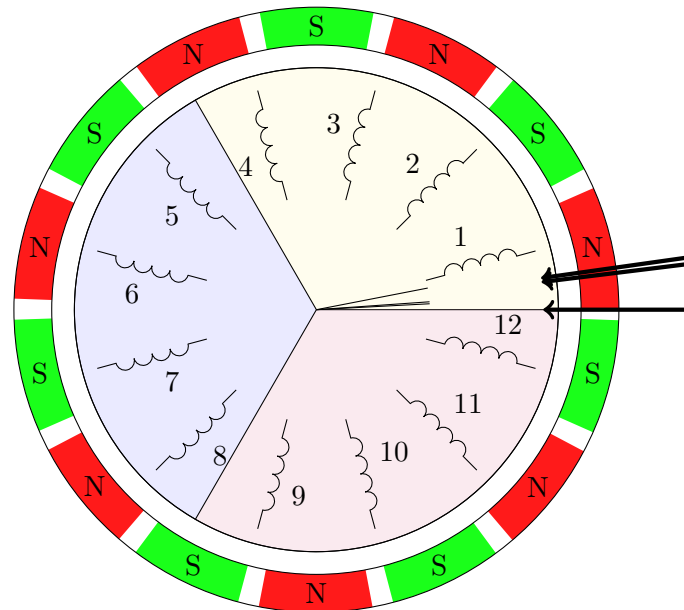
Hierbei sind die Eingangsgrößen zu erkennen

- $U_a, U_b, U_c$  an den drei Phasen U/V/W (an Spulen anliegende Spannung)
- Angle (Drehwinkel)

sowie die Ausgangsgrößen

- $B_{ThroughA}, B_{ThroughB}$  und  $B_{ThroughC}$  (Spulen durchdringendes Magnetfeld)
- $Torque$  (Drehmoment)

Der Motor ist wie in Abbildung 6.3 zu sehen ist aufgebaut:



**Bild 6.3:** Motor Aufbau

Hierbei sind die 14 Permanentmagneten im außen laufenden Rotor zu erkennen. Diese sind so angeordnet, dass sich ihre Polung kontinuierlich abwechseln.

Im Inneren des Motors sind die einzelnen Spulen des Stators zu erkennen. Mithilfe der farbigen Zuordnung ist es möglich zu identifizieren welche Spulen zu einer großen Spule (U,V oder W) zusammengeschlossen sind.

Es ist zu erkennen, dass die großen Spulen sternförmig (über die Teil-Spulen 4, 5 und 12) zusammengeschaltet sind (der Sternpunkt des Motors ist von außen nicht zugänglich). Die U/V/W-Anschlüsse sind mithilfe der Pfeile markiert (vgl. Spule 1, 8 und 9). Es sollte noch erwähnt werden, dass die Spulen unterschiedlich gewickelt sind. So sind die Spulen 1, 3, 6, 8, 9 und 11 im Uhrzeigersinn gewickelt und die Spulen 2, 4, 5, 7, 10 und 12 entgegen diesem.

In der symbolischen Abbildung ist zu erkennen, dass sowohl zwischen den Permanentmagneten im Rotor, als auch zwischen den Spulen im Stator jeweils Abstände existieren. Im realen Motor ist außerdem ein Luftspalt zwischen dem Rotor und dem Stator vorhanden. Die Formgebung der Polschuhe wird nicht dargestellt.

Vereinfachungs-Annahmen: Da ein einfaches Modell erstellt werden sollte, wurden folgende Annahmen getroffen, vgl. auch Abbildung 6.4:

- Zwei der drei Phasen werden in der Darstellung mit Spannung versorgt, die dritte Spule ist offen.

- Alle Luftspalte werden vernachlässigt. D.h. weder die räumlichen Abstände zwischen den Permanentmagneten im Rotor noch der räumliche Verlauf der von den Spulen in Stator erzeugten Magnetfelder werden modelliert.
- Weiterhin wird insbesondere auch der Luftspalt zwischen Rotor und Stator vernachlässigt.
- Es werden nur senkrechte Magnetfeldlinien angenommen, der Einfluss der Polschuhe wird vernachlässigt.
- Die Zuordnung der Spulen zu den großen Spulen, die Verbindung mit den Zuleitungen usw. entspricht der vorhergehenden Abbildung.

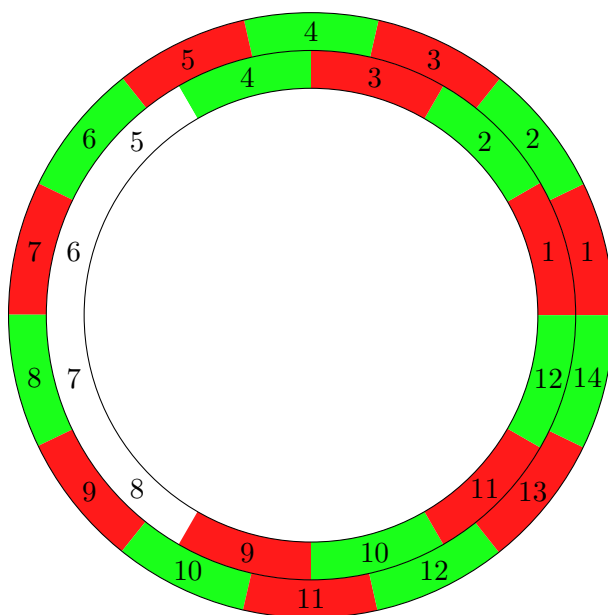


Bild 6.4: Vereinfachtes Modell

Um dieses Modell mithilfe von Arrays in Simulink umsetzen zu können, wurde gedanklich zwischen 1 und 12/14 aufgeschnitten und der Kreis ausgerollt. Dabei ergibt sich dann folgendes Bild 6.5.

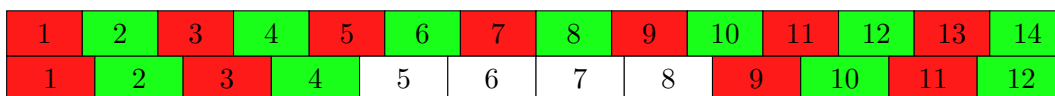


Bild 6.5: Abgerolltes Modell

## 6.5 Implementierung II - neues Motormodell

Für die genaue Modellierung wurde folgender Ansatz verwendet: Sowohl für den Stator als auch den Rotor wird ein Array angelegt. Beim Rotor (Permanentmagnete) werden die Einträge mithilfe einer Konstanten für die magnetische Flussdichte gefüllt, dabei bekommt jeder Nordpol willkürlich gewählt ein positives und jeder Südpol ein negatives Vorzeichen.

Da (wie im Bild zu erkennen ist) der Rotor 14 und der Stator zwölf Einheiten besitzen, wird die Anzahl der Arrayeinträge auf (mindestens) das kleinste gemeinsame Vielfache (84) festgelegt. Somit

ist es mithilfe des Modells möglich den Motor simuliert in 84 Stufen zu drehen was einer Schrittgröße von ca.  $4,3^\circ$  entspricht.

Für das Array des Stators wird berücksichtigt, welche Spulen aktuell bestromt sind und wie groß die angelegte Spannung ist. Mit Hilfe der Spannung und dem Spulen-Widerstand kann eine statische Stromstärke berechnet werden (die Induktivität der Spulen wird zunächst vernachlässigt), daraus lässt sich die magnetische Flussdichte in den Spulen bestimmen.

Abhängig vom Winkel und der angelegten Spannung (d.h. welche Spulen bestromt sind) wird das Array für den Stator entsprechend befüllt.

Anschließend wird abhängig vom Winkel das Rotor Array rotiert.

Das resultierende Magnetfeld wird bestimmt, indem diejenigen Array-Einträge addiert werden, die sich einander gegenüber befinden. Die Einträge des so entstanden resultierenden Arrays werden quadriert (und durch die Konstante  $2 * \mu_0$  geteilt).

Um am Ende die Energie zu errechnen, welche zu diesem Zeitpunkt im resultierenden Magnetfeld steckt, wird über die einzelnen Array-Einträge integriert. Somit ergibt sich ein Energiewert für diesen Zustand bzw. Zeitpunkt.

Um das resultierende Drehmoment zu erhalten wird dieselbe Rechnung erneut durchgeführt, allerdings für den Fall, dass der Rotor  $1/84$  weiter gedreht wird.

Aus der Differenz dieser beiden Energiewerte kann das Drehmoment bestimmt werden unter der Annahme, dass die ganze Energie-Änderung in die Beschleunigung des Rotors eingeht.

Zusätzlich werden die resultierenden Magnetfelder über den Spulen (U,V und W) ebenfalls aufintegriert, um mit Hilfe dieser Werte im weiteren Modellaufbau die Induktionsspannung berechnen zu können.

## 6.6 Bewertung des neuen Modells

Bei der Untersuchung des neuen Modells muss festgehalten werden, dass die Induktivität der Spulen nicht mithilfe dieses Modells berechnet werden kann. Der modellierte Magnetfeld-Verlauf für eine Spule hat die in Abbildung 6.6 dargestellte Form.

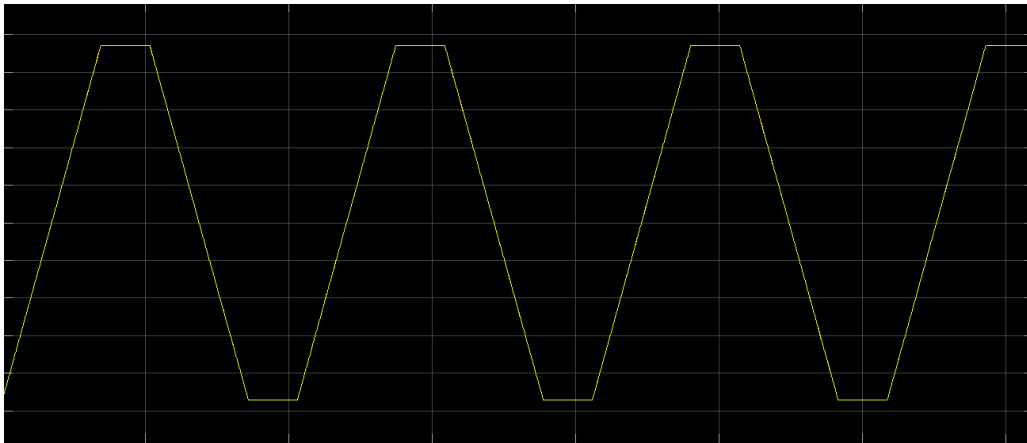


Bild 6.6: Verlauf Magnetfeld in einer Spule

Dies lässt sich dadurch begründen, dass bei der Betrachtung einer Spule (z.B. der V-Spule) für das Feld im Stator  $4*7 = 28$  Array-Elemente des resultierenden Arrays beteiligt sind. Über diese wird integriert, um die Größe für das Feld zu bekommen, was einer Aufsummierung der beteiligten Array-Elemente entspricht  $\sum_{i=1}^{28} V_{resi}$ .

Wobei sich ein Eintrag  $V_{resi}$  aus der Addition des Spuleneintrags  $V_i$  und des Eintrags des darüber

liegenden  $R_i$  also dem Wert des Rotormagnetfeldes an dieser Stelle ergibt. So lässt sich  $\sum_1^{28} V_{res_i}$  zu  $\sum_1^{28} (V_i + R_i)$  umstellen. Diese Summe lässt sich nun allerdings nun in zwei Summen zerlegen, zum einen in  $\sum_1^{28} V_i$  und zum anderen  $\sum_1^{28} R_i$ .

Hierbei ist die erste Summe  $\sum_1^{28} V_i$  immer 0, da die vier Spulen einer großen Spule (im Beispiel Phase V) in der Wicklungsrichtung abwechseln.

Somit bleibt nur noch  $\sum_1^{28} R_i$  übrig um den Verlauf zu erklären: Wie in Abbildung 6.5 zu erkennen ist, ist ein Element des Rotors kleiner als eines des Stators. So kürzen sich zwar auch viele Elemente des Rotors heraus, jedoch nicht alle. Dies führt zum erkennbaren Magnetfeld-Verlauf.

Das hat jedoch nicht wirklich etwas mit dem Verlauf zu tun, der modelliert werden soll. Aus diesem Grund ist das Modell hierfür nicht passend und sollte nicht ohne weitere Änderungen verwendet werden (insbesondere bedeutet eine ausschließlich statische Betrachtung der Magnetfeldverläufe eine unzulässige Vereinfachung).

Hingegen scheint der Verlauf für das Drehmoment durchaus Sinn zu ergeben, wie in Abbildung 6.7 zu erkennen ist: Da in diesem Modell in die Berechnung der resultierenden Energie ein Quadrat eingeht, ist hier das zuvor beschriebene Verhalten des Wegkürzens nicht mehr vorhanden.

Es ist jedoch noch darauf hinzuweisen, dass eine Vergrößerung der Anzahl an Array-Elementen (entspricht einer Erhöhung der Auflösung) nicht dazu führt, dass dieser Verlauf glatter werden würde, sondern es werden immer dieselben zwölf Stufen, welche den Spulen-Polpaaren entsprechen, im Verlauf zu erkennen sein. Dies ist eine Folge der zu Beginn getroffenen Vereinfachungen, wonach Luftspalt, Polform und nicht-senkrechte Magnetfeldlinien vernachlässigt wurden.

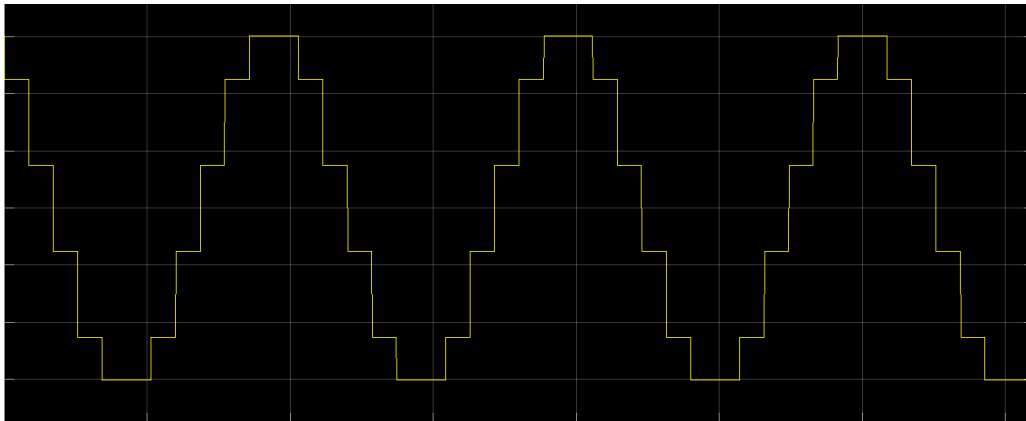


Bild 6.7: Drehmoment bei Spannung an  $U_a$  und Masse bei  $U_b$

## 6.7 Bewertung der beiden Modelle

Für eine Simulation des Motor-Experimentierplatzes ist das zweite Modell (zumindest in der vorliegenden Form) nicht geeignet, es geht von statischen Zuständen aus und vernachlässigt die dominanten Parameter (Induktivität, Luftspaltgestaltung usw.) eines realen Motors.

Ein Vergleich des ersten Modells mit dem Verhalten des realen Experimentierplatzes steht noch aus.

## 6.8 Ausblick

Für das neue, zweite Modell gibt es noch viel Verbesserungs-Potenzial, z.B.:

- So wäre es möglich, bzw. wurde schon damit begonnen, die Kommutierung umzusetzen. Diese soll anhand des Winkels die Spulen unterschiedlich an Masse und Spannung anschließen mit dem Ziel ein konstantes Drehmoment zu erhalten.
- Mit dieser Erweiterung könnte dann ein (Strom-/Drehmoment-)Regler implementiert werden welcher z.B. auf ein konstantes Drehmoment regelt.
- Das reale Drehmoment ist eine Funktion zahlreicher mechanischer und elektromagnetischer Details der Motorgestaltung. Einfluss der Induktivität, Luftspaltgestaltung usw. könnten Gegenstand weiterer Modell-Verfeinerungen sein.
- Um das neue Modell in die Simulationsumgebung anstelle des ersten Modells einbetten zu können, sind dieselben Erweiterungen durchzuführen, mit denen bereits am ursprünglichen Modell begonnen wurde (z.B. damit auch dieses Modell in der Lage wäre mit der GUI zu kommunizieren, um simulierte Größen auszutauschen bzw. von dort Regelgrößen zu bekommen).

Zuvor sollte das erste Modell (Matlab-Bausteine) weiter mit dem Verhalten des realen Motor-Experimentierplatzes verglichen werden (Messwerte).

Da der reale Aufbau noch nicht fertiggestellt war, konnte hierfür noch keine eindeutige Modell-Bewertung gegeben werden.



# List of Figures

1.1	Ordner Struktur Projekt . . . . .	3
2.1	Schematischer Aufbau eines Frames . . . . .	6
2.2	wesentliche Klassen der implementierten Bibliothek für die GUI . . . . .	8
2.3	Implementierte Funktionen des Kommunikationsmoduls auf dem $\mu$ -Controller . . . . .	10
2.4	Grundlegende Einstellungen der UART-App: Diese müssen mit den Einstellungen der PC-Bibliothek übereinstimmen. . . . .	13
2.5	Weiterführende Einstellungen der UART-App: Beide Richtungen werden per DMA gehandhabt. . . . .	13
2.6	Interrupt Einstellungen der UART-App: Hier werden die Callback-Funktionen angegeben. . . . .	13
2.7	GPIO Einstellungen der UART-App . . . . .	13
2.8	Einstellungen der CRC-App: Auch hier müssen die Einstellungen mit denen der PC-Bibliothek übereinstimmen. . . . .	13
2.9	Zuordnung der GPIO-Pins unter "DAVE -> Manual Pin Allocator"; Die Pins P1.4 und P1.5 sind mit dem Debug-Chip verbunden. Siehe (boa16, Table 5) . . . . .	13
2.10	Inhalt eines falsch interpretierten Bearbeitungspuffers . . . . .	14
2.11	Verbessertes Nachrichtendesign mit Start of Frame . . . . .	14
2.12	Zustandsautomat zur Erkennung des Start of Frame . . . . .	14
3.1	Projektdialog DAVE . . . . .	17
3.2	Buildprozess starten . . . . .	18
3.3	Debug-Konfiguration erstellen . . . . .	18
3.4	POSIF Übersicht . . . . .	19
3.5	Hall Sensoren und Pattern . . . . .	21
3.6	Eingelesenes Hall-Pattern . . . . .	22
3.7	Hall-Pattern Lookup-Tables . . . . .	23
3.8	Inkrementalgeber Phasen . . . . .	23
3.9	Spannungsteiler zum einlesen des NTC-Widerstands . . . . .	24
3.10	ADC Einheit der XMC Controller . . . . .	25
5.1	Grober Architektur Entwurf . . . . .	30
5.2	MVVM-Konzept . . . . .	30
5.3	Erster Layout Entwurf der GUI . . . . .	31
5.4	Repository Muster in der GUI . . . . .	32
5.5	Tachometer Anzeige der GUI . . . . .	32
5.6	Liniendiagramm Anzeige der GUI . . . . .	33
5.7	Datenanzeige Control . . . . .	33
5.8	Ansicht der GUI zum Projektende . . . . .	34
6.1	power_pmmotor Simulink Modell . . . . .	37
6.2	Eigenschaften neuer Simulink Motor Block . . . . .	38

---

6.3	Motor Aufbau . . . . .	39
6.4	Vereinfachtes Modell . . . . .	40
6.5	Abgerolltes Modell . . . . .	40
6.6	Verlauf Magnetfeld in einer Spule . . . . .	41
6.7	Drehmoment bei Spannung an $U_a$ und Masse bei $U_b$ . . . . .	42

# List of Tables

## Anhang A

# Codeausschnitt aus der C#-Bibliothek mit ComPort-Settings

```
1 private void port_Init()
2 {
3     try
4     {
5         _uart.BaudRate = 9600;
6         _uart.DataBits = 8;
7         _uart.StopBits = StopBits.One;
8         _uart.Handshake = Handshake.None;
9         _uart.Parity = Parity.Even;
10        _uart.DtrEnable = false;
11        _uart.RtsEnable = false;
12        _uart.DataReceived += uart_DataReceived;
13        _uart.ReceivedBytesThreshold = 10;
14        _uart.Open();
15        _isInit = true;
16    }
17    catch (Exception e)
18    {
19        throw new SystemException("Uart init failed! Message: " + e.Message);
20    }
21 }
```

## Anhang B

# Codeausschnitt aus der C#-Bibliothek zum Senden der Bytes

```
1  buf = Frameparser.EncapsuleFrame(buf);
2  try
3  {
4      // ReSharper disable once UnusedVariable
5      foreach (var elem in buf)
6      {
7          _uart.Write(buf, i, 1);
8          i++;
9          System.Threading.Thread.Sleep(30);
10     }
11     return true;
12 }
13 catch (Exception e)
14 {
15     throw new SystemException("Could not send Params! Error: " + e.Message);
16 }
```

## Anhang C

# Codeausschnitt aus der C#-Bibliothek mit dem SoF-Automaten

```
1 //Start of Frame checker
2 var state = 0; //state 0: no sof; state 1: 0x55 detected; state 2: 0x55 and 0xD5 d
3 do{
4     spL.Read(SofDel, 0, 1);
5     if (state == 0 && SofDel[0] == 0x55)
6     {
7         state = 1;
8         continue;
9     }
10    if(state == 1 && SofDel[0] == 0x55)
11    {
12        state = 1;
13        continue;
14    }
15    if (state == 1 && SofDel[0] == 0xD5)
16    {
17        state = 2;
18        continue;
19    }
20    if (SofDel[0] != 0x55 || SofDel[0] != 0xD5)
21    {
22        state = 0;
23        continue;
24    }
25 }while(state != 2);
26 state = 0;
27 //End of Frame Checker
```

# Quellen

- [boa16] *Board User Manual XMC4700 XMC4800 Relax Kit Series. : Board User Manual XMC4700 XMC4800 Relax Kit Series*, jun 2016. – zuletzt aufgerufen am 28.12.2016
- [crc06] *CRC 16 CCITT in C#*. [http://sanity-free.org/133/crc\\_16\\_ccitt\\_in\\_csharp.html](http://sanity-free.org/133/crc_16_ccitt_in_csharp.html), November 2006. – zuletzt aufgerufen am 24.12.2016
- [hdl07] *Implementing The CCITT Cyclical Redundancy Check*. <http://www.drdoobbs.com/implementing-the-ccitt-cyclical-redundan/199904926>, Juni 2007