Technische Universität Wien

Diplomarbeit

INFORM-Regelung einer permanenterregten Synchronmaschine; C-Codegenerierung mittels MATLAB/Simulink

SCHRÖDL, Manfred; O.Univ.Prof. Dipl.-Ing. Dr.techn.
und ASCH, Daniel; Dipl.-Ing. Mag.rer.soc.oec.
am Institut für Elektrische Energiesysteme und Elektrische Antriebe
Gußhausstraße 25-29/370-2
1040 Wien, Österreich

von:
David Kröter, BSc.
Matr.Nr.: 0526238
Wasserburgergasse 5/14
1090 Wien
17. Oktober 2011

Vorwort

Die vorliegende Diplomarbeit wurde im Zeitraum von März bis September 2011 am Institut für Energiesysteme und Elektrische Antriebe unter der Leitung von O.Univ.Prof. Dipl.-Ing. Dr.techn. Manfred Schrödl verfasst, dem für die umfangreichen Hilfestellungen während der Arbeit und die interessante Themenstellung mein besonderer Dank gilt. Des Weiteren möchte ich sowohl meinen Betreuer Dipl.-Ing. Mag.rer.soc.oec. Daniel Asch und Dipl.-Ing. Wolfgang Staffler hervorheben, die mir im Laufe der Arbeit mit ausgeprägtem Sachverständnis zur Seite standen und zugleich durch ihre persönliche Kompetenzen für ein hervorragendes Arbeitsklima sorgten.
Allen voran bedanke ich mich bei meinen Eltern Werner und Irene, die mir mein Studium durch ihre finanzielle Unterstützung erst ermöglichten und mir in schwierigen Zeiten beistanden. Ferner gilt mein Dank meiner Freundin Stephanie, meinem Bruder Manuel und meinem Studienkollegen Christian.

Liebe Leserin, lieber Leser,
das Genderthema ist umfassend und wird aktuell in der Technik immer bewusster. Leider lässt es sich in seiner Vielfalt und Komplexität nur schwer grammatikalisch erfassen, wenn beide Geschlechter gleichsam angesprochen werden. In diesem Bewusstsein benutze ich im Weiteren vereinfachend die männliche Form, um einen durchgängigen Lesefluss zu garantieren. Alle Leserinnen bitte ich hiermit um Verständnis.

Wien im Oktober 2011

David Kröter

Nomenklatur

Abkürzungen

ADC ..... Analog-Digital-Converter
AHC ..... Active High Complementary
API ..... Application Programmer Interface
ASCII ..... American Standard Code for Information Interchange
ASM ..... Asynchronmaschine
BNC ..... Bayonet Neill-Concelman Anschluss
CAN ..... Controller Area Network
CPU ..... Central Processing Unit
DSP ..... Digital Signal Processor
HANSL ..... Hochsprache für Anspruchsvolle Numerische SystemLösungen
HTML ..... HyperText Markup Language
IC ..... Integrated Circuit
INFORM ..... Indirekte Flussermittlung durch On-line Reaktanz-Messung
ISR ..... Interrupt Service Routine
JTAG ..... Joint Test Action Group
LCT ..... Legacy Code Tool
LSB ..... Least Significant Bit
MATLAB ..... Matrix Laboratory
MOSFET ..... Metal-Oxide-Semiconductor Field-Effect-Transistor
MSB ..... Most Significant Bit
PLL ..... Phase Locked Loop
PSM ..... Permanenterregte Synchronmaschine
PWM ..... Pulse Width Modulation
SCI ..... Serial Communication Interface
SPI ..... Serial Peripherial Interface
UML ..... Unified Modeling Language

Formelzeichen

γ ..... Lagewinkel des Rotors [rad]
$\Im$ ..... Imaginäranteil [1]
Ω ..... Kreisfrequenz [$\frac{\mathrm{rad}}{\mathrm{s}}$]
Φ ..... magnetischer Fluss [Vs]
Ψ ..... magnetischer Verkettungsfluss [Vs]
$\Re$ ..... Realanteil [1]

I ..... Elektrischer Strom [A]
i ..... normierter elektrischer Strom [1]
j ..... Imaginäre Einheit $\sqrt{-1}$ [1]
L ..... Induktivität [H]
l ..... normierte Induktivität [1]
M ..... Drehmoment [Nm]
R ..... Elektrischer Widerstand [Ω]
r ..... normierter Widerstand [1]
U ..... Elektrische Spannung [V]
u ..... normierte elektrische Spannung [1]
Z ..... elektrische Impedanz [1]

Indizes

α, β ..... Statorfestes Koordinatensystem, Raumzeigerrechnung
$Bezug$ ..... Bezugsgröße
d, q ..... Rotorfestes Koordinatensystem, Raumzeigerrechnung
el ..... Elektrische Größe
M ..... Von Dauermagneten her rührend
m ..... Mechanische Größe
N, Str ..... Nennangabe zu jeweiliger Größe, Strangwert
r ..... Größe bezogen auf den Rotor, Raumzeigerrechnung
s ..... Größe bezogen auf den Stator, Raumzeigerrechnung
$Skalierung$ ... Skalierungsfaktor des ADC-Rohwerts
u, v, w ..... Bezeichner der Phasen U,V,W
ZK ..... Zwischenkreis

Kurzfassung

Die folgende Diplomarbeit behandelt die sensorlose Regelung einer permanenterregten Synchronmaschine mittels einer Kombination der am Institut für Energiesysteme und Elektrische Antriebe entwickelten INFORM-Methode und des EMK-Modells. Beide Verfahren dienen der sensorlosen Ermittlung des Rotorflusswinkels der Maschine, jedoch in unterschiedlichen Drehzahlbereichen.
Der Quellcode des digitalen Signalprozessors wird hierbei, anders als bei bisherigen Projekten am Institut, nicht herkömmlich zeilenbasiert programmiert, sondern mit Hilfe des Entwicklerwerkzeugs MATLAB/Simulink aus einem Regelungsmodell generiert. Dieser Ansatz wird neben der Übersichtlichkeit der verwendeten Blockstrukturen auch durch die Unabhängigkeit bezüglich der verwendeten Hardware gestützt: Anhand mitgelieferter Bausteine der Simulink-Bibliotheken ist ein Wechsel auf eine unterschiedliche Rechnerarchitektur leicht möglich. Ohne detailliertes Wissen über den gewählten Prozessor kann das Regelungsmodell über Dialogfenster umfangreich konfiguriert werden. Trotzdem bleiben in der Flexibilität in Spezialbereichen, wie es die INFORM-Methode darstellt, bei alleiniger Verwendung der mitgelieferten Funktionalitäten Lücken bestehen, welche durch Einbindung benutzerspezifischer CQuellcode-Fragmente geschlossen werden können. Neben der eben erwähnten Integration von benutzereigenem Code über sogenannte S-Funktionen wird die schrittweise Umsetzung der Regelstruktur in MATLAB/Simulink dargestellt: Ausgehend von der Inbetriebnahme der verwendeten Hardware wird die Ansteuerung des Zwischenkreisumrichters, die Implementierung des Stromreglers und in Folge die Umsetzung des INFORM-Verfahrens dokumentiert. Abschließend sind die Ergebnisse der Codegenerierung denen der konventionellen Programmierung gegenübergestellt.

Schlagwörter: PSM, sensorlose Regelung, INFORM-Methode, EMK-Modell, MATLAB/Simulink, Real Time Workshop, Codegeneration

Abstract

The following thesis outlines the sensorless control of a permanent synchronous motor by combining the results of the INFORM method, which was developed at the Institue of Electrical Drives and Energy Systems, and the EMF-model. Both techniques achieve a sensorless identification of the current rotor position for the electrical drive, but are intentionally designed for different speed levels.
Instead of programming the source code for the digital signal processor the usual way, as it had been done in many preceeding projects at the institute, the source files emerge directly from the control model by means of the engineering tool MATLAB/Simulink. The use of this approach is based on the fact that accessing a block structure is much easier to grasp as well as the independence from being bound to certain hardware: Using predesigned functions from Simulink block libraries permits a fast and simple changeover to another processor architecure. Even without detailled knowledge about the chosen processor, the control model can be configured extensively using the provided dialogue forms. Even though a high level of flexibility is given, some gaps remain open as soon as only pre-built blocks are utilised, especially whilst dealing with special topics such as the INFORM method. This drawback can be overcome by embedding user-specific source code fragments into the control model. Besides the integration of existing code, namely S-functions, the stepwise implementation of the control mechanism in MATLAB/Simulink will be described: Beginning from the start-up of the involved hardware, the paper documents the electrical actuation for the frequency converter, the realisation of both the current control structure and the INFORM-method. For a final conclusion, results of code generation are compared to conventionally programming the source code.

Keywords: PSM, sensorless control, INFORM method, EMF model, MATLAB/Simulink, Real Time Workshop, code generation

Inhaltsverzeichnis

1. Aufgabenstellung

Die permanenterregte Synchronmaschine ist in der heutigen Antriebstechnik ein weit verbreiteter Maschinentyp in den unterschiedlichsten Industriesparten, da sie universell einsetzbar, im hochdynamischen Betrieb leicht zu regeln und überdies nahezu wartungsfrei ist[1]. Die große Anzahl der involvierten Industriebereiche resultiert folglich in einem interdisziplinären Zusammenspiel von Fachkräften unterschiedlichster Ausbildungsrichtungen. Für Elektrotechniker und Informatiker mag vorliegender Quellcode für die Regelung einer Maschine/Produktionsstraße ohne großen Aufwand les- und ebenso nachvollziehbar sein, doch falls beispielsweise Personal der betriebswirtschaftlichen Ebene zu technischen Besprechungen hinzugezogen wird, ist dies nicht unbedingt der Fall. Die Kompetenz der Erfüllung von definierten Spezifikationen liegt hier einzig und allein beim Programmierer bzw. den Programmierenden selbst. Dieses Defizit in der Designphase einer Produktlinie hat im letzten Jahrzehnt zu einem neuen Ansatz des Softwaredesigns geführt. Zwar hat die Verwendung von Modellen in der Softwareentwicklung eine lange Tradition und spätestens seit der Definition der „Unified Modeling Language“ (UML) eine zunehmende Verbreitung, doch in aller Regel handelte es sich dabei zuerst „nur“ um eine Dokumentation von Softwaresystemen, weil es lediglich eine gedankliche Verbindung zwischen Modell und Softwareimplementierung gab[2]. Die konkrete Realisierung obliegt, wie bereits zuvor erwähnt, dem Programmierer. Der Evolutionsschritt von *modellbasierter* zu *modelgetriebener* Softwareentwicklung rückte das abstrahierte Modell eines designierten Endprodukts mehr in den Mittelpunkt, da dieses den zentralen Ausgangspunkt diverser Phasen der Entwicklung bildet und zugleich von unterschiedlichen Personengruppen aus unterschiedlichen Blickwinkeln betrachtet wird (siehe Abb. 1.1).
Der Vorteil, der sich aus dem oben genannten Sachverhalt ergibt, ist, dass die Beteiligten näher zusammenrücken und sich aktiv am Entstehungsprozess der Software beteiligen können. Dem Programmierer kommt nun eine abgeänderte Rolle zu: er hat die Werkzeuge und Toolchains so zu bedienen, dass aus dem Modell am Ende lauffähiger Programmcode erzeugt wird, der dem eines von Hand programmierten Quellcodes in Funktionalität, Lesbarkeit und Laufzeit in Nichts nachsteht.
Die Aufgabe dieser Arbeit besteht darin, den soeben beschriebenen Ansatz auf eine Regelstruktur einer PSM anzuwenden, welche bisher am Institut für Energiesysteme und Elektrische Antriebe ausschließlich mittels konventioneller C-Programmierung umgesetzt wurde. Um die Leistungsfähigkeit dieser Herangehensweise zu demonstrieren, soll die PSM mit INFORM- und EMK-Verfahren sensorlos geregelt betrieben werden können. Anhand gewonnener Messdaten kann im Anschluss ein Vergleich zu bestehenden Projekten gezogen werden. Als wichtigstes Werkzeug wird das in Ingenieurswissenschaften bewährte Programm MATLAB/Simulink in Kombination mit diversen Erweiterungen eingesetzt. Für eine detaillierte

[1] Vgl. Schrödl (1998): Elektrische Antriebe und Maschinen, S.218
[2] Vgl. Stahl/Völter/Efftinge/Haase (2007): Modellgetriebene Softwareentwicklung, S.3ff

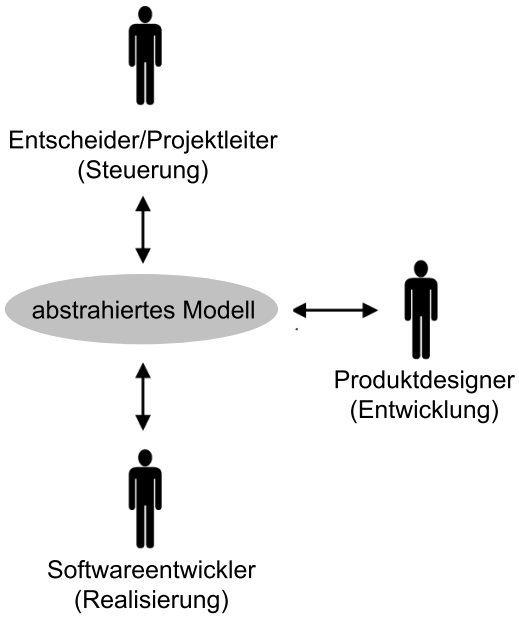

Abb. 1.1: Beteiligte Parteien bei der modellgetriebenen Softwareentwicklung

Beschreibung des kompletten Softwarepakets wird auf das Kapitel 3.2 verwiesen.
Weiters hat diese Arbeit nicht nur den Nachweis der Machbarkeit der modellgetriebenen Programmierung einer PSM-Regelung zur Aufgabe, sondern auch den Zweck eines Leitfadens zur Realisierung weiterer Projekte am Institut für Energiesysteme und Elektrische Antriebe auf der Basis von MATLAB/Simulink. Deshalb ist der logische Aufbau dem einer Inbetriebnahme-Anleitung nachempfunden. Wichtige Punkte und häufige Fehlerquellen sind in rot markierter Schrift hervorgehoben.

2. Theoretische Grundlagen der PSM

Um die späteren Kapitel in einen klar nachvollziehbaren Kontext zu stellen, werden im Kapitel 2 zuerst die mechanischen, mathematischen und regelungstechnischen Grundlagen wiederholt, welche für die PSM relevant sind.

2.1. Mechanischer Aufbau und Funktionsweise

Die PSM unterscheidet sich von den anderen Drehfeldmaschinen vor allem in jenem Punkt, dass der Rotorfluss durch Permanentmagneten hervorgerufen wird und zumindest diese Komponente des Flusses somit zu jedem Zeitpunkt vorhanden und räumlich konstant ist. Elektrischer Fluss, der u.a. für die Drehmomentbildung verantwortlich ist, muss nicht erst wie bei der Asynchronmaschine durch einen elektrischen Strom eingeprägt werden, was die Regelung im hochdynamischen Betrieb vereinfacht und in einem hohen Wirkungsgrad resultiert[3]. Weiters zeichnet sich die PSM durch Kennzeichen wie

- kompakte Bauweise
- hohe Effizienz
- nahezuhe Wartungsfreiheit aufgrund des Wegfalls eines mechanischen Kommutators,
- hohe kurzzeitige Überlastungsfähigkeit und
- große Drehzahlen

aus. Allerdings müssen die genannten Vorteile durch einen höheren Preis im Vergleich zur ASM erkauft werden.

Im Zuge dieser Arbeit wird eine Außenläufermaschine verwendet, deshalb sind die folgenden Skizzen ebenso für diesen Maschinentypus ausgelegt. Die prinzipielle Funktionsweise unterscheidet sich bei Innen- und Außenläufern jedoch nicht. In Abbildung 2.1 ist der typische Aufbau schematisch dargestellt.

[3] Vgl. Schrödl (1998): Elektrische Antriebe und Maschinen, S.218

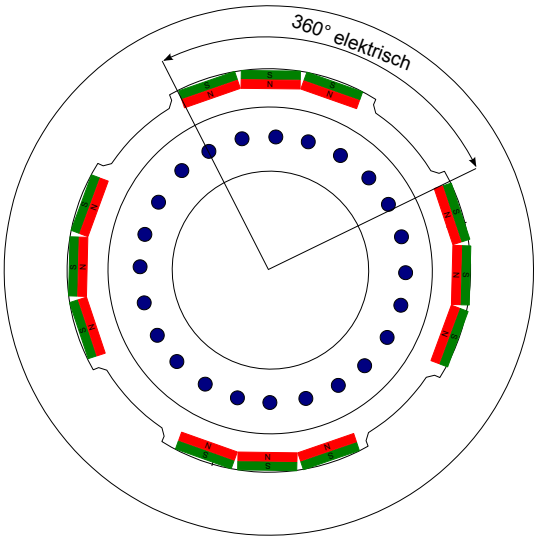

Abb. 2.1: Stator der PSM mit Drehstromwicklungen (blau), sowie Rotor mit Dauermagneten (rot/grün)

Die Dauermagnete, hier in rot (Nordpol) und grün (Südpol) gefärbt, sind innen entlang des Rotors aufgeklebt. Der Stator ist geblecht ausgeführt, um Wirbelstromverluste zu minimieren. Er beinhaltet die Drehstromwicklungen, welche die gewünschte Feldverteilung im Luftspalt und dem restlichen magnetischen Kreis bewirkt. Da das Drehmoment im Luftspalt erzeugt wird und der Außenläufermotor i.A. einen größeren Radius aufweist, ist sein Drehmoment bei gleicher Baugröße höher als bei Innenläufermotoren.

2.2. Mathematische Beschreibung im Raumzeigerkalkül

2.2.1. Die Raumzeigerdarstellung

Die mathematischen Beschreibungen, die dieser Arbeit zugrunde liegen, gehen von einem symmetrischen Drehstromsystem aus. Betrachtet man dieses System rein mit den drei linear voneinander abhängigen Sinusgrößen, gestaltet sich die Steuerung oder Regelung unnötig aufwendig, da diese Größen zeitlich und räumlich voneinander abhängen. Rechnet man hingegen mit dem zu diesem Zweck von Kovács/Rácz eingeführten Raumzeigerkalkül, so reduzieren sich die Gleichungen mittels der Raumzeigertransformation auf ein komplexes Koordinatensystem[4]. Zu beachten ist, dass die Raumzeigerrechnung räumlich sinusförmige Größen in der komplexen Ebene abbildet, ohne eine Einschränkung an den zeitlichen Verlauf der Größe zu treffen[5]; Die elektromagnetischen Größen werden in der Maschine als sinusförmig verteilt angenommen. Im Zusammenhang mit der PSM muss darauf hingewiesen werden, dass die Raumzeigerrechnung nur die Grundwelle der Größen erfasst, da die Dauermagneten eine trapezförmige Induktionsverteilung hervorrufen.

[4]Vgl. Kovács/Rácz (1959): Transiente Vorgänge in Wechselstrommaschinen
[5]Vgl. Schrödl (1998): Elektrische Antriebe und Maschinen, S.8ff

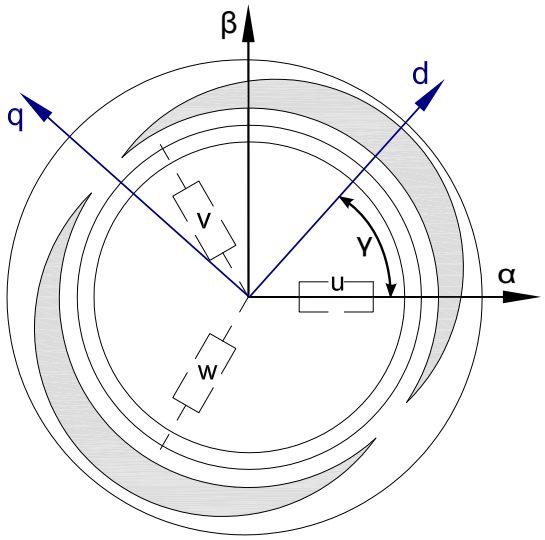

Abb. 2.2: Lage des α-β- und des d-q-Koordinatensystems der Raumzeigerrechnung bei einer zweipoligen Ersatzmaschine

Abbildung 2.2 ist zu entnehmen, dass

- die reale α-Achse des statorfesten Koordinatensystems in Richtung des Stranges U zeigt, die imaginäre β-Achse rechtwinklig dazu liegt.
- die d-Achse des rotorfesten Koordinatensystem per Konvention in Richtung der Magnetisierung der Dauermagneten und die q-Achse weiters orthogonal zur d-Achse angeordnet ist.

Um von Stranggrößen auf die Raumzeigerdarstellung zu gelangen, wendet man untenstehende Transformation 2.1, auch *Clarke-Transformation* genannt, an:

$$\begin{bmatrix} x_\alpha \\ x_\beta \end{bmatrix} = \frac{2}{3} \begin{bmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \end{bmatrix} \begin{bmatrix} x_u \\ x_v \\ x_w \end{bmatrix} \tag{2.1}$$

oder in komplexer Darstellung (mit $\underline{a} = \mathrm{e}^{j\frac{2\pi}{3}}$):

$$\underline{x}_{\alpha,\beta} = \frac{2}{3}\left(x_u + \underline{a}x_v + \underline{a}^2 x_w\right) \tag{2.2}$$

Ein weiterer wichtiger Zusammenhang ist die Umrechnung in das rotorfeste Koordinatensystem (Gleichung 2.3), was einer Weiterdrehung der statorfesten Größen um einen Transformationswinkel γ entspricht und auch als *Park-Transformation* bekannt ist.

$$\begin{bmatrix} x_d \\ x_q \end{bmatrix} = \begin{bmatrix} \cos(\gamma) & -\sin(\gamma) \\ \sin(\gamma) & \cos(\gamma) \end{bmatrix} \begin{bmatrix} x_\alpha \\ x_\beta \end{bmatrix} \tag{2.3}$$

oder

$$\underline{x}_{d,q} = \underline{x}_{\alpha,\beta} \cdot e^{-j\gamma} \tag{2.4}$$

Für umfassende Beschreibungen, unter anderem über die Herleitung der Transformationsgleichungen und der inversen Abbildungen wird auf KOVÁCS/RÁCZ (1959) verwiesen. Anmerkung: Die Gleichungen 2.1 und 2.3 wurden der Vollständigkeit halber zusätzlich in Komponentendarstellung aufgeführt, da sie später im Modell der Regelung wieder in Form einer Blockdarstellung verwendet werden.

2.2.2. Normierung verwendeter Größen auf die Maschinendaten

Um die Ergebnisse transparent aufzubereiten, nimmt man eine Normierung der Maschinengrößen auf eine Referenzgröße gleicher Dimension nach dem Schema $x = \frac{X}{X_{Bezug}}$ vor. Prinzipiell sind die Bezugsfaktoren beliebig wählbar, doch es empfiehlt sich, bestehende Konventionen einzuhalten und die Bezugsfaktoren von bekannten Maschinendaten abzuleiten:

- Nennspannung: $U_N = 30\text{V}$
- Nennstrom: $I_N = 20\text{A}$
- Nenndrehzahl: $n_N = 600\text{min}^{-1}$
- Polpaarzahl: $2p = 20$
- Statorwiderstand: $R_s = 170\text{m}\Omega$
- Statorinduktivität: $L_s = 479\mu\text{H}$

Es ergeben sich hieraus die folgenden Bezugsgrößen:

- Bezugsspannung: $U_{Bezug} = \sqrt{2} \cdot U_{N,Str} = 42.426\text{V}$
- Bezugsstrom: $I_{Bezug} = \sqrt{2} \cdot I_{N,Str} = 28.284\text{A}$
- Bezugsdrehzahl: $\Omega_{Bezug} = 2\pi \cdot p \cdot \frac{n_N}{60} = 1.257 \cdot 10^3 \frac{\text{rad}}{\text{s}}$
- Bezugsimpedanz: $Z_{Bezug} = \frac{U_{Bezug}}{I_{Bezug}} = 1.500\Omega$
- Bezugsinduktivität: $L_{s,Bezug} = \frac{U_{Bezug}}{I_{Bezug}} \cdot \Omega_{Bezug} = 1.200 \cdot 10^{-3}\text{H}$
- Bezugsmoment: $M_{Bezug} = \frac{3 \cdot U_{N,Str} \cdot I_{N,Str}}{\Omega_{Bezug}} = 1.432\text{Nm};$

Bei Realisierung der Regelung auf einem Festkomma-Rechner, wie es in dieser Arbeit der Fall ist, bringt die Normierung der Zahlendarstellung einen großen Vorteil mit sich: Die normierten Zahlen umfassen einen engen Zahlenbereich, welcher es ermöglicht, die verfügbaren Bits so aufzuteilen, dass ausreichend Stellen für eine hohe Präzision übrig bleiben. Genaueres dazu folgt in Abschnitt 4.3 bei der Ausführung über Fixpunktdatentypen.

2.2.3. Differentialgleichungen der PSM

Elektrotechnisch gesehen lässt sich die PSM einfach durch mathematische Formeln beschreiben, da es im Rotor keine Wicklungen gibt. Somit fallen, verglichen mit der ASM, die Rotorspannungs- und Rotorflussverkettungsgleichungen weg.

Statorspannungsgleichung

$$\underline{u}_s(\tau) = \underline{i}_s \cdot r_s + \frac{d\underline{\psi}_s}{d\tau} + j \cdot \omega_k \cdot \underline{\psi}_s \tag{2.5}$$

Die Statorspannungsgleichung beschreibt den Zusammenhang von Spannungsabfall am Statorwiderstand, sowie der durch die Rotation induzierten Spannung. Der Zusatzterm mit rührt daher, dass die Gleichung aus Sicht eines allgemeinen mit der Winkelgeschwindigkeit ω_k umlaufenden Koordinatensystems formuliert ist.

Statorflussverkettungsgleichung

$$\underline{\psi}_s(\tau) = \underline{i}_s \cdot l_s + \underline{\psi}_M \tag{2.6}$$

Der magnetische Statorfluss ergibt sich als Summe des Flusses, welcher durch den Statorstrom über die Statorinduktivität aufgebaut wird und dem der Dauermagneten des Rotors.

Drehmomentgleichung

$$m_r(\tau) = -\Im(\underline{i}_s^* \cdot \underline{\psi}_s) = i_{sq} \cdot |\psi_M| \tag{2.7}$$

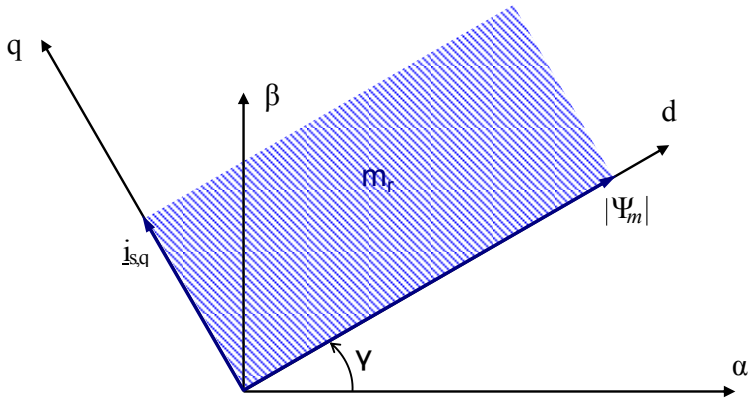

Abb. 2.3: Veranschaulichung zur Bildung des Drehmoments der PSM

Das auf den Rotor wirkende innere Moment m_r der Maschine wird durch die Statorflussverkettung und dem Statorstrom in exakt gleicher Weise wie bei der ASM hervorgerufen[6]. In Schrödl (1998) wird gezeigt, dass für das Drehmoment jedoch nur die q-Komponente des Statorstroms und der Betrag der Flussverkettung der Dauermagneten verantwortlich ist.

[6] Vgl. Schrödl (1998): Elektrische Antriebe und Maschinen, S.226f

Grafisch ist dies in Abbildung 2.3 veranschaulicht: das Drehmoment m_r entspricht hier der blau schraffierten Fläche.

Mechanische Gleichung

$$\tau_m \cdot \frac{d\omega_m}{d\tau} = m_r + m_L \tag{2.8}$$

Die mechanische Gleichung, der erste Drallsatz der Physik, sei hier zur Vollständigkeit des Gleichungssatzes angeführt, findet aber im weiteren Verlauf der Arbeit keine Verwendung.

2.3. Hochdynamischer Regelbetrieb durch Rotorflussorientierung

Die PSM ist aufgrund ihrer schlechten Dämpfung nur sehr schlecht spannungsgesteuert zu betreiben[7]. Aufgrunddessen ist ein hochdynamischer Betrieb nur geregelt möglich. Es wird angestrebt, einen Stromraumzeiger einzuprägen, der zu jedem Zeitpunkt rechtwinklig gegenüber dem Magnetflussraumzeiger orientiert ist. Dieses Konzept wird als *feldorientierte Regelung* bezeichnet. Wie im Kapitel 2.2.3 erläutert und in Abbildung 2.3 verdeutlicht, ergibt sich bei einem 90°-Winkel die optimale Drehmomentausbeute. Der gewünschte Stromraumzeiger wird mittels der Umrichterhardware durch Anlegen entsprechender Steuerspannungen eingestellt. Genaueres hierzu wird im Kapitel 3.1.1 ausgeführt.
Unerlässlich für eine reibungslose Funktionsweise der Stromregelung im rotorfesten Koordinatensystem ist neben einer genauen Messung der aktuellen Phasenströme zusätzlich das Vorliegen der Winkelinformation des Rotors bezüglich des Stators, um den Strom räumlich gesehen richtig einstellen zu können. Diese Winkelinformation kann auf meherer Arten gewonnen werden. Die rechentechnisch einfachste und transparenteste Lösung ist die Verwendung eines Lagegebers, allerdings kann der elektrische Transformationswinkel γ_{el} auch mittels sensorloser Verfahren gewonnen werden. Die wichtigsten Zusammenhänge der feldorientierten Regelung der PSM können in Schrödl (2000) oder Mohan (1998) nachgeschlagen werden. Es sei hier lediglich die Blockstruktur der Regelung aufgeführt (Abbildung 2.4), da diese eine gute Vorstellung der Aufgabe der beteiligten Funktionsblöcke liefert.

[7] Vgl. Schrödl (1998): Elektrische Antriebe und Maschinen, S.228

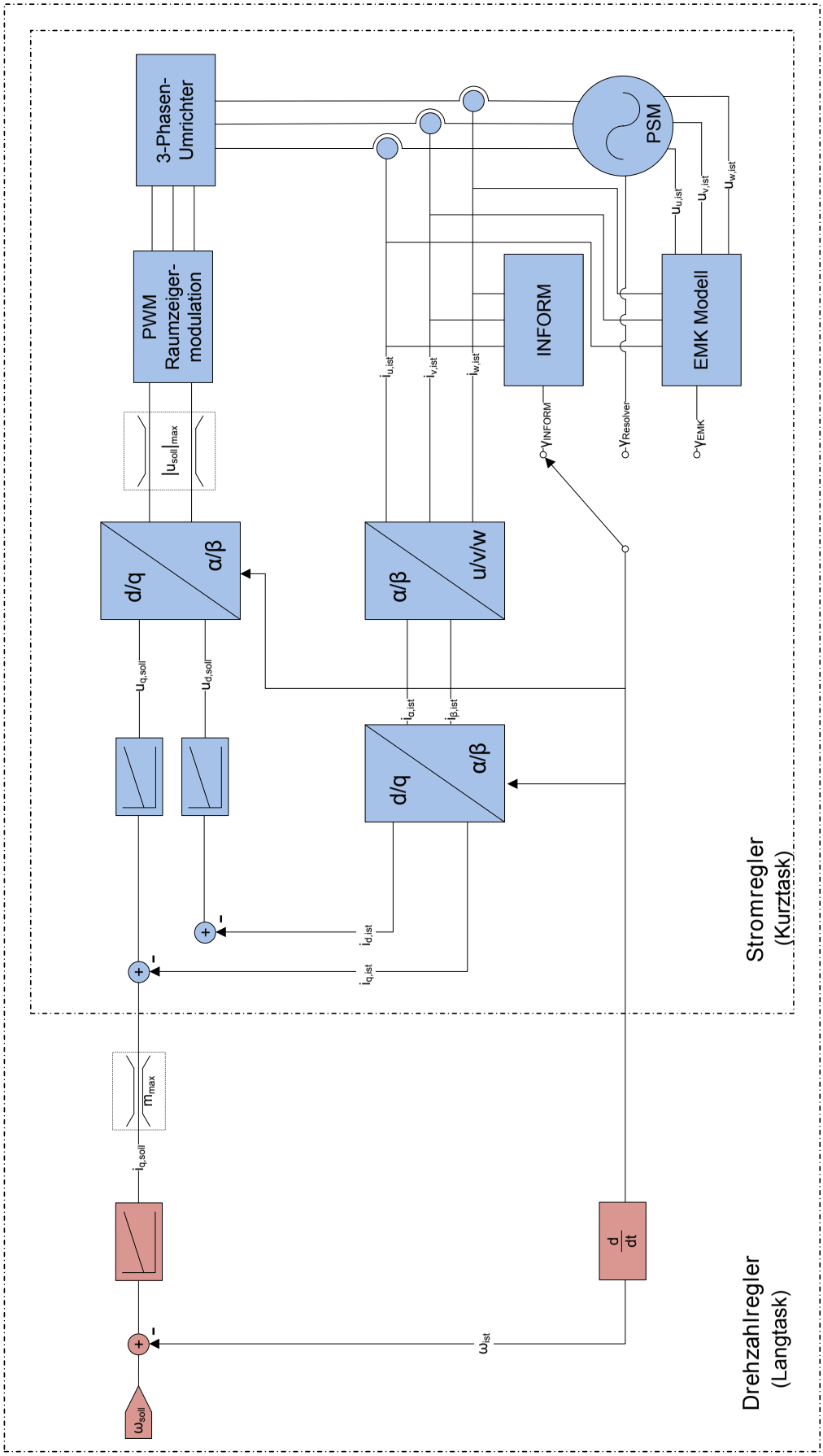

Abb. 2.4: Blockschaltbild der feldorientierten Regelung

3. Verwendete Mittel

3.1. Hardware

Bevor auf die konkrete Realisierung der Regelstruktur eingegangen wird, folgt in Kapitel 3.1 eine Zusammenstellung der verwendeten Hardware.

3.1.1. Umrichter

Der eingesetzte Umrichter ist eine Eigenentwicklung des Insituts, es existiert hierzu also kein offizielles Datenblatt. Die verbauten Elektronikbauteile wurden jedoch für einen maximalen Phasenstrom von 20A bei einer zulässigen Zwischenkreisspannung bis maximal 48V ausgelegt. Dies umfasst bei weitem den Nennbereich der verwendeten PSM. Bei der Inbetriebnahme sind keine besonderen Eigenschaften des Umrichters zu beachten, außer dass die Highside-Spannungsversorgung durch Anlegen eines Rechtecksignals einer Frequenz von 100kHz an die Eingänge PWM7, PWM9 und PWM11 aufgebaut werden muss (siehe Abbildung 3.1). Die Gate-Treiber sind nicht-invertierend, somit wird eine PWM mit AHC-Logik angelegt: Der PWMxB-Kanal ist immer eine logisch negierte Form des PWMxA-Kanals, mit Berücksichtigung der einzuhaltenden Totzeit von mindestens 1μs, um die MOSFETs nicht kurzzuschließen.

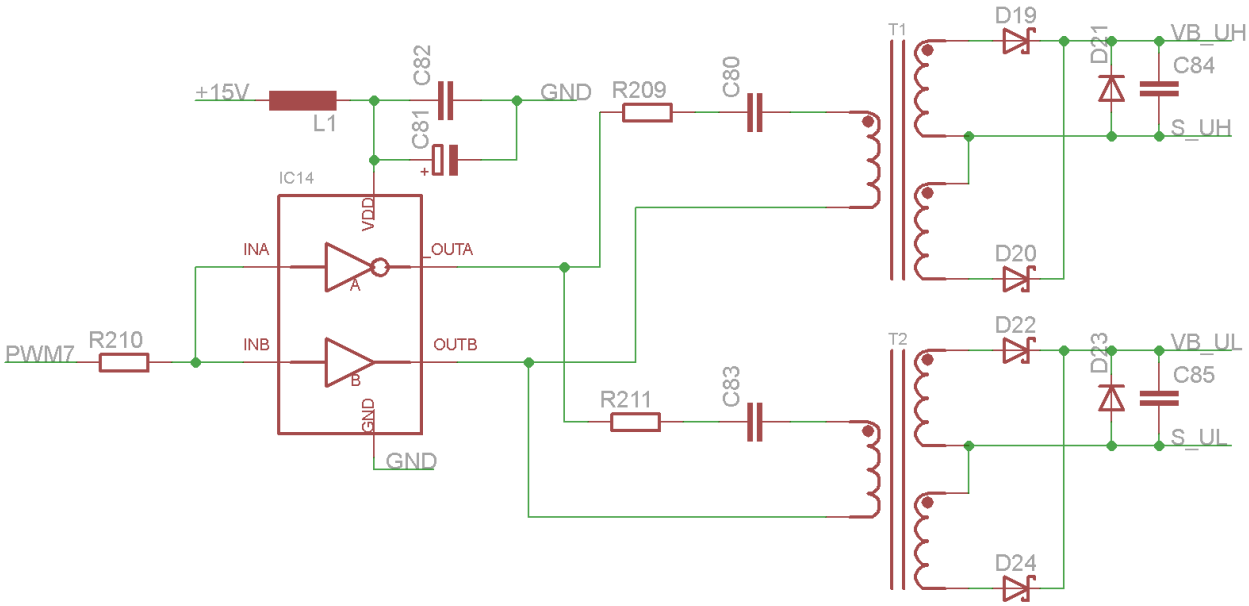

Abb. 3.1: Highside-Spannungsversorgung für Phase U, Eingang PWM7

Eine funktionierende Regelung erfordert eine genaue Strommessung. Diese kann mittels Spannungsmessung an Shunt-Widerständen oder LEMs durchgeführt werden. Für diese Arbeit wird die LEM-Messung herangezogen. Für die Bürde des Stromwandler wurde ein Wert

von 120Ω gemessen, der Anschluss des Stromwandlers ist entsprechend einer Übersetzung von 1:500 zu verschalten. Somit ergibt sich mit Verwendung der Bauteilwerte aus Abbildung 3.2 ein Skalierungsfaktor des gemessenen ADC-Rohwerts von:

$$I_{Skalierung} = \frac{2 \cdot 2.5 \cdot 500}{0.33 \cdot 120 \cdot I_{Bezug}} = 2.2320 \tag{3.1}$$

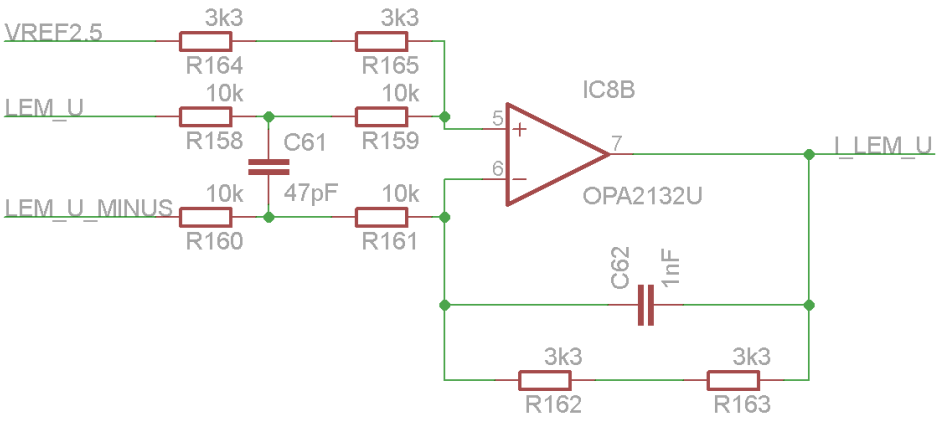

Abb. 3.2: OPV-Eingangsschaltung der LEM-Spannungsmessung für die Phase U

Für die Spannungsmessung gibt es eine ähnliche OPV-Eingangsschaltung. Der einzige Unterschied besteht in unterschiedlichen Widerstandswerten. Es folgt ein Spannungs-Skalierungsfaktor von:

$$U_{Skalierung} = ... = 4.6017 \tag{3.2}$$

3.1.2. Hauptplatine und Prozessorplatine

Die Hauptelektronik wurde ebenfalls am Institut entwickelt. Der Vorteil der institutseigenen Hardware liegt in der modularen Bauweise, die eine Erweiterung ihrer Funktionalität durch Einstecken verschiedener Einschubkarten zulässt. Ebenso kann der gewünschte Prozessor leicht den Bedürfnissen der Anwendung angepasst werden und nach dem Baukastenprinzip ausgewechselt werden. In diesem Falle wird der 32-Bit-Fixpunktrechner TMS320f2808 der Firma Texas Instruments ausgewählt. Dieser digitale Signalprozessor hat gegenüber anderen Fabrikaten den Vorteil, dass er zum einen bereits am Institut mehrmals verwendet und zum anderen von der zum Einsatz kommenden Programmiersoftware MATLAB/Simulink unterstützt wird.

Wichtigste Eigenschaften des DSP:

- Taktrate: 100MHz
- RAM Speicher: 36kB
- Flash Speicher: 128kB
- Anzahl PWM-Kanäle: 16
- Anzahl ADC: 1 x 16-Kanal 12-Bit
- Peripherie: SPI, SCI, CAN

Die Programmierung des DSP nimmt man mittels eines sogenannten JTAG-Adapters vor, der auf die Hauptplatine aufgesteckt wird und per USB-Anbindung die Datenverbindung zum PC herstellt.

3.1.3. SPI- und SCI-Kommunikationsmodule, Lagegeber

Die Entwicklungsphase des kompletten Systems erfordert unbedingt Online-Debuggingmöglichkeiten, da ansonsten die Zustände innerhalb des DSP nicht abgefragt werden können. Die dafür notwendige Hardware bilden zwei Einschubkarten für die Hauptplatine. Die SCI-Steckkarte ermöglicht die Kommunikation mittels eines am Institut entwickelten seriellen Kommunikationsprotokolls über den SCI-Bus des DSP. Genaueres hierzu ist unter 3.2.2 nachzulesen. Weiters können über die Softauge-Einschubplatine die Zeitverläufe von internen Signalen und Variablen visualisiert werden, indem die BNC-Buchsen mit einem Oszilloskop verbunden werden.
Wie im Kapitel 2.3 ab Seite 16 wurde, benötigt der hochdynamische Regelkreis die aktuelle Winkelstellung des Rotors, um den Statorstromraumzeiger richtig einzuprägen. Eine von der Anbindung an den DSP her komfortable Lösung stellt der Resolver-Chip AS5040 der Firma „Austrian Microsystems“ dar: eine frei SPI-Empfangsleitung reicht aus, um die Messdaten des

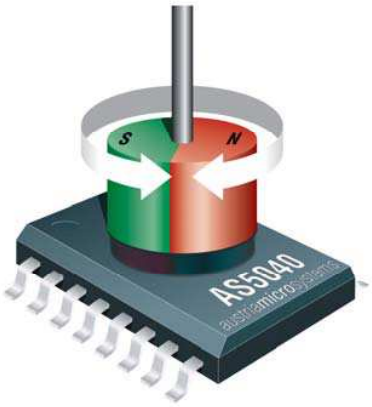

Abb. 3.3: Funktionsprinzip des Winkelresolvers
Quelle:
Datenblatt AS5040 Rev.2.10, S.1, Austrian Microsystems

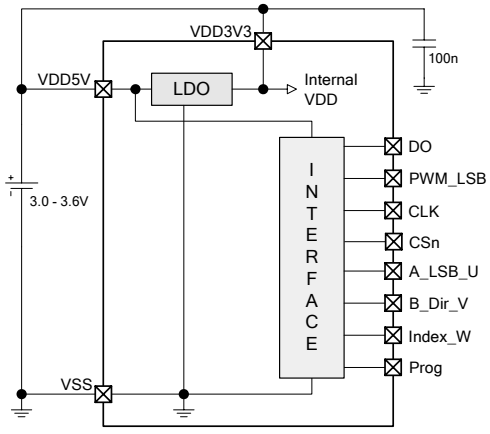

Abb. 3.4: Beschaltung für 3.3V Betriebsmodus
Quelle:
Datenblatt AS5040 Rev.2.10, S.22, Austrian Microsystems

Chips einzulesen. Die Beschaltung ist nach 3.4 vorzunehmen. Auf das Ende der Motorwelle wird eine diametral magnetisierte Magnetscheibe möglichst zentral aufgeklebt und der Chip so nah wie möglich - jedoch berührungsfrei - vor dem Magnet starr angeordnet. Je zentraler der Magnet aufgeklebt ist und je näher er am IC zu liegen kommt (siehe Abbildung 3.3), desto störunempfindlicher ist in Folge das Messsignal.

3.2. Software

Um eine erfolgreiche Realisierung der Aufgabenstellung zu ermöglichen, ist es vor allem bei dieser Arbeit von besonderem Augenmerk, Softwarepakete zu verwenden, die genau aufeinander abgestimmt sind. Die folgenden Kapitel beschreiben knapp die eingesetzten Programme und eventuell zu beachtende Besonderheiten.

3.2.1. MATLAB/Simulink, Real-Time-Workshop

Die komplette Modellbildung der PSM-Regelung erfolgt in MATLAB/Simulink 2010b unter Windows XP. Es werden hierzu für den verwendeten DSP spezielle Erweiterungen benötigt, welche von Texas Instruments bereitgestellt werden. Zu beachten ist, dass bei der Installation die 32-Bit-Version von MATLAB/Simulink auszuwählen ist, da in der 64-Bit-Ausgabe die benötigten Toolboxen nicht zur Verfügung stehen.

> Bei der Installation ist unbedingt die 32-Bit-Version von MATLAB/Simulink auszuwählen, da in der 64-Bit-Ausgabe die benötigten Toolboxen nicht zur Verfügung stehen.

Folgende Programmteile müssen installiert werden, um die volle Funktionalität der Codegenerierung auszuschöpfen:

- Simulink: Grafische Programmierung und Modellbildung
- Embedded IDE Link: Texas Instruments Schnittstelle für MATLAB
- Target Support Package
- Fixed-Point Toolbox & Simulink Fixed Point: Fixpunktberechnungen in MATLAB und Simulink
- Real-Time Workshop & Real-Time Workshop Embedded Coder: Umsetzen des grafischen Modells in C-Quellcode

Bei bereits installierten MATLAB-Versionen können die Programmkomponenten mit dem Befehl

```
>> ver
```

aufgelistet und verifiziert werden. Das Plugin Embedded IDE Link stellt die Verbindung von MATLAB/Simulink zur Texas Instruments-Entwicklungsumgebung Code Composer Studio her. Nach erfolgter Codegenerierung ist ein neu angelegtes Projekt in der Entwicklungsumgebung vorhanden.

3.2.2. Serielles Kommunikationsprogramm „COM“

Bereits in Kapitel 3.1.3 wurde erwähnt, dass für die Entwicklung des Projekts eine Debugging-Umgebung unbedingt erforderlich ist. Sofern alle hardwaretechnischen Voraussetzungen vorhanden sind, kann das serielle Kommunikationsprotokoll des Instituts in leicht abgeänderter Form in das Simulink-Modell integriert werden. Das Frontend auf dem PC bildet hierzu das Schnittstellenprogramm COM. Es ist unbedingt darauf zu achten, jene Version zu verwenden, welche speziell auf die in Simulink verwendeten Fixpunktdatentypen angepasst ist (siehe Abbildung 3.5). Einzelheiten der Änderungen können unter Kapitel 4.3 (Verwendung von Fixpunktdatentypen) nachgelesen werden.

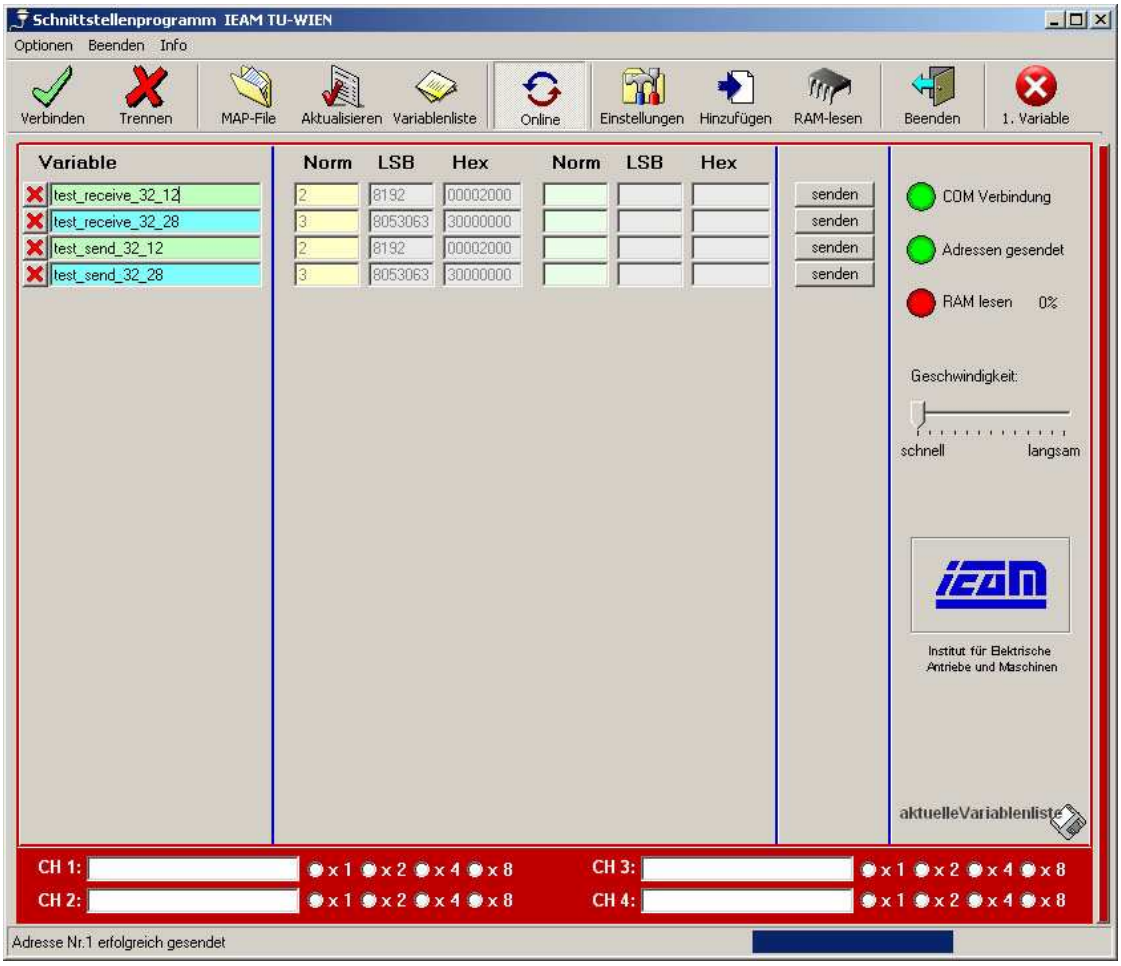

Abb. 3.5: „COM“ für Simulink-Fixpunktdarstellung; Hauptfenster

Es ist unbedingt darauf zu achten, das COM-Programm in jener Version zu verwenden, welche speziell auf die in Simulink verwendeten Fixpunktdatentypen angepasst ist.

Um die Kompabilität des COM-Programms zur verwendeten Software zu bewahren, ist das leicht abgeänderte Kommunikationsprotokoll ebenso in den Quellcode einzubinden. Für weitere Projekte reicht es allerdings aus, den im Zuge dieser Arbeit erstellten Simulink-Block für die serielle Kommunikation einzubinden.

3.2.3. Texas Instruments Code Composer Studio

Die Kompilierung des Quellcodes und das Übertragen des Programms auf den Prozessor erfolgt durch die Entwicklungsumgebung Texas Instruments Code Composer Studio in der Version 3.3 (CCS 3.3) (siehe Abbildung 3.6). Die zum Zeitpunkt des Erscheines der Arbeit aktuelle Version 4.0 der Software kann nicht verwendet werden, da hierfür noch keine Unterstützung seitens von Mathworks geboten wird.

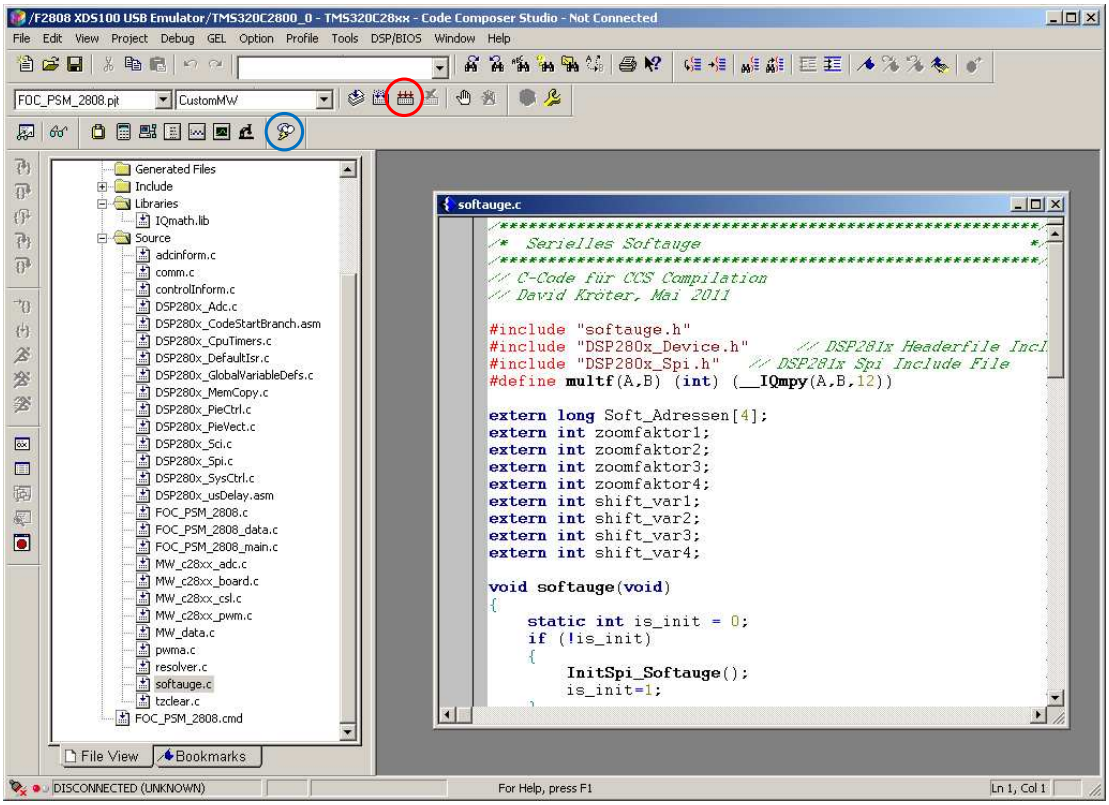

Abb. 3.6: CCS Hauptfenster; Kompilierung des akutellen Projekts starten (rote Umrahmung), Flashvorgang ausführen (blaue Umrahmung)

Die Software „Code Composer Studio“ in der Version 4.0 kann momentan nicht zur Codegenerierung verwendet werden, da hierfür noch keine Unterstützung seitens von „Mathworks“ geboten wird.

Alle wichtigen Einstellung können auch unter „MATLAB/Simulink“ vorgenommen werden. Es ist auch sinnvoll, diesen Weg zu wählen, da jeder Codegenerierungsvorgang ein neues Projekt in CCS erstellt und nicht etwa das alte aktualisiert. Somit gehen eventuell vorgenommene Einstellungen verloren.

4. C-Codegenerierung mittels MATLAB/Simulink

Bevor mit der konkreten Aufgabenstellung begonnen wird, müssen prinzipielle Zusammenhänge und Sachverhalte der Codegenerierung mittels „MATLAB/Simulink“ erläutert werden.

4.1. Der Vorgang der Codegenerierung

Der folgende Abschnitt ist eine Zusammenfassung des Vorgangs der Codegenerierung unter „MATLAB/Simulink“ und beruht vor allem auf den Quellen MathWorks (2004) und MathWorks (2010). In Abbildung 4.1 wird der Vorgang zusätzlich grafisch veranschaulicht. Die Stellen, an denen der Benutzer aktiv in den Prozess eingreifen kann, sind durch das Benutzer-Symbol gekennzeichnet. Ausgangsbasis des kompletten Vorgangs ist die Simulink-Modelldatei im *mdl*-Format. Initialwerte übernimmt die Modelldatei aus einem *m*-Skript.

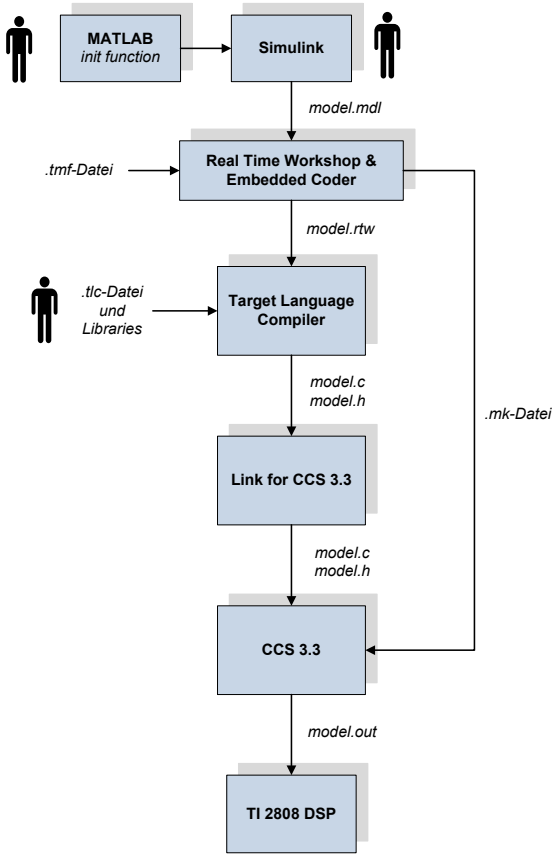

Abb. 4.1: Schritte der Codegenerierung

Nun kommt der Embedded Coder ins Spiel und übersetzt das Modell in ein Metamodell, repräsentiert durch die Datei *model.rtw*: hierin sind alle Beziehungen der verwendeten Blöcke, Signale und Variablen in ASCII-Format angegeben, das Modell wird hier von einer grafischen in eine textuelle Ebene kompiliert. Zugleich wird ein vorgefertigtes Template-Makefile mit der Endung *tmf* an die spezifischen Bedürfnisse des zu kompilierenden Modells

angepasst und daraus ein C-Makefile erstellt. Im nächsten Schritt verarbeitet der Target Language Compiler das textuelle Metamodell und benutzereigenen C-Quellcode mit Hilfe plattformspezifischer Hilfsdateien (Dateiendung *tlc*) und erstellt die fertigen Quellcode- und Headerdateien in einem Ordner „./_ticcs“. Weitere Informationen zur Verzeichnisstruktur und der Bedeutung der einzelnen Quellcodedateien sind in MathWorks (2004) ab S.2-22 nachzuschlagen. Das Plugin „Link für CCS“ übernimmt diese Dateien und legt daraus ein neues Projekt in der TI-Entwicklungsumgebung an. In der Entwicklungsumgebung wird das Programm nun durch den C-Compiler in ein auf dem Signalprozessor lauffähiges Programm übersetzt und mittels des JTAG-Adapters auf den DSP geladen. Den zentralen Schlüssel zur Generierung eines lauffähigen Codes auf einem „Texas Instruments“ DSP stellen die *tlc*-Dateien dar, da sie der Wegweiser in Richtung der eingestetzten Hardware sind. Geübte und erfahrene Programmierer haben zwar die Möglichkeit, die Vorlagedateien zu editieren, jedoch wird aufgrund deren Komplexität davon abgeraten und empfohlen, auf den voreingestellten Vorlagen zu verbleiben.

4.2. Einbindung vorhandener C-Quellcode-Dateien in Simulink

Ein essentieller Punkt in der Gestaltung von flexiblem und benutzerspezifischem Quellcode ist das Einbinden vorhandener C-Dateien in das Simulink-Modell durch sogenannte *S-Functions*. Die Sinnhaftigkeit dieser Aufgabe kann mit zwei Anwendungsfällen belegt werden:

1. Eingliedern von Gerätetreibern oder Interfaces[8]: Am Institut wurden bereits grundlegende Funktionalitäten in C-Code realisiert. Genannt sei hier das für dieses Projekt relevante serielle Kommunikationsprotokoll oder die Raumzeigermodulation. Es wäre kaum, oder wenn, dann nur mit sehr viel Aufwand möglich, diese Funktionen genauso effizient in Simulink-Blockstruktur nachzubilden. Ein Eingliedern bereits bestehender Module ist viel effizienter und zeitsparender.

2. Bewahren der Flexibilität von handgeschriebenem C-Code: Im Laufe der Arbeit werden Stellen deutlich, an denen die Flexibilität der Codegenerierung an ihre Grenzen stößt, dies wird vor allem in Kapitel 4.6.1 deutlich: Einmal über die Dialogfenster konfigurierte Blöcke können zur Laufzeit des Programms nicht mehr umkonfiguriert werden. Ist dies aber dennoch verlangt, so muss dies über C-Code passieren, welcher genau auf die anzupassenden Register des DSP zugreift.

Durch das Einbinden von Code mittels S-Functions kann eine große Stärke von Simulink ausgespielt werden: Teile des Modells können durch Simulation getestet und verifiziert werden, was direkt auf der Zielplattform nicht ohne Weiteres möglich ist. Hierzu folgt später unter Kapitel 5.2 ein Beispiel.
Obwohl S-Functions eine flexible Lösung zur Integration komplexer Algorithmen sind, verursacht die darauf basierende Programmiererschnittstelle (API) einen Mehraufwand in Re-

[8] Vgl. MathWorks (2011a): Developing S-Functions, S.1-6

chenzeit und Speicherauslastung[9], welcher in Embedded-Systemen wegen des Anspruchs der Echtzeitfähigkeit so gut es geht durch Optimierung vermieden werden muss. Die hierfür relevante Technologie stellt der Target Language Compiler dar, der mit Hilfe vorliegender *tlc*-Dateien optimierten Quellcode erzeugt und die involvierten Funktionen als *inlined* definiert.

> Inlining has the additional effect of eliminating the procedure-call overhead as well. The point of making a function inline is to hint to the compiler that it is worth making some form of extra effort to call the function faster than it would otherwise - generally by substituting the code of the function into its caller. As well as eliminating the need for a call and return sequence, it might allow the compiler to perform certain optimizations between the bodies of both functions[10].

Es besteht also oberste Priorität, die S-Functions so zu gestalten, damit sie später durch den Compiler optimiert eingebunden werden können. Allerdings darf der Begriff *inlined* nicht mit dem gleichnamigen C-Schlüsselwort verwechselt werden, für den die obige zitierte Aussage gilt. Im Kontext des Simulink Coders zur Codegenerierung haben die beiden Begriffe sinngemäß zwar den Hintergrund der Optimierung, die dahinterstehende Technologie ist jedoch eine unterschiedliche.
Grundsätzlich wird bei der Erstellung von S-Funktionen die Strategie verfolgt, den einzubindenden Algorithmus auch zur Simulationslaufzeit ausführen zu können. Der Lösungsalgorithmus greift hierzu auf die kompilierte Version des C-Quellcodes im *mex*-Format zurück. Die Erstellung einer *mex*-Datei zur Simulation mit Simulink verlangt zusätzlich zur C-Datei eine komplexe Struktur aus Parametern, welche das Verhalten zur Laufzeit definieren. Auf Teile dieser Struktur wird in jedem Simulationsschritt zurückgegriffen. Da die Inhalte der Struktur aber für die Codegenerierung irrelevant sind, müssen für eine effiziente Implementierung zwei Ausgaben des gewünschten Algorithmus erstellt werden: Eine für die Simulation (*mex*-Datei) und eine für die Codegenerierung (*tlc*-Datei). Der hierfür nötige Aufwand kann bereits bei kleinen Funktionen beträchtlich sein. Aus diesem Grund wird mit dem Simulink Coder das Werkzeug Legacy Code Tool mitgeliefert.

[9] Vgl. MathWorks (2010): RTW Embedded Coder, S.32-3
[10] Siehe Aho et al. (2007): Compilers, S.914
Sinngemäße Übersetzung in das Deutsche: Inlining hat den Zweck, zusätzlichen Rechenaufwand von Funktionsaufrufen zu eliminieren. Der zentrale Punkt, eine Funktion als inlined zu deklarieren, besteht darin, dem Compiler mitzuteilen, dass dieser im Vorhinein zusätzlichen Aufwand beim Funktionsaufruf betreiben soll. Normalerweise werden hierdurch dann die Codefragmente direkt in die aufrufende Funktion eingebettet. Sowohl wie die Beseitigung von Aufruf-, als auch von Rückgabesequenzen erlaubt dem Compiler, weitere Optimierungen der Funktionen auszuführen

Das Legacy Code Tool

Das Legacy Code Tool ermöglicht auf einfache Weise die Erstellung effizienter S-Funktionen zur Codegenerierung durch die automatisierte Durchführung der folgenden Schritte[11]:

1. Datenstruktur initialisieren: `>> def = legacy_code('initialize')`
2. S-Funktions-Quellcode erzeugen: `>> legacy_code('sfcn_cmex_generate', def)`
3. S-Funktions-Quellcode kompilieren: `>> legacy_code('generate_for_sim', def)`
4. Maskierten Simulink-Block erstellen: `>> legacy_code('slblock_generate', def)`
5. *tlc*-Datei generieren: `>> legacy_code('sfcn_tlc_generate', def)`

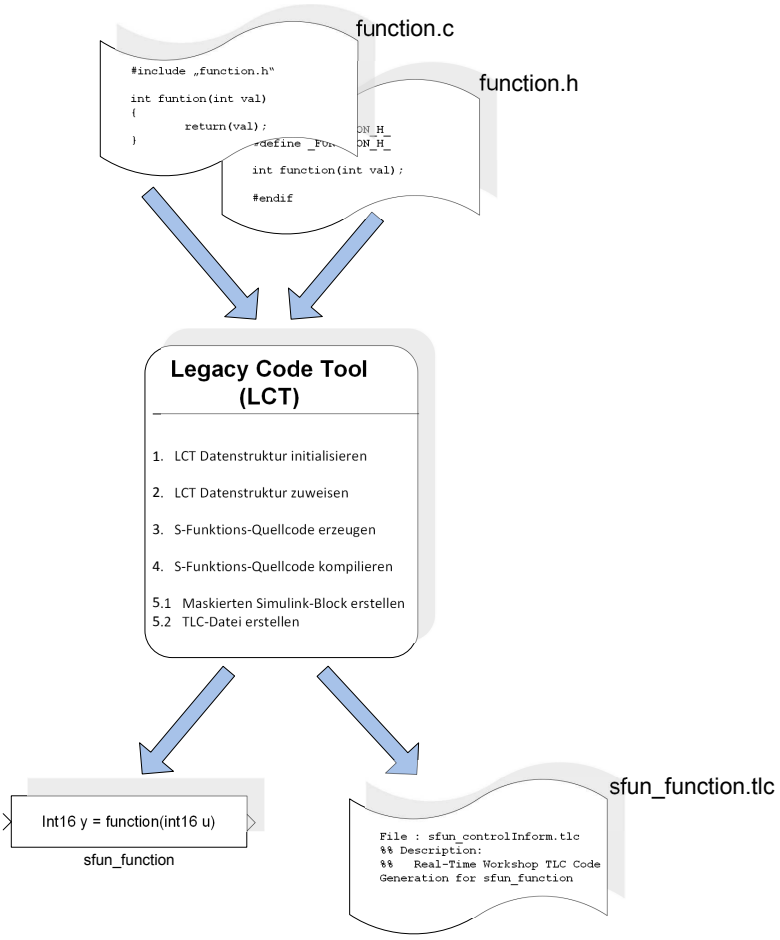

Abb. 4.2: Funktionsweise des Legacy Code Tool

[11] Vgl. MathWorks (2011a): Developing S-Functions, S.4-55ff

Punkt 3 obiger Aufzählung funktioniert nur, sofern ein gültiger C-MEX Compiler installiert ist. Dies ist durch Eingabe des Befehls

```
>> mex -setup
```

in die MATLAB Kommandozeile nachzuprüfen.

Im Zuge dieser Arbeit wurde das Legacy Code Tool mehrmals zur Integration von handgeschriebenem C-Code verwendet. Deshalb wurden hierfür Skriptdatein erstellt, welche pro Funktion nur noch ein Minimum an Eingaben benötigen. Diese MATLAB-Skripten sind im Anhang A.1 dieser Arbeit gelistet.

4.3. Verwendung von Fixpunktdatentypen

Kostengünstige DSP wie der eingesetzte TMS320f2808 verlangen meist die Berechnung in Fixpunktdatentypen. Die verfügbaren Bits werden in diesem sogenannten *IQ*-Format benutzerdefiniert auf Ganzzahlwert und Nachkommastellen aufgeteilt. Da die Werte in normierter Zahlendarstellung in der Regel um ± 1 liegen, werden für den Ganzzahlwert nur wenige Bits verwendet. Am Institut hat sich das *IQ4.12*-Format - auch *HANSL*-Format - durchgesetzt, welches dem Ganzzahlwert vier Bits zur Verfügung stellt und die restlichen Stellen als Kommastellen interpretiert. Somit ergibt sich ein Zahlenbereich von

$$-\frac{2^{15}}{2^{12}} \ldots 0 \ldots \frac{2^{15}-1}{2^{12}} = -8 \ldots 0 \ldots 7.9998$$

mit einer Auflösung von $\frac{1}{2^{12}} = \frac{1}{4096}$ über den kompletten Wertebereich.
Wenige bestimmte Rechenoperationen verlangen nach einem größeren Zahlenbereich: hierfür werden zwei 16-Bit Register zu einer Long-Variable zusammengefasst und dann 20 Bits für den Ganzzahlwert reserviert (siehe Abbildung 4.3).

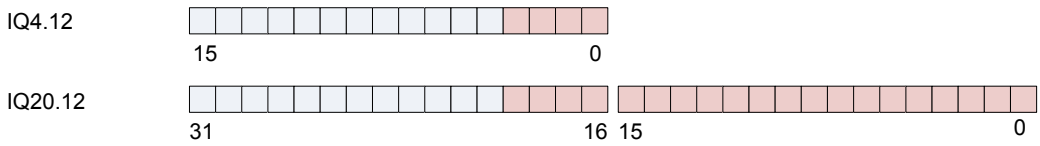

Abb. 4.3: Zahlendarstellung im *HANSL-Format*; Ganzzahlbits rot gefärbt

Diese Zahlendarstellung kann allerdings bei der Verwendung von Simulink nicht gänzlich beibehalten werden. Ein wesentlicher Bestandteil der Arbeit liegt daran, effizienten Code zu generieren. Dies bedingt die Verwendung eigens von Texas Instruments entwickelter Bibliotheken für die Fixpunktrechnung, welche in der Toolbox Target Support Package als Sammlung von Simulink-Funktionsblöcken integriert sind. Allerdings bestehen diese Blöcke auf der Verwendung von 32-Bit-Variablen, was die Ausweitung des *IQ4.12*-Formats auf

das *IQ4.28*-Format nötig macht (siehe Abbildung 4.4). [Prinzipiell hätte auch ein beliebiges anderes Format gewählt werden können, doch dies hätte die bisherigen Konventionen des Instituts verletzt.] Die Verwendung von 32-Bit Variablen führt bei diesem Prozessor zwar zu keiner Reduktion der Rechengeschwindigkeit, da der TMS320f2808 mit einer 32-Bit-ALU ausgestattet ist, jedoch steigt der Speicherverbrauch mit dem Faktor zwei. Im Zuge dieser Arbeit ist die Speicherauslastung aber nicht der limitierende Faktor. Weiters wird angemerkt, dass sich die Erhöhung der zur Verfügung gestellten Nachkommastellen nicht in einer Verbesserung der Rechengenauigkeit des Regelalgorithmus äußert, da das ungenaueste beteiligte Glied, der ADC, mit einer Auflösung von 12 Bit arbeitet.

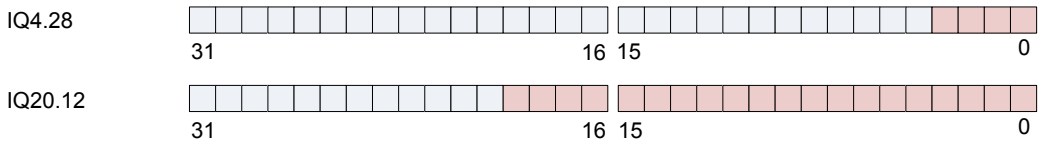

Abb. 4.4: Zahlendarstellung im *Simulink-Format*; Ganzzahlbits rot gefärbt

Im Zusammenhang mit der Umstellung des Zahlenformats tauchen weitere Probleme auf: Das Kommunikationsprotokoll und die zugehörige COM-Schnittstelle sind bisher auf das *IQ4.12*-Format ausgelegt. Andere Zahlenformate können mit der bisherigen Konfiguration nicht korrekt interpretiert werden. Also muss das Kommunikationsprotokoll auf der DSP-, sowie der PC-Seite angepasst werden.
Bisherige Ausführungen über das Zahlenformat hängen direkt mit der C-Programmstruktur zusammen. Um den erwünschten Code zu erhalten, muss bei der Modellbildung in Simulink beachtet werden, immer den richtigen Datentyp eines Signals oder einer Variablen auszuwählen. Dies lässt sich direkt im Dialogfenster des jeweiligen Blocks konfigurieren, wie in Abbildung 4.5 ersichtlich ist. Vergisst der Programmierer diesen Punkt, so wird dies mit hoher Wahrscheinlichkeit in einem nicht funktionstüchtigen Programm resultieren, da MATLAB in diesem Falle nach einem vorgegebenem Muster eine rein semantisch korrekte Möglichkeit wählt.

> Ein Grundlegender Punkt bei der Modellbildung in Simulink ist die Definition von Fixpunktdatentypen in den Dialogfenstern des jeweiligen Blocks. Wird dies vergessen, so resultiert mit hoher Wahrscheinlichkeit ein nicht funktionstüchtiger Code.

Das Eingabeformat lehnt sich hierbei an das *IQ*-Format an: Der Bezeichner *fixdt(a,b,c)* beschreibt den Fixpunktdatentyp, welcher durch die Parameter a, b und c bestimmt wird.

a: Vorzeichen: Gibt an, ob der Datentyp vorzeichenbehaftet ist (1 für ja, 0 für nein).

b: Länge: Bezeichnet die Länge n des Datentyps in Bit

c: Nachkommastellen: Steht für die Anzahl m der Nachkommastellen. Somit ergibt sich für die Anzahl der Ganzzahlstellen: $i = n - m$

Für ein Signal im *IQ4.28*-Format ist somit *fixdt(1,32,28)* einzugeben, für das *IQ20.12*-Format wählt man *fixdt(1,32,12)*.

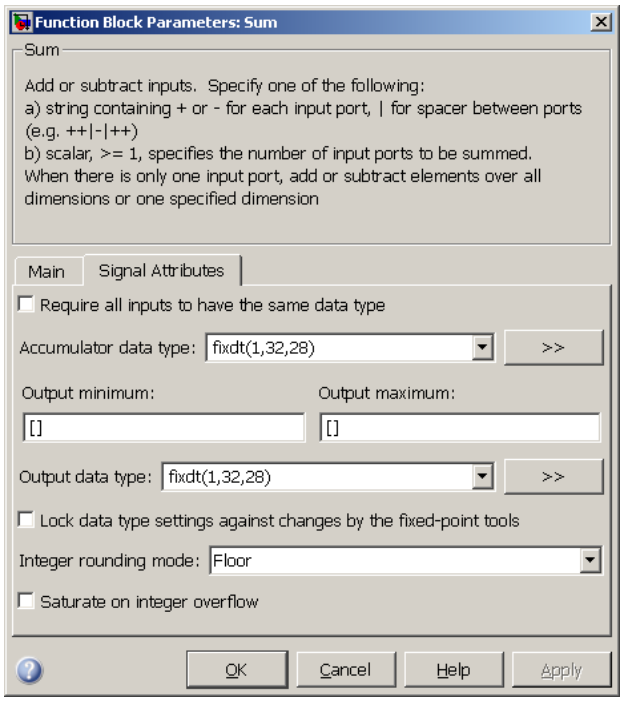

Abb. 4.5: Einstellung des Fixpunktdatenformats unter Simulink

4.4. Einflussnahme auf den generierten Code

Ein Vorteil der Verwendung von Codegeneratoren ist, dass der Fokus des Programmierers nicht vorwiegend auf den C-Code gerichtet sein muss. Gänzlich aus dem Auge lassen kann er diesen trotzdem nicht, da die Leistungsfähigkeit der Regelung durch ihn bestimmt ist. Werden keinerlei Optimierungen vorgenommen, so stellt man schnell fest, das man den Prozessor zu sehr auslastet und dieser nicht die gewünschten Ergebnisse liefert. Neben *Inlining* (siehe 4.2) und Verwendung von Fixpunktdatentypen bieten sich noch weitere Möglichkeiten der Einflussnahme auf den generierten Code an: im Hauptfenster des geöffneten Modells kann durch Betätigen der Tastenkombination *Ctrl+E* das Dialogfenster von Abbildung 4.6 hierzu geöffnet werden.
Die Einstellungsoptionen des Menüpunktes „Optimization“ sind bereits mit den richtigen Voreinstellungen belegt. Hier muss bis auf den Punkt „Inline Parameters“ nichts abgeändert werden. Die Option „Inlined Parameters“ bestimmt, wie die Parameter von Funktionsblöcken im Code erscheinen: ist das Kontrollfeld aktiviert, werden sämtliche Variablen als Konstanten

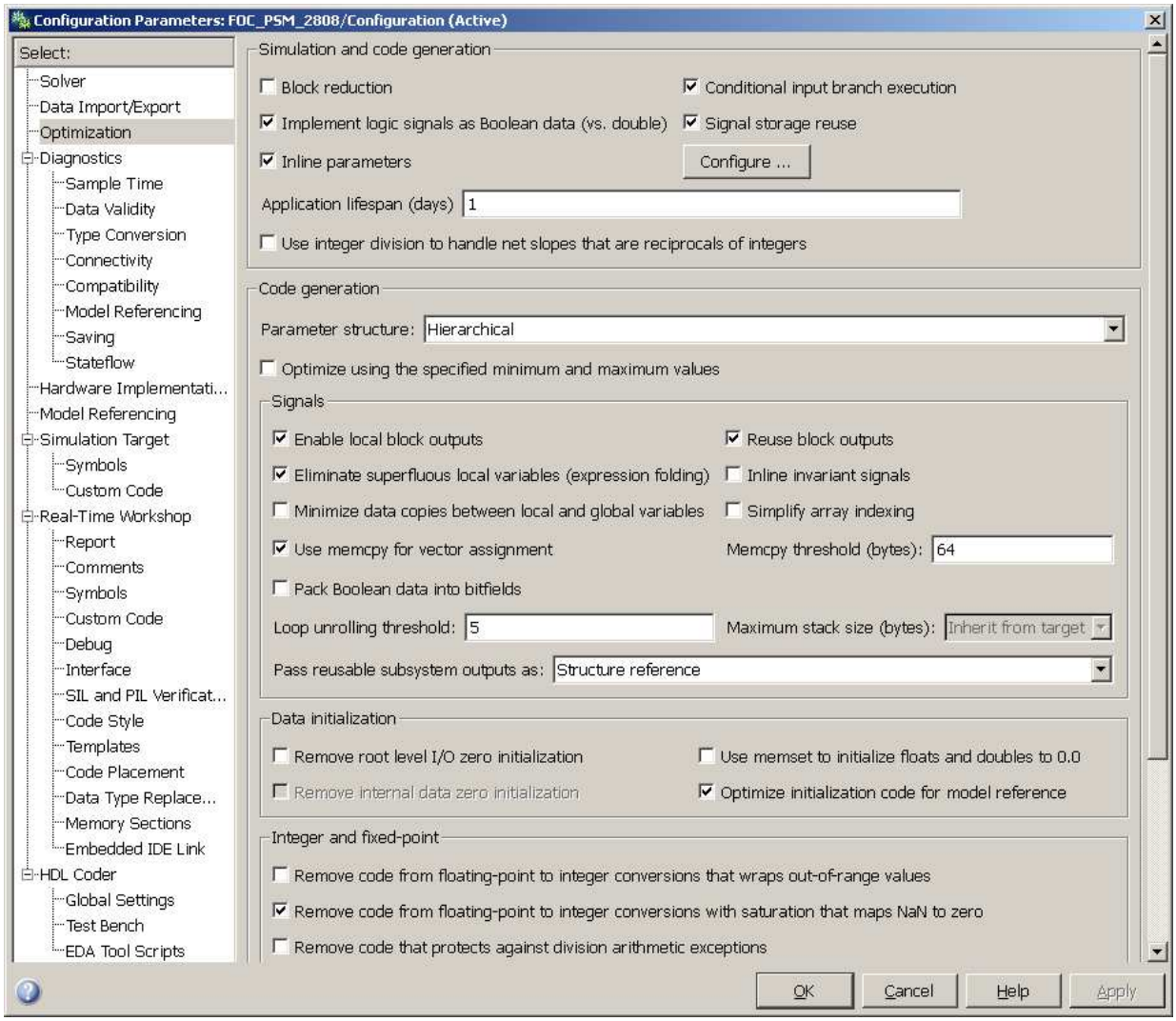

Abb. 4.6: Hauptkonfigurationsfenster des Simulink-Modells

deklariert, außer jene, die man per Hand exkludiert (hierzu mehr unter 4.5). Dies reduziert den Speicherverbrauch und erhöht die Rechengeschwindigkeit des DSP[12]. Weitere relevante Optionen finden sich unter dem Menüpunkt „Real Time Workshop“. Dessen Unterpunkt lassen selbstklärend folgende Justierungen am Erscheinungsbild des generierten Codes zu:

- Kommentare im Quellcode je nach Benennung der Simulink-Blöcke und Signale
- Variablen und Funktionsnamen
- Vordefinierte Datei-, und Funktionsköpfe

Vor allem ein konsequentes Schema für die Bennenung von Simulink-Blöcken und Signalen hilft später beim eventuellen Debuggen des Quellcodes, da sich diese Bezeichnungen im Code exakt wiederspiegeln.

[12]Vgl. MathWorks (2011b): Simulink User's guide, S.19-16

Die weitaus wichtigste Einstellung betrifft zwar nicht direkt die Erscheinung des Codebildes, wird aber aufgrund deren Nähe zu den anderen Optionen im Konfigurationsfenster hier behandelt. Der Unterpunkt Embedded IDE Link gestattet es dem Benutzer, der Texas Instruments Entwicklungsumgebung weitere Argumente für den Compiler und Linker mitzugeben. Hier sind folgende Befehle unerlässlich für eine optimierte Kompilierung und folglich eine hohe Ausführungsgeschwindigkeit des Programms auf dem DSP.

- Compiler Options String: *-o3 -fr*
- Linker Options String: *-c -m*

  Werden die Compiler- und Linkerbefehle nicht gesetzt, laufen bereits kleine Programme auf dem DSP nicht schnell genug ab.

4.5. „Tunable Parameters“, Ausgabe von Signalverläufen

Tunable Parameters stellen Typen von Variablen dar, deren Wert während der Laufzeit der Regelung geändert werden kann. Die Integration dieses Konzept ist unerlässlich, da nur so gewünschte Sollwerte von Variablen über die serielle Schnittstelle vom Benutzer eingestellt werden können. Bereits im vorherigen Abschnitt 4.4 wurde auf *Inlined Parameters* und deren Ausnahmen hingewiesen. Exakt diese Ausnahmen stellen die einstellbaren Parameter dar und werden im Konfigurationsfenster (siehe Abbildung 4.6) über die Schaltfläche „Configure“ definiert.
Auf der linken Seite des Fensters sind die Variablen des aktuellen MATLAB Workspace aufgereiht, rechts die bereits ausgewählten Variablen. Jene Variablen, die rechts als „Exported Global“ gekennzeichnet sind, erscheinen im C-Code als globale Variablen und erhalten so einen Eintrag in der *map*-Datei des CCS-Projekts, welche die Adressen der Variablen im Speicher des DSP kennzeichnet. Dies ermgöglicht den Zugriff auf die Variablen durch das COM-Programm.
Wird die Option „SimulinkGlobal (Auto)“ gewählt, so erscheint die Variable nicht als global deklariert. Es wird allerdings sichergestellt, dass der Variablenname beibehalten wird, was vor allem für Konstanten nützlich ist, um die Lesbarkeit des Code zu gewährleisten.
Es ist zu beachten, dass nur wenige Blöcke, darunter der meist verwendete „Konstanten“-Block, eine Änderung ihrer Parameter zur Laufzeit zulassen. Wird ein einstellbarer Parameter in einem nicht unterstützten Funktionsblock verwendet, so ist die Codegenerierung nicht möglich und wird durch einen Fehler abgebrochen.

Um die gewünschten Ergebnisse zu erreichen, wird daran erinnert, bei Funktionsblöcken wie etwa dem „Konstanten-Block“ das Fixpunktdatentypformat so zu konfigurieren wie es in Kapitel 4.3 erläutert wurde.

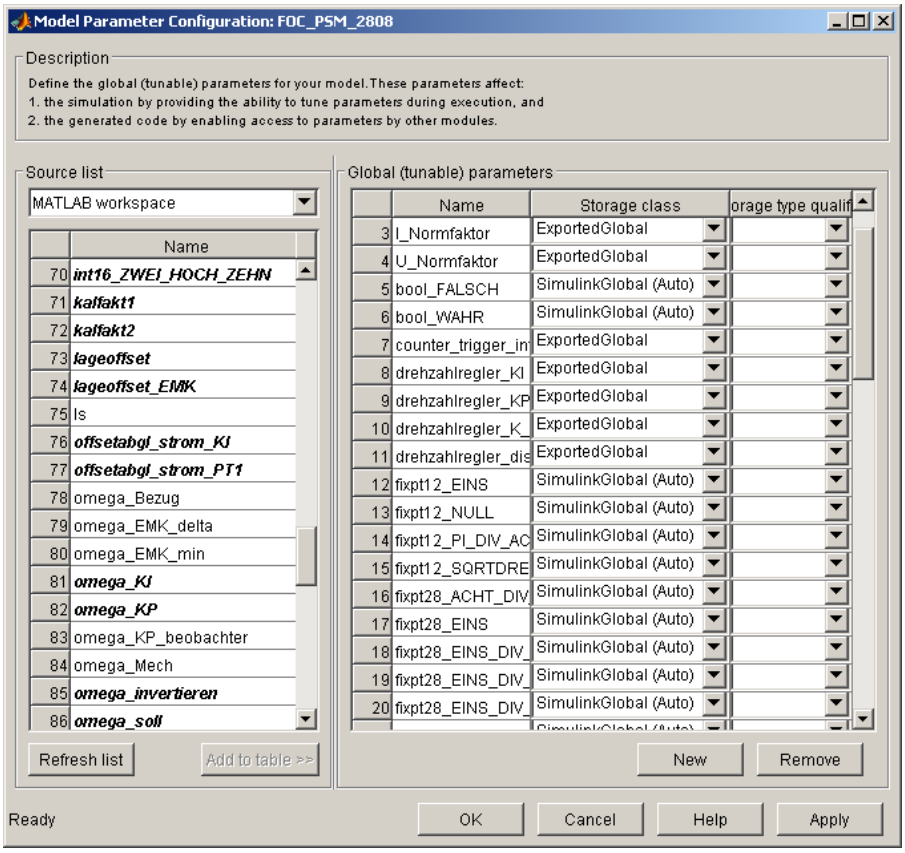

Abb. 4.7: Konfigurationsfenster der *Tunable Parameters*

Signalverläufe sind eng mit den einstellbaren Parametern verwandt. Der Unterschied zwischen den beiden Typen besteht darin, dass Signale nur zur Ausgabe vorgesehen und vom Benutzer in ihren Werten nicht direkt änderbar sind. Soll eine Variable an das Softauge oder über die COM-Schnittstelle ausgegeben werden, ist dies durch Benennung eines Signalastes nach einem Doppelklick auf selbigen einfach möglich. Anschließend ist darauf zu achten, im Kontextmenü, welches mit einem rechts-Mausklick auf das Signal geöffnet wird, die Option „Exported Global“ im Reiter „Real Time Workshop“ auszuwählen. Wählt man „Simulink-Global (Auto)“, so betrifft dies nur die Benennung des Signals im C-Code, ein Eintrag in der *map*-Datei wird nicht vorgenommen.

4.6. Essentielle Blöcke

Aus den vielen verfügbaren Simulink-Funktionsblöcken wird im folgenden Kapitel eine Auswahl präsentiert, die gerade für die feldorientierte Regelung einer PSM von besonderer Bedeutung sind.

4.6.1. „Target Preferences“

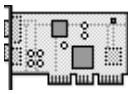

Abb. 4.8: „Target-Preferences“ Simulink-Block

Der Funktionsblock „Target Preferences“ bestimmt die Eigenschaften des verwendeten DSP, dessen Hardware- und Softwareschnittstellen sowie die Speichereinteilung. Wichtig ist, dass dieser Block überhaupt vorhanden ist, da sonst der Real Time Workshop über keine Information der eingesetzten Hardware verfügt. Einstellungen müssen allerdings kaum vorgenommen werden, da die meisten Felder bereits mit gültigen Voreinstellungen belegt sind. Einzig die DSP-Taktfrequenz ist mit Hinsicht auf die eingestellten Baudraten der Kommunikationsschnittstelle oder des Winkelresolvers zu verifizieren, da diese sonst nicht funktionieren. Weiters würde das Timing des Interruptschemas aufgrund fehlerhaft konfigurierter CPU-Timer nicht wie gewollt ablaufen. Die Einstellungsmöglichkeiten für die CPU-Taktfrequenz finden sich im Dialogfenster des Blocks sowohl unter dem Reiter „Board“ als auch unter „Peripherials“.

> Unbedingt sind im Dialogfenster des Blocks „Target Preferences“ die korrekten Einstellungen für die CPU-Taktfrequenz zu verifizieren, da ansonsten zeitkritische Vorgänge nicht wie gewollt ablaufen.

Weiters hat der Programmierer die Möglichkeit, im Dialogfenster den Ordnerpfad für die bei der Codegenerierung eingebundenen Quellcode- und Headerdateien zu spezifizieren. Werden keine Änderungen vorgenommen, so werden die Dateien aus dem Verzeichnis der Simulink Toolbox herangezogen, welche mit den von Texas Instruments zur Vefügung gestellten Dateien übereinstimmen. Falls keine außergewöhnlichen Anforderungen gestellt werden, besteht keine Notwendigkeit, den Ordnerpfad benutzerspezifisch anzupassen.

4.6.2. „Analog Digital Converter“

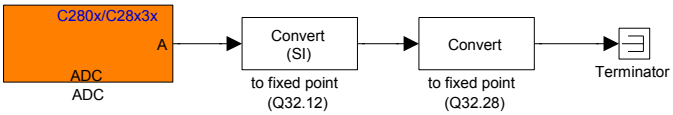

Abb. 4.9: „ADC“ Simulink-Block

Das für den geschlossenen Regelkreis unverzichtbare Messglied stellt in diesem Falle der 12-Bit „Analog Digital Converter“ dar. In Simulink wird dieser durch einen einzigen Block repräsentiert, welcher umfassend konfiguriert werden kann. Sämtliche Einstellungsoptionen sind selbstklärend. Zu achten ist in dieser Arbeit nur darauf, die Kontrolbox „Post interrupt at end of conversion“ zu aktivieren. Dies bewirkt in weiterer Folge, dass der Kurztask nach einem Konversionszyklus aufgerufen wird. Wird dieser Punkt übersehen, so funktioniert die komplette Regelstruktur nicht.

> Es ist darauf zu achten, die Kontrolbox „Post interrupt at end of conversion“ zu aktivieren. Wird dieser Punkt übersehen, so funktioniert die komplette Regelstruktur nicht.

Da die Daten des ADC-Blocks nicht im Fixpunkt-Format ausgegeben werden können, sondern höchstens als 16- oder 32-Bit Integer, muss dem Block eine Datentypenkonversion mittels zweier „Conversion“-Blöcke ins gewünschte Fixpunktformat folgen. Der „Conversion“-Block ist in der Simulink-Bibliothek in die Sammlung der meistgenutzten Bausteine aufgenommen.
Ebenfalls ist zu erwähnen, dass der Algorithmus des ADC-Blocks die ADCRESULTx-Register des DSP abruft und sofort um vier Bits nach rechts schiebt, um wirklich das LSB an die erste Stelle zu setzen. Bei der Programmierung in C erfolgt dies, falls gewünscht, per Hand. Dies ergibt für den Rückgabebereich des ADC in Simulink Werte von 0 bis 4095, was bei der Skalierung der Messdaten berücksichtigt werden muss.

4.6.3. „Pulse Width Modulation“

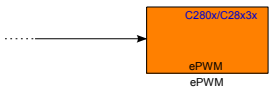

Abb. 4.10: „ePWM“ Simulink-Block

Die Ansteuerung des Umrichters wird, wie in Abbildung 2.4 ersichtlich, mittels PWM-Signalen vorgenommen. Die PWM-Hardware wird mit Simulink durch die PWM Funktionsblöcke angesprochen. Die umfangreichen Einstellungsmöglichkeiten sichern eine flexible Handhabung, führen aber auch leicht zu zeitraubenden Fehlern. Deshalb wird der Konfiguration dieses Blocks besonderes Augenmerk geschenkt. Exemplarisch sind die relevanten

Dialogfenster eines der drei PWM-Module in den Abbildungen 4.11 bis 4.16 abgebildet. Irrelevante Reiter, wie z.B. „PWM Chopper Control“, bei denen nichts geändert werden muss, sind nicht dargestellt.

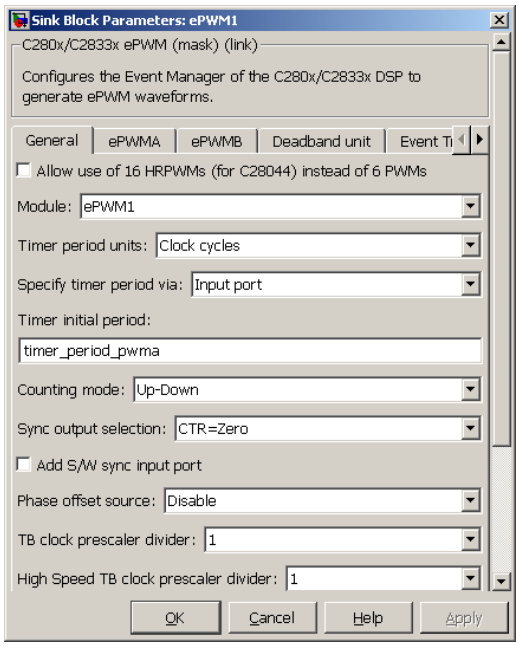

Abb. 4.11: PWM-Dialogfenster „General“

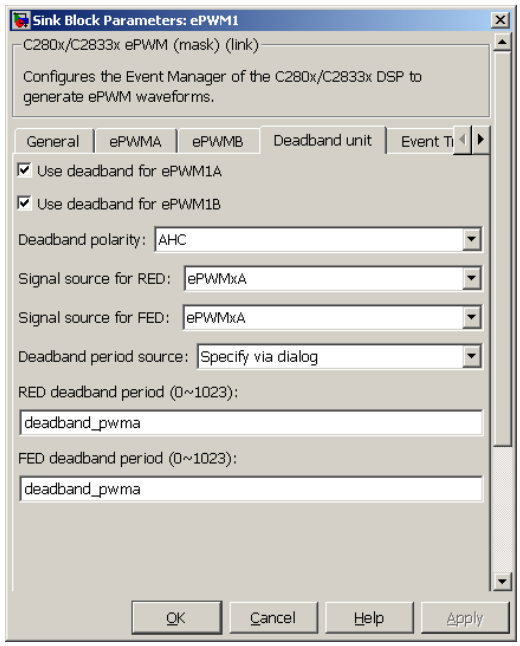

Abb. 4.12: PWM-Dialogfenster „Deadband Unit“

Im Grunde werden alle drei PWM-Kanäle identisch konfiguriert, die wenigen Ausnahmen seien in folgender Aufzählung genannt:

- Da die Kanäle PWM2 und PWM3 auf das erste Modul synchronisiert werden, muss unter dem Reiter „General“ die Option „Sync Output Selection“ auf „EPWMxSYNCI or SWFSYNC“ gesetzt werden. Zusätzlich wird „Phase Offset Source“ auf „Specify via Dialog“ eingestellt und in das sich öffnende Textfeld „Phase Offset Value“ der Wert „0“ eingetragen.
- Der Interrupt „ADC start of conversion“ darf nur einmal ausgelöst werden. Deshalb werden die Kontrollboxen „Enable ADC start of module A“ im Reiter „Event Trigger“ bei PWM2 und PWM3 deaktiviert.
- Auch der „Trip Zone Event“ kann für PWM2 und PWM3 deaktiviert werden, da es aufgrund der Hardwareausführung des Umrichters ausreicht, wenn ein Kanal ein TripZone-Ereignis sendet.

Angemerkt sei kurz, dass die Option „Second Event“ im Dialogfenster unter dem Reiter „Event Trigger“ bewirkt, dass der PWM-Zähler nur bei jedem zweiten Erreichen des Wertes

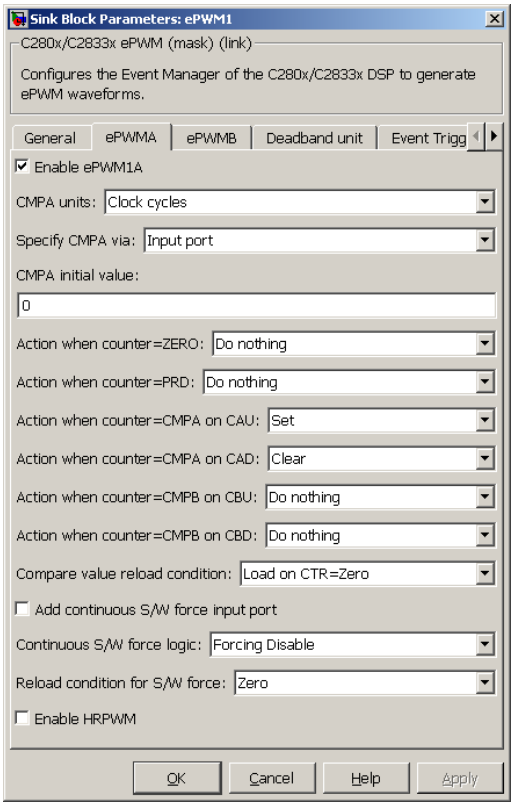

Abb. 4.13: PWM-Dialogfenster „PWMA“

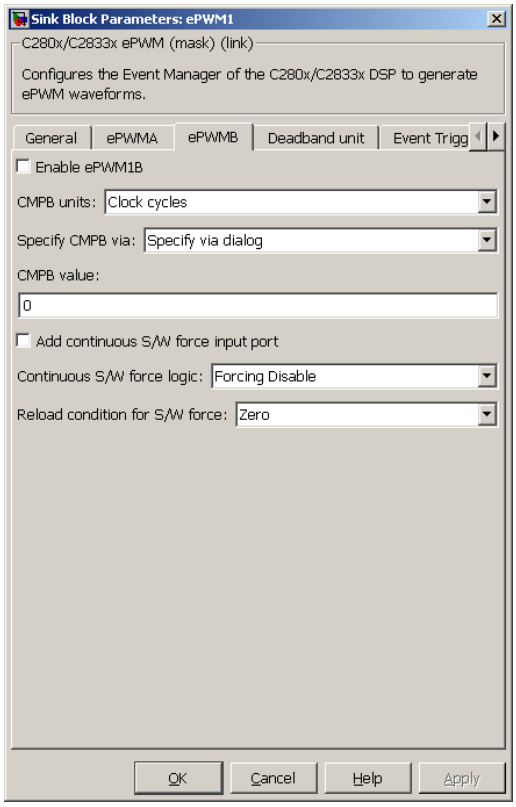

Abb. 4.14: PWM-Dialogfenster „PWMB“

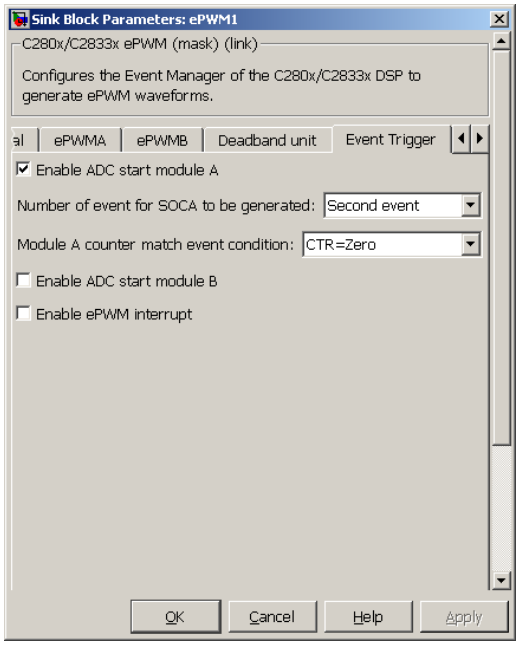

Abb. 4.15: PWM-Dialogfenster „Event Trigger“

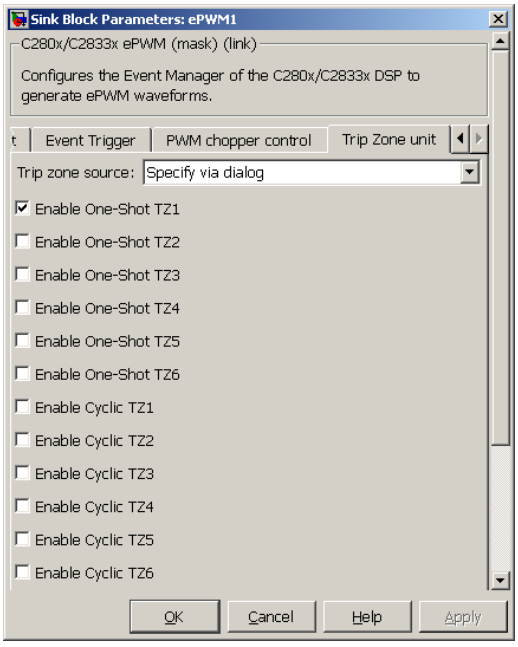

Abb. 4.16: PWM-Dialogfenster „Trip Zone unit“

„0“ einen Interupt auslöst. Diese Realisierung wurde hier bewusst gewählt, da sonst mit der PWM-Zählerperiode von 50μs eine PWM-Frequenz von 10 kHz folgen würde[13], was bereits in einem hörbaren Bereich liegt. Somit wurde die Periode auf 25μs verkürzt, die Frequenz folglich auf 20 kHz gehoben und nur jede zweite Periode gewertet. Die Dauer der beiden Tasks bleibt dadurch unangetastet.
In den meisten Arbeiten wird das Zusammenspiel von ADC und PWM und deren Interruptauslösung als bekannt vorausgesetzt. Ist dies beim Leser jedoch nicht der Fall, ist es ihm unmöglich, den Ablauf der Regelung bis ins kleinste Detail nachzuvollziehen. Deshalb sei die zeitliche Abfolge Schritt für Schritt mit Hilfe von Abbildung 4.17 genau erklärt.

- Der PWM1-Timer zählt bei der symmetrischen PWM schrittweise von „0“ bis zu einem benutzerdefinierten Wert (hier: *timer_period_pwma*) und von dort wieder zurück. Die Dauer eines Zählschrittes kann über die Taktfrequenz der CPU und zugehörige Multiplikatoren errechnet werden. Näheres hierzu kann in TexasInstruments (2004) nachgeschlagen werden.
- Sind zwei Zählzyklen abgelaufen, hat der PWM1-Zähler also zum zweiten Mal den Wert „0“ erreicht, wird ein ADC-Konversionszyklus initiert.
- Nach der Messphase folgt ein ADC-Interrupt, welcher den Kurztask startet. Anmerkung: Die Messphase ist hier zur Verdeutlichung unverhältnismäßig lang eingezeichnet, in Wirklichkeit läuft der Messvorgang jedoch im Nanosekunden-Bereich ab.

Ein analoger Vorgang läuft bei PWM6 ab: Hier jedoch wird nach jedem Erreichen des Wertes „0“ ein Interrupt direkt durch das PWM-Modul ausgelöst und der lange Task unmittelbar aufgerufen.

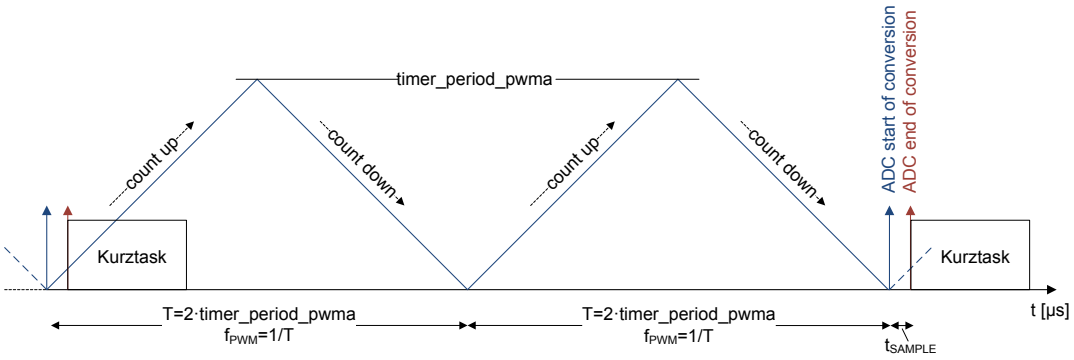

Abb. 4.17: Zusammenhang zwischen ADC, PWM-Zähler und Kurztask

Die eigentliche Aufgabe der PWM besteht jedoch im Schalten der beiden digitalen Ausgänge A und B=$\overline{A}$. Der Schaltzeitpunkt wird durch den PWM-Zähler in Kombination mit der benutzerdefinierten Schwelle (*CMPA*) bestimmt: Erreicht der Zähler diese Schwelle beim

[13] Erklärung der Zusammenhänge der PWM folgt im nächsten Absatz

Aufwärtszählen, so wird der Ausgang A auf 3.3V (logisch „ein“) gesetzt, Ausgang B hat dann denn Wert 0V (logisch „aus“). Beim erneuten Passieren der Schwelle, jedoch diesmal im Abwärtszählen, tauschen die Ausgänge dann ihren Zustand (siehe Abbildung 4.18). Beim Schaltvorgang selbst muss beachtet werden, dass nie beide Ausgänge (A und B) eines PWM-Kanals zugleich auf logisch „ein“ liegen, da dies sonst einen Brückenkurzschluss des betroffenen Zweiges zur Folge hätte. Diese Maßnahme wird von der Totzeit-Einheit der PWM getroffen, indem Kanal B immer unter Berücksichtigung einer gewissen Totzeit in Bezug auf Kanal A geschaltet wird, um den Kurzschluss eines Zweiges zu vermeiden.

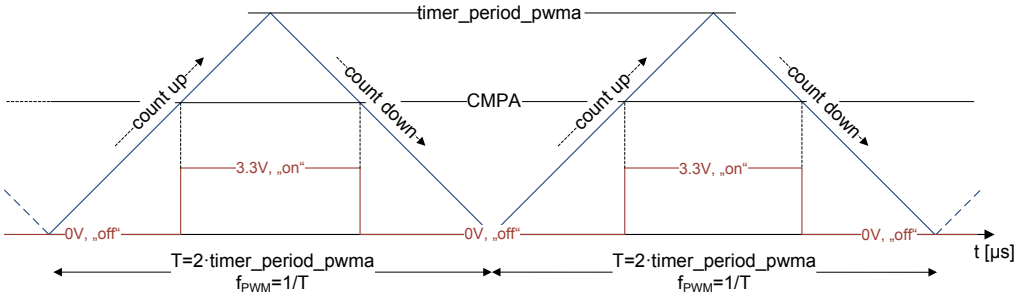

Abb. 4.18: Zusammenhang zwischen PWM-Zähler und digitalem Ausgang A

4.6.4. „Idle Task“

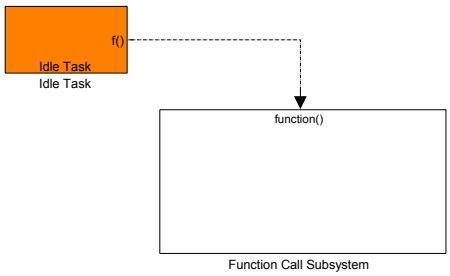

Abb. 4.19: „Idle-Task“ Simulink-Block

Der „Idle Task“ (Idle - engl.: Leerlauf) übernimmt Aufgaben, die nicht unbedingt zeitkritischen Anforderungen unterliegen, sondern je nach Auslastungszustand der CPU ausgeführt werden können: ist gerade „nichts zu tun“, übernimmt der „Idle Task“ und führt die mit ihm verknüpften Funktionen aus. Dies wird insbesondere für die Kommunikation und das Softauge in Anspruch genommen, da es in beiden Fällen nicht auf deren exakten Ausführungszeitpunkt ankommt. Weiters sind hier Funktionen ausgelagert, die auf die Hardware des Umrichters oder der Hauptplatine zugreifen, sobald der Benutzer eine entsprechende Anforderung gibt. Ein Beispiel hierfür ist die Umrichterfreigabe, die nur nach Befehl des Benutzers ausgelöst werden kann.

4.6.5. „External Interrupt“

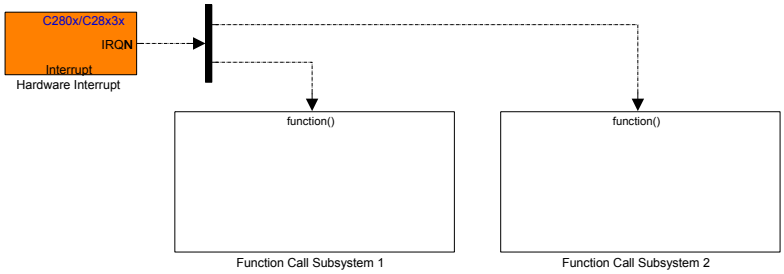

Abb. 4.20: „Interrupt“ Simulink-Block

Der „External Interrupt“ Block aus Abbildung 4.20 ist unerlässlich für die Programmablaufsteuerung. In der C-Programmierung werden die Adressen der Funktionen zur Interruptbehandlung per Hand konfiguriert, in Simulink geschieht dies grafisch durch Blockstrukturen. Ausgangspunkt ist hierzu das Konfigurationsfenster des Interruptblocks (siehe Abbildung 4.21): die vier vorhandenen Textzeilen ermöglichen die Eingabe von Interruptvektoren, welche das Verhalten des Programms bei Auftreten eines Interrupts definieren. Die Anzahl der Elemente dieser Vektoren bestimmt die Anzahl der Interrupts und muss für alle vier Textfelder dieselbe sein. Die Elemente der ersten beiden Zeilen sind der Interrupttabelle des DSP zu entnehmen, die dritte Zeile „Simulink Task Priorities“ kennzeichnet die Prioritäten der Interrupts während der Simulation. Für die Codegenerierung selbst ist diese Zeile bedeutungslos. Wichtig hingegen ist der Inhalt des letzten Textfeldes: Hier kann festgelegt werden, ob eine Funktion, welche durch einen Interrupt ausgelöst wurde, durch einen nächsten Interrupt unterbrochen werden („1“) kann oder nicht („0“).
Ist das Dialogfenster korrekt konfiguriert, kann mit der Verschaltung der Blöcke fortgefahren werden: Für jeden eingestellten Interrupt muss im Folgenden ein *funktionsgetriggertes Subsystem* eingerichtet werden, wie dies auch in Abbildung 4.20 geschehen ist. Innerhalb dieser Subsysteme werden die gewünschten Algorithmen implementiert.

Interruptschema der feldorientierten Regelung

Der eben beschriebene Block wurde in dieser Arbeit hierzu verwendet, um eine Interruptstruktur mit einem kurzen (100μs) und einem langen Task (400μs) zu verwirklichen, welche verschieden zeitkritische Aufgaben zu bewältigen haben. Das Ablaufschema der beiden Tasks ist in Abbildung 4.22 dargestellt.

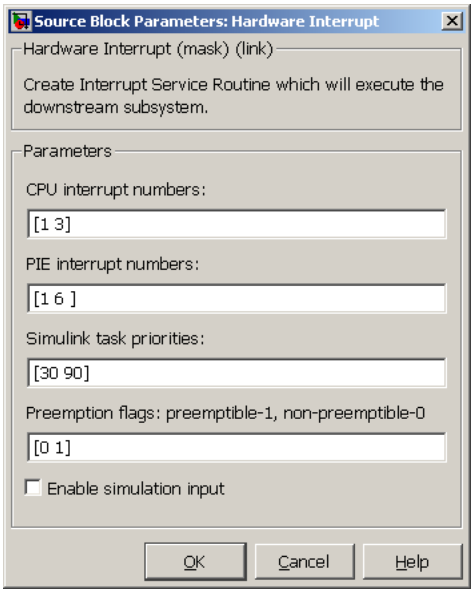

Abb. 4.21: Einstellungsmöglichkeiten des „Interrupt“ Simulink-Block

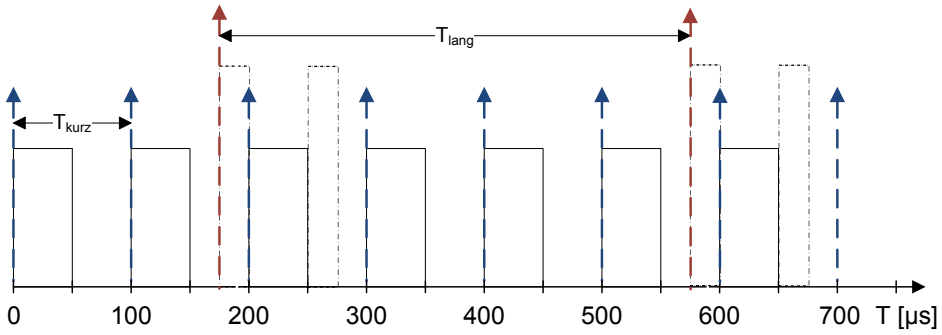

Abb. 4.22: Interruptschema der Regelung: Kurztask blau, Langtask rot

Die Auslösezeitpunkte der Interrupts sind durch Impulsvektoren mit verschiedenen Farben eingezeichnet: blau entspricht dem kurzen und rot dem langen Task. Den Impulsvektoren nachfolgende Rechteckflächen verdeutlichen die Verarbeitungsdauer der Tasks. Gestrichelte Ränder symbolisieren einen unterbrechbaren Task, dessen Berechnung nach Beendigung des höher priorisierten Tasks fortgesetzt wird. Die Aufgaben der beiden Tasks sind in Abbildung 2.4 grafisch aufbereitet.

4.7. Einschränkungen der Programmierung mittels Codegenerierung

Die Codegenerierung über MATLAB/Simulink ist ein nützliches Tool, um auch ungeübten Programmierern die Realisierung von aufwendigeren Regelstrukturen zu ermöglichen. Allerdings wird dieser Vorteil zum Preis verschiedener Einschränkungen erkauft.

4.7.1. Flexibilität der Codegenerierung

Da für die vorhandenen Hardware-Funktionsblöcke umfangreiche Dialogfenster zu deren Konfiguration exisitieren, können die meisten Anforderungen durch gezieltes Einstellen umgesetzt werden. Der Codegenerator richtet sich dann strikt nach den gewählten Vorgaben. Das bedeutet aber, dass es nicht möglich ist, mit den konventionellen Mitteln eine Änderung am Hardwareverhalten zur Laufzeit des Programms zu realisieren. Soll beispielsweise das PWM-Muster beim Auftreten eines bestimmten Ereignisses von „symmetrisch“ auf „unsymmetrisch“ umgestellt werden, so bleibt dies dem erfahrenem Programmierer vorenthalten, welcher durch Integration von S-Funktionen direkt auf die relevanten Register des DSP zugreifen kann. Dies legt eine der großen Schwachstellen der Codegenerierung über MATLAB/ Simulink offen: Die Flexibilität, welche man von der konventionellen C-Programmierung gewohnt ist, muss in gewissen Bereichen der Einfachkeit der Bedienung weichen.
Im Zuge dieser Arbeit wird diese Schwachstelle besonders beim Umsetzen des INFORM-Algorithmus augenscheinlich. Da während einer INFORM-Periode eine andere Interruptroutine aktiv ist, muss der ADC-Interrupt mit der INFORM-Routine anstatt mit dem Kurztask verknüpft werden. Im Modell sind diese beiden Interrupts jedoch zwei separaten Interrupts zugeordnet, also wird auch der Code dementsprechend ausgelegt. Die Lösung des Problems stellt eine S-Funktion dar, welche die DSP-Interrupttabellen direkt modifiziert. Genaueres folgt hierzu unter Kapitel 5.8.

4.7.2. Unterstützung der Simulink-Blockbibliothek

Während der Entwicklungsphase stellen sich immer wieder neue Probleme, für deren Lösung es scheinbar bereits vorgefertigte Blöcke aus den zahlreichen Simulink-Bibliotheken gibt. Oft ist die Zuhilfenahme dieser Blöcke nicht möglich: teils unterstützen nicht alle Bausteine die diskretisierte Datenverarbeitung, teils wird das Fixpunktdatentypformat nicht unterstützt. Weiters ist eine Kombination mit *Tunable Parametern* (siehe Kapitel 4.5) nur in den wenigsten Fällen erlaubt. So bleibt vielmals nur noch der Ausweg, die geforderte Funktionalität über Standardblöcke aufzubauen. Dies birgt weitere Fehlerquellen und erfordert zusätzlichen Zeitaufwand, welcher in der Planungsphase des Projektes schwer kalkulierbar ist.
Da diese Arbeit jedoch Pioniercharakter im Bereich der Codegenerierung für das Institut für Energiesysteme und Elektrische Antriebe hat, kann der soeben erläuterte Mehraufwand in zukünftigen Projekten durch Wiederverwendung der geschaffenen Strukturen deutlich reduziert werden. Um einen Ansatz der Fortführung dieser Arbeit vorwegzunehmen, könnte diskutiert werden, ob eine Erstellung einer eigens für das Institut entwickelten Bibliothek sinnvoll ist.

5. Vorgehensweise zur Erreichung der Zielsetzung

Nachdem in Kapitel 4 sowohl die Grundlagen der Codegenerierung als auch die wichtigsten Funktionsblöcke erklärt wurden, hat folgender Abschnitt die konkrete Implementierung des Regelungsalgorithmus zum Ziel. Bereits in der Aufgabenstellung wurde erläutert, dass diese Arbeit einen deskriptiven Charakter beinhaltet. Deshalb sind folgende Punkte strukturell einer Inbetriebnahmeanleitung nachempfunden.

Konventionen bei der Modellbildung

Um die Übersichtlichkeit des Modells zu bewahren, werden die Modellblöcke mit kennzeichnenden Farben hervorgehoben.

- Orange: Der Block bezieht sich auf die Hardware des Umrichters, des DSP oder der Hauptplatine.
- Hellblau: „Tunable Parameter"-Block, welcher ein Signal im Format *fixdt(1,32,28)* ausgibt, also einen Wertebereich von -7.9998 bis +8 umfasst.
- Dunkelgrün: „Tunable Parameter"-Block mit Ausgang eines Signals im *fixdt(1,32,12)*-Format, entsprechend dem Wertebereich einer traditionellen Long-Variable am Institut.
- Dunkelgrau: Konstanten-Block, Format je nach auszugebener Variable.

Vorbereiten einer leeren Modelldatei

Nach der Erstellung einer neuen Modelldatei *model.mdl*, muss diese noch für die Codegenerierung ausgelegt werden. Dies geschieht am einfachsten über die MATLAB-Kommandozeile durch Eingeben folgender Befehle:

```
>> load_system('model')
>> set_param('outsourced','Solver','FixedStepDiscrete')
>> set_param('model','FixedStep','dt_solver')
>> set_param('model','InitFcn','init_model')
```

Die soeben aufgeführte Befehlsreihe setzt den richtigen Algorithmus zur Lösung der Differenzengleichungen in Simulink und die zugehörige Schrittweite. Weiters wird die Skriptdatei *init_model.m* mit den Initialwerten für das Modell festgelegt.

Vorwegnahme: Übersicht über das endgültige Gesamtsystem

Um sich einen Überblick über die Eingliederung der im Folgenden beschriebenen Subsysteme und das fertige Gesamtsystem verschaffen zu können, ist letzteres in Abbildung 5.1 dargestellt. Es ist zu diesem Zeitpunkt nicht nötig, alle Details zu verstehen, vielmehr wird Wert auf die Zusammenhänge der Systeme und deren Subsysteme gelegt.

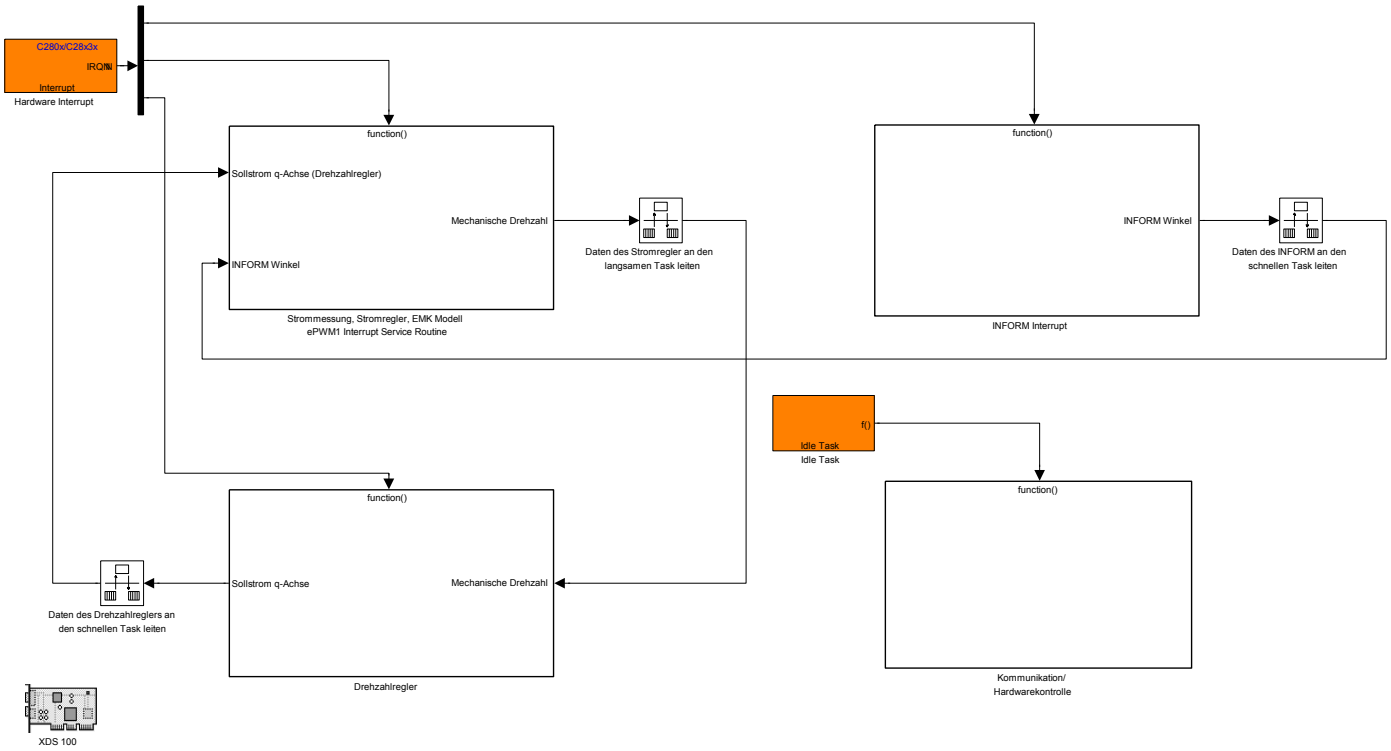

Abb. 5.1: Übersicht über das endgültige Gesamtsystem

Hauptaufgaben der vier Subsysteme:

- Stromregler-Subsystem (Kurztask): Strommessung, Messwertskalierung, Stromregler, Drehzahlbeobachter, EMK-Modell, Raumzeigermodulation und PWM-Ansteuerung.
- Drehzahlregler-Subsytem (Langtask): PI-Regler zur Führung des Drehzahlsollwerts.
- INFORM-Subsystem: Timing der INFORM-Interrupts, Strommessung und Winkelauswertung für die INFORM-Methode.
- Kommunikations-Subsystem: SCI-Kommunikation zum PC und Hardwareansteuerung.

5.1. Inbetriebnahme und Testen der eingesetzten Hardware

Bevor die ersten Schritte mit der Codegenerierung gewagt werden, wird dringend empfohlen, die verwendete Hardware auf ihre Funktion zu prüfen, um spätere Fehler ausschlie-

ßen zu können. Am einfachsten geschieht dies durch Einspielen einer bestehenden Software, welche gegebenenfalls noch angepasst werden muss. Die Zeit, welche eventuell zur Adaptierung aufgewendet werden muss, darf nicht als „verloren“ angesehen werden, da man im Falle nichtfunktionierender Hardware während der Codegenerierung schnell eingeholt wird: Dem unerfahrenen Benutzer ist es ad hoc nicht möglich, Fehlerquellen zwischen falsch in MATLAB-konfigurierter oder defekter Hardware zu unterscheiden.

> Bevor die ersten Schritte mit der Codegenerierung gewagt werden, wird dringend empfohlen, die verwendete Hardware auf dessen Funktion zu prüfen, um so spätere Fehler ausschließen zu können. [...] Dem unerfahrenen Benutzer ist es ad hoc nicht möglich, Fehlerquellen zwischen falsch in MATLAB-konfigurierter oder defekter Hardware zu unterscheiden.

5.1.1. Einbindung des Kommunikationsprotokolls

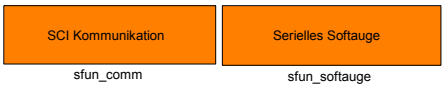

Abb. 5.2: S-Funktionsblöcke zur Kommunikation

Sinnvollerweise wird mit der Integration der Kommunikation des DSP zum Entwicklungsrechner begonnen, um bereits zu Beginn der Umsetzung des Regelalgorithmus die Möglichkeit des Zugriffs auf interne Variablen zu haben. Hierfür werden die bestehenden Algorithmen zur SCI-Kommunaktion, wie in Kapitel 3.2.2 erwähnt, an die Simulinkdatentypen angepasst und mit dem Legacy Code Tool in das Modell integriert. Die Verwendung des LCT wird über die MATLAB-Skriptdatei *create_sfun_comm.m* automatisiert aufgerufen. Die Skriptdatei, welche im Anhang A.1 gelistet ist, bringt in der Entwicklungsphase den Vorteil, den Vorgang bei Bedarf einer Anpassung wiederholt auszuführen. In weiteren Projekten, in denen die Kommunikation benötigt wird, muss dies jedoch nicht mehr vorgenommen werden: Es genügt, die vorhandenen Blöcke in das Modell aufzunehmen und die Quellcodedateien der Kommunikation in die Verzeichnisstruktur einzugliedern.
Bei weiteren Projekten ist zu beachten, dass der Quellcode für Softauge und/oder SCI-Kommunikation abgeändert werden muss, falls eine der Kenndaten nicht mit den folgenden Werten übereinstimmt:

- CPU-Taktfrequenz: 96MHz
- Baudrate: 115200 baud pro Sekunde
- High Speed Prescaler: 1
- Kommunikation über SCI-Kanal A (GPIO Pins 28 und 29)
- Softauge über SPI-Bus A (GPIO Pins 16, 17, 18 und 19)

5.1.2. Ansteuerung der Umrichterhardware und der Hauptplatine

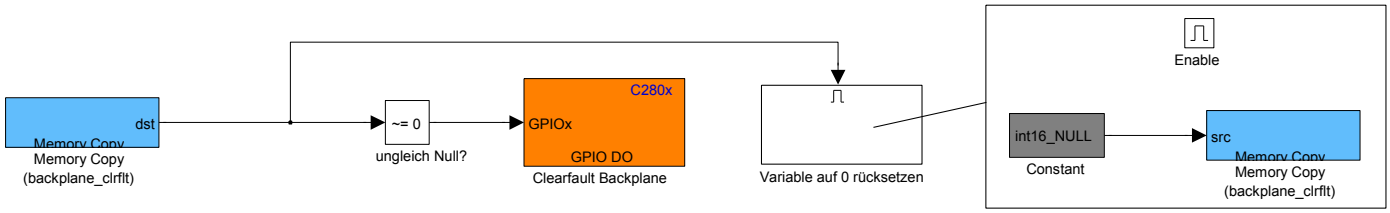

Abb. 5.3: Hardwareansteuerung der Hauptplatine eines einfachen Taktsignals

Um die Regelung in Gang zu setzen, müssen zu Beginn eine Reihe an Sicherheitsmechanismen per Hand initialisert werden. Dies betrifft unter anderem die Überstromabschaltung des Umrichters, welche auf der Hauptplatine einen Trip-Zone-Interrupt auslösen kann, oder das Schalten eines Laderelais. Da sich die verwendete Hardware ohnehin von Projekt zu Projekt unterscheidet, wird hier nur auf die Prinzipien der Hardwareansteuerung eingegangen. Die genaue Realisierung in dieser Arbeit wird als irrelevant erachtet. In Abbildung 5.3 ist die Ausgabe eines einfachen Taktsignals an einem digitalen Ausgang mit Hilfe des vorgefertigten „GPIO“-Blocks dargestellt. Diese Signalform ist beispielsweise zum Rücksetzen von FlipFlops erforderlich. In jedem Regelschritt wird eine Referenzvariable auf ihren Wert geprüft. Wurde dieser vom Benutzer auf einen Wert ungleich Null gesetzt, wird der Ausgang auf logisch „ein“ gelegt. Im Anschluss wird ein Subsystem ausgeführt, das die Referenzvariable wieder in den Ausgangszustand versetzt.
Ist kein Taktsignal, sondern eine einfache Flanke gefordert, wird das Subsystem zum Rücksetzen einfach weggelassen. Der Benutzer bestimmt dann direkt durch seine Eingabe den Zustand des digitalen Ausgangs. Mit dem Oszilloskop wird die Funktion des digitalen Ausgangs nachgeprüft.

5.1.3. Integration des Lagegebers

Zur Integration des Lagegebers muss wiederum ein Simulink S-Funkionsblock erstellt werden, um den C-Quellcode, welcher für die Konfiguration des SPI-Datenbus und den Zugriff auf das Empfangsregister verantwortlich ist, einzubinden. Die Hardwareanbindung erfolgt, wie erwähnt, über den SPI-Datenbus. Da SPI-A bereits vom Softauge belegt ist, wird SPI-B herangezogen und der Resolver als Slave konfiguriert. Damit der Resolverchip Daten sendet, muss dieser durch die Taktleitung zuerst ein gültiges Taktsignal im Bereich von 500kHz bis maximal 1MHz empfangen. Dies geschieht allerdings nur, wenn dem DSP zyklisch Daten in das Senderegister geschrieben werden, da nur dann an der Taktleitung ein Signal gesendet wird.
Die empfangenen Daten liegen im Empfangsregister als Werte von 0 bis 32767 vor und werden durch den S-Funktionsblock ausgelesen (siehe Abbildung 5.4). Die darauffolgenden Rechenoperationen dienen zur Umrechnung des Winkels ins übliche Format von -8 bis 7.9998.

Weiters wird einmalig ein Offsetableich vorgenommen, um die d-Achse als Winkel „0“ zu definieren. Da für die feldorientierte Regelung jedoch nicht der ermittelte mechanische Winkel ausschlaggebend ist, wird der offsetbefreite Winkel mit der Polpaarzahl multipliziert, um den elektrischen Winkel zu erhalten. Bei letztgenannter Umrechnung musste in dieser Arbeit ein Trick angewandt werden, um die zu Polpaarzahl (2p=20) der eingesetzten PSM im *fixdt(1,32,28)*-Zahlenformat verarbeiten zu können: Statt einer wird auf zwei Multiplikationen ausgewichen und somit der Wert „20“ durch die Faktoren „5“ und „4“ gebildet. Eine Umrechnung ins *fixdt(1,32,12)*-Zahlenformat, in dem der Wert „20“ direkt darstellbar ist, wäre nicht zielführend gewesen, weil dort der Überlauf des Winkels nicht stattfindet.

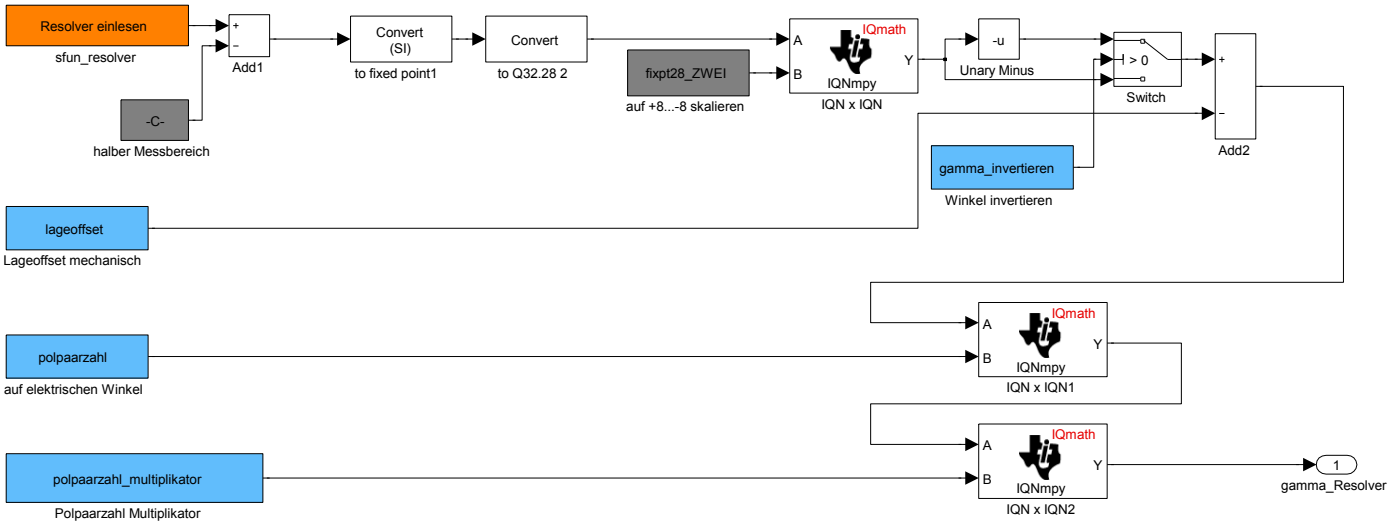

Abb. 5.4: Subsystem zum Einlesen des Winkelresolvers und Umrechnung der Messdaten

5.2. PWM-Raumzeigermodulation

Um in der Maschine einen definierten energetrischen Zustand hervorzurufen, ist in erster Linie ein Stellglied nötig, welches durch den Dreiphasenumrichter dargestellt wird. Dieser legt, basierend auf den Eingangssignalen, den drei PWM-Kanälen, einen Raumzeiger an die in Stern geschaltete PSM (siehe Abbildung 5.5). Liegt beispielsweise PWM-Kanal 1A auf logisch „ein“ und die beiden anderen Kanäle auf logisch „aus“ (Vektor „100“), so liegt die Spannung in U-Richtung an, welche mit der Raumzeigerachse α zusammenfällt. Durch Kombination der verschiedenen Eingangszustände kann ein Bereich in der Raumzeigerebene abgedeckt werden, der, wie in Abbildung 5.6 dargestellt, einem Sechseck entspricht.
Die Lage des gewünschten Raumzeigers, des Referenzraumzeigers, wird von zwei Basisspannungsraumzeigern in einem der sechs Sektoren begrenzt. Liegt der Referenzraumzeiger beispielsweise im Sektor drei, wie in Abbildung 5.7, so wird dieser durch eine Linearkombination aus den Vektoren „110“, „100“ und den Nullspannungsraumzeigern „000“ und „111“

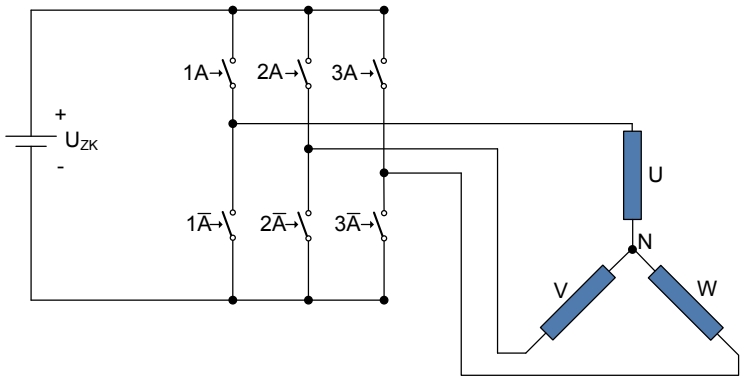

Abb. 5.5: Funktionsprinzip des Spannungszwischenkreisumrichters

zusammengesetzt. Grob gesprochen sind die Basisspannungsraumzeiger für die Richtung in der Raumzeigerebene verantwortlich, die Betragskorrektur wird durch die Nullspannungsraumzeiger vorgenommen. Ein hier nicht im Detail beschriebener Algorithmus, welcher als „Raumzeigermodulation“ bezeichnet wird, führt zu einem spezifisichen PWM-Muster. Für den interessierten Leser besteht die Möglichkeit, die genaue Funktionsweise der Raumzeigermodulation in Mohan (1998) nachzuschlagen.

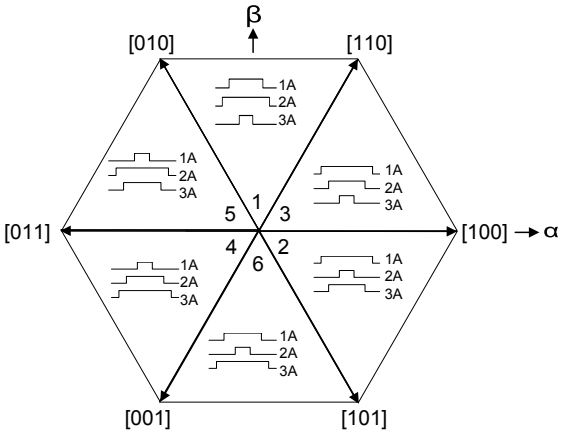

Abb. 5.6: Sechseck der Raumzeigermodulation

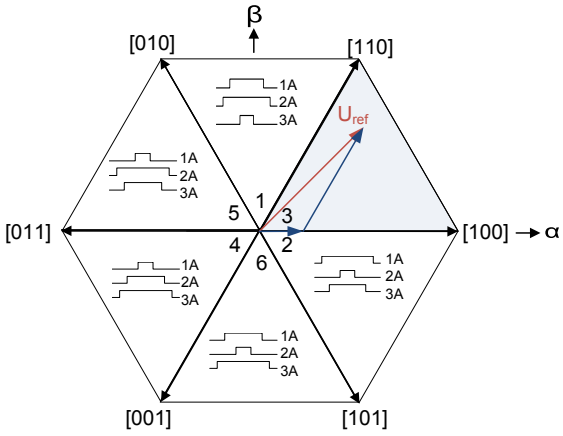

Abb. 5.7: Linearkombination der Basisraumzeiger

Im Rahmen dieser Arbeit ist es zuerst nötig, den bereits bestehenden C-Quellcode zur Raumzeigermodulation in das Simulink-Modell zu integrieren. Dies erfolgt nach dem in Kapitel 4.2 beschriebenem Schema mit Hilfe der Skriptdatei *create_sfun_pwma.m* und resultiert in einem Funktionsblock, welcher sowohl den Referenzraumzeiger in Komponentendarstellung des α-β-Koordinatensystems als auch die Zeitdauer einer PWM-Periode in CPU-Zyklen erwartet. Zurückgegeben werden die Compare-Werte für die drei PWM-Kanäle (siehe Kapitel 4.6.3). Somit kann jeder geforderte (betragsmäßig mögliche) Spannungsraumzeiger über den Umrichter angelegt werden.

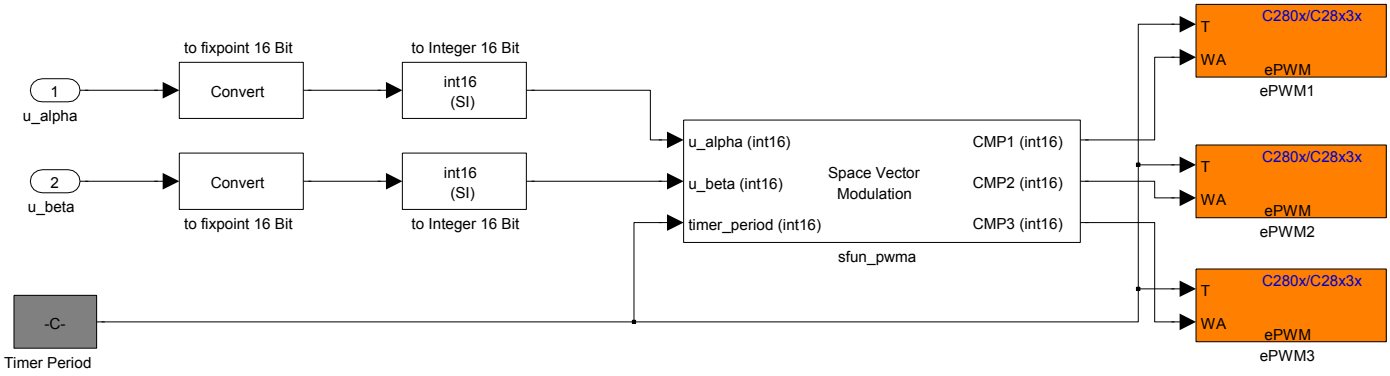

Abb. 5.8: S-Funktion „Raumzeigermodulation“ in Simulink

5.3. Gesteuerter Betrieb, Spannungsraumzeigervorgabe

Mit Verfügung über die PWM-Raumzeigermodulation kann bereits ein Drehfeld in die PSM eingeprägt werden, sofern als Eingang ein rotierender Raumzeiger aufgeschaltet wird. Es ist in Folge ein gesteuerter Betrieb mit Spannungsraumzeigervorgabe möglich. Allerdings ist nicht zu erwarten, dass die PSM mit besonderer Laufruhe oder großem Drehmoment angetrieben wird, da sich der Strom in der Maschine rein als Folge der Spannungseinprägung einstellt. Doch zur Verifizierung des implementierten Algorithmus der Raumzeigermodulation ist die Spannungsraumzeigervorgabe allemal ein hilfreiches Mittel.

An den Eingang der S-Funktion werden zwei um 90° versetzte Sinusschwingungen angelegt, die in ihrer Amplitude und Frequenz durch den Benutzer verstellbar sind. Leider kann dies mit dem vorgefertigten Sinusblock nicht realisiert werden, da dieser Block keine *Tunable Parameters* unterstützt. Der rotierende Raumzeiger wird also durch Kombination von Basisblöcken realisiert.

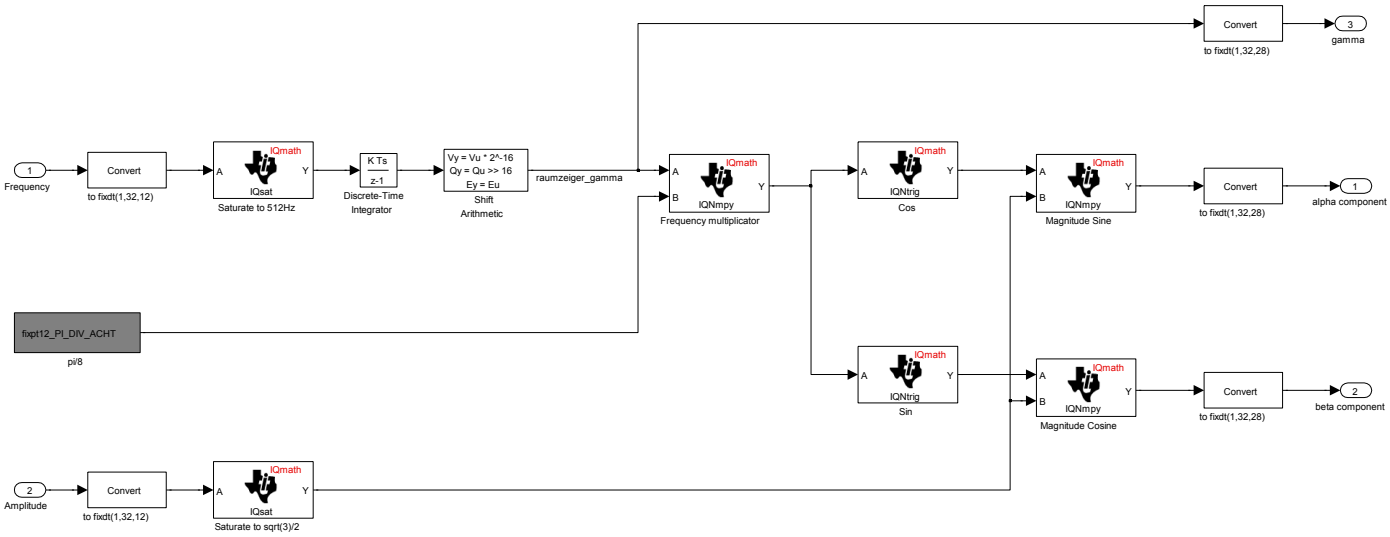

Abb. 5.9: Subsystem zur Bildung eines modifizierbaren Raumzeigers

Funktionsprinzip:

- Integrieren eines einstellbaren Wertes resultiert in einem unterschiedlich schnell laufendem Winkel.
- Umrechnen des Steigungswertes auf den Wertebereich -7.9998 bis +8. Überlauf des Zahlenbereichs ist erlaubt.
- Von diesem Winkel wird der Sinus und Cosinus berechnet
- Schließlich werden die Ergebnisse der trigonometrischen Funktionen mit einem Amplitudenfaktor gewichtet.

5.4. Strom- und Spannungsmessung, Skalierung der Messwerte

Um als nächsten Schritt die Stromregelung in Gang zu setzen, muss zuerst sichgestellt werden, dass die Strommessung korrekt funktioniert. Hierfür werden zuerst die zu messenden Kanäle des „ADC“-Funktionsblocks aus Kapitel 4.6.2 eingestellt. Da laut Staffler (2005) die Strommessung als Mittelwert vier gemittelter Messpunkte den zufälligen Messfehler für das später folgende INFORM-Verfahren deutlich reduziert, wird für die Strommessung des „normalen“ Regelkreises ebenfalls selbige Konfiguration verwendet. Dies bringt den Vorteil, dass die Messkanäle für die INFORM-Messungen nicht extra umkonfiguriert werden müssen und nach dem Schema aus Tabelle 5.1 belassen werden können. Der Vollständigkeit halber wird hier auch die Spannungsmessung angeführt, da diese in Folge auch für das EMK-Modell benötigt wird.

ADC-Register	Hardwareeingang	Messgröße
ADCRESULT0	ADCINA2	Strom in Phase U, 1. Messung
ADCRESULT1	ADCINA2	Strom in Phase U, 2. Messung
ADCRESULT2	ADCINA2	Strom in Phase U, 3. Messung
ADCRESULT3	ADCINA2	Strom in Phase U, 4. Messung
ADCRESULT4	ADCINA3	Strom in Phase V, 1. Messung
ADCRESULT5	ADCINA3	Strom in Phase V, 2. Messung
ADCRESULT6	ADCINA3	Strom in Phase V, 3. Messung
ADCRESULT7	ADCINA3	Strom in Phase V, 4. Messung
ADCRESULT8	ADCINA4	Strom in Phase W, 1. Messung
ADCRESULT9	ADCINA4	Strom in Phase W, 2. Messung
ADCRESULT10	ADCINA4	Strom in Phase W, 3. Messung
ADCRESULT11	ADCINA4	Strom in Phase W, 4. Messung
ADCRESULT12	ADCINB1	Spannung an Phase U
ADCRESULT13	ADCINB2	Spannung an Phase V
ADCRESULT14	ADCINB3	Spannung an Phase W

Tab. 5.1: Zuordnung der ADC-Kanäle

Die Abbildung 5.10 zeigt die Mittelung als Summierung vierer Messpunkte eines Kanals mit anschließender Bit-Verschiebung um zwei Bits nach rechts. Die Skalierung der Messwerte erfolgt weiters im einem Subsystem, welches aufgrund seiner Einfachkeit nicht abgebildet ist. Die Strommesswerte der drei Phasen werden hier ins Fixpunktformat umgewandelt und mit dem Skalierungsfaktor aus Kapitel 3.1.1 multipliziert.
Nach der Skalierung werden die konstanten Messfehler jeder Phase bei stehender Maschine ermittelt und nach Wunsch des Benutzers vom Messwert abgezogen. Dieser Vorgang ist vor allem erforderlich, um den Temperaturdrift der Messeinrichtung zu kompensieren. Ein einmaliger Abgleich nur beim Einschalten des Systems ist daher nicht ausreichend. Wenn die PSM stillsteht, erfolgt ein automatischer Abgleich, dessen Wert bis zum nächsten Abgleich gespeichert und verwendet wird.

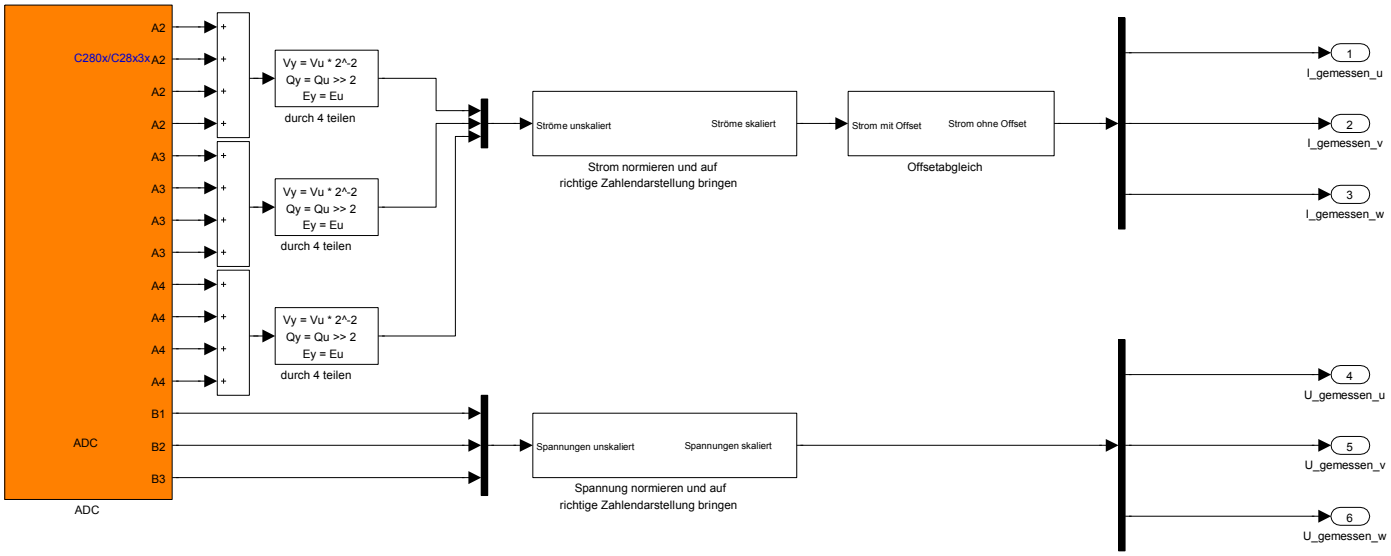

Abb. 5.10: Strom- und Spannungsmessung, Skalierung der Messwerte

Für die Spannungsmesswerte wird ein ähnliches Vorgehen angewandt. Ein Unterschied besteht nur im Skalierungsfaktor und in der Tatsache, dass die Spannungsmesswerte keine Offsetverschiebung aufweisen. Weil als Bezugspotential der Spannungsmessung U_{ZK-} gewählt wird, sind die skalierten Messwerte im Gegensatz zur Strommessung bis auf den hier vernachlässigten Wert einer Diodenflussspannung nicht bipolar. Es sei angemerkt, dass letztgenannte Tatsache für die Raumzeigerrechnung nicht gilt: Die gebildeten Spannungsraumzeiger sind dann sehr wohl bipolare Größen, da Offsetwerte bei der Bildung der Raumzeiger herausfallen.

5.5. Geregelter Betrieb, Stromregler-Implementierung

Der Stromregler ist in dieser Arbeit als PI-Regler mit zwei Komponenten ausgeführt, welcher die d- und q-Komponente des Statorstroms unabhängig voneinander ausregelt. Der P-Anteil stellt eine konstante Verstärkung der Regeldifferenz von Soll- und Iststrom dar, wohingegen der I-Anteil im übertragenen Sinne als „Gedächtnis“ des Systems agiert und auf eine bleibende Regelabweichung mit wachsender Ausgangsgröße und je nach Nachstellzeit unterschiedlich rasch antwortet. Der Regler ist nur dann aktiv, wenn die PWM freigegeben oder die Sollgröße betragsmäßig unterhalb der Begrenzung ist. Trifft dies nicht zu, wird der Propotionalanteil deaktiviert, während der I-Anteil mit einer festgelegten Nachstellzeit abintegriert (Anti-Windup-Mechanismus). In Abbildung 5.11 ist die Simulink-Ausführung eines solchen PI-Reglers für die d-Komponente des Stroms dargestellt.

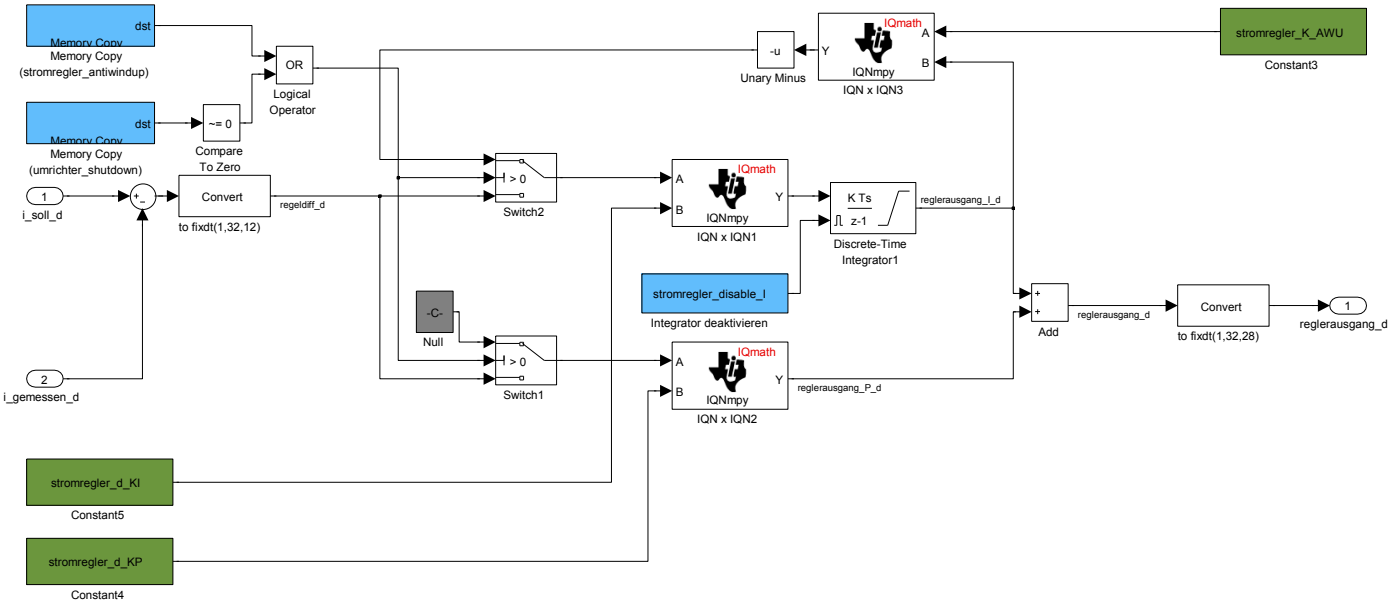

Abb. 5.11: Simulink-Subsystem „Stromregler“, hier: d-Anteil

Zu beachten ist, dass die Reglerparameter im *fixdt(1,32,12)*-Format zu konfigurieren sind, da der Regler intern in diesem Format arbeitet. Dies gründet auf der Tatsache, dass zum Einen theoretisch größere P-Verstärkungen als die im *fixdt(1,32,28)*-Format möglichen Werte eingegeben werden können, und zum Anderen vor allem darauf, dass sich die Nachstellzeit des Simulink-Integratorblocks als Integralverstärkung versteht, welche dem inversen Wert einer gewünschten Nachstellzeit entspricht. Eine Nachstellzeit von 500μs entspricht beispielsweise einer Integralverstärkung von 2000. Somit wird klar, weshalb für den PI-Regler das ausgeweitete Zahlenformat von Nöten ist.
Für den Stromregler wurde der Parametersatz aus Tabelle 5.2 als beste Lösung gefunden:
In der Praxis ergeben sich die typischen Verläufe des PI-Reglers (siehe Abbildungen 5.12 und 5.13), wobei der Sollwert blau und die gemessenen Ströme grün gekennzeichnet sind.

Reglerkonstante	d-Anteil	q-Anteil
$K_{p,Strom}$	1.40	0.96
$K_{i,Strom}$	1400	1100
$K_{AntiWindup,Strom}$	1.00	1.00
$T_{i,Strom} \widehat{=} \frac{1}{K_{i,Strom}}$	714.29μs	909.09μs

Tab. 5.2: Gewählte Parameter des Stromreglers

Bei einem Nennsprung nimmt der Strom etwa nach einer Milisekunde seinen stationären Endwert ein, sowohl in d- als auch in q-Richtung. Prinzipiell wäre nach Schrödl (2000) auch die Einbindung eines Entkopplungsnetzwerkes in Frage gekommen. Doch auch die Wahl verschiedener Parameter für d- und q-Regler erzielte zufriedenstellende Ergebnisse.
Zur Erklärung der Oszilloskopdarstellung: Aufgetragen ist die Signalform in Zeitdarstellung mit einer Y-Skalierung am Oszilloskop von 1 Volt pro Division. Das Softauge liefert Ausgangsspannungen in Bereich von -8 Volt bis +8 Volt, um den kompletten Zahlenbereich des *fixdt(1,32,28)*-Formats abdecken zu können. Der Wert „1“ in diesem Format resultiert in einer Spannung von 1 Volt am Softaugeausgang. Bei einer Verstärkung um den Faktor zwei liegen am Ausgang folglich 2 Volt an. Deshalb nimmt ein Nennsprung bei doppelter Verstärkung hier zwei Divisions am Oszilloskopschirm ein.

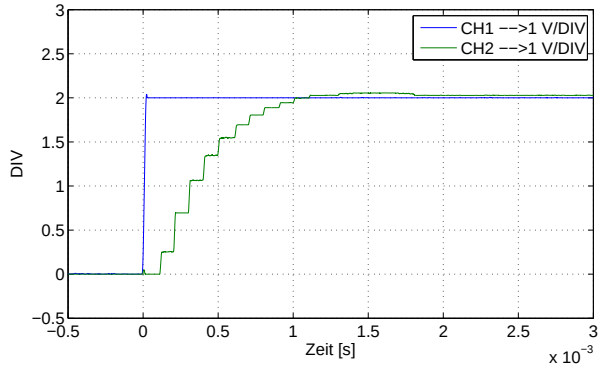

Abb. 5.12: Nennsprung in d-Richtung, Softauge ×2

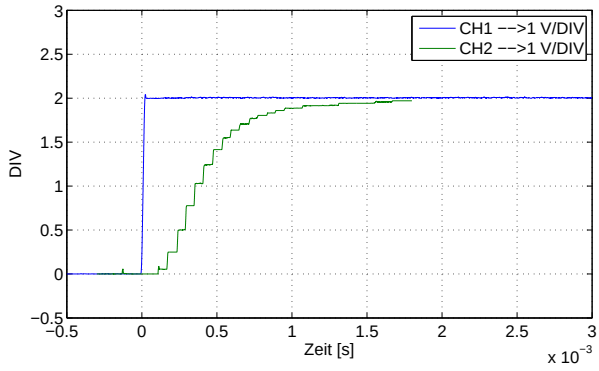

Abb. 5.13: Nennsprung in q-Richtung, Softauge ×2

5.6. Drehzahlregler-Implementierung

Als nächster Schritt soll der Drehzahlregler in einem langsameren Task in das Modell integriert werden. Dies setzt voraus, dass die Interruptstruktur nach Abbildung 4.22 mit Kurz- und Langtask aufgesetzt wird. Der Langtask benötigt ebenso wie der Kurztask einen Interruptauslöser, welcher wiederum durch einen PWM-Zähler realisiert werden kann. Die PWM-Kanäle 1 bis 3 werden für die Raumzeigermodulation verwendet, der 4. Kanal ist für die Highside-Spannungsversorgung des Umrichters erforderlich, also wird der freie Kanal 6 für die Interruptsteuerung des Langtasks herangezogen. Die Konfiguration des „PWM“-

Blocks des Drehzahlreglers erfolgt wie in Abbildung 5.14 und 5.15. Alle anderen Reiter des Dialogfensters sind nicht weiter ausschlaggebend, bis auf die Tatsache dass unter „PWMA“ die Kontrollbox „Enable PWMA“ aktiviert sein muss.
Der Drehzahlregler selbst ist von der Struktur ident mit dem Stromregler. Die Reglerparameter wurden entsprechend der Tabelle 5.3 gewählt. Die Führungsgröße bildet nun eine Solldrehzahl, die Messgröße ist die aktuelle Rotordrehzahl und der Reglerausgang wird durch den drehmomentbildenden q-Strom dargestellt. Wichtig ist der Datenaustausch zwischen dem kurzen und dem langen Task, dessen Datenintegrität durch den „Transfer“-Block gewährleistet ist[14].

Reglerkonstante	Wert
$K_{p,Drehzahl}$	1.45
$K_{i,Drehzahl}$	5.40
$K_{AntiWindup,Drehzahl}$	1.00
$T_{i,Drehzahl} \widehat{=} \frac{1}{K_{i,Drehzahl}}$	185.19ms

Tab. 5.3: Gewählte Parameter des Drehzahlreglers

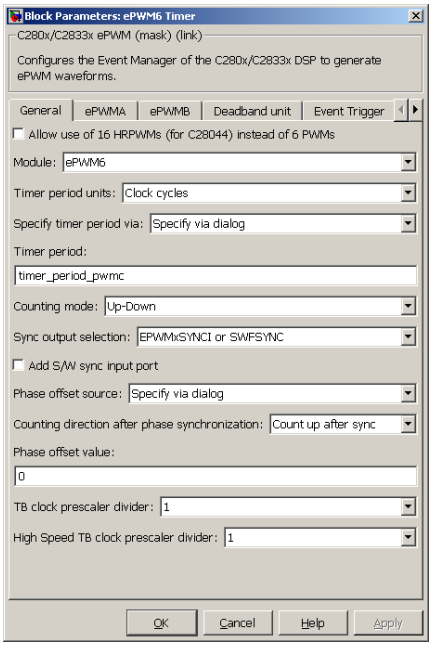

Abb. 5.14: PWM-Dialogfenster „General“

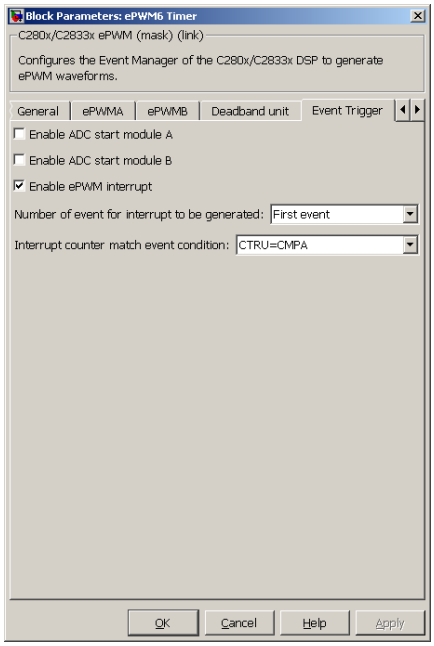

Abb. 5.15: PWM-Dialogfenster „Event Trigger“

Um aus den aktuellen Winkelinformationen die Drehzahl zu ermitteln, werden die Winkeldifferenzen jedes Verarbeitungsschrittes wie in Abbildung 5.16 über eine gewissen Zeit mit einem PT1-Filter gemittelt und mit einer Maschinenkonstante ω_{Kp} gewichtet. Diese Kon-

[14]Siehe Abbildung 5.1: Block zwischen den Subsystemen von Kurz- und Langtask

stante hängt von Maschinenparametern ab (siehe Gleichung 5.1) und hat den Zweck, die Winkeldifferenzen so umzurechnen, dass eine bezogene mechanische Drehzahl am Ausgang folgt.

$$\omega_{Kp} = \frac{60}{T_{kurz} \cdot \omega_N \cdot 2p \cdot 16} \tag{5.1}$$

Filterkonstante	Wert
ω_{Kp}	3.125
ω_{Ki}	200
$T_{i,PT1} \widehat{=} \frac{1}{\omega_{Ki}}$	5.0ms

Tab. 5.4: Gewählte Parameter des PT1-Drehzahlfilters

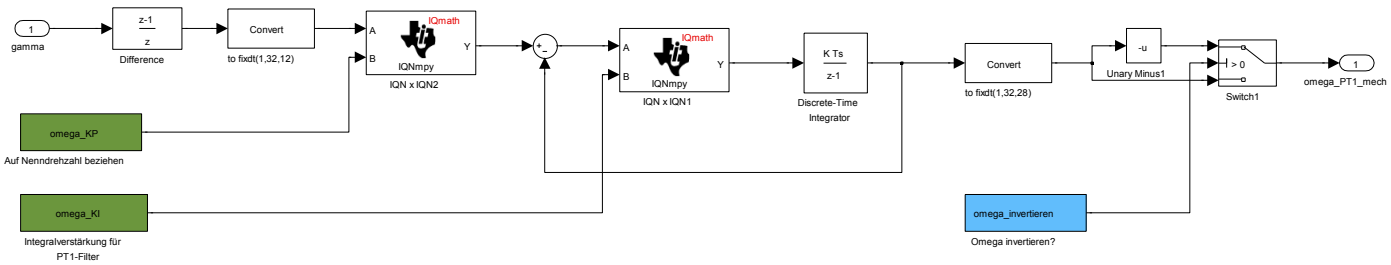

Abb. 5.16: Subsystem zur Drehzahlermittlung mittels eines PT1-Filters

In Abbildung 5.17 ist die Ansicht einer Oszilloskopmessung dargestellt:

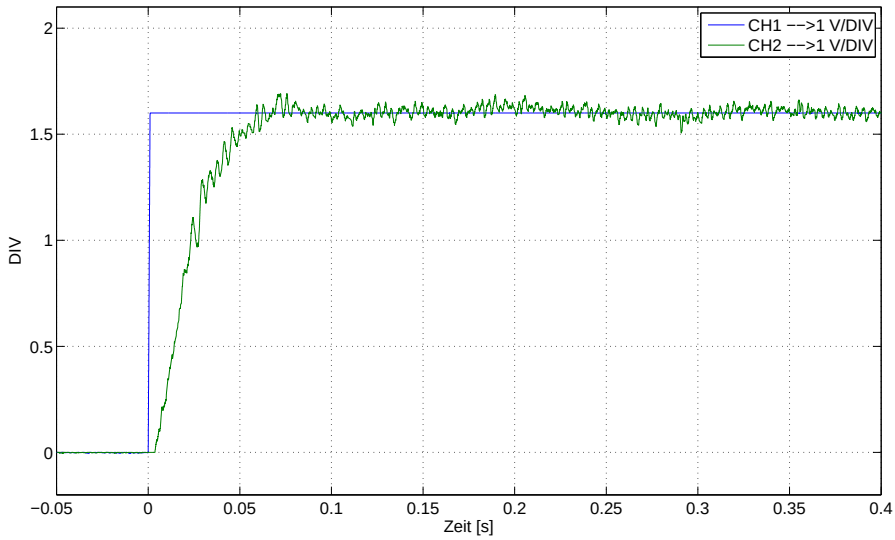

Abb. 5.17: Sprung auf 20% der Nenndrehzahl, Softauge ×8

Bei stehender Maschine wurde ein Drehzahl-Sollwert von 20% der Nenndrehzahl aufgeschaltet. Dieser Wert wurde gezielt gewählt, da hier die Stellgröße des Stromreglers noch nicht in Begrenzung läuft und somit die Drehzahlkennlinie den bekannten Verlauf eines PI-Reglers einnimmt. Bei unbelasteter Maschine ist der eingestellte Sollwert nach etwa 60ms erreicht. Messungen mit Last wurden in dieser Arbeit nicht durchgeführt, da in erster Linie die Machbarkeit und nicht die Optimierung der Regelung im Vordergrund stand.

5.7. EMK-Modell

Mit dem Drehzahlregler wurde die Basisanforderung der feldorientierten Regelung mit Lagegeber geschaffen. Im Weiteren konzentriert sich die Arbeit auf den sensorlosen Betrieb: Ziel ist es, eine Kombination aus EMK-Modell, INFORM-Methode und Drehzahlbeobachter zu realisieren, wie dies in Schrödl/Hofer/Staffler (2006) vorgeschlagen wird. Das EMK-Modell liefert hierbei den Transformationswinkel für die höheren Maschinendrehzahlen ab ca. 5% bis 8% der Nenndrehzahl. Ab dieser Richtmarke bekommt man aus der Integration der Statorspannungsgleichung 2.5 zuverlässige Werte für die Statorflussverkettung, aus der schließlich der Winkel berechnet werden kann. Die Eingangsgrößen der Statorspannungsgleichung bilden die im Umrichter gemessenen Größen Statorstrom und Statorspannung aus Abbildung 5.18 und 5.19.

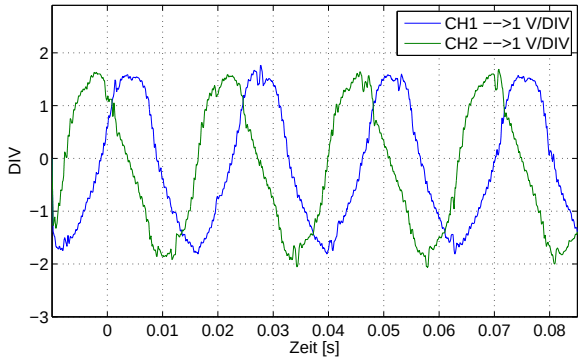

Abb. 5.18: Spannungsverlauf u_α (blau) und u_β (grün)

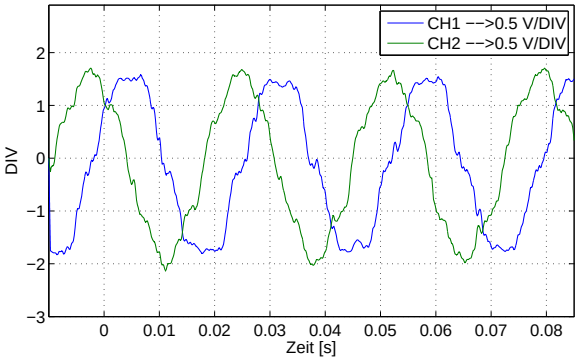

Abb. 5.19: Stromverlauf i_α (blau) und i_β (grün)

Die Werte des Statorwiderstandes und der Statorinduktivität werden als bekannt vorausgesetzt, beeinflussen über deren Genauigkeit jedoch stark die Ergebnisse des EMK-Modells. Deshalb ist es unerlässlich, die Bestimmung der Maschinenparameter sorgfältig durchzuführen. Zuerst wird Gleichung 2.5 umgeformt zu:

$$\underline{\psi}_s(\tau) = \int (\underline{u}_s(\tau) - \underline{i}_s(\tau) \cdot r_s) d\tau \tag{5.2}$$

Nach Einsetzen der Gleichung 2.6 ergibt sich:

$$\underline{\psi}_M(\tau) = \int (\underline{u}_s(\tau) - \underline{i}_s(\tau) \cdot r_s) d\tau - \underline{i}_s(\tau) \cdot l_s \tag{5.3}$$

Hieraus kann prinzipiell der Winkel bereits aus dem Argument von $\underline{\psi}_M(\tau)$ errechnet werden.

$$\gamma_{EMK} = arg\left(\underline{\psi}_M(\tau)\right) = arctan\left(\frac{\psi_{M,\beta}(\tau)}{\psi_{M,\alpha}(\tau)}\right) \tag{5.4}$$

Allerdings führt die offene Integration in diesem Ansatz leicht zu einem Driftverhalten. Dem kann abgeholfen werden, indem der Integrator mit einem Stabilisierungsfaktor K_{Stab} rückgekoppelt wird. Regelungstechnisch gesehen verhält sich die resultierende Struktur wie ein PT1-Glied: Weit überhalb der Knickfrequenz entspricht das Verhalten dem eines idealen Integrators, weit unterhalb überwiegt das Proportionalverhalten und begrenzt den Einfluss niederfrequenter Störungen. Durch geeignete Wahl des Rückkoppelfaktors kann das Modell bis zu sehr kleinen Drehzahlen (etwa 5% der Nenndrehzahl) zuverlässig betrieben werden. Allerdings muss immer ein Kompromiss zwischen Robustheit der Regelung und Möglichkeit des Betriebs mit niedrigen Drehzahlen eingegangen werden (siehe Abbildung 5.20). Bewährt haben sich Faktoren im Bereich von $K_{Stab} = 0.06 \pm 0.02$.

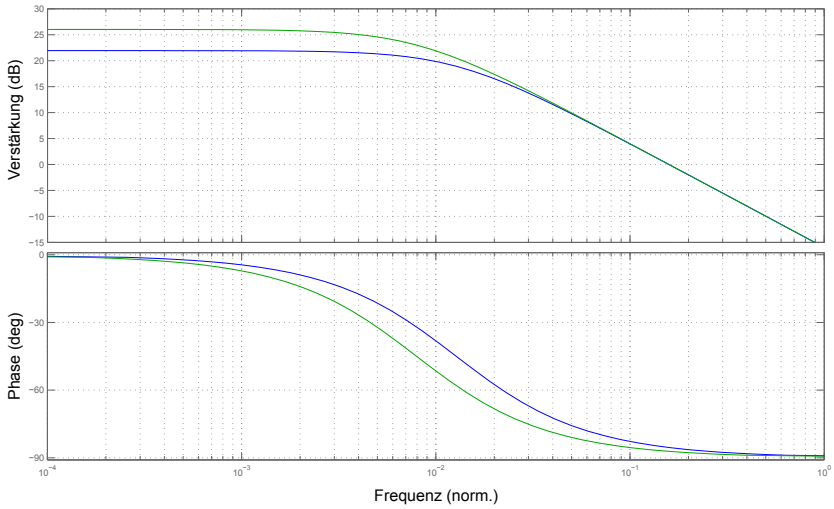

Abb. 5.20: Übertragungsfunktion mit Rückkopplung, Stabilisierungsfaktor $K_{Stab} = 0.05$ (grün), Stabilisierungsfaktor $K_{Stab} = 0.08$ (blau)

Die Parameter des EMK-Modells wurden im konkreten Fall entsprechend Tabelle 5.5 gewählt.

In Simulink wurde das EMK-Modell wie in Abbildung 5.21 dargestellt eingebunden. Da für die α- und β-Komponenten die gleichen Operationen erfolgen, werden die beiden Signale zuerst zu einem Signalvektor zusammengefasst. Dies vereinfach in Folge die Struktur

Reglerkonstante	Wert
K_{Stab}	0.05
$K_{i,EMK} = \omega_{Bezug}$	200
$T_{i,EMK} \widehat{=} \frac{1}{K_{i,EMK}}$	5ms

Tab. 5.5: Gewählte Parameter des EMK-Modells

und verbessert die Übersichtlichkeit des Systems. Wie beim Stromregler oder Drehzahlregler sind hier diskrete Integratoren im Spiel, die ihre Nachstellzeit als Integralverstärkung verstehen. Um mittels großer Integralverstärkungen schnelle Integratoren zu ermöglichen, muss der mögliche Wertebereich vor der Integration durch Konvertierung der Signale ins *fixdt(1,32,12)*-Format ausgeweitet und nachher wieder ins Ausgangsformat zurückgewandelt werden. Schließlich wird aus den beiden Komponenten des Flussraumzeigers der Winkel ermittelt und in die übliche Darstellungsform skaliert. Da es beim EMK-Modell zu einem belastungsabhängigen Winkelfehler kommt, sollte dieser durch einen stromabhängigen Lageoffset ausgeglichen werden. Üblicherweise wird der Winkelfehler über die Belastung linear modelliert und kann hierdurch zufriedenstellend beseitigt werden.

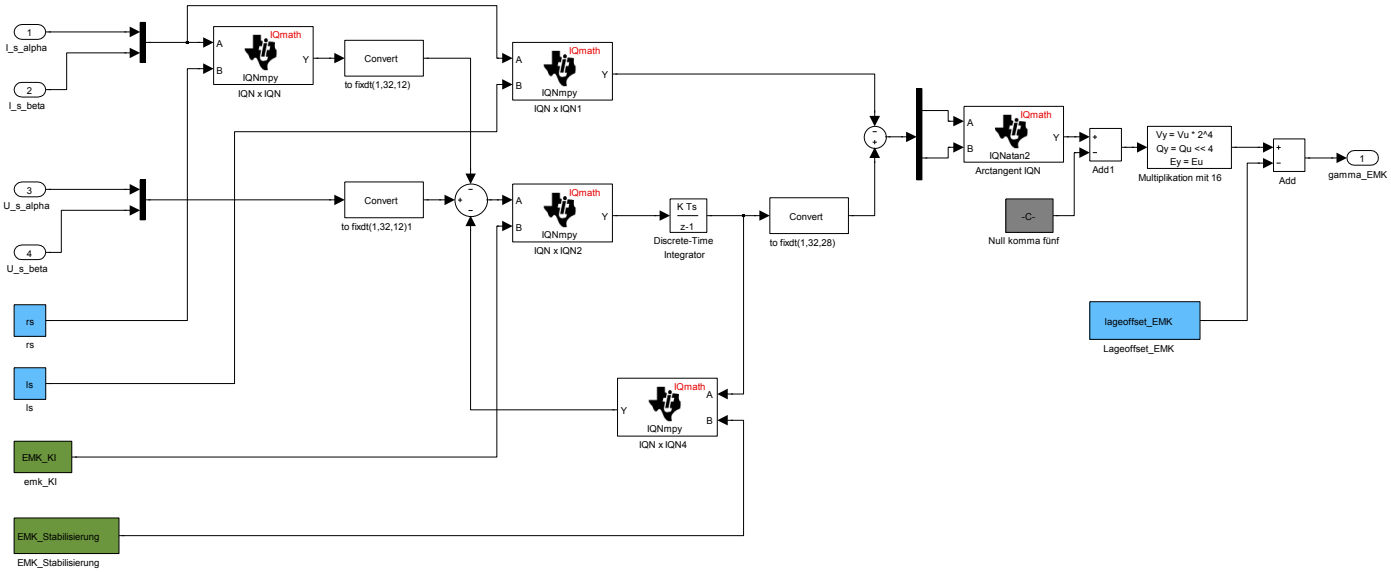

Abb. 5.21: Subsystem des EMK-Modells

Abbildung 5.22 zeigt die Gegenüberstellung des resultierenden EMK-Winkels mit dem Resolverwinkel bei einer Solldrehzahl von $\omega_{soll} = 0.05$. Bereits ab diesem niedrigen Wert kann die Rotorposition sehr gut abgeschätzt werden. Der Betrieb mit dem EMK-Winkel kennzeichnete sich durch ein deutlich niedrigeres Betriebsgeräusch des Reglers.

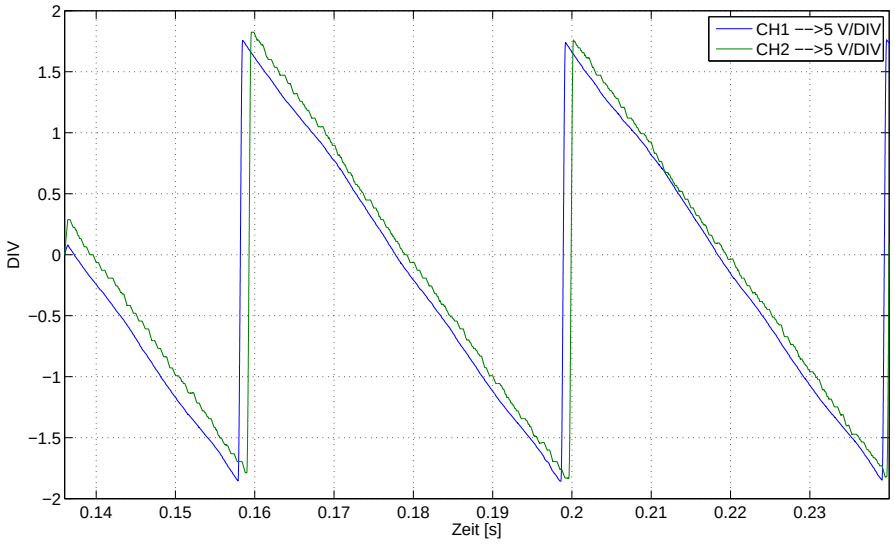

Abb. 5.22: EMK- (blau) und Resolverwinkel (grün)

5.8. INFORM-Methode

Grundlagen des INFORM-Verfahrens

Um die Maschine auch aus dem Stillstand und bei niedrigen Drehzahlen sensorlos regeln zu können, wird das aktive INFORM-Messverfahren implementiert. Hierzu werden zyklisch Spannungspulse an die PSM angelegt und die resultierenden Stromänderungen gemessen, welche von der aktuellen Rotorposition abhängen. Diese Abhängigkeit folgt aus der *Anisotropie*, dem schwankenden magnetischen Leitwert in der Maschine. Der Grund hierfür ist, dass in d-Richtung des Rotors durch die Permanentmagneten merklich weniger Eisen vorhanden ist. Als Folge magnetisiert sich das Ständereisen bei Durchflutung im Bereich um die Magnete stärker als im restlichen Bereich. Bei Erhöhung der Feldstärke bzw. des feldbildenden Stromes sättigen diese Teile des Stators früher aus und die Flussdichte steigt nur noch schwach an (siehe Abbildung 5.23).
Wie erwähnt wird dieses lageabhängige Verhalten beim INFORM-Verfahren erfasst und ausgewertet. Je größer die Anisotropie der Maschine ausgebildet ist, desto leichter lässt sich in Folge eine d-q-Abhängigkeit erkennen. Weiters wird die Genauigkeit durch Erhöhung der Anzahl auf vier Messpunkte pro Messung verbessert. Die Ermittlung der Achse der maximalen magnetischen Leitfähigkeit, welche in Folge die Ermittlung der Rotorlage zulässt, erfolgt durch Anlegen einer zyklischen Folge von Statorspannungsraumzeigern $\underline{u}_s$ und Messen der sich darauf einstellenden Stromänderungen $\frac{di_s}{d\tau}$. Zu beachten ist, dass sich der Rotor bei der Verknüpfung von Spannungs- und Stromänderungsraumzeiger im Stillstand befinden muss, da ansonsten die durch die Rotation induzierte Spannung ebenfalls berücksichtigt wer-

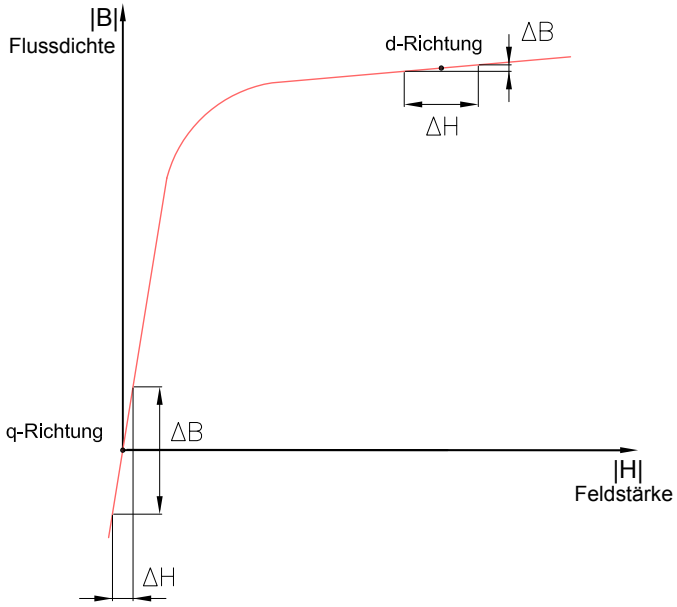

Abb. 5.23: Magnetisierungskennlinie der Neukurve der PSM im Bereich positiver magnetischer Feldstärken

den müsste. Im Folgenden wird kurz das Prinzip des Verfahrens als Zusammenfassung von Rieder (2005) erklärt. Für detaillierte Zusammenhänge wird auf diese Arbeit verwiesen.
Der Parameter

$$\underline{l}_s = \underline{u}_s \cdot \left(\frac{d\underline{i}_s}{d\tau} \right) \tag{5.5}$$

wird im Weiteren zur Lageberechnung herangezogen.
Die sechs für das INFORM-Verfahren relevanten mit dem Umrichter einstellbaren Spannungsraumzeiger wurden bereits in Kapitel 5.2 beschrieben und grafisch dargestellt (Abbildung 5.6).
Da der magnetische Leitwert auf Grund der geometrischen Bauweise der Maschine 180°-symmetrisch ist, wird die INFORM-Reaktanz $\underline{l}_s$ in weiterer Folge durch eine π-periodische Funktion charakterisiert.

$$\underline{l}_s = f\left(2\gamma_M - 2\gamma_U\right) \tag{5.6}$$

Die Bezeichner γ_M und γ_U stehen für die elektrische Lage des Rotorwinkels und die des eingeprägten Spannungsraumzeigers. Bei der Auswertung der Lage im DSP wird von der komplexen INFORM-Reaktanz (Gleichung 5.5) der inverse Wert verarbeitet, deshalb ist es geschickt, von Beginn an den Kehrwert heranzuziehen. Dies vermeidet im Prozessor unnötige Rechenzeit, da Divisionen besonders zeitintensiv sind.

$$\underline{y}_{INFORM} = \frac{1}{\underline{l}_s} \tag{5.7}$$

Der Verlauf der Ortskurve in Abhängigkeit beider Winkelparameter γ_M und γ_U bezogen auf die Richtung der maximalen Leitfähigkeit γ_{INFORM} wird durch die folgende komplexe Darstellung der inversen komplexen INFORM-Reaktanz beschrieben.

$$\underline{y}_{INFORM} = y_0 - \Delta y \cdot \mathrm{e}^{2 \cdot j \cdot \left(\gamma_{INFORM} - arg(\underline{u}_s)\right)} \tag{5.8}$$

Die konkreten Formeln zur Auswertung der Messsignale werden aus der Statorspannungsgleichung unter Vernachlässigung des Statorwiderstandes (Formel 5.9) hergeleitet.

$$\underline{u}_s(\tau) = \frac{d\underline{\psi}_s}{d\tau} + j \cdot \omega_k \cdot \underline{\psi}_s \tag{5.9}$$

Unter der Annahme, dass $\psi_M = konst$ gilt, wird die Statorflussgleichung in obige Formel eingesetzt.

$$\underline{u}_s(\tau) = l_s \cdot \frac{d\underline{i}_s}{d\tau} + j \cdot \omega_m \cdot \underline{\psi}_M \tag{5.10}$$

Werden nun an die Maschine hintereinander zwei entgegengesetzte gleich große Spannungsraumzeiger u_1 und $u_2 = -u_1$ angelegt, so kann durch Subtraktion der Gleichungen 5.11 und 5.12 und anschließende Umformung die Statorinduktivität ausgerechnet werden. Vorausgesetzt wird, dass sich die Rotorposition während der Zeit des Anliegens der beiden Spannungsraumzeiger nur marginal ändert.

$$\underline{u}_1 = l_s \cdot \frac{d\underline{i}_1}{d\tau} + j \cdot \omega_m \cdot \underline{\psi}_M \tag{5.11}$$

$$\underline{u}_2 = -u_1 = l_s \cdot \frac{d\underline{i}_2}{d\tau} + j \cdot \omega_m \cdot \underline{\psi}_M \tag{5.12}$$

$$\underline{l}_s = \frac{2 \cdot \underline{u}_1}{\frac{d\underline{i}_1}{d\tau} - \frac{d\underline{i}_2}{d\tau}} \tag{5.13}$$

Gleichung 5.13 beschreibt eine Erweiterung der Formel 5.5, wobei hier zusätzlich die Bewegung des Rotors berücksichtigt wird. Wie bereits erwähnt, wird die komplexe Reaktanz für die Verarbeitung im DSP als Kehrwert verwendet.

$$\underline{y}_{INFORM} = \frac{1}{\underline{l}_s} = \frac{\frac{d\underline{i}_1}{d\tau} - \frac{d\underline{i}_2}{d\tau}}{2 \cdot \underline{u}_1} \tag{5.14}$$

Die Stromanstiege $\frac{d\underline{i}_1}{d\tau}$ und $\frac{d\underline{i}_2}{d\tau}$ werden im konkreten Fall als Quotient der Differenzen der gemessenen Ströme Δi geteilt durch die jeweiligen Messintervalle $\Delta\tau$ dargestellt. Genaueres hierzu folgt in den Abschnitten 5.8.1 und 5.8.2 bei der genauen Erklärung der Messsequenzen. Ein großes Problem der INFORM-Messung stellt die 180°-Unsicherheit des ermittelten Winkels in Folge der unbekannten Polarität der Dauermagneten dar. Allerdings ist diese Unsicherheit für den laufenden Betrieb nicht mehr ausschlaggebend, sofern zu Beginn mit einer

korrekten Rotorposition initialisert wurde. Die Initialisierung kann mittels einer Großsignalmessung vorgenommen werden, deren Winkel dann der Kleinsignalmessung kontinuierlich aufgeschalten wird.

Das INFORM-Verfahren unter MATLAB/Simulink

Für die Implementierung der INFORM-Messung ist es von höchster Priorität, die Messzeitpunkte zum richtigen Zeitpunkt anzusetzen. Dies geschieht in der gewählten Realisierung in Simulink mittels eines in einer S-Funktion abgesetzten Kontrollblocks, welcher in Abbildung 5.24 dargestellt ist.

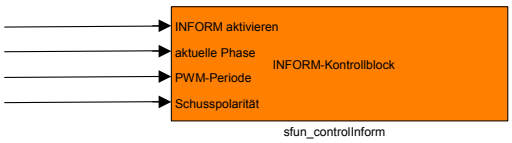

Abb. 5.24: Simulink INFORM-Kontrollblock

Der Kontrollblock übernimmt für das Messverfahren folgende Aufgaben:

- Umleiten des ADC-Interrupts *end of conversion* auf die INFORM-Routine zu Beginn der Messsequenz und Rücksetzen auf den Stromregler bei Beenden der Messsequenz: Liegt am Eingang logisch „ein“, so wird die INFORM-Routine aktiv, bei logisch „aus“ übernimmt der Stromregler wieder seine Arbeit.
- Einstellen der PWM-Periodendauern: Es werden die Zählerwerte übergeben, welche für die PWM im nächsten Interruptzyklus gültig sein sollen.
- Vorgeben der INFORM-Schusspolarität für die aktuell aktive Phase: Die Schusspolaritäten werden entsprechend dem Vorzeichen der Spannung spezifiziert. „+1“ entspricht einem positiven Schuss, „-1“ einem negativen.

Ist das INFORM-Verfahren gerade aktiv, so werden der Stromregler und der Drehzahlregler unterbrochen, solange, bis die Messsequenz wieder abgeschlossen ist. Während dieser Zeit tritt eine vom Messsignal abhängige Anzahl an INFORM-Interrupts auf, deren Zeitpunkte durch die Messzeitpunkte definiert sind. Ein Schema dieses Ablaufs ist in Abbildung 5.25 skizziert. Die Zeitintervalle und Dauern der Interruptroutinen sind hier nicht maßstabsgetreu abgebildet, sondern dienen nur dem Zwecke der Illustration.
In Abbildung 5.26 ist die Simulink-Implementierung des ISR-Blocks für die INFORM-Messung dargestellt. Je nachdem, ob eine Kleinsignal- oder eine Großsignalmessung durchgeführt wird, werden die Strommesswerte einem unterschiedlichen Subsystem zugeführt. Diese werden später genauer erklärt. Als Ausgangssignal wird, unabhängig von der Messsequenz, immer der Kleinsignal-Winkel zurückgegeben. Dies liegt daran, dass der Großsignalwinkel

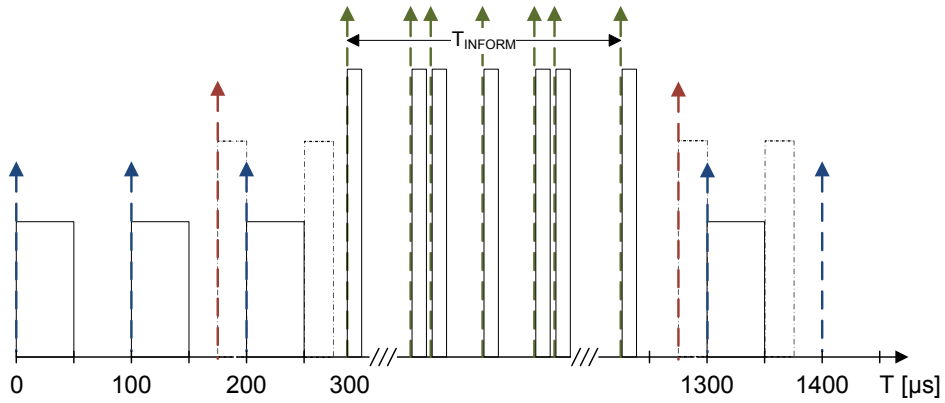

Abb. 5.25: Interruptschema der Regelung mit INFORM-Sequenzen

zur Initialisierung des INFORM-Verfahrens in einer Variablen abgelegt wird, auf welche der in Kapitel 5.9 beschriebene Drehzahlbeobachter zurückgreift.

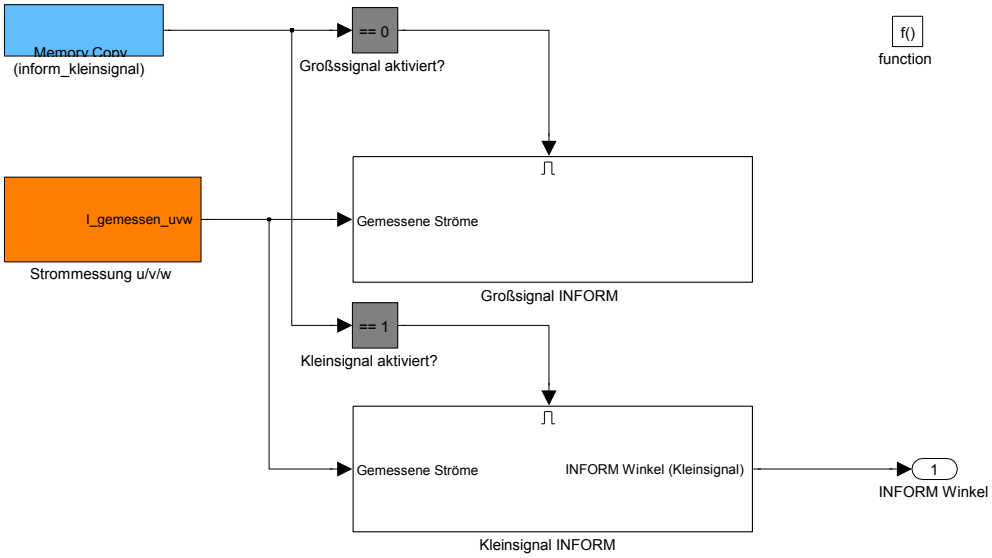

Abb. 5.26: Subsystem der INFORM-ISR

5.8.1. Großssignal-INFORM

Auswertung des Großssignals

Die Großsignal-Messsequenz dient, wie bereits erwähnt, der Initialisierung des Verfahrens durch die eindeutige Bestimmung des Startwinkels und somit der Eliminierung der 180°-Unsicherheit. Die Eindeutigkeit des Winkels wird durch eine viel stärkere Auslenkung aus dem magnetischen Arbeitspunkt als bei der Kleinsignalmessung gewährleistet, so dass der magnetische Kreis beidseitig in Sättigung geht. Wie aus dem oberen Teil der Abbildung 5.27

hervorgeht, liegt die Spannung an einem Strang der Maschine so lange an, bis sich Nennstrom einstellt. Anschließend wird die Spannung negativ angelegt, bis letztlich negativer Nennstrom erreicht wird. Zuletzt wird wieder in den ursprünglichen Arbeitspunkt zurückgekehrt. Während dieses Spannungs-/Stromverlaufs wird der Strom an geeigneten Zeitpunkten gemessen und bis zur Verarbeitung der Daten zwischengespeichert. Abbildung 5.28 zeigt die Oszilloskopmessung eines Großssignal-Schusses in der Phase U. Das blaue Signal entspricht den Interrupt-Zeitpunkten, welche durch den PWM-Zähler ausgelöst werden und an einem digitalen Ausgang mit dem Tastkopf abgegriffen werden. Das grüne Signal hingegen kennzeichnet den Stromverlauf. Deutlich zeigt sich Übereinstimmung mit dem theoretischen Schussmuster.

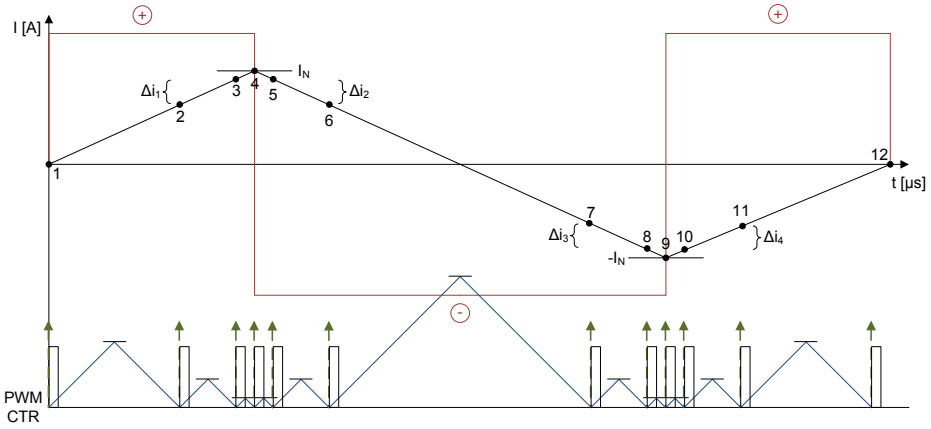

Abb. 5.27: Großsignal INFORM-Sequenz

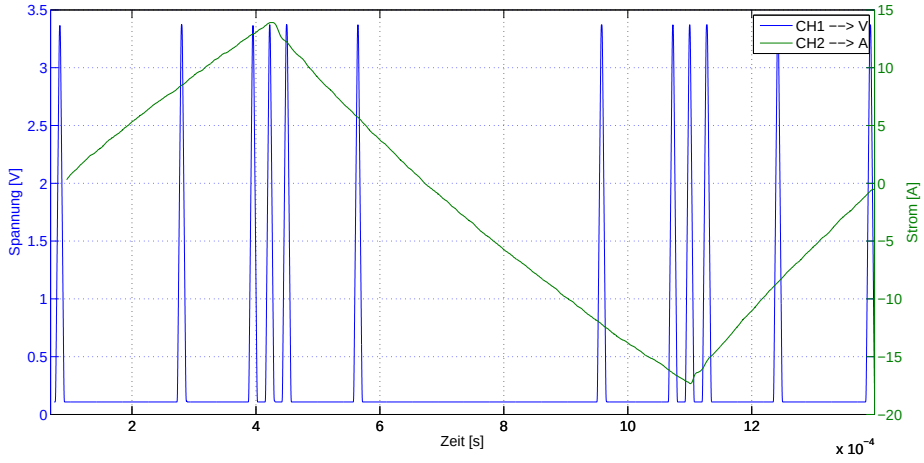

Abb. 5.28: Großsignal-Oszilloskopmessung: digitaler I/O (blau), Strom in Phase U (grün)

Durch Anwendung der in 5.7 hergeleiteten Formeln wird der Winkel mit den Strommess-

werten aus den Messpunkten 2, 3, 5, 6, 7, 8, 10 und 11 (siehe Abbildung 5.27) und den entsprechenden Messzeitpunkten $\Delta\tau$ ausgewertet. Beschrieben ist exemplarisch das Vorgehen für die Phase U; durchgeführt wird die Messung allerdings hintereinander für alle drei Phasen. Zuerst bildet man aus den einzelnen Messpunkten die Differenzen

$$\begin{aligned} \Delta i_1 &= i_3 - i_2 \\ \Delta i_2 &= i_6 - i_5 \\ \Delta i_3 &= i_8 - i_7 \\ \Delta i_4 &= i_{11} - i_{10} \end{aligned} \tag{5.15}$$

und führt diese Werte anschließend weiter zusammen durch:

$$\begin{aligned} \Delta i_{u+} &= \Delta i_1 - \Delta i_2 \\ \Delta i_{u-} &= \Delta i_3 - \Delta i_4 \end{aligned} \tag{5.16}$$

Für die anderen Stränge gilt dies analog, wodurch schließlich für jeden Strang ein Differenzwert ermittelt wird:

$$\begin{aligned} \overline{\Delta i_u} &= \Delta i_{u+} - \Delta i_{u-} \\ \overline{\Delta i_v} &= \Delta i_{v+} - \Delta i_{v-} \\ \overline{\Delta i_w} &= \Delta i_{w+} - \Delta i_{w-} \end{aligned} \tag{5.17}$$

Aus den Strang-Differenzwerten wird nun die komplexe Linearkombination gebildet und als Stromänderungsraumzeiger $\overline{\Delta \underline{i}}$ bezeichnet.

$$\overline{\Delta \underline{i}} = \frac{2}{3}\left(\overline{\Delta i_u} + \overline{\Delta i_v} \cdot \mathrm{e}^{j\frac{2\pi}{3}} + \overline{\Delta i_w} \cdot \mathrm{e}^{j\frac{4\pi}{3}}\right) \tag{5.18}$$

Den komplexen Stromänderungsraumzeiger aus Gleichung 5.18 setzt man in Gleichung 5.14 ein, um die inverse komplexe INFORM-Reaktanz zu erhalten.

$$\underline{y}_{INFORM} = \frac{\overline{\Delta \underline{i}}}{2 \cdot \underline{u} \cdot \Delta\tau} \tag{5.19}$$

Stellt man die Exponentialdarstellung aus Gleichung 5.8 nach $\underline{u}$ um und setzt diese in Gleichung 5.19 ein, so folgt der Zusammenhang

$$\overline{\Delta \underline{i}} \cdot \mathrm{e}^{-j \cdot arg(\underline{u})} = 2 \cdot \Delta\tau \cdot |\underline{u}| \cdot \left(y_0 - \Delta y \cdot \mathrm{e}^{j \cdot 2 \cdot \left(\gamma_{INFORM} - arg(\underline{u})\right)}\right) \tag{5.20}$$

Die Kombination der Realteile der Gleichungen 5.20 und 5.18 ergeben die Terme

$$\begin{aligned} \Re\left\{\overline{\Delta \underline{i}} \cdot \mathrm{e}^{-j \cdot 0}\right\} &= \overline{\Delta i_u} = 2 \cdot \Delta\tau \cdot |\underline{u}| \cdot [y_0 - \Delta y \cdot \cos\left(2 \cdot \gamma_{INFORM}\right)] \\ \Re\left\{\overline{\Delta \underline{i}} \cdot \mathrm{e}^{-j \cdot \frac{4\pi}{3}}\right\} &= \overline{\Delta i_u} = 2 \cdot \Delta\tau \cdot |\underline{u}| \cdot [y_0 - \Delta y \cdot \cos(2 \cdot \gamma_{INFORM} - \frac{4\pi}{3})] \\ \Re\left\{\overline{\Delta \underline{i}} \cdot \mathrm{e}^{-j \cdot \frac{2\pi}{3}}\right\} &= \overline{\Delta i_u} = 2 \cdot \Delta\tau \cdot |\underline{u}| \cdot [y_0 - \Delta y \cdot \cos(2 \cdot \gamma_{INFORM} - \frac{2\pi}{3})] \end{aligned} \tag{5.21}$$

Aus dem Argument der komplexen Linearkombination folgt schließlich der doppelte elektrische Rotorwinkel.

$$\begin{aligned} \underline{c}_{INFORM} &= \overline{\Delta i_u} + \overline{\Delta i_v} \cdot \mathrm{e}^{j \cdot \frac{4\pi}{3}} + \overline{\Delta i_w} \cdot \mathrm{e}^{j \cdot \frac{2\pi}{3}} \\ arg\left(\underline{c}_{INFORM}\right) &= 2 \cdot \gamma_{INFORM} + \pi \end{aligned} \tag{5.22}$$

Umsetzung mittels MATLAB/Simulink

Die Umsetzung des oben beschriebenen Auswertungsalgorithmus erfolgt in Simulink in einem eigenen Subsystem, welches in Abbildung 5.29 dargestellt ist. Ein wesentlicher Bestandteil ist der bereits erläuterte Kontrollblock. Diesem werden die Parameter zur korrekten Steuerung des Pulsmusters vorgegeben. Da für die Großsignal-Sequenz der drei Phasen 33 Interrupts zur Abhandlung notwendig sind (m=3 Phasen mit je n=11 Interrupts, siehe Abbildung 5.27), werden drei Kontrollvektoren mit je m·n=33 Elementen erstellt[15]. Jedes Element beschreibt einen bestimmten Parameter des INFORM-Kontrollblocks.
Die Erklärung der drei Vektoren erfolgt im Folgenden allgemein, da das selbe Schema analog für die Kleinsignal-Sequenz zum Einsatz kommt:

- *phase_inform_gross_uvw* (Phasen des Schussmusters): Besteht aus einer Aneinanderreihung der Kennzeichner für die jeweiligen Phasen.

$$phase\_inform\_gross\_uvw = \left[\underbrace{0_1\ 0_2 \ldots 0_n}_{Phase\ U}\ \underbrace{1_1\ 1_2 \ldots 1_n}_{Phase\ V}\ \underbrace{2_1\ 2_2 \ldots 2_n}_{Phase\ W}\right]$$

- *timer_inform_gross_uvw* (PWM-Zählerperioden): Der Vektor der Zählerperioden wird nach dem folgenden Schema gebildet. Ausgangsbasis hierfür ist ein Vektor mit 12 Elementen, welcher die auf Eins normierten Auftrittszeitpunkte der INFORM-Interrupts enthält (siehe auch Abbildung 5.27 unten).

  $$p\_inform\_gross = \begin{bmatrix} 0.145 & 0.23 & 0.25 & 0.27 & 0.355 & 0.645 & 0.73 & 0.75 & 0.77 & 0.855 & 1 \end{bmatrix}$$

  Dieser Vektor wird mit dem Vierfachen der Zeitdauer des Stromanstiegs $t_{0 \to I_N}$ bis zum Erreichen des Nennstroms multipliziert, dies entspricht genau der Zeitdauer einer Großsignal-Sequenz.

$$t_{i=0 \to i=I_N} = L_s \cdot \frac{I_N}{\frac{2}{3} U_{ZK}}$$

$$t\_inform\_gross = 4 \cdot t_{i=0 \to i=I_N} \cdot p\_inform\_gross$$

[15] Der Grund, weshalb in der Abbildung 12 Interrupts eingezeichnet sind, liegt daran, dass der erste Interrupt nur bei der Aktivierung der Messequenz durch den Stromreglertask auftritt. In der Folge fällt I_{12} mit I_1 der nächsten Phase zusammen.

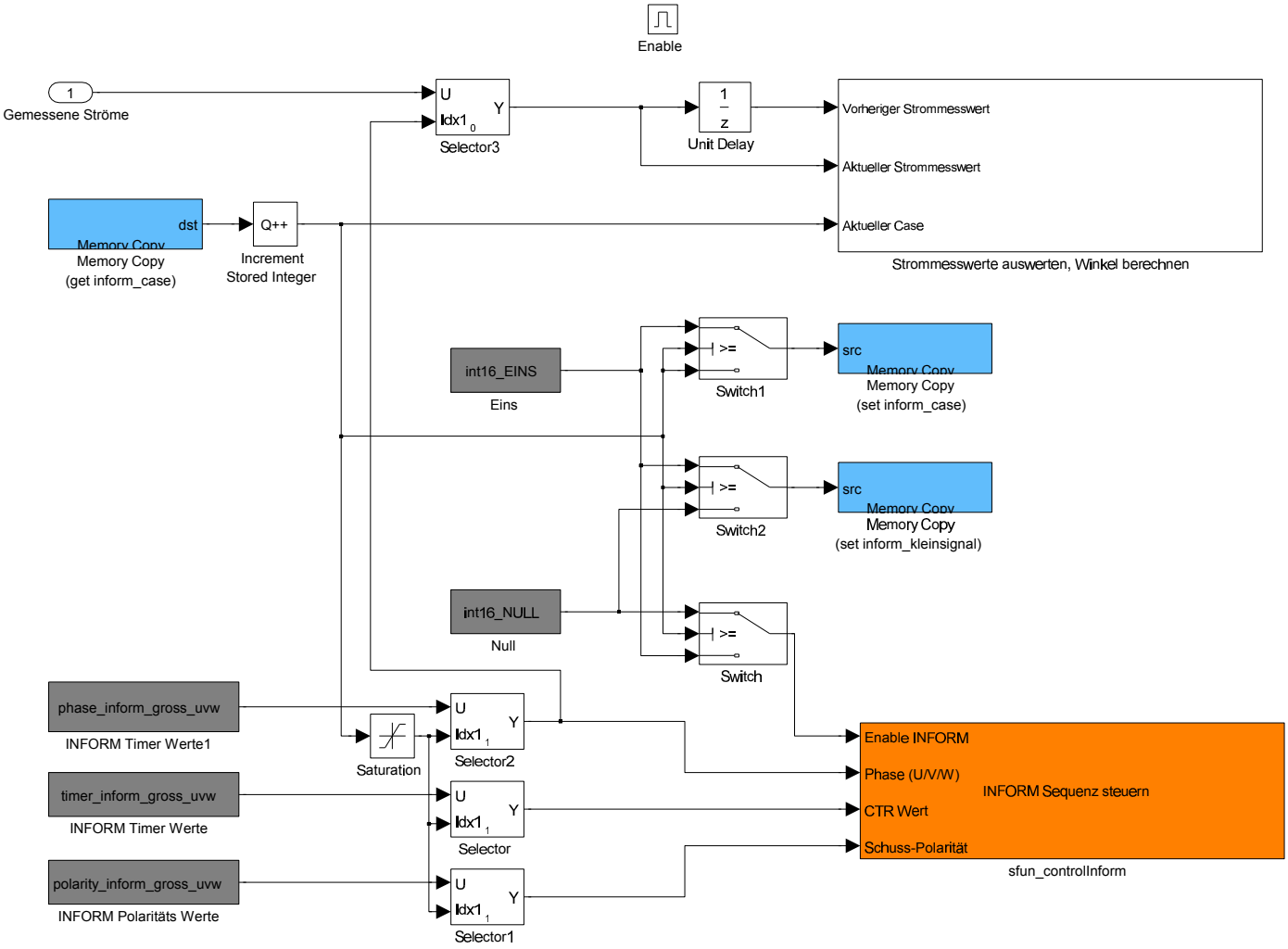

Abb. 5.29: Subsystem des Großsignal-INFORMS

Im eben erhaltenen Vektor stehen die absoluten Zeitpunkte der Interrupts, relevant jedoch sind die Periodendauern, welche die PWM zwischen den Interrupts einnehmen muss. Um diese Periodendauern zu gewinnen, wird der zeitdiskret differenzierte Zeitvektor mit der CPU-Taktfrequenz multipliziert. Da die PWM symmetrisch konfiguriert wurde, ist zu beachten, dass nur die Hälfte jener Werte zu nehmen ist.

$$timer\_inform\_gross = \frac{1}{2} \cdot \begin{bmatrix} 0 & t\_inform\_gross' \end{bmatrix} \cdot f_{CPU}$$

Schließlich werden diese Werte für den endgültigen Vektor $timer\_inform\_gross\_uvw$ entsprechend der drei Phasen U,V und W dreimal hintereinander angeordnet.

- $polarity\_inform\_gross\_uvw$ (Polaritäten der angelegten Spannungen): Im Falle des Großsignal-INFORMS wird folgender Polaritätsvektor konfiguriert.

$$polarity\_inform\_gross = \begin{bmatrix} +1 & +1 & +1 & -1 & -1 & -1 & -1 & -1 & +1 & +1 & +1 \end{bmatrix}$$

Der Vektor $polarity\_inform\_gross\_uvw$ entsteht wie zuvor aus der dreimal hintereinanderfolgenden Reihung des Einzelvektors.

Bei jedem Aufruf der Interrupt Service Routine wird ein Zähler inkrementiert, welcher für die Auswahl des Kontrollelements aus den Kontrollvektoren maßgeblich ist. Je nach aktueller Phase wird aus dem Vektor der gemessenen Ströme der richtige Wert ausgewählt und an die Auswertung weitergeleitet, welche durch das Subsystem in Abbildung 5.30 repräsentiert wird. Stimmt der Zähler mit einem der Cases überein, wird ein Sample-and-Hold-Glied aufgerufen, dessen Ergebnis bis zum nächsten Aufruf am Ausgang stehen bleibt. Im allerletzten Case wird die Berechnung des Winkels ausgelöst (siehe Abbildung 5.31). Dort ist Gleichung 5.22 als Simulink-Blockdiagramm abgesetzt. Das Ergebnis wird ins *fixdt(1,16,12)*-Format umgewandelt und in einer Variablen zwischengespeichert.

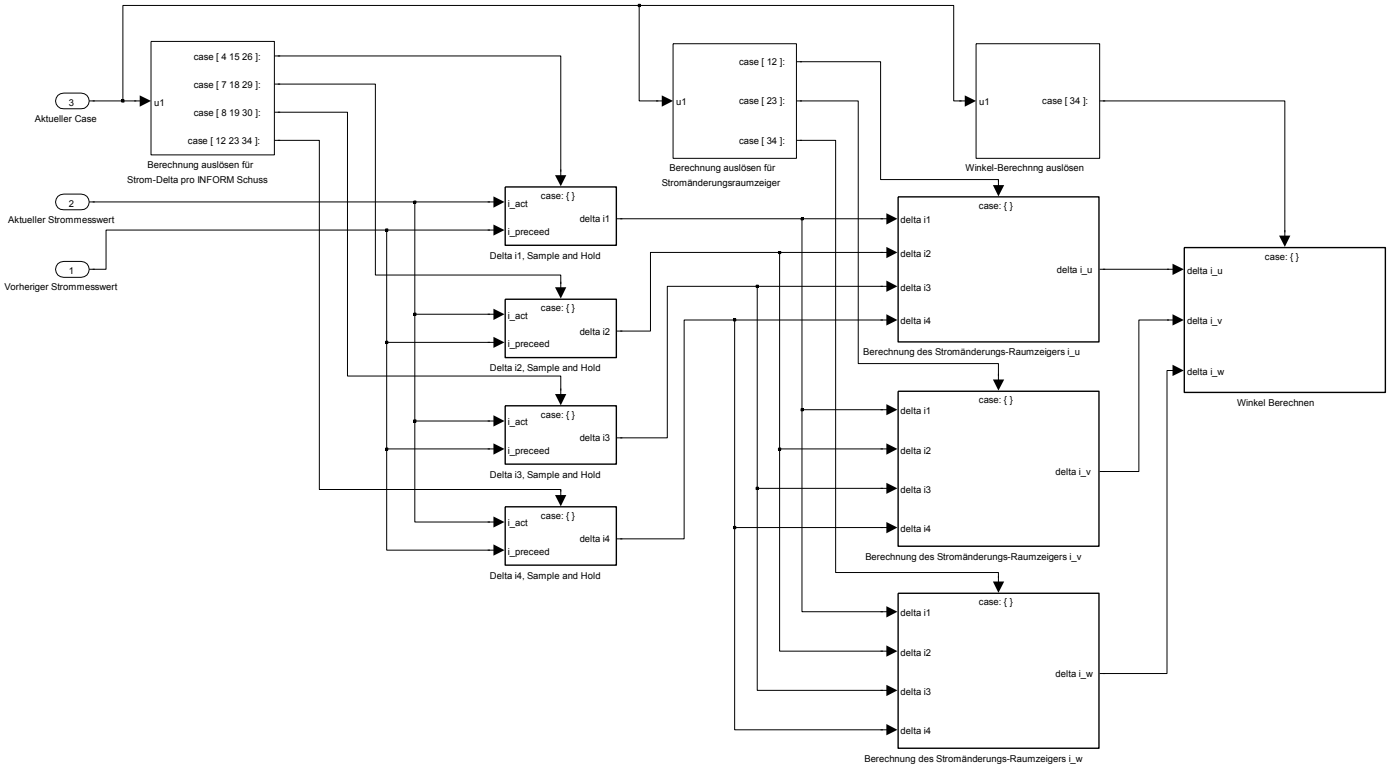

Abb. 5.30: Sample-and-Hold Mechanismus zur Bildung der Differenzterme

5.8.2. Kleinsignal-INFORM

Die Kleinsignal-Messung unterscheidet sich im Wesentlichen vom Großsignal insofern, als die Maschine nicht so weit in Sättigung getrieben wird. Dadurch verkürzen sich auch die Messsequenzen, da die Spannungsraumzeiger nicht so lange anliegen müssen, um einen gewissen Strom hervorzurufen (siehe Abbildung 5.32). Das Kleinsignal wird durch einen Vorschuss

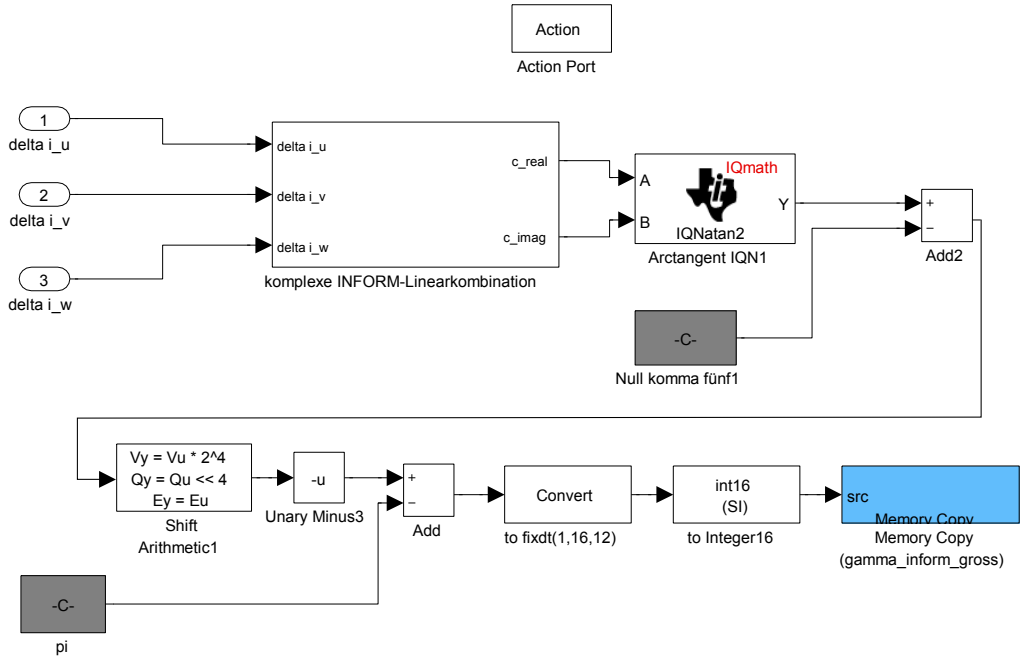

Abb. 5.31: Subsystem zur Berechnung des INFORM-Winkels

um den Arbeitspunkt zentriert und durch den Nachschuss wieder in den ursprünglichen Arbeitspunkt zurückgeführt. Während der beiden Hauptschüsse erfolgen die Messungen.

> Da sich für jede Stromanstiegsmessung in die drei Strangrichtungen (bzw. zwei im Falle von lediglich vier jeweils paarweise antiparallelen Schaltzuständen) eine andere Offsetgröße ergibt, d.h., die Messungen in voneinander abweichenden Arbeitspunkten durchgeführt werden, empfiehlt es sich, das angelegte Schussmuster zu zentrieren, damit die Stromanstiege im eigentlichen Arbeitspunkt gemessen werden und für alle Strangrichtungen identisch ist[16].

In Rieder (2005) werden neben der symmetrierten Schussfolge weitere INFORM-Sequenzen vorgestellt, welche hinsichtlich des Stromoberschwingungsgehaltes optimiert sind. Es wird hier jedoch auf die Standardmethode zurückgegriffen, da sie die am öftesten implementierte Methode des INFORM-Verfahrens darstellt.
Beim Kleinsignal werden die Ströme aus den Messpunkten 3, 4, 6 und 7 herangezogen.

$$\begin{aligned} \Delta i_{u-} &= i_4 - i_3 \\ \Delta i_{u+} &= i_7 - i_6 \end{aligned} \tag{5.23}$$

Für die anderen Stränge gilt dies analog, wodurch schließlich für jeden Strang ein Summenwert ermittelt wird:

[16]Siehe Rieder (2005): Optimierung der sensorlosen Regelung, S.51

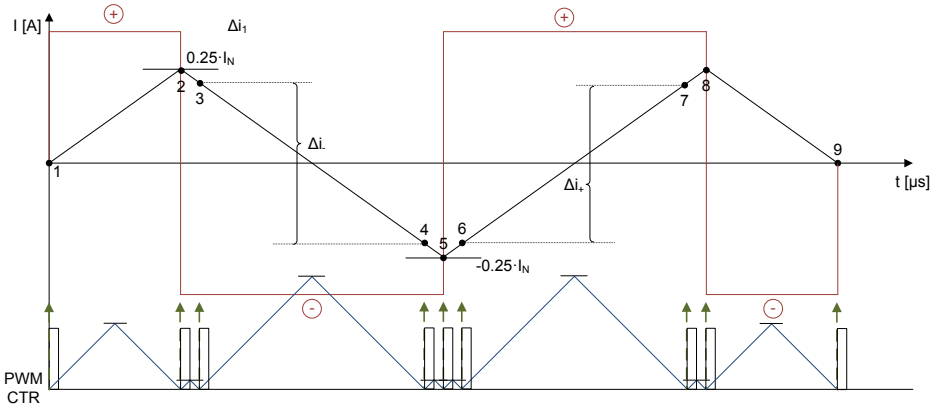

Abb. 5.32: Kleinsignal INFORM-Sequenz

$$\begin{aligned} \overline{\Delta i_u} &= \Delta i_{u+} + \Delta i_{u-} \\ \overline{\Delta i_v} &= \Delta i_{v+} + \Delta i_{v-} \\ \overline{\Delta i_w} &= \Delta i_{w+} + \Delta i_{w-} \end{aligned} \tag{5.24}$$

Die weitere Berechnung erfolgt wie beim Großsignal und wird deshalb nicht erneut erläutert. In Abbildung 5.33 ist der Signalverlauf der ermittelten Winkel bei einer Solldrehzahl von 2% der Nenndrehzahl dargestellt. In Blau erkennt man deutlich den INFORM-Winkel, da dieser 125 mal pro Sekunde aktualisiert wird und in der gewählten Zeitauflösung dadurch stufig erscheint. Das grüne Signal kennzeichnet den Resolverwinkel als Referenzsignal. Man sieht, dass die Änderungsrate der Winkelgeschwindigkeit $\frac{d\omega}{dt}$ des Rotors nicht konstant ist, was an der Beeinflussung durch die INFORM-Pulse liegt. Allerdings stimmt der geschätzte Winkel sehr gut mit dem Resolverwinkel überein. Eine Qualitätsanalyse des INFORM-Winkels erfolgt später im Abschnitt 6.2.

5.9. Beobachter

Um das Tiefdrehzahlmodell (INFORM-Verfahren) mit dem Hochdrehzahlmodell (EMK-Modell) zu einem robusten Regelungsmodell zu vereinen, wird in der Publikation des Instituts Schrödl/Hofer/Staffler (2006) vorgeschlagen, die Messinformation der beiden Modelle einem linearen Beobachter zuzuführen, welcher die mechanische Struktur des Motors modelliert.
Der Zustandsbeobachter wurde zuvor in zahlreichen Werken, darunter Schröder (2009) und Lunze (2010), detailliert behandelt. Deshalb wird hier nur kurz auf das Funktionsprinzip und die mathematische Beschreibung eingegangen: Ein Beobachter hat die Aufgabe, die Zustandsgrößen eines Systems dynamisch genau zu rekonstruieren[17]. In diesem Fall handelt

[17]Siehe Schröder (2009): Regelung von Antriebssystemen, S.143ff

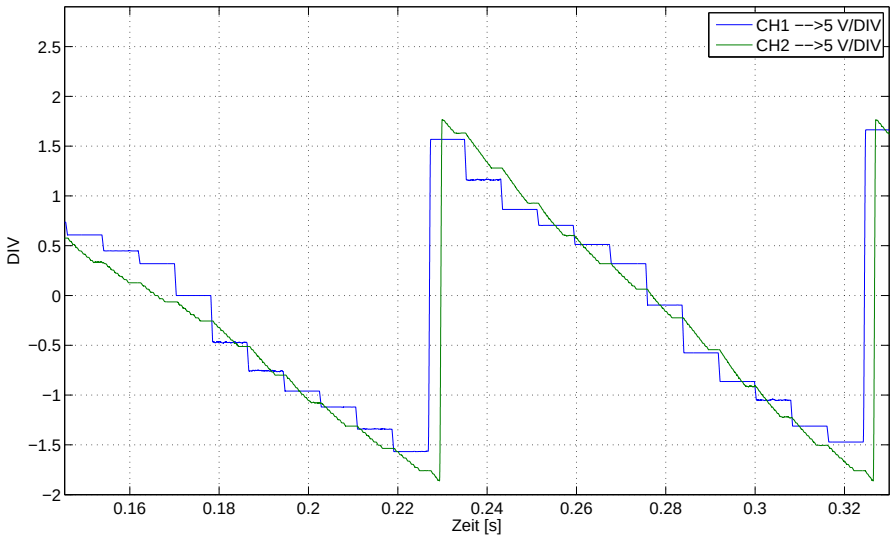

Abb. 5.33: INFORM- (blau) und Resolverwinkel (grün)

es sich um die Istdrehzahl und das Lastmoment, welche durch den Beobachteransatz nach Gleichung 5.25 zugänglich gemacht werden können.

$$\begin{bmatrix} \omega(k+1) \\ \gamma(k+1) \\ m_L(k+1) \end{bmatrix} = \begin{bmatrix} 1 & 0 & \frac{\tau_A}{\tau_m} \\ \tau_A & 1 & \frac{\tau_A^2}{2 \cdot \tau_m} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \omega(k) \\ \gamma(k) \\ m_L(k) \end{bmatrix} + \begin{bmatrix} \frac{\tau_A}{\tau_m} \\ \frac{\tau_A^2}{2 \cdot \tau_m} \\ 0 \end{bmatrix} \cdot i_{q,ist}(k) \tag{5.25}$$

Bei der PSM stellt die q-Komponente des Statorstromes die drehmomentbildende Stromkomponente dar, was im Prädiktionsmodell aus Gleichung 5.26 berücksichtigt wird:

$$\begin{bmatrix} \omega^*(k+1) \\ \gamma^*(k+1) \\ m_L^*(k+1) \end{bmatrix} = \begin{bmatrix} 1 & 0 & \frac{\tau_A}{\tau_m} \\ \tau_A & 1 & \frac{\tau_A^2}{2 \cdot \tau_m} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \hat{\omega}(k) \\ \hat{\gamma}(k) \\ \hat{m}_L(k) \end{bmatrix} + \begin{bmatrix} \frac{\tau_A}{\tau_m} \\ \frac{\tau_A^2}{2 \cdot \tau_m} \\ 0 \end{bmatrix} \cdot i_{q,ist}(k) \tag{5.26}$$

Auf die Schätzung des Lastmoments und den Einfluss des additiven Terms durch das innere Moment wird aus Gründen der Vereinfachung verzichtet. Somit folgt die reduzierte Gleichung 5.27:

$$\begin{bmatrix} \omega^*(k+1) \\ \gamma^*(k+1) \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \tau_A & 1 \end{bmatrix} \cdot \begin{bmatrix} \hat{\omega}(k) \\ \hat{\gamma}(k) \end{bmatrix} \tag{5.27}$$

Das Korrekturmodell mit den beiden Kalmanfaktoren, welches für die Kombination aus INFORM-Verfahren und EMK-Modell verwendet wird, hat die Form

$$\begin{bmatrix} \hat{\omega}(k) \\ \hat{\gamma}(k) \end{bmatrix} = \begin{bmatrix} \omega^*(k) \\ \gamma^*(k) \end{bmatrix} + \begin{bmatrix} k_{kalman,1} \\ k_{kalman,2} \end{bmatrix} \cdot \left(\gamma_{inf}(k) - \begin{bmatrix} 0 & 2 \end{bmatrix} \cdot \begin{bmatrix} \omega^*(k) \\ \gamma^*(k) \end{bmatrix} \right) \tag{5.28}$$

Die Realisierung von Gleichung 5.28 wurde in Simulink nach Abbildung 5.34 vorgenommen. Zur Initialisierung des Winkels wird der durch die Großsignalmessung ermittelte, in einer Variable zwischengespeicherte Wert herangezogen. Hierdurch wird dem diskreten Verzögerungsglied durch einen Reset der Wert des Großsignalwinkels zugewiesen. Dies geschieht immer dann, wenn der Zustand der Statusvariablen *inform_kleinsignal* seinen Wert ändert, was im Normalfall beim erstmaligen Antreiben der Maschine nach dem Einschalten oder durch einen Benutzerbefehl erfolgt. Am Eingang des Systems liegt je nach aktueller Drehzahl der Maschine der EMK-Winkel oder der INFORM-Kleinsignalwinkel an. Die im Umschaltbereich der beiden Messverfahren unvermeidlichen Winkelunstetigkeiten zwischen γ_{INFORM} und γ_{EMK} werden durch die Tiefpasscharakteristik des Kalmanfilters ausgeglichen. Die Größe der beiden Kalmanfaktoren entscheidet hierbei über die Dynamik des Filters. Gewählt wurden folgende Werte:

Filterkonstante	Wert
$k_{kalman,1}$	0.05
$k_{kalman,2}$	0.01

Tab. 5.6: Gewählte Parameter des Kalmanfilters

Im Betrieb wurden die in Abbildung 5.35 gezeigten Signalverläufe gemessen. Die niedrige Drehzahl von $w_{soll} = 0.02$ liegt unterhalb des Umschaltpunktes zum EMK-Modell, weshalb dem Filter laufend der Kleinsignalwinkel (blau) zuführt wird. Das Ausgangssignal (rot) folgt nun dem Winkel des Resolvers (grün) deutlich besser. Ein statistischer Vergleich des Winkelfehlers zwischen Beobachter- und Resolverwinkel erfolgt später in Abschnitt 6.2 durch die Analyse einer zehnsekündigen Messreihe.

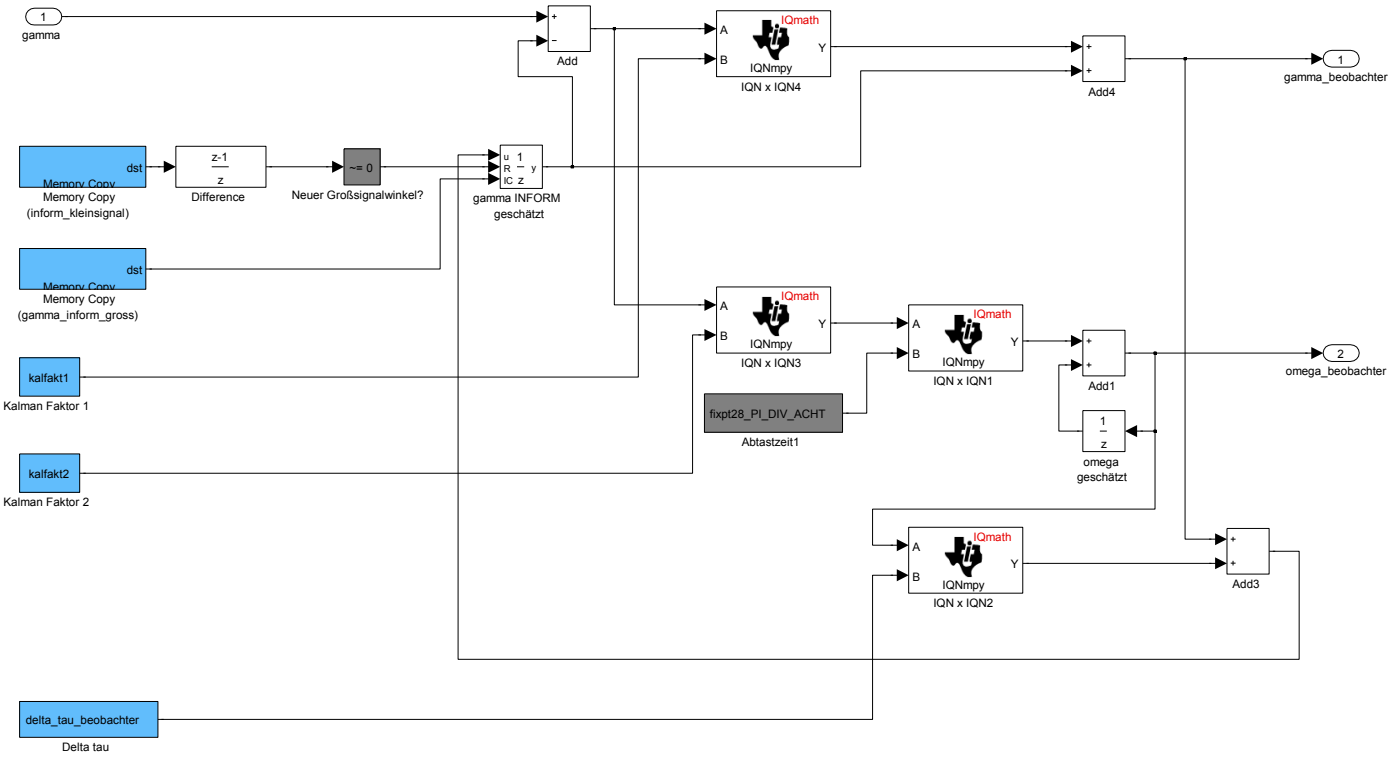

Abb. 5.34: Subsystem des Drehzahlbeobachters

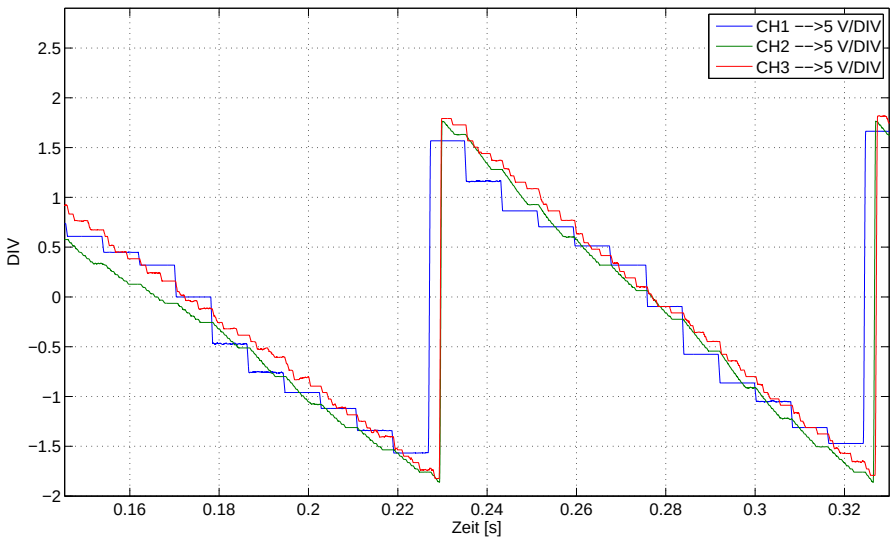

Abb. 5.35: INFORM- (blau) Resolver- (grün) und Beobachterwinkel (rot)

5.10. Betriebsmoduswechsel

Um die implementierten Funktionssysteme im Betrieb zu vereinen, wurde der Block zur Betriebsmoduswechsel entwickelt. Je nach Benutzerverhalten und Drehzahl der Maschine werden die folgenden Betriebszustände unten stehender Auflistung unterschieden. Die Realisierung des Blockes in Simulink ist von der Logik gesehen leicht zu durchschauen, erweckt aber auf den ersten Blick durch viele Verzweigungen einen unübersichtlichen Anschein. Eine Abbildung wird aus diesem Grunde nicht für sinnvoll erachtet. Die Aufgaben der Umschaltung des Betriebsmodus liegen in den folgenden Punkten:

- Entsprechend der aktuellen Geschwindigkeit wird zwischen den Winkeln des EMK-Modells und des INFORM-Verfahrens unter Berücksichtigung einer Hysterese von 1% der Nenndrehzahl umgeschaltet. Dieser Winkel dient als Eingangssignal für den Drehzahlbeobachter im sensorlosen Betriebsmodus.
- Aktiviert der Benutzer den sensorlosen Betriebsmodus, wird der Winkel des Drehzahlbeobachters für die Raumzeigertransformationen verwendet. Andernfalls wird der Resolverwinkel herangezogen.
- Die eben genannten beiden Punkte stehen für den geregelten Betrieb im Normalzustand. Weiters wurden Funktionen zu Entwicklungszwecken eingebaut, welche die Vorgabe eines bestimmten Winkelarguments in Verbindung mit einer frei einstellbaren Raumzeigeramplitude des Statorspannungsraumzeigers oder in weiterer Folge des Stromraumzeigers ermöglichen.

6. Vergleich der Codegenerierung mit konventioneller Programmierung

Im Kapitel 5 wurde die Machbarkeit der Regelung mittels MATLAB/Simulink gezeigt. Interessant ist nun ein kurzer Vergleich der Ergebnisse dieser Herangehensweise mit denen der konventionellen C-Programmierung.

6.1. Lesbarkeit und Nachvollziehbarkeit des Quellcodes

Zuerst wird der generierte C-Quellcode auf Lesbarkeit und dann die Ausführungsgeschwindigkeit geprüft.
Betrachtet man ein genertiertes CCS-Projekt, so fallen rein von der Dateistruktur keine Unterschiede auf. Die Quellcodedateien selbst hängen jedoch stark von der Fähigkeit des Benutzers ab, den Mathworks Real Time Workshop richtig zu bedienen. Zwar ist die Benennung von Subsystemen, Signalen und Variablen, wie in Kapitel 4.4 erklärt, nicht zwingend notwendig, beeinflusst aber entscheidend die Lesbarkeit bei größeren Projekten. Deshalb wird dem Einsteiger geraten, die Basisfunktionalitäten zuerst mit kleinen Projekten zu schaffen und sich somit mit den Mechanismen der Codegenerierung vertraut zu machen, anstatt unstruktiert nach der Wasserfallmethode vorzugehen. Wird eine konsequente Benennung durchgezogen, kann eine gute Lesbarkeit und Nachvollziehbarkeit gesichert werden. Andernfalls wird der Quellcode nach voreingestellten Regeln erstellt, die oft auf einer fortlaufenden Nummerierung der Signale basieren. Jene Benutzer, welche die nachträgliche Inspizierung des Quellcodes in Betracht ziehen, werden anhand des folgenden Beispiels die Sinnhaftigkeit der benutzerdefinierten Signal- und Funktionsblockbenennung erkennen. Abbildung 6.1 stellt ein Demosystem ohne Sinnhaftigkeit dar, bei dem es rein darum geht, dessen Struktur in C-Code umzusetzen.

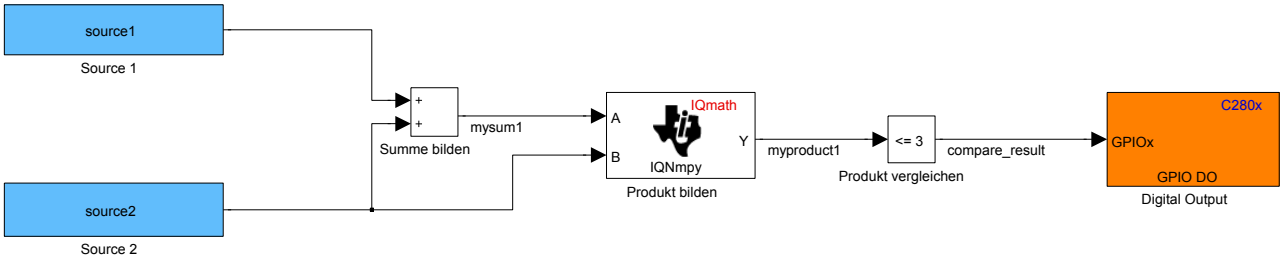

Abb. 6.1: System zur Demonstration der Zweckmäßigkeit von Signalbenennungen

Wird auf eine konsequente Benennung der Signale Wert geleget, generiert der Real Time Workshop aus der Blockstruktur folgenden Quellcode:

```
...
   /* Sum: '<Root>/Summe bilden' incorporates:
    *  Constant: '<Root>/Source1'     */
  mysum1 = source1 + source2;

  /* C28x IQmath Library (stiiqmath_iqmpy) - '<Root>/Produkt bilden' */
  {
    myproduct1 = _IQ28mpy (mysum1, source2);
  }

  /* RelationalOperator: '<S1>/Compare' incorporates:
   *  Constant: '<S1>/Constant'  */
  compare_result = (myproduct1 <= ((int32_T)805306368L));

  /* S-Function Block: <Root>/Digital Output (c280xgpio_do) */
  {
    GpioDataRegs.GPASET.bit.GPIO0 = (compare_result != 0);
    GpioDataRegs.GPACLEAR.bit.GPIO0 = !(compare_result != 0);
  }
...
```

include/clean_naming.c

Verzichtet man allerdings auf die Benennung, resultiert ein längerer, unnötig kompliziert erscheinender Code:

```
...
  /* local block i/o variables */
  int32_T rtb_Add;
  int32_T rtb_IQN1xIQN2;
  int32_T rtb_Constant1;
  int32_T rtb_Constant2;

  rtb_Constant1 = source1;
  rtb_Constant2 = source2;

  /* Sum: '<Root>/Add' incorporates:
   *  Constant: '<Root>/Constant 1'
   *  Constant: '<Root>/Constant 2'     */
  rtb_Add = rtb_Constant1 + rtb_Constant2;

  /* C28x IQmath Library (stiiqmath_iqmpy) - '<Root>/IQN1 x IQN2' */
  {
    rtb_IQN1xIQN2 = _IQ28mpy (rtb_Add, rtb_Constant2);
  }

  /* RelationalOperator: '<S1>/Compare' incorporates:
   *  Constant: '<S1>/Constant'     */
  readability_B.Compare = (rtb_IQN1xIQN2 <= ((int32_T)805306368L));

  /* S-Function Block: <Root>/Digital Output (c280xgpio_do) */
  {
    GpioDataRegs.GPASET.bit.GPIO0 = (readability_B.Compare != 0);
    GpioDataRegs.GPACLEAR.bit.GPIO0 = !(readability_B.Compare != 0);
  }
...
```

include/dirty_naming.c

Dieses Minimalbeispiel mag zwar aufgrund der geringen Anzahl von involvierten Blöcken noch keine Probleme beim Nachvollziehen bereiten, zeigt jedoch bereits die potentiellen Probleme auf. Bei komplexen Strukturen wie dem Gesamtmodell der Regelung ist schließlich die Verständlichkeit ohne Berücksichtigung der Namensgebung nicht mehr gegeben. Es ist allerdings berechtigt, die Haltung vieler Softwareingenieure zu unterstützen, welche nicht unbedingt die Lesbarkeit eines generierten Codes fordern. Schließlich handelt es sich hier um einen der großen Vorteile der modellbasierten Programmierung, sich nicht mit dem Quellcode im Speziellen befassen zu müssen. Befindet sich man jedoch mitten in der Entwicklungsphase, ist ein kurzer Blick auf den erstellten Quellcode oft unumgänglich.

6.2. Effizienz und Ausführungsgeschwindigkeit

Beruht die Bewertung der Lesbarkeit und Nachvollziehbarkeit des Codes auf noch auf subjektiver Empfindung, kann die Ausführungsgeschwindigkeit des Codes objektiv gemessen werden. Hierbei interessiert vor allem der zeitkritische Kurztask, da dieser am öftesten ausgeführt wird. Die Ermittlung der Performance des Codes zur Laufzeit kann über zwei verschiedene Methoden erfolgen: Zum einen ist es möglich, zu Beginn und am Ende des Kurztasks einen digitalen Ausgang zu setzen, um die verstrichene Zeitspanne mit dem Oszilloskop zu messen. Zum anderen kann ein *Profiling*-Tool, vergleichbar einem Benchmark, eingesetzt werden, um die selben Ergebnisse in einem Report-Schema zu erhalten. Diese Methode wird später in diesem Absatz noch näher ausgeführt. Während der laufenden Entwicklungsphase wird allerdings erstere Methode nahegelegt, da hierdurch ein schnelles und unkompliziertes Messen mit dem Oszilloskop ohne zusätzliche Änderung des Programmes möglich ist. Solange das Modell erweitert wird, ist eine sporadische Überprüfung der noch zur Abarbeitung des Kurztasks verfügbaren Zeit unerlässlich, da im Falle des Auftretens von Timing-Problemen die Ursache schnell gefunden werden kann. Während dieser Diplomarbeit wurde zuerst angedacht, die PWM-Frequenz zusammen mit der Kurztask-Frequenz auf 20 kHz festzusetzen. Dieses Vorhaben wurde allerdings aufgrund der langen gemessenen Ausführungszeit des Kurztasks von knapp 75μs verworfen und die Frequenz auf 10 kHz reduziert. Somit stehen pro Arbeitsschritt 100μs zur Verfügung.
Neben der direkten Messung durch digitale Ausgänge liefert das *Profiling*-Tool dennoch interessante Ergebnisse. Hierzu muss zuerst das Konfigurationsmenü des Real Time Workshop durch Drücken der Tastenkombination *CTRL+E* geöffnet und dann die Option „Profile Real Time Execution“ aktiviert werden. Gewählt wird weiters „Profile by Tasks“, um die Dauer der Interruptroutinen zu bestimmen. Anschließend muss das Projekt erneut in den DSP geflasht werden. Ist dies erfolgt, wird das Programm mit der Taste „F5“ gestartet und nach wenigen Sekunden wieder beendet. Durch Eingabe des Befehls

```
>> profile(IDE_Obj, 'execution', 'report')
```

in der MATLAB-Kommandozeile kann die Erstellung des HTML-Reports ausgelöst werden.

Der Report selbst ist zur Vollständigkeit dem Anhang A.3 beigefügt, die dazugehörige Grafik ist jedoch in Abbildung 6.2 zu sehen, wobei *Task 2* für den Kurztask und *Task 3* für den Langtask stehen:

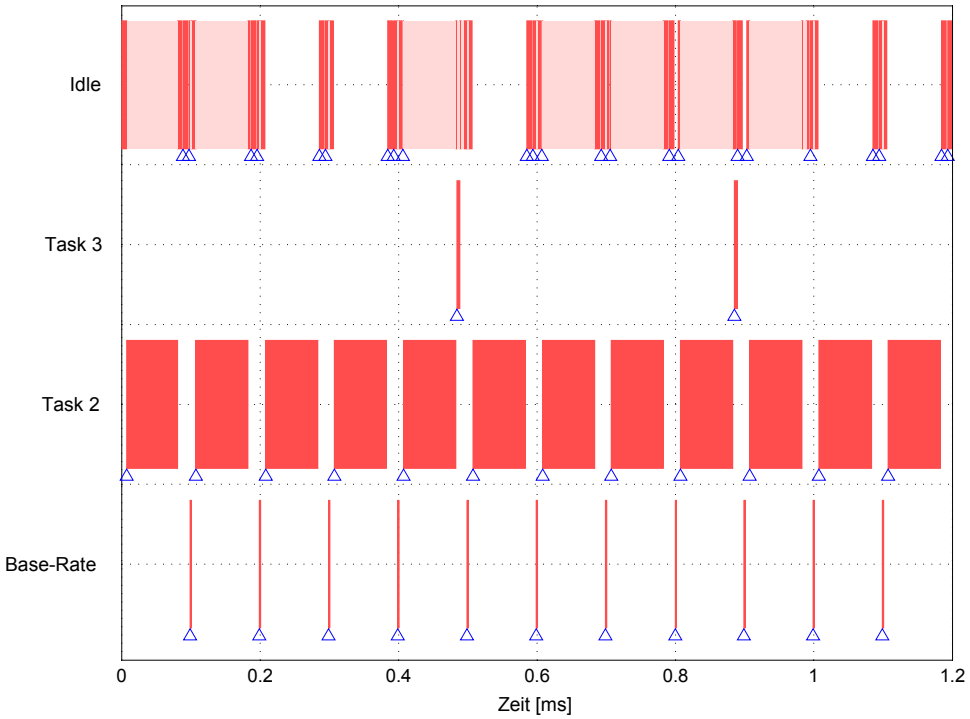

Abb. 6.2: Ergebnisse des Profiling-Tools über einen Zeitraum von 1.2ms

Der HTML-Report liefert folgende Ergebnisse für die Laufzeiten des Programms (vgl. Anhang A.3):

Task	⌀ Ausführungszeit [μs]	⌀ Periodendauer [μs]
Basis-Samplingrate	1.894	99.990
Kurztask	75.236	99.992
Langtask	4.274	399.984
Idle-Task	8.695	52.187

Tab. 6.1: Ergebnisse des HTML-Reports der Programmlaufzeiten

Die Messung der Dauer des Kurztasks bestätigt die Profiling-Angaben von 75.236μs. Da in dieser Arbeit eigentlich lediglich die Basisfunktionalität einer Regelung implementiert wurde, erscheint diese Zeitspanne als relativ lang im Vergleich zu ähnlichen Projekten, die mittels konventioneller Programmierung geschaffen wurden. Somit scheint die Codegenerierung langsamere Programme zu liefern. Einen stichfesten Beweis gibt es allerdings aufgrund eines nicht vorhandenen Referenzprojekts für diese Vermutung nicht.
Weiters ergibt der HTML-Report, dass der Langtask lediglich 4.274μs benötigt, was ange-

sichts seines einfachen Aufbaus nicht verwunderlich ist. Der Idle-Task, welcher die Kommunikation und das Softauge beinhaltet, dauert mit 8.695μs knapp doppelt so lange und wird im Schnitt alle 50μs aufgerufen. Bei Nichtbeanspruchung der CPU kann es allerdings sein, dass mehrere Aufrufe kurz aufeinander folgen, wie Abbildung 6.2 bestätigt.

6.3. Regelungsverhalten, Qualität des INFORM-Winkels

Ob sich die Regelung im Vergleich zu einem Standard C-Projekt unterscheidet, kann nicht objektiv ermittelt werden, da es auf dem Institut nur Projekte gibt, die allenfalls gute Ähnlichkeit mit dieser Arbeit aufweisen. Ein direkter Vergleich könnte nur angestellt werden, indem das Simulink-Projekt noch einmal exakt in C-Code abgebildet würde. Dies wäre aus Zeitgründen nicht möglich gewesen. Trotzdem ist ein Vergleich erhaltener Messdaten mit den Ergebnissen anderer Arbeiten zulässig. In Staffler (2005) wurden die Varianten einzelner INFORM-Messungen miteinander verglichen. Abbildung 6.3 zeigt die Darstellung des INFORM-Fehlerwinkels aus Staffler bei konstanter Drehzahl und ungefilterter Strommessung. Dies entspricht der Beschreibung nach der selben Messumgebung, die auch in dieser Arbeit geschaffen wurde: Ebenfalls wurde der INFORM-Winkel über eine längere Zeit (hier: 10 Sekunden) gemessen, während die Maschine langsam unbelastet rotierte. Der Verteilung des Fehlers wurde eine Normalverteilung mit den selben statistischen Parametern (Mittelwert -1.89°, Standardabweichung 7.17°) hinterlegt und aufgrund der guten Ähnlichkeit der beiden Kurven behauptet, dass der Fehlerwinkel ebenfalls normalverteilt sei.
Die Messung in dieser Arbeit (Abbildung 6.4) lässt ebenfalls eine Normalverteilung des Fehlers vermuten. Der Mittelwert des Fehlers von -0.4827° liegt sogar unter dem Wert von -1.89°, worauf es aber ankommt, ist die Standardabweichung, welche zu 14.44° ermittelt wurde und somit doppelt so hoch ist wie bei Staffler. Dies lässt auf keine gute INFORM-Tauglichkeit des Motors schließen. Dieser Tatsache wird jedoch nicht übermäßig viel Aufmerksamkeit geschenkt, da in dieser Diplomarbeit vordergründig die Machbarkeit des INFORM-Verfahrens mit MATLAB/Simulink interessiert, was nachgewiesen wurde.

6.4. Persönliche Erfahrungen während der Entwicklungsphase

Die Entwicklungsphase verlief zeitweise nicht so flüssig wie gewünscht, da noch nicht auf Vorwissen im Bereich der Codegenerierung zurückgegriffen werden konnte. Auch die notwendige Einhaltung der Insitutskonventionen stellte eine große Herausforderung dar. So musste das Kommunikationsprotokoll modifiziert werden, um die bisher übliche Zahlendarstellung mit dem in MATLAB/Simulink üblichen Format zu vereinen. Dennoch wurde der rote Faden niemals aus den Augen verloren, da die Betreuer der Diplomarbeit mit neuen Denkanstößen zielführende Ideen erweckten oder das umfangreiche Manual in Kombination mit den Demobeispielen unter Simulink helfend beistanden. Zu Beginn wurde die Wichtigkeit klein erscheinender Bestandteile unterschätzt. So kann eine falsche Datentypumwandlung durch

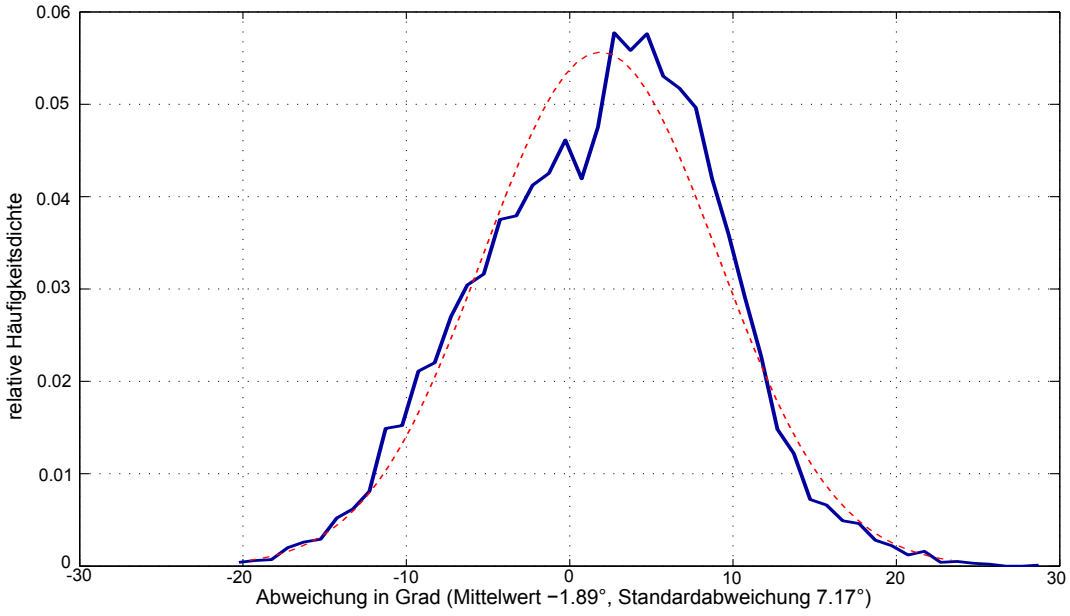

Abb. 6.3: Normalverteilung des INFORM-Fehlers, aus Staffler (2005)

eine falsch gewählte Option das komplette Modell beeinträchtigen; die anschließende Fehlersuche gestaltet sich dann sehr zeitaufwändig. Deshalb wird dem Einsteiger geraten, sich zuerst mit den Grundzusammenhängen aus Kapitel 4 vertraut zu machen.
Als weiteres Hindernis wurde empfunden, dass keine Möglichkeit gefunden wurde, die Codegenerierung ohne eine aufrechte Verbindung zum DSP durchzuführen. Obwohl die Eigenschaften der Zielplattform in MATLAB durch den „Target Preferences"-Block und ebenfalls im Texas Instruments Code Composer Studio definiert sind, ist eine aufrechte Verbindung während des Arbeitens ohne erkennbaren Grund zwingend notwendig. Somit kann das Modell zwar unabhängig von der Hardware geändert, aber die Codegeneriung nicht durchgeführt werden. In den Fällen, in denen lediglich eine fehlerlose Codeerstellung verifiziert werden soll, ohne das Programm unmittelbar auf den DSP zu flashen, ist diese unumgängliche Einschränkung hinderlich. So ist beispielsweise ein ortsunabhängiges Arbeiten nicht möglich.
Der „Target Preferences"-Block ist für den TI2808-DSP mit dem eZdsp™F2808 Entwicklerboard vorbelegt. Auf diesem Entwicklerboard ist ein 20 MHz-Quartz verbaut, welcher über die Taktmultiplikatoren eine CPU-Frequenz von 100 MHz liefert. Auf der institutseigenen Prozessorsteckplatine wurde jedoch ein 16 MHz-Quartz gewählt, was auf eine CPU-Frequenz von 96 Mhz führt. Bei der Implementierung von zeitkritischen Funktionen, wie dem

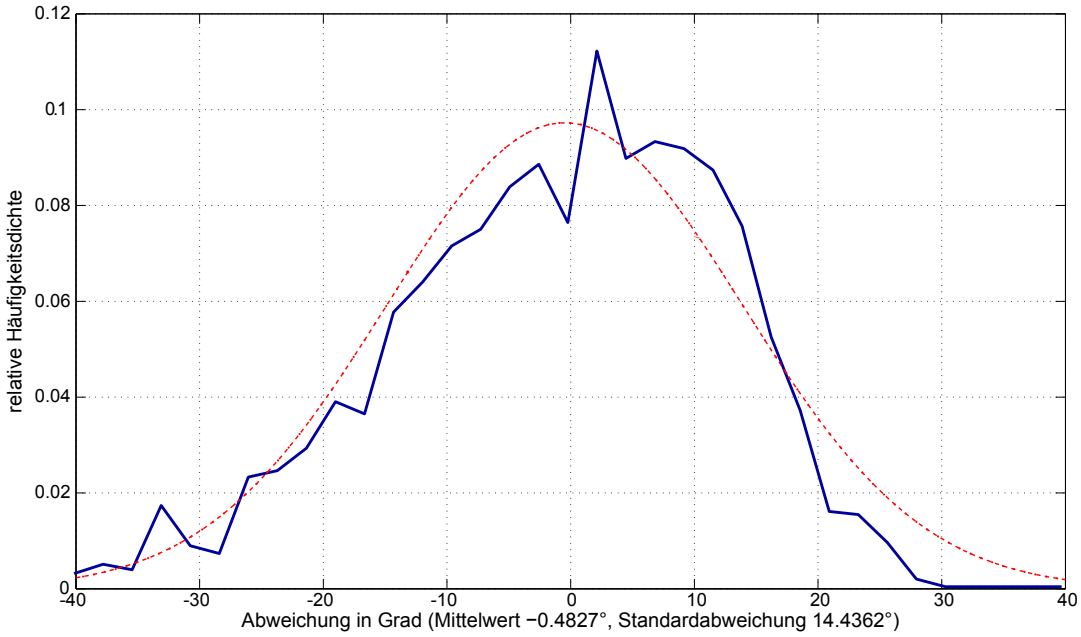

Abb. 6.4: Normalverteilung des INFORM-Fehlers, Messung

Kommunikationsprotokoll, konnte lange keine Lösung für eine abweichende reale Baudrate gefunden werden, bis schließlich festgestellt wurde, dass der PLL-Baustein der CPU nicht korrekt initialisiert wurde. Trotz der Einstellmöglichkeiten für die Taktmultiplikatoren und die CPU-Frequenz im Menü des „Target Preferences“-Block sollte die Initialisierung durch Begutachten der Quellcodedatei `DSP280x_Examples.h` verifiziert werden.

Trotz der Einstellmöglichkeiten für die Taktmultiplikatoren und die CPU-Frequenz im Menü des „Target Preferences“-Blocks sollte die Initialisierung durch Begutachten der Quellcodedatei `DSP280x_Examples.h` verifiziert werden.

Ein weiterer Punkt, welcher folgenden Benutzern nicht vorenthalten sein sollte, betrifft die Einbindung von C-Dateien mit anschließender Codegenerierung. Durch das Legacy Tool wird für jede C-Datei eine MATLAB *mex*-Datei erstellt (siehe Abschnitt 4.2). Das Vorhandensein dieser Dateien ist u.a. für die Generierung des Codes nötig und sie müssen im aktuellen Arbeitsverzeichnis oder dem MATLAB-Standardpfad abgelegt sein. Obwohl dies beim Auftreten eines bestimmten Fehlers der Fall war, wurde fälschlicherweise das Fehlen der *mex*-Datei beanstandet. Die Lösung des Problems liegt in der erneuten Kompilierung der C-Datei durch das Legacy-Tool.

7. Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde ein alternativer Ansatz zur Programmierung von digitalen Signalprozessoren für die Regelung von permanenterregten Synchronmaschinen vorgestellt. Diese Variante findet durch die Entwicklung ebenso moderner wie komplexer Toolchains - wie MATLAB/Simulink/Real Time Workshop - immer mehr Gebrauch in hochspezialisierten Industriezweigen wie z.B. der Automobilindustrie. Die Symbiose aus Softwaredesign und Endfunktion lässt die Lücke zwischen den verschiedenen beteiligten Parteien kleiner werden: Durch die Verschaltung der Blockstrukturen werden Subsysteme sowohl logisch als auch funktional und somit durch den Codegenerator programmtechnsich verknüpft. Der Programmierer kann sich im Weiteren auf Details konzentrieren, die auf der Designebene nicht relevant sind. Sind einmal die Voraussetzungen der Codegenerierung für eine Systemarchitektur geschaffen, so ist es kein weiter Schritt, dies auch auf anderen Systemen anzuwenden. Die Programmbibliotheken der Entwicklungstools lassen ein leichtes modulares Austauschen der Funktionsblöcke zu, die trotz unterschiedlicher Hardware keine Unterschiede in deren Konfiguration aufweisen. Gerade dieser Aspekt verdeutlicht die Mächtigkeit der modellgetriebenen Programmierung: Da moderne Systeme zur Abschätzung des Betriebsverhaltens meist ohnehin zuerst in Simulationsprogrammen diverse Tests durchlaufen, kann das Simulationsmodell ohne große Anpassungen direkt zur Codegenerierung eingesetzt werden. Das bringt zum einen den Vorteil, dass Funktionen nicht doppelt in unterschiedlichen Entwicklerwerkzeugen erstellt werden müssen und zum anderen eine immense Zeitersparnis. Moderne Systemdesigner lassen sich nicht von der Tatsache beirren, dass mitunter eine gewisse Einarbeitungszeit erforderlich sein kann, da diese später durch eine Reihe von Vorteilen kompensiert wird.
Diese Diplomarbeit dient dem Institut für Elektrische Energiesysteme und Elektrische Antriebe an der TU Wien dazu, das Basiswissen der Codegenerierung mit MATLAB/Simulink bereitzustellen. Die bereits in zahlreichen Varianten umgesetzte sensorlose Regelung einer permanenterregten Synchronmaschine in Kombination mit dem INFORM-Verfahren dient als Beispielprojekt, da sich die Institutsmitarbeiter mit den verwendeten Methoden identifizieren können und somit eine größtmögliche Nachvollziehbarkeit gegeben ist.
Kapitel 1 definiert die genaue Aufgabenstellung, im **Kapitel 2** folgt eine kurze theoretische Einführung in das Gebiet der permanenterregten Synchronmaschine und deren mathematische Beschreibung. Das **Kapitel 3** beschreibt die nötigen Mittel der Hard- und Software, welche für die Umsetzung der gestellten Fragestellung erforderlich sind. Hierbei wird einerseits auf die institutseigenen Komponenten, die Haupt- und Prozessorplatine sowie das Kommunikationsinterface eingegangen, andererseits das entscheidende Entwicklerwerkzeug MATLAB/Simulink/Real Time Workshop in dessen relevanten Punkten erklärt. **Kapitel 4** schildert die grundlegenden Vorgänge der Codegenerierung, stellt die Möglichkeit der Einbindung eines vorhandenen C-Quellcodes in das Simulationsmodell dar und hebt eine Zahl wichtiger Funktionsblöcke der mitgelieferten Programmbibliotheken hervor. Des Weiteren ist

der Einsatz des Softauges unter MATLAB/Simulink dokumentiert.Der Hauptteil, **Kapitel 5**, präsentiert Schritt für Schritt die Implementierung der Regelung, ausgehend von der Inbetriebnahme der eingesetzten Hardware, über die Einbindung des Stromreglers hin zu einem kombiniertem Betrieb mit EMK-Modell und INFORM-Verfahren. Besonders wichtige, nicht zu übersehende Punkte, sind in roter Schrift hervorgehoben. Zum Schluss ist in **Kapitel 6** ein Vergleich des erhaltenen Programmcodes, der Messergebnisse und der Programmlaufzeiten mit Projekten der konventionellen C-Programmierung dargestellt. Ebenso wird bündig über die gewonnenen Erfahrungen während der Entwicklungsphase berichtet.
Im Laufe der Diplomarbeit wurde die Zielsetzung nie aus den Augen verloren; auf die Realisierung interessanter Themen abseits des Lösungsweges konnte jedoch aus Zeitgründen nicht Rücksicht genommen werden. So wäre es interessant gewesen, wieviel Aufwand es wirklich bedarf, das vorhandene Projekt auf eine andere Zielplattform, beispielsweise neuere Prozessoren von Texas Instruments, zu portieren. Ebenso erscheint es nach dem Schaffen der dem Institut benötigten Basisfunktionalitäten als sinnvoll, die Funktionsblöcke in verbesserter Form in einer eigenen Simulink-Bibliothek zu integrieren. So könnte das Kommunikationsinterface, um nur ein Beispiel zu nennen, so angepasst werden, dass dessen Konfiguration über ein Dialogfenster erfolgt, ähnlich den von Texas Instruments zur Verfügung gestellten Subsystemen.
Im Zuge dieser Arbeit konnte die Erkenntnis gewonnen werden, dass generierter Code nicht die selbe Ausführungsgeschwindigkeit erreicht, wie dies bei herkömmlich programmierten C-Code der Fall ist. Die Vielzahl der Anpassungsmöglichkeiten des Real Time Workshop wurden jedoch noch nicht gänzlich ausgelotet, was in Zukunft weitere Arbeiten in diesem Gebiet zur Gewinnung weiterer Erkenntnisse als sinnvoll erscheinen lässt. Somit wurde mit dieser Diplomarbeit ein Anstoß hin zur vertieften Arbeit in einem neuen Themengebiet gegeben.

A. Anhang

A.1. Code-Listings: Skripten zur automatisierten Erstellung von S-Funktionen

MATLAB-Init-Skript für das Regelungsmodell

```
clear

% ------------------------------------------------------------------------
% Konstanten
% ------------------------------------------------------------------------
% Boolean
bool_WAHR                   = true;
bool_FALSCH                 = false;
% Integer 16
int16_NULL                  = 0;
int16_EINS                  = 1;
int16_MAXINT_HALBE          = 16384;
int16_ZWEI_HOCH_ZEHN        = 4096;
int16_ZWEI_HOCH_ELF         = 2048;
int16_INFORM_PHASE_U        = 0;
int16_INFORM_PHASE_V        = 1;
int16_INFORM_PHASE_W        = 2;
% Fixpoint IQ32.28
fixpt28_SQRTDREI_DIV_ZWEI = sqrt(3)/2;
fixpt28_SQRTDREI_DIV_DREI = sqrt(3)/3;
fixpt28_EINS_DIV_ZWEI       = 0.5;
fixpt28_PI_DIV_ACHT         = pi/8;
fixpt28_ACHT_DIV_PI         = 8/pi;
fixpt28_EINS_DIV_SECHZEHN = 1/16;
fixpt28_NULL                = 0;
fixpt28_EINS                = 1;
fixpt28_ZWEI                = 2;
fixpt28_VIER                = 4;
fixpt28_MINUS_ACHT          = -8;
fixpt28_ZWEI_DIV_DREI       = 2/3;
fixpt28_EINS_DIV_DREI       = 1/3;
% Fixpoint IQ32.12
fixpt12_NULL                = 0;
fixpt12_EINS                = 1;
fixpt12_PI_DIV_ACHT         = pi/8;
fixpt12_SQRTDREI_DIV_ZWEI = sqrt(3)/2;
% ------------------------------------------------------------------------



% ------------------------------------------------------------------------
% Soll-Initialwerte
% ------------------------------------------------------------------------
i_soll_d = 0; % Sollwert d-Strom
i_soll_q = 0; % Sollwert q-Strom

% ------------------------------------------------------------------------
% Timing
% ------------------------------------------------------------------------
CPU_Frequenz      = 96e6;           % CPU-Frequenz [Hz]
Kurztask_Periode  = 100e-6;         % Zeit des Kurztask  [s]
```

```
52 Langtask_Periode  = 400e-6;           % Zeit des Langtask [s]
53 PWMa_Frequenz     = 2/Kurztask_Periode;% Kurztask-Frequenz [Hz]
54 PWMb_Frequenz     = 100e3;             % PWM-Frequenz Highside [Hz]
55 PWMc_Frequenz     = 1/Langtask_Periode;% Langtask-Frequenz [Hz]
56 PWMa_Totzeit     = 1.2e-6;           % Totzeit der PWM-Kanäle [s]
57 timer_period_pwma = int32(CPU_Frequenz/(2*PWMa_Frequenz));
58 timer_period_pwmb = int32(CPU_Frequenz/(2*PWMb_Frequenz));
59 timer_period_pwmc = int32(CPU_Frequenz/(2*PWMc_Frequenz));
60 deadband_pwma     = int32(PWMa_Totzeit*CPU_Frequenz);
61 dt_solver         = Kurztask_Periode; % Solver-Schrittweite = Kurztaskperiode
62 %-------------------------------------------------------------------------
63
64
65 %-------------------------------------------------------------------------
66 % Raumzeiger Initialwerte
67 %-------------------------------------------------------------------------
68 raumzeiger_amplitude = 0; % Amplitude für Raumzeiger-Vorgabe, max sqrt(3)/2
69 raumzeiger_frequenz  = 0; % Frequenz in Hz für Raumzeiger-Vorgabe, 512 Hz
70 %-------------------------------------------------------------------------
71
72
73 %-------------------------------------------------------------------------
74 % Stromregler
75 %-------------------------------------------------------------------------
76 betriebsmodus           = 5; % 1-3: Debugging-Modi, 4: Resolverwinkel, 5: Sensorlos
77 stromregler_q_KI        = 1100; % Integralverstärkung q-Anteil
78 stromregler_q_KP        = 0.96; % Proportionalverstärkung q-Anteil
79 stromregler_d_KI        = 1400; % Integralverstärkung d-Anteil
80 stromregler_d_KP        = 1.40; % Proportionalverstärkung d-Anteil
81 stromregler_K_AWU       = 1.00; % Anti-Windup-Konstante Stromregler
82 stromregler_disable_I   = 0;    % mit I-Anteil [0] ; I-Anteil abdrehen [1]
83 %-------------------------------------------------------------------------
84
85
86 %-------------------------------------------------------------------------
87 % Drehzahlregler
88 %-------------------------------------------------------------------------
89 drehzahlregler_KI          = 5.4; % Integralverstärkung Drehzahlregler
90 drehzahlregler_KP          = 1.45; % Proportionalverstärkung Drehzahlregler
91 drehzahlregler_K_AWU       = 1.00; % Anti-Windup-Konstante Drehzahlregler
92 drehzahlregler_disable_I   = 0; % mit I-Anteil [0] ; I-Anteil abdrehen [1]
93 drehzahlregler_ein         = 1; %Drehzahlregler aktiv [1] ; inaktiv [0]
94 %-------------------------------------------------------------------------
95
96
97 %-------------------------------------------------------------------------
98 % Umrichter-Initialwerte
99 %-------------------------------------------------------------------------
100 umrichter_shutdown   = 1; % Umrichter nicht freigegeben [1] ; Freigabe [0]
101 umrichter_laderelais = 0; % Laderelais ein [1] ; aus [0]
102 %-------------------------------------------------------------------------
103
104 %-------------------------------------------------------------------------
105 % Maschinendaten
106 %-------------------------------------------------------------------------
107 % Nennstrom:
108 polpaarzahl               = 5; % Polpaarzahl der Maschine
109 polpaarzahl_multiplikator = 4;  %(wenn Polpaarzahl > 8, dann Multiplikator anwenden)
```

```
110                                 %Bsp: Polpaarzahl = 10 =>  Polpaarzahl  = 5;
111                                 %                          Muliplikator = 2;
112 I_Nenn            = 20;  % Nennstrom [Ampere]
113 U_Nenn            = 30;  % Nennspannung [Volt]
114 I_Bezug           = I_Nenn*sqrt(2);  % Bezugsstrom [Ampere]
115 U_Bezug           = U_Nenn*sqrt(2);  % Bezugsspannung [Volt]
116 Rs_Bezug          = U_Bezug/I_Bezug; % Bezugswiderstand [Ohm]
117 omega_Mech       = 600; % Nenndrehzahl [rpm]
118 omega_Bezug      = omega_Mech * polpaarzahl * polpaarzahl_multiplikator *2*pi / 60;
119 Ls_Bezug         = U_Bezug / (I_Bezug * omega_Bezug); % Bezugsinduktivität [H]
120 Rs               = 0.17; % Statorwiderstand [Ohm]
121 Ls               = 479e-6; % Statorinduktivität [H]
122 rs               = Rs/Rs_Bezug; % bez. Statorwiderstand [1]
123 ls               = Ls/Ls_Bezug; % bez. Statorinduktivität [1]
124 M_Bezug          = 3*I_Nenn*U_Nenn/omega_Bezug; % bez. Drehmoment [1]
125 lageoffset      = 6.635; % Mechanischer Lageoffset [+/- 8 Format]
126 % ------------------------------------------------------------------------
127
128
129 % ------------------------------------------------------------------------
130 % Strommessung auf dem Umrichter und der Backplane (Lem)
131 % ------------------------------------------------------------------------
132 % Bürde des Stromwandlers (Umrichter): 120 Ohm
133 % Übersetzung Stromwandler (Umrichter): 1/500
134 % Offset der OPV Schaltung (Umrichter): 2.5 V
135 % Widerstandsverhältnis der OPV Verstärkerschaltung (Umrichter): 3.3kOhm/10kOhm =
      0.33
136 % Eingangspegel des ADC-Treibers (Backplane): 0...5V
137 I_Umrichter_Steigung = 0.33*120/500;
138 I_Messbereich        = 2*(2.5/I_Umrichter_Steigung);
139 I_Normfaktor         = I_Messbereich/I_Bezug;
140 % ------------------------------------------------------------------------
141
142
143 % ------------------------------------------------------------------------
144 % Spannungsmessung auf dem Umrichter und der Backplane (Shunt)
145 % ------------------------------------------------------------------------
146 % Offset der OPV Schaltung (Umrichter): 2.5 V
147 % Spannungsteilerverhältnis vor OPV Schaltung: (2.2e3 + 220)/(2.2e3 + 220 + 2*47e3)
148 % Widerstandsverhältnis der OPV Verstärkerschaltung (Umrichter): 20kOhm/19.6kOhm =
      1.02
149 % Eingangspegel des ADC-Treibers (Backplane): 0...5V
150 U_Umrichter_Steigung        = 1.0204*((2.2e3 + 220)/(2.2e3 + 220 + 2*47e3));
151 U_Messbereich               = 2*(2.5/U_Umrichter_Steigung);
152 U_Normfaktor                = U_Messbereich/U_Bezug;
153 % ------------------------------------------------------------------------
154
155
156 % ------------------------------------------------------------------------
157 % EMK Modell
158 % ------------------------------------------------------------------------
159 EMK_Stabilisierung    =  0.05; % Rückführungsfaktor EMK-Modell
160 EMK_KI                =  200;  % Integralverstärkung EMK-Modell
161 omega_EMK_min         =  0.1;  % Umschaltpunkt für EMK-Modell [% Nenndrehzahl]
162 lageoffset_EMK        = -0.1;  % Korrekturwinkel [+/- 8 Format]
163 % ------------------------------------------------------------------------
164
165
```

```
166 % ------------------------------------------------------------------------
167 % INFORM Großsignal
168 % ------------------------------------------------------------------------
169 % Dauer der INFORM-Sequenz [s]
170 dauer_inform_vor_gross      = 1.4e-3;
171 % Punkte der Interruptaufrufe relativ zur Gesamtdauer des Schusses
172 p_inform_gross              = [0 0.145 0.230 0.250 0.270 0.355 0.645 0.730 0.750
        0.770 0.855 1];
173 % Zeitpunkte der Interruptaufrufe
174 t_inform_gross              = p_inform_gross *dauer_inform_vor_gross;
175 % PWM-Perioden
176 timer_inform_gross          = int16(diff(t_inform_gross )*CPU_Frequenz/2);
177 % INFORM-Schusspolaritäten
178 polarity_inform_gross       = [1 1 1 -1 -1 -1 -1 -1 1 1 1];
179 % Auswertungscases
180 deltacase_inform_gross      = [3 6 8 11]+1;
181 % PWM-Perioden für alle drei Phasen
182 timer_inform_gross_uvw      = [timer_inform_gross timer_inform_gross
        timer_inform_gross];
183 % INFORM-Schusspolaritäten aller drei Phasen
184 polarity_inform_gross_uvw   = [polarity_inform_gross polarity_inform_gross
        polarity_inform_gross];
185 % INFORM-Phasenvektor
186 phase_inform_gross_uvw      = [repmat(int16_INFORM_PHASE_U,1,length(
        polarity_inform_gross)) ...
187                                repmat(int16_INFORM_PHASE_V,1,length(
                                       polarity_inform_gross)) ...
188                                repmat(int16_INFORM_PHASE_W,1,length(
                                       polarity_inform_gross))];
189 % Auswertungscases aller drei Phasen
190 deltacase_inform_gross_uvw  = [deltacase_inform_gross'...
191                                (deltacase_inform_gross+length(polarity_inform_gross))
                                       '...
192                                (deltacase_inform_gross+2*length(polarity_inform_gross
                                       ))'];
193
194 % ------------------------------------------------------------------------
195 % INFORM Kleinsignal
196 % ------------------------------------------------------------------------
197 % Dauer der INFORM-Sequenz [s]
198 dauer_inform_vor_klein      = 0.40e-3;
199 % Punkte der Interruptaufrufe relativ zur Gesamtdauer des Schusses
200 p_inform_klein              = [0 0.1667 0.2167 0.45 0.5 0.55 0.7833 0.833 1];
201 % Zeitpunkte der Interruptaufrufe
202 t_inform_klein              = p_inform_klein *dauer_inform_vor_klein;
203 % PWM-Perioden
204 timer_inform_klein          = int16(diff(t_inform_klein )*CPU_Frequenz/2);
205 % INFORM-Schusspolaritäten
206 polarity_inform_klein       = [1 -1 -1 -1 1 1 1 -1];
207 % Auswertungscases
208 deltacase_inform_klein      = [4 7]+1;
209 % PWM-Perioden für alle drei Phasen
210 timer_inform_klein_uvw      = [timer_inform_klein timer_inform_klein
        timer_inform_klein];
211 % INFORM-Schusspolaritäten aller drei Phasen
212 polarity_inform_klein_uvw   = [polarity_inform_klein polarity_inform_klein
        polarity_inform_klein];
213 % INFORM-Phasenvektor
```

```
214 phase_inform_klein_uvw      = [repmat(int16_INFORM_PHASE_U,1,length(
        polarity_inform_klein)) ...
215                                repmat(int16_INFORM_PHASE_V,1,length(
                                       polarity_inform_klein)) ...
216                                repmat(int16_INFORM_PHASE_W,1,length(
                                       polarity_inform_klein))];
217 % Auswertungscases aller drei Phasen
218 deltacase_inform_klein_uvw  = [deltacase_inform_klein '...
219                                (deltacase_inform_klein+length(polarity_inform_klein))
                                       '...
220                                (deltacase_inform_klein+2*length(polarity_inform_klein
                                       )) '];
221
222
223 % ------------------------------------------------------------------------
224 % INFORM Parameter
225 % ------------------------------------------------------------------------
226 ratio_stromregler_inform = 7/3; % [Zeitverhältnis Stromregler/INFORM]
227 counter_trigger_inform   = int16((6*dauer_inform_vor_klein*ratio_stromregler_inform)
        /(Kurztask_Periode));
228 % ------------------------------------------------------------------------
229
230
231 % ------------------------------------------------------------------------
232 % Beobachter
233 % ------------------------------------------------------------------------
234 kalfakt1                    = 0.05; % 1. Kalman-Faktor
235 kalfakt2                    = 0.01; % 2. Kalman-Faktor
236 delta_tau_beobachter        = omega_Bezug*Kurztask_Periode*fixpt28_ACHT_DIV_PI;
237 omega_KP_beobachter         = 1/(polpaarzahl*polpaarzahl_multiplikator);
238 % ------------------------------------------------------------------------
239
240
241 % ------------------------------------------------------------------------
242 % Auto-Temperaturdrift-Abgleich der Phasenstrommessung
243 % ------------------------------------------------------------------------
244 offsetabgl_strom_KI         = 10; % Integralverstärkung für Strom-Offsetabgleich
245 offsetabgl_strom_PT1        = 1;  % Offsetabgleich aktiv [1] ; inaktiv [0]
246 % ------------------------------------------------------------------------
247
248
249 % ------------------------------------------------------------------------
250 % Geschwindigkeit, Winkel
251 % ------------------------------------------------------------------------
252 omega_KI                    = 200; % [1/s]
253 omega_KP                    = 60/(Kurztask_Periode*omega_Mech*
        polpaarzahl_multiplikator*polpaarzahl*16);
254 omega_soll                  = 0; % Solldrehzahl [% Nenndrehzahl]
255 gamma_invertieren           = 0; % gamma Resolver invertieren [1] ; nicht inv. [0]
256 omega_invertieren           = 0; % Drehrichtung umkehren [1] ; nicht umkehren [0]
257 % ------------------------------------------------------------------------
```

include/init_model.m

MATLAB-Skript für die Erstellung der S-Funktion der Kommunikationsschnittstelle

```
clear
clc

% S-Function und Simulink-Block Eigenschaften
sfun_name = 'comm';
modelname = 'FOC_PSM_2808';
simblk_desc = sprintf([...
        'Transmit bytes over a serial interface. '...
        'The block is designed for a baud rate of 115200 and SCIA Interface \nn'...
        'This block was generated automatically by running the function '...
        mfilename '.']);
simblk_disp = 'SCIA Communication';

% Initialisierungsstruktur
fprintf('Initialisiere MATLAB Legacy Tool S-Function builder')
def = legacy_code('initialize'); %#ok<*NOPTS>
def.SourceFiles = {[sfun_name '_mex.c'],'DSP280x_Sci_mex.c'};
def.HeaderFiles = {[sfun_name '.h'],'SciA.h'};
def.SFunctionName = ['sfun_' sfun_name];
def.OutputFcnSpec = ['void ' sfun_name '(void)'];
def.IncPaths = {'Include'};
def.SrcPaths = {'Source'};
def.SampleTime = 'inherited';
n=0;
while (n<5)
    n=n+1;
    fprintf('.');
    pause(0.5);
end
fprintf('ok!\n\n')

% S Function Quellcode erzeugen und Ordnerstruktur des Zielprojekts
% festlegen (Include, Source etc.)
fprintf('Erstelle C-Mex Quellcode')
legacy_code('sfcn_cmex_generate', def)
legacy_code('rtwmakecfg_generate', def);
n=0;
while ( (n<5) || (~exist(['sfun_' sfun_name '.c'],'file')) )
    n=n+1;
    fprintf('.');
    pause(0.5);
end
fprintf('ok!\n\n')

% Quellcode kompilieren
fprintf('Kompiliere C-Mex Quellcode')
legacy_code('generate_for_sim', def)
arch=computer('arch');
n=0;
while ( (n<5) || (~exist(['sfun_' sfun_name '.mexw' arch(end-1:end)],'file')) )
    n=n+1;
    fprintf('.');
    pause(0.5);
end
fprintf('ok!\n\n')
```

```
56
57 % Target language compiler file anpassen
58 fprintf('Target Language Compiler Datei anpassen')
59 replaceinfile('\"DSP280x_Sci_mex.c\"','\"DSP280x_Sci.c\"',['sfun_' sfun_name '.tlc'
       ],'-nobak');
60 replaceinfile(['\"' sfun_name '_mex.c\"'],['\"' sfun_name '.c\"'],['sfun_' sfun_name
       '.tlc'],'-nobak');
61 n=0;
62 while ( (n<5) || (~exist(['sfun_' sfun_name '.c'],'file')) )
63     n=n+1;
64     fprintf('.');
65     pause(0.5);
66 end
67 fprintf('ok!\n\n')
68
69 % Simulink Block erstellen, falls gewünscht
70 reply = input('Neuen Simulink Block erstellen? J/N [N]: ', 's');
71 if isempty(reply)
72     reply = 'N';
73 end
74 if reply(1) == 'J'
75     fprintf('Erstelle neuen Simulink Block...')
76     legacy_code('slblock_generate', def, modelname);
77     sim_blk = gcb;
78     displayCmd = ['text(0.5, 0.5,''' simblk_disp ''','horizontalAlignment','
           center');'];
79     set_param(sim_blk,'MaskDescription', simblk_desc);
80     set_param(sim_blk,'MaskDisplay',displayCmd);
81     set_param(sim_blk,'MaskDisplay',displayCmd);
82     displayCmd=strrep(displayCmd,'''',''''');
83     displayCmd = ['set_param(gcb,''MaskDisplay'',''' displayCmd ''');'];
84     set_param(gcb, 'StartFcn', displayCmd);
85     fprintf('ok!\n\n')
86 end
```

include/create_sfun_comm.m

MATLAB-Skript für die Erstellung der S-Funktion der PWM-Raumzeigermodulation

```
clear
clc

% S-Function und Simulink-Block Eigenschaften
sfun_name = 'pwma';
modelname = 'FOC_PSM_2808';
simblk_desc = sprintf([...
        'Generate PWM Compare values out of a space Vector with Ualpha and '...
        'Ubeta components\n\n'...
        'This block was generated automatically by running the function '...
        mfilename '.']);
simblk_disp   = 'PWM CMPR Values';
simblk_input1 = 'U_alpha';
simblk_input2 = 'U_beta';
simblk_input3 = 'Timer period';
simblk_output1 = 'CMPR1';
simblk_output2 = 'CMPR2';
simblk_output3 = 'CMPR3';

% Initialisierungsstruktur
fprintf('Initialisiere MATLAB Legacy Tool S-Function builder')
def = legacy_code('initialize'); %#ok<*NOPTS>
def.SourceFiles = {[sfun_name '_mex.c']};
def.HeaderFiles = {[sfun_name '.h']};
def.SFunctionName = ['sfun_' sfun_name];
def.OutputFcnSpec = ['void ' sfun_name '(int16 u1, int16 u2, int16 u3, int16 y1[1],
    int16 y2[1], int16 y3[1])']; % three inputs, three outputs
def.IncPaths = {'Include'};
def.SrcPaths = {'Source'};
def.SampleTime = 'inherited';
n=0;
while (n<5)
    n=n+1;
    fprintf('.');
    pause(0.5);
end
fprintf('ok!\n\n')

% S Function Quellcode erzeugen und Ordnerstruktur des Zielprojekts
% festlegen (Include, Source etc.)
fprintf('Erstelle C-Mex Quellcode')
legacy_code('sfcn_cmex_generate', def)
legacy_code('rtwmakecfg_generate', def);
n=0;
while ( (n<5) || (~exist(['sfun_' sfun_name '.c'],'file')) )
    n=n+1;
    fprintf('.');
    pause(0.5);
end
fprintf('ok!\n\n')

% Quellcode kompilieren
fprintf('Kompiliere C-Mex Quellcode')
legacy_code('generate_for_sim', def)
arch=computer('arch');
```

```
n=0;
while ( (n<5) || (~exist(['sfun_' sfun_name '.mexw' arch(end-1:end)],'file')) )
    n=n+1;
    fprintf('.');
    pause(0.5);
end
fprintf('ok!\n\n')

% Target language compiler file anpassen
fprintf('Target Language Compiler Datei anpassen')
replaceinfile(['\"' sfun_name '_mex.c\"'],['\"' sfun_name '.c\"'],['sfun_' sfun_name
    '.tlc'],'-nobak');
n=0;
while ( (n<5) || (~exist(['sfun_' sfun_name '.c'],'file')) )
    n=n+1;
    fprintf('.');
    pause(0.5);
end
fprintf('ok!\n\n')

% Simulink Block erstellen, falls gewünscht
reply = input('Neuen Simulink Block erstellen? J/N [N]: ', 's');
if isempty(reply)
    reply = 'N';
end
if reply(1) == 'J'
    fprintf('Erstelle neuen Simulink Block...')
    legacy_code('slblock_generate', def, modelname);
    sim_blk = gcb;
    displayCmd = ['text(0.5, 0.5,''' simblk_disp ''',''horizontalAlignment'',''
        center'');'...
                  'port_label(''input'',1,''' simblk_input1 ''');'...
                  'port_label(''input'',2,''' simblk_input2 ''');'...
                  'port_label(''input'',3,''' simblk_input3 ''');'...
                  'port_label(''output'',1,''' simblk_output1 ''');'...
                  'port_label(''output'',2,''' simblk_output2 ''');'...
                  'port_label(''output'',3,''' simblk_output3 ''')'];
    set_param(sim_blk,'MaskDescription', simblk_desc);
    set_param(sim_blk,'MaskDisplay',displayCmd);
    set_param(sim_blk,'MaskDisplay',displayCmd);
    displayCmd=strrep(displayCmd,'''',''''''');
    displayCmd = ['set_param(gcb,''MaskDisplay'',''' displayCmd ''');'];
    set_param(gcb, 'StartFcn', displayCmd);
    fprintf('ok!\n\n')
end
```

include/create_sfun_pwma.m

MATLAB-Skript für die Erstellung der S-Funktion des INFORM-Kontollblocks

```
clear
clc

% S-Function und Simulink-Block Eigenschaften
sfun_name = 'controlInform';
modelname = 'FOC_PSM_2808';
simblk_desc = sprintf([...
        'Control INFORM Sequence'...
        'This block was generated automatically by running the function '...
        mfilename '.']);
simblk_disp   = 'Control INFORM Sequence';

% Initialisierungsstruktur
fprintf('Initialisiere MATLAB Legacy Tool S-Function builder')
def = legacy_code('initialize'); %#ok<*NOPTS>
def.SourceFiles = {[sfun_name '_mex.c']};
def.HeaderFiles = {[sfun_name '.h']};
def.SFunctionName = ['sfun_' sfun_name];
def.OutputFcnSpec = ['void ' sfun_name '(int16 u1, int16 u2, int16 u3, int16 u4)'];
    % four inputs
def.IncPaths = {'Include'};
def.SrcPaths = {'Source'};
def.SampleTime = 'inherited';
n=0;
while (n<5)
    n=n+1;
    fprintf('.');
    pause(0.5);
end
fprintf('ok!\n\n')

% S Function Quellcode erzeugen und Ordnerstruktur des Zielprojekts
% festlegen (Include, Source etc.)
fprintf('Erstelle C-Mex Quellcode')
legacy_code('sfcn_cmex_generate', def)
legacy_code('rtwmakecfg_generate', def);
n=0;
while ( (n<5) || (~exist(['sfun_' sfun_name '.c'],'file')) )
    n=n+1;
    fprintf('.');
    pause(0.5);
end
fprintf('ok!\n\n')

% Quellcode kompilieren
fprintf('Kompiliere C-Mex Quellcode')
legacy_code('generate_for_sim', def)
arch=computer('arch');
n=0;
while ( (n<5) || (~exist(['sfun_' sfun_name '.mexw' arch(end-1:end)],'file')) )
    n=n+1;
    fprintf('.');
    pause(0.5);
end
fprintf('ok!\n\n')
```

```
55
56 % Target language compiler file anpassen
57 fprintf('Target Language Compiler Datei anpassen')
58 replaceinfile(['\"' sfun_name '_mex.c\"'],['\"' sfun_name '.c\"'],['sfun_' sfun_name
       '.tlc'],'-nobak');
59 n=0;
60 while ( (n<5) || (~exist(['sfun_' sfun_name '.c'],'file')) )
61     n=n+1;
62     fprintf('.');
63     pause(0.5);
64 end
65 fprintf('ok!\n\n')
66
67 % Simulink Block erstellen, falls gewünscht
68 reply = input('Neuen Simulink Block erstellen? J/N [N]: ', 's');
69 if isempty(reply)
70     reply = 'N';
71 end
72 if reply(1) == 'J'
73     fprintf('Erstelle neuen Simulink Block...')
74     legacy_code('slblock_generate', def, modelname);
75     sim_blk = gcb;
76     displayCmd = ['text(0.5, 0.5,''' simblk_disp ''','horizontalAlignment','
           center');'];
77     set_param(sim_blk,'MaskDescription', simblk_desc);
78     set_param(sim_blk,'MaskDisplay',displayCmd);
79     displayCmd=strrep(displayCmd,'''',''''');
80     displayCmd = ['set_param(gcb,''MaskDisplay'',''' displayCmd ''');'];
81     set_param(gcb, 'StartFcn', displayCmd);
82     fprintf('ok!\n\n')
83 end
```

include/create_sfun_controlInform.m

MATLAB-Skript für die Erstellung der S-Funktion des Winkelresolvers

```
clear
clc

% S-Function und Simulink-Block Eigenschaften
sfun_name = 'resolver';
modelname = 'FOC_PSM_2808';
simblk_desc = sprintf([...
        'Serieller Resolver '...
        'This block was generated automatically by running the function '...
        mfilename '.']);
simblk_disp   = 'Resolver';

% Initialisierungsstruktur
fprintf('Initialisiere MATLAB Legacy Tool S-Function builder')
def = legacy_code('initialize'); %#ok<*NOPTS>
def.SourceFiles = {[sfun_name '_mex.c'],'DSP280x_Spi_mex.c'};
def.HeaderFiles = {[sfun_name '.h']};
def.SFunctionName = ['sfun_' sfun_name];
def.OutputFcnSpec = ['int16 y1 = ' sfun_name '(void)'];
def.IncPaths = {'Include'};
def.SrcPaths = {'Source'};
def.SampleTime = 'inherited';
n=0;
while (n<5)
    n=n+1;
    fprintf('.');
    pause(0.5);
end
fprintf('ok!\n\n')

% S Function Quellcode erzeugen und Ordnerstruktur des Zielprojekts
% festlegen (Include, Source etc.)
fprintf('Erstelle C-Mex Quellcode')
legacy_code('sfcn_cmex_generate', def)
legacy_code('rtwmakecfg_generate', def);
n=0;
while ( (n<5) || (~exist(['sfun_' sfun_name '.c'],'file')) )
    n=n+1;
    fprintf('.');
    pause(0.5);
end
fprintf('ok!\n\n')

% Quellcode kompilieren
fprintf('Kompiliere C-Mex Quellcode')
legacy_code('generate_for_sim', def)
arch=computer('arch');
n=0;
while ( (n<5) || (~exist(['sfun_' sfun_name '.mexw' arch(end-1:end)],'file')) )
    n=n+1;
    fprintf('.');
    pause(0.5);
end
fprintf('ok!\n\n')

```

```
56 % Target language compiler file anpassen
57 fprintf('Target Language Compiler Datei anpassen')
58 replaceinfile(['\"' sfun_name '_mex.c\"'],['\"' sfun_name '.c\"'],['sfun_' sfun_name
       '.tlc'],'-nobak');
59 replaceinfile('\"DSP280x_Spi_mex.c\"','\"DSP280x_Spi.c\"',['sfun_' sfun_name '.tlc'
     ],'-nobak');
60 n=0;
61 while ( (n<5) || (~exist(['sfun_' sfun_name '.c'],'file')) )
62     n=n+1;
63     fprintf('.');
64     pause(0.5);
65 end
66 fprintf('ok!\n\n')
67
68 % Simulink Block erstellen, falls gewünscht
69 reply = input('Neuen Simulink Block erstellen? J/N [N]: ', 's');
70 if isempty(reply)
71     reply = 'N';
72 end
73 if reply(1) == 'J'
74     fprintf('Erstelle neuen Simulink Block...')
75     legacy_code('slblock_generate', def, modelname);
76     sim_blk = gcb;
77     displayCmd = ['text(0.5, 0.5,''' simblk_disp ''','horizontalAlignment','
         center');'];
78     set_param(sim_blk,'MaskDescription', simblk_desc);
79     set_param(sim_blk,'MaskDisplay',displayCmd);
80     set_param(sim_blk,'MaskDisplay',displayCmd);
81     displayCmd=strrep(displayCmd,'''',''''');
82     displayCmd = ['set_param(gcb,''MaskDisplay'',''' displayCmd ''');'];
83     set_param(gcb, 'StartFcn', displayCmd);
84     fprintf('ok!\n\n')
85 end
```

include/create_sfun_resolver.m

MATLAB-Skript für die Erstellung der S-Funktion des Softauges

```
clear
clc

% S-Function und Simulink-Block Eigenschaften
sfun_name = 'softauge';
modelname = 'FOC_PSM_2808';
simblk_desc = sprintf([...
        'Serielles Softauge '...
        'This block was generated automatically by running the function '...
        mfilename '.']);
simblk_disp   = 'Softauge';

% Initialisierungsstruktur
fprintf('Initialisiere MATLAB Legacy Tool S-Function builder')
def = legacy_code('initialize'); %#ok<*NOPTS>
def.SourceFiles = {[sfun_name '_mex.c'],'DSP280x_Spi_mex.c'};
def.HeaderFiles = {[sfun_name '.h']};
def.SFunctionName = ['sfun_' sfun_name];
def.OutputFcnSpec = ['void ' sfun_name '(void)'];
def.IncPaths = {'Include'};
def.SrcPaths = {'Source'};
def.SampleTime = 'inherited';
n=0;
while (n<5)
    n=n+1;
    fprintf('.');
    pause(0.5);
end
fprintf('ok!\n\n')

% S Function Quellcode erzeugen und Ordnerstruktur des Zielprojekts
% festlegen (Include, Source etc.)
fprintf('Erstelle C-Mex Quellcode')
legacy_code('sfcn_cmex_generate', def)
legacy_code('rtwmakecfg_generate', def);
n=0;
while ( (n<5) || (~exist(['sfun_' sfun_name '.c'],'file')) )
    n=n+1;
    fprintf('.');
    pause(0.5);
end
fprintf('ok!\n\n')

% Quellcode kompilieren
fprintf('Kompiliere C-Mex Quellcode')
legacy_code('generate_for_sim', def)
arch=computer('arch');
n=0;
while ( (n<5) || (~exist(['sfun_' sfun_name '.mexw' arch(end-1:end)],'file')) )
    n=n+1;
    fprintf('.');
    pause(0.5);
end
fprintf('ok!\n\n')

```

```
% Target language compiler file anpassen
fprintf('Target Language Compiler Datei anpassen')
replaceinfile(['\"' sfun_name '_mex.c\"'],['\"' sfun_name '.c\"'],['sfun_' sfun_name '.tlc'],'-nobak');
replaceinfile('\"DSP280x_Spi_mex.c\"','\"DSP280x_Spi.c\"',['sfun_' sfun_name '.tlc'],'-nobak');
n=0;
while ( (n<5) || (~exist(['sfun_' sfun_name '.c'],'file')) )
    n=n+1;
    fprintf('.');
    pause(0.5);
end
fprintf('ok!\n\n')

% Simulink Block erstellen, falls gewünscht
reply = input('Neuen Simulink Block erstellen? J/N [N]: ', 's');
if isempty(reply)
    reply = 'N';
end
if reply(1) == 'J'
    fprintf('Erstelle neuen Simulink Block...')
    legacy_code('slblock_generate', def, modelname);
    sim_blk = gcb;
    displayCmd = ['text(0.5, 0.5,''' simblk_disp ''','horizontalAlignment','center');'];
    set_param(sim_blk,'MaskDescription', simblk_desc);
    set_param(sim_blk,'MaskDisplay',displayCmd);
    set_param(sim_blk,'MaskDisplay',displayCmd);
    displayCmd=strrep(displayCmd,'''','''''');
    displayCmd = ['set_param(gcb,''MaskDisplay'',''' displayCmd ''');'];
    set_param(gcb, 'StartFcn', displayCmd);
    fprintf('ok!\n\n')
end
```

include/create_sfun_softauge.m

MATLAB-Skript für die Erstellung der S-Funktion für die Rücksetzung des TripZone-Fehlers

```
clear
clc

% S-Function und Simulink-Block Eigenschaften
sfun_name = 'tzclear';
modelname = 'FOC_PSM_2808';
simblk_desc = sprintf([...
        'Clear Trip Zone '...
        'This block was generated automatically by running the function '...
        mfilename '.']);
simblk_disp   = 'Clear Trip Zone';

% Initialisierungsstruktur
fprintf('Initialisiere MATLAB Legacy Tool S-Function builder')
def = legacy_code('initialize'); %#ok<*NOPTS>
def.SourceFiles = {[sfun_name '_mex.c']};
def.HeaderFiles = {[sfun_name '.h']};
def.SFunctionName = ['sfun_' sfun_name];
def.OutputFcnSpec = ['void ' sfun_name '(void)'];
def.IncPaths = {'Include'};
def.SrcPaths = {'Source'};
def.SampleTime = 'inherited';
n=0;
while (n<5)
    n=n+1;
    fprintf('.');
    pause(0.5);
end
fprintf('ok!\n\n')

% S Function Quellcode erzeugen und Ordnerstruktur des Zielprojekts
% festlegen (Include, Source etc.)
fprintf('Erstelle C-Mex Quellcode')
legacy_code('sfcn_cmex_generate', def)
legacy_code('rtwmakecfg_generate', def);
n=0;
while ( (n<5) || (~exist(['sfun_' sfun_name '.c'],'file')) )
    n=n+1;
    fprintf('.');
    pause(0.5);
end
fprintf('ok!\n\n')

% Quellcode kompilieren
fprintf('Kompiliere C-Mex Quellcode')
legacy_code('generate_for_sim', def)
arch=computer('arch');
n=0;
while ( (n<5) || (~exist(['sfun_' sfun_name '.mexw' arch(end-1:end)],'file')) )
    n=n+1;
    fprintf('.');
    pause(0.5);
end
fprintf('ok!\n\n')

```

```
% Target language compiler file anpassen
fprintf('Target Language Compiler Datei anpassen')
replaceinfile(['\"' sfun_name '_mex.c\"'],['\"' sfun_name '.c\"'],['sfun_' sfun_name '.tlc'],'-nobak');
n=0;
while ( (n<5) || (~exist(['sfun_' sfun_name '.c'],'file')) )
    n=n+1;
    fprintf('.');
    pause(0.5);
end
fprintf('ok!\n\n')

% Simulink Block erstellen, falls gewünscht
reply = input('Neuen Simulink Block erstellen? J/N [N]: ', 's');
if isempty(reply)
    reply = 'N';
end
if reply(1) == 'J'
    fprintf('Erstelle neuen Simulink Block...')
    legacy_code('slblock_generate', def, modelname);
    sim_blk = gcb;
    displayCmd = ['text(0.5, 0.5,''' simblk_disp ''','horizontalAlignment','center');'];
    set_param(sim_blk,'MaskDescription', simblk_desc);
    set_param(sim_blk,'MaskDisplay',displayCmd);
    displayCmd=strrep(displayCmd,'''',''''''');
    displayCmd = ['set_param(gcb,''MaskDisplay'',''' displayCmd ''');'];
    set_param(gcb, 'StartFcn', displayCmd);
    fprintf('ok!\n\n')
end
```

include/create_sfun_tzclear.m

MATLAB-Skript für die Erstellung der S-Funktion des ADC für die INFORM-Messungen

```
clear
clc

% S-Function und Simulink-Block Eigenschaften
sfun_name = 'adcinform';
modelname = 'FOC_PSM_2808';
simblk_desc = sprintf([...
        'Read out ADC Values for inform measurements'...
        'This block was generated automatically by running the function '...
        mfilename '.']);
simblk_disp   = 'ADC for INFORM';

% Initialisierungsstruktur
fprintf('Initialisiere MATLAB Legacy Tool S-Function builder')
def = legacy_code('initialize'); %#ok<*NOPTS>
def.SourceFiles = {[sfun_name '_mex.c']};
def.HeaderFiles = {[sfun_name '.h']};
def.SFunctionName = ['sfun_' sfun_name];
def.OutputFcnSpec = ['void ' sfun_name '(int16 y1[1], int16 y2[1], int16 y3[1])'];
   % three outputs
def.IncPaths = {'Include'};
def.SrcPaths = {'Source'};
def.SampleTime = 'inherited';
n=0;
while (n<5)
    n=n+1;
    fprintf('.');
    pause(0.5);
end
fprintf('ok!\n\n')

% S Function Quellcode erzeugen und Ordnerstruktur des Zielprojekts
% festlegen (Include, Source etc.)
fprintf('Erstelle C-Mex Quellcode')
legacy_code('sfcn_cmex_generate', def)
legacy_code('rtwmakecfg_generate', def);
n=0;
while ( (n<5) || (~exist(['sfun_' sfun_name '.c'],'file')) )
    n=n+1;
    fprintf('.');
    pause(0.5);
end
fprintf('ok!\n\n')

% Quellcode kompilieren
fprintf('Kompiliere C-Mex Quellcode')
legacy_code('generate_for_sim', def)
arch=computer('arch');
n=0;
while ( (n<5) || (~exist(['sfun_' sfun_name '.mexw' arch(end-1:end)],'file')) )
    n=n+1;
    fprintf('.');
    pause(0.5);
end
fprintf('ok!\n\n')
```

```
55
56 % Target language compiler file anpassen
57 fprintf('Target Language Compiler Datei anpassen')
58 replaceinfile(['\"' sfun_name '_mex.c\"'],['\"' sfun_name '.c\"'],['sfun_' sfun_name
       '.tlc'],'-nobak');
59 n=0;
60 while ( (n<5) || (~exist(['sfun_' sfun_name '.c'],'file')) )
61     n=n+1;
62     fprintf('.');
63     pause(0.5);
64 end
65 fprintf('ok!\n\n')
66
67 % Simulink Block erstellen, falls gewünscht
68 reply = input('Neuen Simulink Block erstellen? J/N [N]: ', 's');
69 if isempty(reply)
70     reply = 'N';
71 end
72 if reply(1) == 'J'
73     fprintf('Erstelle neuen Simulink Block...')
74     legacy_code('slblock_generate', def, modelname);
75     sim_blk = gcb;
76     displayCmd = ['text(0.5, 0.5,''' simblk_disp ''','horizontalAlignment','
           center');'];
77     set_param(sim_blk,'MaskDescription', simblk_desc);
78     set_param(sim_blk,'MaskDisplay',displayCmd);
79     displayCmd=strrep(displayCmd,'''',''''');
80     displayCmd = ['set_param(gcb,''MaskDisplay'',''' displayCmd ''');'];
81     set_param(gcb, 'StartFcn', displayCmd);
82     fprintf('ok!\n\n')
83 end
```

include/create_sfun_adcinform.m

A.2. Code-Listings: Skript zur Verarbeitung der Messdaten

```
function [] =  plotresults(data, channels, time_limits)
    warning('off') %#ok<*WNOFF>

    % Signale auswählen
    signals = zeros(length(channels),size(data.X1,2));
    times   = zeros(length(channels),size(data.X1,2));
    for i =1:length(channels)
        signals(i,:) = eval(['data.Y' num2str(channels(i))]);
        times(i,:)   = data.X1;
    end

    if nargin == 3
        % Zeitfenster beschränken
        startInd = find(times(1,:) > time_limits(1));
        startInd = startInd(1);
        endInd   = find(times(1,:) > time_limits(2));
        endInd   = endInd(1) - 1;
        signals  = signals(:,startInd:endInd);
        times    = times(:,startInd:endInd);
    end

    % Running average filter
    windowSize = 200;
    signals_filtered = filter(ones(1,windowSize)/windowSize,1,signals')';

    %-----------------------------------------------------------------
    % DARSTELLUNG in Volt
    %-----------------------------------------------------------------
    % Zwei Plots nebeneinander in Latex: width = 20
    % Ein Plot in Latex: width = 31.37254
    width    =     31.37254 ;
    height   =     width*(16/9)^-1 ;
    xoffset     =   1;
    yoffset     =   1;
    rect  = [xoffset  , yoffset , width  , height] ;
    figure1 = figure(...
      'PaperOrientation','landscape',...
      'PaperType','A4', ...
      'Units', 'centimeters',...
      'PaperPositionMode','manual',...
      'Position',rect) ; %#ok<*NASGU>
    plot(times',signals_filtered');
    grid on;
    xlabel('Zeit [s]','FontSize',14);
    ylabel('Spannung [V]','FontSize',14);
    set(gca, 'FontSize',14,'FontWeight','light');
    legend( ...
            strcat('CH1 --> ',  ' V'),...
            strcat('CH2 --> ', ' V'),...
            strcat('CH3 --> ',  ' V'),...
            strcat('CH4 --> ',  ' V'),...
            'Location','NorthEast');

    set(ax(1),'xlim',[time_limits]);
    set(ax(2),'xlim',[time_limits]);
    %-----------------------------------------------------------------
```

```
57      % DARSTELLUNG in DIV angepasst wie oszi
58      %------------------------------------------------------------------
59
60      figure2 = figure(...
61        'PaperOrientation','landscape',...
62        'PaperType','A4', ...
63        'Units', 'centimeters',...
64        'PaperPositionMode','manual',...
65        'Position',rect) ;
66
67      signals_div = zeros(size(signals));
68      for i = 1:length(channels)
69          posX = eval(['data.position_channel' num2str(channels(i))]);
70          YX = signals_filtered(i,:);
71          scaleX = eval(['data.scale_value_channel' num2str(channels(i))]);
72          signals_div(i,:) = (posX+YX)*scaleX^-1;
73      end
74      p=plot(times(:,windowSize:end)',signals_div(:,windowSize:end)');
75      grid on;
76      xlabel('Zeit [s]','FontSize',14);
77      ylabel('DIV','FontSize',14);
78      legend( ...
79              strcat('CH1 --> ', num2str(data.scale_value_channel1,6) , ' V/DIV'),...
80              strcat('CH2 --> ', num2str(data.scale_value_channel2,6) , ' V/DIV'),...
81              strcat('CH3 --> ', num2str(data.scale_value_channel3,6) , ' V/DIV'),...
82              strcat('CH4 --> ', num2str(data.scale_value_channel4,6) , ' V/DIV'),...
83              'Location','NorthEast');
84
85      set(gca, 'FontSize',14,'FontWeight','light');
86      ylim_act = get(gca,'ylim');
87      axis('tight');
88      set(gca,'ylim',[ylim_act(1) ylim_act(2)+0.9]);
89
90      warning('on') %#ok<*WNON>
91  end
```

include/plotresults.m

A.3. HTML-Report des Profiling-Tools

Model Execution Profiling Results

This report contains following sections:

Task Overrun Status

Task overrun status since model execution started. Task overrun status updates continually while the model executes. After the task overrun status flag is set to 1, it is not cleared during the model execution.

Task	Overrun
Base-Rate	No

Analysis of Profiling Data

Execution profiling data was recorded during the 1.199 ms of the model execution starting at time t = 0. The profiling timer resolution is 10.416 ns. The data recorded for task turnaround times and task execution times is presented in the following table.

Task	Maximum turnaround time	Average turnaround time	Maximum execution time	Average execution time	Average sample time
Base-Rate	1.937 µs at t = 998.884 µs	1.894 µs	1.937 µs at t = 998.884 µs	1.894 µs	99.990 µs
Task 2	75.849 µs at t = 1.007 ms	75.236 µs	75.849 µs at t = 1.007 ms	75.236 µs	99.993 µs
Task 3	4.448 µs at t = 984.406 µs	4.274 µs	4.448 µs at t = 984.406 µs	4.274 µs	399.984 µs
Idle	99.879 µs at t = 406.255 µs	33.933 µs	14.249 µs at t = 406.255 µs	8.695 µs	52.187 µs

Values shown as N/A (Not Available) may be caused by one of the following reasons: not enough data was collected or the base rate and sub-rate 1 have been combined.

Task overrun occurs when a timer task does not complete before that same task is scheduled to run again. In such a case, the running task instance is allowed to complete and the following task instance is ignored. Depending on model settings, a task overrun may be reported or handled by a custom function.

Task turnaround time is the elapsed time between the start and finish of the task. If the task is not pre-empted, then the task turnaround time is equal to the task execution time. Task turnaround times may vary from one instance to another. Some typical reasons for task turnaround time variations are: preemption by higher-priority tasks, cache architecture effects, conditional execution paths, and waiting for an unavailable resource.

Task execution time is that part of the time between the task start and finish when the task is actually running and is not preempted by another task. Task execution time is equal to task turnaround time if the task is never preempted.

The task execution time cannot be measured directly, but is inferred from the task start and finish times and the intervening periods during which it was preempted by another task. In performing these calculations, processor time consumed by the scheduler while switching tasks is not considered. In cases where preemption has occurred, the reported task execution times overestimate the true values.

Task sample time is the time between two consecutive executions of the same task. The difference between the measured task sample time and the task sample time specified in the model may be caused by overrunning.

Literatur

Aho, Alfred et al.: Compilers Principle, Techniques and Tools, Second Edition. Boston: Pearson Addision Wesley, Januar 2007

Kovács, Károly Pál/Rácz, István: Transiente Vorgänge in Wechselstrommaschinen. Budapest: Verlag der Ungarischen Akademie der Wissenschaften, 1959

Lunze, Jan: Regelungstechnik 2; Mehrgrößensysteme, Digitale Regelung, 6. Auflage. Heidelberg: Springer-Verlag Berlin Heidelberg, 2010

MathWorks, Inc.: Real-Time Workshop for use with Simulink - Getting started, Version 6. Natick: The MathWorks, Inc., Juni 2004

MathWorks, Inc.: Real-Time Workshop Embedded Coder for use with Real-Time Workshop, Version 3. Natick: The MathWorks, Inc., Juli 2010

MathWorks, Inc.: Simulink 7,Developing S-Functions, Version 3. Natick: The MathWorks, Inc., April 2011a

MathWorks, Inc.: Simulink User's guide for Version 7.7. Natick: The MathWorks, Inc., April 2011b

Mohan, Ned: Field Orientated Control of 3-Phase AC-Motors, BPRA073. Northampton: TexasInstruments Europe, Inc., Februar 1998

Rieder, Ulf-Helmut: Optimierung der sensorlosen Regelung von permanenterregten Außenläufer-Synchronmaschinen. Wien: Institut für Elektrische Antriebe und Maschinen, TU Wien, Oktober 2005

Schröder, Dierk: Elektrische Antriebe; Regelung von Antriebssystemen. Heidelberg: Springer-Verlag Berlin Heidelberg, 2009

Schrödl, Manfred: Skriptum zur Lehrveranstaltung Elektrische Antriebe und Maschinen. Wien: Institut für Elektrische Antriebe und Maschinen, TU Wien, 1998

Schrödl, Manfred: Skriptum zur Lehrveranstaltung Drehstromantriebe mit Mikrorechnern. Wien: Institut für Elektrische Antriebe und Maschinen, TU Wien, 2000

Schrödl, Manfred/Hofer, Matthias/Staffler, Wolfgang: Combining INFORM method, voltage model and mechanical observer for sensorless control of PM synchronous motors in the whole speed range including standstill. e & i Elektrotechnik und Informationstechnik 5 2006

Staffler, Wolfgang: Sensorloser Extruderantrieb mit einer permanenterregten Synchronmaschine, Diplomarbeit. Wien: Institut für Elektrische Antriebe und Maschinen, TU Wien, März 2005

Stahl, Thomas et al.: Modellgetriebene Softwareentwicklung. Heidelberg: dpunkt.verlag GmbH, Mai 2007

TexasInstruments, Inc.: TMS320x280x Enhanced Pule Width Modulator (ePWM) Module, Reference Guide. Northampton: TexasInstruments, Inc., November 2004

Abbildungsverzeichnis

Tabellenverzeichnis