Smart Mechatronics

Hochschule Bochum
Bochum University
of Applied Sciences

Masterarbeit

zum Erlangen des akademischen Grades
Master of Science (M.Sc.)

Modellbasierte Entwicklung und Verifikation mit MATLAB/Simulink eines modularen Frequenzumrichters

Autor: Baris Akgül
baris.akguel@stud.hs-bochum.de
Matrikelnummer: 018352539

Erstgutachter: Prof. Dr.-Ing. Arno Bergmann
Zweitgutachter: M.Sc. Christoph Peters

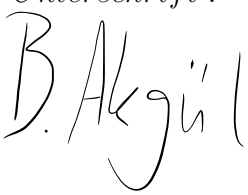
Abgabedatum: 18.12.2023

Eidesstattliche Erklärung

Eidesstattliche Erklärung zur Masterarbeit:
»**Modellbasierte Entwicklung und Verifikation mit MATLAB/Simulink eines modularen Frequenzumrichters**«

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Ort, *Datum* :
Bochum, 18.12.2023

Unterschrift :


Danksagung

An dieser Stelle möchte ich allen danken, die mich im Rahmen meiner Masterarbeit begleitet haben.

Ganz besonders möchte ich Herrn Prof. Dr. Arno Bergmann und Christoph Peters von der Smart Mechatronics GmbH für die persönliche und fachliche Unterstützung danken, die mich in jeglichen Bereichen dieser Masterarbeit weitergebracht hat.

Ein besonderer Dank geht auch an Sean Dalton von der Smart Mechatronics GmbH. Die intensiven Debug-Sitzungen haben einen großen Beitrag geleistet.

Zum Schluss möchte ich Enes Darici für die hervorragende Kooperation und Zusammenarbeit im CarPediem-Projekt danken.

Inhaltsverzeichnis

Abkürzungsverzeichnis

ACQPS	Acquisition Prescaler
ADC	Analog-to-Digital Converter
CAN	Controller Area Network
CEVT	Capture Event
CM	Connectivity Manager
CMP	Counter Compare
CPU	Central Processing Unit
eCAP	Enhanced Capture
ePWM	Enhanced Pulse Width Modulation
GND	Ground
GPIO	General Purpose Input/Output
HTTP	Hypertext Transfer Protocol
IPC	Inter-Process Communication
ISR	Interrupt Service Routine
LAN	Local Area Network
LED	Light-Emitting Diode
LSB	Least Significant Bit
MCDC	Modified Condition/ Decision Coverage

MDIO Management Data Input / Output

MIL Model-in-the-loop

MOSFET Metal Oxide Semiconductor Field Effect Transistor

MSB Most Significant Bit

OSI Open System Interconnection

PHY Ethernet physical layer

RJ45 Registered Jack-45

SYSCLK System Clock

TCP Transmission Control Protocol

TSCTR Time-Stamp Counter

UDP User Datagram Protocol

Symbolverzeichnis

Symbol	Bedeutung	Einheit
C	elektrische Kapazität	F
$GVDD$	Gate Driver Supply Voltage	V
$I_{\mathrm{d,q}}$	Statorstrom-Komponenten im D,Q - Koordinatensystem	A
$PVDD$	Power Supply Voltage	V
R	elektrischer Widerstand	Ω
rpm	revolutions per minute	min^{-1}

1 Einleitung

Die vorliegende Masterarbeit wurde in Kooperation mit der Firma Smart Mechatronics GmbH am Institut für Systemtechnik der Hochschule Bochum angefertigt. Es handelt sich um die dritte Generation eines am Institut für Systemtechnik entwickelten Frequenzumrichters. [1, 2]

Dieser Masterarbeit ging eine Projektarbeit voraus, in der die Rahmenbedingungen in Form einer Anforderungserhebung festgelegt wurden [3].

1.1 Motivation und Aufgabenstellung

Die Entwicklung des Frequenzumrichters ist in zwei separate Masterarbeiten aufgeteilt. Eine beschäftigt sich mit der Hardwareentwicklung ("Entwicklung und Implementierung der Hardware eines modularen Frequenzumrichters" [4]), während diese den modellbasierten Teil des Frequenzumrichters mit dem Entwicklungstools Simulink entwickelt und auf der Zielhardware [4] verifiziert. Zudem werden die erstellten Systemmodellanforderungen verifiziert. Hierzu wird der entwickelte External Mode verwendet.

Die modellbasierte Entwicklung bietet bedeutende Vorteile im Vergleich zur konventionellen C-Code-Entwicklung, darunter verkürzte Entwicklungszeiten, Kostenreduktion und verbesserte Produktqualität [5].

Die Ableitung der Anforderungen aus dem Lastenheft (siehe Anhang A.1.3) für das Implementierungsmodell (siehe Anhang A.1.6) markiert den Beginn. Ein Verifikationsplan wird erstellt, um das Verhalten sämtlicher Komponenten zu gewährleisten.

Zunächst wird eine Software-Architektur erstellt, bevor mit der Implementierung in Simulink begonnen wird. Wichtig hierbei ist es, die vollen Vorteile des modellbasierten Entwickelns auszuschöpfen und schon frühzeitig Komponententests mit der Hilfe von Simulink Test durchzuführen.

In vorherigen Entwicklungen [6] stellte die langsame Echtzeit-Kommunikation zum Target *(deutsch: Zielgerät)* ein Problem dar. Daher wird der External Mode über Ethernet entwickelt, um den sogenannten Flaschenhals-Effekt *(engl.: bottleneck-effect)* zu umgehen.

Der External Mode dient dazu, Echtzeit-Kommunikation zu ermöglichen und zu überprüfen, ob das Zielgerät während der Laufzeit die korrekten Signalberechnungen für die Leistungsstufe durchführt. Für dieses Ziel wird entweder TCP[1] oder UDP[2] verwendet [7].

1.2 Gliederung der Arbeit

Die vorliegende Arbeit beinhaltet folgende Themen:

1. Die theoretischen Grundlagen der verwendeten Module und Methoden
2. Erstellung eines Lastenhefts auf Systemebene und abgeleitete Systemmodellanforderungen
3. Beschreibung und Vorstellung der erarbeiteten Software-Architektur
4. Erklärung und Beschreibung der Implementierung des Regelalgorithmus und Konfiguration des External Modes
5. Der Aufbau, die Durchführung und Auswertung der modellbasierten Tests
6. Verifikation der Systemmodellanforderungen und des External Modes
7. Fazit und Ausblicke

[1] **T**ransmission **C**ontrol **P**rotocol
[2] **U**ser **D**atagram **P**rotocol

2 Theoretische Grundlagen

In diesem Kapitel werden die Grundlagen zu den folgenden Themen vorgestellt:

1. Analog-to-Digital Converter und das *Acquisition Window*
2. Enhanced Capture Module und relevante Register
3. Vektorregelung
4. Softwaretests
5. Die fünf Schichten des OSI-Modells
6. Interrupt Service Routine vs. Polling-Ansatz

2.1 Analog-to-Digital Converter

Im vorliegenden Kapitel wird nur grob auf die Funktionsweise eines ADCs[1] eingegangen und das *Acquisition Window* vorgestellt.

Ein ADC führt eine analoge Signalmessung in einer Abfolge diskreter (digitaler) Messungen durch [8, 9]. Idealerweise wird das analoge Signal perfekt im diskreten Messmoment erfasst. Stattdessen muss die Stärke des analogen Signals über einen gewissen Zeitraum aufgebaut werden. Hier spielt das *Acquisition Window* eine wichtige Rolle. Die Breite dieses Fensters muss ausreichend lang sein, um die volle Stärke des Signals aufzubauen [9].

[1] **A**nalog-to-**D**igital **C**onverter

Der Prozess bei dem das Signal seine volle Stärke aufbaut, entspricht dem Funktionsprinzip eines RC-Schaltkreises. In Abbildung 2.1 ist dieser Schaltkreis dargestellt. Durch einen Schalter, der in bestimmten Zeitabständen geöffnet und geschlossen wird, lädt sich ein Kondensator auf. Bis der Kondensator die Eingangsspannung erreicht, benötigt es einen bestimmten Zeitraum, vergleichbar mit dem *Acquisition Window*.

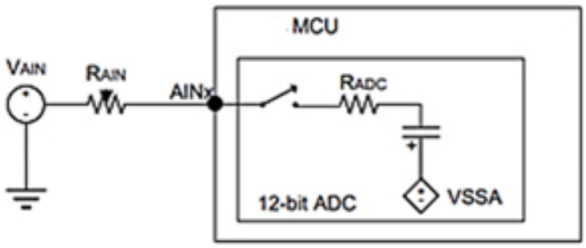


Abbildung 2.1: Übersicht eines RC-Schaltkreises [10]

In Abbildung 2.2 ist das analoge Signal und die Breite des *Acquisition Windows* zu sehen. Nach Ablauf dieser Zeit erfolgt die ADC-Messung.

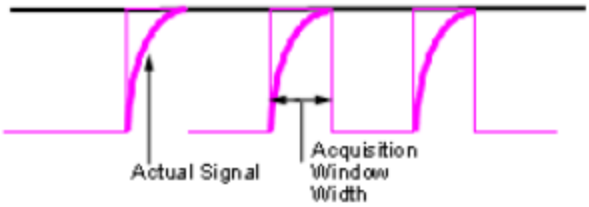


Abbildung 2.2: Übersicht einer digitalen Messung an diskreten Punkten mit *Acquisition Window* [9]

Im Folgenden werden die Auswirkungen beschrieben, wenn das *Acquisition Window* zu kurz oder zu lang ist [9].

- *Acquisition Window* zu kurz:

  Das Signal erreicht nicht seine volle Stärke, was zu einer unvollständigen oder fehlerhaften Messung führt.

- *Acquisition Window* zu lang:

Die Signalquelle kann sich während des Messvorgangs verändern, und die Abtastung könnte unregelmäßig sein, sodass der tatsächliche Wert nicht korrekt wiedergegeben wird. Auch hierdurch kann eine fehlerhafte Messung resultieren.

2.2 Enhanced Capture Module

In diesem Abschnitt wird keine ausführliche Erklärung zur Funktionsweise der verwendeten eCAP[2]-Module gegeben, sondern lediglich ein Überblick vermittelt. Zudem wird das *TSCTR*[3]-Register beschrieben, welches eine wesentliche Rolle bei der Implementierung gespielt hat.

Gemäß des Blockdiagramms im Datenblatt [11] gibt es vier Polaritätsprüfungen, welche die Änderung der Flanken zählen. Die Polaritätsprüfung zweiter Ordnung, bei dem eine steigende Flanke gefolgt von einer fallenden Flanke jeweils ein Event auslöst, ist in dieser Arbeit relevant.

Das Register *TSCTR* ist ein Zähler, welcher in dieser Arbeit verwendet wird, um die Geschwindigkeit des Motors zu berechnen. Jeder Flankenwechsel erzeugt ein Interrupt. Hierbei wird der Zähler im jeweiligen eCAP Register *TSCTR* gespeichert [11].

In Abbildung 2.3 wird eine Darstellung der Capture-Sequenz für einen Delta-Mode Time-Stamp mit steigender und fallender Flankendetektion gezeigt. Für das Verständnis ist hier hauptsächlich der obere Teil der Abbildung relevant. Ungeachtet dessen, dass es sich um einen Polaritätscheck 4. Ordnung handelt, kann dennoch der Zweck des *TSCTR*-Registers verstanden werden. Die empfangenen Flanken gelangen an die CAPx-Pins. Bei jedem Flankenwechsel wird ein Interrupt mit *CEVTx*[4] ausgelöst. Der *TSCTR* (hier: CTR[0-31]) zählt hoch, bis ein Event ausgelöst wird. Nach der Ereignisauslösung wird der *TSCTR* zurückgesetzt.

[2] **E**nhanced **C**apture
[3] **T**ime-**S**tamp **C**oun**t**e**r**
[4] **C**apture **Ev**en**t**

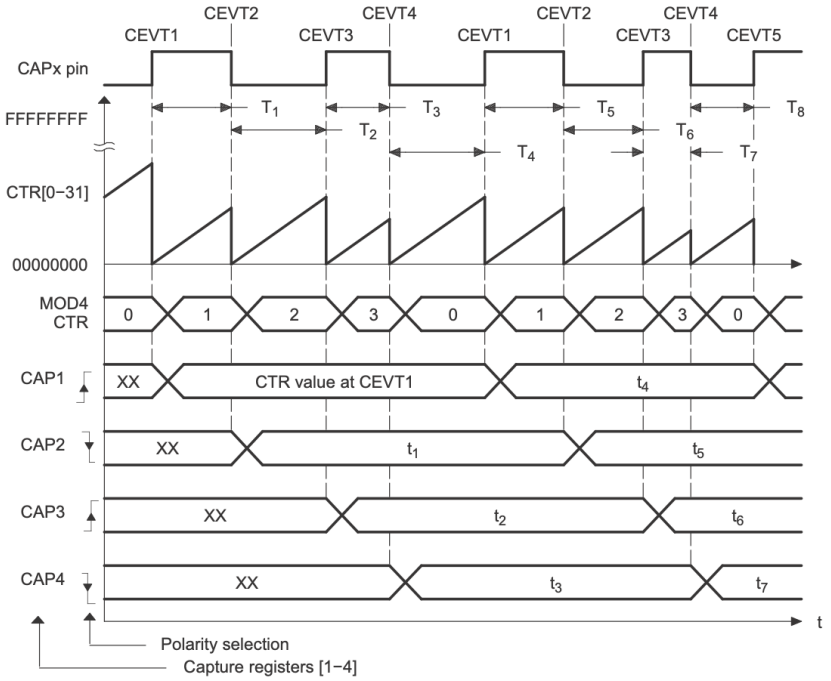


Abbildung 2.3: Capture Sequenz für einen Delta Mode Time-stamp mit steigender und fallender Flankenwechseldetektion [11]

2.3 Vektorregelung

Die Feldorientierte Regelung *(engl.:Field Oriented Control)*, auch als Vektorregelung bekannt, ist eine gängige Methode zur Regelung für Wechselstrommotoren. Die Vektorregelung wird für die Steuerung des Drehmoments verwendet. Vorteile der Vektorregelung sind eine sanfte Betriebsweise und schnelle Reaktion auf Veränderungen [12].

Das maximale Drehmoment entsteht, wenn der Winkel zwischen dem Stator- und dem Rotor-Magnetfeld 90 Grad beträgt. Die Feldorientierte Regelung passt kontinuierlich das Stator-Magnetfeld an, um es rechtwinklig zum Rotor-Magnetfeld auszurichten, daher der Name Feldorientierte Regelung [13].

Mindestens zwei der drei Strangstromwerte werden als Eingangssignale benötigt. Mithilfe der Kirchhoffschen Knotenregel kann der fehlende dritte Strangstrom berechnet werden [14].

Zusätzlich zur Strangstrominformation wird auch die aktuelle Rotorlage benötigt, um

die Feldorientierte Regelung anzuwenden.

Für detailliertere Informationen, wie z. B. die Transformation in das rotorfeste Koordinatensystem, wird auf die Literaturen [12, 13, 15] verwiesen.

2.4 Softwaretests

An dieser Stelle wird rund um das Thema Softwaretests auf [2] verwiesen.

In [2] werden unter anderem White-/ und Blackbox-Testverfahren behandelt. Dabei ist bei Whitebox-Verfahren das innere Verhalten der Komponente bekannt. Bei Blackbox-Verfahren ist der innere Aufbau unbekannt und es werden nur die hineingehenden/herausgehenden Signale betrachtet.

Darauf aufbauend erfolgt eine Einleitung in das modellbasierte Testen. Die Kernidee von modellbasiertem Testen ist sowohl das Erreichen von Zeit-/ und Ressourcenersparnissen bei der Erstellung und Wartung von Tests, als auch eine Erhöhung der Testabdeckung und somit der Testqualität [2].

2.5 Das OSI-Modell

In Abbildung 2.4 sind die fünf Schichten des OSI[5]-Modells dargestellt. Zur Vereinfachung wurden die Layer (*deutsch: Schichten*) 5-7 als Application Layer (textitdeutsch: Schichten) zusammengefasst. Im Folgenden wird auf die einzelnen Schichten kurz eingegangen und auf weiterführende Literaturen verwiesen.

[5] **O**pen **S**ystem **I**nterconnection

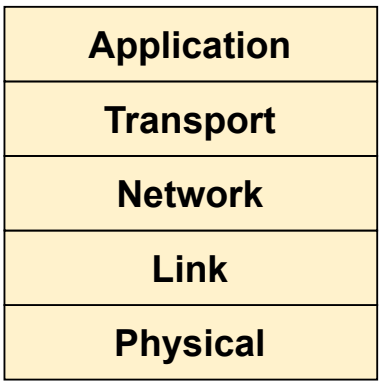


Abbildung 2.4: Das Vereinfachte OSI-Modell

Die *Application*-Schicht ist der Bereich, in dem Netzwerkanwendungen und ihre entsprechenden Protokolle angesiedelt sind. Diese Schicht umfasst viele Protokolle, darunter auch das HTTP[6]-Protokoll [16].

Im OSI-Modell wird auf der *Transport*-Schicht zwischen verschiedenen Protokollen unterschieden, wobei die zwei bekanntesten die folgenden beiden sind [16]:

- TCP - Bietet einen verbindungsorientierten Service an
- UDP - Bietet einen verbindungslosen Service an

Die *Network*-Schicht stellt ein Protokoll bereit, das die Kommunikation zwischen beliebigen Endgeräten ermöglicht. Ihre Hauptaufgabe besteht darin, Pakete (auch als *datagrams* bezeichnet) von einem Endgerät zum anderen zu transportieren [16]. Zusätzlich legt es fest, wie Endgeräte und Router mit diesen Feldern umgehen sollen [17].

Die Aufgabe der *Link*-Schicht ist es, die ganzen Pakete von einem Netzelement zu einem benachbarten Netzelement zu übertragen [16].

Die *Physical*-Schicht transportiert die individuellen Bits in einem Paket von einem Knoten zum nächsten. Als Medium wird meist ein *twisted-pair copper Kabel* (im Volksmund: LAN[7]-Kabel) verwendet [16].

[6] **H**yper**t**ext **T**ransfer **P**rotocol
[7] **L**ocal **A**rea **N**etwork

Nun wird noch kurz auf die folgenden Begriffe eingegangen, welche im späteren Verlauf relevant werden.

- IP-Adresse:

  Eine IP Adresse ist eine logische Adresse im Netzwerk (z. B. Adresse von einem PC). Damit die Rechner im Netzwerk miteinander kommunizieren können braucht jeder Rechner im Netz eine eigene IP-Adresse. Server besitzen statische IP-Adressen und verändern diese nicht [18].

- Subnetzmaske:

  Alle Rechner in einem lokalen Netzwerk (z. B. im Büro-Netz), bei denen der Netzanteil (=Subnetz) gleich ist, befinden sich im gleichen Subnetz und können miteinander kommunizieren. Wenn ein Rechner die Daten an einen anderen Rechner senden möchte, welcher sich im anderen Subnetz befindet, dann sendet er diese Daten an seinen Router. Der Router leitet diese Daten an die passende IP-Adresse weiter [18].

- Standardgateway:

  Mit der Option Standardgateway wird unter den Netzwerkeinstellungen angegeben, welche IP-Adresse der zuständige Router hat [18].

2.6 Interrupt Service Routine

Ein typischer Programmcode setzt sich in der Regel aus verschiedenen Elementen zusammen, darunter die Hauptfunktion (main), Definitionen, Funktionen, Aufrufen, Methoden und anderen ausführbaren Anweisungen [19].

In der Programmierung gibt es eine Unterscheidung zwischen Polling und Interrupts. Polling kann mit einer Dauerschleife (z. B. while(1)) verglichen werden. Diese Schleife wird kontinuierlich ausgeführt, während ein Interrupt nur unter bestimmten Events aufgerufen wird.

Im Vergleich zum Polling-Ansatz liegt der Vorteil eines interrupt-basierten Programms in der Effizienz des Mikrocontrollers. Bei einem Interrupt wird der Code nur ausgeführt, wenn der spezifische Ereignisaufruf stattfindet. Dadurch werden Ressourcen nur

verwendet, wenn sie benötigt werden. Im Gegensatz dazu verbraucht Polling konstant Ressourcen, unabhängig davon, ob etwas zu tun ist oder nicht [20, 21].

In Tabelle 2.1 sind Unterschiede und Merkmale zwischen den verschiedenen Betriebsarten dargestellt, darunter ihre Funktionsweise, Effizienz und der entstehende Ressourcenverbrauch [20, 21, 22].

Eigenschaft	**Polling**	**Interrupt**
Funktionsweise	Überprüft periodisch den Zustand in Form einer Schleife	Unterbricht die Schleife, um auf bestimmte Ereignisse zu reagieren
Effizienz	Verbraucht kontinuierlich Ressourcen durch ständiges Polling	Verbraucht Ressourcen nur bei bestimmtem Ereigniseintritt
Ressourcenverbrauch	Verursacht einen Overhead	Verbraucht Ressourcen nur im Moment eines Ereignisses

Tabelle 2.1: Vergleich von Interrupt und Polling

Die unten zusammengefassten Regeln [23] sind als Design-Guidelines beim Implementieren einer ISR[8] zu beachten.

1. ISR muss kurz gehalten werden, da die System-Latenz dadurch vergrößert wird.
2. Um Probleme bei der Prioritätsumkehrung zu vermeiden, sollten andere Interrupts während der ISR nicht laufen.

Ein visuelles Beispiel einer ISR ist in Abbildung 2.5 zu sehen.

[8] **I**nterrupt **S**ervice **R**outine

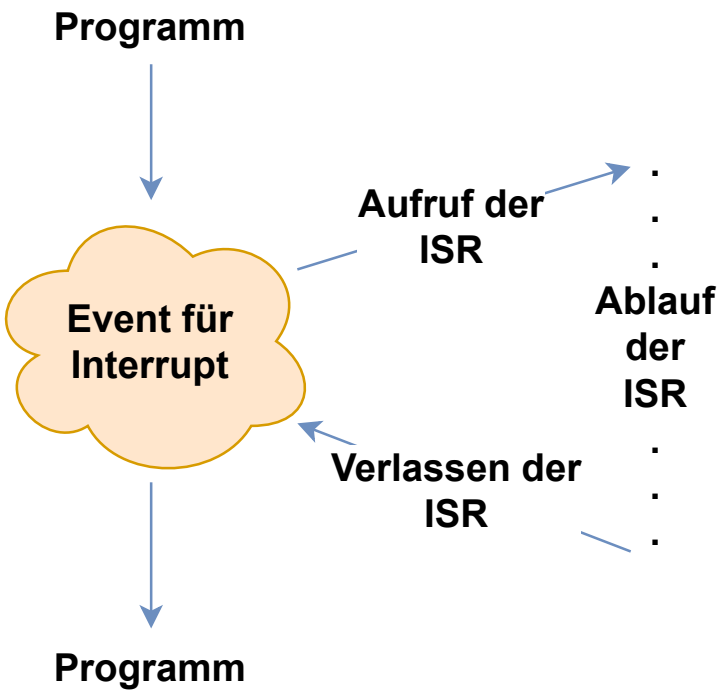


Abbildung 2.5: Aufruf einer Interrupt Service Routine

Wenn eine ISR während der Ausführung eines Programms aufgerufen wird, beispielsweise wenn sich das Programm in einer Dauerschleife befindet, stoppt das Programm und springt zur ISR. Nachdem die ISR abgeschlossen ist, kehrt das Programm an die Stelle zurück, an der es unterbrochen wurde, und setzt seine Ausführung fort. Dies ermöglicht eine reibungslose Fortsetzung des Programmablaufs nach der Bearbeitung des Interrupts.

Im Folgenden wird kurz auf die Vorteile vom Polling-Ansatz eingegangen.

Die Wahl vom Polling-Ansatz kann aufgrund der einfachen Implementierung und Portabilität stattfinden [22].

Aufgaben, welche in periodischen Abständen abgefragt werden müssen, können aufgrund der einfachen Implementierung mit dem Polling-Ansatz realisiert werden.

3 Anforderungserhebung

In [3, 4] erfolgt die Anforderungserhebung dieser Arbeit. Hierunter wird die Erstellung des Umfeldmodells (siehe Anhang A.1.1) beschrieben. Darüber hinaus werden die aufgestellten Anwendungsszenarien (siehe Anhang A.1.2) und das erstellte Lastenheft (siehe Anhang A.1.3) auf Systemebene vorgestellt.

Zusätzlich enthält diese Arbeit abgeleitete Systemmodellanforderungen, gegen welche das Implementierungsmodell (siehe Anhang A.1.6) verifiziert wird. Jedoch werden diese abgeleiteten Systemmodellanforderungen nicht an dieser Stelle aufgelistet, sondern auf den Anhang A.1.4 verwiesen, um die Übersichtlichkeit zu wahren.

Zur Gewährleistung einer strukturierten Projektplanung wird ein Meilensteinplan (siehe Anhang A.1.5) für die Masterarbeit erstellt.

4 Software-Architektur

Dieses Kapitel stellt die erarbeitete Software-Architektur in den folgenden drei Schritten vor:

1. Vorstellung des Regelalgorithmus
2. Vorstellung der Evaluierungsumgebung
3. Implementierung des External Modes über TCP

4.1 Anforderungen an das Modell

Die definierten Anforderungen (siehe Anhang A.1.4) wurden aus dem im Rahmen der Projektarbeit erstellten Lastenhefts (siehe Anhang A.1.3) abgeleitet. Diese spezifizierten Anforderungen (siehe Anhang A.1.4) werden dann in Simulink implementiert, verlinkt und verifiziert. Ein wesentlicher Vorteil dieses Vorgehens liegt in der Rückverfolgbarkeit der Anforderungen. Zudem bietet es einen schnellen Überblick über den Status der Implementierung und Verifikation.

4.2 Regelalgorithmus

In Abbildung 4.1 ist die erarbeitete Software-Architektur für die Regelung des Motors dargestellt. Zu sehen ist, dass es drei (Haupt-)Controller gibt:

1. Input Controller:

   Dieser Controller bereitet die Eingangssignale vom Benutzer für den Speed Controller vor.

2. Speed Controller:

   Der Speed Controller berechnet die nötige drehmomentbildende Stromkomponente und das Signal für das Fahren ohne elektrischen Antrieb.

3. Torque Controller:

   Der Torque Controller verwendet die vom Speed Controller kommenden Signale zur Erzeugung der PWM[1]-Signale für die Leistungsstufe und gibt dem Speed Controller die aktuelle Geschwindigkeit vor.

Des Weiteren sind die CAN[2]-Nachrichten dargestellt, welche empfangen und gesendet werden.

[1] **P**ulse **W**idth **M**odulation
[2] **C**ontroller **A**rea **N**etwork

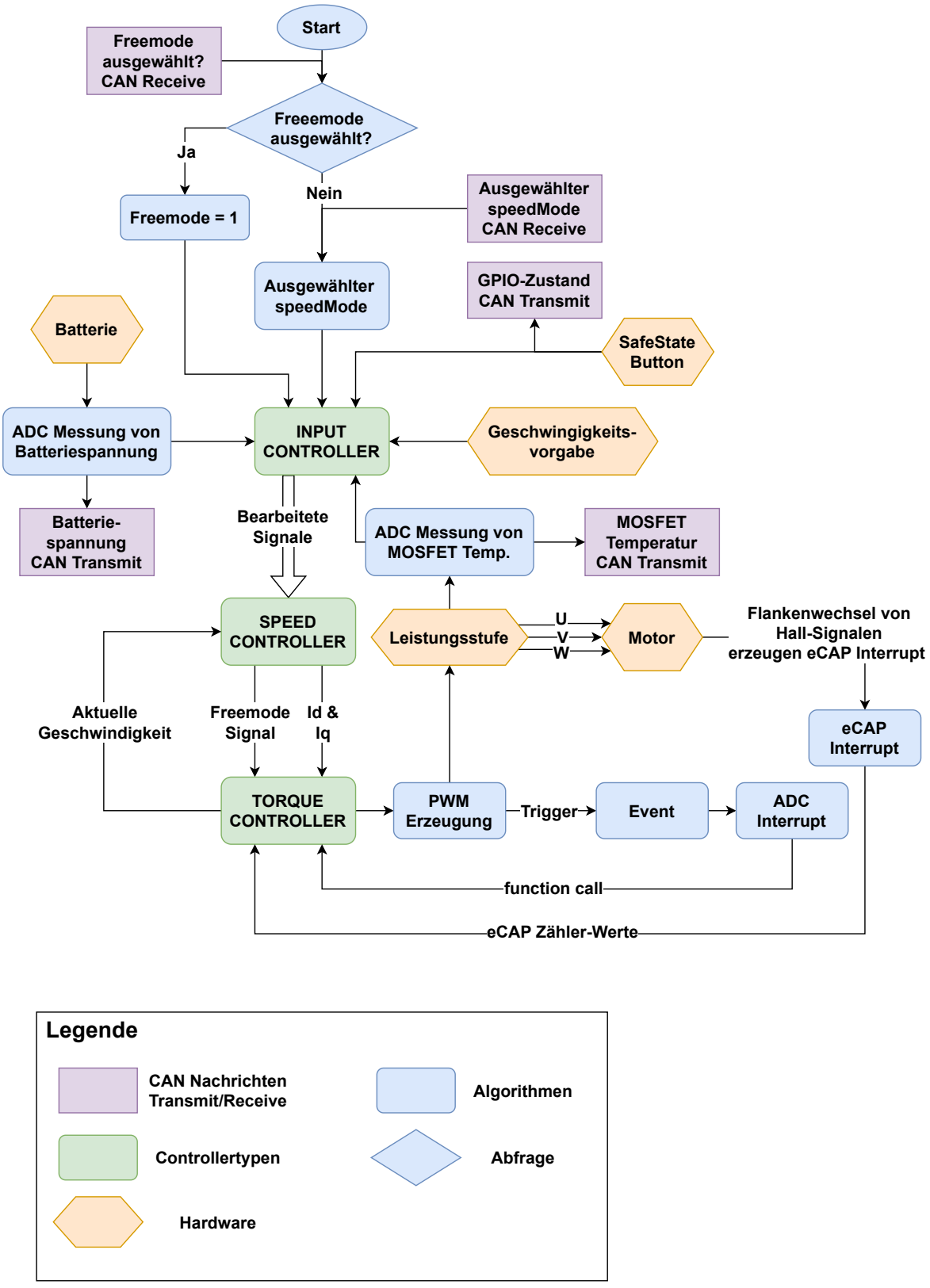


Abbildung 4.1: Software-Architektur in der Single-Core-Ausführung

Der Input Controller bereitet die Eingangssignale für den Speed Controller vor. Hierbei werden folgende Signale berücksichtigt:

1. Gewünschte Geschwindigkeit
2. Freemode (Das Fahren ohne elektrischen Antrieb)
3. SpeedMode (Verfügbare Geschwindigkeitsmodi)
4. Batteriespannung
5. MOSFET[3]-Temperatur
6. GPIO[4]-Zustand (Button für den sicheren Zustand)

Im Gegensatz zum Auto wird die gewünschte Geschwindigkeit nicht durch eine Drehmomentanforderung, sondern durch eine direkte Geschwindigkeitsanforderung gesteuert. Diese Geschwindigkeitsanforderung wird vorab für den Speed Controller vorverarbeitet. Um diese ohne eine zusätzliche Umwandlung verwenden zu können.

Die Variable *Freemode* verfügt über den Wert, ob der Benutzer den Modus des Fahrens ohne elektrischen Antrieb ausgewählt hat. Wenn der *Freemode* aktiviert ist, werden die High- & Low-Side der Leistungsstufe dauerhaft geöffnet, um ein Fahren ohne elektrischen Antrieb zu ermöglichen.

Mit der Variable *speedMode* kann der Benutzer unter den verschiedenen Geschwindigkeitsmodi auswählen. Die Geschwindigkeitsmodi unterscheiden sich in der maximal erreichbaren Geschwindigkeit.

Die Batteriespannung wird verarbeitet an den Speed Controller weitergegeben. Dadurch erhält der Controller die Information, ob die Unterspannungsgrenze unterschritten wird und kann zur richtigen Zeit *Freemode* auf 1 setzen.

Die MOSFET-Temperaturüberwachung ist ähnlich wie die Unterspannungsabschaltung. Sie ermittelt mithilfe der Leistungsstufe die Umgebungstemperatur der MOSFETs. Bei Überschreitung eines bestimmten Schwellwertes, wird *Freemode* auf 1 gesetzt und somit das Fahren ohne elektrischen Antrieb aktiviert.

[3] **M**etal **O**xide **S**emiconductor **F**ield **E**ffect **T**ransistor
[4] **G**eneral **P**urpose **I**nput/**O**utput

Der GPIO-Zustand dient als mechanische Sicherheitsmaßnahme. Wenn der Antrieb nicht mehr richtig funktioniert oder ein Fehler auftritt, welcher nicht über ein Interaktionsmenü behoben werden kann, erlaubt dieser GPIO, die Stromversorgung des Antriebs dauerhaft zu unterbrechen.

Die vorbereiteten und verarbeiteten Eingaben werden an den Speed Controller übermittelt, um die erforderliche drehmomentbildende Stromkomponente zu berechnen.

Die berechnete drehmomentbildende Stromkomponente und das *Freemode*-Signal werden dem Torque Controller übergeben. Zusätzlich bekommt der Speed Controller die aktuelle Geschwindigkeit vom Torque Controller.

Der Torque Controller berechnet die Tastverhältnisse für die Leistungsstufe um die gewünschte Geschwindigkeit zu erreichen. Die Erzeugung der PWMs für die Leistungsstufe löst ein Event aus, welcher die ISR für die Strommessung startet.

Die beschriebene Architektur beschreibt das gewünschte Signalverhalten. In dieser Darstellung handelt es sich um eine Single-Core-Ausführung. Hierdurch wird der Code auf einer CPU[5] ausgeführt und nicht auf mehreren Prozessoren partitioniert [24].

Alle Blöcke haben eine spezifische *sample time*. Damit die Komponenten untereinander Signale austauschen können, sind sogenannte Über-/ oder Unterabtastungen nötig. In der Zeichnung wurden diese spezifischen Über-/ und Unterabtastungen bewusst weggelassen. Auf die Über-/ und Unterabtastungen wird in Kapitel 5 erneut eingegangen.

Abschließend wird nun in Tabelle 4.1 gezeigt, welche Bereiche des Codes im Interrupt-Kontext und welche Bereiche in Dauerschleife ausgeführt werden.

Interrupt	**Dauerschleife**
ADC	Input Controller
eCAP1	Speed Controller
eCAP2	ePWM-Module
eCAP3	eCAN-Nachrichten
	Heartbeat LED

Tabelle 4.1: Aufteilung der Komponenten in Interrupts & Polling

[5] **C**entral **P**rocessing **U**nit

Aufgrund der Tatsache, dass der Benutzer jederzeit neue Inputs geben kann, die Batteriespannung zum eigenen Schutz abschalten wird oder die MOSFETs zu heiß werden, wird in periodischen Zeitabständen der Input Controller aufgerufen. Aus diesem Grund befindet sich der Input Controller in der Dauerschleife.

Gleiches gilt für den Speed Controller. Der Speed Controller bestimmt mithilfe des Input Controllers die neuen Werte. Daher läuft diese Komponente in periodischen Zeitabständen und befindet sich deswegen in der Dauerschleife.

Die CAN-Nachrichten werden auch in periodischen Zeitabständen empfangen bzw. gesendet. Aufgrund dessen befinden sich diese auch in Dauerschleife.

Die Leistungsstufe wird (periodisch) mit einer PWM-Frequenz von 20 kHz betrieben. Daher befinden sich die ePWM-Module in der Dauerschleife.

Die *Heartbeat LED* signalisiert dem Benutzer, dass keine Probleme bezüglich der Codeausführung vorhanden sind. Aufgrund der periodischen Ausführung dieser Komponente befindet sie sich in Dauerschleife.

Die ADC-Routine ist interrupt-basiert, da diese mithilfe der ePWM-Module getriggert wird.

Durch die Flankenwechsel der Hall-Signale werden die jeweiligen Events ausgelöst, welche die eCAP-Interrupts aufrufen. Diese sind somit ebenfalls interrupt-basiert.

4.3 Mikrocontroller

Aufgrund positiver Erfahrungen in früheren Projekten fiel die Wahl auf die Texas Instrument C2000-Derivate. Passend zu den Anforderungen (siehe Anhang A.1.3) des Mikrocontrollers wird der TMS320F28388D ausgewählt. Die Abbildung 4.2 zeigt diesen integrierten Controller in einer controlCARD mit der Bezeichnung TMDSCNCD28388D [25].

Zu der Peripherie und verfügbaren Schnittstellen des Controllers wird an dieser Stelle auf [4] verwiesen. Da die vorgesehene TCP Kommunikation realisiert wird, wird der von der controlCARD zur Verfügung gestellte RJ45[6]-Anschluss verwendet.

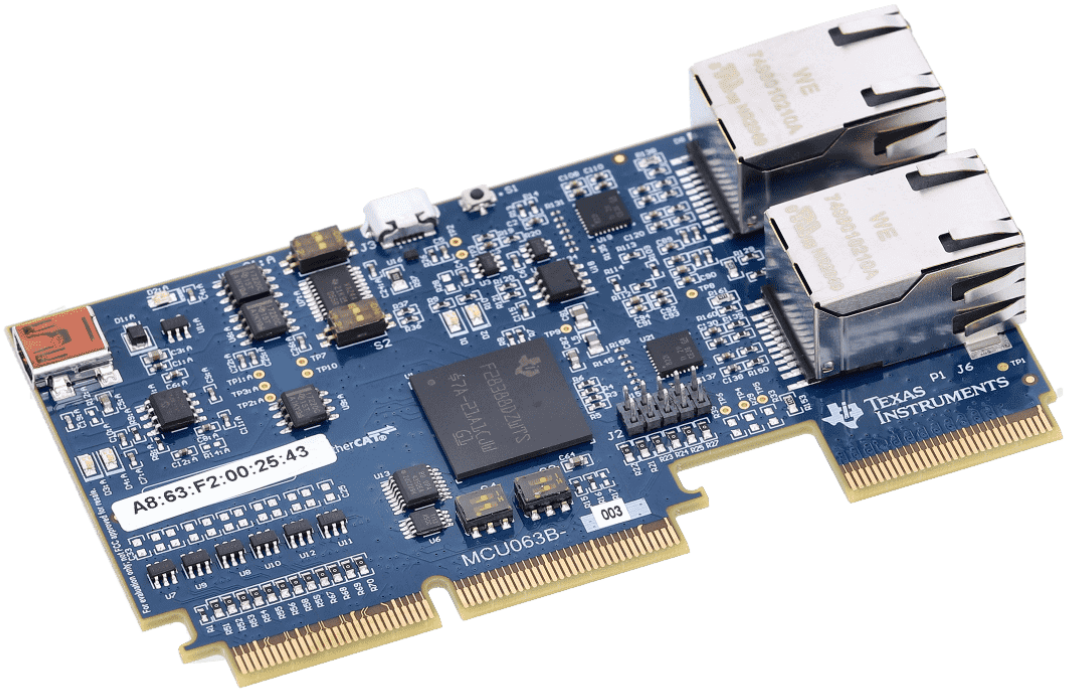


Abbildung 4.2: Evaluationsboard des F28388D integriert auf dem TMDSCNCD28388D[25]

Die Wahl der controlCARD des erwähnten Controllers ermöglicht eine kontinuierliche Evaluation, sodass auf dieser die erstellte Software jederzeit getestet werden kann. Um

[6] **R**egistered **J**ack-**45**

die Pins des Controllers zu nutzen, wird eine Docking-Station benötigt. Da nur eine passende Docking-Station [25] verfügbar ist, wird der TMDSHSECDOCK verwendet. In Abbildung 4.3 ist die verwendete Docking-Station abgebildet, welche die controlCARD in einen Slot einfügt und die Ein- und Ausgänge mit vertikal stehenden Pins zur Verfügung stellt.

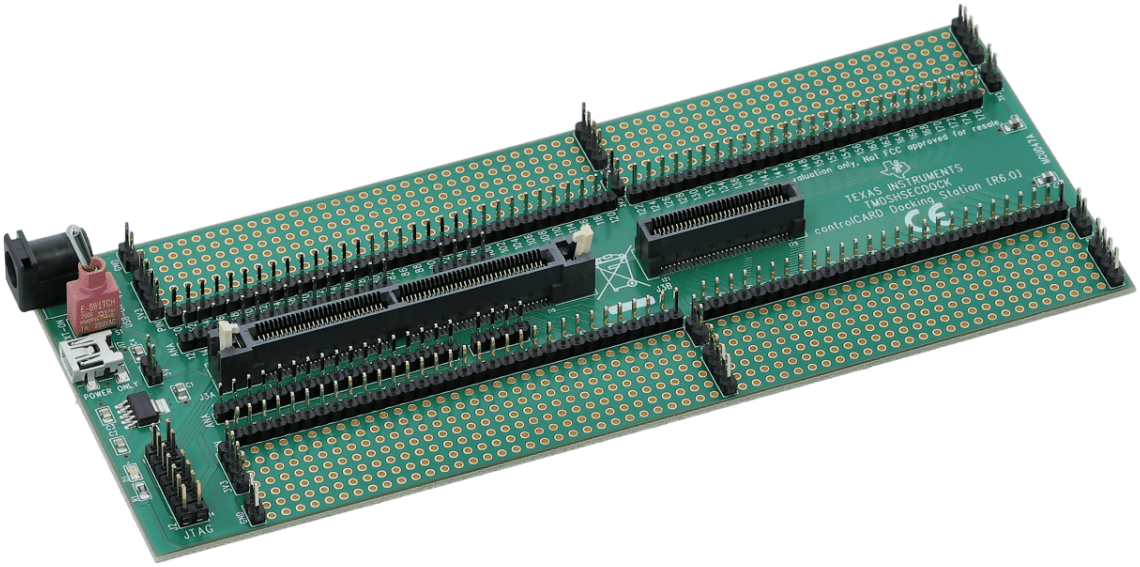


Abbildung 4.3: TMDSHSECDOCK - Docking Station für die TMS320F28388D-controlCARD [26]

4.4 Leistungsstufe

In dieser Arbeit wird die Leistungsstufe DRV8300DIPW-EVM verwendet, welche in Abbildung 4.4 zu sehen ist. Über den vorhandenen Gatetreiber und Peripherie wird an dieser Stelle auf [4] verwiesen.

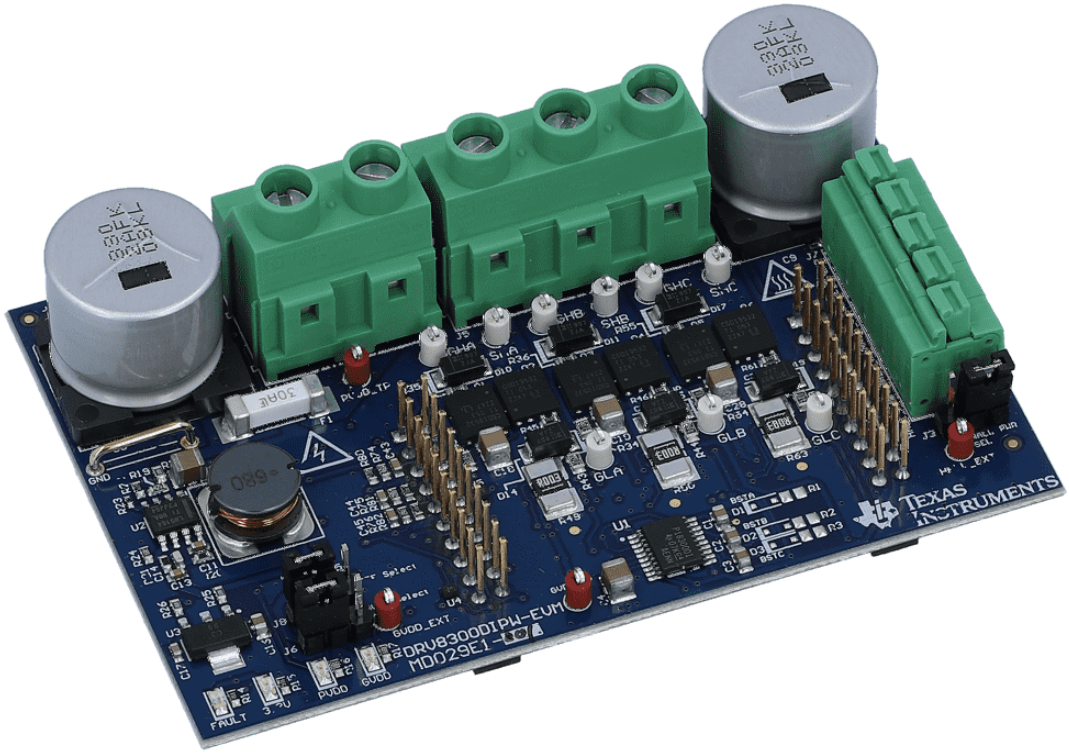


Abbildung 4.4: DRV8300DIPW-EVM Leistungsstufe [27]

Die Messungen für die Batteriespannung (PVDD[7]), GVDD[8] und den Strangströmen werden somit von der Leistungsstufe übernommen. Diese werden softwareseitig umgerechnet und für die Motoransteuerung verwendet.

[7]Power Supply Voltage
[8]Gate Driver Supply Voltage

4.5 External Mode

Vorgesehen ist die Implementierung des External Modes [28]. Aufgrund früherer Erfahrungen [6] mit träge reagierendem Verhalten wird in dieser Arbeit der Ansatz verfolgt, den External Mode [28] über TCP [7] zu betreiben. Bei Performance-Problemen kann optional auch UDP verwendet werden.

Es ist vorgesehen, dass der Server auf dem Target eingerichtet wird und über einen Host Computer als Client auf den Server zugegriffen wird. Eine Übersicht der Implementierung ist in Abbildung 4.5 dargestellt.

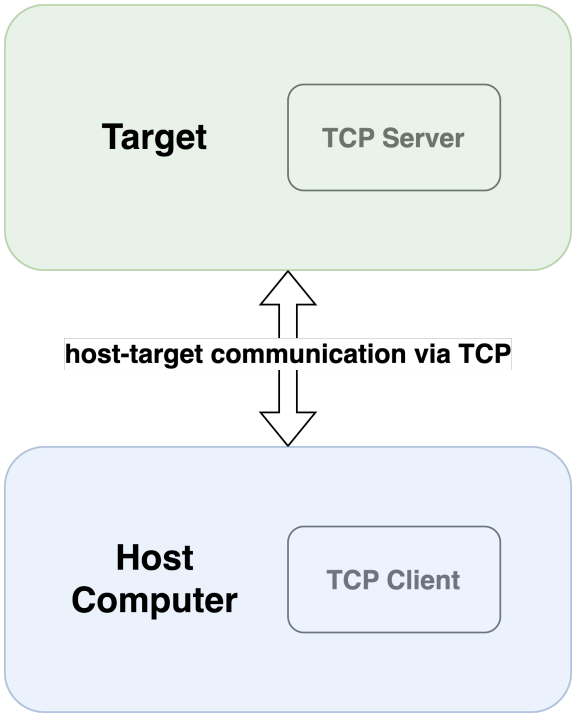


Abbildung 4.5: Übersicht des External Modes über TCP

In Kapitel 5.7 wird die Implementierung samt Konfiguration der Blöcke vorgestellt. Entscheidend ist die Vergabe der IP-Adressen und Port Nummer. Hierdurch wird ein Anfangs- und Endpunkt der Ethernet-Verbindung erstellt und es entsteht eine sog. *host-to-target communication* über TCP [7].

5 Implementierung

In diesem Kapitel wird die Implementierung anhand des Implementierunsgmodells (siehe Anhang A.1.6) erklärt und beschrieben. Dieses Kapitel umfasst folgende Themen:

1. Vorbereitung des Implementierungsmodells
2. Implementierung der drei beschriebenen Controller
3. Verwendete Modellvarianten
4. Implementierung und Konfiguration des External Modes

Die in dieser Arbeit verwendeten Toolboxen sind in der folgenden Tabelle zusammengefasst.

Toolbox / Target Support Package	**Version**
C2000 Microcontroller Blockset	1.0
Embedded Coder	7.10
Motor Control Blockset	2.0
Requirements Toolbox	2.2
Simscape	5.5
Simscape Electrical	7.9
Simulink Coder	9.9
Simulink Coverage	5.6
Simulink Real-Time	8.2
Simulink Real-Time Target Support Package	23.1.1
Simulink Test	3.8
SoC Blockset	1.8
Variant Manager for Simulink	23.1.0

Tabelle 5.1: Verwendete Toolboxen und Target Support Packages

Die Evaluierung der entwickelten Software mit der im Rahmen der Arbeit "Entwicklung und Implementierung der Hardware eines modularen Frequenzumrichters"[4] entwickelten Hardware wird erst in Kapitel 7 beschrieben.

Da die in [4] zu entwickelnde Hardware (nahezu) dieselben Spezifikation wie die controlCARD aus Kapitel 4.3 aufweist, wurde diese zu Zwischentestzwecken verwendet.

5.1 Allgemeines

In diesem Kapitel werden folgende Themen besprochen:

1. Das verwendete Einheitensystem
2. Vorstellung des Workspaces
3. Die verwendeten Datentypen
4. Beschränkungen des Modells in Abhängigkeit des verwendeten Motors

5.1.1 Per Unit-System

Das PU[1]-System wird in der Elektrotechnik häufig verwendet, um die Werte von Größen wie Spannung, Strom, Leistung usw. auszudrücken. Es wird für Transformatoren und Wechselstrommaschinen zur Analyse von Energiesystemen verwendet. Ingenieure für eingebettete Systeme verwenden dieses System auch zur optimierten Codegenerierung und Skalierbarkeit, insbesondere bei der Arbeit mit Targets, welche mit Festkommadarstellung (siehe Kapitel 5.1.3) arbeiten [29]. Die Vorteile für das Verwenden des PU-Systems werden im folgenden aufgezählt.

1. Verbessert die rechnerische Effizienz der Codeausführung und ist daher ein bevorzugtes System für Targets mit Festkommadarstellung [29].
2. Schafft einen skalierbaren Kontrollalgorithmus, der in vielen Systemen verwendet werden kann [29].

An dieser Stelle wird auf die Berechnung der einzelnen Parameter verzichtet, stattdessen wird auf ergänzende Literaturen verwiesen. [29, 30, 31]

[1] **P**er-**U**nit

5.1.2 Workspace

Wenn die beigefügte Datei `params_f2838.mat` (siehe Anhang A.1.7) in das *Command Window* eingefügt wird, erstellt diese Variablen im Workspace, welche in Abbildung 5.1 dargestellt sind. Die erstellten Variablen werden im Implementierungsmodell an verschiedenen Stellen verwendet.

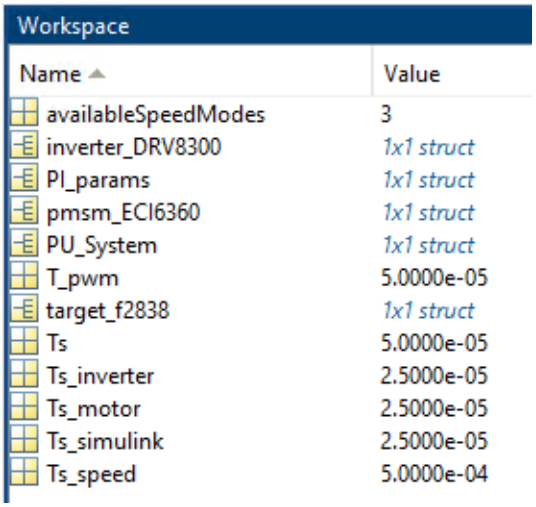


Abbildung 5.1: Erstelltes Workspace zur Definition von Variablen im Implementierungsmodell

Zunächst werden die zwei verschiedenen Typen der erstellten Variablen vorgestellt. In dieser Arbeit wurden 1-Dimensionale MATLAB Variablen und sog. Strukturarrays (*engl.: struct*) verwendet.

Ein Strukturarray ist ein Datentyp, der zusammengehörige Daten in Datenbehältern, den so genannten Feldern, gruppiert. Jedes Feld kann jede Art von Daten enthalten. Der Zugriff auf Daten in einem Feld erfolgt über eine Punktnotation der Form `structName.fieldName` [32].

In dieser Arbeit wurden die Parameter vom Controller, der Leistungsstufe, des Motors, des PU-Systems und der PI-Regler für den Torque Controllers und Speed Controllers in Form von Strukturarrays definiert.

Als Beispiel sind die Parameter des Controllers in Abbildung 5.2 zu sehen.

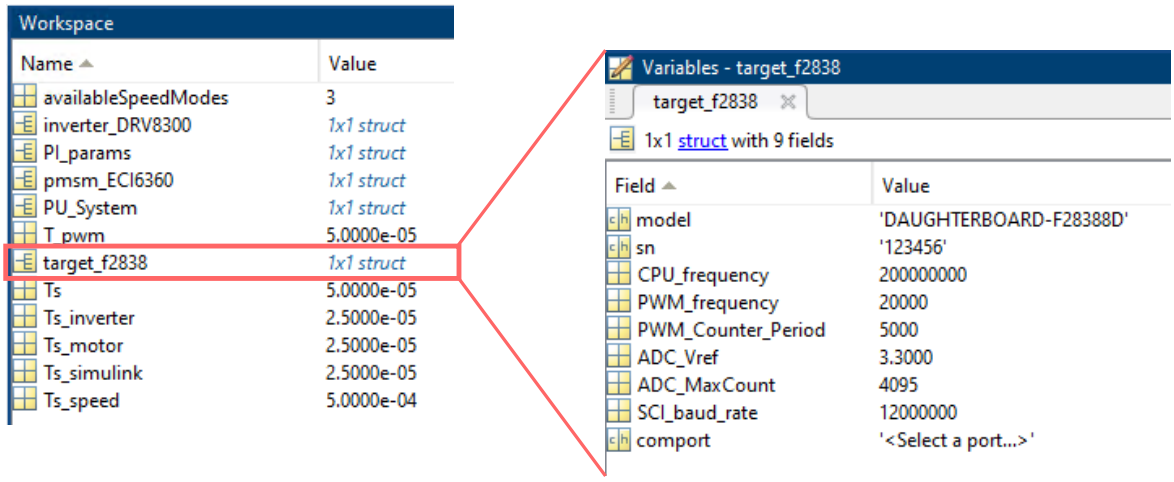


Abbildung 5.2: Definierte Parameter vom Controller

Die beschriebene Art und Weise der Definition der Variablen erfolgt im sog. *Base Workspace*. Im Gegensatz zu diesem gibt es das *Model Workspace*. Dieser wurde nicht (wie in den früheren Arbeiten [1, 2]) verwendet. Grund dafür ist die verbesserte Übersicht und Differenzierung von Modell und Variablen. Zusätzlich werden, unter der Verwendung eines *Scripts*, viele Variablen mit einer Änderung angepasst, da bestimmte Parameter voneinander abhängig sind.

Zum Beispiel ist die PWM-Frequenz und die Periodendauer voneinander abhängig. Die *sample times* vom Streckenmodell (siehe Kapitel 5.6) sind abhängig von der Periodendauer der PWMs. Wie also zu sehen ist, besitzen viele Parameter eine große Abhängigkeit.

Hierdurch können MATLAB-Funktionen verwendet werden, welche bei der Implementierung behilflich sind.

Der Strukturarray *PI_params* speichert die Variablen, welche im Torque- und Speed Controller verwendet werden. Um eine Verifikation mit *gutem* Laufverhalten zu gewährleisten, müssen die PI-Regler entsprechende Koeffizienten besitzen. Die Regler-Koeffizienten werden mithilfe der folgenden MATLAB-Funktion berechnet.

```
PI_params = mcb.internal.SetControllerParameters(pmsm_ECI6360,
inverter_DRV8300,PU_System,T_pwm,Ts,Ts_speed)
```

Die berechneten Parameter werden im Implementierungsmodell (siehe Anhang A.1.6) verwendet und in Kapitel 7 diskutiert.

5.1.3 Verwendete Datentypen im Implementierungsmodell

Das Nutzen des PU-Einheitensystems ist besonders vorteilhaft bei der Nutzung von Festkommadarstellung (*engl. Fixed-Point*).

An vielen Orten des Implementierungsmodells (siehe Anhang A.1.6) wird folgende Festkommadarstellung zu finden sein:

```
fixdt(1,32,17)
```

Die `1` in der Notation sagt aus, dass es sich um eine *signed* Zahl handelt, sodass ein positiver und negativer Zahlenwert darstellbar ist. Die `32` gibt die *word length* an. Die Wortlänge gibt die Anzahl der Bits an, die für die Festkommazahl verwendet wird [33]. Die `17` gibt die *fraction length* an. Mit dieser ist es möglich die Genauigkeit auf ≈7.629e-06 zu steigern. Der darstellbare Zahlenbereich dieser Festkommadarstellung geht von -16384 bis ≈+16384.

Hierbei handelt es sich um die verwendete Festkommadarstellung. Mithilfe dieser wird zum Beispiel der gemessene Strom in das PU-System umgewandelt. Näheres hierzu später in Kapitel 5.4. Eine Übersicht der Details über diese Festkommadarstellung ist in Abbildung 5.3 dargestellt.

Abbildung 5.3: Festkommadarstellung im *Data Type Conversion*-Block

Aufgrund vorgegebener (Eingabe-)Datentypen bestimmter Blöcke, wie zum Beispiel für das Empfangen bzw. Senden von CAN-Nachrichten, kann die Festkommadarstellung nicht immer verwendet werden. Daher müssen an vielen Stellen *Data Type Conversion*-Blöcke verwendet werden. Dieser Block konvertiert einen Datentyp in einen anderen Datentyp.

5.1.4 ECI63.60

Um die Vorteile der modellbasierten Entwicklung zu nutzen, wird der verwendete Motor [34] ebenfalls in Simulink implementiert. Laut dem Datenblatt [34] sind bereits einige Parameter gegeben, jedoch sind viele Parameter undefiniert. Zum Beispiel der Parameter *Flux Linkage*, welcher sowohl für die berechneten Regler-Koeffizienten als auch für Parameter des PU-Systems wichtig ist. Unabhängig davon sind die Induktivitäten der D-/ und Q-Achsen unbekannt.

Es existieren Verfahren diese Parameter des Motors zu messen. Aufgrund der Einarbeitung und fehlenden Zeit, wurden diese in dieser Arbeit nicht durchgeführt. Stattdessen wurden die Werte aus dem letzten Projekt [2] und dem Datenblatt [34] übernommen.

Genaueres zur virtuellen Umgebung erfolgt in Kapitel 5.6.

5.2 Input Controller

Der Input Controller bereitet mithilfe der Benutzereingaben die Signale für den Speed Controller vor.

Der Input Controller ist in Abbildung 5.4 zu sehen. Die farbliche Symbolisierung beschreibt die *sample times*. Der gelbe Rahmen dieses Subsystems deutet darauf hin, dass nicht alle Komponenten in diesem Subsystem die gleiche *sample time* besitzen. Alle Variablen bis auf *Speed_Ref* besitzen eine *sample time* von 0.2 s. Die Variable *Speed_Ref* besitzt eine *sample time* von 0.01 s. Die schnellere *sample time* dieser Variable lässt sich damit begründen, dass die Geschwindigkeitsanforderung höher priorisiert wird und somit öfter abgefragt werden muss.

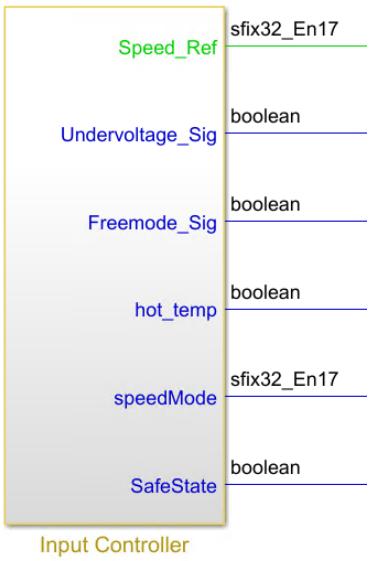


Abbildung 5.4: Input Controller im Implementierungsmodell

Im nächsten Schritt wird auf die Komponenten im Input Controller eingegangen.

Die Geschwindigkeitsanforderung wird in dieser Arbeit mithilfe eines Potentiometers gesteuert. Durch die Anwendung eines ADCs wird der digitale Wert normalisiert. Der normalisierte Wert wird anschließend weitergeführt und gelangt in den Speed Controller.

Mithilfe der Leistungsstufe wird die Versorgungsspannung an einen unbelasteten Spannungsteiler angeschlossen. Durch Termumformung wird die Versorgungsspannung ermittelt, welche im Folgenden erklärt wird.

In Gl. 5.1 ist die Gleichung für die Ausgangsspannung eines unbelasteten Spannungsteilers zu sehen.

$$U_A = U_E \cdot \frac{R2}{R1 + R2} \tag{5.1}$$

Durch Termumformung der Gl. 5.1 nach der Eingangsspannung, entsteht Gl. 5.2. Diese Gleichung kann somit verwendet werden um die Versorgungsspannung zu ermitteln. Dieser Ansatz wurde implementiert und kann auch aus Abbildung 5.5 entnommen werden.

$$U_E = U_A \cdot \frac{R1 + R2}{R2} \tag{5.2}$$

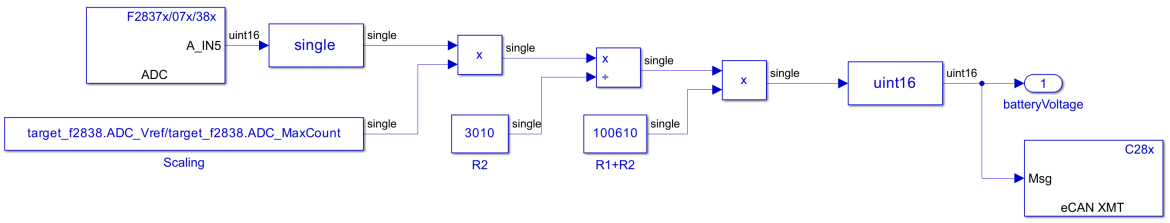


Abbildung 5.5: Berechnung der Versorgungsspannung

Zuletzt wird die berechnete Spannung über den CAN-Bus gesendet. Dieses wird mithilfe eines PCAN-Adapters verifiziert.

Eine Übersicht der empfangenen bzw. gesendeten CAN-Nachrichten kann aus Anhang A.1.17 entnommen werden.

Nach der Berechnung der Versorgungsspannung wird geprüft, ob diese unter einem bestimmten Schwellwert liegt. Der Ausgang dieser Abfrage wird als *Undervoltage_Sig* weiterverarbeitet.

Die Variable *Freemode* wird mithilfe einer empfangenen CAN-Nachricht implementiert. In Abbildung 5.6 ist die Implementierung der Bedingung des Fahrens ohne elektrischen Antrieb zu sehen.

Die empfangene CAN-Nachricht ist ein Vektor mit der Dimension von 4. Da nur eine skalare Größe nötig ist, wird ein *Selector* verwendet, um eine bestimmte Nachricht auszuwählen. Ansonsten kann es im späteren Verlauf der Software zu Dimensionsproblemen kommen. In diesem Fall besitzen alle vier Nachrichten den gleichen Wert. Daher wird bei eingehenden CAN-Nachrichten immer nur das erste Element betrachtet.

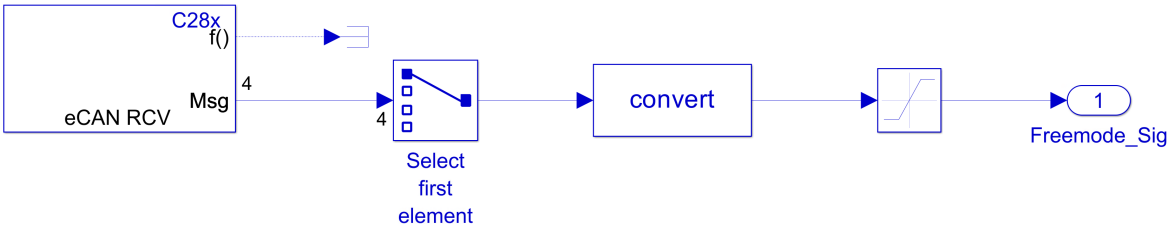


Abbildung 5.6: Implementierung der Variable *Freemode*

Da das Modell hauptsächlich mit der Festkommadarstellung arbeitet, muss der Wert noch in den richtigen Datentyp konvertiert werden. Hierzu wird der *Data Type Conversion*-Block verwendet.

Im weiteren Verlauf wird überprüft ob die Variable *Freemode* wahr (*True*) oder falsch (*False*) ist. Daher wird am Ausgang des Input Controllers geprüft, ob der Wert größer als Null ist und als *Freemode_Sig* weiterverarbeitet.

Der Sättigungsblock (*hier: Saturation*) ist als Sicherheitsmaßnahme eingesetzt, um Fehler bei empfangenen CAN-Nachrichten auszuschließen. Damit diese Komponenten eindeutig prüfbar ist, liegt die Wertebegrenzung zwischen 0 und 1.

Die Variable *temperatureMOS* wird ähnlich ermittelt wie die Versorgungsspannung. Zunächst wird über ein ADC die anliegende Spannung des Temperatursensors gemessen und umgerechnet.

Laut dem Datenblatt des Temperatursensors [35] kann mit der Gl. 5.3 die (Umgebungs-)Temperatur berechnet werden. In Abbildung 5.7 ist die Implementierung der Temperaturermittlung zu sehen. Da die Temperatur (in dieser Arbeit beispielsweise) bis 80 °C berücksichtigt wird, werden die Koeffizienten aus der Tabelle [35] für den Temperaturbereich -40 °C bis +100 °C verwendet. Hierdurch entfällt die Addition mit T_{INFL}, welches der Temperatur-Wendepunkt für ein stückweises Segment in °C ist [35].

$$T_A = \frac{(V_{OUT} - V_{OFFS})}{T_C} + T_{INFL} \tag{5.3}$$

Nach der Berechnung der Temperatur wird geprüft, ob diese über einem bestimmten Schwellwert liegt. Der Ausgang dieser Abfrage wird als *hot_ temp* weiterverarbeitet.

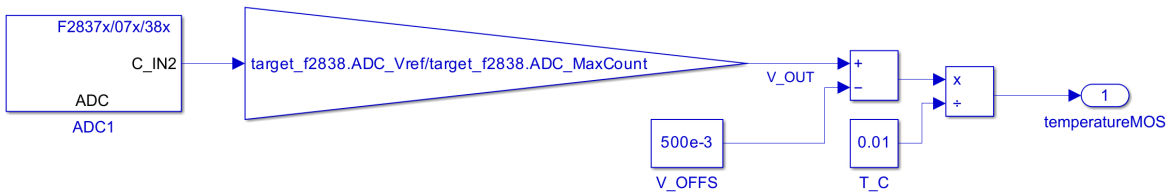


Abbildung 5.7: Berechnung der MOSFET-Temperatur

Auf gleiche Weise wie die Variable *Freemode* wird die Variable *speedMode* in Abbildung 5.8 ermittelt. Zu Beginn wird mithilfe einer empfangenen CAN-Nachricht der *speedMode* ausgewählt.

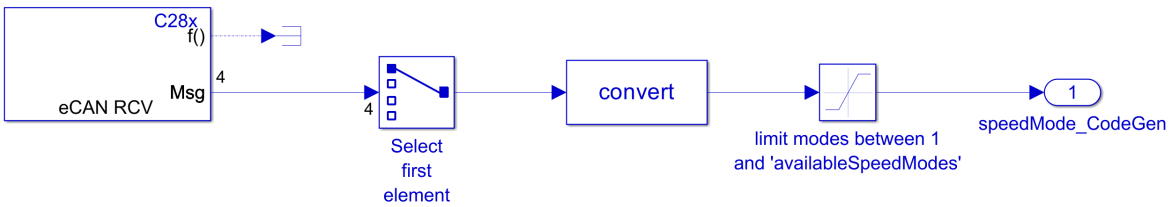


Abbildung 5.8: Ermittlung des *speedModes*

Der *Selector* sorgt erneut dafür, dass nur eine skalare Größe des Vektors ausgewählt wird.

Damit die Festkommadarstellung weiterhin gewährleistet wird, erfolgt erneut eine *Data Type Conversion*.

Mithilfe des Sättigungsblocks wird der Wertebereich begrenzt. Hier wird der Wertebereich abhängig von den verfügbaren Modi festgelegt. In dieser Arbeit werden drei verschiedene *speedModes* verfügbar sein, daher liegt der zulässige Wertebereich zwischen 1 und 3. Grund hierfür ist erneut die Gewährleistung des richtigen Wertebereiches der empfangenen CAN-Nachricht.

Die letzte Komponente des Input Controllers ist eine Sicherheitskomponente. Die Implementierung einer mechanischen Einrichtung ist für den Fall nötig, wenn sich das System nicht mehr ordnungsgemäß verhält. Hierdurch wird der sichere Zustand verlassen. Durch Verlassen des sicheren Zustands wird die Stromversorgung des Motors unterbrochen.

In Abbildung 5.9 ist die Implementierung des sicheren Zustands zu sehen.

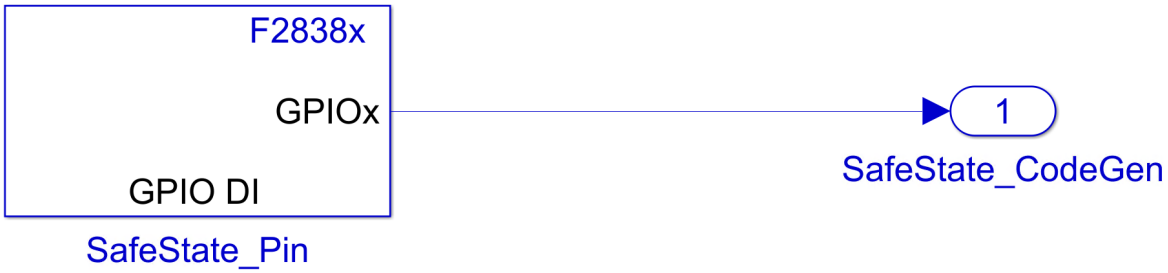


Abbildung 5.9: Implementierung des sicheren Zustands mit einem *GPIO Digital Input*-Block

Die mechanische Sicherheitseinrichtung ist in dieser Arbeit als *button* realisiert. Mithilfe eines *GPIO Digital Input*-Blocks wird der Pegel des GPIOs überprüft. Bei Knopfdruck wird der jeweilige Pin auf GND[2] gezogen. Beim nicht betätigtem Fall wird der Pin mit dem internen Pull-Up auf die Versorgungsspannung gezogen. Der interne Pull-Up wird über den *System Initialize*-Block realisiert.

[2] **Ground**

Im *System Initialize*-Block kann in ein jeweiliges Register reingeschrieben werden. Das Register *GPAPUD* aktiviert oder deaktiviert die internen Pull-Up Widerstände von GPIO 0-31. Die folgende Zeile aktiviert den internen Pull-Up Widerstand für GPIO 30.

```
//Enable internal Pull-Up Resistor for GPIO30
GpioCtrlRegs.GPAPUD.bit.GPIO30 = 0;
```

Um in das jeweilige Register zu schreiben muss zunächst das Oberregister ausgewählt werden. Das Register *GpioCtrlRegs* steuert die GPIOs. Im Register *GPAPUD* kann dann mithilfe einer Bitauswahl, der Pull-Up Widerstand des jeweiligen GPIOs aktiviert oder deaktiviert werden.

Es wird aufgrund der verfügbaren Peripherie mithilfe der Leistungsstufe die Versorgungsspannung des Gatetreibers (GVDD) gemessen. In Abbildung 5.10 ist die Berechnung vom GVDD zu sehen.

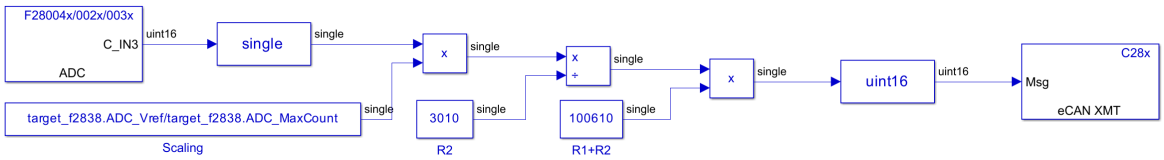


Abbildung 5.10: Berechnung der Versorgungsspannung des Gatetreibers

Die Berechnung der Versorgungsspannung des Gatetreibers (GVDD) erfolgt analog zur Berechnung der Versorgungsspannung (PVDD). Daher ist die Umformung von Gl. 5.1 auch hier vorhanden. Anschließend wird das Ergebnis in den richtigen Datentypen konvertiert und per CAN-Nachricht gesendet.

Die Implementierung verschiedener *DriveModes* konnte aufgrund der Komplexität nicht innerhalb des Zeitrahmens durchgeführt werden. Die Anforderung mit der ID *SW_ANF_02* konnte hiermit nicht erfüllt werden.

5.3 Speed Controller

In diesem Kapitel wird die Implementierung des Speed Controllers beschrieben und erläutert.

In Abbildung 5.11 ist eine Übersicht der eingehenden Signale und ausgehenden Signale des Speed Controllers zu sehen.

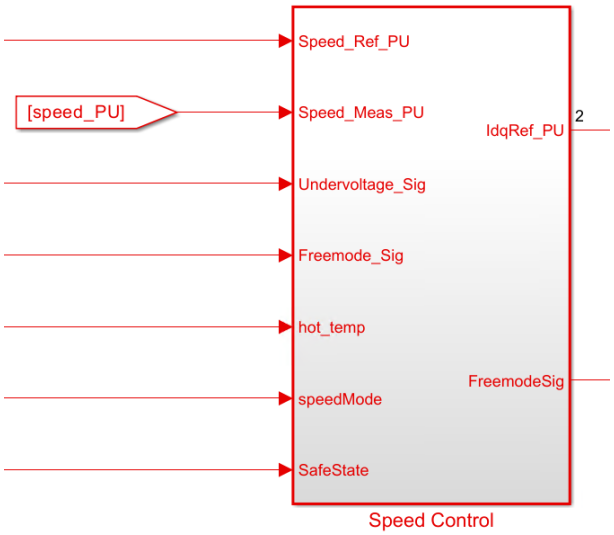


Abbildung 5.11: Speed Controller im Implementierungsmodell

In den Speed Controller gelangen alle vom Input Controller ausgehenden Signale. Das Signal der aktuellen Geschwindigkeit wird zudem vom Torque Controller an den Speed Controller geleitet.

Der *Speed calculator* im Speed Controller erhält als Input die bereits normalisierte angeforderte Geschwindigkeit und den entsprechenden *speedMode*.

In Gl. 5.4 erfolgt die Beschreibung des verwendeten Vorgehens. Die normalisierte angeforderte Geschwindigkeit $Speed\_Ref\_PU$ wird mit dem Bruch aus der Nenndrehzahl (hier: $PU\_System.N\_base$) und der Anzahl der verfügbaren *speedModes* multipliziert. Das Ergebnis dieser Berechnung wird mit dem ausgewählten *speedMode* multipliziert. Um zurück ins PU-System zu gelangen wird der letzte Faktor noch multipliziert.

$$N_{Ref} = (Speed\_Ref\_PU \cdot \frac{PU\_System.N\_base}{availableSpeedModes}) \cdot speedMode \cdot \frac{1}{PU\_System.N\_base} \tag{5.4}$$

In Abbildung 5.12 ist die Implementierung der Gl. 5.4 dargestellt.

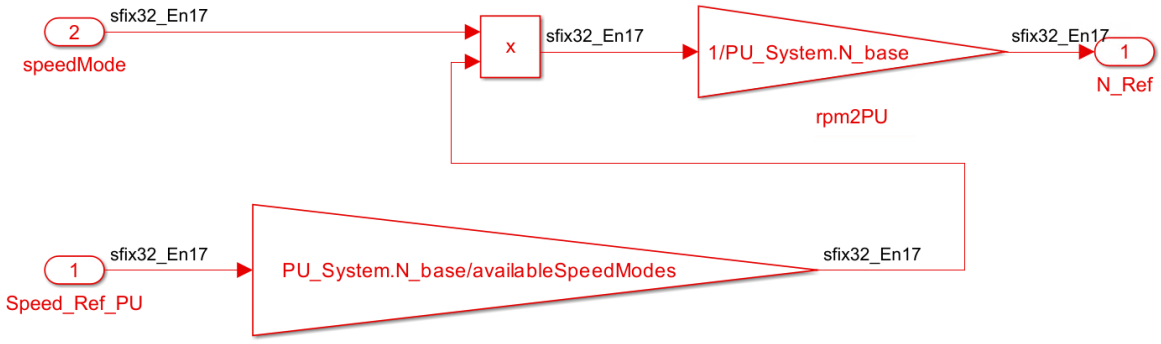


Abbildung 5.12: Implementierung des *Speed calculators*

Es folgt nun ein kleines Beispiel, um das Verständnis für die oben erstellte Gleichung zu steigern. Es werden folgende Koeffizienten betrachtet:

1. $speedMode = 3$
2. $Speed\_Ref\_PU = 0.5$
3. $availableSpeedModes = 3$
4. $PU\_System.N\_base = 4000$

Das Resultat dieser Gleichung liegt bei einer resultierenden Geschwindigkeitsanforderung von 0.5 PU oder 2000 rpm[3]. Um den Einfluss des *speedModes* auf die Umdrehungszahl zu zeigen, wird ein weiteres Beispiel betrachtet.

1. $speedMode = 1$
2. $Speed\_Ref\_PU = 0.5$
3. $availableSpeedModes = 3$
4. $PU\_System.N\_base = 4000$

Eingesetzt in Gl. 5.4 ergibt die Geschwindigkeitsanforderung 0.1667 PU oder $\approx$ 667 rpm. Anhand der beiden Beispiel wird deutlich, welchen Einfluss die Variable *speedMode* besitzt.

Bevor das Subsystem *PI_Controller_Speed* vorgestellt wird, werden zunächst die Subsysteme für die Überprüfung des *Freemodes* beschrieben.

Als nächstes wird das Subsystem *Digital_Freemode_Check* vorgestellt. In Abbildung 5.13 ist das Subsystem zu sehen.

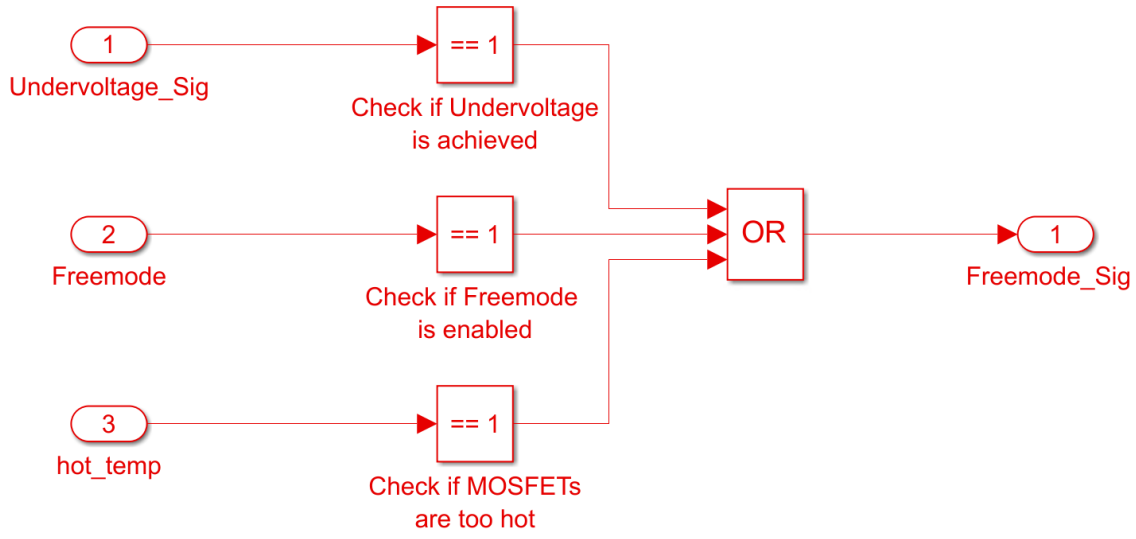


Abbildung 5.13: Implementierung des *Digital_Freemode_Checks*

[3]**R**evolutions **p**er **m**inute

Der *Digital_ Freemode_ Check* überprüft, ob mindestens eine digitale Maßnahme für das Fahren ohne elektrischen Antrieb erfüllt wird. Hierbei werden die folgenden Signale vom Input Controller berücksichtigt:

1. *Undervoltage_ Sig*
2. *Freemode*
3. *hot_ temp*

Anhand der Wahrheitstabelle aus Tab. 5.2 werden die möglichen Eingangs und Ausgangskombinationen dargestellt.

Undervoltage_ Sig	***Freemode***	***hot_ temp***	***Freemode_ Sig***
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Tabelle 5.2: Wahrheitstabelle für den *Digital_ Freemode_ Check*

Somit wird die Variable *Freemode_ Sig* am Ausgang dieses Subsytems auf 1 gesetzt, sobald mindestens ein Eingangssignal auf 1 ist.

Im Anschluss dieses Subsystems erfolgt die Abfrage des GPIOs, welcher für den sicheren Zustand zuständig ist. In Abbildung 5.14 ist der *Mechanical_ Freemode_ Check* zu sehen.

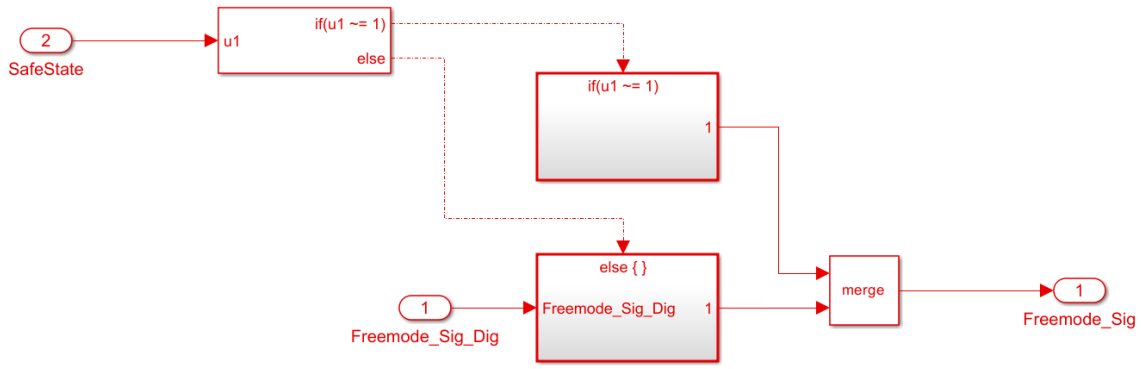


Abbildung 5.14: Implementierung des *Mechanical_ Freemode_ Checks*

Das Ziel ist es hier erneut zu überprüfen, ob die Bedingung für das Fahren ohne elektrischen Antrieb erfüllt ist. Die Redundanz hier ist aufgrund der verbesserten Übersicht auf diese Weise implementiert. Hierdurch können die Abfragen der digitalen Maßnahmen von der mechanischen Maßnahme differenziert werden.

Es wird zunächst überprüft, ob der GPIO für den sicheren Zustand $\neq 1$ ist. Falls diese Abfrage mit *True* beantwortet wird, weist dies auf einen Knopfdruck hin. Somit wird der Ausgang dieses Subsystems auf 1 gesetzt, was das Fahren ohne elektrischen Antrieb aktiviert.

Bei nicht Betätigen des Knopfs wird der *else*-Pfad genommen. Abhängig vom Ausgang des *Digital_ Freemode_ Checks* wird dieser weitergeführt.

Auch hier werden die möglichen Eingangs-/Ausgangskombinationen in Form einer Wahrheitstabelle (siehe Tab. 5.3) festgehalten.

Digital_ Freemode_ Check	***SafeState***	***Freemode_ Sig***
0	0	0
0	1	1
1	0	1
1	1	1

Tabelle 5.3: Wahrheitstabelle für den *Mechanical_ Freemode_ Checks*

Nun folgt in Abbildung 5.15 die Vorstellung des Subsystems *PI_ Controller_ Speed*.

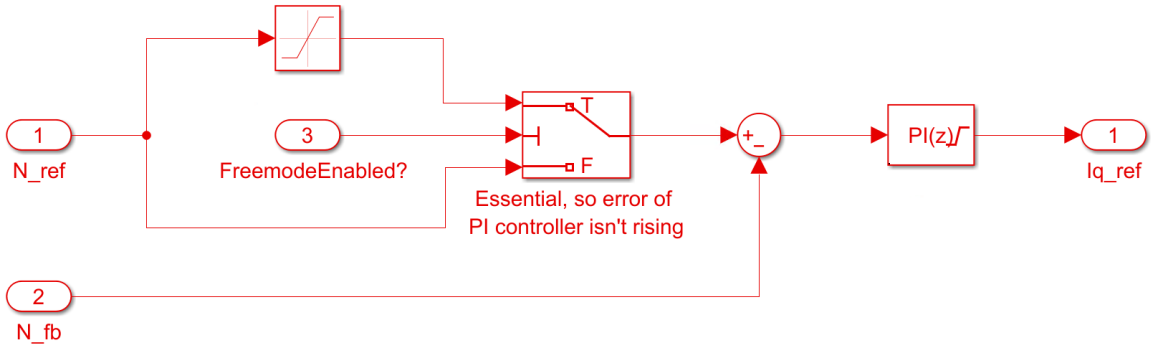


Abbildung 5.15: Implementierung des *PI_ Controller_ Speed*

Der Ausgang des *Mechanical_ Freemode_ Checks* wird an dieser Stelle verwendet. Dieser wird überprüft, um zu entscheiden, ob die angeforderte Geschwindigkeit auf 0 skaliert wird, sodass keine Geschwindigkeitsanforderung möglich ist.

Dieser Wert wird mit der aktuellen Geschwindigkeit subtrahiert und gelangt in den PI-Regler. Da die angeforderte Geschwindigkeit kleiner als die aktuelle Geschwindigkeit ist, wird als Ausgang somit ein Drehmoment ≤ 0 gefordert. Dies hängt jedoch von der Begrenzung des Ausgangs ab. Ist beispielsweise ein aktives Bremsen gewünscht, kann das untere Limit bei -1 liegen. In dieser Arbeit wurde ein leichtes Bremsen mit dem Wert von -0.5 implementiert. Falls kein aktives Bremsmanöver gewünscht ist, kann aufgrund der einfachen Modularität als unteres Limit 0 notiert werden. Hierdurch bremst der Motor durch das Trägheitsmoment und der Reibung ab.

Die Abfrage, ob *Freemode* aktiviert ist, muss hier stattfinden damit die Geschwindigkeitsanforderung nicht mehr möglich ist. Zusätzlich ist aufgefallen, dass ohne diese Abfrage der Stromverbrauch sich bei Stillstand drastisch erhöht hat. Eine mögliche Ursache hierfür kann der sich dadurch steigende *Error* des PI-Reglers sein.

Im weiteren Verlauf dieser Arbeit wird der Grund dieses Verhaltens nicht weiter verfolgt.

Als Resultat wird I_q_Ref an den Ausgang dieses Subsystems gelegt. Gemäß der feldorientierten Regelung ist die Soll-I_d-Stromkomponente für maximales Drehmoment gleich Null. Daher wird I_d_Ref als Konstante 0 und dem korrekten Datentypen (siehe Kapitel 5.1.3) zusammen mit I_q_Ref an einen Mux angeschlossen.

Zusammen mit dem Signal für das Fahren ohne elektrischen Antrieb, werden diese an den Torque Controller weitergegeben.

5.4 Torque Controller

Die Signale der Soll-Stromkomponenten und das Freemode-Signal werden vom Torque Controller verwendet. Der Torque Controller gibt die berechnete aktuelle Geschwindigkeit an den Speed Controller zurück.

An dieser Stelle wird grob auf die Umsetzung der Unter-/ und Überabtastungen des Modells eingegangen. Viele Komponenten besitzen unterschiedliche *sample times*. Damit Signale zwischen Komponenten mit unterschiedlichen *sample times* ausgetauscht werden können, werden sog. *Rate Transition*-Blöcke verwendet. Als Eingang kommt das zu verschickende Signal und am Ausgang wird dieses Signal auf die richtige *sample time* transformiert. In den Parametern dieses Blocks muss die gewünschte *sample time* notiert sein. Ohne diese funktioniert die Unter-/ oder Überabtastung nicht.

5.4.1 Hardware-Interrupt

Im Pfad *Model Configurations* können mit dem *Hardware Mapping*-Konfigurator die jeweiligen Hardware Interrupt-Blöcke konfiguriert werden. Ein Beispiel des *Hardware Mappings* ist in Abbildung 5.16 zu sehen.

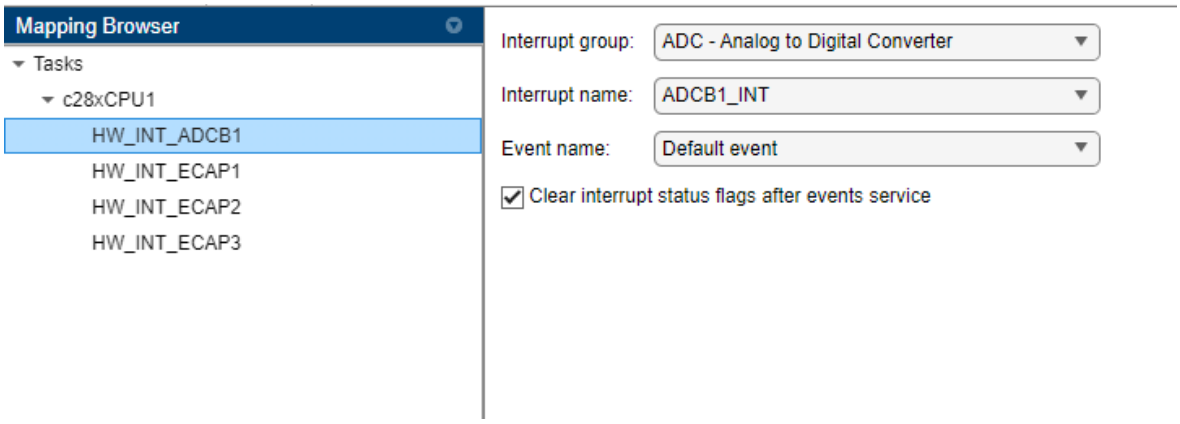


Abbildung 5.16: Übersicht vom *Hardware Mapping*-Konfigurator

Der ADC-Interrupt, welcher den Torque Controller über einen Funktionsaufruf in der ISR aufruft, ist ein *preemtable* Interrupt. Das bedeutet, dass dieser Interrupt pausiert, wenn ein anderer Interrupt (mit einer höheren Priorität) aufgerufen wird. Des Weiteren ist dieser ADC-Interrupt mit dem ADC-Modul B verbunden, was für den nachfolgenden Teil der Arbeit relevant ist.

Die eCAP-Interrupts besitzen alle die gleiche Priorität und sind alle *non-preemtable*. Wenn die eCAP-Interrupts unterbrochen werden würden, könnte es zu falschen Geschwindigkeitsmessungen kommen. Dies begründet auch die Vergabe der Prioritäten der Interrupts.

In Tabelle 5.4 ist die Übersicht der Interrupts, mit den Prioritäten und der Eigenschaft, ob diese *preemtable* sind dargestellt. Je kleiner die Zahl der Priorität, desto höher die Priorisierung des Interrupts.

Interrupt Name	**Priorität**	***Preemtable?***
ADCB1	30	Ja
eCAP1	25	Nein
eCAP2	25	Nein
eCAP3	25	Nein

Tabelle 5.4: Übersicht der Hardware Interrupts

5.4.2 Initialisierung

Beim Initialisieren wird der *System Initialize*-Block ausgeführt. Ein Auszug von Inhalt dieses Blocks ist in Abbildung 5.17 zu sehen.

Zu sehen sind die Aktivierungen sämtlicher interner Pull-Up Widerstände. Die GPIOs 56-61 sind aufgrund eines undefinierten Zustands bei Fingerberührung am *springen*. Aus diesem Grund werden die internen Pull-Up Widerstände aktiviert.

Wie in Kapitel 5.2 erwähnt, ist der GPIO für den sicheren Zustand auch mit einem Pull-Up Widerstand auf die Versorgungsspannung gezogen.

System Initialize Function Execution Code

```
EALLOW;
//Pull up for SafeState GPIO
GpioCtrlRegs.GPAPUD.bit.GPIO30 = 0;
//Pull up for GPIOs
GpioCtrlRegs.GPBPUD.bit.GPIO56 = 0;
GpioCtrlRegs.GPBPUD.bit.GPIO57 = 0;
GpioCtrlRegs.GPBPUD.bit.GPIO58 = 0;
GpioCtrlRegs.GPBPUD.bit.GPIO59 = 0;
GpioCtrlRegs.GPBPUD.bit.GPIO60 = 0;
GpioCtrlRegs.GPBPUD.bit.GPIO61 = 0;
EDIS;
```

Abbildung 5.17: Auszug vom *System Initialize*-Block

Des Weiteren wird bei der Initialisierung der ADC-Offset gemessen. Die Wichtigkeit des ADC-Offsets wird in [36] erläutert. Da der von Simulink entwickelte Algorithmus [37] zur Messung des ADC-Offsets bereits ein gutes Verhalten aufweist, wurde dieser im Implementierungsmodell (siehe Anhang A.1.6) verwendet.

5.4.3 Rotorpositionsbestimmung

Wie in Kapitel 2.3 erwähnt, muss die Lage des Rotors für eine optimale Regelung bekannt sein. Die Rotorpositionsbestimmung erfolgt mithilfe der im Motor integrierten Hall-Sensoren [34].

Hierfür wird bei jedem Flankenwechsel der zugehörige eCAP Interrupt aufgerufen. Die nachfolgende Abbildung 5.18 visualisiert das beschriebene Verfahren.

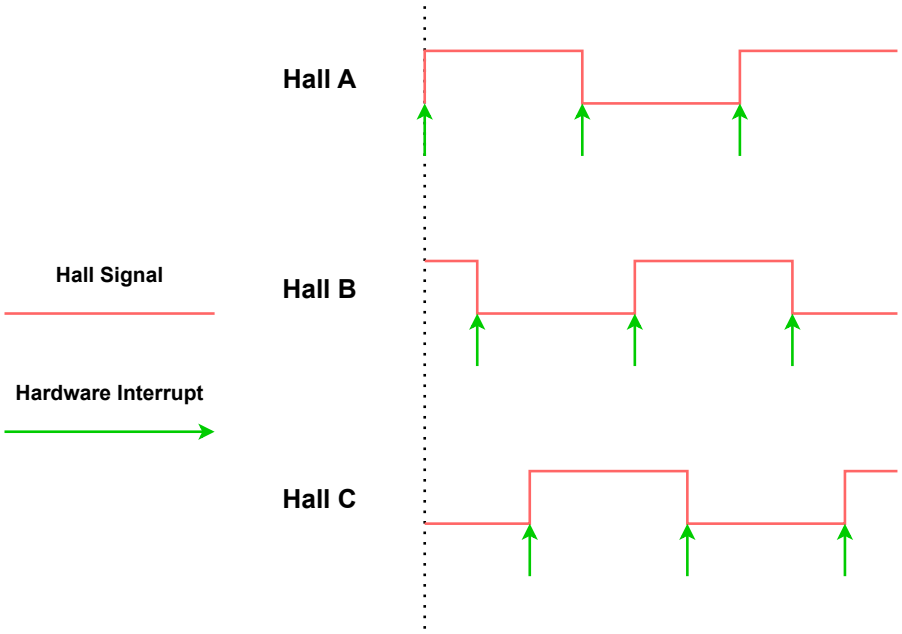


Abbildung 5.18: eCAP-Interrupt bei jedem Flankenwechsel der Hall-Signale

Der Algorithmus, der die Position und Geschwindigkeit bestimmt, ist im Torque Controller implementiert. Dies ist erforderlich, da bei jedem ADC-Interrupt die aktuellen Werte für Position und Geschwindigkeit benötigt werden.

Beim verwendeten Algorithmus wurde sich an einem Beispiel [38] von Simulink orientiert.

Wenn ein Flankenwechsel eintritt und somit ein eCAP-Interrupt ausgelöst wird, wird der Wert des eCAP-Zählers an den *Hall Validity*-Block gegeben. Zusätzlich werden die Drehrichtung des Motors, der aktuelle und letzte *Hall State* an den *Hall Validity*-Block gegeben. Dies wird durch sämtliche *Data Store Read/Memory/Write*-Blöcke realisiert. Diese Blöcke werden dazu verwendet, den Wert der Variablen zu lesen oder zu überschreiben. Hierdurch können viele Block-Verbindungen eliminiert werden und das Modell sieht übersichtlicher aus.

Die beschriebene Implementierung ist in Abbildung 5.19 dargestellt.

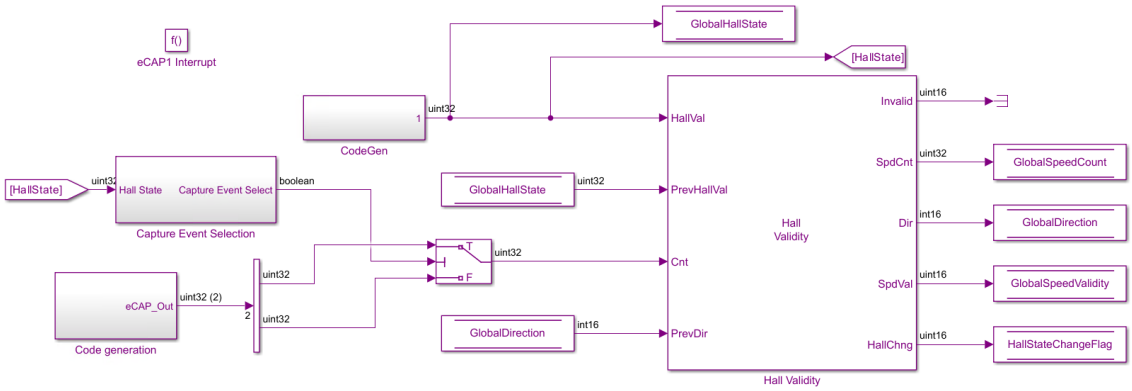


Abbildung 5.19: Übersicht des eCAP-Interrupt für Hall A

In Abbildung 5.20 ist die Pin-Zuweisung der jeweiligen eCAP-Module zu sehen.

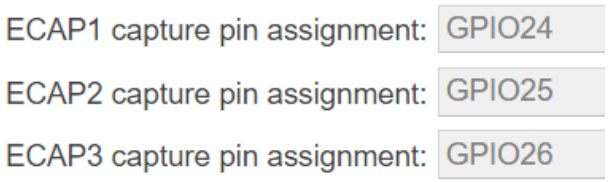


Abbildung 5.20: Pin-Zuweisung der eCAP-Module

Die folgende Auflistung zeigt die mit den Hall-Signalen verbundenen GPIOs:

1. GPIO24 - Hall A
2. GPIO25 - Hall B
3. GPIO26 - Hall C

Wie zu sehen ist, sind die eCAP-Module mit den dazugehörigen Hall-Signalen verbunden. Hierdurch wird gewährleistet, dass die eCAP-Interrupts bei den richtigen Flankenwechsel ausgelöst werden. Diese Annahme wird in Kapitel 7.2 verifiziert.

Damit der *Hall Validity*-Block den aktuellen Zustand der Hall-Signale lesen kann, muss das jeweilige Register im Controller ausgelesen werden und mithilfe von bitweise Operationen umgeformt werden.

Das Register `GpioDataRegs.GPADAT` beobachtet den aktuellen Zustand des jeweiligen GPIOs. Dieses Register ist zuständig für GPIO 0-31.

Nachdem der Zustand von den GPIOs 24, 25, 26 gelesen wird, erfolgt zunächst eine *Bitwise AND* Operation und Verschiebung der Bits. Am Ende wird Hall A als MSB[4] und Hall C als LSB[5] sortiert sein.

In Abbildung 5.21 ist dieser Ansatz visualisiert.

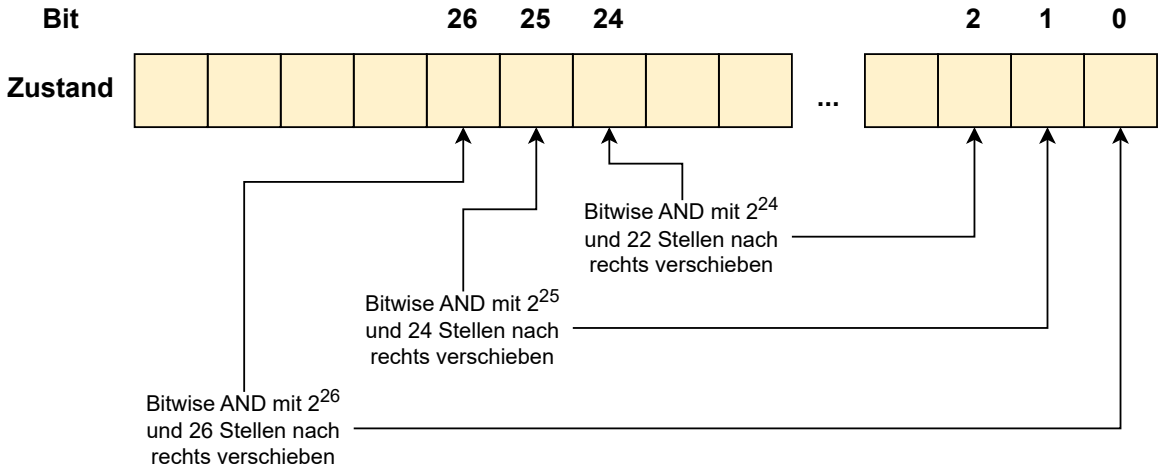


Abbildung 5.21: Ermittlung und Sortierung der Hall-Zustände

Das Subsystem *Calculate Position from Hall* im Torque Controller kalkuliert die Position und die Geschwindigkeit mithilfe des *Hall Speed and Position*-Blocks.

Die Ermittlung der Hall-Zustände erfolgt dabei auf die gleiche Art wie in Abbildung 5.21 dargestellt.

Des Weiteren werden, neben dem Auslesen der *TSCTR*-Register, für die aktuellen Zählwerte verschiedenste *Data Store Read*-Blöcke, welche die Drehrichtung, Plausibilitätsüberprüfungen beinhalten, als Inputs für den *Hall Speed and Position*-Block verwendet.

[4] **M**ost **S**ignificant **B**it
[5] **L**east **S**ignificant **B**it

Damit Geschwindigkeiten größer als die Nenndrehzahl möglich sind, kann ein Glitchfilter von Simulink [37] verwendet werden. In dieser Arbeit sorgt der Glitchfilter jedoch für ein sanfteres Drehen des Motors. Daher wird dieser auch weiterhin verwendet.

Das Problem bei dem bisher beschriebenen Verfahren ist, dass der Offset der Hall-Sensoren nicht berücksichtigt wurde. Ohne diesen werden die entstehenden Geräusche und Stromverbrauch deutlich erhöht. Daher erfolgt am Ausgang der berechneten Position ein *Mechanical to Electrical Position*-Block. In diesem ist wichtig die Polpaarzahl bei 1 zu lassen, da der *Hall Speed and Position*-Block bereits die Polpaarzahl enthält. Hier wird der Hall-Offset eingetragen.

In dieser Arbeit war der Parameter des Hall-Offsets unbekannt. Daher wurde sich diesem mit Zwischentest angenähert. Es besteht jedoch die Möglichkeit den Hall-Offset mit einem Simulink-Modell [39] zu berechnen. Dieses wurde aufgrund fehlender Zeit nicht durchgeführt.

5.4.4 Strommessung

Wenn der ADC-Interrupt ausgelöst wird, ruft dieser den Torque Controller auf. Beim Starten werden zuerst die Strangströme gemessen. Die Strommessung wird mit den 12-bit ADC's vom B-Modul durchgeführt, da der Trigger die Messungen vom ADC Modul B startet.

Da die Shunt-Widerstände der Leistungsstufe an der Low-Side angeschlossen sind, muss der Auslöser des ADC-Interrupts während des High-Pegels der Low-Side stattfinden. Eine Visualisierung ist in Abbildung 5.22 zu sehen.

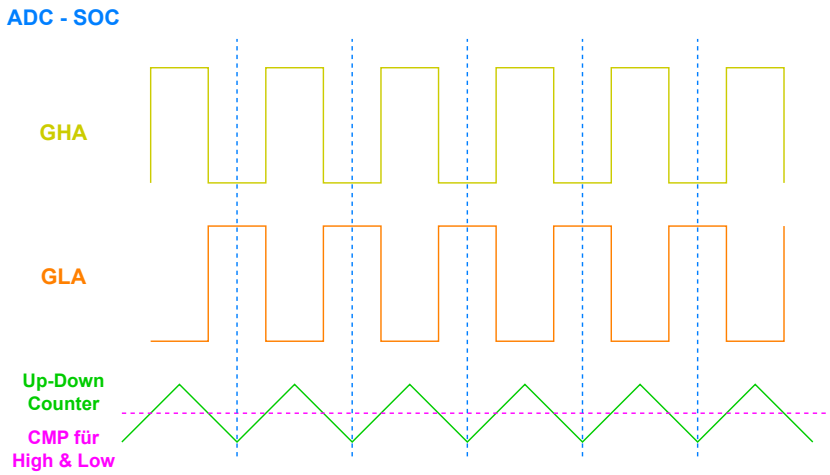


Abbildung 5.22: Übersicht des Zeitpunkts vom ADCB1-Interrupt

In gelb ist die High-Side und in orange die Low-Side der Leistungsstufe zu sehen. Der Trigger für den ADC-Interrupt muss ausgelöst werden, wenn die Low-Side einen High-Pegel besitzt. Darüber hinaus ist in grün der PWM-Zähler dargestellt. Beim Erreichen des CMP[6]-Werts während des hoch-/ oder runterzählen wird der Ausgang auf die Versorgungsspannung oder auf GND gezogen. Je nachdem ob der PWM-Zähler hoch-/ oder runterzählt.

Wie in Kapitel 2.1 erwähnt, ist die Wahl des *SOCx acquisition windows* ausschlaggebend dafür, wann die Messung angefangen wird. Ist der Wert zu klein oder zu groß, kann es zu falschen Werten kommen, sodass sich der weitere Programmablauf verfälscht.

In Simulink kann im ADC-Block das *Acquisition Window* eingestellt werden. Der Parameter *SOCx acquisition window* bestimmt, wie in Kapitel 2.1 erwähnt, die Breite des (Auflade-)Fensters, wo das Signal die volle Stärke aufbaut. Jedoch wird hier die Anzahl der Taktzyklen notiert und nicht die zeitliche Dauer, die umgesetzt wird.

In Gl. 5.5 erfolgt die Berechnung der Breite des *SOCx acquisition windows*.

$$AcquisitionWindow = (ACQPS^{7} + 1) * (SYSCLK^{8} cycleTime) \tag{5.5}$$

[6]**C**ounter **Comp**are
[7]**Acq**uisition **P**rescaler
[8]**Sys**tem **C**lock

SYSCLK beträgt 200 MHz, sodass eine Periodendauer von 5 ns resultiert. Die Eingabe im ADC-Block unter *SOCx acquisition window* repräsentiert den Term *ACQPS+1*. Wenn hier beispielweise eine 10 notiert wird ergibt das ein *Acquisition Window* von 50 ns.

Laut dem Datenblatt [40] muss die Breite des *Acquisition Windows* mindestens 1 ADCCLK[9] Taktzyklus lang sein, um eine korrekte ADC-Operation zu gewährleisten. Es wird ein ADCCLK von 50 MHz verwendet, was zu einem Taktzyklus von 20 ns führt. Weitere Anforderungen wie zum Beispiel in [41] sagen, dass mindestens eine Breite des *Acquisition Windows* von 75 ns vorhanden sein muss.

Aus diesem Grund wird in dieser Arbeit ein Wert von 15 beim Parameter *SOCx acquisition window* verwendet, was zu einer Breite des *Acquisition Windows* von 75 ns führt. Hierdurch werden beide Anforderungen erfüllt.

Die gemessenen ADC-Werte werden dann vom Subsystem *Convert ADC value to PU* im *Input Scaling*-Block ins PU-System umgewandelt. Die Umrechnung ist in Abbildung 5.23 zu sehen. Die farbliche Darstellung zeigt erneut die *sample time*. Da der Torque Controller über einen Funktionsaufruf in der ISR aufgerufen wird, besitzen alle Komponenten im Torque Controller eine vererbte (*engl.: inherit*) *sample time*. Das bedeutet dass die Ausführung asynchron stattfindet und vom Auslöser (hier: das PWM-Modul) abhängig ist.

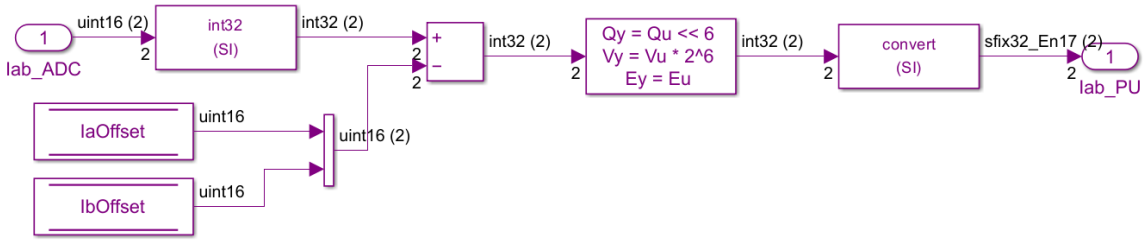


Abbildung 5.23: Umrechnung der Strangströme ins PU-System

Zunächst werden die gemessenen ADC-Werte mit dem Offsetwert (siehe Kapitel 5.4.2) des jeweiligen ADC-Kanals subtrahiert, sodass auch negative Stromwerte möglich sind.

[9] **A**nalog-to-**D**igital **C**onverter **C**lo**k**

Hierdurch wird (bei einer stromlosen Umgebung) ein Wert von 0 erreicht. Die Multiplikation mit 2^6 steigert die Genauigkeit. Abschließend erfolgt mit dem *Data Type Conversion*-Block die Änderung des Datentyps in eine Festkommadarstellung (siehe Kapitel 5.1.3).

5.4.5 Feldorientierte Regelung

In Abbildung 5.24 ist implementierte die Feldorientierte Regelung zu sehen. Aufgrund der Überschaubarkeit des Modells, wird nicht auf die einzelnen Transformationen eingegangen.

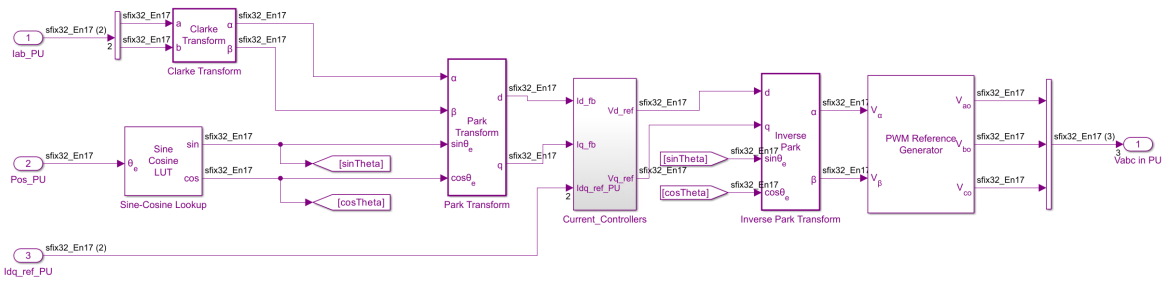


Abbildung 5.24: Übersicht des FOCs

Die Lookup-Tabelle für die Position des Motors muss aufgrund des verwendeten PU-Systems auch auf diesen konfiguriert werden.

5.4.6 Erzeugung der PWM

Die Raumzeigermodulation (siehe [42]) gibt Signale im Wertebereich zwischen -1 und 1 aus. Damit die berechneten Raumzeiger angewendet werden können, müssen diese noch zuvor auf 0 bis 1 skaliert werden. Die Problematik an dieser Stelle ist jedoch, dass Texas Instruments und Simulink zwei unterschiedliche positive Drehrichtungen besitzen. Daher müssen die Signale zusätzlich noch negiert werden, wie im Subsystem *Scaling DutyCycles* zu sehen ist.

Die skalierten Duty Cycles werden vor der Benutzung noch bei der Abfrage des *Freemodes* verwendet. Wenn das Fahren ohne elektrischen Antrieb ausgewählt wurde bzw.

eine der in Kapitel 5.2 erwähnten Bedingungen erfüllt ist, ist die Variable Freemode auf 1 gesetzt. Das führt dazu, dass der Motorcontroller PWM-Signale an den Gatetreiber mit einem Tastverhältnis von 0% senden wird.

Aufgrund der Tatsache, dass der verwendete Gatetreiber [43] invertierend ist, muss der Ansatz der PWM-Erzeugung mit 0% Tastverhältnis anders gelöst werden. In Abbildung 5.25 ist das Verhältnis der Eingangssignale (INHx,INLx) zu den Ausgangssignalen (GHx,GLx) zu sehen.

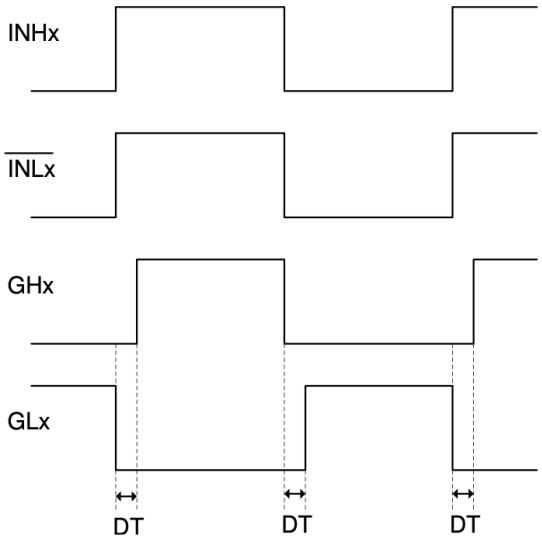


Abbildung 5.25: Invertierende INLx Inputs [43]

Da der verwendete Gatetreiber invertierend ist, erfolgt bei *wahrer* Aussage des *Freemodes* die PWM-Erzeugung wie in Abbildung 5.26 dargestellt.

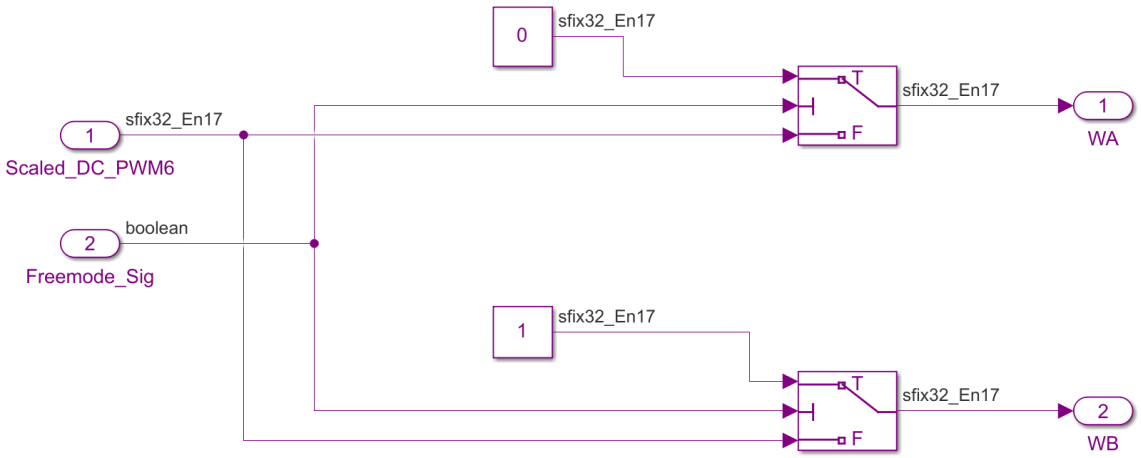


Abbildung 5.26: PWM-Erzeugung in Abhängigkeit vom *Freemode*

Der Ausgang *WA* ist für die High-Side verantwortlich und *WB* für die Low-Side der Leistungsstufe.

Wenn *Freemode* auf 1 gesetzt wurde und somit *wahr* ist, werden die Konstanten Werte verwendet. Bei *falscher* Aussage, werden die skalierten Duty Cycles zu dem jeweiligen Ausgang geführt.

Damit die erzeugten skalierten Duty Cycles von den ePWM-Blöcken ausgewertet werden können, muss der Zahlenbereich stimmen. Hiermit wird die Skalierung auf den Wert vom *Timer Period* nötig.

Der *Timer Period* ist die Anzahl der Taktzyklen, welche für eine Periode bei einem 20 kHz PWM-Signal gebräuchlich ist [11].

Die Gl. 5.6 zeigt, dass für eine PWM-Frequenz von 20 kHz bei einer CPU clock Frequenz von 200 MHz eine *Timer Period* von 5000 nötig ist.

$$PWMTimerPeriodCounter = \frac{CPUclockfrequency}{PWMfrequency} \cdot 0.5 \tag{5.6}$$

Der *EPWMCLKDIV* im Menü für die ePWM-Module in den *Hardware Settings* muss auf *SYSCLKOUT/1* geändert werden, um mit einer *Timer Period* von 5000 eine PWM mit der Frequenz von 20 kHz zu erzeugen.

Die PWM-Signale werden *center aligned* erzeugt. *Center aligned* bedeutet, dass das Zentrum (Mitte) der PWM-Signale der 3 Halbbrücken an der gleichen Stelle ist.

Mittig ausgerichtete PWMs werden am häufigsten bei der Steuerung von Wechselstrommaschinen verwendet, um die Phasenausrichtung beizubehalten [44]. In dieser Arbeit wird die *Center Aligned* PWM-Erzeugung aufgrund ihrer einfachen Implementierung [45] verwendet.

Damit die Leistungsstufe die Gate-Signale ohne Fehlinterpretationen vorbereiten kann, werden zwei identische Signale ohne Totzeit an die PWM-Ausgänge des Controllers gegeben.

In den ePWM-Blöcken wird deshalb bei den ePWMxA/ePWMxB Kanälen darauf geachtet, dass das gleiche Verhalten beim Hoch/- und Runterzählen gewährleistet wird. In Abbildung 5.27 ist ein Beispiel für das ePWM-Modul 6A zu sehen. Die Einstellungen müssen für das ePWM-Modul 6B beim CMPB-Wert identisch sein.

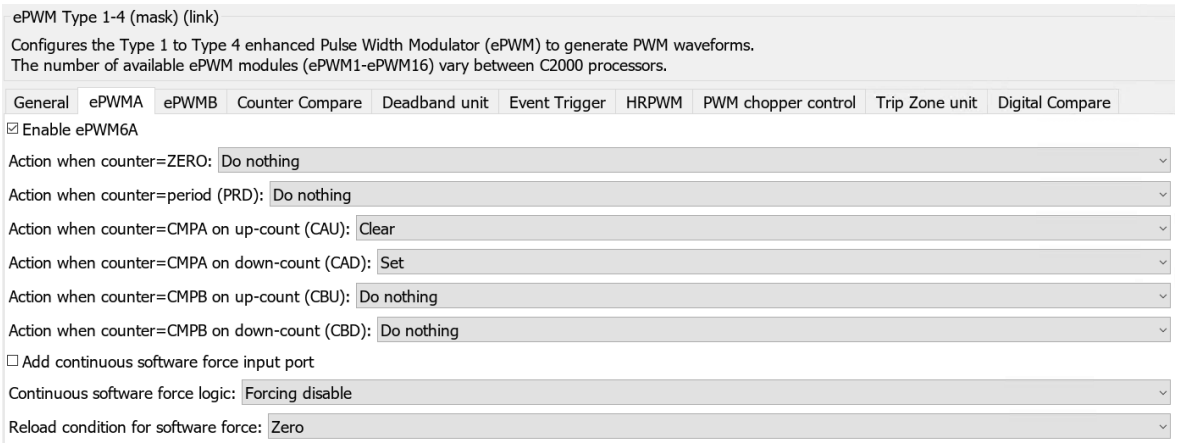


Abbildung 5.27: Konfiguration des ePWM-Moduls 6A

5.5 Modellvarianten

Um die Vorteile der modellbasierten Entwicklung zu nutzen, werden im Implementierungsmodell (siehe Anhang A.1.6) die *Variant*-Blöcke verwendet. Mithilfe dieser kann zwischen Modellvarianten unterschieden werden.

Im Implementierungsmodell wird zwischen *Simulation* und *Code Generation* unterschieden. Wenn das Modell auf das Target hochgeladen wird, werden im Modell die *Variant*-Blöcke automatisch auf die *Code Generation*-Variante gestellt und somit Code auf das Target hochgeladen.

Im Gegensatz sorgt das Simulieren über den *Simulation*-tab dazu, dass die *Variant*-Blöcke auf *Simulation* gestellt werden.

Eine Übersicht ist am Beispiel eines *Variant Sink*-Blocks in Abbildung 5.28 dargestellt.

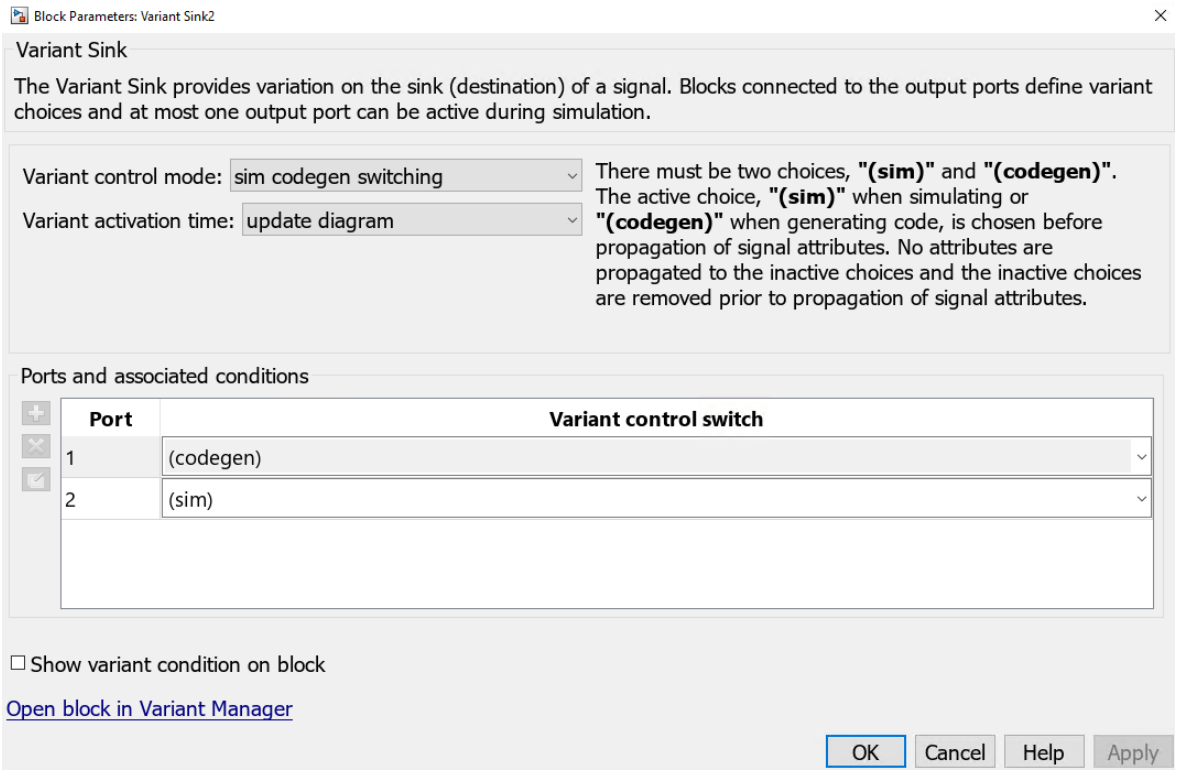


Abbildung 5.28: Übersicht eines *Variant Sink*-Blocks

5.6 Streckenmodell

In dieser Arbeit wurde die Hardware in Simulink virtuell erstellt. Hierunter fallen besonders:

- Die Erstellung des Motors

- Die Erstellung einer Längsdynamik
- Die Implementierung der ADC Gains der Leistungsstufe
- Weitere Parameter der Leistungsstufe und des Mikrocontrollers

Hierfür wurde an dieser Stelle ein von Simulink entwickeltes Streckenmodell [37] verwendet und auf die richtigen Parameter angepasst.

Wie bereits in Kapitel 5.1.4 erwähnt, konnten die exakten Motor-Parameter nicht verwendet werden. Stattdessen wurden die Werte aus [2] und des Datenblatts [34] übernommen. Daher kann es zu Abweichungen zwischen Simulink und der echten Anwendung kommen.

In Abbildung 5.29 ist das verwendete Streckenmodell zu sehen.

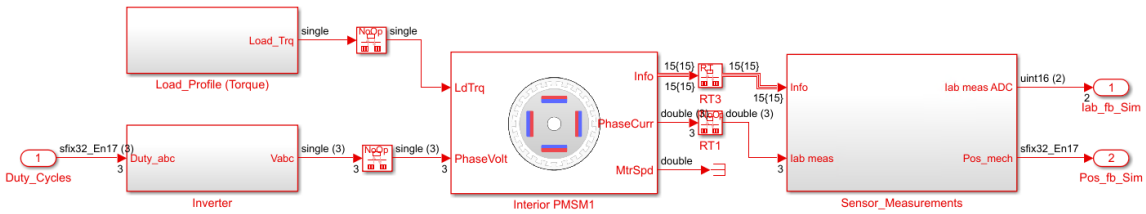


Abbildung 5.29: Übersicht des Streckenmodells

Wie zu sehen ist, gelangen die skalierten Duty Cycles in den Inverter. Der Motor wird sich durch die vom Inverter kommenden Strangspannungen (virtuell) drehen.

Um eine Längsdynamik mit ins Implementierungsmodell (siehe Anhang A.1.6) zu integrieren, wird eine Last verwendet. Da der Motor in dieser Arbeit nicht in eine Applikation eingebaut wird, ist kein Lastwechsel vorhanden. Daher bleibt die Last konstant.

In Abbildung 5.30 sind die Messungen im Streckenmodell zu sehen. Hierbei werden die gemessenen Ströme (in Ampere) Schritt für Schritt in einen ADC-Wert umgerechnet.

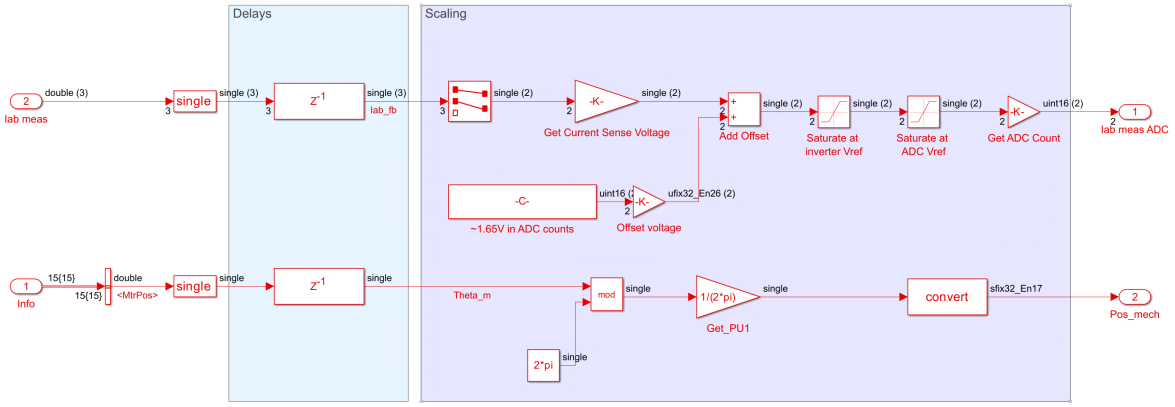


Abbildung 5.30: Positions-/ und Strommessung im Streckenmodell

Zum Beispiel realisiert der Gain-Block (*Get Current Sense Voltage*) den Shunt-Widerstand auf der Leistungsstufe. Dieser wird mit den weiteren Berechnung auf einen Wertebereich eines 12-bit ADCs begrenzt.

5.7 External Mode

Die Implementierung des External Modes über TCP wurde mit Anpassungen der erstellten Software-Architektur (aus Kapitel 4.5) entwickelt.

Die Architektur sah vor, dass auf dem Target ein Server eingerichtet wird und mit einem Host-Computer als Client auf diesen Server zugegriffen wird.

Der verwendete Controller verfügt über mehrere CPUs und einem sog. CM[10]. Die Ethernet-Schnittstelle ist nur über den CM zugänglich. Da das Implementierungsmodell (siehe Anhang A.1.6) auf CPU1 laufen wird, müssen die Signale von CPU1 zum CM gelangen.

Damit die Signale zum CM verschickt werden können, ist ein IPC[11]-Channel nötig. Mithilfe des IPC-Channels können Daten zwischen Prozessoreinheiten ausgetauscht werden.

[10] **C**onnectivity **M**anager
[11] **I**nter-**P**rocess **C**ommunication

Am Beispiel der Hall-Signale und *Hall States* wird das Versenden über dem IPC-Channel beschrieben. Um Block-Verbindungen zu sparen werden an dieser Stelle *Data Store Write/Read*-Blöcke verwendet. Die Signale werden im Subsysten *IPC Write for eCAP INTs* versendet.

Vor der Versendung werden die Signale in den entsprechenden Datentyp (hier: uint32) umgewandelt. Auf der Empfängerseite kann ausgewählt werden, welcher Datentyp erwartet wird. Ein Einblick ist in Abbildung 5.31 zu sehen.

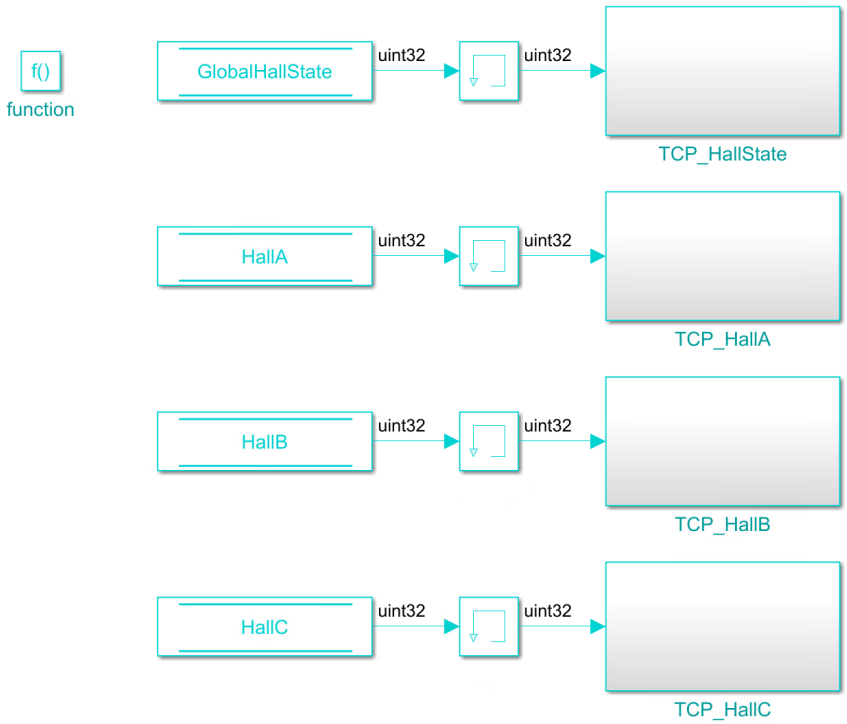


Abbildung 5.31: Implementierung der Versendung der Signale über dem IPC-Channel

Der Server für die TCP Kommunikation befindet sich auf dem CM. Um die vom Implementierungsmodell empfangenen Signale richtig zu visualisieren, müssen die beim Senden benutzten Konvertierung rückgängig gemacht werden. In Abbildung 5.32 ist der Empfang der Signale von CPU1 mit sämtlichen *Interprocess Data Read*-Blöcken realisiert.

Nach der Rückkonvertierung der jeweiligen Signale werden diese mithilfe des *TCP Send*-Blocks versendet.

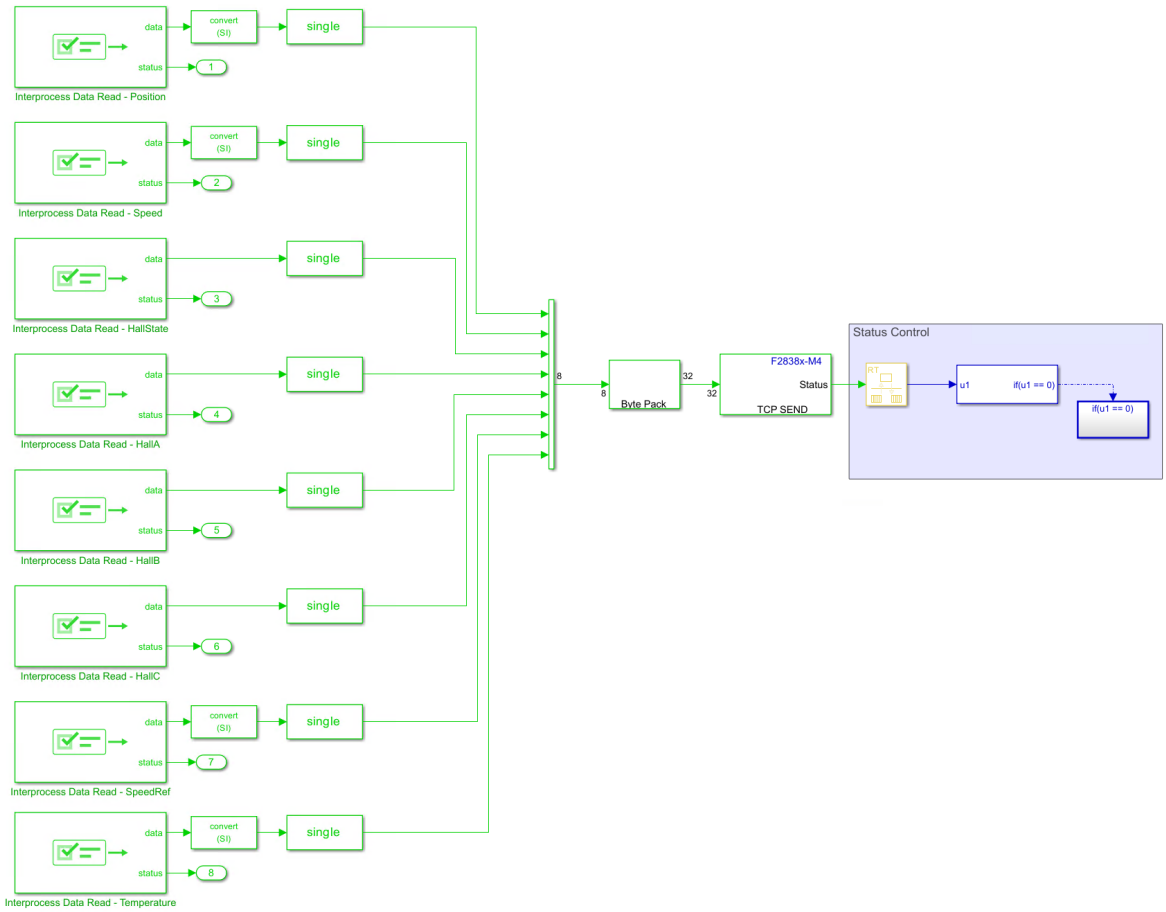


Abbildung 5.32: Implementierung des TCP Servers

In dem Modell des Servers ist zusätzlich eine Status Kontrolle implementiert. Wenn das Senden der Daten auf dem TCP-Protokoll fehlerfrei erfolgt, wird die mit GPIO 57 verbundene LED[12] anfangen zu blinken. Wenn die LED nicht blinkt ist die Kommunikation fehlerbehaftet.

Im Implementierungsmodell können für verschiedene Signale an verschiedenen Stellen *Interprocess Data Write*-Blöcke platziert werden. Um die einzelnen Channels unterscheiden zu können, müssen sowohl der *Interprocess Data Write*-Block als auch der *Interprocess Data Read*-Block die gleiche Channel Nummer besitzen. In dieser Arbeit sind die folgenden Signale über den External Mode zu sehen:

- Position des Motors

[12] **L**ight-**E**mitting **D**iode

- Geschwindigkeit des Motors
- Die verschiedenen *Hall States*
- Signal von Hall A
- Signal von Hall B
- Signal von Hall C
- Angeforderte Geschwindigkeit
- MOSFET-Temperatur

Die detailliertere Übersicht der versendeten und empfangenen Signale kann aus Anhang A.1.13 entnommen werden. Hierbei werden die versendeten und empfangenen Signale mit den jeweilige Datentypen und weiteren Informationen wie *sample times* vorgestellt.

An dieser Stelle befinden sich die Signale, welche visualisiert werden, auf dem CM. Da die Konfiguration des Servers und Clients fehlt, wird der External Mode mit diesem Aufbau nicht funktionieren.

Da der Server auf dem CM eingerichtet wird, muss zunächst in den *Hardware Settings* die korrekte *Processing Unit* ausgewählt werden.

In Abbildung 5.33 wird der Ethernet-Schnittstelle vom CM eine IP-Adresse zugewiesen. Da die MAC-Adresse in dieser Arbeit irrelevant ist, werden alle Parameter (bis auf die MAC-Adresse) für die nächsten Schritte benötigt.

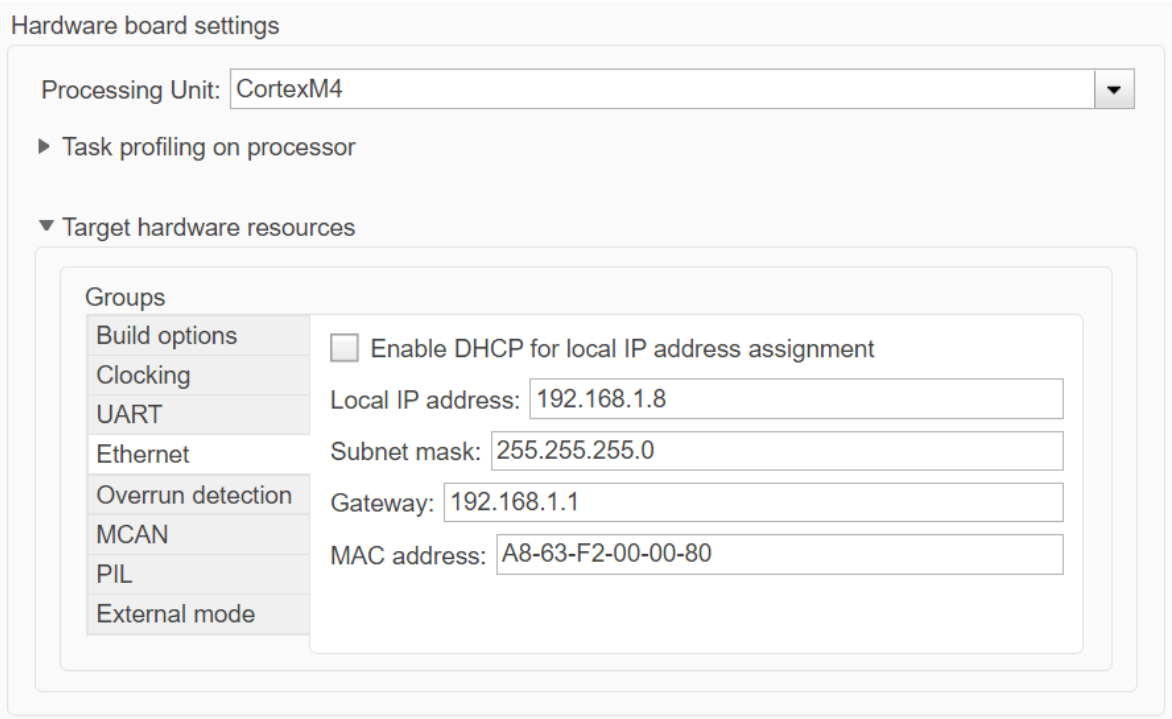


Abbildung 5.33: Vergabe der IP-Adresse in den *Hardware Settings*

Der Server muss eine *Port Nummer* im *TCP Send*-Block angeben. Außerdem ist es wichtig den *Connection Mode* auf *Server* zu ändern. Hierdurch ist ein Server auf dem CM eingerichtet.

Im nächsten Schritt wird der Client eingerichtet.

Wie in Kapitel 4.5 erwähnt, wird mithilfe der IP-Adressen ein Anfangs- und Endpunkt definiert, sodass der Aufbau in Abbildung 5.34 realisiert wird. Der rote Bereich repräsentiert den Server, welcher auf dem Target läuft und der grüne Bereich den Host Computer, welcher als Client fungiert.

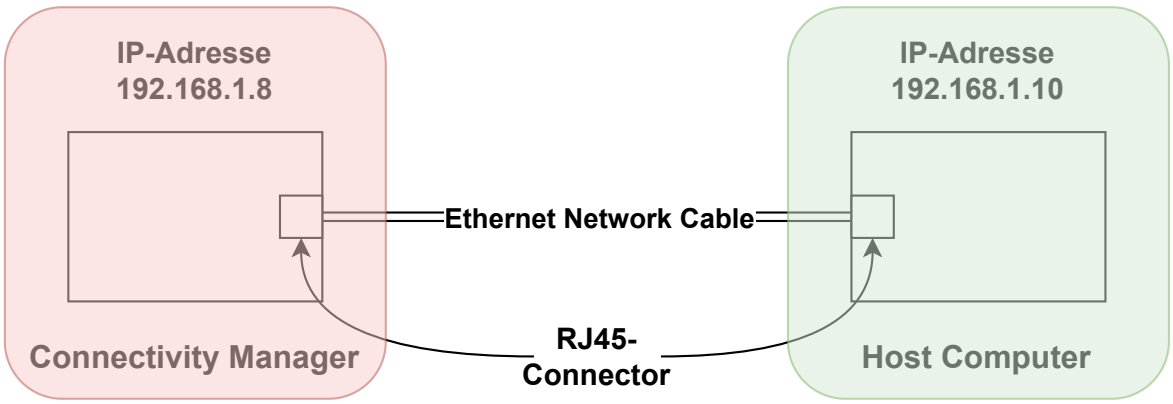


Abbildung 5.34: Übersicht der IP-Adressen

Da der Client und der Server direkt verbunden sind, müssen diese im selben Subnetz liegen. Ansonsten kann keine Verbindung hergestellt werden. Hierfür muss in den Netzwerkeinstellungen des ausgewählten Netzwerkadapters eine IP-Adresse vergeben werden.

In Abbildung 5.35 kann die ausgewählte IP-Adresse, die Subnetzmaske und der Standardgateway entnommen werden. Da in diesem Setup kein Router/Gateway existiert, ist der Wert nicht ausschlaggebend.

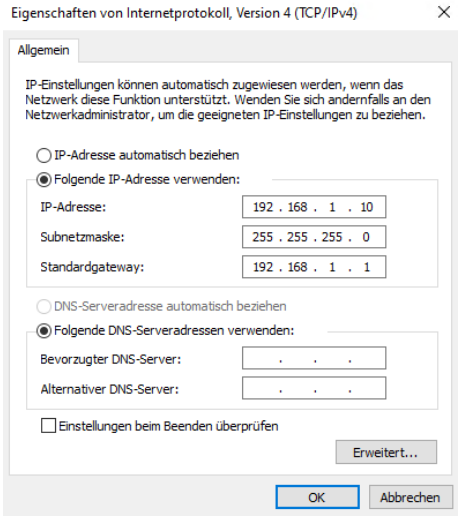


Abbildung 5.35: Vergabe der IP-Adresse in den Adapteroptionen auf dem Host-Computer

Darüber hinaus ist die Implementierung des Clients in Abbildung 5.36 zu sehen.

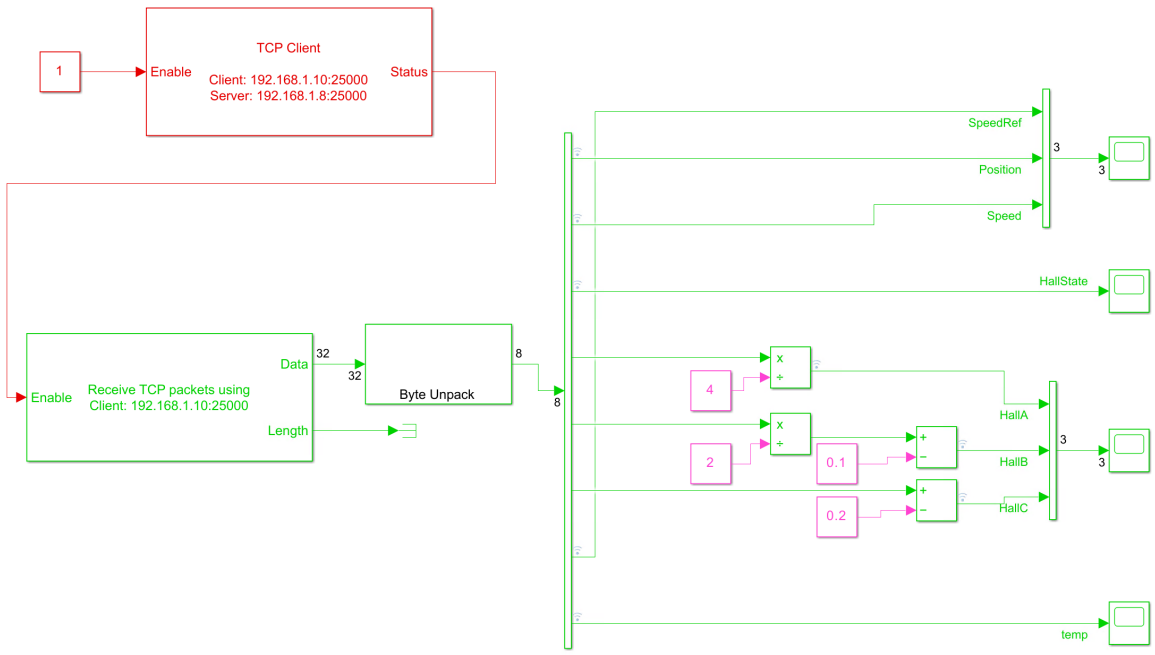


Abbildung 5.36: Implementierung des Clients

Die Erstellung eines Clients erfolgt mit dem *TCP Client*-Block. Hier wird die IP-Adresse aus den Einstellungen wie in Abbildung 5.35 vergeben. Die Port Nummer vom TCP Server aus Abbildung 5.32 muss auch hier vergeben werden.

In Abbildung 5.32 und Abbildung 5.36 sind *Byte (Un-/)Pack*-Blöcke vorhanden. Diese sind für die jeweiligen Umwandlungen nötig. Über das TCP-Protokoll können nur Signale als *uint8* verschickt werden. Damit die Eingangssignale nicht ihre Integrität verlieren, werden mit dem *Byte Pack*-Block das *single*-Signal in ein *uint8*-Signal konvertiert. Dieses Signal wird über das TCP-Protokoll verschickt und auf der Seite des Clients wieder zurück in den ursprünglichen Datentypen konvertiert. Dieses ermöglicht der *Byte Unpack*-Block.

Mehr zu diesem Thema kann in [46, 47] gelesen werden.

Aufgrund der unterschiedlichen Bit-Positionen müssen die Hall Signale noch dividiert werden, um den gleichen Pegel von 1 zu gewährleisten. Damit eine saubere Übersicht vorhanden ist, werden die Hall-Signale zueinander (in der Y-Achse) verschoben. Im Scope liegen diese Signale ohne die Subtraktionen aufeinander und es ist nicht klar, ob das Signal einen High-/ oder Low-Pegel besitzt.

Zwischentests haben gezeigt, dass diese Implementierung bis hier auf der controlCARD funktioniert, aber nicht mit der Hardware aus [4]. Der Grund dieses Verhaltens liegt in der unterschiedlichen Pin-Belegung der zwei Targets [4, 25].

Im Anschluss wurde der generierte Code systematisch zurückverfolgt. Hierbei wurden die generierten Header und C-Code Dateien untersucht, bis die Initialisierung der jeweiligen GPIOs gefunden wurde.

In der generierten Datei `MW_c28xx_board.c` wird die Funktion `init_board` implementiert. In dieser Funktion wird die Funktion `initSetGPIOIPC` aufgerufen. Die Implementierung der Funktion `initSetGPIOIPC` ist in der Datei `MATLAB/R2023a/toolbox/c2b/c2838xBoard_Realtime_Support.c` enthalten.

Hier kann gesehen werden, dass in der Funktion `setGPIOForEthernet` feste Adressen für die Ethernet-GPIO-Konfiguration verwendet werden. Anhand von Abbildung 5.37 und des folgenden Beispiels wird die Konfiguration der GPIOs erklärt.

```
406  Uint32 setGPIOForEthernet(Uint32 data)
407  {
408      // MDIO Signals
409
410      GPIO_SetupPinMux(0x00000E69U & 0x000000FF, GPIO_MUX_CPU1, (0x00000E69U & 0x0000FF00)>>8);
411      GPIO_SetupPinMux(0x00000E6AU & 0x000000FF, GPIO_MUX_CPU1, (0x00000E6AU & 0x0000FF00)>>8);
412
413      //MII Signals
414      GPIO_SetupPinMux(0x00000E6DU & 0x000000FF, GPIO_MUX_CPU1, (0x00000E6DU & 0x0000FF00)>>8);
415      GPIO_SetupPinMux(0x00000E6EU & 0x000000FF, GPIO_MUX_CPU1, (0x00000E6EU & 0x0000FF00)>>8);
416
417      GPIO_SetupPinMux(0x00000B4BU & 0x000000FF, GPIO_MUX_CPU1, (0x00000B4BU & 0x0000FF00)>>8);
418      GPIO_SetupPinMux(0x00000E7AU & 0x000000FF, GPIO_MUX_CPU1, (0x00000E7AU & 0x0000FF00)>>8);
419      GPIO_SetupPinMux(0x00000E7BU & 0x000000FF, GPIO_MUX_CPU1, (0x00000E7BU & 0x0000FF00)>>8);
420      GPIO_SetupPinMux(0x00000E7CU & 0x000000FF, GPIO_MUX_CPU1, (0x00000E7CU & 0x0000FF00)>>8);
```

Abbildung 5.37: Auszug von der Funktion `setGPIOEthernet`

Die Berechnung `0x00000E69U & 0x000000FF` leert alle Bytes bis auf die ersten beiden. Zählrichtung ist hierbei wie folgt: MSB ist ganz links und LSB ist ganz rechts. Die Berechnung ergibt somit den Wert `0x69`. Umgerechnet ins Dezimalsystem ergibt das eine 105. Anschließend wird die MUX Position mit `(0x00000E69U & 0x0000FF00) ≫ 8` berechnet. Als Ergebnis wird 0x0E erhalten, welches die MUX Position angibt.

Die Berechnungen sorgen somit dafür, dass die benötigten GPIOs für die Verwendung der Ethernet-Schnittstelle konfiguriert werden. Die verwendete Konfiguration ist für die controlCARD [25]. Mit der controlCARD ist die Verwendung des External Modes über TCP möglich, da per Default die korrekten GPIOs konfiguriert werden.

Damit der External Mode über TCP auf der Zielhardware [4] verwendet werden kann, müssen die konfigurierten GPIOs und MUX Positionen angepasst werden. Im Anhang unter A.1.12 ist eine Übersicht des Ethernet-Mappings zu sehen. In dieser sind die von der controlCARD verwendeten GPIOs und Hex-Adressen für die Ethernet-Schnittstelle zu sehen. Auf der rechten Hälfte sind die verwendeten GPIOs und Hex-Adressen der Zielhardware [4] zu sehen. Die Werte der Hex-Adressen und die Benamung wurden an dieser Stelle von der Datei `pin_map.h` aus dem Verzeichnis `ti/c2000/C2000Ware_4_00_00_00/driverlib/f2838x/driverlib` übernommen.

Da diese Werte fest im generierten Code eingetragen sind, werden zwei separate Funktionen in der Datei `MW_c28xGPIO.c` hinzugefügt, die für die Steuerung und Konfiguration der GPIOs verantwortlich sind.

Die erstellte Funktion `getCarPediemGPIONumber` korrigiert die GPIOs. Wenn das vorherige Beispiel weitergeführt wird, bekommt die Funktion den Wert 105 rein und gibt eine 42 zurück. Bei der controlCARD ist GPIO 105 für die Clock des MDIOs[13] zuständig. Bei der Zielhardware ist GPIO 42 für die Clock des MDIOs zuständig. Im Anhang E ist die erstellte Funktion zu sehen. Mit der Übersicht aus A.1.12 und der Kommentare im Code sind die Methoden nachvollziehbar.

Die erstellte Funktion `getCarPediemMuxPosition` korrigiert die MUX Position. Diese Funktion verwendet die Ausgabe von der Funktion `getCarPediemGPIONumber` und gibt die korrigierte MUX Position zurück. Im Anhang F ist die erstellte Funktion zu sehen. Die Funktion gibt die korrigierte MUX Position aus, wenn als Eingabe der korrekte GPIO eingegeben wird. Die Funktion `getCarPediemGPIONumber` gewährleistet, dass die korrekten GPIOs verwendet werden.

Die Funktion `GPIO_SetupPinMux` hat die folgenden Eingabeparameter `Uint16 gpioNumber, Uint16 cpu, Uint16 muxPosition`. Die erstellten Funktionen werden am Anfang der Funktion `GPIO_SetupPinMux` aufgerufen. Hierdurch

[13] **M**anagement **D**ata **I**nput / **O**utput

wird gewährleistet, dass die korrigierte `gpioNumber` und `muxPosition` verwendet werden. Weitere Funktionen, wie zum Beispiel `GPIO_ReadPin`, wurden gleichermaßen angepasst.

Damit der Wechsel auf die controlCARD nicht mit Aufwand verbunden ist, wird um die erstellten Funktionen ein Makro hinzugefügt. Durch diese Präprozessoranweisung wird gewährleistet, dass der hinzugefügte Code nur dann mitgebaut wird, wenn das Makro *CARPEDIEM_PIN_CORRECTION* gesetzt ist. Diese Variable wird als *Custom Code* in den *Hardware Setting* wie in Abbildung 5.38 eingefügt. Dies muss sowohl im Implementierungsmodell (siehe Anhang A.1.6) als auch im Server-Modell (siehe Anhang A.1.8) eingefügt werden.

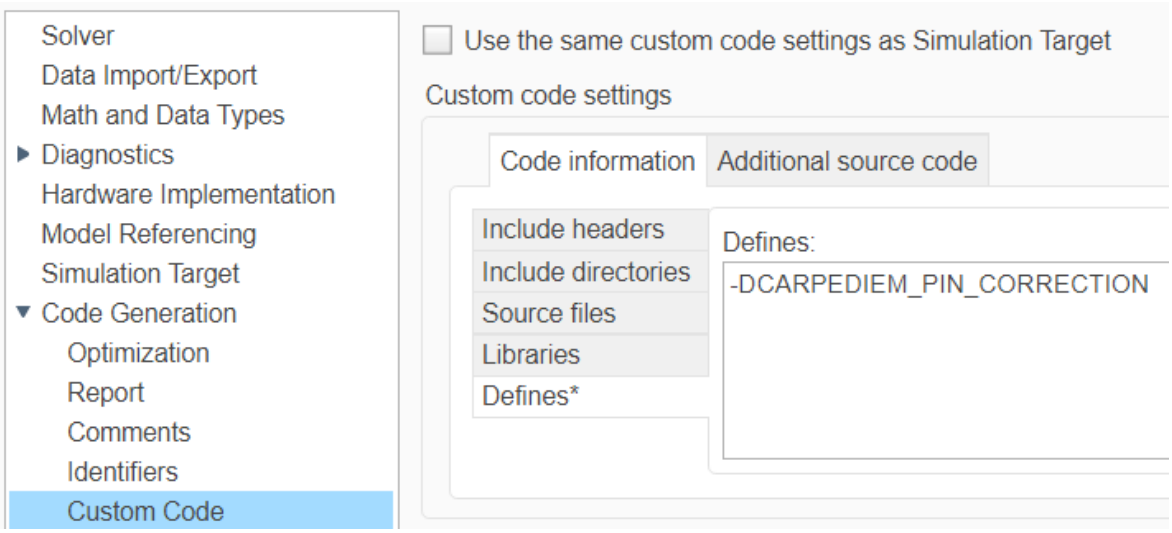


Abbildung 5.38: Anpassung der *Code Generation* in den *Hardware Settings*

Wenn die controlCARD verwendet wird, muss das Feld im *Custom Code* leer bleiben. Hierdurch werden die Default-Werte verwendet und die erstellten Funktionen werden übersprungen.

An dieser Stelle wird angemerkt, dass mit der Bearbeitung der Dateien von Texas Instruments oder The MathWorks die Funktionalität der Programme wie beispielsweise MATLAB hierunter leiden und kaputt gehen können. Da MATLAB bei der Codegenerierung die Dateien in Form einer Basissoftware verwendet und diese nicht mehr

gleich den Herstellerangaben sind, könnte das Programm bei unsachgemäßen Änderungen nicht mehr funktionieren oder Probleme bei der Codegenerierung auftreten. Falls Probleme dieser Art auftreten, muss die Firmware auf die Herstellereinstellungen zurückgesetzt werden.

6 Modellbasiertes Testen

Da alle Test-Cases einen ähnlichen Aufbau haben, wird in diesem Kapitel der vollständige Testablauf für einen *Test-Case* detailliert erklärt.

Das modellbasierte Testen wird in folgende Abschnitte unterteilt:

- Verlinkung der Anforderungen mit Tests
- Testdatenerstellung
- Testrahmenerstellung
- Testdurchführung
- Testauswertung
- Testabdeckung

Die Vorstellung der folgenden Tools erfolgt in [2]:

- *Simulink Test*

  Mit Simulink Test ist es möglich, systematische, simulationsbasierte Tests von Modellen zu erstellen, zu verwalten und auszuführen [2].

- *Simulink Coverage*

  Mit *Simulink Coverage* der *Simulink Verification and Validation*-Toolbox wird die Modellüberdeckung während der Modellsimulation und dem modellbasiertem Test gemessen und analysiert [2].

Des Weiteren werden die Abdeckungskriterien auch in [2] beschrieben. Hierunter fallen die folgenden Kriterien:

- *Decision Coverage*

  Anteil der vom Test berührten Kanten/Transitionen im Graphen [2].

- *Execution Coverage*

  Anteil der vom Test berührten Knoten im Graphen [2].

- *Condition Coverage*

  Anteil der vollständig durchlaufenden Bedingungen als true oder false [2].

- *MCDC*[1]

6.1 Verlinkung der Anforderungen

Als Vorbereitung der Testumgebung werden die abgeleiteten Systemmodellanforderungen (siehe Anhang A.1.4) mit dem Modell verknüpft. Hierfür wird der *Requirements Editor* verwendet. Mit diesem ist es möglich Anforderung aus einer Word-Datei zu importieren.

In Abbildung 6.1 ist die Verlinkung der Anforderungen gegen das Implementierungsmodell (siehe Anhang A.1.6) zu sehen.

[1] **M**odified **C**ondition/ **D**ecision **C**overage

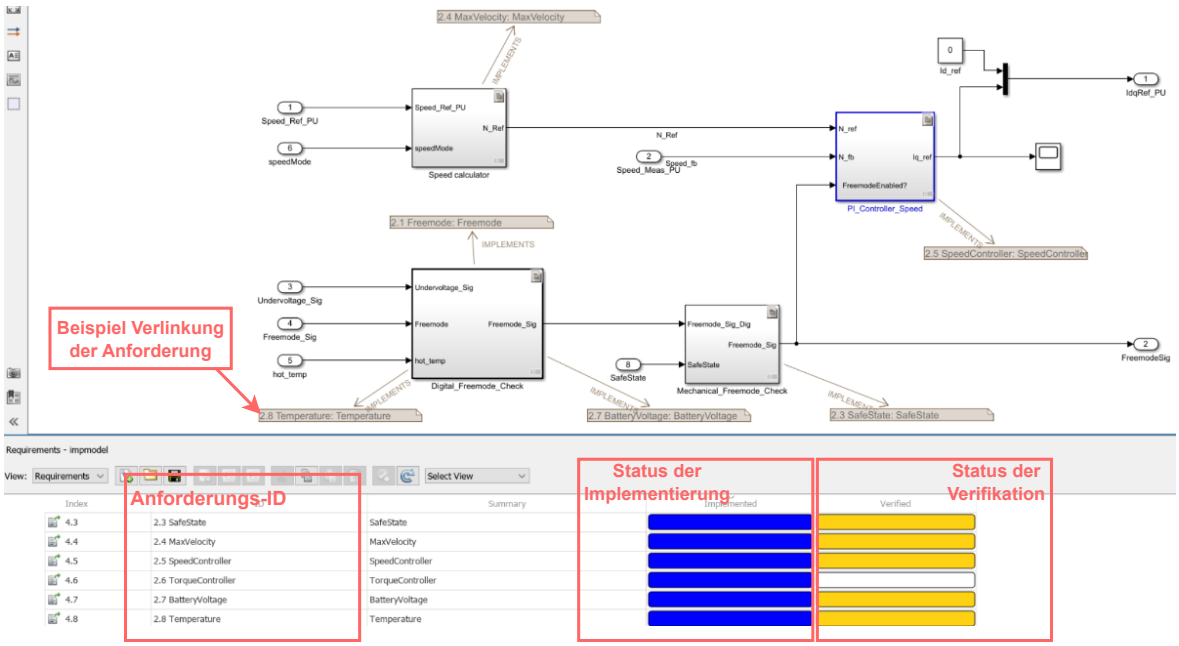


Abbildung 6.1: Verlinkung der Anforderung mit Subsystemen

Anhand der Farben und Spaltenbeschreibung kann in der unteren Ansicht gesehen werden, auf welchem Stand die Implementierung und Verifikation ist. Der blaue Balken stellt den Stand der Implementierung dar. Der gelbe Balken stellt den Stand der Testverlinkung dar.

Die Anforderungen (siehe Anhang A.1.16) werden per *Drag and Drop* auf die Subsysteme geschoben. Hierdurch wird die Implementierung mit der Anforderung verlinkt. Das Verlinken der einzelnen Tests mit den Anforderungen wurde im Test-Manager händisch durchgeführt. Dieses Vorgehen ist in Kapitel 6.4 näher beschrieben.

Der Torque-Controller wurde aufgrund fehlender Zeit nicht mit MIL[2]-Tests verifiziert. Dementsprechend ist der Balken für die Verifikation leer. Stattdessen wird in Kapitel 7 diese Komponente bei der Systemintegration getestet. Die Implementierung der Berechnung der Position über die Hall Sensoren ist in der *Simulation* anders implementiert als in der *Code Generation*. Daher wird diese Komponente ebenfalls in der Systemintegration getestet und verifiziert.

[2]**M**odel-**i**n-the-**l**oop

6.2 Testdatenerstellung

Die Testdatenerstellung beinhaltet die Methode die Test-Inputs bereitzustellen. Für die Erstellung der Daten gibt es verschiedene Ansätze [2]. In dieser Arbeit wurden die Testdaten mithilfe des *Test Sequence*-Blocks erstellt.

Die Erstellung der Testdaten erfolgt für das Subsystem *Speed calculator* (*SW_ANF_04*) wie in Abbildung 6.2 dargestellt.

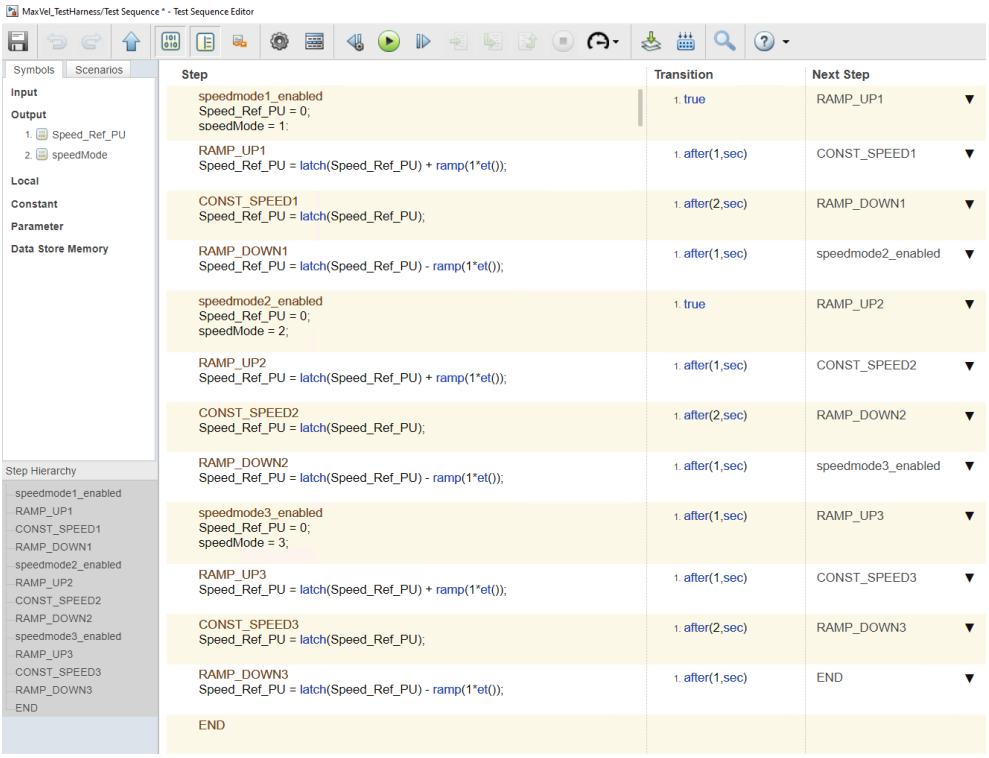


Abbildung 6.2: Testdatenerstellung mithilfe des *Test Sequence*-Blocks im Subsystem *Speed calculator*

Im *Test Sequence*-Block werden die Stufen definiert, wodurch verschiedene Eingangsbedingungen simuliert werden können. Hier werden durch die Verwendung der vorgegeben Spalten die aktuelle Stufe bzw. Unterstufe, Übergänge und die darauffolgende Stufe bzw. Unterstufe definiert. Mehr hierzu kann aus [2] entnommen werden.

6.3 Testrahmenerstellung

Um einen Testrahmen zu erstellen ermöglicht Simulink im Modell die Erzeugung eines *Test Harness*-Modells. Mit der Erstellung eines *Test Harness* wird das zu testende Subsystem vom Gesamtmodell isoliert. Gleichzeitig wirken sich Änderungen im *Harness*-Modell auf die "originale" Komponente im Hauptmodell aus.

In Abbildung 6.3 ist die Erstellung des Testrahmens für das Subsystem *Speed calculator* zu sehen.

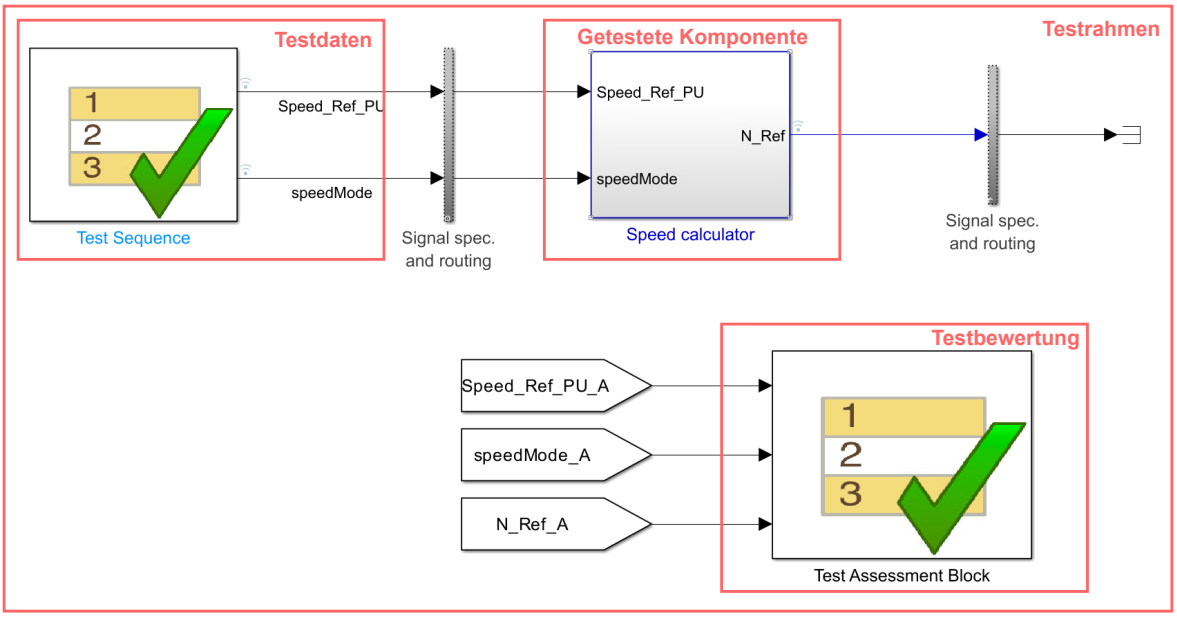


Abbildung 6.3: Erstellung des Testrahmens vom Subsystem *Speed calculator*

In Abbildung 6.4 ist eine Bewertung des Tests dargestellt. Auch hier wird wie beim *Test Sequence*-Block mithilfe von Stufen und Unterstufen einzelne Verifikation-Statements definiert.

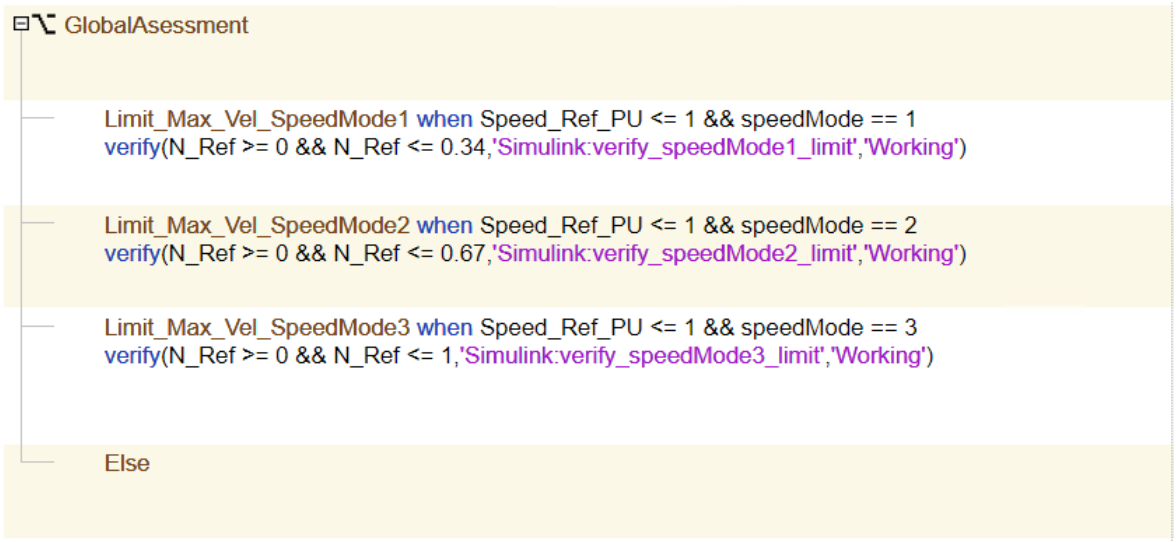


Abbildung 6.4: Erstellung der Testbewertung mit dem *Test Assessment*-Block vom Subsystem *Speed calculator*

Die Verwendung von sog. *Verify Statements* ermöglicht automatisierte Testauswertungen im Zuge der späteren Testdurchführung. Ein *Verify Statement* besteht aus einem zu überprüfenden logischen Ausdruck, einer Bezeichnung zur Nachverfolgung des Bewertungskriteriums im Zuge der Testauswertung, sowie einer Fehlermeldung [2].

6.4 Testdurchführung

Durch das Tool *Test Manager* wird die Testdurchführung realisiert. In Abbildung 6.5 ist die Übersicht des *Test Managers* zu sehen. Hier ist es möglich die durchgeführten Tests und dazugehörigen Testkonfigurationen zu bearbeiten.

Die Tests werden alle in einer vom Modell unabhängigen *Testfile* (siehe Anhang A.1.14) konfiguriert.

In den Testkonfigurationen kann beispielsweise zu einem *Test-Case* der jeweilige Testrahmen ausgewählt werden.

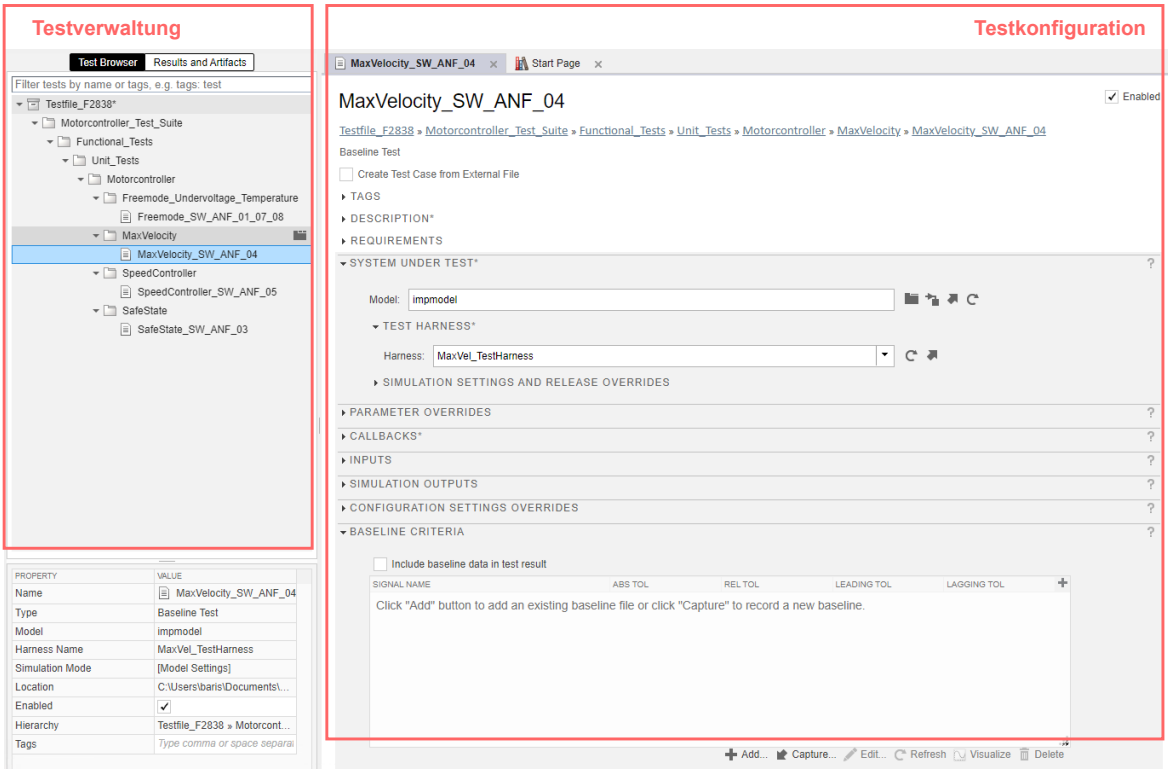


Abbildung 6.5: Übersicht vom *Test Manager*

Im *Test Manager* können sog. *Test Suites* erstellt werden. Mithilfe der *Test Suites* kann die Gruppierung mehrerer Tests realisiert werden.

Die Verlinkung der Verifikation mit den Systemmodellanforderungen (siehe Anhang A.1.4) wird in dieser Arbeit über die *Test Suites* realisiert. Dieses Vorgehen ist am Beispiel des *Speed calculators* in Abbildung 6.6 zu sehen.

Im Untermenü unter dem Punkt *Requirements* kann die Verlinkung durchgeführt werden. Wenn die Testergebnisse vorliegen, werden diese in der *Requirements perspective* zu sehen sein.

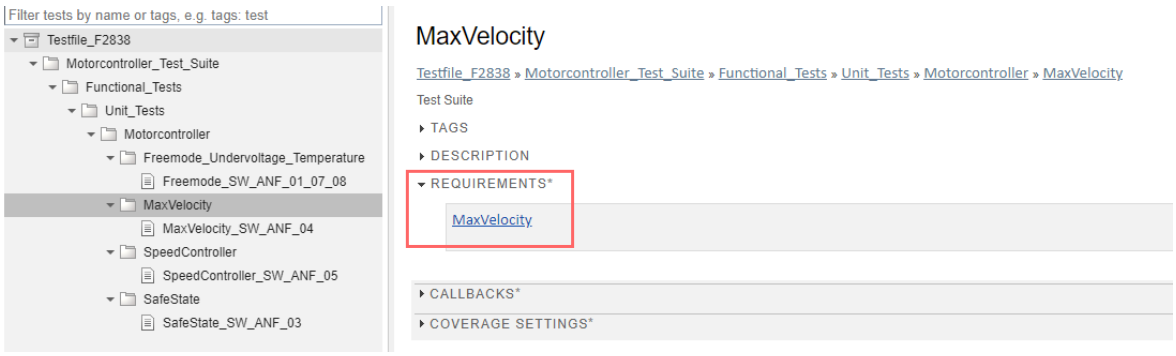


Abbildung 6.6: Verlinkung der Anforderung im Test Manager

Es wird unter den folgenden *Test-Case*-Varianten unterschieden:

- *Baseline Test*
- *Equivalence Test*
- *Simulation Test*
- *Real-Time Test*

Näheres zu den verschiedenen *Test-Case*-Varianten kann aus [2] entnommen werden.

In dieser Arbeit wurden lediglich *Baseline Tests* erstellt, da es sich um die Erstversion handelt und nicht verschiedene Versionen bzw. Implementierungen miteinander verglichen werden.

6.5 Testauswertung

Die Auswertung der Tests erfolgt ebenso im *Test Manager*. Nach Ausführung der Tests (über den *Run*-button) öffnet sich ein neuer Tab. Aus diesem können die Testergebnisse entnommen werden. Auf Wunsch kann ein Report über die Testergebnisse generiert werden.

In Abbildung 6.7 ist die Testauswertung des Subsystems *Speed calculator* dargestellt.

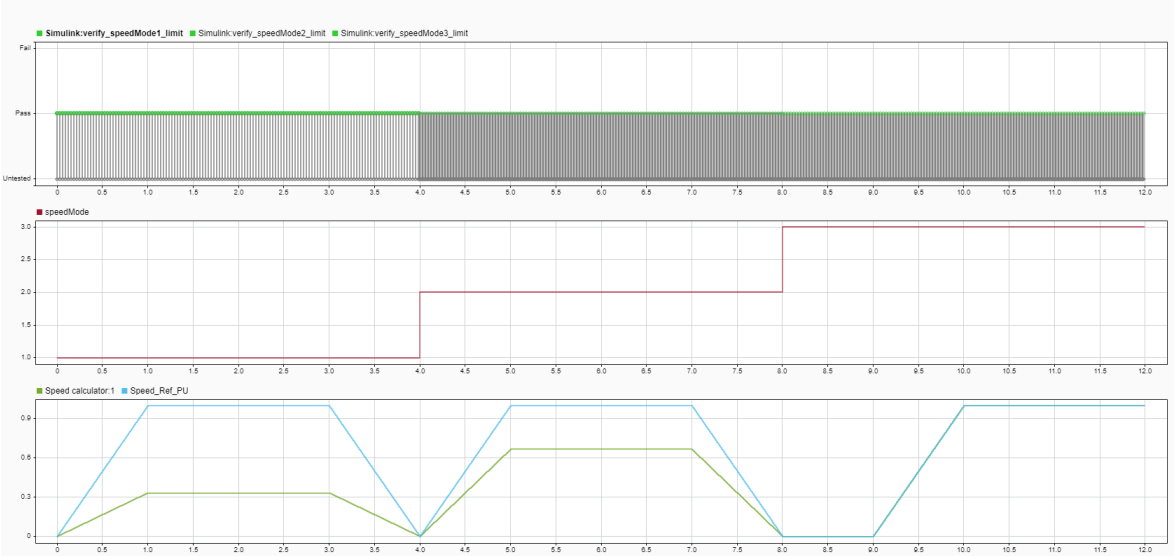


Abbildung 6.7: Testauswertung vom *Speed calculator*

Im ersten Subplot ist das Ergebnis des *Verify Statements* zu sehen. Ein Test ist erfolgreich durchgelaufen, wenn die Kriterien im *Test Assessment*-Block bestanden sind. Wie zu sehen ist, existieren Zeitpunkte wo das Resultat *Untested* bleibt. Grund hierfür ist, dass zum Beispiel beim Testablauf für *speedMode 1* nicht der Testbereich für *speedMode 2* oder *3* gedeckt wird und umgekehrt. Aus diesem Grund wurde im *Test Assessment*-Block für jeden *speedMode* das dazugehörige *Verify Statement* erstellt.

Im zweiten Subplot sind die drei *speedModes* zu sehen. Abhängig vom *speedMode* wird die Geschwindigkeitsanforderung am Ausgang berechnet.

Im dritten Subplot ist in blau die Geschwindigkeitsanforderung zu sehen. Da die Geschwindigkeit auch im PU-System integriert wurde, ist ein Wertebereich von 0 bis 1 möglich. Das in grün dargestellte Signal ist die berechnete Geschwindigkeitsanforderung, welche am Ausgang des *Speed calculators* geführt wird.

Die Testauswertungen der restlichen modellbasierten Tests können aus Anhang B entnommen werden.

- SW_ANF_01 in Anhang B.1
- SW_ANF_03 in Anhang B.2

- SW_ANF_05 in Anhang B.3
- SW_ANF_07 in Anhang B.1
- SW_ANF_08 in Anhang B.1

Der Report der Testergebnisse aller Komponenten befindet sich im Anhang A.1.15.

Nach Ablauf der Testreihe kann, wie in Abbildung 6.8 dargestellt, in der *Requirements perspective* der Stand der Verifikation eingesehen werden.

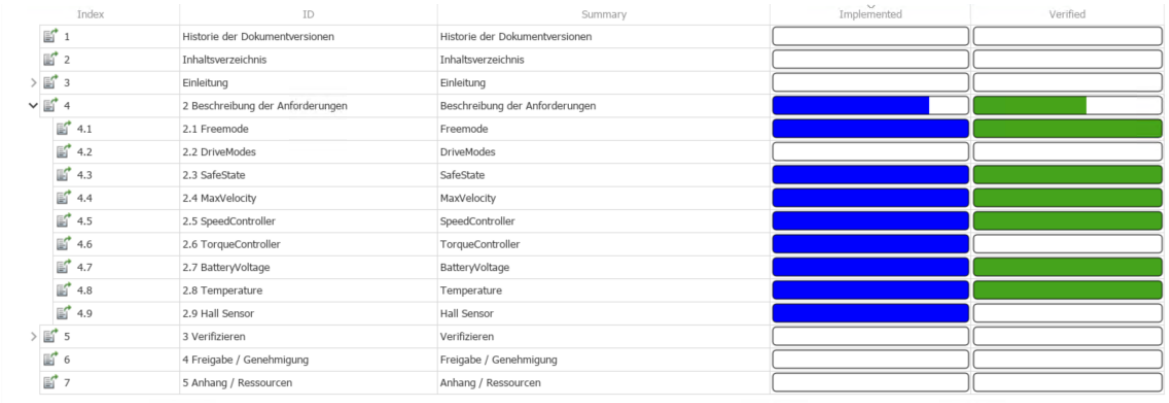


Abbildung 6.8: Stand der Implementierung und Verifikation

In Kapitel 5.2 wurde darauf hingewiesen, dass aufgrund nicht vorhandener Zeit die Implementierung verschiedener *DriveModes* nicht erreicht wurde. Dementsprechend konnten diese auch nicht modellbasiert getestet werden.

6.6 Testabdeckung

Die Abdeckung der Tests erfolgt durch die Toolbox *Simulink Coverage*. Die am Anfang des Kapitels erwähnten Abdeckungskriterien werden bei der Testdurchführung geprüft.

Im Anhang A.1.15 ist neben den Testergebnissen auch die Testabdeckung zu sehen. Alle Tests haben mit den Abdeckungskriterien (siehe Anfang Kapitel 6) die vollste Abdeckung erreicht. Es fällt in den Ergebnissen auf, dass die Komponente *PI_Controller_Speed* "nur" eine Abdeckung von 92 % hat. Dies kann jedoch ignoriert

werden, da die Bedingung, welche in diesem Abdeckungskriterium überprüft wurde, nie eintreten kann. Die Geschwindigkeitsanforderung wird aufgrund des ADC-Blocks immer ≥ 0 sein. Aus diesem Grund kann bei dieser Komponente eine Abdeckung von 100 % angenommen werden.

7 Verifikation

In diesem Kapitel werden folgende Themen verifiziert:

- Grundfunktionalität des External Modes über TCP
- Systemmodellanforderungen mithilfe des External Modes über TCP

Die Verifikation wurde mit der Zielhardware aus [4] durchgeführt.

7.1 Verifikation des External Modes über TCP

In dieser Arbeit wurde der Server (siehe Anhang A.1.8) auf dem CM erstellt. Die LAN-Verbindung und Konfiguration muss gemäß Kapitel 5.7 erfolgen. Nach der Erstellung kann die Grundfunktionalität der Netzwerkkonfiguration im Powershell Fenster mit `ping 192.168.1.8` verifiziert werden. Bei einem erfolgreichem Ping kann davon ausgegangen werden, dass die Netzwerkkarte auf dem Target korrekt konfiguriert wurde und die IP-Adresse vom Client aus erreicht werden kann. In Abbildung 7.1 ist ein Auszug des `ping`-Befehls im Powershell Fenster zu sehen.

```
C:\Users\baris> ping 192.168.1.8

Ping wird ausgeführt für 192.168.1.8 mit 32 Bytes Daten:
Antwort von 192.168.1.8: Bytes=32 Zeit<1ms TTL=255
Antwort von 192.168.1.8: Bytes=32 Zeit<1ms TTL=255
Antwort von 192.168.1.8: Bytes=32 Zeit<1ms TTL=255
Antwort von 192.168.1.8: Bytes=32 Zeit<1ms TTL=255

Ping-Statistik für 192.168.1.8:
    Pakete: Gesendet = 4, Empfangen = 4, Verloren = 0
    (0% Verlust),
Ca. Zeitangaben in Millisek.:
    Minimum = 0ms, Maximum = 0ms, Mittelwert = 0ms
```

Abbildung 7.1: Auszug vom `ping`-Befehl auf der Zielhardware

Wie in Abbildung 7.1 dargestellt ist, ist die Erreichbarkeit der IP-Adresse des Targets verifiziert.

Als weitere Verifikation wird im Server durch den *TCP Send*-Block ein 8-bit Zählsignal veröffentlicht. Mit den Server-/ und Client-Konfigurationen gemäß Kapitel 5.7 wird der Client (siehe Anhang A.1.9) mit dem Server verbunden. In Abbildung 7.2 ist das empfangene Zählsignal zu sehen. Hierdurch erfolgt die Verifikation des External Modes über TCP.

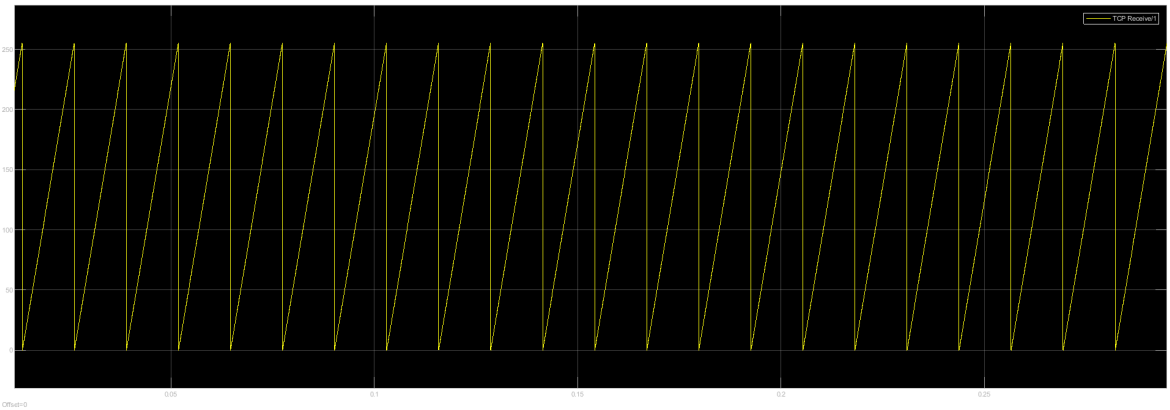


Abbildung 7.2: Empfangenes Signal vom Client

Im nächsten Kapitel werden die Systemmodellanforderungen mit dem entwickelten External Mode verifiziert.

7.2 Verifikation der Systemmodellanforderungen

Um die Verifikation der Systemmodellanforderungen (siehe Anhang A.1.4) durchzuführen, wird der generierte Code auf das Target hochgeladen. Diese Anforderungen werden mit dem entwickelten External Mode verifiziert.

Die Anforderungen mit der ID *SW_ANF_06* und *SW_ANF_09* wurden nicht modellbasiert getestet, daher kann kein Vergleich gezogen werden. Stattdessen wird die Funktionalität auf dem Target [4] unter Beweis gestellt.

In Abbildung 7.3 ist die Verifikation von verschiedenen *speedModes* (*SW_ANF_04*) zu sehen.

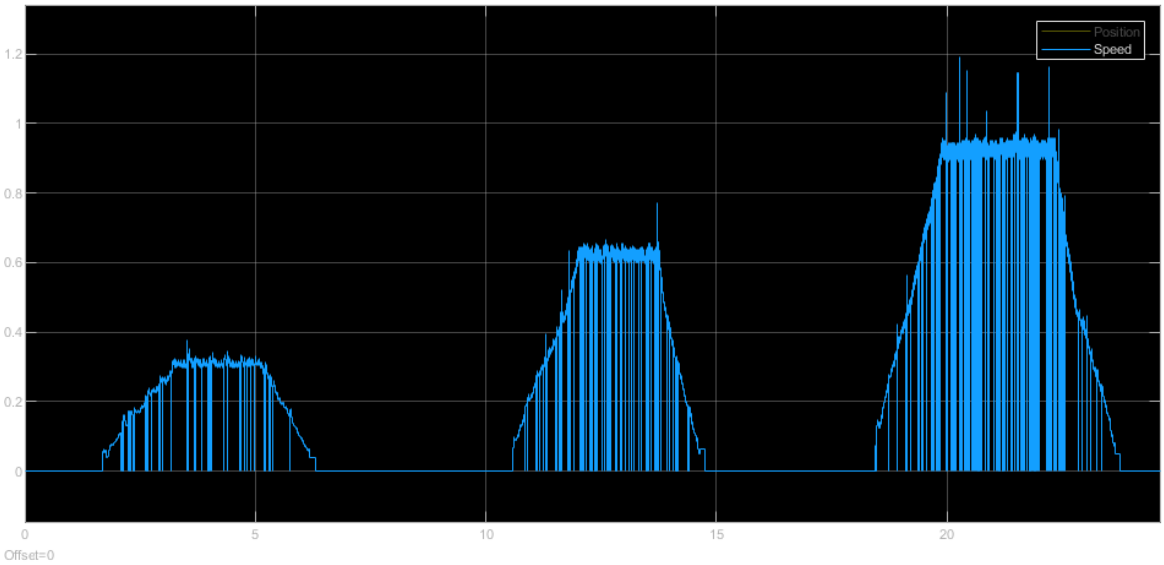


Abbildung 7.3: Verifikation der Komponente *speedMode*

Die aktuelle Geschwindigkeit ist abhängig vom *speedMode*. Im Verlauf der Abbildung 7.3 sind viele Fehler vorhanden. Die Messung der aktuellen Geschwindigkeit wird verfälscht, da an vielen Stellen die Geschwindigkeit auf Null fällt oder ungültige Geschwindigkeiten entstehen.

Zu den Zeitpunkten, wo die Geschwindigkeit auf Null fällt, ist die Position konstant. Statt des extrapolierten Signals, welches eine Sägezahnsignal darstellt, sind Zeitpunkte vorhanden wo die Position konstant bleibt. An diesen Zeitpunkten ist die Geschwindigkeit logischerweise Null.

Mit einem Angestellten von MathWorks [48] wurde mit verschiedenen Ansätzen versucht, den Fehler zu beheben. Hierunter fallen besonders die Eingangssignale des *Hall Speed and Position*-Blocks. Die Signale *Hall State, Speed Validity, Hall State Change Flag* wurden hierbei genauer untersucht. Nachfolgend sind die erwähnten Signale mithilfe des seriellen External Modes (siehe Anhang A.1.10) festgehalten. In gelb sind auf beiden Bildern die *Hall States* zu sehen.

In Abbildung 7.4 ist in blau das *Hall State Change Flag*-Signal zu sehen. Der Wert zeigt die Änderung des Hall-Zustands und den Status der Blockausführung an. Der Wert Eins gibt an, dass sich der Hall-Zustand geändert hat, aber die Ausführung des

Blocks noch aussteht. Der Wert Null zeigt an, dass der Block die Ausführung der letzten Hall-Zustandsänderung abgeschlossen hat [49].

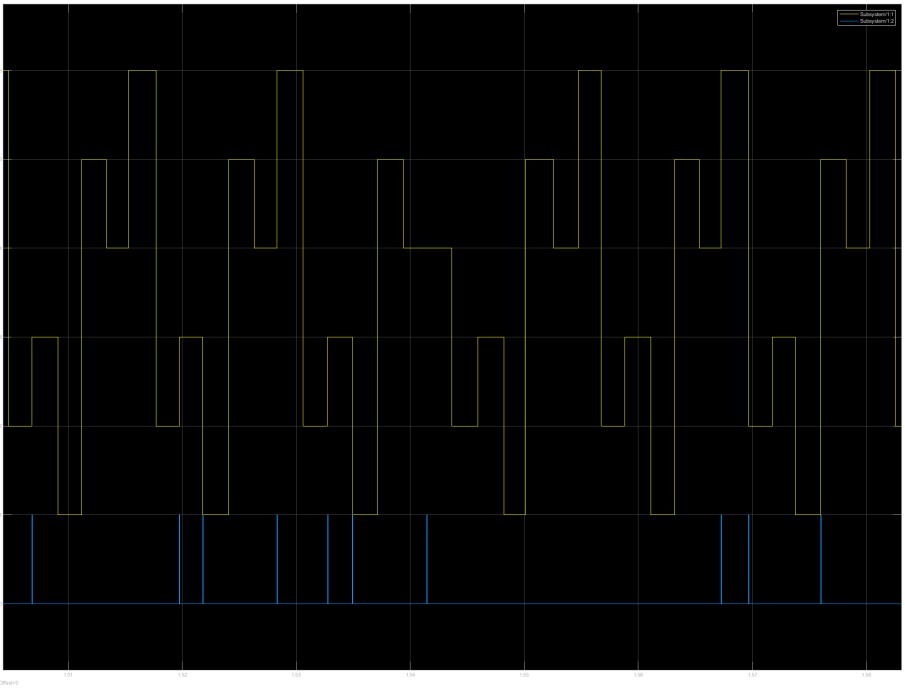


Abbildung 7.4: *Hall State* und *Hall State Change Flag*

In Abbildung 7.5 ist in blau das *Speed Validity*-Signal zu sehen. Der Wert gibt die Gültigkeit des Hall-Zustands an. Der Wert Null zeigt, dass der aktuelle oder der vorherige Hall-Zustand ungültig ist und dass der Block die Geschwindigkeit und die Position nicht berechnen kann. Der Wert Eins gibt an, dass sowohl der aktuelle als auch der vorherige Hall-Zustand gültig sind. Dies ermöglicht es dem Block, die Geschwindigkeit und Position zu berechnen. [49].

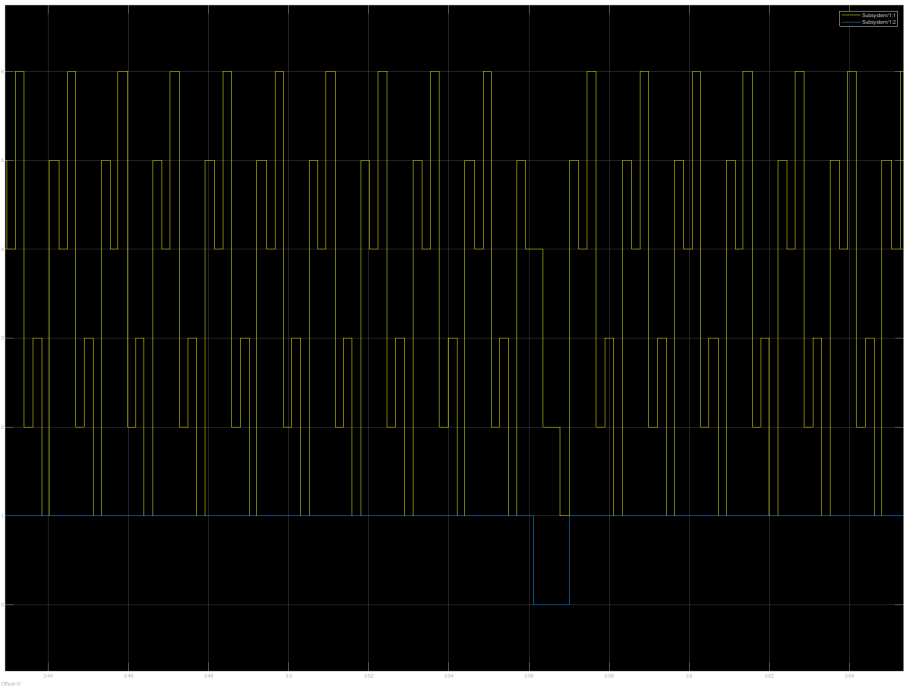


Abbildung 7.5: *Hall State* und *Speed Validity*

Aus den genannten beiden Untersuchungen kann abgeleitet werden, dass die Berechnung der Position und der Geschwindigkeit nicht einwandfrei funktioniert und es des Öfteren zu Fehlern kommt. Nach weiteren Debug-Sitzungen konnte der Fehler nicht in der Software identifiziert werden und es stellte sich die Vermutung auf, dass hardwareseitig etwas nicht ordnungsgemäß funktioniert [48]. Eine Ursache kann sein, dass aufgrund der nicht vorhandenen *Level Shifter* sehr viel Rauschen vorhanden ist, wie in der Abbildung 7.6 zu sehen ist.

Um die Vermutung bestätigen zu können, wurde eine andere Hardware in Betrachtung gezogen. Das verwendete LaunchPad F280049C [50] besitzt einen integrierten *Level Shifter*. Nach Anpassungen sämtlicher GPIOs, ePWM, ADC-Blöcke und logischem Zusammenhang konnte die Software auf das LaunchPad hochgeladen werden. Hierbei wurden Algorithmen und Berechnungen nicht verändert, sondern nur auf die neue Peripherie angepasst. Dadurch kann die Funktionalität der Software ebenfalls verifiziert werden.
Die Messung hat gezeigt, dass mit Hinzunahme des *Level Shifters* die *Hall States*, der *Speed Validity* und der *Hall State Change Flag* und somit die Positionsbestimmung zuverlässig abläuft. Jedoch sind weitere Unterschiede in der Hardware vorhanden, sodass

die *Level Shifter* nicht ausschlaggebend für die Problemursache sein müssen. Das erstellte Modell befindet sich im Anhang unter A.1.11. In Anhang C.1 ist die resultierende Geschwindigkeit zu sehen.

Um weitere Softwareprobleme auszuschließen, werden neben den Hall-Signale die dazugehörigen eCAP-Interrupts auf der Zielhardware [4] auf einem Oszilloskop in Abbildung 7.6 dargestellt.

Abbildung 7.6: Übersicht der Hall-Signale mit den dazugehörigen eCAP-Interrupts

Im beobachteten Zeitraum ist zu sehen, dass die Hall-Signale nicht sauber sind. Aufgrund von Rauschen werden falsche Pegel interpretiert. Das Rauschen erzeugt falsche eCAP-Interrupts, welche in rot markiert sind. Zusätzlich muss jedoch verdeutlicht werden, dass es sich hier nicht um analoge Tastköpfe handelt, sondern um digitale Tastköpfe. Die Konsequenz hieraus ist, dass das Rauschen der Signale nicht gesehen wird.

Stattdessen wird der vom Mikrocontroller interpretierte Pegel dargestellt. In diesen Zeitpunkten werden neue Geschwindigkeiten berechnet, obwohl die Drehzahl gleich bleibt. Die Bewertung der Hardwareanpassungen erfolgt in Kapitel 7.3.

Zusätzlich ist in Abbildung 7.6 die Ausführungszeit der eCAP-Interrupts zu sehen. Die Ausführung der Interrupts muss nach Kapitel 2.6 schnell erfolgen, um die Latenz des Systems nicht zu erhöhen. Nachdem der jeweilige eCAP-Interrupt ausgelöst wird, ist dieser nach ≤ 3 µs wieder auf GND. Durch das Verkleinern der Zeitachse auf dem Oszilloskop kann dies gesehen werden.

Um die Interrupts auf dem Oszilloskop zu visualisieren, werden *Digital-Output*-Blöcke verwendet. Ein GPIO wird mithilfe der Priorität in den Block-*Properties* als erstes aufgerufen. Somit befindet sich bei Aufruf des Interrupts der Pegel des betrachteten GPIOs auf High. Um den Pegel wieder herunterzusetzen, wird der gleiche GPIO am Ende der Berechnung platziert und mit Null multipliziert, um den Pegel mit Null (GND) zu versehen.

Im Anhang ist die Realisierung am Beispiel des eCAP1-Interrupts (siehe Anhang C.2) zu sehen. Über die *Information Overlays* im Pfad *Debug* kann die Ausführungsreihenfolge *(engl.: Execution Order)* (siehe Anhang C.3) eingesehen werden. Wenn bei den Block-*Properties* die Priorität mit 0 deklariert wird, wird dieser Block als erstes (im Subsystem) aufgerufen.

Als nächstes erfolgt die Verifikation der Komponente *Freemode*. Um eine bessere Übersicht zu gewährleisten, erfolgt die gemeinsame Verifizierung sowohl des digitalen als auch des mechanischen *Freemode-Checks*. An diese Komponente sind die folgenden Systemmodellanforderungen verlinkt:

- *SW_ANF_01* im digitalen *Freemode-Check*
- *SW_ANF_03* im mechanischen *Freemode-Check*
- *SW_ANF_07* im digitalen *Freemode-Check*
- *SW_ANF_08* im digitalen *Freemode-Check*

Durchgeführt wird die Verifikation in zwei Schritten.

- Zu Beginn wird eine feste Geschwindigkeit vorgegeben.

- Mit dem External Mode wird die aktuelle Geschwindigkeit beobachtet. Über den jeweiligen Auslöser muss die Geschwindigkeit auf Null fallen.

In Abbildung 7.7 erfolgt die Aktivierung des Fahrens ohne elektrischen Antrieb über die Unterspannungsabschaltung.

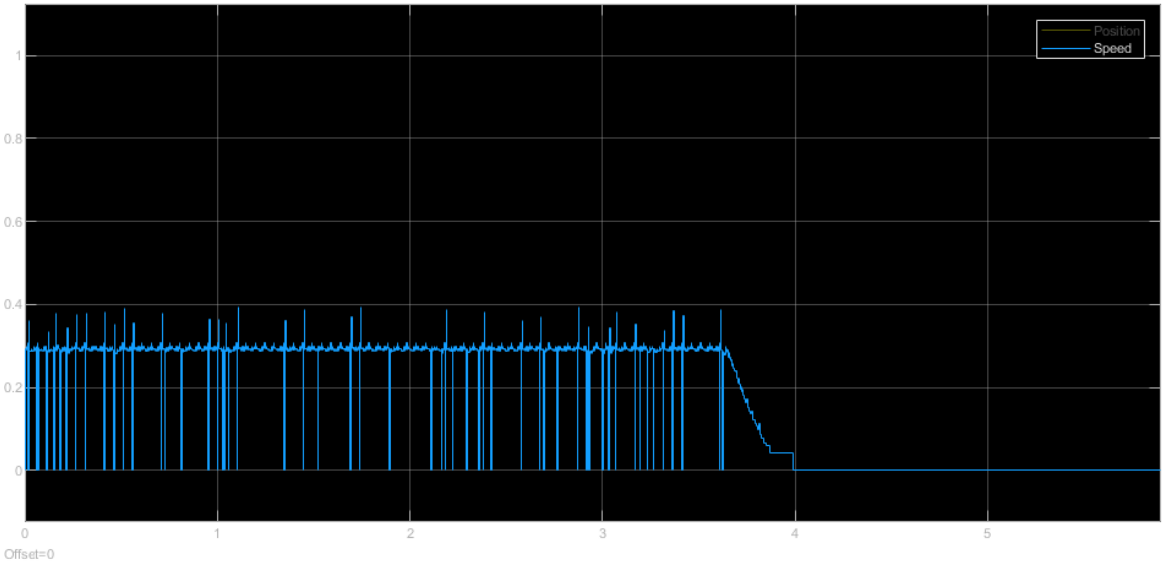


Abbildung 7.7: Verifikation der Unterspannungsabschaltung

Außerdem werden mit einem Oszilloskop die Verläufe der Gate-Signale von einer Halbbrücke bei Erreichen der Unterspannungsabschaltung festgehalten.

In Abbildung 7.8 ist auf einem Oszilloskop die Unterspannungsabschaltung festgehalten. In gelb ist die High-Side und in grün die Low-Side zu sehen. Die Versorgungsspannung ist in blau visualisiert. Wenn die Versorgungsspannung unter 17 V liegt, wird das Fahren ohne elektrischen Antrieb aktiviert. Dies ist durch die geöffneten MOSFETs zu sehen. Während der Verifikation wurde bemerkt, dass der tatsächliche Schwellwert bei ca. 16.5 V liegt. Hierdurch ist der Motor frei drehbar.

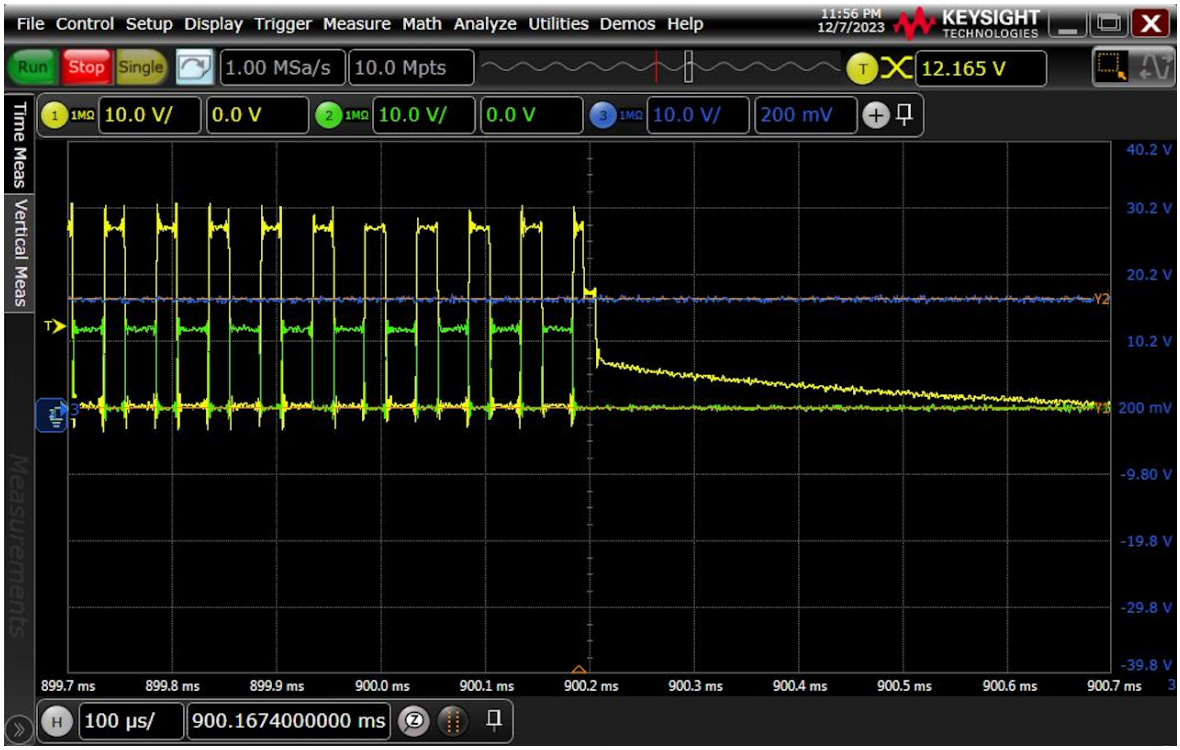


Abbildung 7.8: Visualisierung der Unterspannungsabschaltung

Aufgrund des korrekten Verhaltens mit leichten Abweichungen ist die Funktionalität dieser Komponente verifiziert.

In Abbildung 7.9 ist die Verifikation der Aktivierung des Fahrens ohne elektrischen Antrieb über die empfangene CAN-Nachricht zu sehen. Auch hier wird eine konstante Geschwindigkeit vorgegeben. Zu gegebenem Zeitpunkt wird über den PCAN-Adapter das *Freemode*-Signal versendet.

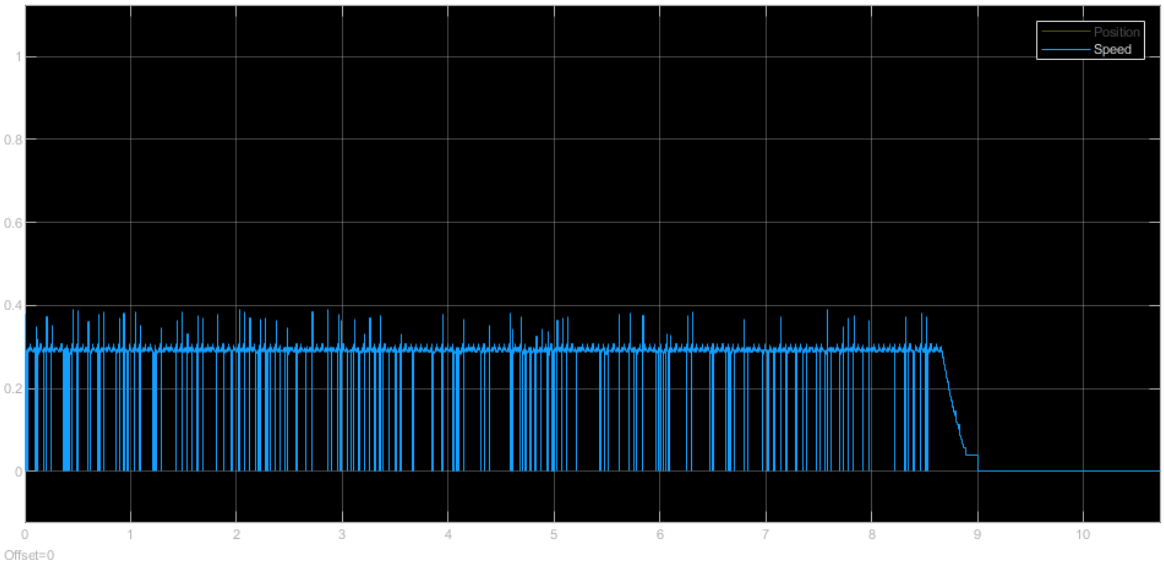


Abbildung 7.9: Verifikation des *Freemode*-Modus über die CAN-Nachricht

Wie zu sehen ist, erfolgt die Aktivierung des Fahrens ohne elektrischen Antrieb hier beim Empfangen der CAN-Nachricht. Dieses Verhalten ist korrekt, somit ist auch diese Komponente verifiziert.

Die Aktivierung des Fahrens ohne elektrischen Antrieb über dem *safeState-button* weist das gleiche Verhalten auf. Bei Knopfdruck fällt die Geschwindigkeit auf Null und die Gates der MOSFETs bleiben auch hier geöffnet.

Die Komponenten für das Fahren ohne elektrischen Antrieb haben alle einen anderen Auslöser, jedoch ist das Ausgangsverhalten bei allen gleich.

Das Fahren ohne elektrischen Antrieb über die MOSFET-Temperatur (SW_ANF_08) konnte nicht geprüft werden, da die Einrichtung hierfür fehlt und es sich um eine Testeinrichtung handelt. Diese darf nicht durch die hohen Temperaturen beschädigt werden. Stattdessen wird die gemessene Temperatur verifiziert.

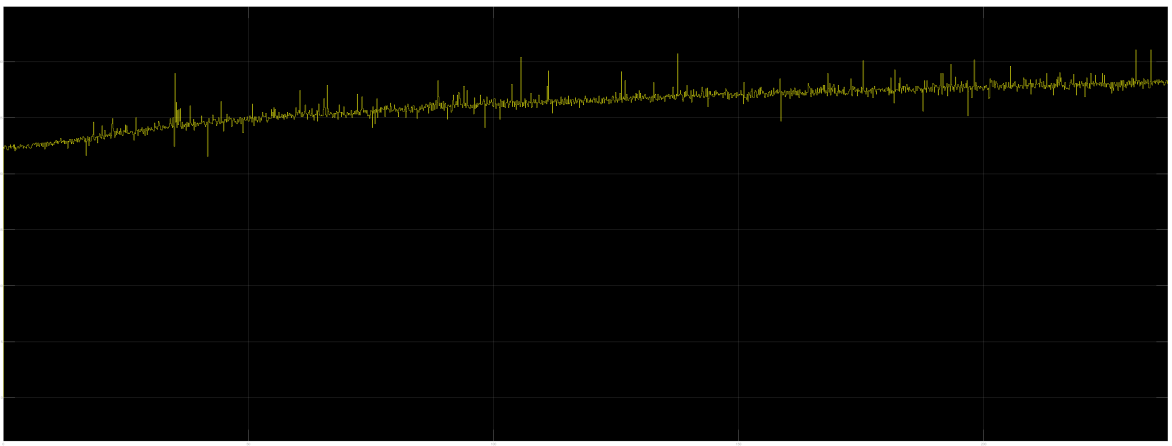


Abbildung 7.10: Verifikation der Temperaturmessung

Bei der Verifikation wird die maximale Geschwindigkeit angefordert und dabei die Temperatur beobachtet. Da an dem Motor keine Last angehängt wird, ist der Stromverbrauch gering und es wird nicht viel Hitze erzeugt. In Abbildung 7.10 ist der beobachtete Zeitraum zu sehen. Die Temperatur steigt hierbei leicht. Die Umgebungstemperatur betrug zum Messzeitpunkt ca. 23 °C. Anfänglich wird auch eine Temperatur von 23 °C gemessen. Diese steigt bis ca. 28 °C hoch. Es wurde begleitend die Temperatur auf dem Temperatursensor mit einem Thermometer verfolgt. Aufgrund sehr geringer Toleranzen mit einer Abweichung von ca. +/- 2 °C gilt diese Komponente als verifiziert.

Der Torque Controller wird wie in Kapitel 5.4 bereits beschrieben vom ADC-Interrupt aufgerufen. In Abbildung 7.11 ist der Aufruf und die Ausführungsdauer des ADC-Interrupts (Torque Controllers) zu sehen. Auch hier muss der Interrupt gemäß Kapitel 2.6 schnell ausgeführt werden.

Abbildung 7.11: Aufruf des ADC-Interrupts in Abhängigkeit vom Low-Side Gate-Signal

Der ADC-Interrupt erfolgt in der Mitte des Gate-Signals für die Low-Side. Die Ausführungsdauer des ADC-Interrupts liegt mit ca. 13.24 µs unter dem geforderten Zeitintervall. Der ADC-Interrupt muss spätestens bis zum nächsten ADC-Interrupt abgeschlossen werden, da sonst die Berechnung fehlschlägt oder ein Interrupt übersprungen wird. Da wie in Kapitel 5.4 beschrieben, der ADC-Interrupt ein *preemtable* Interrupt ist, müssen die Ausführungszeiten der einzelnen eCAP-Interrupts berücksichtigt werden. Unabhängig davon muss die Dauerschleife für den Input-Controller, Speed-Controller und dem Versenden der Nachrichten über den IPC-Channel ebenso ausgeführt werden.

Somit muss der ADC-Interrupt schnell ausgeführt werden, um die Latenz nicht signifikant zu erhöhen. Aufgrund dieser Resultate ist der Torque Controller hiermit erfolgreich verifiziert. Hierbei ist zu berücksichtigen, dass die berechneten PI-Regler-Koeffizienten

(siehe Kapitel 5.1.2) verwendet worden sind. Die Funktionalität dieser Koeffizienten ist ebenso verifiziert.

Die Messung des ADC-Interrupts erfolgt auf die gleiche Weise, wie die eCAP-Interrupts gemessen wurden.

Die Spannungsspitzen sind auf die langen Messleitungen zurückzuführen.

In Abbildung 7.12 ist die Positionsbestimmung zu sehen. Hierbei handelt es sich um das extrapolierte Positionssignal. Um die Positionen zwischen zwei bekannten Punkten zu berechnen, erfolgt eine Extrapolation des Signals im *Hall Speed and Position*-Block.

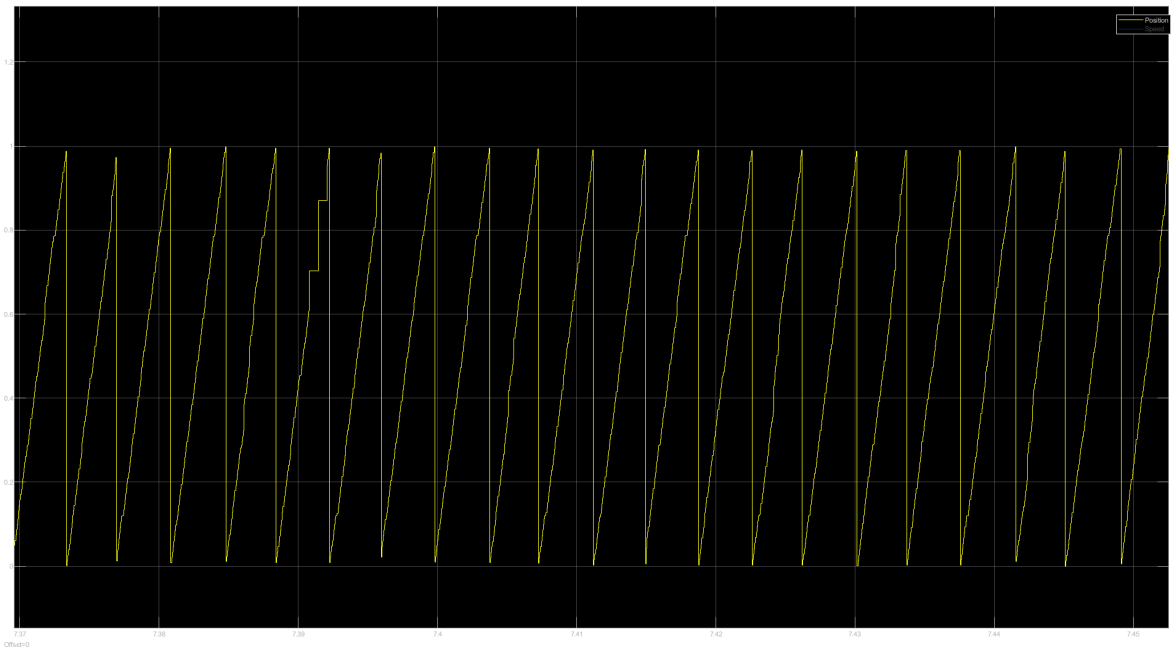


Abbildung 7.12: Verifikation der Positionsbestimmung

Hier sind erneut die Problematiken mit dem verrauschten Signal vorhanden, sodass die Extrapolation fehlschlägt. Diese werden in Kapitel 7.3 behoben.

7.3 Optimierungen

Wie in Kapitel 7.2 zu sehen ist, sind in den Hall-Signalen Rauschen vorhanden, wodurch falsche eCAP-Interrupts ausgelöst werden. Um das Rauschen in den Hall-Signalen zu

minimieren, wurden Anpassungen in der Hardware unternommen, welche in [4] diskutiert werden.

In Abbildung 7.13 ist die Verbesserung der Hardwareanpassungen zu sehen. Bei gleicher Geschwindigkeit wie in Abbildung 7.6 ist hier zu sehen, dass das Rauschen der Hall-Signale verringert wurde. Die eCAP-Interrupts werden somit nur bei den Flankenwechsel ausgelöst.

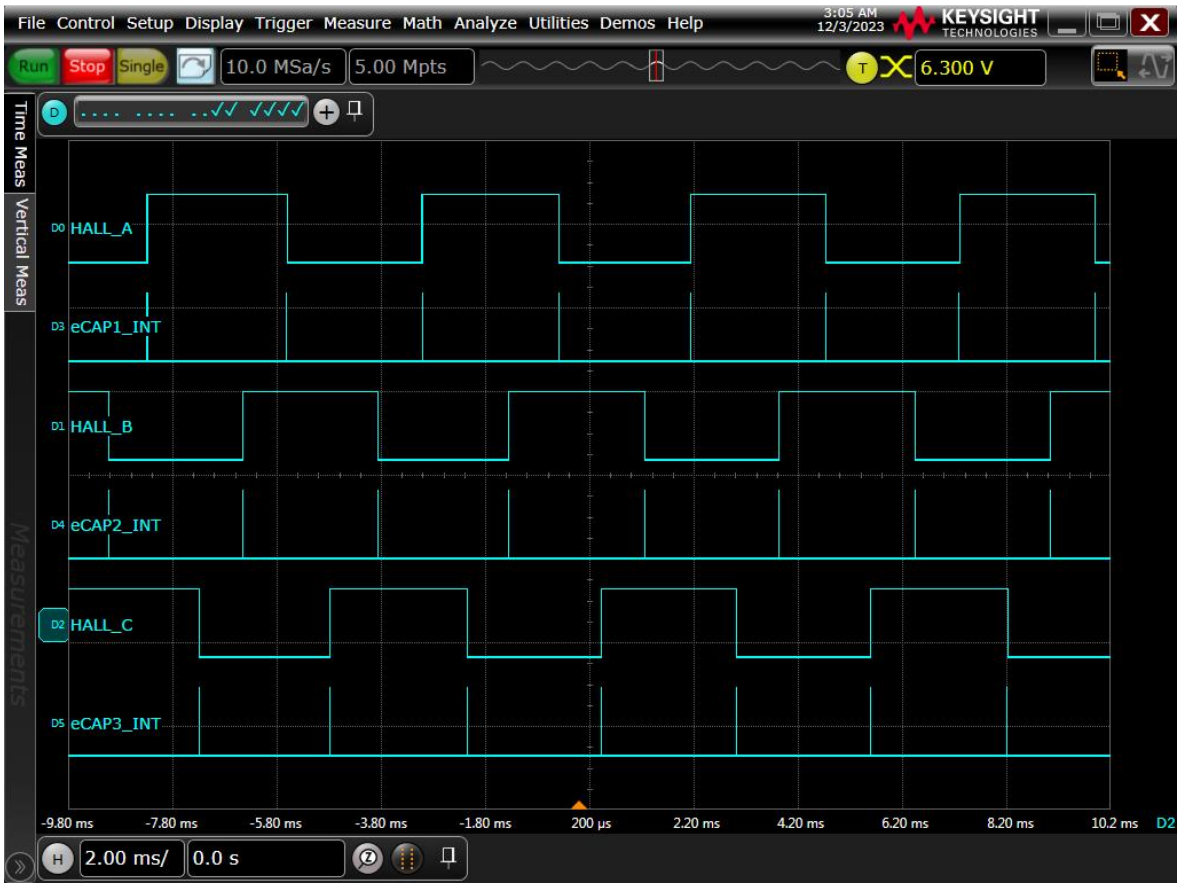


Abbildung 7.13: Übersicht der Hall-Signale mit den dazugehörigen eCAP-Interrupts nach den Hardwareanpassungen

Da nun die eCAP-Interrupts nicht mehr zu den falschen Zeitpunkten ausgelöst werden, sind die Geschwindigkeitsmessungen bei den verschiedenen *speedModes* verbessert. In Abbildung 7.14 ist der Verlauf der gemessenen Geschwindigkeit bei verschiedenen *speedModes* zu sehen.

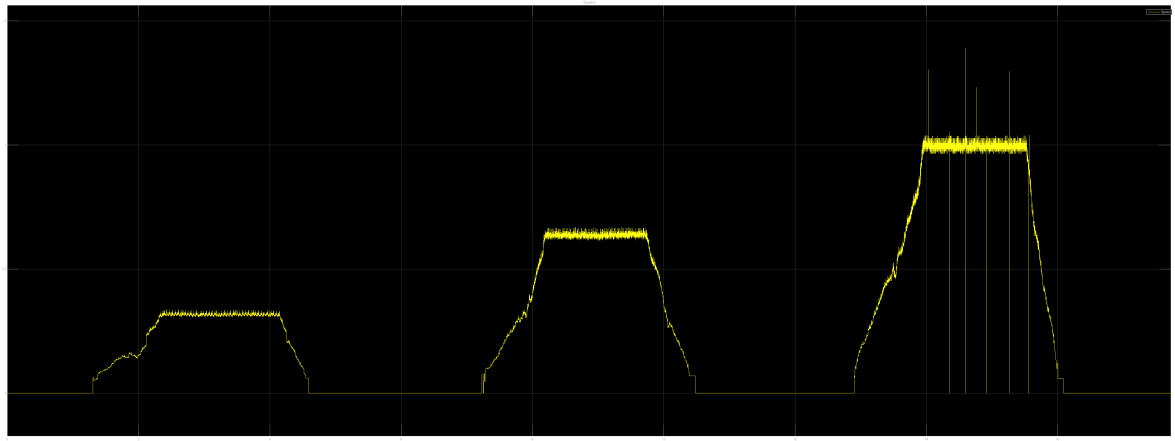


Abbildung 7.14: Übersicht der *speedModes* nach den Hardwareanpassungen

Bei der Verifikation eines *speedMode* wird die Geschwindigkeitsanforderungen jeweils voll auf-/ und abgedreht, sodass die jeweilige maximale Geschwindigkeit erreicht wird.

Verglichen mit Abbildung 7.3 ist der Signalverlauf signifikant verbessert. Jedoch sind im dritten *speedMode* noch Fehlinterpretationen vorhanden. An diesen Momenten bleibt die ermittelte Position konstant was zu einer Geschwindigkeit von Null führt. Diese Ursache konnte leider nicht weiter eingegrenzt werden.

Des Weiteren ist in Abbildung 7.3 zu sehen, dass die volle Geschwindigkeit nicht erreicht werden kann. Die 1 auf der Y-Achse repräsentiert 4000 rpm. Die Ursache hierbei ist, dass die Geschwindigkeit durch einen Potentiometer gesteuert wird. Die gemessene Spannung erreicht hierbei nicht 3.3 V, was einer maximale Geschwindigkeitsanforderung entsprechen würde. Die Ursache hierfür ist, dass die Referenzspannung der ADCs gleichzeitig den Temperatursensor und die Hall Sensoren versorgt. Dadurch können kleinste Abweichungen die angeforderte Geschwindigkeit manipulieren. Durch Änderungen bei der Aufrundung und Hinzufügen eines Filters, konnte das volle Geschwindigkeitsband verwendet werden.

In Abbildung 7.15 ist die Positionsbestimmung und ermittelte Geschwindigkeit des ersten *speedModes* zu sehen.

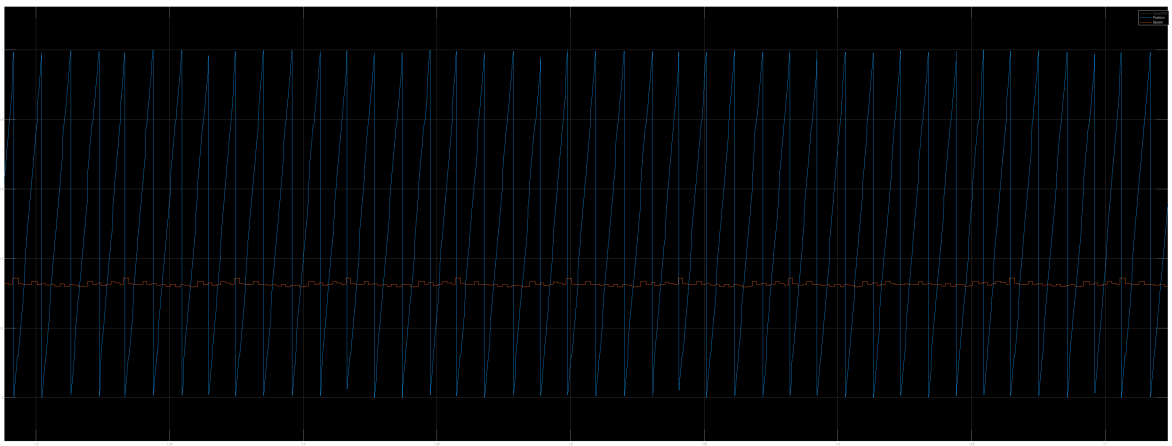


Abbildung 7.15: Positionsbestimmung und ermittelte Geschwindigkeit nach den Hardwareanpassungen

Abschließend sind im Anhang D die folgenden Systemmodellanforderungen enthalten, welche vor den Hardwareanpassungen problematisch waren.

- SW_ANF_01
- SW_ANF_03
- SW_ANF_07

8 Fazit

Die modellbasierte Entwicklung und Verifikation eines modularen Frequenzumrichters in Simulink waren aus Sicht der Software ein großer Erfolg. Die modellbasierten Tests und die Verifikation auf dem Target sind übereinstimmend.

Aufgrund der besonders modularen Entwicklung kann die Software für jegliche Applikationen verwendet werden. Mit kleinen Änderungen im Modell ist es möglich, zwischen Applikationen zu wechseln. Diese These gilt nur dann, wenn die verwendete Leistungsstufe für die einzelnen Leistungsklassen vorgesehen ist.

Das Gesamtsystem konnte bis auf kleine Probleme durchgeführt werden. Die Benutzung der von MATLAB zur Verfügung gestellten Bestimmung der PI-Regler-Koeffizienten war ein großer Erfolg. Da hierdurch das Bestimmen optimaler Werte übersprungen werden konnte.

Anfänglich problematisch waren die Hall-Signale, welche mit Rauschen überdeckt waren. Diese haben zu falschen Positions-/ und Geschwindigkeitsmessungen geführt. Nach Anpassung der Hardware wurden diese Probleme erfolgreich beseitigt.

Größter Erfolg dieser Masterarbeit war die Implementierung des External Modes über TCP. Mit diesem wurden die Systemmodellanforderungen verifiziert. Nach unerwarteten Startschwierigkeiten, wie die unterschiedlichen Pin-Belegungen, konnte nach Anpassung der gelieferten Basissoftware, das System erfolgreich zum laufen gebracht werden.

9 Ausblicke

In dieser Arbeit wurde ein Controller verwendet, welcher über zwei CPUs verfügt. Interessant wäre ein Ansatz in Multi-Core-Ausführung. Hierbei könnte beispielsweise der Torque Controller auf CPU2 laufen und die restlichen Komponenten auf CPU1.

Ist beim zukünftigen Motor das Datenblatt nicht aussagekräftig genug, sollte zunächst eine Bestimmung der Motorparameter von Anfang an geplant werden.

Ein weiterer Vorteil ist es die verwendeten Variablen in Form eines Scripts zu erstellen, da bei Anpassungen nur wenige Parameter angepasst werden müssen und alle abhängigen automatisch angepasst werden.

In Zukunft sollte ein eigenes Streckenmodell abgestimmt auf die verwendete Hardware entwickelt werden, um mit der virtuellen Umgebung nahezu das identische Verhalten zu simulieren.

Das Potential des External Modes über TCP wurde nicht voll ausgeschöpft. Es wäre interessant zu wissen, ob die Übertragung noch schneller sein kann. Falls in Zukunft die Auslastung der CPU(s) zu hoch ist und es zu signifikanten Latenzen kommt, sollte der External Mode bearbeitet werden Aktuell werden über dem IPC-Channel sämtliche Signale in Dauerschleife übertragen. Auch wenn der External Mode *nicht verwendet* wird, werden die Daten *verschickt*. Die aktuelle Auslastung der CPU1 ist nicht hoch, sodass keine Latenz aufgrund Auslastung vorliegt. Es wäre interessant herauszufinden, ob die Kommunikation über den IPC-Channel mithilfe der Register der PHY[1] für den Ethernet-Anschluss aktiviert werden kann. Bei Anschluss des LAN-Kabels könnten durch die Power-Signale der IPC-Channel aktiviert werden.

Bei Performance-Problemen könnte optional der External Mode über UDP betrieben werden. Das UDP-Protokoll garantiert keine Einhaltung der Reihenfolge der versendeten Pakete. Somit könnte die Möglichkeit bestehen, dass der External Mode über

[1] Ethernet **phy**sical layer

UDP performanter sein könnte. In diesem Fall sollte der Speedup und der Paketverlust analysiert werden.

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[1] Kevin Leiffels. »Entwicklung eines E-Skateboards«. Masterarbeit. Bochum - University of Applied Sciences, 2014/15.

[2] Florian Wagner. »Modellbasierte Entwicklung und Verifikation einer modularen Antriebsplattform mit MATLAB/Simulink«. Masterarbeit. Bochum - University of Applied Sciences, 2017.

[3] Baris Akgül und Enes Darici. »Planung und Konzeption eines modularen Frequenzumrichters«. Entwicklungsprojekt. Bochum - University of Applied Sciences, 2023.

[4] Enes Darici. »Entwicklung und Implementierung der Hardware eines modularen Frequenzumrichters«. Masterarbeit. Bochum - University of Applied Sciences, 2023.

[5] Alexey Romanov und Slaschov Bogdan. »Open source tools for model-based FPGA design«. In: *2015 International Siberian Conference on Control and Communications (SIBCON)*. 2015, S. 1–6. DOI: 10.1109/SIBCON.2015.7147193.

[6] Felix Stefan Schneider. »Modellbasierte Entwicklung einer FPGA-Logik für Echtzeit-Bildverarbeitung«. Bachelorarbeit. Bochum - University of Applied Sciences, 2018.

[7] The MathWorks. *Target to Development Computer Communication by Using TCP*. zuletzt aufgerufen am 10.November.2023. URL: https://de.mathworks.com/help/slrealtime/ug/target-to-host-communication-using-tcp.html.

[8] S Jamuna, P Dinesha und KP Shashikala. »A brief review on types and design methods of ADC«. In: *J. Eng. Res. Appl.* 8.6 (2018), S. 85–91.

[9] The MathWorks. *Configuring Acquisition Window Width for ADC Blocks*. zuletzt aufgerufen am 18.November.2023. URL: https://de.mathworks.com/help/ti-c2000/ug/configuring-acquisition-window-width-for-adc-blocks.html.

[10] Farnell An Avnet Company. *How to improve analog to digital converter accuracy*. zuletzt aufgerufen am 04.Dezember.2023. URL: https://at.farnell.com/how-to-improve-analog-to-digital-converter-accuracy.

[11] Texas Instruments. *TMS320F2838x Real-Time Microcontrollers With Connectivity Manager - Technical Reference Manual*. Literature Number: SPRUII0E.

[12] Dave Wilson Texas Instruments. *Motor Control Compendium - TI MCU Application Manager for Motor Control*. Navigating the complexities of motor control. 2010-2011.

[13] Shawon Kumar Baral. *Closed loop Control of PMSM Motor - Field Oriented Control using Hall sensors*. Masterarbeit. Linköping University. 2021.

[14] Len Trombetta. »Experimental Verification of Kirchhoff's Voltage Law and Kirchhoff's Current Law«. In: (2013).

[15] Omer Cihan Kivanc und Salih Baris Ozturk. »MATLAB Function Based Approach to FOC of PMSM Drive«. In: *2015 IEEE European Modelling Symposium (EMS)*. 2015, S. 96–102. DOI: 10.1109/EMS.2015.81.

[16] Keith W. Ross James F. Kurose. *Computer Networking A Top-Down Approach*. 6. Aufl. Pearson, 2013.

[17] imperva. *OSI Model*. zuletzt aufgerufen am 19.November.2023. URL: https://www.imperva.com/learn/application-security/osi-model/.

[18] itslot. *Erklärung: IP Adresse, Subnetzmaske, Subnetz und Standardgateway*. zuletzt aufgerufen am 19.November.2023. 2019. URL: https://www.itslot.de/2018/04/ip-adresse-subnetzmaske-router-einfache-erklaerung.html.

[19] Scott Wallask. *Quellcode (Sourcecode)*. zuletzt aufgerufen am 16.September.2023. URL: https://www.computerweekly.com/de/definition/Quellcode-Sourcecode#:~:text=Der%20Quellcode%20ist%20der%20Code,und%20anderen%20operativen%20Anweisungen%20zusammensetzt..

[20] K. Langendoen u. a. »Integrating polling, interrupts, and thread management«. In: *Proceedings of 6th Symposium on the Frontiers of Massively Parallel Computation (Frontiers '96)*. 1996, S. 13–22. DOI: 10.1109/FMPC.1996.558057.

[21] Krisitan Saether. *Elektronik-Entwicklung*. zuletzt aufgerufen am 12.November.2023. 2012. URL: https://www.all-electronics.de/elektronik-entwicklung/leistungsgrenzen-von-mikrocontrollern-ueberwinden.html.

[22] Marc Feeley. »Polling efficiently on stock hardware«. In: *Proceedings of the conference on Functional programming languages and computer architecture*. 1993, S. 179–187.

[23] Elecia White. *Making Embeded Systems: Design Patterns for Great Software*. 1. Aufl. O'Reilly Media, 2011. ISBN: 978-1-449-30214-6.

[24] The MathWorks. *Partition Motor Control for Multiprocessor MCUs*. zuletzt aufgerufen am 10.November.2023. URL: https://de.mathworks.com/help/ti-c2000/ug/partition-motor-control-example.html.

[25] Texas Instruments. *TMDSCNCD28388D F28388D evaluation module for C2000™ MCU controlCARD™*. zuletzt aufgerufen am 03.November.2023. URL: https://www.ti.com/tool/TMDSCNCD28388D.

[26] Texas Instruments. *TMDSHSECDOCK HSEC180 controlCARD baseboard docking station*. zuletzt aufgerufen am 04.November.2023. URL: https://www.ti.com/tool/TMDSHSECDOCK.

[27] Texas Instruments. *DRV8300DIPW-EVM DRV8300DIPW evaluation module for three-phase BLDC*. zuletzt aufgerufen am 04.November.2023. URL: https://www.ti.com/tool/DRV8300DIPW-EVM.

[28] The MathWorks. *Host-Target Communication with External Mode Simulation*. zuletzt aufgerufen am 05.November.2023. URL: https://de.mathworks.com/help/supportpkg/xilinxzynq7000ec/ug/set-up-and-use-hosttarget-communication-channel.html.

[29] The MathWorks. *Per-Unit System*. zuletzt aufgerufen am 15.November.2023. URL: https://de.mathworks.com/help/mcb/gs/per-unit-system.html.

[30] The MathWorks. *Per-Unit System of Units*. zuletzt aufgerufen am 15.November.2023. URL: https://de.mathworks.com/help/sps/powersys/ref/per-unit-and-international-systems-of-units.html.

[31] Circuit Globe. *Per Unit System*. zuletzt aufgerufen am 15.November.2023. URL: https://circuitglobe.com/what-is-a-per-unit-system.html.

[32] The MathWorks. *struct - Structure Array*. zuletzt aufgerufen am 15.November.2023. URL: https://de.mathworks.com/help/matlab/ref/struct.html.

[33] NATIONAL INSTRUMENTS CORP. *Specifying the Word Length and Integer Word Length (Digital Filter Design Toolkit)*. zuletzt aufgerufen am 15.November.2023. URL: https://www.ni.com/docs/de-DE/bundle/labview-digital-filter-design-toolkit-api-ref/page/lvdfdtconcepts/specify_wl_iwl.html.

[34] ebmpapst. *ECI-Motor - ECI 63.60*. Seite 85.

[35] Texas Instruments. *TMP23xLow-Power, High-Accuracy Analog Output Temperature Sensors*. Revised May 2019. SBOS857E, 2017.

[36] Maxim Integrated. *Application Note 748: The ABCs of ADCs: Understanding How ADC Errors Affect System Performance*. zuletzt aufgerufen am 16.November.2023. 2002. URL: https://www.analog.com/media/en/technical-documentation/tech-articles/the-abcs-of-analog-to-digital-converters-how-adc-errors-affect-system-performance--maxim-integrated.pdf.

[37] The MathWorks. *Field-Oriented Control of PMSM Using Hall Sensor*. zuletzt aufgerufen am 28.November.2023. URL: https://de.mathworks.com/help/mcb/gs/foc-pmsm-using-hall-sensor-example.html.

[38] The MathWorks. *How to Use Hall Validity and Hall Decoder Blocks*. zuletzt aufgerufen am 17.November.2023. URL: https://de.mathworks.com/help/mcb/gs/using-hall-validity-hall-decoder-blocks.html.

[39] The MathWorks. *Hall Offset Calibration for PMSM*. zuletzt aufgerufen am 28.November.2023. URL: https://de.mathworks.com/help/mcb/gs/hall-offset-calibration-pmsm-motor.html.

[40] Texas Instruments. *TMS320F2838x Real-Time Microcontrollers With Connectivity Manager*. zuletzt aufgerufen am 05.November.2023. URL: https://www.ti.com/lit/ds/symlink/tms320f28388d.pdf?ts=1678365254973&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FTMS320F28388D.

[41] Devin Cottier Texas Instruments. *TMS320F28379D: ADC acquisition window length and position with SOC triggered by ePWM*. zuletzt aufgerufen am 28.November.2023. URL: https://e2e.ti.com/support/microcontrollers/c2000-microcontrollers-group/c2000/f/c2000-microcontrollers-forum/618008/tms320f28379d-adc-acquisition-window-length-and-position-with-soc-triggered-by-epwm.

[42] Dorin O Neacsu. »Space vector modulation-An introduction«. In: *IECON*. Bd. 1. 2001, S. 1583–1592.

[43] Texas Instruments. *DRV8300: 100-V Three-Phase BLDC Gate Driver*. zuletzt aufgerufen am 16.November.2023. 2020.

[44] Cypress Semiconductor Corporation. *Pulse Width Modulator (PWM)*. Document Number: 001-73568 Rev. 198 Champion Court, Revised 2011.

[45] Vieri Xue. »Center-aligned SVPWM realization for 3-phase 3-level inverter«. In: *Application Report SPRABS6 October* 18 (2012).

[46] The MathWorks. *Byte Pack - Convert input signals to 8-, 16-, or 32-bit vector*. zuletzt aufgerufen am 28.November.2023. URL: https://de.mathworks.com/help/soc/ref/bytepack.html#.

[47] The MathWorks. *Byte Unpack - Unpack 8-, 16-, or 32-bit input vector to multiple output vectors*. zuletzt aufgerufen am 28.November.2023. URL: https://de.mathworks.com/help/soc/ref/byteunpack.html.

[48] Stefan Kerber. Schriftliche und mündliche Kommunikation mit der Firma The Mathworks zwischen Mai 2023 und November 2023.

[49] The MathWorks. *Hall Speed and Position - Compute speed and estimate position of rotor by using Hall sensors*. zuletzt aufgerufen am 29.November.2023. URL: https://de.mathworks.com/help/mcb/ref/hallspeedandposition.html#.

[50] Texas Instruments. *LAUNCHXL-F280049C - F280049C LaunchPad™ development kit C2000™ Piccolo™ MCU*. zuletzt aufgerufen am 04.Dezember.2023. URL: https://www.ti.com/tool/LAUNCHXL-F280049C.

A Anhang

CarPediem\...

1. reqs\Umfeldmodell.pdf
2. reqs\Use_Cases.pdf
3. reqs\Lastenheft_CarPediem.pdf
4. reqs\Systemmodellanforderungen_CargoPedelec.pdf
5. milestone \SW_Meilensteinplan.xlsx
6. models \impmodel.slx
7. models \params_f2838.mat
8. models \TCP_Server.slx
9. models \TCP_Client.slx
10. models \Serial_host_model.slx
11. models \F280049C
12. ethernetConfigs \config_CC_Custom.xlsx
13. ipcMessages \IPC_messages.xlsx
14. testing \Testfile_F2838.mldatx
15. testing \CarPediem_TestReport.pdf
16. testing \2023_06_26_Systemmodellanforderungen_CargoPedelec_2_.slreqx
17. canMessages \CAN_messages.xlsx

B Anhang

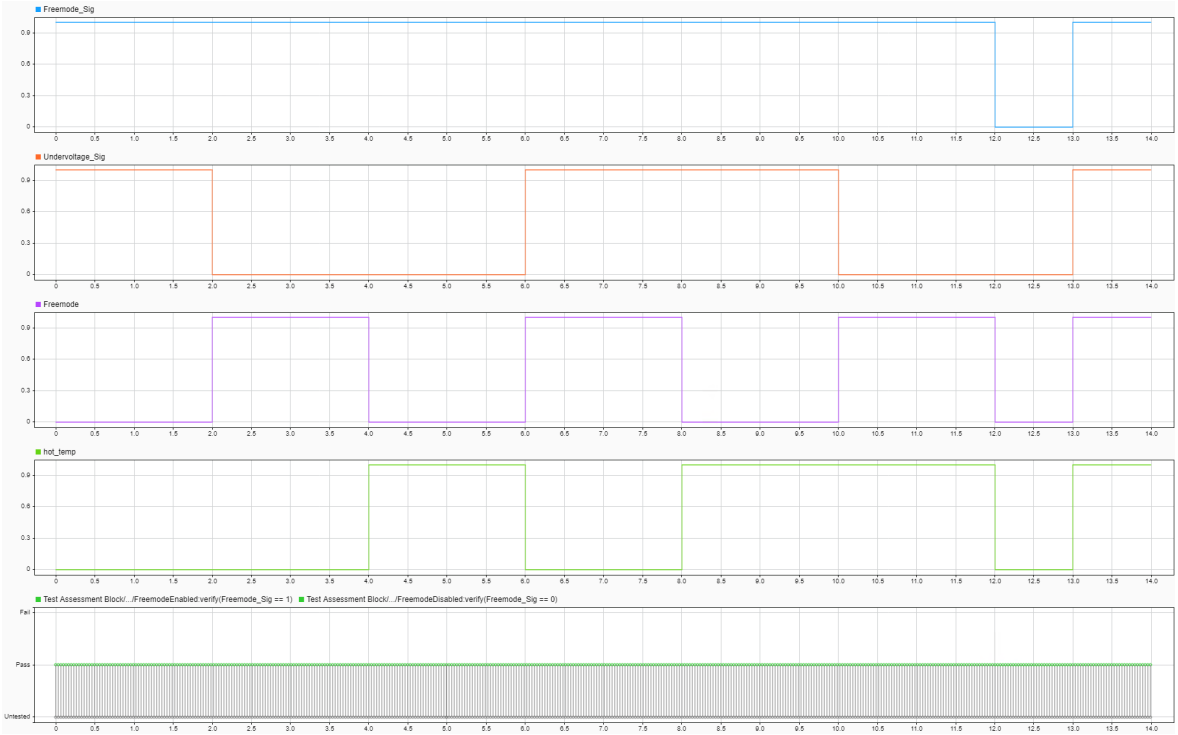


Abbildung B.1: Testauswertung des *Digital_Freemode_Checks*

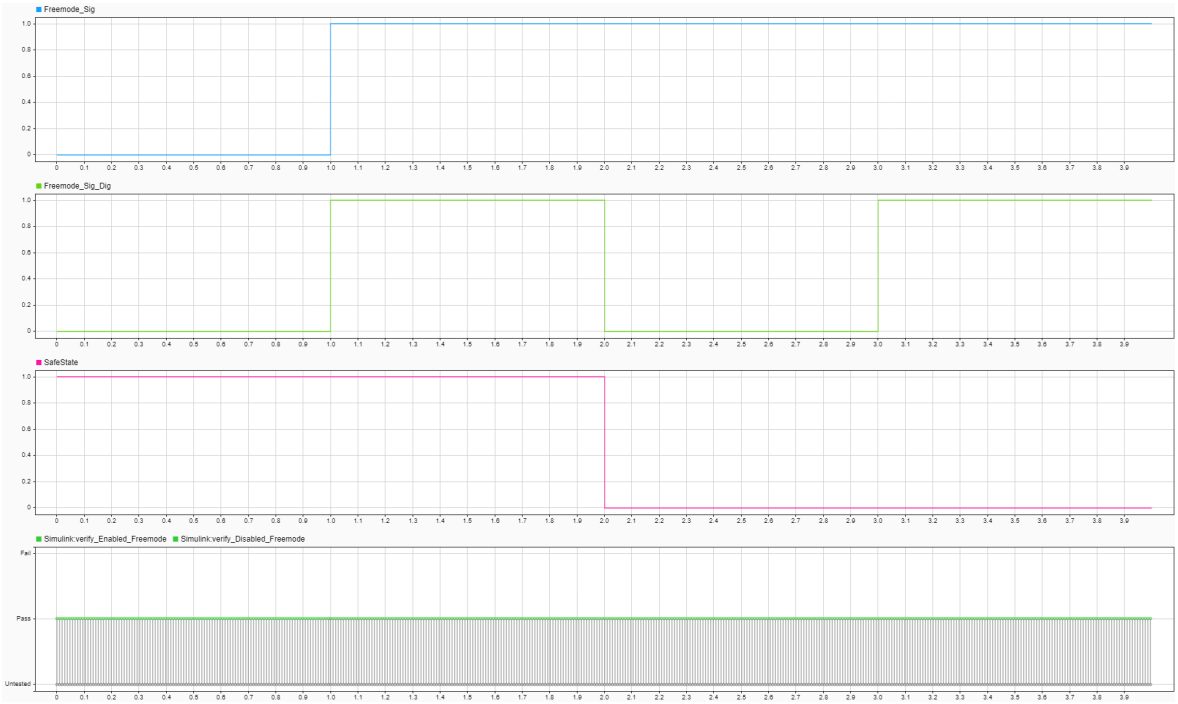


Abbildung B.2: Testauswertung des *Mechanical_Freemode_Checks*

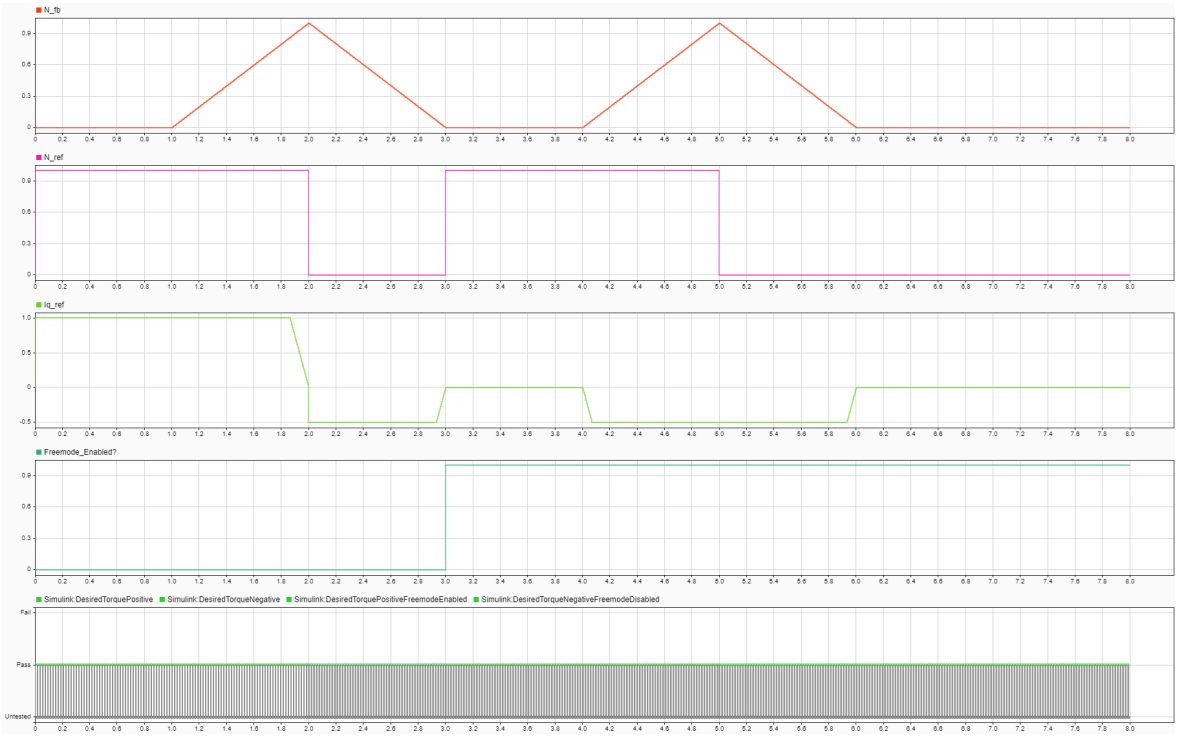


Abbildung B.3: Testauswertung des Speed Controllers

C Anhang

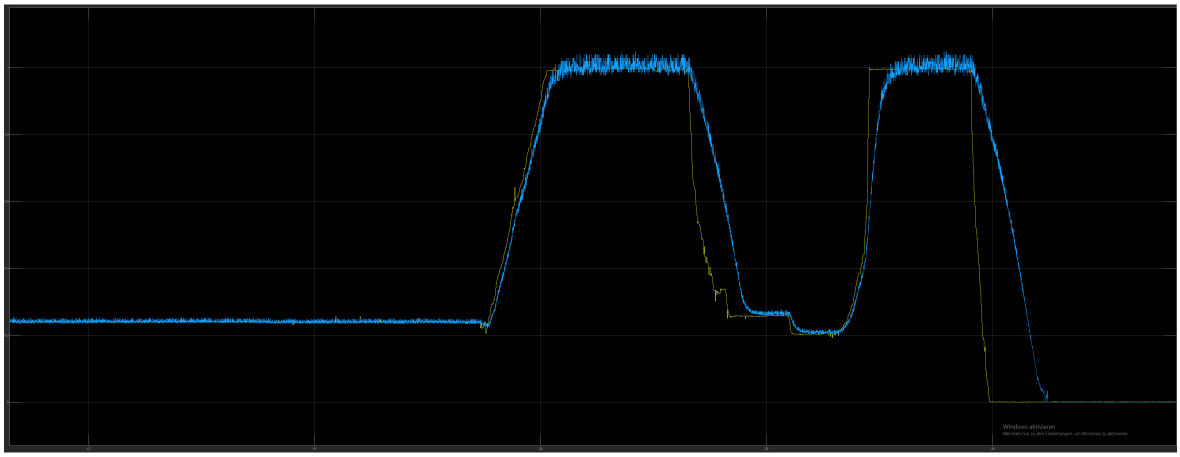


Abbildung C.1: Geschwindigkeitsverlauf auf dem LaunchPad F280049C

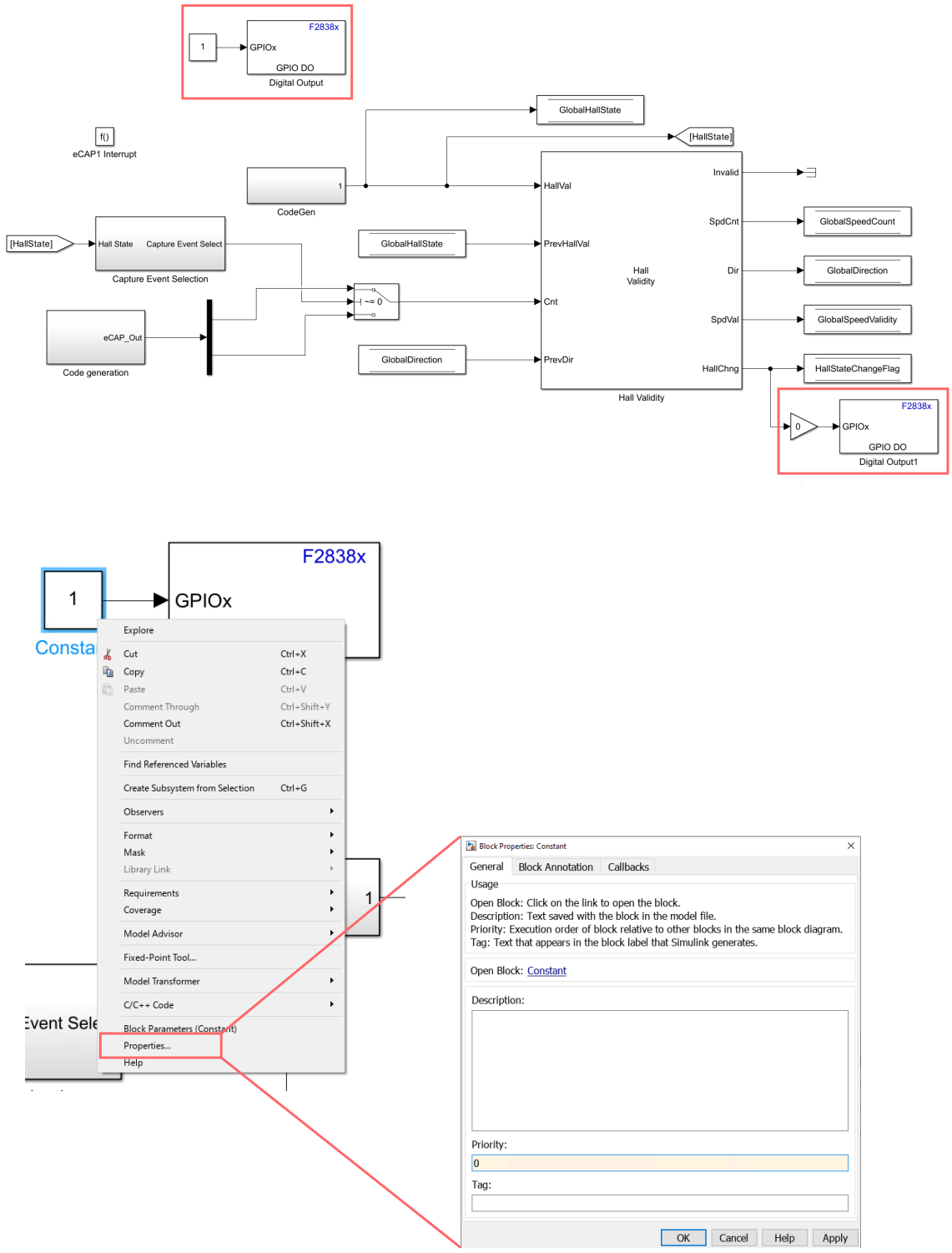


Abbildung C.2: Realisierung der Interrupt-Messung

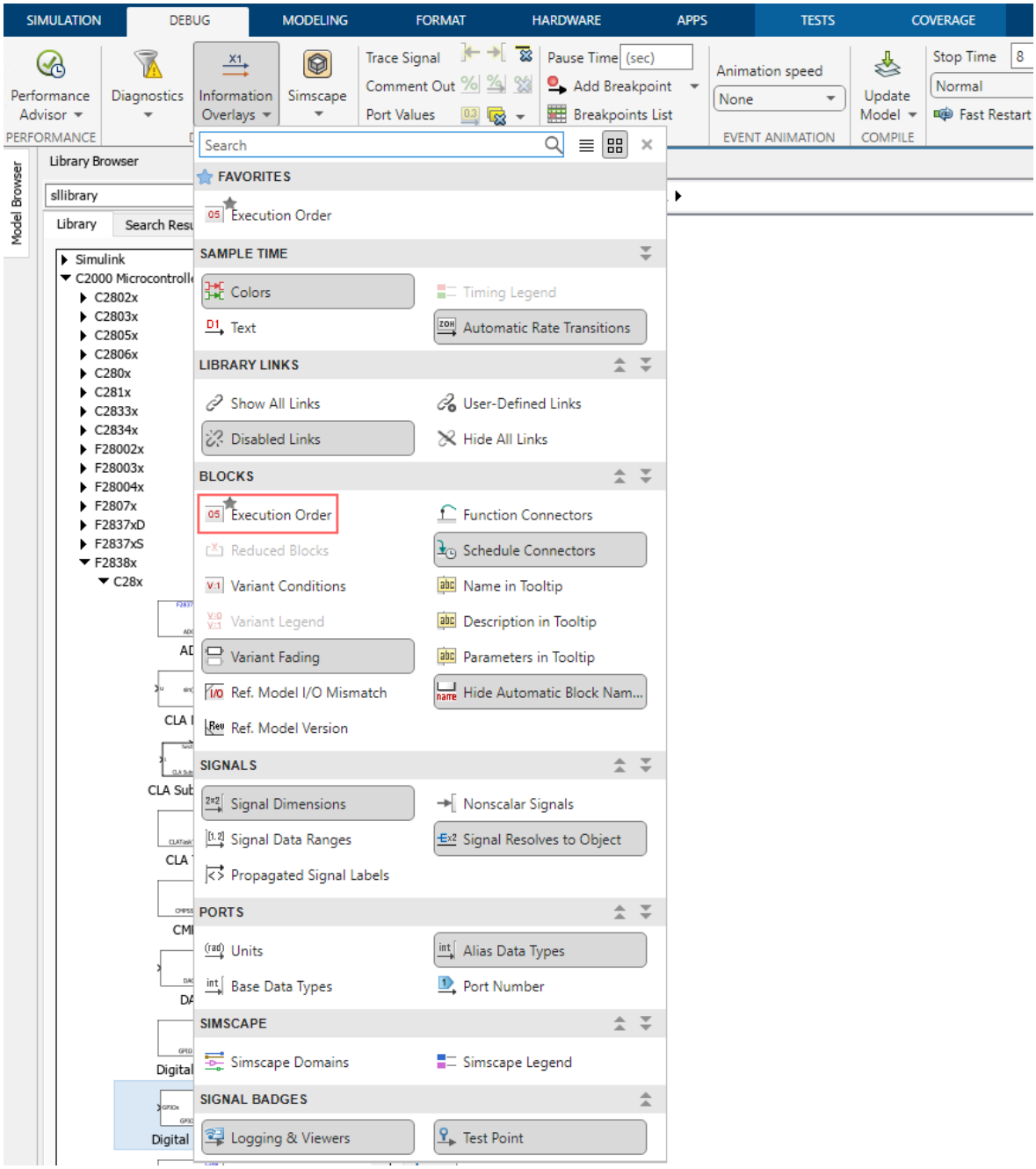


Abbildung C.3: Aufruf der *Execution Order*

D Anhang

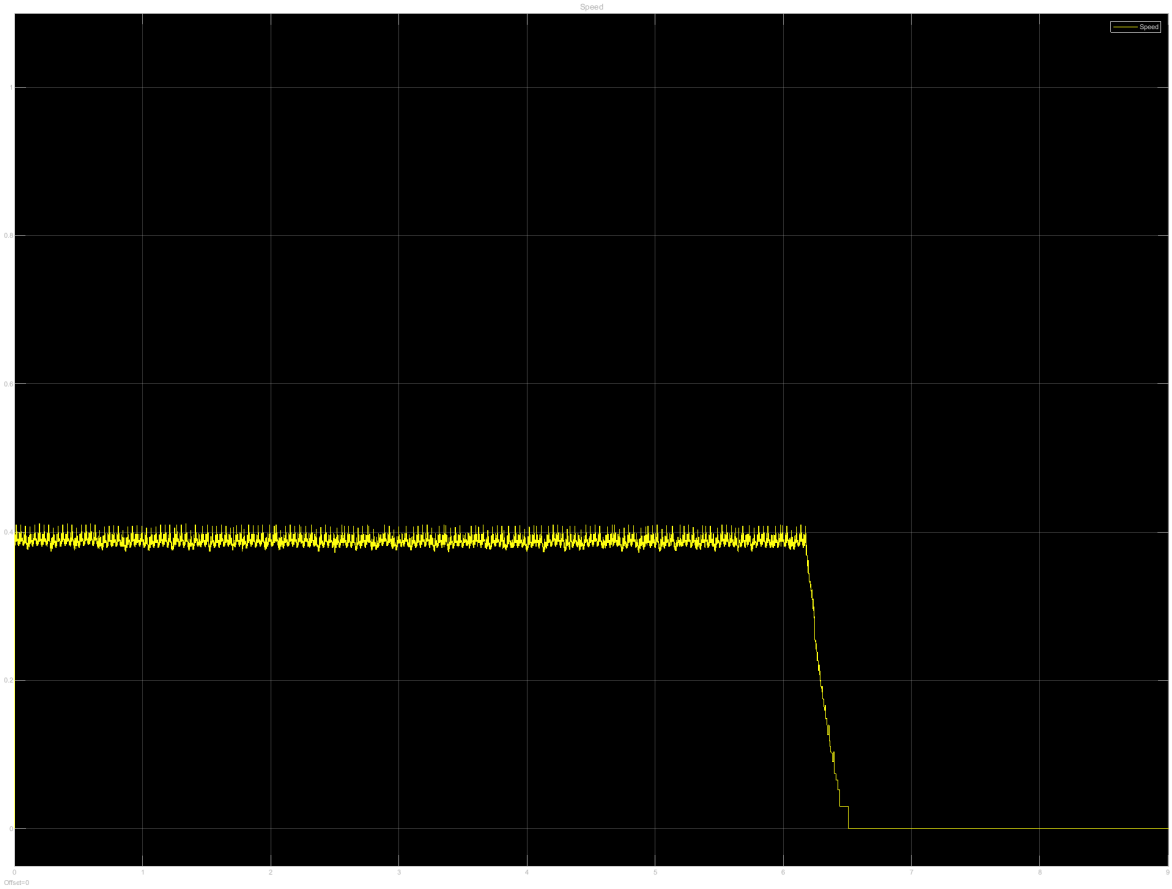


Abbildung D.1: Verifikation der Systemmodellanforderung SW_ANF_01 - *Freemode*

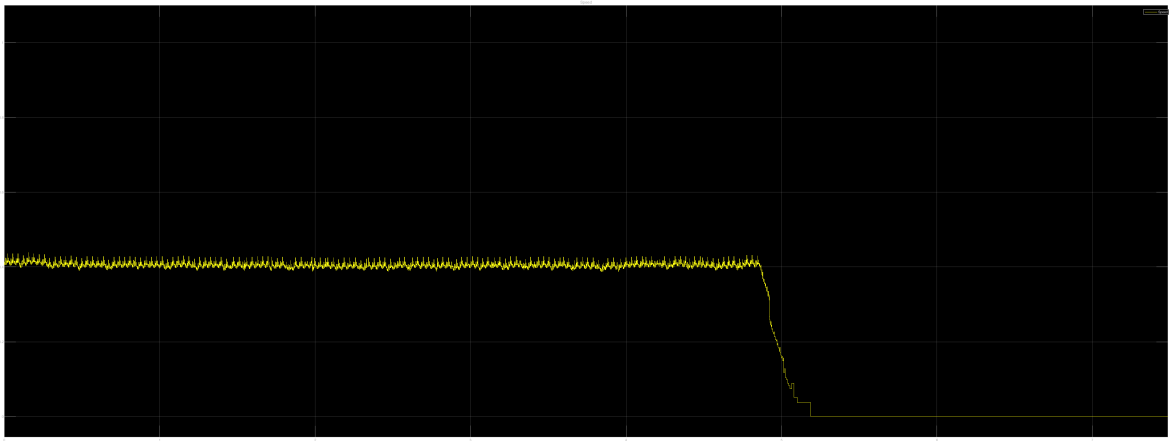


Abbildung D.2: Verifikation der Systemmodellanforderung SW_ANF_03 - *SafeState*

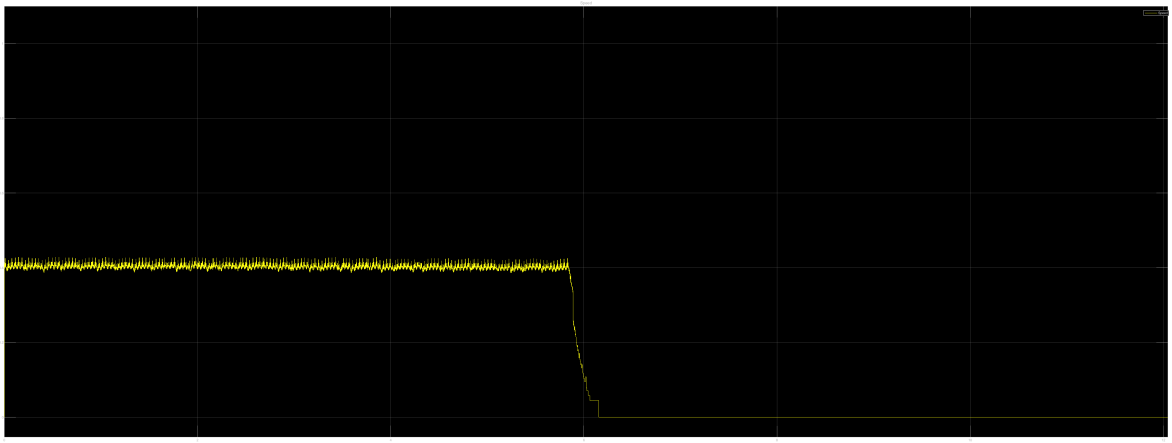


Abbildung D.3: Verifikation der Systemmodellanforderung SW_ANF_07 - *BatteryVoltage*

E Anhang

```
#ifdef CARPEDIEM_PIN_CORRECTION
/**
* This function corrects the gpio number.
*/
Uint16 getCarPediemGPIONumber(Uint16 gpioNumber) {
  switch (gpioNumber) {
    // TX_EN
    case 118: return 45;
    // TX_CLK
    case 44: return 44;
    // TX_D0
    case 75: return 75;
    // TX_D1
    case 122: return 74;
    // TX_D2
    case 123: return 73;
    // TX_D3
    case 124: return 72;
    // RX_DV
    case 112: return 70;
    // RX_ER
    case 113: return 71;
    // RX_CLK
    case 111: return 69;
    // RX_D0
    case 114: return 52;
    // RX_D1
    case 115: return 53;
    // RX_D2:
    case 116: return 54;
    // RX_D3:
    case 117: return 55;
```

```
33            // COL
34            case 110: return 41;
35            // CRS
36            case 109: return 40;
37            // MDC
38            case 105: return 42;
39            // MDIO
40            case 106: return 43;
41            // INT_PWDN
42            case 108: return 68;
43            // RESET
44            case 119: return 46;
45            // Otherwise - do not modify gpio number
46            default: return gpioNumber;
47          }
48
49        }
50
```

F Anhang

```
Uint16 getCarPediemMuxPosition(Uint16 gpioNumber, Uint16 muxPosition){
  switch (gpioNumber) {
  // TX_EN
  case 45: return 0xBU;
  // TX_CLK
  case 44: return 0xBU;
  // TX_D0
  case 75: return 0xBU;
  // TX_D1
  case 74: return 0xBU;
  // TX_D2
  case 73: return 0xBU;
  // TX_D3
  case 72: return 0xBU;
  // RX_DV
  case 70: return 0xBU;
  // RX_ER
  case 71: return 0xBU;
  // RX_CLK
  case 69: return 0xBU;
  // RX_D0
  case 52: return 0xBU;
  // RX_D1
  case 53: return 0xBU;
  // RX_D2:
  case 54: return 0xBU;
  // RX_D3:
  case 55: return 0xBU;
  // COL
  case 41: return 0xBU;
  // CRS
  case 40: return 0xBU;
```

```
33      // MDC
34      case 42: return 0xAU;
35      // MDIO
36      case 43: return 0xAU;
37      // INT_PWDN
38      case 68: return 0xBU;
39      // RESET
40      case 46: return 0xBU;
41      // Default - do not modify
42      default: return muxPosition;
43      }
44  }
45
```