FACHHOCHSCHULE REGENSBURG
UNIVERSITY OF APPLIED SCIENCES
HOCHSCHULE FÜR
TECHNIK
WIRTSCHAFT
SOZIALWESEN
Fakultät Informatik

Diplomarbeit

Thema:

Codegenerierung mit Matlab/Simulink für Mikrocontroller und FPGAs

Verfasser: Martin Froschhammer
Erstprüfer: Prof. Dr. Richard Roth
Zweitprüfer: Prof. Dr. Hans-Jürgen Wagner
Betreuung: Prof. Dr. Richard Roth
Ausgabe: 15.09.2006
Abgabe: 14.02.2007

Erklärung

1. Mir ist bekannt, dass die Diplomarbeit als Prüfungsleistung in das Eigentum des Freistaats Bayern übergeht. Hiermit erkläre ich mein Einverständnis, dass die Fachhochschule Regensburg diese Prüfungsleistung die Studenten der Fachhochschule Regensburg einsehen lassen darf, und dass sie die Abschlussarbeit unter Nennung meines Namens als Urheber veröffentlichen darf.

2. Ich erkläre hiermit, dass ich diese Diplomarbeit selbständig verfasst, noch nicht anderweitig für andere Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benützt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

_____

Ort, Datum Unterschrift

Inhaltsverzeichnis

1 Einleitung

Es gibt heutzutage kaum mehr ein elektronisches Gerät, das nicht mit mindestens einem Mikrocontroller ausgestattet ist. Tatsächlich finden mehr als 95% der produzierten Prozessoren innerhalb von eingebetteten Systemen ihren Einsatz.
Eine wesentliche Gemeinsamkeit mit ihren großen Brüdern, den interaktiven Rechnersystemen wie PC oder Workstations haben sie. Um ihren Dienst zu verrichten benötigen sie Software.

Angesichts dieser Zahl kann man sich ausmalen, wie viel Aufwand für die Programmierung von Embedded-Systems aufgebracht wird, vor allem wenn man betrachtet wie kurz die Produktzyklen in der Elektronik geworden sind. Manche Geräte, gerade im „Consumer“-Bereich, sind nicht mal mehr ein Jahr im Handel erhältlich.
Und um die Nachfrage nach neuen Produkten zu schüren werden gleichzeitig immer neue Raffinessen in die Geräte eingebaut.
Bei großen Stückzahlen kommt noch hinzu, dass möglichst viel Funktionalität in die Software verlagert und möglichst wenig mit Hilfe elektronischer Bauelemente realisiert wird, denn diese verursachen Kosten, die bei jeder hergestellten Baugruppe neu anfallen. Software hingegen muss nur ein einziges Mal entwickelt werden.

Softwareentwickler stehen also vor einem Dilemma. Kürzere Entwicklungszeiten, mehr Funktionalität und Komplexität bei gleichen Anforderungen an Qualität und Robustheit des Systems.
Neben dem Massenmarkt für Elektronik müssen häufig eingebettete Systeme für Individuallösungen und Kleinserien, besonders für den industriellen Einsatz, entwickelt werden. Auch hier gilt, dass sich die Entwicklungszeiten aufgrund der Marktgegebenheiten ständig verkürzen.
Gerade in diesem Bereich werden häufig programmierbare Logikbausteine eingesetzt, vor allem wenn zeitkritische Anwendungen realisiert werden sollen.
Einen flexiblen Ansatz bieten hierzu Field Programmable Gate Arrays kurz FPGAs. Auf ihnen lassen sich komplexe logische Schaltungen erstellen und vor Ort programmieren.

Gehen die Stückzahlen über die von Kleinserien hinaus, können die entwickelten Schaltungen von FPGA-Prototypen auf kostengünstigere Application Specific Integrated Circuits kurz ASICs übertragen werden.

Um vor dem Hintergrund dieser komplexen Technologien produktive Entwicklungsarbeit leisten zu können, bedarf es moderner Methoden der Hard- und Softwareentwicklung.

Diese Diplomarbeit beschäftigt mit einer dieser Methoden, dem so genannten Model-based-Design, genauer gesagt mit der Codegenerierung aus einem simulierbaren graphischen Modell heraus.
Dazu kommt die Software Matlab/Simulink der Firma The Mathworks, dem derzeit führendem Hersteller auf diesem Gebiet, zum Einsatz.
Anhand eines Regelkreises, der aus einem Hubmagneten besteht, dessen Ankerposition regelungstechnisch einstellbar ist, soll der Entwicklungsprozess vom Entwurf bis zu Implementierung gezeigt werden.
Letztere erfolgt hierbei sowohl auf dem Mikrocontroller XC167 von Infineon, als auch auf dem programmierbaren Logikbaustein Spartan-3E von Xilinx.

2 Überblick

Dieses Kapitel umreißt den Aufbau der Diplomarbeit. Im ersten Abschnitt wird der Kontext der Diplomarbeit zur e-Learning-Plattform hergestellt. Der zweite Teil geht auf den Aufbau des Dokuments sowie auf die Vorgehensweise bei der Behandlung des Diplomarbeitsthemas ein.

2.1 Das Remote-Labor

Die Beispielanwendung eines Regelkreises, die aus der Diplomarbeit hervorgeht, ist Teil einer neuen e-Learning-Plattform für Hard- und Softwaredesign.
Da es sich nicht um eine rein virtuelle Lernplattform handelt, sondern auf den Entwicklungsrechner sowie die angeschlossene Hardware über Internetverbindung entfernt zugegriffen werden kann, wird die e-Learningplattform im Folgenden als Remote-Labor bezeichnet.
Zielsetzung des Projekts ist es, einer großen Zahl von Studenten eine Experimentierumgebung zur Verfügung zu stellen, unabhängig von Vorlesungs- und Laboröffnungszeiten. Es soll dabei aktuellen Entwicklungen im Embedded Systems Bereich folgen. Diese sind beispielsweise rechnergestützte Logikentwicklung mit VHDL, Kommunikation über Bussysteme wie FlexRay, CAN und LIN, sowie moderne Methoden der Softwareentwicklung wie etwa Modellgetriebene Architektur (MDA) und Codegenerierung mit Matlab/Simulink. Wobei letzteres das eigentliche Thema dieser Diplomarbeit ist.

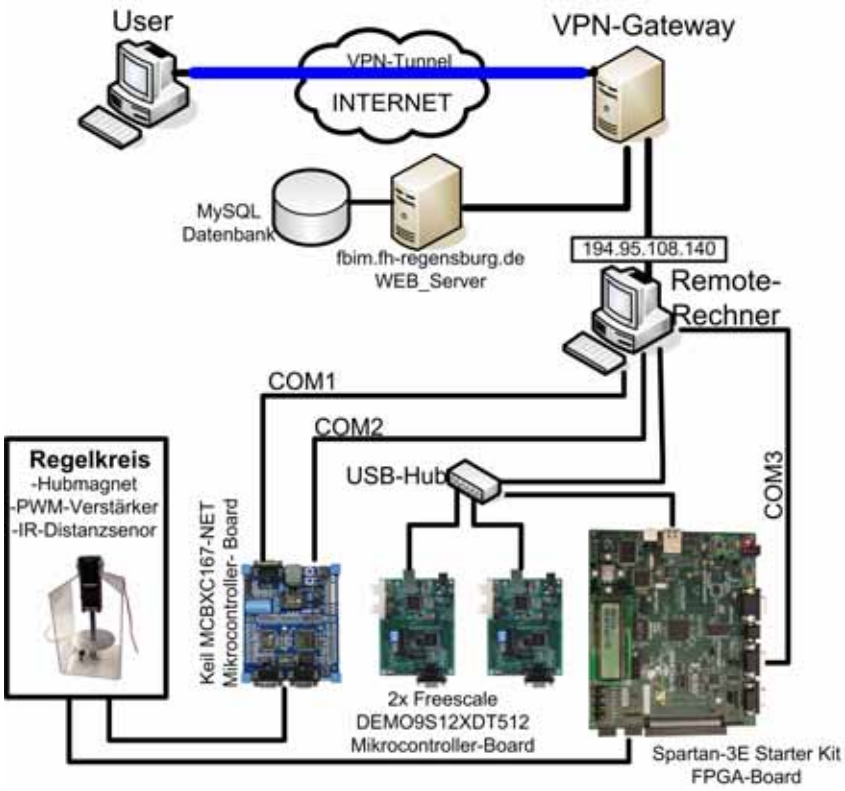


Abbildung 2.1: Laboraufbau

Das Remote-Labor besteht aus einem handelsüblichen Desktop-PC, der über Internet mit der Außenwelt verbunden ist. Dieser Rechner ist via Remote Desktop Verbindung, welche von Windows XP bereitgestellt wird, von einem entfernten Rechner steuerbar (Abbildung 2.1).
An das Labor angeschlossen sind mehrere Entwicklerboards. Dies sind zwei Freescale DEMO9S12XDT512-Boards mit Freescale MC9S12X -Mikrocontrollern. Darüber hinaus ist das Labor mit einem Keil MCBXC167-NET Board, sowie ein Xilinx Spartan-3E FPGA-Board bestückt (Abbildung 2.2).

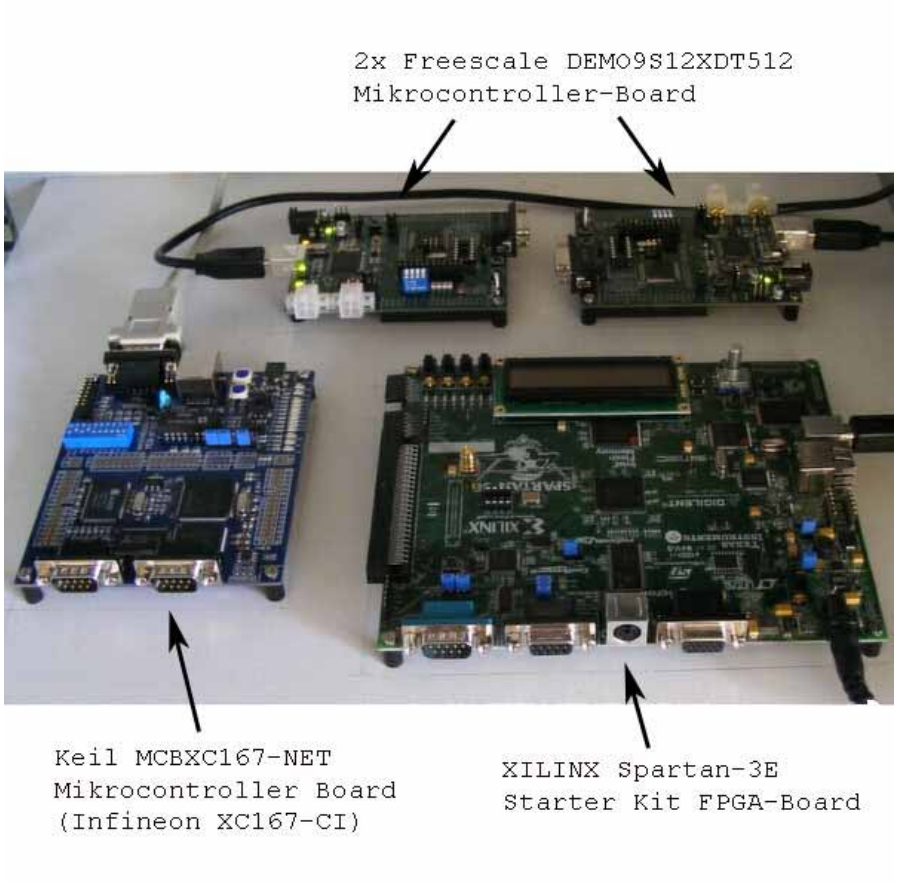


Abbildung 2.2: Hardware des Remote-Labors[1]

Freescale DEMO9S12XDT512:

Das Herz des DEMO9S12XDT512-Entwicklerboards bildet der 16-Bit Mikrocontroller MC9S12XDT512 von Freescale.
Eine Besonderheit des MC9S12XDT512 ist der XGATE Koprozessor, der separat programmiert, für den Datentransfer zwischen der Peripherie sorgt, ohne den Prozessor zu belasten. Laut eigenen Angaben von Freescale handelt es sich beim MC9S12XDT512 um einen Allround Mikrocontroller. Er erfreut sich aber im Automotive-Umfeld, nicht zuletzt wegen des geringen Preises, der niedrigen Leistungsaufnahme und den CAN und LIN Schnittstellen, großer Beliebtheit. Als Entwicklungsumgebung dient das Programm CodeWarrior 4.1 für HCS12X.

Technische Daten:

Mikrocontroller:	Freescale MC9S12XDT512 mit X-GATE Koprozessor , 80MHz
Speicher:	512 KB Flash EEPROM, 4KB EEPROM, 32 KB SRAM
Schnittstellten, Peripherie:	59 I/O Ports, SCI und SPI Kommunikationsports, Key Wake-up Port, 2 AD Wandler mit 8 und 16 Kanälen, LIN und CAN 2.0 Schnittstelle, 4 Dip-Schalter, 3 Drucktaster, 4 LEDs, Lichtsensor, Potentiometer, USB Anschluss zum Programmieren und Debuggen

Keil MCBX167-NET:

Auf dem MCBX167-NET Entwicklerboard befindet sich der XC167-CI 16-Bit Mikrocontroller von Infineon. Dabei handelt es sich um eine Weiterentwicklung der verbreiteten C16x-Reihe des gleichen Herstellters.
Die wesentlichen Neuerungen dieses Mikrocontrollers sind die höhere Taktrate, beschleunigte mathematische Operationen, Multiply-and-Accumulate – Befehle (MAC) für Signalverarbeitung, sowie ein neu organisierter Block zur Pulsweitenmodulation.
Das Haupteinsatzgebiet des XC167 sind Industrie- und Automotive-Anwendungen.
Die Programmierung des Systems erfolgt über die Entwicklungsumgebung µVision 3 der Firma Keil.

Technische Daten:

Mikrocontroller:		Infineon XC167-CI, 40 MHz
Speicher:	On-Chip:	RAM 2K, Flash 128K
	On-Board:	RAM 512K, Flash 2048K
Schnittstellen, Peripherie:		2 Drucktaster, 8 LEDs an I/O Ports, RS-232 Schnittstelle 2 CAN Ports, LIN, 10/100 Mbit Ethernet Schnittstelle, 10-Bit AD-Wandler mit 16 Kanälen.

Xilinx Spartan-3E Starter Kit:

Das Spartan-3E Starter Kit ist vielseitiges Entwicklerboard für programmierbare Logik. Auf ihm befinden sich ein Spartan 3-E FPGA sowie ein CoolRunner™-II CPLD von Xilinx.
Auf der Leiterplatte befindet sich eine umfangreiche Peripherie, die mit dem Spartan-3E angesteuert werden kann.
Schaltungen für das FPGA lassen sich mit der Software ISE von Xilinx entwerfen und Synthetisieren. Das Aufspielen des Programms erfolgt über USB-Schnittstelle.

Technische Daten:

Programmierbare Bausteine:	FPGA Spartan-3E (XC3S500E-4FG320C) CPLD CoolRunner™-II (XC2C64A-5VQ44C)
Speicher:	128 Mbit Parallel Flash, 16 Mbit SPI Flash, 64 MByte DDR SDRAM
Schnittstellen, Peripherie:	Ethernet 10/100Mbit, JTAG USB Port, 2x 9-pin RS-232, PS/2 Maus-/Tastatureingang, 4 Schiebeschalter, 8 LEDs, 4 Drucktaster, 100-Pin Hirose Erweiterungsstecker, 3x 6-Pin Pfostenstecker, LCD Display 2x16 Zeichen

2.2 Codegenerierung mit Matlab Simulink

Dieser Abschnitt gibt einen Überblick des eigentlichen Thema der Diplomarbeit, der Codegenerierung mit Matlab/Simulink für Mikrocontroller und FPGA. Anhand eines Reglerkreises wird exemplarisch ein Entwicklungsprozess, angefangen mit der Modellbildung über die Codegenerierung bis hin zur Implementierung in eine Mikrocontroller- oder FPGA-Umgebung, gezeigt (Abbildung 2.3).

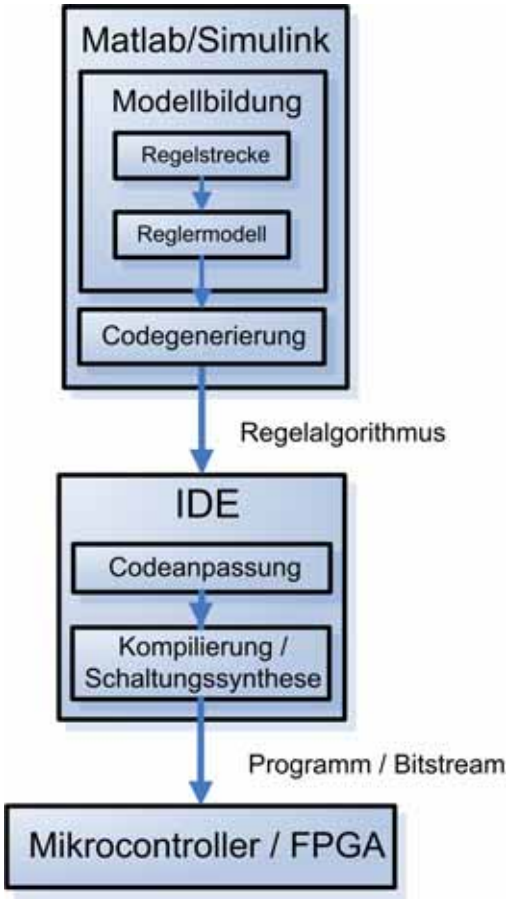


Abbildung 2.3: Entwicklungsprozess

In der Diplomarbeit wird als erstes auf den Regelkreis im Allgemeinen eingegangen, gefolgt vom speziellen Fall eines Regelkreises für einen Hubmagneten, dessen Ankerposition durch einen Digitalregler eingestellt werden kann. Bei dem Hubmagneten handelt es sich dabei um einen realen Versuchsaufbau im Remote-Labor, für den der Regler entworfen wird.

Es folgt ein Überblick über Matlab Simulink, dessen Funktionsweise am Beispiel der Modellierung einer Regelstrecke gezeigt wird.
Das Modell ist dabei ausgehend von physikalischen Gesetzmäßigkeiten hergeleitet.
Durch Ergänzung dieses Modells mit dem eines Reglers entsteht ein geschlossener Regelkreis.

Die so erstellte Nachbildung eines Reglers dient im Folgenden als Basis für weitere Reglermodelle aus denen sich Programmcode für Mikrocontroller und HDL-Code für FPGAs generieren lässt. HDL steht dabei für Hardware Description Language und ist ein Oberbegriff für verschiedene Hardwarebeschreibungssprachen.
Die Entwicklung des Reglers erfolgt schrittweise von einem einfachen virtuellen Analogregler, der noch keine Randbedingungen wie Messgrößen oder Steuergrößen berücksichtigt, bis zum endgültigen Regler, der bereits die Schnittstellen zur Hardware beinhaltet.

Insgesamt werden drei Reglerimplementierungen vorgestellt, die sich, obwohl sie auf dem gleichen Regelalgorithmus beruhen, zum Teil erheblich unterscheiden.

Den Anfang bildet die Reglerimplementierung auf dem XC167-Mikrocontroller von Infineon. Das Reglermodell muss hierfür um Elemente erweitert werden, die benutzerdefinierten, hardwarespezifischen Code zulassen. Die Basissoftware, in die der generierte Code eingebettet wird, ist mit der Software DAvE von Infineon erstellt worden.

Die beiden anderen Reglerimplementierungen haben das Spartan-3E FPGA als Ziel.
Beim ersten erfolgt die Codierung mit dem HDL-Coder in Simulink. Das Reglermodell muss hierfür auf Festkommaarithmetik umgestellt werden. Des Weiteren ist es nötig einige nicht unterstützte Elemente zu ersetzten.

Die letzte Implementierung des Reglers erfolgt mit dem System Generator von Xilinx. Dieser erweitert Simulink um Modellblöcke die speziell für die HDL-Codegenerierung ausgelegt sind. Um einen Regler zu erstellen, ist es notwendig den gesamten Regler mit diesen Blöcken nachzubilden.
Der Vorteil zum HDL-Coder ist der, dass mit dem System Generator direkt ein Modell erstellt werden kann, das ohne manuelle Codeanpassung synthetisiert werden und auf das FPGA übertragen werden kann.

Den Abschluss der Diplomarbeit bildet eine kurze Zusammenfassung.

3 Der Regelkreis

In diesem Kapitel werden die Grundlagen der Regelungstechnik kurz umrissen. Diese sind für das Verständnis der folgenden Kapitel notwendig. Beginnend mit dem Aufbau eines Regelkreises im Allgemeinen und dessen Komponenten, werden einzelne Reglertypen näher beleuchtet um schließlich den auf den Regelkreise des Hubmagneten, der in der Diplomarbeit ungesetzt wird, näher einzugehen.

3.1 Aufbau eines Regelkreises

Als Regelkreis wird ein geschlossener Wirkkreis bezeichnet, der durch fortlaufende Anpassung einer Stellgröße eine Regelgröße so ändert, dass sie eine möglichst geringe Differenz zu einem vorgegebenen Sollwert aufweist.
Regelkreise können in zwei Teile zerlegt werden, den Regler und die Regelstrecke. Letztere ist hierbei das System, das geregelt wird. Eingangsgröße des Regelkreises ist die Führungsgröße oder auch Sollwert. Ist die Führungsgröße konstant, wird der Regelkreis als Festwertregelung bezeichnet, ansonsten heißt er Folgeregelung. Führungsgrößen können z.B. Temperatur, Abstand oder Drehzahl sein. Ausgangsgröße des Regelkreises ist die Regelgröße, was als die zu regelnde Größe zu interpretieren ist. Zudem können noch Störgrößen auf unerwünschte Weise auf den Regelkreis einwirken. Abbildung 3.1 zeigt die Grundstruktur eines Regelkreises.

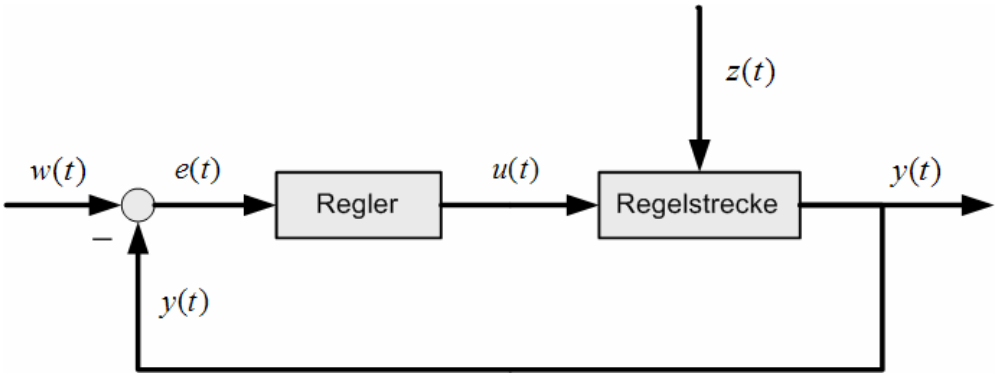


Abbildung 3.1: Grundstruktur eines Regelkreises

Bezeichnungen:

$w(t)$:= Führungsgröße, Sollwert
$y(t)$:= Regelgröße, Istwert
$u(t)$:= Stellgröße
$e(t)$:= Regeldifferenz $e(t) = w(t) - y(t)$
$z(t)$:= Störgröße

Die obige Darstellung gilt für Regelkreise allgemein. Es gibt auch Regelkreise mit mehreren Regelgrößen, die, wenn sie miteinander im Zusammenhang stehen, Mehrgrößenregler genannt werden. Im Rahmen der Diplomarbeit wird nur eine Eingrößenregelung behandelt.

Die Funktionsweise eines Regelkreises ist folgende: Zwischen dem vorgegebenen Sollwert und dem Istwert wird eine Differenz gebildet. Diese Regeldifferenz liegt am Eingang des Reglers, der daraus eine Stellgröße ermittelt, die die Regeldifferenz verkleinern soll. Die neue Stellgröße wiederum wirkt auf die Regelstrecke ein, was eine Änderung der Regelgröße, also des Istwerts zur Folge hat. Im optimalen Fall nähert sich der Istwert dem Sollwert schnell an bis er identisch ist und bleibt dort stabil.

Abbildung 3.1 ist eine vereinfachte Darstellung eines Regelkreises. Tatsächlich muss für jede elektrische oder elektronische Regelung die Regelgröße zuerst in ein elektrisches Signal, das Regelsignal $\tilde{y}(t)$, umgewandelt werden, bevor eine Regeldifferenz zum Sollwert ermittelt wird. Auch die Stellgröße wirkt nicht direkt auf die Regelstrecke ein. Meist ist verstärkendes oder energiewandelndes Element zwischengeschaltet, das als Stellglied bezeichnet wird. Der Ausgang des Reglers ist dann nicht mehr die Stellgröße, sondern das Stellsignal $\tilde{u}(t)$. Abbildung 3.2 wird der Praxis eher gerecht.

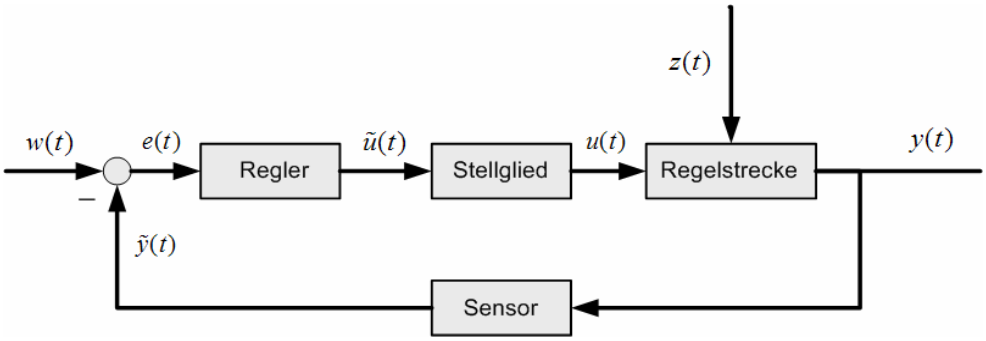


Abbildung 3.2: Erweiterte Grundstruktur des Regelkreises

Diese Darstellung gilt für Analog- wie für Digitalregler gleichermaßen. Analogregler zeichnen sich dadurch aus, dass sie zeit- und wertkontinuierlich arbeiten. Sie können durch eine analoge elektronische Schaltung, beispielsweise mit Operationsverstärkern, aber auch mechanisch realisiert werden. Das Kernstück eines digitalen Reglers hingegen bildet ein Digitalrechner, der den Regelalgorithmus enthält. Dieser kann durch einen PC oder wie in diesem Projekt durch einen Mikrocontroller bzw. programmierbare Logik realisiert sein. Der Digitalregler (Abbildung 3.3) unterscheidet sich vom Analogregler dadurch, dass der Ist-Wert der Regelung nicht kontinuierlich erfasst, sondern in konstanten zeitlichen Abständen gemessen wird. Die Frequenz dieser Messungen wird als Abtastrate bezeichnet. Zudem müssen Messwerte zuerst mit Hilfe eines Analog-Digital-Wandlers in ein für einen Digitalrechner verwertbares Format gebracht werden. Des Weiteren muss auch das, in digitaler Form vorliegende, Stellsignal wieder zu einem Analogwert rückgewandelt werden.
Die Abtastrate hängt im Wesentlichen von den verwendeten Sensoren, der Wandlungszeit der AD-Wandler der Berechnungsdauer des Regelalgorithmus, sowie den Erfordernissen der Regelstrecke selbst ab.

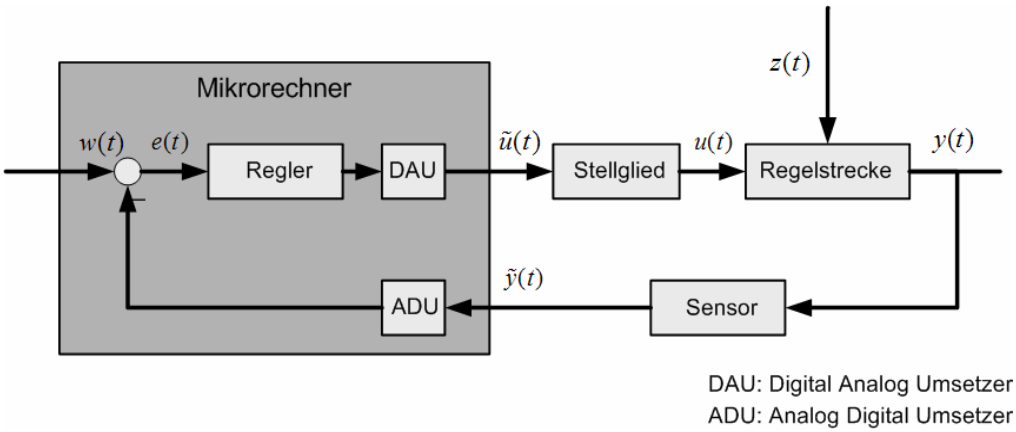


Abbildung 3.3: Digitaler Regelkreis

Der grundlegende Aufbau des Digitalreglers kann nun auf das Projekt der Diplomarbeit, der Regelung eines Hubmagneten, übertragen werden.

Die Regelstrecke bildet der Hubmagnet. Die Hubhöhe ist die Regelgröße. Störgrößen können beispielsweise zusätzliche Belastung des Ankers sein oder Magnetisierungseffekte des Eisenkerns.
Die momentane Hubhöhe wird über einen Infrarot Distanzsensor ermittelt. Als Regler dient wahlweise ein Mikrocontroller oder ein FPGA. Die Regelalgorithmen, die sich in diesen Digitalreglern befinden, werden mit Hilfe von Matlab/Simulink erstellt. Die Sollwertvorgabe w(t) erhält der Regler über die serielle Schnittstelle von einem PC. Das Stellsignal $\tilde{u}(t)$ ist ein pulsweitenmoduliertes Signal, dessen Pulsweite proportional zur Stellgröße $u(t)$ ist. Als Stellglied kommt ein PWM-Leistungsverstärker zum Einsatz. Die einzelnen Komponenten werden im nächsten Unterabschnitt näher beschrieben. Abbildung 3.4 zeigt schematisch den Regelkreis des Diplomarbeitsprojekts.

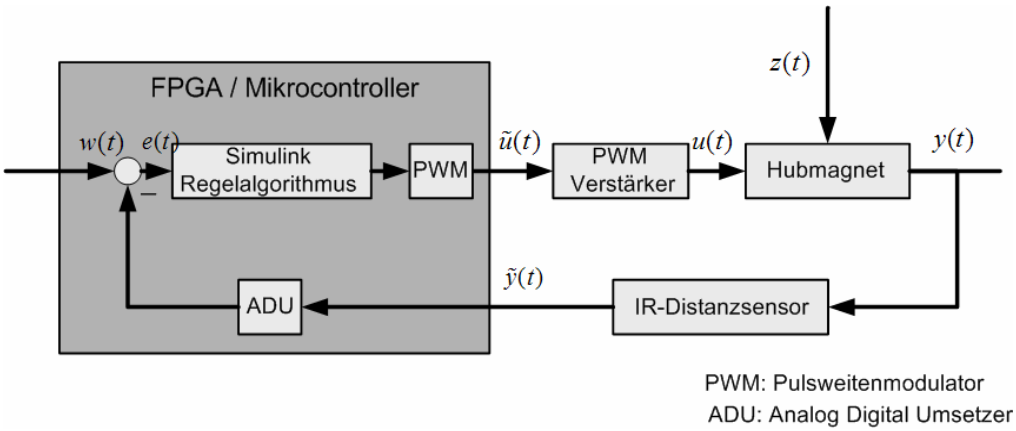


Abbildung 3.4: Regelkreis des Hubmagneten

3.2 Komponenten des Regelkreises

3.2.1 Der Hubmagnet

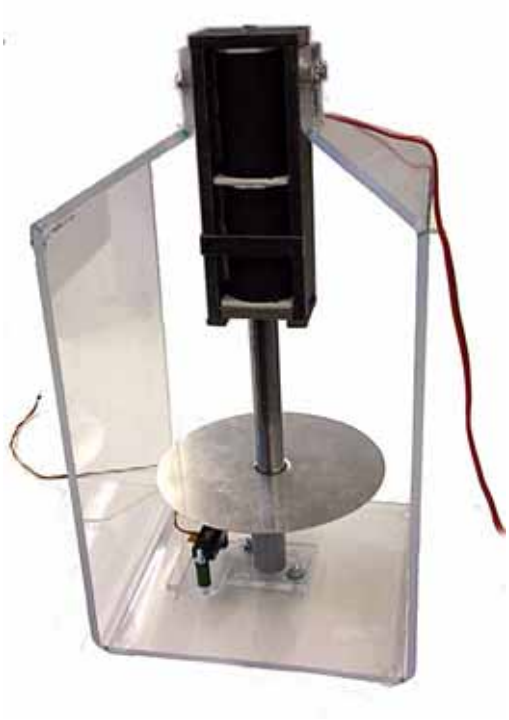


Abbildung 3.5: Hubmagnet

Die Regelstrecke des Versuchsaufbaus bildet ein eigens hierfür konstruierter Hubmagnet, dessen Anker, das ist der bewegliche Teil des Hubmagneten, entgegen der Schwerkraft auf einer vorgegebenen Höhe gehalten werden soll.
Der Hubmagnet besteht aus zwei aufeinander stehenden, parallel geschalteten Spulen mit jeweils 1900 Windungen. Die beiden Spulen sind von einem Metallrahmen umgeben, der die magnetischen Feldlinien des Hubmagneten bündelt. Im innern der Spulen befindet sich der Anker, der in seiner Längsachse beweglich ist. Am Anker ist eine Scheibe befestigt, die der Messsensorik als Bezugsfläche dient.
Die Funktionsweise ist folgende: Wird eine Spannung an die Spulen angelegt, fließt ein entsprechender Strom. Das dadurch erzeugte Magnetfeld zieht den Anker ins Spuleninnere. Dadurch verkürzt sich der Luftspalt, wodurch die Kraft auf den Anker erhöht wird. Es ist ersichtlich, dass, wenn der Anker auf einer bestimmten Höhe gehalten werden soll, regulative Maßnahmen notwendig sind, da es sich um ein labiles Gleichgewicht handelt.

3.2.2 Infrarot Distanzsensor

Zur Messung der momentanen Hubhöhe des Ankers kommt ein Infrarot Distanzsensor zum Einsatz. Das Funktionsprinzip besteht darin, das ein Infrarot LED einen Strahl zu dem vor ihm stehenden Objekt sendet. Dieses reflektiert ihn und trifft, zurück beim Sensor, auf einen Position Sensetive Detector kurz PSD.
In Abhängigkeit von der Entfernung des Objekts ändert sich der Auftreffpunkt im PSD (Abbildung 3.6). Dieser und die Auswertelektronik wandeln die Position des ankommenden Strahls in einen analogen Spannungswert um. Diese Art der Entfernungsmessung ist weitgehend unabhängig von den Reflektionseigenschaften und der Farbe des Objekts.

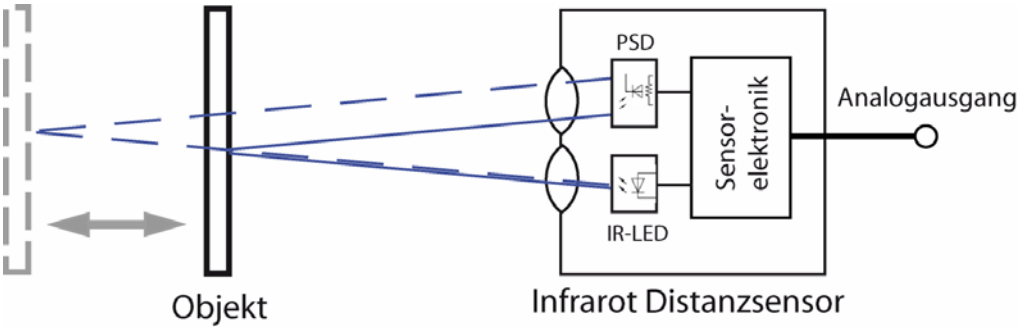


Abbildung 3.6: Funktionsprinzip eines Infrarot Distanzsensors

Im Versuchsaufbau wird der GP2D120 Infrarot Distanzsensor der Firma SHARP verwendet (Abbildung 3.7.), der senkrecht zur am Anker befestigten Scheibe steht.

Abbildung 3.7: SHARP GP2D120[2]

Es handelt sich dabei um einen Typen, der für den Einsatz im Nahbereich, 4-30 cm, konzipiert wurde. Der Messwert liegt am Ausgang des IR-Sensors als analoge Spannung vor. Der Messvorgang dauert 38,2 ms ± 9.6 ms und liegt 5 ms später als Spannungswert am Ausgang des IR-Sensors an. Dieser wird solange gehalten, bis ein neuer Messwert ermittelt wurde. Abbildung 3.8 zeigt dabei die Ausgangsspannung in Abhängigkeit von der Objektdistanz.

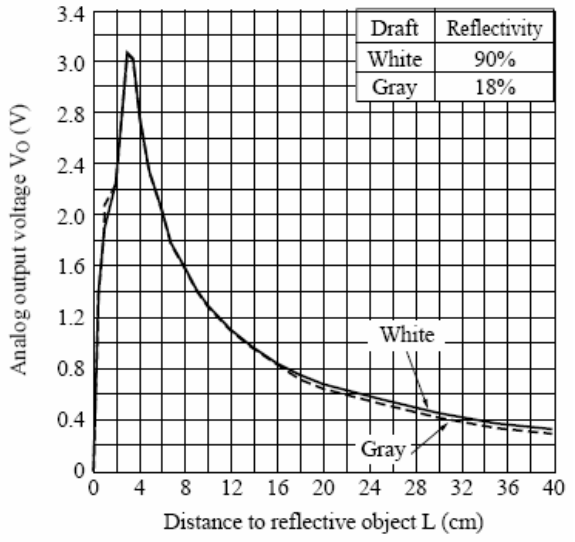


Abbildung 3.8: Ausgangspannung in Abhängigkeit von der Objektdistanz[2]

Der mathematische Zusammenhang zwischen Spannung U und Abstand x ist hierbei [3],

$$x = \frac{Y_1}{U - Y_2}; \quad \text{für x >= 4 cm.}$$

x := Abstand [m]
U:= Spannung am Ausgang [V]
Y_1:= Steigung [Vm]
Y_2:= Offset [V]

Die beiden Parameter Y_1 und Y_2 werden wie folgt ermittelt:

$$Y_1 = \frac{(U_2 - U_1)x_2 x_1}{x_2 - x_1};$$

$$Y_2 = \frac{U_2 x_2 - U_1 x_2}{(x_2 - x_1)};$$

U_1, U_2 := Spannungen aus Probemessungen
x_1, x_2 := Messdistanzen

Die dazu notwendigen Wertepaare aus Spannung und Entfernung wurden direkt am Versuchsaufbau ermittelt. Die Umwandlung ist notwendig, um den Sensorwert, der den Istwert in der Regelung repräsentiert, mit dem Sollwert vergleichen zu können und daraus eine Regeldifferenz zu bilden.
Es wurden die Werte $Y_1 = 0{,}10151\text{Vm}$ und $Y_2 = 0{,}15310\text{V}$ ermittelt.
Abbildung 3.9 zeigt die Graphen für berechnete und gemessene Spannung bei gleicher Objektdistanz.

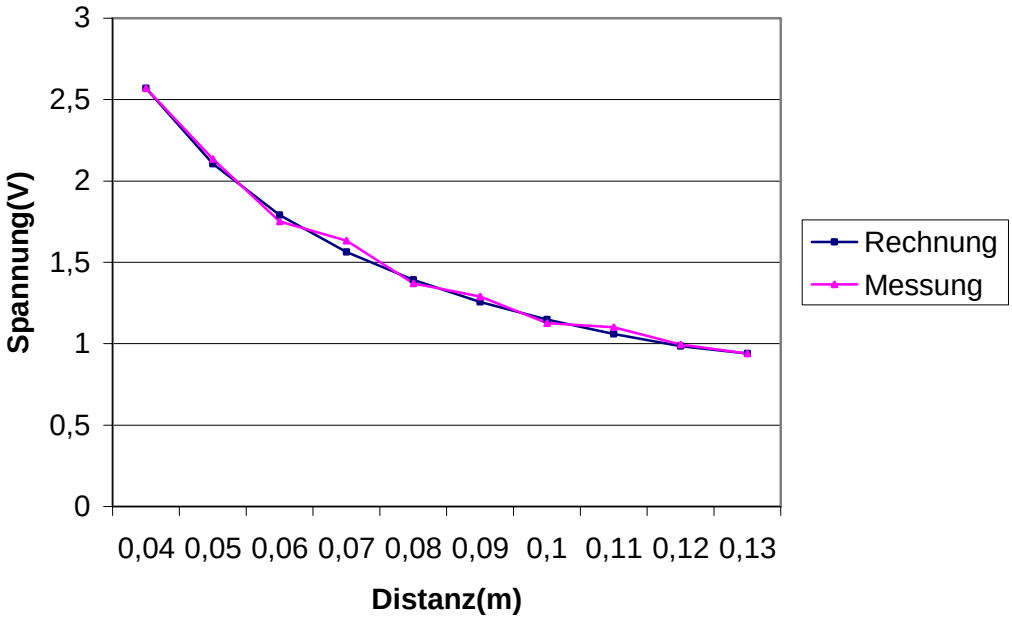


Abbildung 3.9: Spannung in Abhängigkeit der Entfernung

3.2.3 PWM-Verstärker

Die Ansteuerung des Hubmagneten erfolgt über ein Pulsweitenmoduliertes Signal. Da aber die Ausgänge von Mikrocontroller als auch des FPGA nur Signale mit geringer Leistung liefern, müssen diese verstärkt werden.

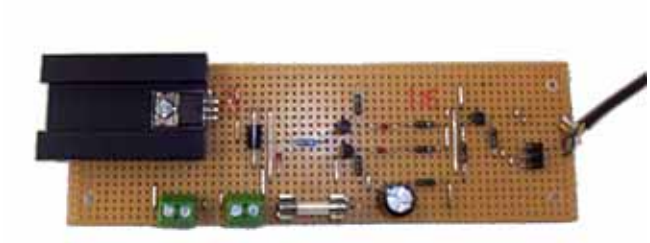


Abbildung 3.10: PWM-Verstärker

Vorgaben für die Schaltung des PWM Verstärkers sind folgende:

- Eingang für PWM-Eingang sowohl für Keil Mikrocontroller-Board als auch für Spartan-3E Starterkit
- Arbeitsfrequenz von 3 bis 6 kHz.
- Spannungsverstärkung, proportional zur Pulsbreite des PWM-Eingangs.
- Betriebsspannung 12 V, Ausgangsstrom min. 1,8A bei Induktiver Last.
- Neben der Betriebsspannung und dem eingehenden PWM Signal soll keine weitere Spannungsversorgung benötigt werden.
- Änderungen im Eingangssignal sollen mit minimaler Verzögerung an den Ausgang weitergereicht werden.

Es wurde eine Schaltung entwickelt, die auch am Verstärkerausgang mit Pulsweitenmodulation arbeitet. Abbildung 3.10 zeigt diese Schaltung. Teile der Schaltung wurden aus einem Bausatz zur Pulsweitenmodulation der Firma Conrad übernommen.

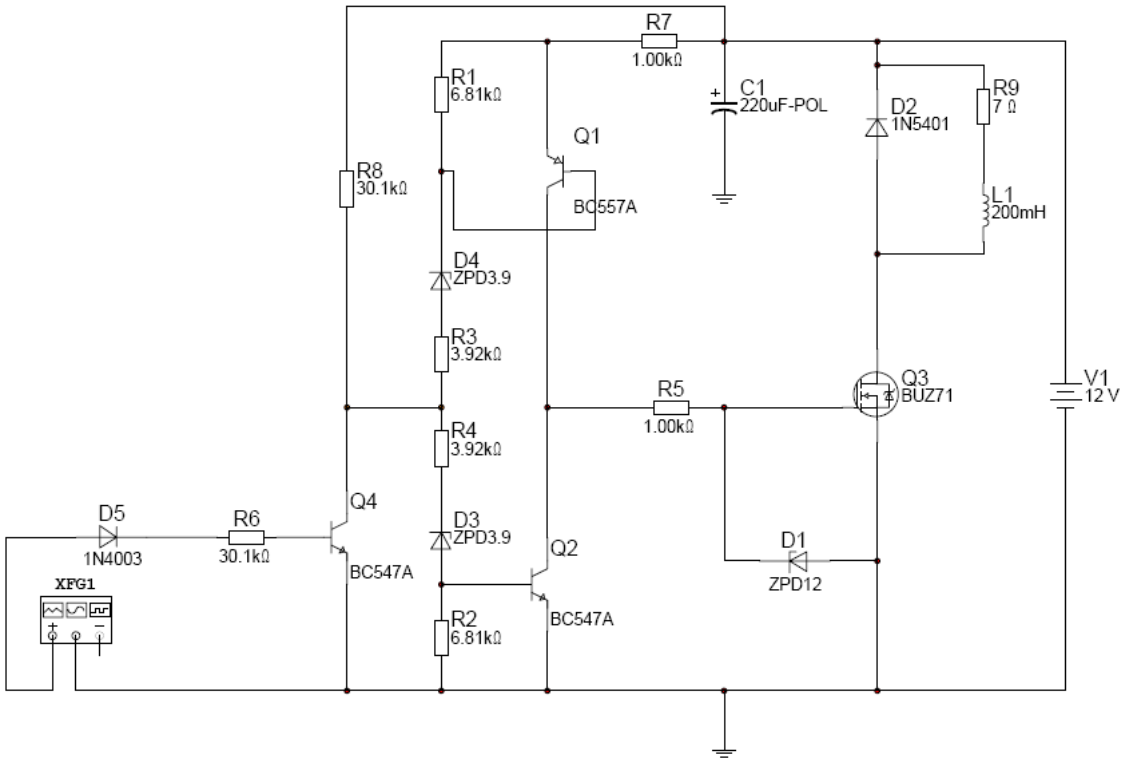


Abbildung 3.11: PWM-Verstärkerschaltung

Schaltungsbeschreibung:

Der Widerstand R9 und die Induktivität L1 stellen das Ersatzschaltbild des Hubmagneten dar. Parallel dazu ist eine Freilaufdiode geschalten, die die Selbstinduktionsspannungen bei den Schaltvorgängen der Pulsweitenmodulation kurzschließt. In Reihe zur Last befindet sich ein MOSFET (Metal Oxide Semiconductor Feld Effekt Transistor). Dieser führt die eigentliche Pulsweitenmodulation im Leistungsteil der Schaltung durch. Im Gegensatz zu einem Bi-Polar Transistor, ist dieser Transistortyp nicht Strom- sondern Spannungsgesteuert. Im voll durchgeschalteten Zustand beträgt der Widerstand des verwendeten Typen BUZ71 nur 0,14 Ohm, weshalb er sich hervorragend als Leistungsschalter eignet.
Um den MOSFET optimal anzusteuern werden 2 Bipolar Transistoren Q1 und Q2 vorgeschaltet. Diese arbeiten als Schalter und ziehen das Gate des MOSFET entweder auf +12V wenn Q1 durchgeschaltet ist oder auf 0V wenn Q2 durchgeschaltet ist. Liegt kein Signal oder 0V am PWM-Eingang an, so ist standardmäßig Q2 durchgeschaltet. Das Gate des MOSFET liegt an 0V, wodurch er sperrt.
Zur Pegelanpassung kommt ein weiterer Transistor Q4 zum Einsatz. Er dient zum einen dazu, dass die Schaltung sowohl mit dem 5V-Signal des XC167 Mikrocontroller, als auch mit dem 3,3V-Pegel des Spartan-3E angesteuert werden kann. Zum anderen sorgt er dafür, dass die Schaltung nicht invertiert angesteuert werden muss. Liegt ein High-Pegel am PWM-Eingang schaltet Q4 durch. Der Längszweig wird auf 0V gezogen und Q1 schaltet ebenfalls durch, so dass das Gate des MOSFET auf 12V liegt, woraufhin dieser durchlässt.

3.2.4 Keil MCBXC167 Entwickler Board

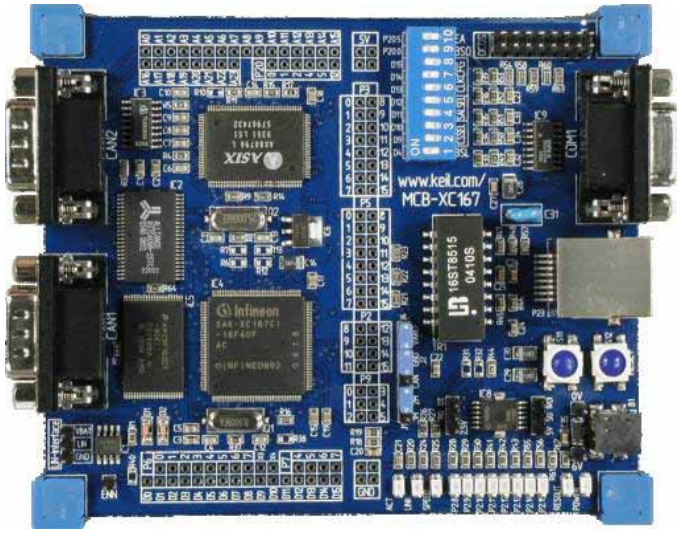


Abbildung 3.12: Keil MCBXC167 Entwickler Board

Der digitale Regler wird im Zuge der Diplomarbeit sowohl auf einem Mikrocontroller als auch auf einem FPGA realisiert. Als Mikrocontroller kommt der XC167-CI, der sich auf dem Keil MCBXC167 Entwicklerboard befindet, zum Einsatz. Die Eckdaten dieses Boards wurden bereist in Abschnitt 2.2 erläutert.

Eine auf das Keil-Board zugeschnittene Entwicklungsumgebung steht mit dem Programm Keil µVision 3 zur Verfügung. µVision stellt einen Codeeditor, Projektnavigation und Compiler für die C166-Architektur und deren Derivate zur Verfügung. Die erstellten Programme für den XC167 Mikrocontroller können bereits am PC simuliert werden. Um Programme auf den Mikrocontroller zum Ablauf zu bringen, müssen sie zuerst auf diesen übertragen werden. Dies kann über die serielle Schnittstelle aus der Entwicklungsumgebung heraus erfolgen oder über das externe Programm Memtool von Infineon, welches das kompilierte Programm auf den internen Flashspeicher des Mikrocontrollers schreibt. Zudem besteht die Möglichkeit, das Programm während der Ausführung auf dem Mikrocontroller zu debuggen.

Die benutzte Peripherie des Mikrocontrollers für die Regelungsanwendung ist der integrierte Analog-Digital-Wandler zum Erfassen der Messwerte, ein Timer, der den Regelalgorithmus zyklisch aufruft. Des Weiteren findet die Capture Compare Unit als PWM Modul ihren Einsatz. Die Führungsgröße und andere Regelparameter werden über den seriellen Port an den Mikrocontroller übertragen. Darüber hinaus werden noch einige Ausgabepins benutzt, die den Status über LEDs ausgeben.

3.2.5 Xilinx Spartan 3E Starterkit

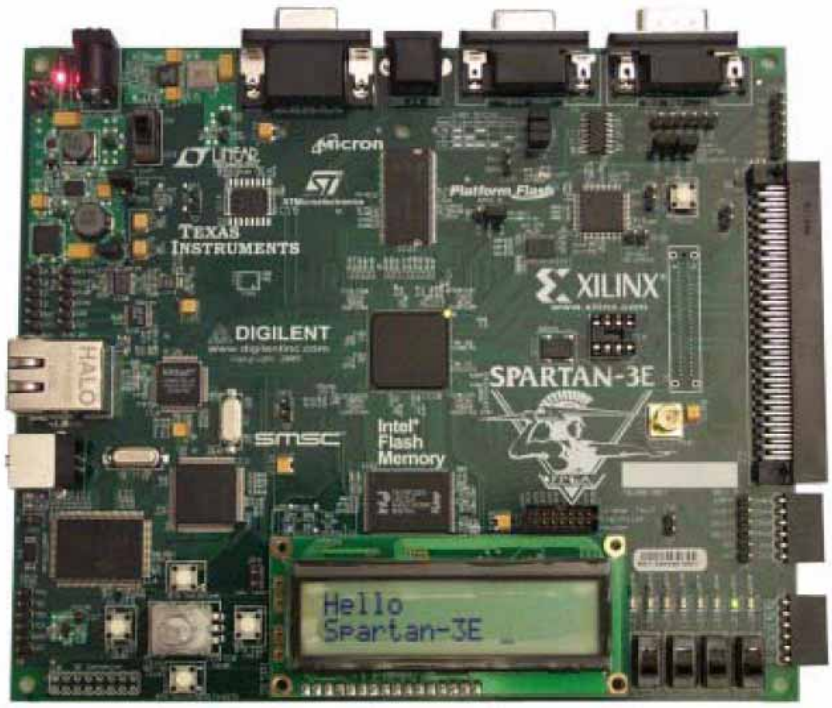


Abbildung 3.13: Xilinx Spartan-3E Starterkit[7]

Die FPGA-Implementierung des Reglers erfolgt auf dem Spartan-3E Starter Kit von Xilinx. Zur Erstellung des Regelalgorithmus wird wiederum Matlab/Simulink verwendet. Der Regler für das FPGA wird in zwei Varianten erstellt. Zum einen mit dem System Generator von Xilinx, welcher Simulink um Funktionen zur Generierung von herstellerspezifischem HDL-Code erweitert.
Zum anderen wird die HDL-Coder Toolbox für Simulink verwendet um generischen VHDL-Code zu erzeugen.
Die Schnittstellen für AD-Wandler, serielle Schnittstelle und PWM sind für beide Implementierungen als VHDL-Module programmiert. Dazu wurde die Software ISE 8.1 von Xilinx verwendet. Dieses Programm führt auch die Schaltungssynthese des von Simulink generierten HDL-Codes durch und sorgt für die Übertragung des erzeugten Bitstreams auf das FPGA-Board. Der Bitstream konfiguriert dabei den FPGA mit der gewünschten Logik.

Bis auf die serielle Schnittstelle und den AD-Wandler mit dem zugehörigen Vorverstärker, die beide über Serial Pheripheral Interface (SPI) angesprochen werden, wird keine weitere externe Hardware des Entwickler-Boards benutzt.

4 Modellbildung mit Matlab Simulink

Die Simulation ist nach VDI-Richtlinie 3633 wie folgt definiert:
„Simulation ist das Nachbilden eines Systems mit seinen dynamischen Prozessen in einem experimentierfähigen Modell, um zu Erkenntnissen zu gelangen, die auf die Wirklichkeit übertragbar sind.“[4]
Mit dynamischen Prozessen sind Prozesse gemeint, deren Ausgangsgröße sich im zeitlichen Verlauf ändert. Die Nachbildung solcher Prozesse wird als Modellbildung bezeichnet. Diese kann theoretisch oder experimentell erfolgen. Bei der theoretischen Modellbildung kann man dabei auf physikalische Gesetzmäßigkeiten zurückgreifen.
Als Ergebnis der physikalischen Herleitung erhält man, wenn es sich um ein System mit Rückkopplung handelt, Differentialgleichungen, welche in Matlab/Simulink modelliert werden können. Durch Simulation des Modells können die gewonnenen Resultate dann mit Messungen verglichen und bei ungenügender Übereinstimmung Änderungen am Modell vorgenommen werden. Abbildung 3.1 zeigt den Arbeitsablauf der Modellbildung.

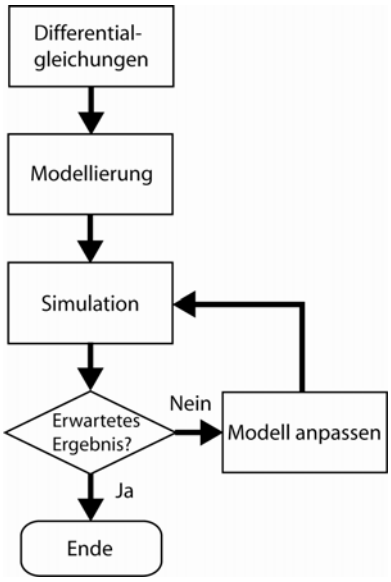


Abbildung 4.1: Arbeitsablauf Modellbildung

Zu welchem Grad das Modell mit der Wirklichkeit übereinstimmen muss hängt dabei immer vom Anwendungsfall ab. Der im Rahmen der Diplomarbeit entwickelte Regelkreis lässt sich in zwei Teilmodelle zerlegen. Zum einen ist das das Modell des Hubmagneten, der die Regelstrecke bildet, zum anderen das des Reglers, aus dem dann eigenständig ausführbarer Code generiert werden soll. Beide sollen in der Simulation zusammenarbeiten. Daraus folgt, dass die Güte des Reglers und dessen Funktionsfähigkeit in der Praxis in direktem Zusammenhang mit Genauigkeit des Modells des Hubmagneten steht.

Bei der Simulation von Modellen werden die zugrunde liegenden Differentialgleichungen in Differenzengleichungen umgeformt und numerisch integriert. Matlab stellt dazu eine Vielzahl von Integrationsmethoden zur Verfügung, deren Eignung von der Art der Differenzialgleichungen abhängt [5]. Bei der Simulation wird, beginnend von einem festen Anfangszustand, schrittweise der zeitlich nächste Wert berechnet. Jeder dieser Schritte baut auf das Ergebnis des letzten auf.

In Simulink kann die Schrittweite fest oder variabel sein. Ist die Schrittweite variabel, so ist sie von der Größer der Veränderung des Rechenergebnisses abhängig. Bei großer Veränderung wird die Schrittweite verkleinert und umgekehrt. Je kleiner die Schritte desto genauer das Ergebnis, desto größer aber der Rechenaufwand.

4.1 Überblick über Matlab Simulink

Simulink der Firma The Mathworks ist eine Software, die das Programm Matlab, des gleichen Herstellers, um ein Werkzeug zur graphischen Modellierung, Simulation und Analyse dynamischer Systeme erweitert. Solche Systeme können z.B. technischer, mathematischer, physikalischer oder auch finanzmathematischer Natur sein.
Die Modellierung erfolgt mithilfe graphischer Blöcke, welche untereinander durch Linien verbunden werden und zusammen einen Wirkplan bilden. Die einzelnen Blöcke sind in Bibliotheken organisiert. Neben der standardmäßig enthaltenen Simulink-Bibliothek, die im folgenden näher erläutert wird, sind auch noch zahlreiche zusätzliche Bibliotheken, so genannte Blocksets, die sowohl vom The Mathworks als auch von anderen Herstellern bezogen werden können, erhältlich. Sie erweitern die Funktionalität von Simulink für spezielle Anwendungen wie etwa Codegenerierung, Bildverarbeitung, Bioinformatik, Signalverarbeitung usw.
Nach dem Starten von Matlab kann Simulink durch drücken des -Buttons in der Werkzeugleiste oder durch Eingabe von `simulink` im Befehlsfenster gestartet werden. Es erscheint der Simulink Library Bowser (Abbildung 3.2, links).

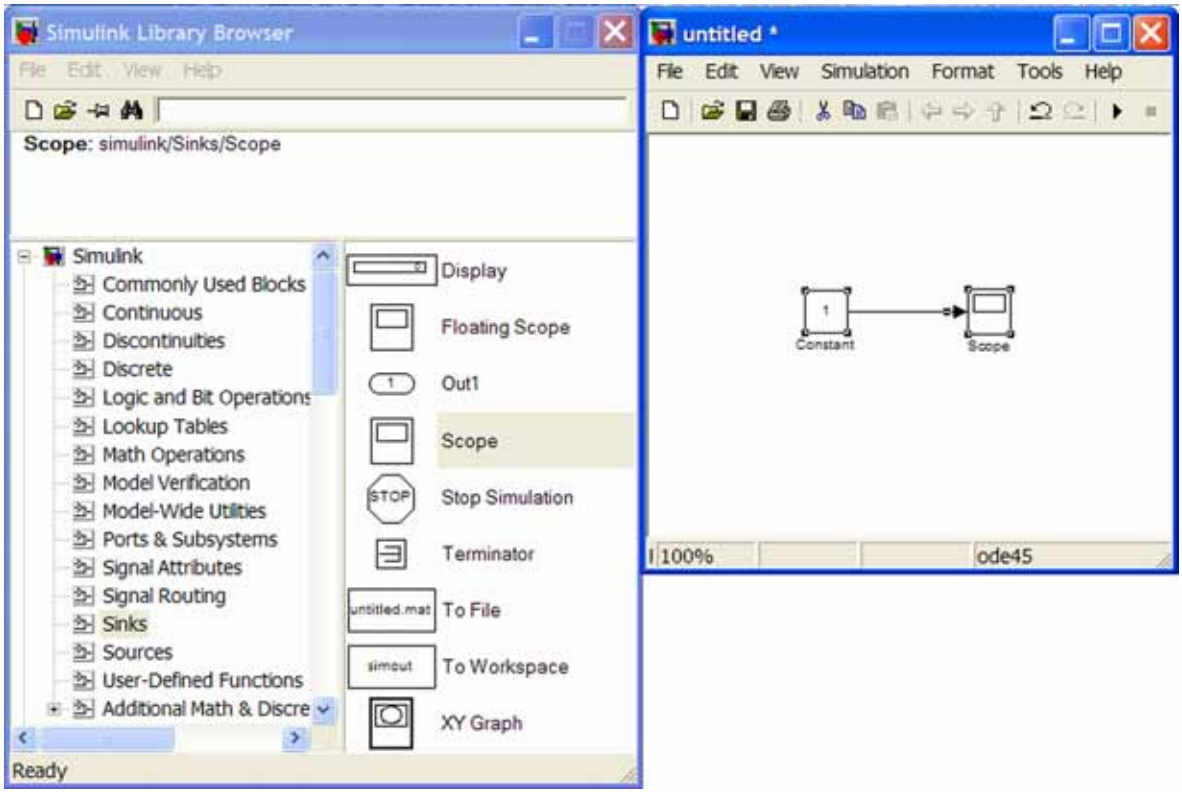


Abbildung 4.2: Links Simulink Library Browser, rechts Arbeitsreich

Von hier können Simulink-Modelle neu angelegt bzw. geöffnet werden. Der Library-Browser enthält eine Verzeichnisstruktur in der die einzelnen Blocks hierarchischen organisiert sind. Auf der rechten Seite befinden sich die einzelnen Blöcke des gewählten Ordners. Sie können einfach in das Fenster des Arbeitsbereichs(Abbildung 3.2, rechts) gezogen und dort platziert werden. Die Verbindung der Blöcke untereinander erfolgt durch ziehen von Signalleitungen mit der Maus. Die Signale dieser Leitungen können einfache Datentypen oder Datenstrukturen sein bzw. Arrays aus solchen. Im oberen Fensterabschnitt des Library-Browsers wird eine kurze Beschreibung des im Moment markierten Blocks oder Ordners angezeigt.

Die in diesem Projekt verwendeten Blocksets und Toolboxes sind:
- Simulink Standard Bibliothek
- Realtime Workshop
- Realtime Workshop Embedded Coder
- Fixedpoint Blockset
- System Generator von Xilinx

Zu den einzelnen Blocksets und Toolboxen:

Die Simulink Standard Bibliothek stellt die grundlegenden Bausteine für die Modellierung dynamischer Systeme zur Verfügung.

Im Simulink Standard Blockset sind Blöcke in folgenden Kategorien organisiert:
- *Commonly Used Blocks*: Beinhaltet eine Ansammlung allgemeiner Blöcke, die nur bedingt einer der unten genannten Kategorien zugeordnet werden können.
- *Continuous*: Integration, Differentiation, Verzögerungsglieder, Übertragungsfunktionen in zeitkontinuierlichen Systemen.
- *Discontinuous*: Blöcke mit nichtstetigem Übertragungsverhalten.
- *Discrete*: Integration, Differentiation, Verzögerungsglieder, Übertragungsfunktionen für zeitdiskrete Systemen.
- *Logical and Bit Operations*: Bitweise, logische und relationale Operationen.
- *Lookup Tables*: Zuordnungsfunktionen, denen nicht zwangsläufig eine mathematische Verknüpfung zugrunde liegt z.B. empirisch ermittelte Kennlinien.
- *Math Operations*: Dieser Unterpunkt beinhaltet Mathematische Verknüpfungen und Operationen. Neben den Grundrechenarten sind hier auch Verstärkerglieder und mathematische Funktionen zu finden.
- *Model-Verification*: Prüfblöcke, die sicherstellen, dass sich das Modell innerhalb bestimmter Grenzen bewegt.
- *Model-Wide Utilities*: Modellweit gültige Blöcke z.B. für Zeitverhalten oder Dokumentation und Information
- *Ports & Subsystems*: Blöcke, die der Hierarchiebildung innerhalb eines Modells dienen. Dies geschieht durch, zusammenfassen logischer zusammenhängende Blöcke in Subsystemen.
- *Signal Attributes*: Elemente zur Angleichung von Abtastraten, setzen von Anfangszuständen und auslesen von Signaleigenschaften.
- *Signal Routing*: Blöcke die den Signalweg beeinflussen wie etwa Multiplexer, Demultiplexer, Schalter, etc.

- *Sinks*: Blöcke dieser Kategorie dienen der Signalausgabe. Diese Ausgabe kann visuell über Graphen erfolgen oder in eine Datei bzw. in den Matlab Workspace geschrieben werden.
- *Sources*: Enthält Böcke zur Signalerzeugung. Signale können beispielsweise Konstanten oder Sprungfunktionen sein, von diversen konfigurierbaren Signalgeneratoren stammen oder aus Dateien oder vom Matlab Workspace gelesen werden.
- *User-Defined Functions*: Blöcke die die Verwendung eigener mathematischer Ausdrücke oder Funktionen zulassen.

Die einzelnen Blöcke dieser Kategorien werden, im Zusammenhang mit ihrer Verwendung im Modell des Hubmagneten und Reglers, näher erläutert.

Real-Time Workshop und Real-Time Workshop Embedded Coder:
Diese Pakete erweitert Simulink nicht um zusätzliche Funktionsblöcke, sondern baut Simulink um die Möglichkeit selbständig ablaufenden C-Code zu generieren aus. Dabei kann mit dem Real-Time Workshop nur Code erzeugt, der auf x86 kompatiblen Rechnern lauffähig ist. Für andere Plattformen ist zusätzlich die Real-Time Workshop Embedded Coder-Toolbox notwendig.

Fixedpoint Toolbox:
Auch dieses Paket erweitert Simulink nicht um neue Blöcke, sondern um Möglichkeit Modelle mit Festkomma Datentypen zu erstellen. Da aber die meisten einfachen Mikrocontroller keine Floating Point Unit, kurz FPU, besitzen, wird eine eigene Fixedpoint Bibliothek zur Verfügung gestellt, so dass festkommaoptimierter Code erzeugt werden kann. Zwar verarbeiten die meisten C-Compiler das ohne Fixedpoint-Bibliothek generierte ANSI-C, aber ein so erzeugtes Programm wäre wenig performant. Für den Einsatz des HDL-Coders ist diese Toolbox Vorraussetzung.

HDL-Coder:
Der HDL-Coder erweitert Simulink um die Fähigkeit, synthetisierbaren VHDL- oder Verilog-Code aus Modellen zu erzeugen.

Xilinx System Generator for DSP:
Neben der Produktion von programmierbaren Bausteinen bietet der Chiphersteller Xilinx auch Entwicklungssoftware für seine Bausteine an. Die vom System Generator bereitgestellten Blöcke dürfen im Modell nicht mit anderen Blöcken gemischt werden.

Der System Generator beinhaltet drei Blocksets:

XILINX Blockset:
Dieses Blockset enthält grundlegende Modellblöcke. Diese reichen von hardwarenahen Bausteinen für logische und relationalen Operationen, Register oder Ein-/Ausgängen bis zu abstrakteren Blöcken beispielsweise für die schnelle Fourier-Transformation.

Xilinx Reference Blockset:
Hier sind Blöcke zu finden, die aus Blöcken des Xilinx Standard Blocksets zusammengestellt sind. Darunter sind auch komplizierte Algorithmen wie Winkelfunktionen und digitale Filter zu finden.

Die beiden oben aufgeführten Gruppen sind in folgende Kategorien eingeteilt:

Basic Elements: Standard-Blöcke
Communication: Blöcke für digitale Kommunikationssysteme
Control Logic: Steuerung und Zustandsautomaten
Data Types: Datenkonvertierung, Ein-/Ausgange
DSP: Digitale Signalverarbeitung
Math: Mathematische Funktionen
Memory: Blöcke für Speicherzugriff
Shared Memory: Implementierung für Shared-Memory-Zugriff
Tools: Codegenerierung, Resourcenschätzung, Analyse

XILINX Extrem DSP Blockset:
Enthält Blöcke für das BenADDA FPGA-Entwicklungsboard von Nallatech, das speziell für Digitale Signalverarbeitung konzipiert wurde und wird daher an dieser Stelle nicht näher erläutert.

4.2 Das Modell des Hubmagneten in Simulink

Wie bereits geschildert liegt jeder Computersimulation ein Simulationsmodell zu Grunde. Da ein Modell eine Abstraktion der Wirklichkeit ist, muss dies nur genau genug sein um die nötigen Schlüsse aus ihm zu ziehen. Parameter, die nur unwesentlichen Einfluss auf das Verhalten haben, werden vernachlässigt.
Die Anforderung an dieses Modell ist, dass es den real existierenden Hubmagneten präzise genug nachbildet, um einen Regler zu entwerfen, der sich auch im praktischen Einsatz auf einem Mikrocontroller oder FPGA bewährt.

Abbildung 4.3 zeigt das Schnittbild des Hubmagneten, das als Grundlage der Modellbildung dient. Er besteht aus zwei parallel geschalteten Spulen, einem Joch aus Stahl, sowie dem Anker, der durch die Regelung in einer festgelegen Position gehalten werden soll.

Die folgenden Vereinfachungen wurden für das Modell Hubmagneten vorgenommen:

- Magnetische Streufelder die außerhalb des Jochs verlaufen wurden vernachlässigt.
- Die Ankerbewegung verläuft reibungsfrei.
- Innenwiderstand der Spannungsquelle wird als 0 angenommen.
- Effekte, die sich durch die Magnetisierung der ferromagnetischen Werkstoffe ergeben, bleiben unberücksichtigt.

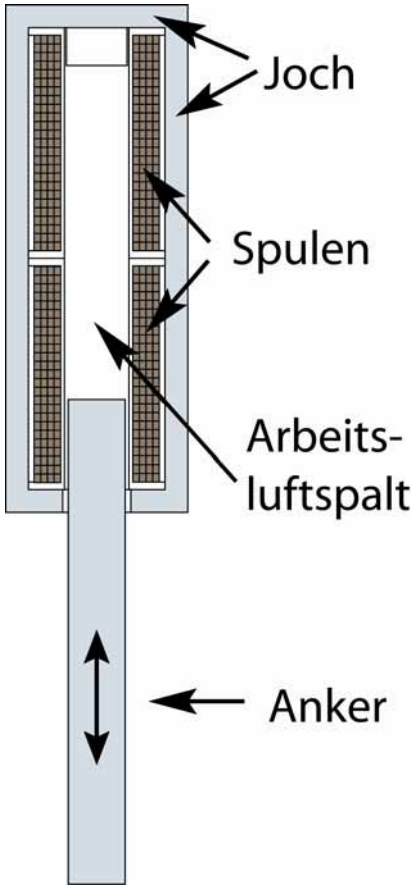


Abbildung 4.3: Elemente des Hubmagneten im Schnittbild

4.2.1 Physikalische Herleitung

Abmessungen, Parameter und Konstanten

Abbildung 3.4 zeigt die Abmessungen des Hubmagneten. Aus ihnen und den elektromagnetischen Eigenschaften, werden die Parameter für das Simulationsmodell hergeleitet.

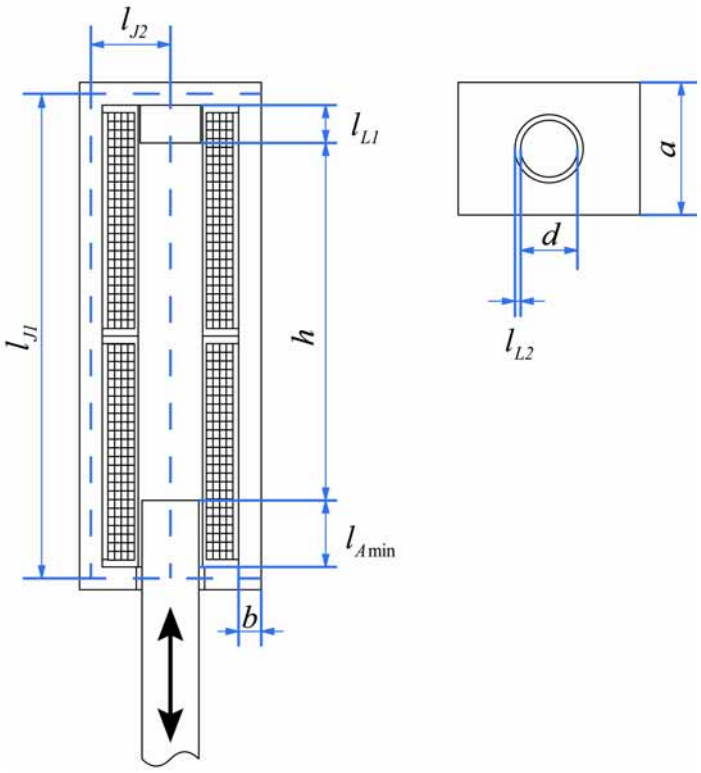


Abbildung 4.4: Abmessungen des Hubmagneten: Links im Querschnitt von Vorne; rechts Ansicht von unten

Windungszahl : $n = 1900;$

Ohmscher Widerstand der Spulen : $R_S = 7.6\Omega;$

Ankermasse: $m = 55g;$

Hubstrecke des Ankers: $h = 90mm = 0.09m;$

Minimale Eintauchtiefe des Ankers: $l_{A\min} = 18mm;$

Minimaler Luftspalt : $l_{L\min} = l_{L1} + l_{L2} = 0.012m;$

Mittlere Jochlänge : $l_J = l_{J1} + 2l_{J2} = 0{,}168m$

Querschnittsfläche des Jochs: $A_j = 2ab = 2 \cdot 6mm \cdot 35mm = 420mm^2 = 4{,}2 \cdot 10^{-4} m;$

Querschnittsfläche des Luftspalts: $A_L = \left(\frac{d}{2}\right)^2 \pi = 1{,}77 \cdot 10^{-4} m^2;$

Querschnittsfläche des Ankers: $A_A \approx A_L = 1{,}77 \cdot 10^{-4} m^2;$

Magnetische Feldkonstante: $\mu_0 = 4\pi \cdot 10^{-7} \frac{Vs}{Am};$

Permabilitätskonstante Eisen: $\mu_r = 1200;$

Fallbeschleunigung $g = 9.81 ms^{-2};$

Aufstellen der Gleichungen:

Ausgangsgleichung des Models ist,

(3.1) $F_A = -F_M + m \cdot g;$

F_A: Kraft auf den Anker $[1N]$

F_M: Magnetkraft $[1N]$

m: Masse des Ankers $[1kg]$

g: Erdbeschleunigung $\left[1\frac{m}{s^2}\right]$

welche aussagt, dass die Kraft, die den Anker in Bewegung setzt, gleich der Kraftdifferenz zwischen der Gewichtskraft des Ankers und die auf ihn wirkende magnetische Anziehungskraft ist.

Aus dem zweiten Newtonschen Axiom, dem Aktionsprinzip,

$a = \ddot{x} = \frac{F}{m};$

x: Ankerposition $[1m]$

$\ddot{x}$: Ankerbeschleunigung $\left[1\frac{m}{s^2}\right]$

folgt,

$F_A = m \cdot \ddot{x};$

und

(3.2) $m \cdot \ddot{x} = -F_M + m \cdot g;$

Dies ist die Differentialgleichung des Systems. Gleichungen werden so bezeichnet, wenn die Ableitung einer unbekannten Funktion enthalten ist.

Die Kraft, die ein Magnetfeld auf einen Körper ausübt, lässt sich folgender Formel herleiten:

(3.4) $F_M = \frac{dW_{mag}}{dx};$

W_{mag}: Energiegehalt des Magnetfelds $\left[1W = 1\frac{N}{s}\right]$

Ferner gilt:

(3.5) $W_{mag} = \frac{1}{2} \cdot L \cdot i^2;$

L: Induktivität der Spule $\left[1\frac{Vs}{A} = 1H\right]$

i: Stromsträrke $[1A]$

Die Magnetkraft ist somit die Kraft, die sich durch Veränderung der im Magnetfeld gespeicherten Energie ergibt, wenn sich der Anker bewegt.

(3.6) $F_M = \frac{1}{2} \cdot \frac{dL}{dx} \cdot i^2;$

Die Induktivität der Spule berechnet sich wie folgt

(3.7) $L = n \cdot \frac{\phi}{i}$

ϕ: Magnetischer Fluss $[1V \cdot s = 1Wb]$

n: Windungszahl der Spule $[1]$

um den magnetische Fluss ϕ zu berechen, bedient man sich des magnetischen Widerstands R_m, der definiert ist als:

(3.10) $R_m = \frac{l}{\mu_0 \mu_r \mathrm{A}};$

R_m: magnetischer Widerstand $\left[\frac{A}{Vs} = \frac{A}{Wb} = \frac{1}{H}\right]$

μ_0: magnetische Feldkonstante $\left[4 \cdot \pi \cdot 10^{-7} \frac{V \cdot s}{A \cdot m}\right]$

μ_r: Permeabilitätskonstante $[1]$

Der Gesamtwiderstand eines magnetischen Kreises ergibt sich aus der Summe der Einzelwiderstände. Im Einzelnen sind das hier R_{mJ} für den Magnetischen Widerstand des Jochs, R_{mA} für den des beweglichen Ankers mit sich ändernder Länge und sowie R_{mL} für den des Luftspalts, dessen Länge sich ebenfalls ändert.

$$R_{mJ} = \frac{l_J}{\mu_0 \mu_r A_J};$$
$$R_{mA} = \frac{l_{A\min} + x}{\mu_0 \mu_r A_A};$$
$$R_{mL} = \frac{l_{L\min} + h - x}{\mu_0 A_L};$$
$$R_{mGes} = R_{mJ} + R_{mA} + R_{mL};$$

l_J: mittlere Länge des Jochs $[1m]$

$l_{A\min}$: minimale Eintauchtiefe des des Ankers in die Spule $[1m]$

$l_{L\min}$: Luftspalt bei angezogenem Anker $[1m]$

h: Hubstrecke $[1m]$

$\mathrm{A_J}$: Querschittsfläche des Jochs $[1m^2]$

$\mathrm{A_A}$: Querschittsfläche des Ankers $[1m^2]$

$\mathrm{A_L}$: Querschittsfläche des Luftspalts $[1m^2]$

(3.8) $R_{mGes} = \frac{l_J}{\mu_0 \mu_r A_J} + \frac{l_{A\min} + x}{\mu_0 \mu_r A_A} + \frac{l_{L\min} + h - x}{\mu_0 A_L};$

Der magnetische Fluss ist dann:

$$\phi = \frac{n \cdot i}{R_{mGes}}; \quad (3.9)$$

Unter Verwendung von Gleichung 3.7 folgt:

$$L = \frac{n^2}{R_{mGes}} = \frac{n^2}{\frac{l_J}{\mu_0 \mu_r A_J} + \frac{l_{A\min} + x}{\mu_0 \mu_r A_A} + \frac{l_{L\min} + h - x}{\mu_0 A_L}}; \quad (3.10)$$

Vereinfacht und mit der Definition $A_{AL} := A_A = A_L$, da die Ankerquerschnittsfläche in etwa der des Luftspalts entspricht, ergibt sich,

$$L = \frac{n^2 \mu_0 \mu_r A_J A_{AL}}{((l_{L\min} + h - x)\mu_r + l_{A\min} + x)A_J + l_J A_{AL}}; \quad (3.11)$$

Die so errechnete Induktivität wird für die Berechung des elektrischen Kreises benötigt. Für die Magnetkraft wird nun noch die Ableitung der Induktivität nach *x*, dem Ankerweg, benötigt.

$$\frac{dL}{dx} = -\frac{n^2 \mu_0 \mu_r A_J{}^2 A_{AL}(-\mu_r + 1)}{(((l_{l\min} + h - x)\mu_r + l_{A\min} + x)A_J + l_J A_{AL})^2}; \quad (3.12)$$

eingesetzt in 3.6:

$$F_M = -\frac{1}{2} \cdot \frac{n^2 \mu_0 \mu_r A_J{}^2 A_{AL}(-\mu_r + 1)}{(((l_{l\min} + h - x)\mu_r + l_{A\min} + x)A_J + l_J A_{AL})^2} \cdot i^2; \quad (3.13)$$

Gleichung 3.2 läst sich dann so ausdrücken:

$$m \cdot \ddot{x} = -\frac{1}{2} \cdot \frac{n^2 \mu_0 \mu_r A_J{}^2 A_{AL}(-\mu_r + 1)}{(((l_{l\min} + h - x)\mu_r + l_{A\min} + x)A_J + l_J A_{AL})^2} \cdot i^2 + m \cdot g; \quad (3.14)$$

Neben der oben ausgeführten Herleitung, ist darüber hinaus auch der elektrische Kreis des Modells zu berechnen, der ja ursächlich für den Stromfluss in der Spule und somit das Magnetfeld ist. Der Strom ist aber nicht konstant, da der Quellspannung, während einer Änderung des Magnetfelds, eine selbstinduzierte Spannung entgegenwirkt. Dies ist z.B. bei der Änderung der angelegten Spannung der Fall, wie dies beim Einschalten oder später durch den Regler geschieht. Zudem ist, durch den beweglichen Anker, auch die Induktivität L der Spule nicht konstant, sondern abhängig von der momentanen Ankerposition.

Für die selbstinduzierte Spannung gilt:

(3.15) $u_{ind} = -\frac{di}{dt}L;$ u_{ind} : Induktionsspannung der Spule $[1V]$

Da die reale Spule einen ohmschen Widerstand besitzt, muss der elektrische Kreis nach dem Schema aus Abbildung 4.5 hergeleitet werden.

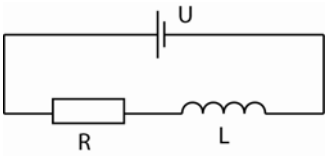


Abbildung 4.5: Ersatzschaltbild einer realen Spule

Es lässt sich die Maschenregel anwenden.

$u_R = u + u_{ind};$

u : Quellspannung $[1V]$

u_R : Spannung am ohmschen Widerstand $[1V]$

Umgestellt nach u ,

$u = u_R - u_{ind};$

mit $u_R = R_{ges} \cdot i;$ und Gleichung 3.15 folgt:

$u = R_{ges} i + \frac{di}{dt}L;$ R_{ges} : Gesamtwiderstand des Stromkreises $[1\Omega]$

Da aber die Induktivität eine Funktion der Ankerposition ist, und diese sich ebenfalls ändern kann, gilt:

(3.16) $u = R_{ges} i + \frac{d(iL(x))}{dt} = R_{ges} i + L(x)\frac{di}{dt} + \frac{dL}{dt} \cdot i;$

Eine Gleichung für die Ableitung der Induktivität nach der Zeit $\frac{dL}{dt}$ gibt es aber im Moment noch nicht direkt. Aber es gibt die Gleichung für $\frac{dL}{dx}$. In *x*, der aktuellen Ankerposition, ist die Zeit indirekt enthalten. Folgender Ansatz führt deshalb zum Erfolg.

$\frac{dL}{dt} = \frac{dL}{dt} \cdot \frac{dx}{dx} = \frac{dL}{dx} \cdot \frac{dx}{dt} = \frac{dL}{dx} \cdot \dot{x};$

Eingesetzt in 3.16 ergibt sich:

(3.17) $u = R_{ges} i + L(x)\frac{di}{dt} + \frac{dL}{dx} \cdot \dot{x} \cdot i;$

Das Ergebnis lässt sich so interpretieren, dass sowohl eine Stromänderung, z.B. durch ändern der Eingangsspannung, als auch die Änderung der Ankerposition das Magnetfeld beeinflussen und somit eine selbstinduzierte Spannung hervorrufen. Diese wirkt der Quellenspannung entgegen. Die Spannungsdifferenz liegt an R_{ges} an und führt zum Stromfluss i.

Zwar ist die Ankergeschwindigkeit $\dot{x}$ bisher noch nicht berechnet, jedoch ergibt sich diese automatisch bei der Modellbildung, wie in Abschnitt 3.2.3 gezeigt wird.
Die Gleichungen 3.11, 3.14 und 3.17 genügen, um das Simulationsmodell des Hubmagneten in Simulink zu modellieren.

4.2.2 Darstellung in Simulink

Um die, im vorigem Abschnitt aufgestellten, Gleichungen als simulierbaren Wirkplan in Simulink zu modellieren, hat es sich als vorteilhaft erwiesen, die Differenzialgleichungen so umzustellen, dass in den einzelnen Summentermen der Gleichungen nur immer eine Ableitung einer Unbekannten enthalten ist.
Diese Vorgehensweise hat schon im Vorfeld positive Auswirkungen auf die Übersichtlichkeit des späteren Modells.

Hier noch ein mal die drei für das Modell wichtigen Gleichungen. Sie entsprechen bereits in der geforderten Form.

(I) $m \cdot \ddot{x} = \frac{1}{2} \cdot \frac{n^2 \mu_0 \mu_r {A_J}^2 A_{AL}(-\mu_r + 1)}{(((l_{l\min} + h - x)\mu_r + l_{A\min} + x)A_J + l_J A_{AL})^2} \cdot i^2 + m \cdot g;$

(II) $L = \frac{n^2 \mu_0 \mu_r A_J A_{AL}}{((l_{L\min} + h - x)\mu_r + l_{A\min} + x)A_J + l_J A_{AL}} = L(x);$

(III) $u = R_{ges} i + L(x)\frac{di}{dt} + \frac{dL}{dx} \cdot \dot{x} \cdot i;$

Folgendes Simulink-Modell (Abbildung 4.6) kann daraus erstellt werden, wobei es sich nicht um die einzig mögliche Darstellung handelt. Das Modell entstand in Anlehnung an ein Modell eines Hufeisenmagneten aus [4]. Diese sind vergleichbar, da für beide Modelle das Magnetfeld im Luftspalt als homogen angenommen wird.

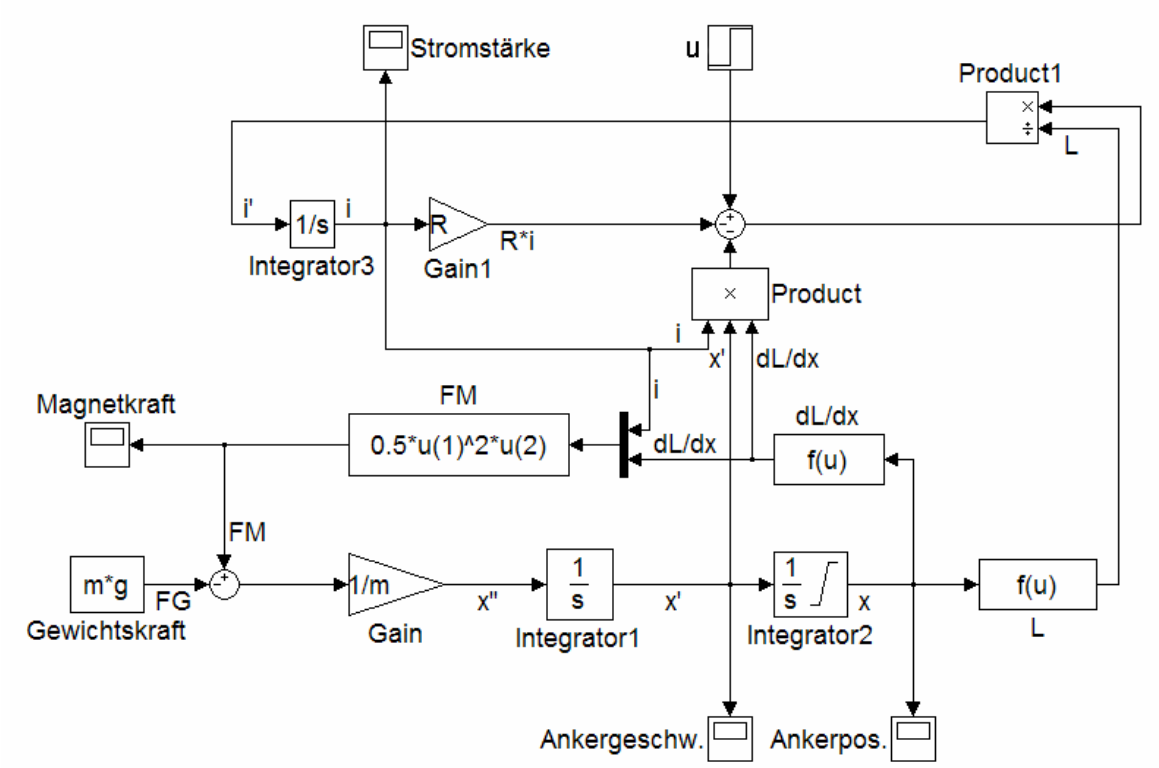
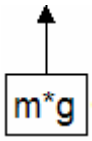


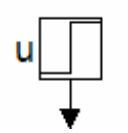
Abbildung 4.6: Simulink-Modell eines Hubmagneten

Es wurden ausschließlich Simulink-Blöcke aus der Standardbibliothek verwendet:

Kategorie *Source*:



Constant: Erzeugt ein konstantes Signal. Dies kann ein vorgegebener Wert sein, aber auch eine Variable, die im Matlab-Workspace definiert wurde.



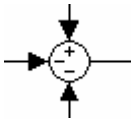
Step: Generiert ein Sprungsignal von einem konstanten Anfangswert zu einem konstanten Endwert. Beide Werte, sowie der Zeitpunkt des Sprunges werden im Eigenschaftsfenster des Blocks festgelegt

.

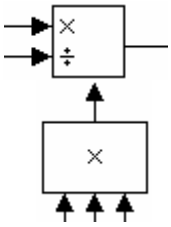
Kategorie *Sink*:

Scope: Visualisiert das Eingangssignal in Abhängigkeit von der Zeit.

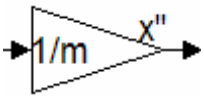
Kategorie *Math Operations*:



Sum: Als Ausgabe wird die Summe aber auch die Differenz der Eingänge berechnet. Welcher der Eingänge dabei Summand oder Subtrahend ist wird im Eigenschaftsfenster im Textfeld „List of signs“ durch Eintragen von „+“ oder „ – “ festgelegt.

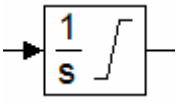


Product: Dividiert bzw. Multipliziert die Eingangssignale und legt das Ergebnis auf den Ausgang. Welcher der Eingänge dabei Multiplikand oder Divisor ist, wird im Eigenschaftsfenster im Textfeld „Number of Inputs“ durch Eintragen von „*“ oder „ / “ festgelegt. Wird stattdessen eine Ganzzahl Angegeben, so sind alle Eingänge Multiplikanden.



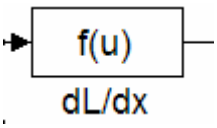
Gain: Multiplikation des Eingangs mit einem konstanten Faktor. Dies kann auch eine Variable des Matlab-Workspace sein.

Kategorie *Continuous*:



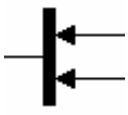
Integrator: Dieser Block integriert fortlaufend das Eingangssignal. Die gewählte Methode der numerischen Integration wird im Menü *Simulation/Configuration Parameters…* im Unterpunkt *Solver* des Arbeitsbereichs festgelegt. Im Eigenschaftsdialogfenster des Integrators kann außerdem sein Anfangswert, sowie obere und untere Begrenzung des Ausgabesignals festgelegt werden.

Kategorie *User- Defined Functions*:



Fcn: Selbstdefinierte Funktion mit Eingangssignal als Funktionsargument. Im Eigenschaftsfenster kann die Funktion in Form eines Matlabtypischen Ausdrucks eingegeben werden. Die Eingangsvariablen sind als u definiert. Bei Multiplex-Leitungen am Eingang kann mittels Index z.B. u(1) auf die einzelnen Signale zugegriffen werden.

Kategorie *Signal Routing*:



Mux: Abkürzung für Multiplexer. Er dient dazu mehrere Signalleitungen zu einer zusammenzufassen. Mittels eines Demux kann eine solche Leitung wieder aufgeteilt werden.

Erläuterung des Simulink-Modells des Hubmagneten:

Der untere Bereich des Modells bildet das Äquivalent zur Gleichung (I). Das Summationsglied (Abbildung 4.7) in der linken unteren Ecke ist der Ausgangspunkt der Betrachtung. Es spiegelt die Kräftegleichung (3.2) wieder. Die Differenz aus Magnetkraft und Gewichtskraft ist die Kraft, die auf den Anker wirkt.

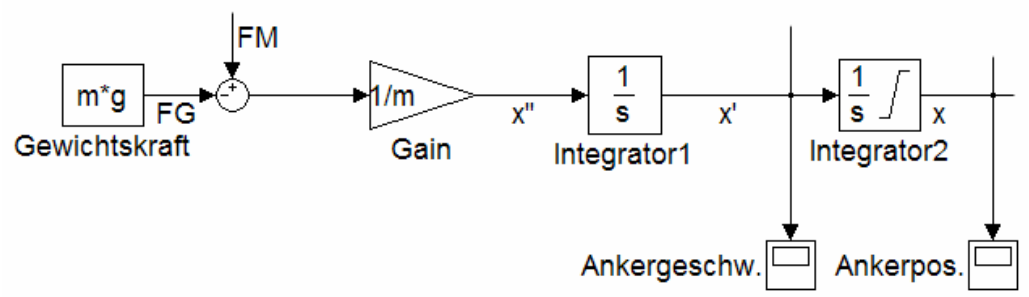


Abbildung 4.7: Modellauszug Kräftegleichung

Die Ausgabe dieses Blocks ist somit $m\ddot{x}$. Im folgenden Gain-Block wird durch die konstante Masse des Ankers dividiert. Die so erhaltene Ankerbeschleunigung wird nun zweimal integriert. Man erhält Ankergeschwindigkeit und Ankerposition. Im zweiten Integrationsblock wird zudem der Initialwert der Ankerposition, $x = 0m$, gesetzt.

Mit der bekannten Ankerposition und Geschwindigkeit können nun auch $L(x)$ und $\frac{dL}{dx}$ berechnet werden. Für beide wird jeweils ein fcn-Block generiert (Abbildung 4.8).

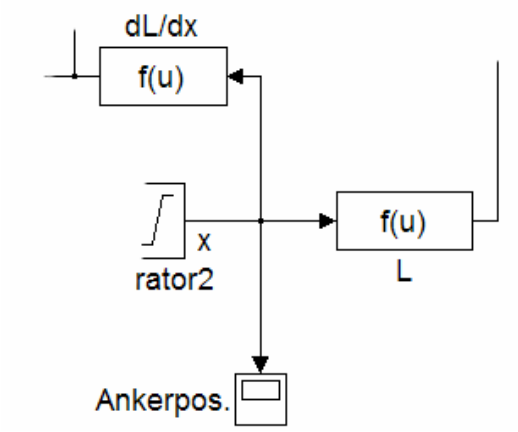


Abbildung 4.8: Modellauszug fcn-Blocks

Für $\frac{dL}{dx}$ wird Gleichung (3.12), die ja wiederum Teil von Gleichung (I) ist, verwendet. Für fcn-Block $L(x)$ kommt Gleichung (II) zum Einsatz. In den beiden fcn-Blocks muss lediglich die Variable *x* des Ausdrucks durch *u* ersetzt werden, da das Eingangssignal so definiert ist.

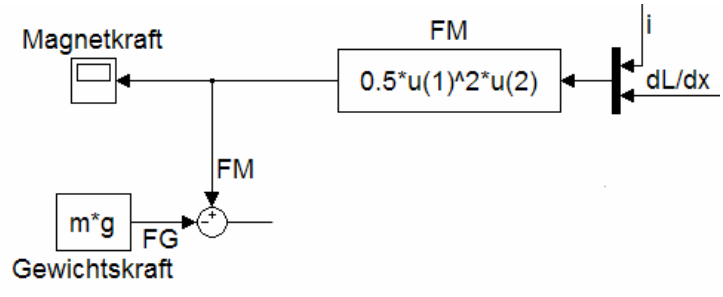


Abbildung 4.9: Modellauszug Magnetkraft

In einem weiteren fcn-Block kann dann mithilfe von $\frac{dL}{dx}$ und *i* aus dem elektrischen Kreis die Magnetkraft berechnet werden(Abbildung 4.9). Um jedoch beide Signale als Eingang nutzen zu können, ist ein Multiplexer vorgeschaltet. Die Magnetkraft liegt nun wieder am Eingang des Summationsgliedes für die Kräftegleichung, das zu Beginn erwähnt wurde an. Die Abhängigkeiten der Differenzialgleichung (I) sind durch den geschlossenen Kreis erfüllt.

Der einzige offene Wert, der noch Einfluss auf Gleichung (I) hat, ist die Stärke des Stromes der durch die Spule fließt. Gleichung (III) formuliert den Zusammenhang von Stromstärke und Induktivität der Spule. Auch hier soll das Summationsglied (Abbildung 4.10, rechts der Mitte) als Ausgangspunkt dienen. Der Step-Block *u* darüber fungiert dabei als Spannungsquelle und simuliert den Einschaltvorgang. Von der Eingangsspannung wird die induzierte Spannung subtrahiert, die von der Ankerbewegung herrührt.

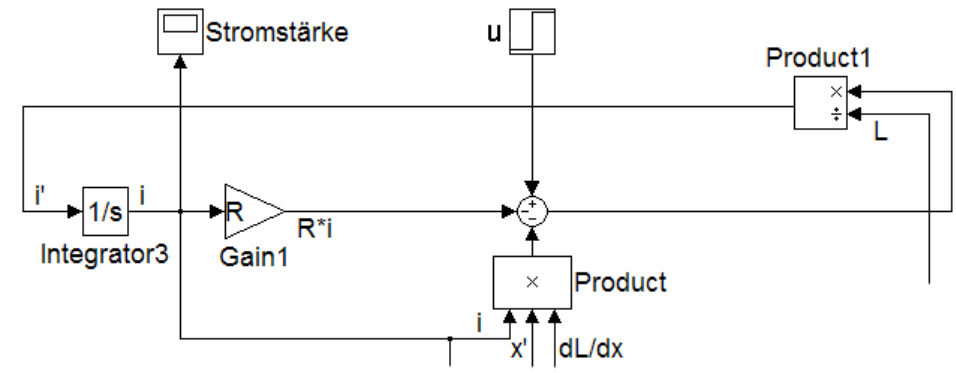


Abbildung 4.10: Modellauszug Stromkreis

Dividiert man diesen Wert laut Gleichung (III) durch die Induktivität, die ja bereits in einem fcn-Block ermittelt wurde, erhält man die Stromänderung, die integriert den tatsächlich fließenden Strom im Stromkreis ergibt.

Erweiterung des Simulink-Modells

Das erstellte Modell, simuliert das Verhalten des realen Hubmagneten. Die Ergebnisse sind aber nur dann korrekt, wenn sich ein Teil des Ankers innerhalb der Spule befindet. Ist dies nicht der Fall, läuft die Simulation außerhalb des spezifizierten Bereichs. Beim realen Hubmagneten sind mechanische Anschläge oben, im Inneren der Spule, und unten, durch eine Auffangeinrichtung, vorhanden. Um das korrekte Betriebsverhalten nachzuahmen, muss das Modell um diese mechanischen Grenzen erweitert werden

Für den Betrieb des Hubmagneten ergeben sich folgende Fälle:

1. Der Anker befindet sich innerhalb der Hubstrecke
2. Der Anker befindet sich am oberen Anschlag:
   2.1 Die Gewichtskraft ist größer als die Magnetkraft
   2.2 Die Gewichtskraft ist kleiner oder gleich der Magnetkraft
3. Der Anker befindet sich am unteren Anschlag:
   3.1 Die Gewichtskraft ist größer als die Magnetkraft
   3.2 Die Gewichtskraft kleiner oder gleich der Magnetkraft

Die Beschränkung der Ankerbewegung hat im Modell Auswirkungen auf drei Parameter. Das sind die Hubhöhe x, die Geschwindigkeit $\dot{x}$ und die Beschleunigung $\ddot{x}$ des Ankers.

Trifft der Anker auf einen Anschlag, so erfolgt eine hohe Gegenbeschleunigung, die die Ankergeschwindigkeit auf 0 abbremst. Der Anker verharrt in der Anschlagsposition.
Diese Gegenbeschleunigung lässt sich im Modell nur sehr schwer rekonstruieren.
Deshalb geht man vom Zustand des Ankers nach dem Abbremsvorgang aus. Solange die resultierende Kraft aus Gewichtkraft und Magnetkraft in Richtung des Anschlags wirkt, erfährt der Anker keine Beschleunigung, die Geschwindigkeit bleibt 0. Dies ist für die oben genannten Fälle 2.2 und 3.1 zutreffend. Für die restlichen Fälle verhält sich das Modell des Magneten normal.

Um festzustellen welcher der oben genanten Fälle gerade zutreffend ist, müssen die drei Werte Gewichtskraft, Magnetkraft und Hubhöhe überwacht werden.
Dies wird mit einem fnc-Block umgesetzt. Eine bisher noch nicht angeführte Eigenschaft dieses Blocks ist, dass er relationale und logische Operationen durchführen kann und dazu die Wahrheitswerte 0 oder 1 ausgibt.

Der Ausgang des fcn-Blocks soll eins sein, wenn sich der Anker innerhalb der Hubstrecke befindet. Ebenso soll der Ausgang eins sein, wenn sich der Anker am unteren Anschlag befindet und die Magnetkraft größer als die Gewichtskraft ist. Der Ausgang soll auch dann eins sein, wenn der Anker am oberen Anschlag befindet und die Gewichtskraft größer als die Magnetkraft ist. Der Folgende logische Ausdruck fast dies zusammen:

$$y = (x > 0 \wedge x < h_{\max}) \vee ((x <= 0) \wedge (F_M > F_g)) \vee ((x >= h_{\max}) \wedge (F_g > F_M));$$

Mit dem kleinen Kunstgriff, die logischen Ausgangswerte mit den Signalleitungen zu multiplizieren, können diese gezielt beeinflusst werden. Liefert der fcn-Block eine Eins für logisch Wahr, bleibt der Signalweg bei der Multiplikation unbeeinflusst. Liefert der fcn-Block hingegen Null, wird der Ausgang des Multiplikators ebenfalls Null.
Da sowohl Geschwindigkeit als auch Beschleunigung des Ankers am Anschlagspunkt 0 sind, werden dort solche Multiplikatoren eingefügt.
Zudem verfügen die Integrator-Blocks über einen flankengesteuerten Reset-Eingang. Dieser kann den Integrationswert auf den Initialisierungswert zurücksetzten, was für den Integrator der Beschleunigung Anwendung findet. Abbildung 4.11 zeigt die Erweiterungen des Modells.

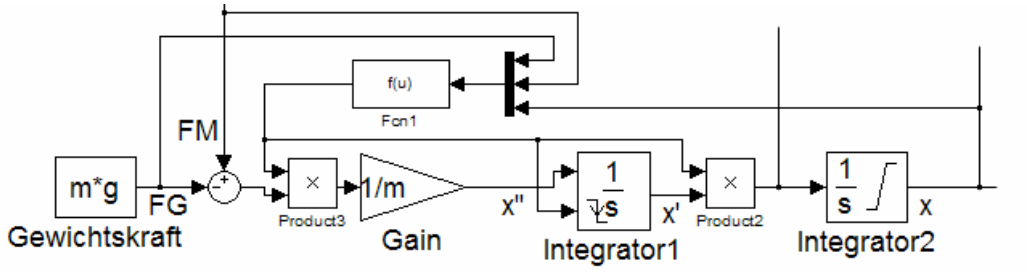


Abbildung 4.11: Erweiterungen am Modell

Modellverifizierung und Modellangleichung

Um zu überprüfen, in wie weit das Modell mit der Realität übereinstimmt, wurde eine Messreihe am realen Hubmagneten vorgenommen und mit dem Modell verglichen.
Dabei wurde gemessen, welche Spannung notwendig ist, um den Anker von unterschiedlicher Ausgangsposition aus anzuheben. Dadurch erhält man einen Vergleichswert über den gesamten Wertebereich (Abbildung 4.12).

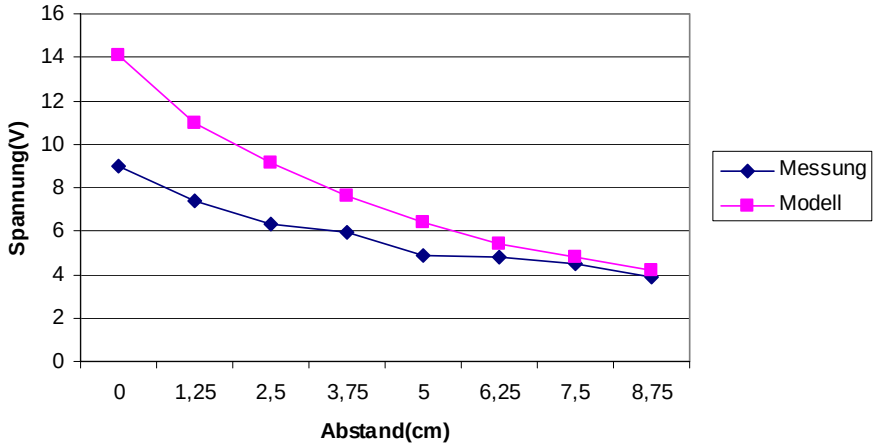


Abbildung 4.12: Vergleich: Modell und realer Hubmagnet

Wie zu erkennen ist, ist der Fehler des Modells nicht unwesentlich. Der Kurvenverlauf ähnelt sich aber. Der auftretende Fehler steht in direktem Zusammenhang mit der Hubhöhe. Je näher sich der Anker am oberen Anschlagpunkt des Hubmagneten befindet, desto geringer der Fehler.

Deshalb wird das Modell um einen wegabhängigen Korrekturfaktor erweitert, der mit der Magnetkraft multipliziert wird. Die Erweiterung erfolgt unter Zuhilfenahme eines fcn-Block, der folgende Gleichung enthält:

$$V = 1 + \left(k \cdot \frac{h - x}{h} \right);$$

$V :=$ Verstärkunkgsfaktor [1]
$k :=$ Korrekturfaktor [1]
$h :=$ max. Hubhöhe [1m]
$x :=$ Hubhöhe [1m]

Mit dem empirisch ermittelten Korrekturfaktor von $k = 0.7$ ergibt sich der Kurvenverlauf in Abbildung 4.13.

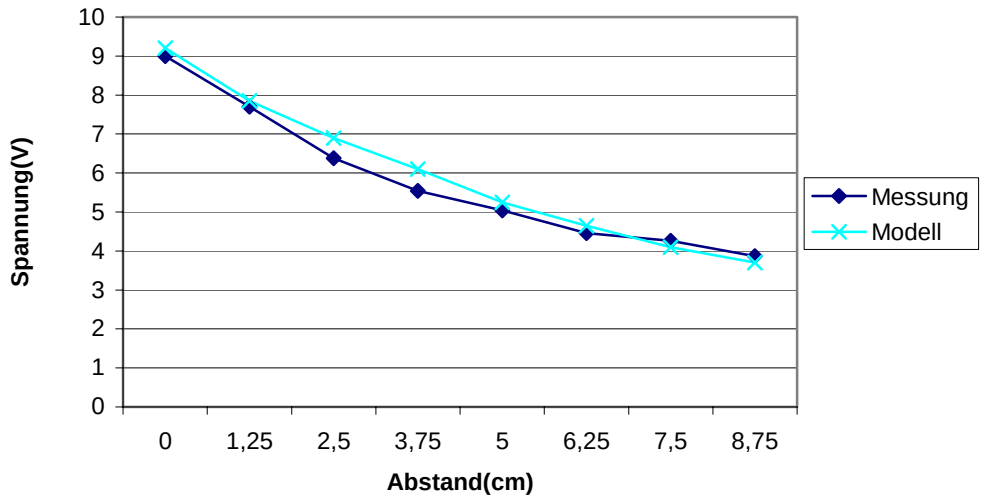


Abbildung 4.13: Vergleich: Modell mit Korrekturfaktor

Der berechnete Wert scheint im Vergleich zum Gemessenen flacher zu verlaufen. Die Genauigkeit des Simulationsmodells sollte jedoch für den Entwurf eines Reglers genügen.

4.3 Simulink Reglermodell

Im letzten Unterabschnitt wurde die Modellierung der Regelstrecke erläutert. Um einen geschlossenen Regelkreis zu erhalten, muss das Modell mit einem Regler ergänzt werden. Eingang des Reglers ist die Hubhöhe x. An die Stelle der Sprungfunktion als Spannungsquelle des Modells tritt der Ausgang des Reglers.

4.3.1 Modell eines stetigen PID-Reglers

Ausgangspunkt des Reglermodells ist die Nachbildung eines stetigen PID-Reglers (Abbildung 4.14).
Dieser kann für viele Regelstrecken eingesetzt werden und zeichnet sich durch ein schnelles Anfahren des Sollwerts mit anschließendem präzisen Ausregeln aus.

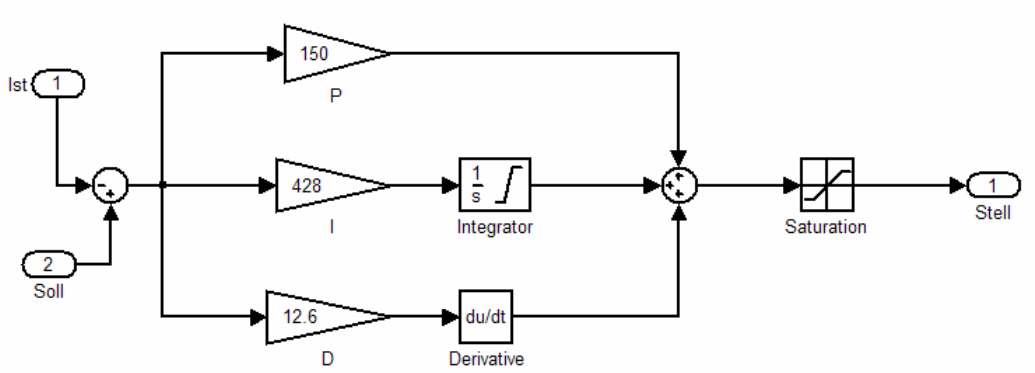


Abbildung 4.14: Modell eines stetigen PID-Reglers

Den Eingang des Reglers bildet ein Subtraktionsglied, das die Regeldifferenz aus Soll- und Istwert bildet. Der Regler selbst setzt sich aus drei parallel geschalteten Einzelreglern zusammen, was durch die drei Buchstaben P, I und D zum Ausdruck kommt.

Der P-Zweig bildet dabei den Proportionalanteil, des Stellsignals. Die Regeldifferenz, wird mit einem konstanten Faktor K_P multipliziert.

I ist der Integralanteil des Stellsignals. Dabei wird die Regeldifferenz über die Zeit Integriert und mit dem Faktor K_I multipliziert. Wird eine Regeldifferenz durch den Proportionalanteil nicht vollständig ausgeglichen, verstärkt sich der Einfluss des integralen Regelanteils mit der Zeit, und kompensiert dies.

D steht für den Differential-Anteil des Reglers. Er wird bei Änderung der Regeldifferenz wirksam und sorgt für ein schnelleres Reagieren des Reglers auf Änderung der Regelgröße, dämpft aber auch die anderen Regelanteile. Ein D-Regler alleine ist nutzlos, da er von der Änderung der Regeldifferenz und nicht von der Regeldifferenz selbst abhängig ist.

Vor dem Ausgang des Reglers ist ein Saturation-Block geschaltet. Dieser beschränkt das Stellsignal auf einen gültigen Wert. In diesem Fall sind dies 12 Volt.

Abbildung 4.15 zeigt die Sprungantwort der einzelnen Regelanteile b),c) und d) nach einer sprunghaften Änderung der Regeldifferenz a).

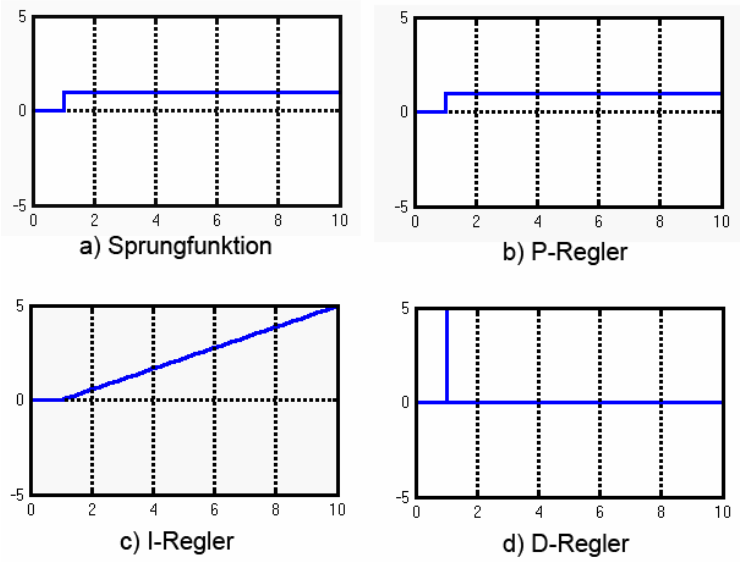


Abbildung 4.15: Sprungantwort der einzelnen Regleranteile

Die drei Anteile werden zum Stellsignal summiert. Abbildung 4.16 zeigt die Wirkung eines Sprungs der Regelgröße auf einen analogen PID-Regler.
Wie man an der Sprungantwort erkennen kann, findet man alle drei Regleranteile aus Abbildung 4.15 summiert wieder.

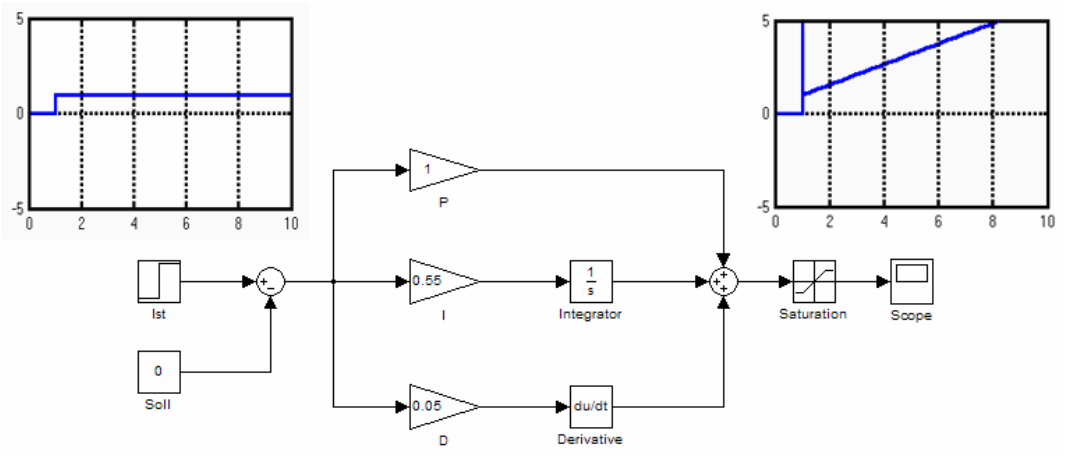


Abbildung 4.16: Modell eines Analog PID Reglers

Der Istwert am Reglereingang liegt jedoch nicht als Hubhöhe in cm, sonder als Spannungswert des Sensors vor. Das Übertragungsverhalten des Sensors muss daher ins Modell aufgenommen werden. Aus Abschnitt 3.2.2 ist folgender genäherter Zusammenhang für die Ermittlung der Distanz in Abhängigkeit von der Ausgangsspannung des Sensors bekannt.

$$x = \frac{Y_1}{U - Y_2};$$

Diese Formel lässt sich problemlos in Simulink nachbilden (Abbildung 4.17). Sie wird, zusammengefasst als Subsystem, in die Signalleitung des Ist-Werts geschalten.

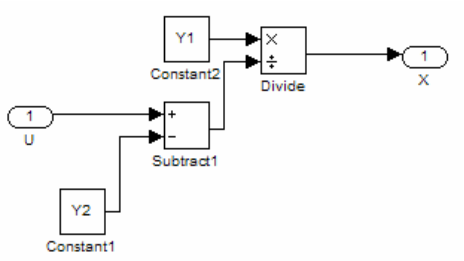


Abbildung 4.17: Subsystem U=>x

Der minimale Abstand zwischen Sensor und Messfläche liegt bei 40 mm. Durch Addition dieses Wertes zur Ankerposition x wird diese Eigenschaft ins Modell übernommen. Innerhalb des Reglers wird der Wert wieder subtrahiert.

Auch das Stellsignal am Reglerausgang ist kein Spannungswert, sondern ein PWM-Signal dessen Pulsweite proportional zur Spannung am Hubmagneten ist. Wie im Abschnitt für die Hardwareimplementierung noch festgelegt wird, wird das Verhältnis der Pulsweite im Vergleich zu Gesamtperiodendauer als Ganzzahlwert von 0 bis 5000, beim Mikrocontroller und von 0 bis 10000, beim FPGA, dargestellt. Übertragen auf die Stellgröße bedeutet dies 0 entspricht 0V und 5000 bzw. 10000 entsprechen 12V an den Spulen des Hubmagneten. Das Stellsignal ist demnach mit $5000 \div 12 \approx 416{,}67$ bzw. $1000 \div 12 \approx 833{,}33$ zu multiplizieren.

Das Gesamtmodell des Reglers ist der Übersicht wegen in einem Subsystem zusammengefasst. Dies ist auch notwendig, um den Regler für die spätere Codegenerierung von der Regelstrecke abzugrenzen.

Außerhalb des Reglersubsystems werden dann die jeweiligen Umkehroperationen, sowohl für Eingangs- als auch für Ausgangssignale, eingefügt. Diese Umkehroperationen repräsentieren das Übertragungsverhalten von Infrarot Distanzsensor und PWM-Verstärker.

4.3.2 Ermittlung der Regelparameter

Damit der Regler auch korrekt funktioniert, d.h. stabil ist und ein gutes Führungsverhalten zeigt, müssen die Reglerparameter, in diesem Fall die einzelnen Faktoren für den P, I und D Anteil, bestimmt werden.
Zur Bestimmung der Parameter wurden die Einstellregeln nach Ziegler und Nichols herangezogen[6].

Das Vorgehen ist dabei folgendes:
Der Regler wird anfangs als P-Regler benutzt, d.h. die Faktoren des I und D Anteils werden auf 0 gesetzt.
Der P-Anteil des Reglers wird nun soweit erhöht, bis er grenzstabil wird. Dies ist der Fall, wenn der Istwert um den Sollwert in eine Dauerschwingung gerät. Mit einem Proportionalfaktor von $K_{kri} = 250$ ist das der Fall. Aus Abbildung 4.18 kann die Schwingungsdauer T_{kri} von 0.7s abgelesen werden.

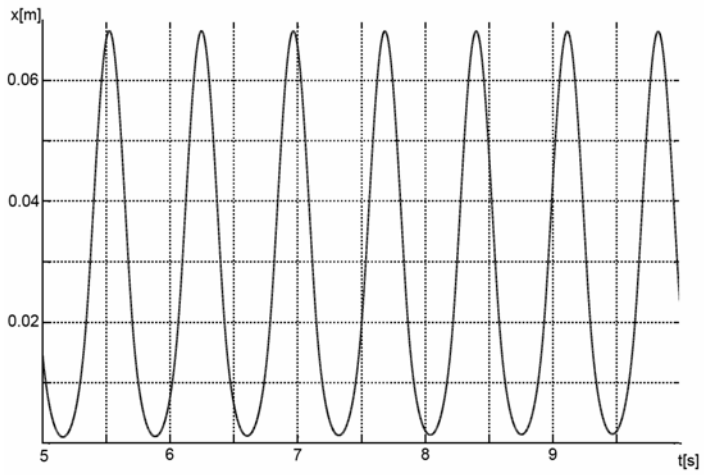


Abbildung 4.18 : Grenzstabile Schwingung des Reglers

Aus diesen Werten lässt sich laut Ziegler und Nichols Vorhaltezeit und Nachstellzeit, sowie der Verstärkungsfaktor ermitteln.

$K_R = 0{,}6 \cdot K_{kri} = 0{,}6 \cdot 250 = 150;$ — K_R : Versträkungsfaktor [1]
$T_N = 0{,}5 \cdot T_{kri} = 0{,}5 \cdot 0{,}7s = 0{,}35s;$ — T_N : Nachstellzeit [1s]
$T_V = 0{,}12 \cdot T_{kri} = 0{,}12 \cdot 0{,}7s = 0{,}084s;$ — T_N : Nachstellzeit [1s]

Ein PID-Regler lässt sich mathematisch wie folgt darstellen:

$$u = K_R\left(e + \frac{1}{T_N}\int_0^t e dt + T_V \dot{e}\right);$$

u : Stellgröße
e : Regeldifferenz

Der Regler im Modell hat jedoch folgende Form:

$$u = K_P \cdot e + K_I \cdot \int_0^t e dt + K_D \cdot \dot{e};$$

K_P : Proportionalfaktor
K_I : Integralfaktor
K_D : Differentialfaktor

Die gesuchten Parameter K_P, K_I und K_D erhält man durch Ausmultiplizieren der ersten Reglergleichung mit K_R.

$$K_P = K_R = 150;$$
$$K_I = \frac{K_R}{T_N} = \frac{150}{0.35s} \approx 428,57;$$
$$K_P = K_R \cdot T_V = 150 \cdot 0,084s = 12,6;$$

Setzt man die so ermittelten Werte als P, I und D Faktoren in das Reglermodell ein, verläuft der Istwert der Regelstrecke wie in Abbildung 4.19 gezeigt. Die Sollwertvorgabe wurde auf 0.045m eingestellt.

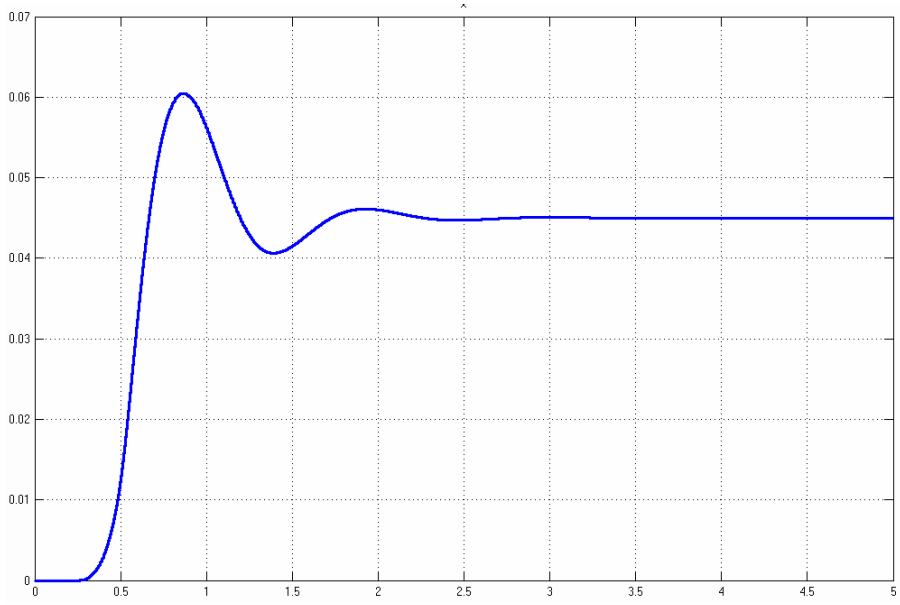


Abbildung 4.19: Regelungsverlauf mit Parametern nach Ziegler und Nichols

4.3.3 Zeitdiskreter Regler

Bei digitalen Reglern handelt es sich immer um Abtastsysteme. Die Erfassung der Messwerte und Berechnung der Stellwerte erfolgt zeitdiskret.

Das bisher entworfene Modell entspricht dem eines Analogreglers. Für alle dynamischen Elemente, das sind im Regler Integrator und Differenzierer, wurden Blöcke der Kategorie *Continuous* verwendet.
Um einen Regler zu erhalten, der dem realen, digitalen Regler gleich kommt, muss dieser stetige Regler in einen diskreten umgewandelt werden.
Die Umwandlung ist notwendig, denn nur aus diskreten Modellen kann überhaupt Code generiert werden. Hierfür sind folgende Schritte notwendig.

Es muss ein Abtastintervall, im Modell als Sample time bezeichnet, festgelegt werden.
Je niedriger dieses ausfällt, desto ähnlicher verhält sich der diskrete zum stetigen Regler.
Für das Modell werden 38ms gewählt. Dies entspricht dem Zeitintervall, in dem der Infrarotsensor neue Messwerte liefert. Abtastraten die darüber hinausgehen, bringen keine Vorteile, da die Stellwertberechnungen dann mehrfach auf dem gleichen Messwert beruhen, was im Falle des Differenzialanteils des Reglers sogar zu einem fehlerhaften Wert führt.

Die zweite nötige Anpassung ist das Einfügen von Rate Transition-Blöcken an den Übergangsstellen vom stetigen zum diskreten Modell. Diese sorgen dafür, dass ein stetiger Eingangswert oder ein Wert mit anderer Abtastrate zu einem Ausgangswert mit vorgegebener Abtastrate gewandelt wird. Dies wird dadurch realisiert, indem der Eingangswert zum Abtastzeitpunkt gespeichert und für die Dauer des Abtastintervalls am Ausgang zur Verfügung gestellt wird.
Da das Modell der Regelstrecke weiterhin stetig bleibt, ist ein Rate Transition-Block am Eingang des Reglers erforderlich.

Wie bereits beschrieben, wurden für die dynamischen Elemente des Reglers Blocks aus der *Continuous*-Kategorie der Simulink-Bibliothek verwendet. Diese müssen durch ihre Pedanten aus der *Discrete*-Kategorie ersetzt werden. Für das Reglermodell sind das der Integrator-Block aus dem I-Zweig des Reglers und Derivative-Block aus dem D-Zweig. In den Eigenschaftsdialogen dieser Blocks lassen sich die Abtastzeiten einstellen.

4.3.4 Anti Wind Up - Algorithmus

Ein wichtiger Bestandteil, der bei digitalen Reglern mit Integralem Anteil nicht fehlen darf, ist ein so genannter Anti Wind Up-Algorithmus. Im Gegensatz zu analogen Reglern, die durch ihre Realisierung mit elektronischen Bausteinen bereits technisch bedingte Grenzen für ihren Ausgangswert haben, besteht innerhalb von Digitalreglern kein direkter Zusammenhang zwischen numerisch ermittelten Stellsignal und dem tatsächlich an der Reglerstrecke anliegenden Stellwert. Diese Fehlende Rückkopplung kann nicht einfach durch Beschränkung des Reglerausgangs kompensiert werden.
Dies wird deutlich, wenn man das interne Verhalten des Reglers betrachtet, im Falle dass der Regler die Regeldifferenz nicht ausgleichen kann. Das ist beispielsweise möglich, wenn die Regelstrecke blockiert ist oder die nötige Hilfsenergie fehlt.

Setzt man eine solche Situation voraus, so summiert sich der Integralwert des Reglers immer weiter auf, obwohl die Ober- oder Untergrenze des Stellsignals bei weitem überschritten ist. Selbst wenn die Regelstrecke später wieder einwandfrei funktioniert, wirkt der aufsummierte Integralanteil so dominant auf das Stellsignal, dass keine Regelung erfolgt.
Demzufolge muss nicht nur der Ausgangswert des Integrators, sondern auch der im Integrator gespeicherte Wert eingeschränkt werden.

Um dies zu realisieren, werden die Signalleitungen vor und hinter dem am Reglerausgang befindlichen Saturation-Block voneinander subtrahiert. Die Differenz ist der Wert um den der Stellwert zu groß oder klein ist. Er wird durch ein Subtraktionsglied vom Eingang des Integrator-Blocks abgezogen. Mit dieser Methode wird verhindert, dass der Integralanteil überläuft und der Regler reaktionsschnell bleibt.

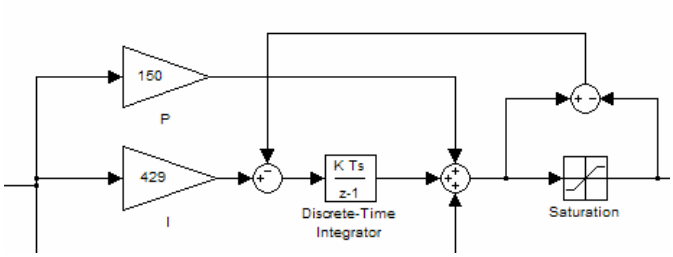


Abbildung 4.20: Anti Wind Up-Algorithmus

Abbildung 4.20 zeigt den diskreten Regler mit Rate Transiton-Block am Eingang und Anti Wind Up-Algorithmus. Dieser Regler dient als Grundlage für die Implementierung auf Mikrocontroller und FPGA.

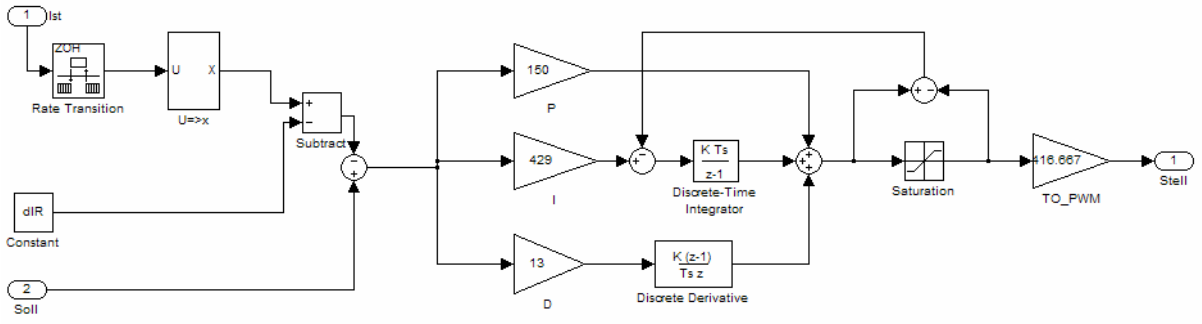


Abbildung 4.21: Diskreter PID-Regler

5 Implementierung des Reglers auf dem Infineon XC167 Mikrocontroller

In diesem Abschnitt wird erläutert, welche Änderungen des diskreten Reglermodells aus dem letzten Kapitel durchgeführt werden müssen, damit aus ihm ein lauffähiges Mikrocontroller-programm generiert werden kann.
Vorab müssen dabei Überlegungen bezüglich der verwendeten Hardwarekomponenten und der Softwarearchitektur getroffen werden. Aus diesen Vorgaben wird ein Programmgerüst erstellt, in das der, aus dem Reglermodell generierte, C-Code eingebettet wird.
Darüber hinaus muss das Modell um Schnittstellen zum Ansprechen der Mikrocontroller-spezifischen Hardware ergänzt werden.

5.1 Verwendete Hardwarekomponenten und Softwarearchitektur

Wie bereits angesprochen, lässt sich aus Simulink-Modellen C-Code generieren. Soll aus diesem die Hardware des Zielsystems direkt angesprochen werden, müssen spezielle Elemente, so genannte S-Function-Blöcke, in das Modell aufgenommen werden. Sie repräsentieren die Hardwareschnittstellen im Modell.

Die S-Functions, wobei S für „System“ steht, beinhalten benutzerdefinierten Code, der zwei Aspekte abdeckt. Zum einen das Verhalten der Hardware innerhalb der Simulation. Zum anderen den hardwarespezifischen Code, durch den Block nach der Codegenerierung vertreten wird. Die einzelnen, für die Mikrocontroller verwendenden, S-Functions werden im Abschnitt 5.3 beschrieben.
An dieser Stelle ist nur wichtig zu wissen, dass bereits vorab bekannt sein muss, welche Peripherie des Mikrocontrollers verwendet wird, bevor eine Anpassung des Modells an die hardwarespezifischen Gegebenheiten erfolgt.
Es ist auch möglich parametrisierbare S-Function-Blöcke für die Peripherie des Zielsystems zu erstellen. Diese können dann per Dialogfeld in Simulink konfiguriert werden. Der Aufwand beim Erstellen solcher Blöcke ist aber weitaus höher und lohnt sich nur, wenn mehrere Projekte für das gleiche Zielsystem geplant sind.

Nur aus zeitdiskreten Modellen kann Simulink Code generieren.
Diese Eigenschaft muss auch für das erstellte Programm erhalten bleiben. Deshalb muss der Code aus diskreten Modellen regelmäßig, im Zeitintervall, das in der Simulation festgelegt wurde, ausgeführt werden.
Für die Mikrocontroller-Implementierung bedeutet dies, dass der generierte Algorithmus, der mit der Funktion `One_Step()` aufgerufen wird, durch einen zyklischen Task eines Betriebssystems, innerhalb einer Interrupt Service Routine (ISR) eines Timers oder der Realtime Clock angestoßen wird.

Wie in Kapitel 3.2.4 beschrieben, benutzt der Mikrocontroller als Digitalregler einen AD-Wandler zur Messwerterfassung des Infrarot Distanzsensors, eine pulsweitenmoduliertes Signal als Stellsignal und eine asynchrone serielle Schnittstelle für die Parameter- und Sollwertübergabe.
Zur Istwerterfassung wird der erste der 16 Kanäle des 10-Bit AD-Wandlers des XC167 verwendet.
Der zyklische Aufruf des Algorithmus wird mit dem Timer 2 sichergestellt, in dessen ISR der entsprechende Funktionsaufruf durchgeführt wird.

Neben der verwendeten Hardware spielt auch der zeitliche Ablauf des Mikrocontrollerprogramms eine wesentliche Rolle. Ein Betriebsystem kommt nicht zum Einsatz, weshalb der zeitliche Ablauf selbst organisiert werden muss.

Die in diesem Projekt angewandte Methode ist die Ablaufsteuerung unter Zuhilfenahme der Interrupt-Priorität. Interrupt Service Routinen mit hohem Interruptlevel lösen solche mit niedrigerem Interruptlevel ab.
Je wichtiger die Ausführung eines Programmteils ist, desto höher ist die zugeordnete Interrupt Priorität.

Um die Regelung optimal zu gestalten, muss nach jedem Abtastzeitpunkt möglichst schnell ein aktueller Stellwert berechnet werden. Also hat für den Regler das Messen des Abtastwerts mit anschließender Berechnung des Stellsignals höchste Priorität. Folglich wird auch der Interruptlevel für den Timer, der den Regelalgorithmus aufruft, hoch, bei Interruptlevel 13, angesetzt.
Des Weiteren wird der Aufruf des Algorithmus nicht nur durch eine Variable getriggert, was durchaus denkbar wäre, sondern direkt in der Interrupt Service Routine ausgeführt. Dies verhindert ein Unterbrechen durch andere Interrupts.

Die Abarbeitung seriell eingehender Daten ist weniger wichtig. Es muss jedoch darauf geachtet werden, dass keine Daten bei der Übertragung durch Nichtabhohlung verloren gehen. Dies könnte Beispielsweise der Fall sein, wenn der Regelalgorithmus fast die gesamte Rechenzeit des Mikrocontrollers beansprucht oder die Übertragungsgeschwindigkeit der eingehenden Daten so groß ist, dass während der Abarbeitung des Regelalgorithmus der Empfangspuffer überläuft.
Um die Geschwindigkeit des Regelalgorithmus zu ermitteln, wurde eine erste lauffähige Implementierung des Modells ohne Hardwareschnittstelle auf dem Mikrocontroller XC167 bei 20 MHz zum Ablauf gebracht. Vor dem Start des Algorithmus wird ein Ausgangspin des Mikrocontrollers gesetzt bzw. nach Beendigung der Abarbeitung wieder zurückgesetzt. An dem Pin wurde eine Zeitmessung mittels Oszilloskop durchgeführt.

Das Ergebnis ergab eine Rechenzeit von ca. 1000µs zum Ermitteln des Stellsignals. Zwar ist hierbei die Wandlungszeit von maximal 440µs des AD-Wandlers noch nicht enthalten, bei einer Gesamtabtastrate von 0.038s bleiben immer noch ca. 0.0366s für alle anderen Aufgaben.

Für die maximale serielle Übertragungsgeschwindigkeit ohne Datenverlust bei einem 8 Byte Puffer ergibt sich aus Gl.5.1 eine maximale Übertragungsrate von 55555 Baud/s. Es wurde dabei mit 1 Startbit, 1 Stopbit und ohne Paritibit gerechnet.

Gl.5.1 $$Baudrate = \frac{8Byte}{0.00144s} \cdot 10\frac{Bit}{Byte} \approx 55555Baud;$$

Aus diesen Vorüberlegungen ergibt sich der zeitliche Ablauf, wie in Abbildung 5.1 gezeigt. Die einzelnen Programmteile kommen gemäß ihrer Interrupt-Priorität zur Ausführung. Der Regelalgorithmus kann aufgrund des höheren Interruptlevels nicht durch die ISR der seriellen Schnittstelle unterbrochen werden.

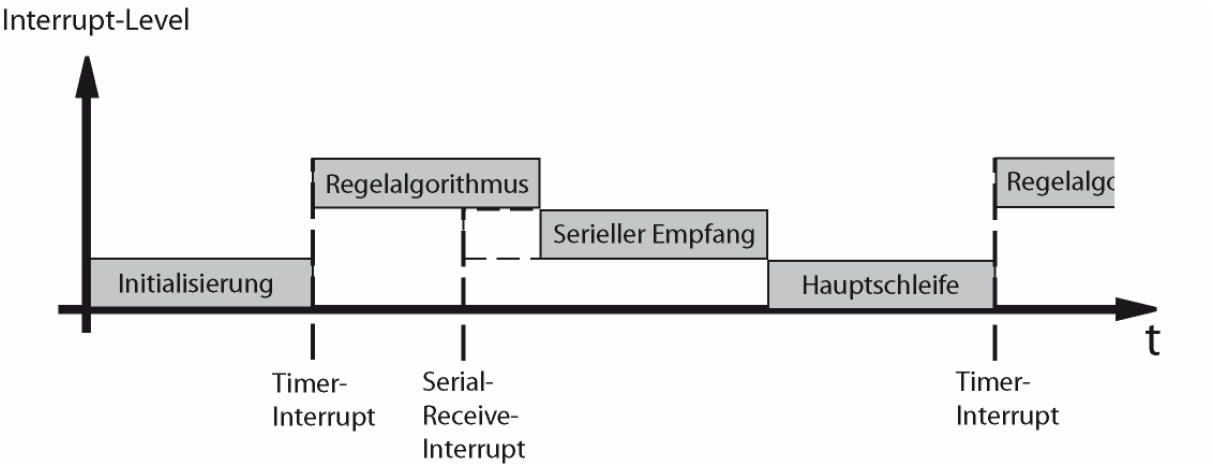


Abbildung 5.1: Zeitlicher Ablauf des Mikrocontrollerprogramms

Der Mikrocontroller als Digitalregler soll eine Sollwertvorgabe sowie die Parameter für P, I und D-Anteil des Reglers von einem PC erhalten. Dazu ist der Mikrocontroller mit der seriellen Schnittstelle eines Steuer-PCs verbunden.

Der XC167 Mikrocontroller verfügt über zwei serielle Schnittstellen ASC0 und ASC1. Erstere wird bereits von einem Monitor-Programm belegt. Dieses ist Bestandteil der Keil Entwicklungsumgebung und ermöglicht das Programmieren des Mikrocontrollers und das Debuggen von Programmen zur Laufzeit.
Deshalb wurde die zweite Asynchrone Schnittstelle ASC1 für die Konfiguration des Digitalreglers verwendet. Ein zusätzlicher Pegelwandler ist notwendig um den TTL-Pegel 5V auf RS232-Pegel zu wandeln und umgekehrt.

Um die Kommunikation mit dem PC zu gewährleisten müssen folgende Einstellungen für die serielle Schnittstelle gewählt werden:

Baudrate: 38400
Datenbits: 8
Parität: keine
Stopbits: 1
Flusskontrolle: keine

Das Nachrichtenformat wurde so gewählt, das die Einstellungen des Digitalreglers auch über ein Terminalprogramm, wie dem in Windows mitgelieferten HyperTerminal, vorgenommen werden kann. Eine Nachricht setzt sich aus ASCII-Zeichen zusammen und pro Nachricht wird ein Kommando übermittelt.

A Kommando-Byte [Vorzeichen] [ASCII-Zahlenwert] E

Folgende Kommandobytes sind verfügbar:

X – Sollwert Hubhöhe in mm 0 bis100
P – Verstärkungsfaktor der Regeldifferenz im P-Anteil -9999 bis 9999
I – Verstärkungsfaktor der Regeldifferenz im I-Anteil -9999 bis 9999
D – Verstärkungsfaktor der Regeldifferenz im D-Anteil -9999 bis 9999

Bsp.:
AX45E Sollwert der Hubhöhe 45mm.
AP+150E Proportionalfaktor des Reglers auf 150 setzten.

Die Nachrichtenauswertung erfolgt in einem Parser, der sich direkt in der Empfangs Interrupt Service Routine der seriellen Schnittstelle befindet.

5.2 Grundgerüst des Mikrocontrollerprogramms

Das Grundgerüst des Programms wird, aus den Vorgaben des letzten Abschnitts heraus, mit DAvE generiert. Dabei handelt es sich um eine Software von Infineon, mit der sich die Basissoftware für Mikrocontroller dieses Herstellers erzeugen lässt.
Für die einzelnen Mikrocontroller von Infineon können Vorlagen geladen werden, die das Programm über die verfügbaren Hardwarekomponenten und Register informiert.
Die Konfiguration des Mikrocontroller, die über gezieltes Setzten von Spezial Funktion Registers umgesetzt wird, erfolgt dialogfeldbasierend. Bei den Dialogfeldern handelt es sich aber nicht nur um die Abbildung der einzelnen Register auf Formulare. Auch die logischen Zusammenhänge und architekturbedingten Restriktionen werden beachtet. Zudem werden die Registerfunktionen textuell erklärt. Abbildung 5.2 zeigt das DAvE- Hauptfenster. Oben rechts findet sich der Architekturüberblick, unten rechts ein Konfigurationsdialogfeld und links die Peripheriekomponenten des XC167.

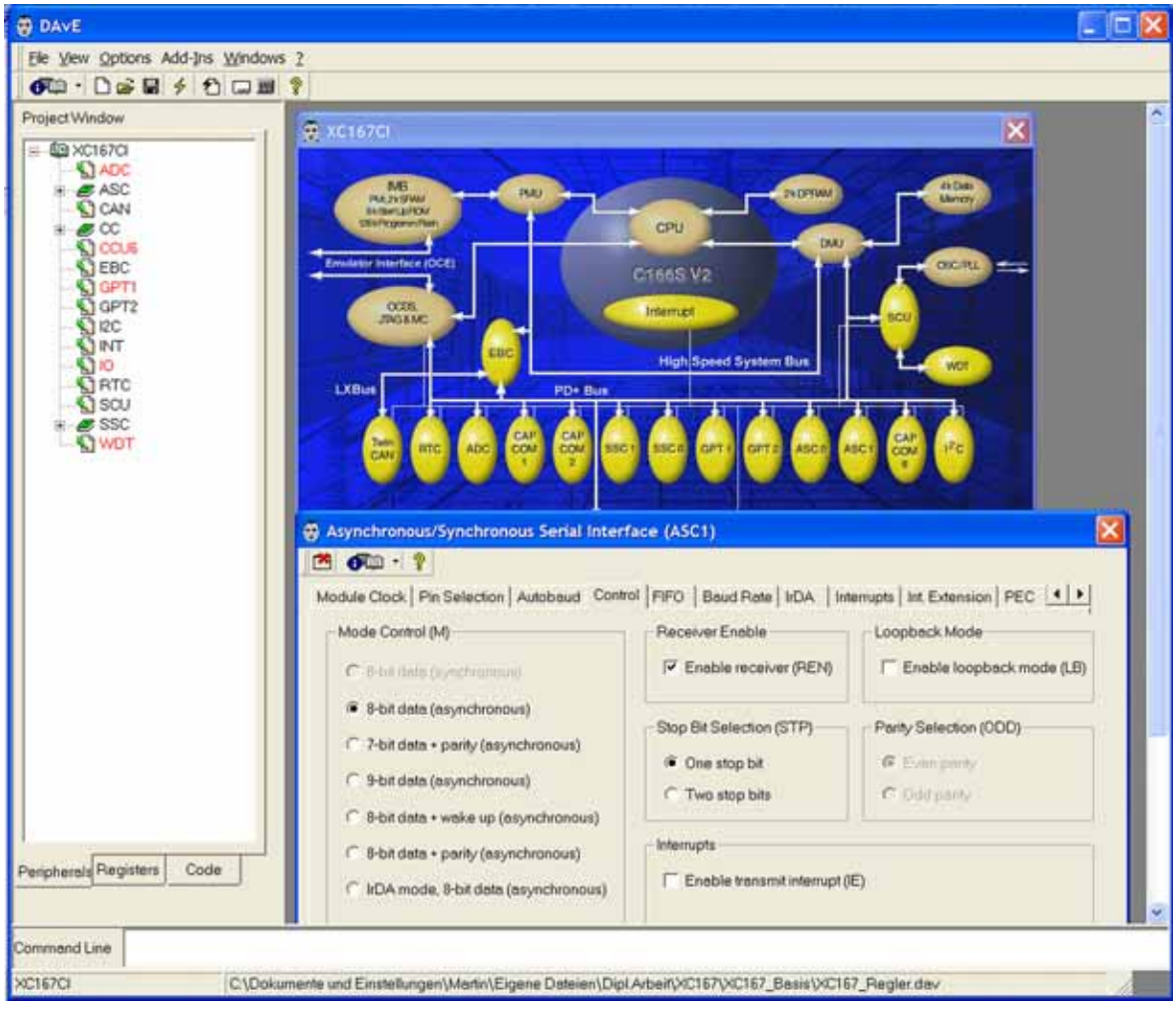


Abbildung 5.2: Programmoberfläche von DAvE

Ist die Konfiguration fertig gestellt, lässt sich das Codegerüst der Basissoftware generieren. Für jede der benutzen Hardwarekomponenten wird eine C-Datei plus eine Header-Datei erzeugt. Je nach getroffener Auswahl werden mehr oder weniger viele C-Funktionen zum Ansprechen der Peripherie bereitgestellt.

Die in DAvE gewählten Einstellungen sind nachfolgend aufgelistet:

Systemkonfiguration:

Eingangsfrequenz:	8 MHz
Systemtakt:	20 MHz (Fcpu)

Timer:

General Purpose Timer Unit (GPT1):

Konfiguration Timer 2:

Betriebsart:	Timer
Vorteiler:	16
Zählrichtung:	Abwärts
Startwert:	0xB98B (entspricht 38ms)

Analog-Digital-Wandler:

Betriebsart:	fixed channel single conversion mode
Auflösung:	10-bit resolution
Wandlungszeit:	89,900 µs
Eingang:	Pin 5.0

Pulsweitenmodulation:

Konfiguration des CCU6 Timers 12:

Vorteiler:	1
Betriebsart:	edge aligned mode
Periodendauer:	250µs

Konfiguration des CCU6 Channel 0:

Betriebsart:	Compare mode
Ausgang:	Pin 1L.1

Serielle Schnittstelle:

Asynchronous/Synchronous Serial Interface (ASC1):

Baudrate:	38400 Baud
Übertragung:	8 Datenbists, keine Pariät, 1 Stopbit
FIFO:	8 Byte
Ausgang:	Pin 3.0 (TxDA1)
Eingang:	Pin 3.1 (RxDA1)

Interrupts:

Timer 2:	Interrupt priority level (ILVL) = 13
	Interrupt group level (GLVL) = 0
ASC1:	Rx interrupt priority level (ILVL) = 9
	Rx interrupt group level (GLVL) = 0

Watchdog:

Eingangsfrequenz:	Fcpu / 128
Watchdog Überlauf:	78,643 ms

Das Programm kann dann aus diesen Vorgaben ein Programmgrundgerüst erstellen. Dieses steht dann als Projekt für die Keil µVision Entwicklungsumgebung zur Verfügung.

5.3 Codierbares Modell des Reglers

Im Abschnitt 4.3 wurde das Modell eines diskreten Reglers erstellt. Dieses beinhaltet aber noch keinerlei Schnittstellten zur Hardware des Zielsystems. Diese sind aber notwendig damit der generierte Code in Interaktion mit der Umgebung treten kann.

Die Schnittstellen zur Hardware werden in Simulink-Modellen mittels S-Functions hergestellt. Mit S-Function-Blöcken lässt sich benutzerdefinierter Code in den Programmiersprachen Ada , C, C++ oder Fortran in ein Modell einbinden. In diesem Projekt werden in C programmierte S-Funktionen verwendet.

Ein S-Function Block benötigt mehrere Dateien. Unter anderem eine C-Quellcode Datei, die das Verhalten des Blocks innerhalb der Simulation beschreibt, sowie eine Target Language Compiler-Datei, die den hardwarespezifischen Code des Zielsystems enthält. Vor dem Einsatz in einer S-Function muss dessen Programmcode mit dem MEX-Kompiler von Matlab übersetzt werden.

Die C-Datei, die das Verhalten des Blocks während der Simulation beschreibt, besteht aus einer Reihe Funktionen, die von der S-Function API festgelegt sind. Diese werden von der Simulationsumgebung zu unterschiedlichen Zeitpunkten aufgerufen.
Wichtige Funktionen für die Einbettung von Hardwareschnittstellen sind hierbei:

`mdlIntitialSizes()`: Diese Funktion wird während der Initialisierung des Modells aufgerufen. Es werden die Blockparameter, sowie die Ein- und Ausgänge des Blocks festgelegt.

`mdlInitializeSampleTimes()`: Legt die Abtastzeit des Blocks fest.

`mdlOutputs()`: Der Aufruf dieser Funktion erfolgt in jedem Simulationsschritt. Hier wird der Blockausgang berechnet. Es kann beispielsweise das Übertragungsverhalten der Hardwarekomponente, die durch diesen S-Funktion Block repräsentiert wird, nachgebildet werden.

`mdlTerminate()`: Verhalten des Blocks am Simulationsende.

Für den hardwarespezifischen Code, der den S-Function-Block des Modells nach der Codegenerierung ersetzt, muss eine .tlc-Datei erzeugt werden. In eine .tlc-Datei kann parametrisierbarer C-Code eingebettet werden.

Wie schon die C-Datei enthält auch die .tlc-Datei Funktionen.
Diese Funktionen geben aber nicht vor, wann der Funktionsinhalt ausgeführt wird, sondern an welcher Stelle im generierten Programmcode der Inhalt der Funktionsrumpfs platziert wird.

.tlc-Dateien enthalten folgende Funktionen:

`BlockTypeSetup()`: Enthält Präprozessoranweisungen oder Makros
`Start()`: Code, der im Initialisierungsteil des generierten Programms platziert wird
`Outputs()`: Anweisungen, die bei jedem Durchlauf des Algorithmus ausgeführt werden
`Terminate()`: Anweisungen am Programmende

Die Parameterübergabe und die Zuweisung von Ein- und Ausgängen innerhalb dieser Funktionen erfolgt über spezielle Funktionsaufrufe der S-Function-API.

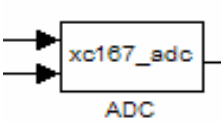
```
%assign py0 = LibBlockOutputSignal(0, "", "", 0)

ADC_vStartConv(); //Wandlung starten
while(ADC_ubBusy()==1); //Warten bis AD-Wandlung ausgeführt
%<py0> = (ADC_uwReadConv() & 0x03ff); // Die untersten 10-Bit auslesen
```

Das Beispiel zeigt einen Auszug aus der `Output()`-Funktion der .tlc-Datei des AD-Wandlers.
In der ersten Zeile erfolgt die Zuordnung einer Variablen zu einem Blockausgang mit Hilfe der Funktion `LibBlockOutputSignal()`. Für Einen Blockeingang würde die Funktion `LibBlockInputSignal()` heißen. Der Datentyp der Variable wird an anderer Stelle festgelegt.
Der Funktionsaufruf `ADC_vStartConv()`bezieht sich auf eine Funktion die von der Basissoftware bereitgestellt wird. Sie startet die Wandlung. Durch die While-Schleife wird gewartet bis die Wandlung abgeschlossen ist. Die dem Ausgang zugewiesene Variable wird nun mit dem Wert des AD-Wandler beschrieben.

Als Hilfe zum Erstellen von S-Functions steht der S-Function-Builders zur Verfügung. Mit ihm lässt sich schnell ein Grundgerüst sowohl für die C als auch für die .tlc-Datei erstellen. Über die Reiter des Dialogfelds lassen sich Ein- und Ausgänge des Blocks festlegen und das gewünschte Simulationsverhalten als C-Code formulieren. Für die .tlc-Datei lässt sich lediglich bestimmen, welche Funktionsrümpfe erzeugt werden sollen. Mit Hilfe der Generate-Schaltfläche werden die nötigen Dateien erzeugt und der C-Quellcode in eine .mex32-Datei übersetzt, die von der Simulationsumgebung zur Modellberechnung verwendet wird.

Im Folgenden werden die einzelnen Schnittstellen für die Reglerimplementierung auf dem Infineon XC167 Mikrocontroller und die dazu notwendigen S-Funktionblöcke kurz erläutert. Die zugehörigen .tlc-Dateien sind im Anhang D1 zu finden.

5.3.1 Schnittstelle zum AD-Wandler



Der S-Function-Block für den AD-Wandler enthält zwei Eingänge. Zum einen den Messeingang, zum anderen den Wert für die Referenzspannung auf die sich der AD-Wandler bezieht.

Das Simulationsverhalten wird in der Funktion `mdlOutputs()` durch folgenden C-Ausdurck beschrieben:

```
y0[0] = ((u0[0]*pow(2.,10.))/u1[0]);
```

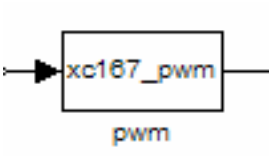
`u0` und `u1` sind die Variablen für den Blockeingang, y für den Blockausgang.
In diesem Fall wird der Messwert mit Auflösung des AD-Wandlers multipliziert und durch die Referenzspannung dividiert. Der so berechnete Wert entspricht dem, den der AD-Wandler des Mikrocontrollers bei gleicher Eingangsspannung liefert.

Die .tlc-Datei für AD-Wandler findet sich im Anhang D1. Als Auszug ist hier der Inhalt der Funktion `Outputs()` aufgeführt:

```
%assign py0 = LibBlockOutputSignal(0, "", "", 0)

ADC_vStartConv(); //Wandlung starten
  while(ADC_ubBusy()==1); //Warten bis AD-Wandlung ausgeführt
  %<py0> = (ADC_uwReadConv() & 0x03ff); // Die untersten 10-Bit auslesen
```

Dieser ist bereits weiter oben als Beispiel verwendet und erklärt.

5.3.2 Schnittstelle zum PWM-Modul



Der PWM-Ausgang des Mikrocontrollers wurde so konfiguriert, dass er Werte von 0 bis 5000 akzeptiert. Der an den S-Funktion-Block übergebene Wert ist dabei proportional zur Pulsweite des Ausgangssignals und damit auch proportional zur durchschnittlichen Ausgangsspannung.
Der Einfachheit halber wurde im Modell auf die Generierung eines PWM-Signals verzichtet. Das Verhalten des Blocks wurde auf den einfachen proportionalen Zusammenhang zwischen Eingangswert und durchschnittlicher Spannung reduziert:

```
y0[0] =u0[0]/416.67;
```

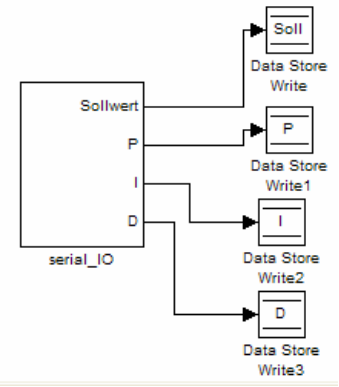
Durch die Division des Eingangswertes mit 416.67 sind Ausgangwerte von 0 bis 12 Volt möglich.

Im `Outputs()`-Funktionsrumpf der .tlc-Datei ist folgender Code zu finden:

```
%assign pu0 = LibBlockInputSignal(0, "", "", 0)

  CCU6_vLoadChannelShadowRegister_CCU6_CHANNEL_0(5000U - %<pu0>);
  CCU6_vEnableShadowTransfer_CCU6_TIMER_12();
```

Durch den ersten Ausdruck wird dem Blockeingang eine Variable zugewiesen.
Bei den beiden folgenden Aufrufen werden Funktionen angesprochen die vom DAvE-Programmgrundgerüst bereitgestellt werden. Die erste Funktion lädt den Wert in ein Register des PWM-Moduls, die zweite sorgt für die Übernahme in den Zähler des PWM-Moduls.

5.3.3 Serielle Schnittstelle



Der S-Function Block für die serielle Schnittstelle hat keine Eingänge. In der zugehörigen C-Datei werden den Ausgängen lediglich Default-Werte zugewiesen.

```
sollwert[0] = 0.045;
p_faktor[0] = 150.0;
i_faktor[0] = 428.0;
d_faktor[0] = 12.6;
```

Anders als bei den anderen S-Functions-Blöcken, spricht dieser nicht direkt die Hardware an. Der serielle Empfang verläuft autonom. Der Nachrichten-Parser wird bereits in der Interrupt Service Routine der seriellen Schnittstelle ausgeführt.
Die empfangenen Reglerparameter werden in globalen Variablen hinterlegt. Die einzige Aufgabe, die in der .tlc Datei erfüllt werden muss, ist die Übernahme dieser Variablen.

```
%assign sl_Soll = LibBlockOutputSignal(0, "", "", 0)
%assign sl_P = LibBlockOutputSignal(1, "", "", 0)
%assign sl_I = LibBlockOutputSignal(2, "", "", 0)
%assign sl_D = LibBlockOutputSignal(3, "", "", 0)

%<sl_Soll> = param_x;
%<sl_P> = p_faktor;
%<sl_I> = i_faktor;
%<sl_D> = d_faktor;
```

Mit der seriellen Schnittstelle, ist die Hardwareanbindung des Modells abgeschlossen. Abbildung 5.3 auf der folgenden Seite zeigt das vollständige Modell des Reglers für die Mikrocontrollerimplementierung. Zu sehen sind die S-Function-Blocks an den Ein- und Ausgängen des Modells. Die serielle Schnittstelle ist nicht direkt an die jeweiligen Blöcke angeschlossen, sondern speichert die Daten in Memory-Blocks zwischen. Mit Hilfe von Read- und Write-Blöcken können dort Daten gespeichert bzw. gelesen werden.

Die Konstanten Multiplikatoren für P, I und D Faktor wurden durch Multiplikatoren mit zwei Eingängen ersetzt, um die Regelparameter der seriellen Schnittstelle aufnehmen zu können.

Das gesamte Reglermodell (Abbildung 5.3) ist in ein Subsystem zusammengefasst. Wie weiter unten gezeigt wird, erleichtert dies die Codegenerierung.

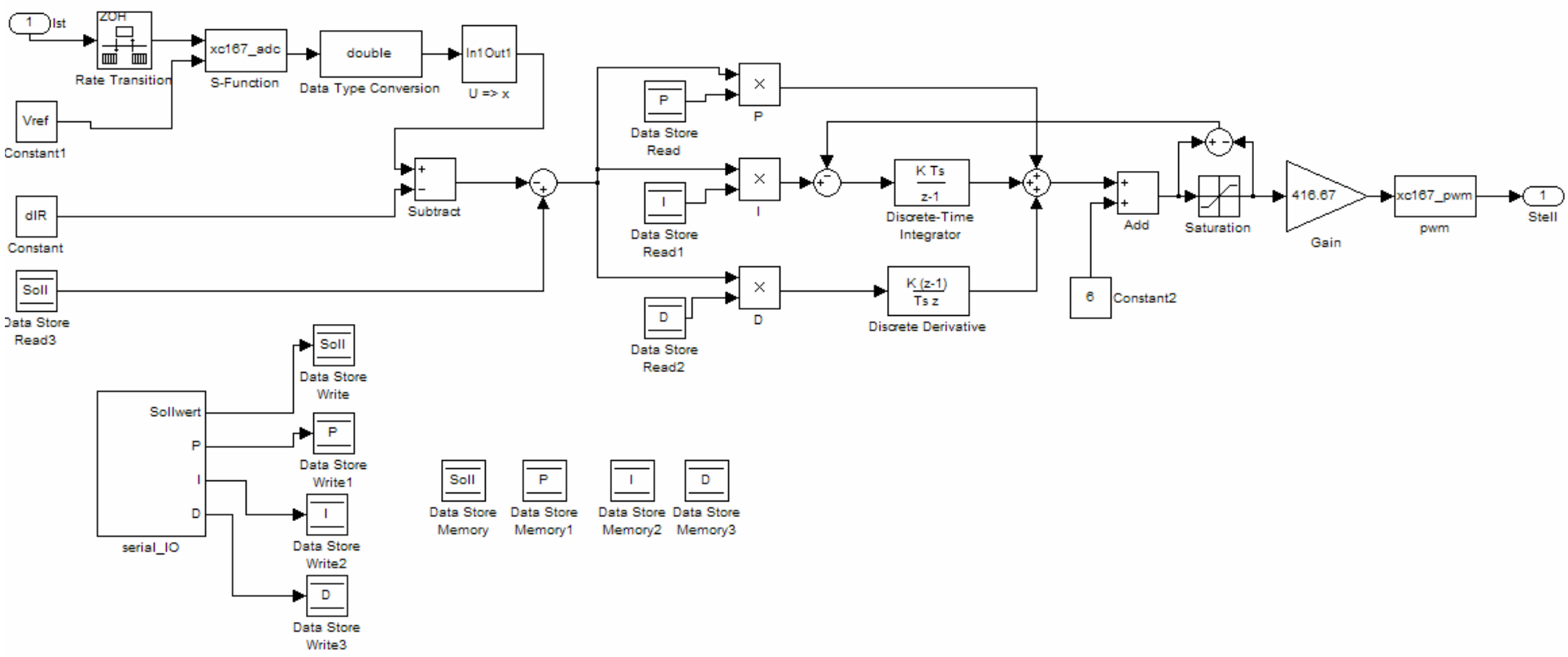


Abbildung 5.3: Reglermodell für Mikrocontroller

5.4 Codegenerierung

Die Codegenerierung in Simulink ist ein weitgehend automatisierter Prozess, der aus einem vorgegebenen Modell oder Subsystem übersetzbaren C-Code erzeugt. Lediglich einige Einstellungen in der Modellkonfiguration sind notwendig. Mit den richtigen Codevorlagen können sogar eigenständig lauffähige Anwendungen generiert werden.

5.4.1 Festlegung des Zielsystems

Im Konfigurations-Dialogfeld des Modells (Abbildung 5.4), zu finden im Menü des Simulink Arbeitsbereichs, unter `Simulation\Parameter Configuration…\Hardware Implementation`, können die Parameter des Zielsystems eingestellt werden. Gängige Plattformen befinden sich bereits in einer Auswahlliste. Darunter auch der XC167 von Infineon.
Die Einstellungen betreffen hierbei die Länge der einzelnen Datentypen, sowie die Byte-Order. Mit diesen Vorgaben lassen sich bereits während der Simulation Konflikte des Modells mit der Hardware des Zielsystems erkennen.

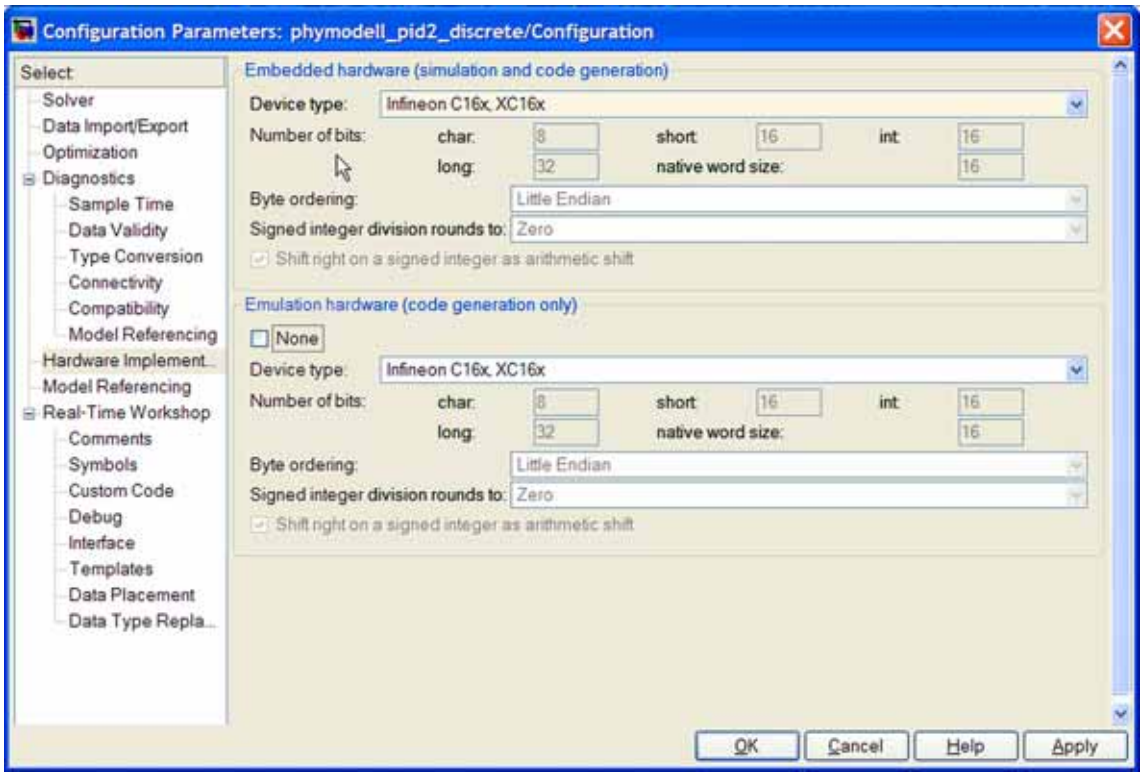


Abbildung 5.4: Simulink Hardwarekonfiguration

5.4.2 Voreinstellungen des Embedded Real-Time Workshop

Die für die Codegenerierung wichtigen Parameter werden Abschnitt Real-Time Workshop-des Modellkonfigurations-Dialogs vorgenommen (Abbildung 5.5).
Es können Programmschablonen in Form von .tlc-Dateien eingebunden werden, aber auch normale C-Source- oder C-Header-Dateien.
Ferner kann festgelegt werden, inwieweit der generierte Code kommentiert werden soll.
Außerdem ist es möglich, automatische Dokumentation in HTML Format erzeugen zu lassen.

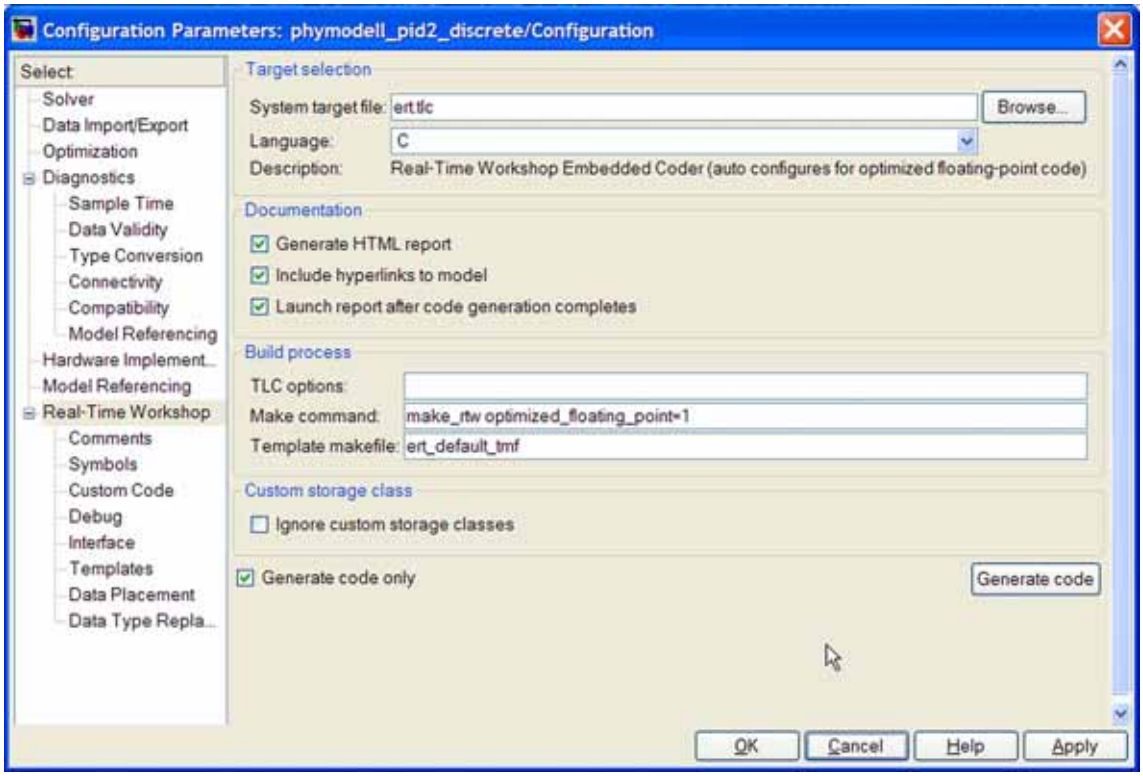


Abbildung 5.5: Real-Time Workshop Konfiguration

Da für den Mikrocontroller bereits ein Codegrundgerüst mit DAvE erstellt wurde, beschränken sich die Einstellungen für dieses Projekt auf einige wenige Punkte.

- Als Zielsprache wird C gewählt.

- Als System target file, das die `main()`-Funktion beinhaltet, wurde die Datei `ert.tlc` gewählt. Dies ist eine generische Datei für eingebettete Zielsysteme.

- Für den Generierungsprozess wurde der Parameter optimized_floating_point=1 angefügt, da im Reglermodell, Fließkomma-Datentypen verwendet wurden.

- Die Option zur Generierung von Dokumentation wurde ausgewählt.

5.4.3 Ausführen der Codegenerierung

Die Codegenerierung kann sowohl für ein gesamtes Modell, als auch für ein ausgewähltes Subsystem erfolgen. Die zweite Methode kommt in diesem Fall zum Einsatz, da die Generierung des Gesamtmodells die Regelstrecke beinhalten würde, aber nur der Regler benötigt wird.
Die Durchführung erfolgt durch Rechtsklick auf das Subsystem und durch Auswahl des Punkts *Real-TimeWorkshop/Build Subsystem…* (Abbildung 5.6).

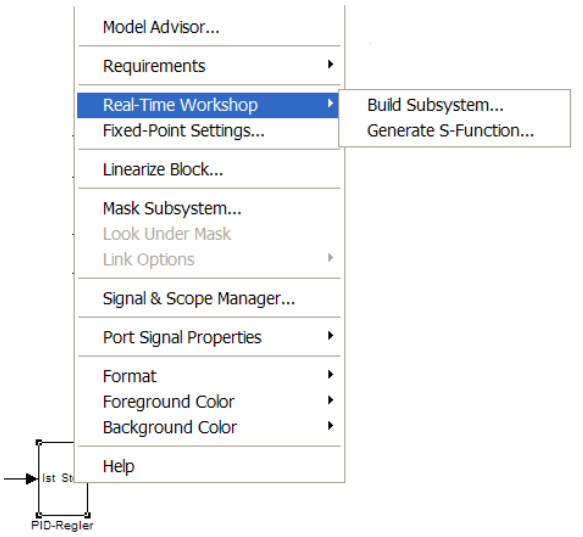


Abbildung 5.6: Durchführung der Codegenerierung

Nach Beendigung der Codeerstellung erscheint ein Bericht über den Generierungsprozess in einem neuen Fenster.

Folgende Dateien werden bei der Codegenerierung erzeugt bzw. angefügt:

`ert_main.c`: Nach einer Vorlage generierte C-Datei mit `main()`-Funktion
`PID.h`: Header-Dateien und Definitionen von Datenstrukturen für den Regelalgorithmus
`PID.c`: In dieser Datei befindet der Algorithmus, aus dem Simulink-Modell.
`PID_private.h`: Header-Datei die nur von `PID.C` eingebunden wird
`PID_types.h`: Modellspezifische Datentypen
`rtwtypes.h`: Definition von Standarddatentypen die vom Real-Time Workshop verwendet werden.

Das Präfix der Dateien, hier PID, folgt aus dem Namen des Subsystems aus dem sie generiert wurden.

Die beiden Dateien `PID.h` und `PID.c` enthalten den generierten C-Code des Reglermodells. Sie sind im Anhang D2 aufgeführt. In PID.c sind zwei Funktionen zu finden.
`PID_initialize()`für die Initialisierung des Algorithmus und `PID_step()`zum Ausführen des Algorithmus.

Der erzeugte Quellcode enthält Standarddatentypen, die über Typendefinition neue Namen erhalten. Ebenso werden Datenstrukturen aus diesen Typen verwendet.

Der Code, der in der Datei `PID.c` zu finden ist, bildet das Modell als sequentiellen Algorithmus ab. Die Namen der Variablen werden von den Blocknamen abgeleitet, dessen Berechnungsergebnis sie enthalten.

Mathematische Standardoperationen wie Addition, Division usw., werden direkt in den C-Code umgewandelt. Blöcke die sich nicht durch C-Operation oder Standardfunktionen darstellen lassen, wie Integration, Ableitung oder Sättigung, werden über Aufrufen von Methoden der `rtlibsrc`-Bibliothek in den Code eingebettet.

5.5 Einbettung in das Grundgerüst

Die Einbettung des Regelalgorithmus erfolgt in die Entwicklungsumgebung Keil µVision. Das mit dem Programm DAvE generierte Programmgrundgerüst liegt bereits als µVision-Projektdatei vor. Lediglich der von Simulink generierte Regelalgorithmus muss eingebunden werden.

Wie für C-Programme üblich, erfolgt die Bekanntmachung der entsprechenden Quellcodedatei durch Einbinden einer Header-Datei gleichen Namens.

Aus der generierten Hauptdatei `ert_main.c` wird die Funktion `One_Step()` in die eigene Main-Datei übernommen.
Der Aufruf dieser Funktion, die ihrerseits den Regelalgorithmus zum Ablauf bringt, muss in regelmäßigen Abstand erfolgen.
Dies erfolgt aus der Timer-ISR `GPT1_viTmr2()` heraus, die vom Codegrundgerüst bereits zur Verfügung gestellt wird. Nur der Funktionsaufruf `One_Step()` ist dort einzutragen.

Zum erfolgreichen Kompilieren ist es notwendig, die Matlab Bibliothek `rtlibsrc` einzubinden. Diese enthält alle vom Regelalgorithmus benötigten Funktionen. In den Kompilierungseinstellungen von µVision erfolgt das Einbinden durch eine entsprechende Pfadangabe.

Zu bemerken ist, dass die Anpassung des Projekts nur einmal durchgeführt werden muss, solange sich die Schnittstellten zum Modell nicht ändern. Bei erneuter Codegenerierung werden lediglich die alten Dateien überschrieben.

6 FPGA-Implementierung des Reglers

In diesem Kapitel wird die Implementierung des Reglers auf einem programmierbaren Logikbaustein behandelt. Als Hardware dient ein Spartan-3E FPGA-Board von Xilinx.
Wie schon beim Mikrocontroller, basiert der Regler auf dem diskreten Simulink-Modell eines solchen.
Es werden zwei Methoden der Implementierung gezeigt.
Bei der ersten Variante wird der Simulink HDL-Coder verwendet. Dieser kann aus einer Teilmenge des Standard-Blocksets VHDL- oder Verilog-Code generieren.
Bei der zweiten Variante erfolgt die Modellbildung mit dem Xilinx System Generator, einer Blockset-Erweiterung für Simulink.
Sowohl HDL-Coder als auch System Generator benutzten als Schnittstelle zur Peripherie des FPGA-Boards VHDL-Module, die als erstes in diesem Kapitel behandelt werden.

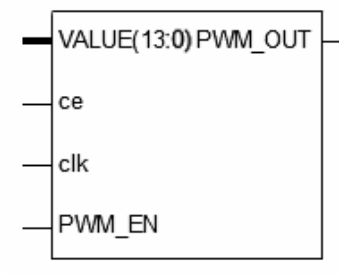
6.1 VHDL-Schnittstellenmodule

Im Laufe der Diplomarbeit hat sich herausgestellt, dass sich für bit- und zyklusgenaue Aufgaben, die Umsetzung mittels gängiger Hardwarebeschreibungssprachen besser eignet, als die im Modell.
Aus diesem Grund wurden für die Ansteuerung der Peripherie des Spartan-3E Boards, sowie für alle anderen Schnittstellen des Reglermodells nach außen, VHDL-Module entwickelt.
Dies sind Module für den Pulsweitenmodulator, den Analog-Digital-Wandler mit Vorverstärker und die serielle Schnittstelle. Die gleichen Module werden sowohl für die HDL-Coder Implementierung als auch für die des System Generators verwendet.
Wie im Abschnitt 6.3 noch gezeigt wird, erfolgt die Einbettung der VHDL-Module in das System Generator-Modell mit Black Box Elementen.
Dabei werden folgende Forderungen an VHDL-Module gestellt:
In der Schnittstellenbeschreibung (Entity) auf der höchsten Ebene dürfen nur die Typen `std_logic` und `std_logic_vector` vorkommen. Wird das Modul durch den Systemtakt angesteuert, müssen in der Schnittstelle zwei Eingänge existieren die den Teilstring „clk“ und „ce“ enthalten. Ersterer ist dabei der Systemtakt, während der Zweite diesen für das Modul freigibt. Die beiden Eingänge müssen paarweise auftreten.

6.1.1 Pulsweitenmodulator

Wie bei der Mikrocontroller-Implementierung, steuert auch das FPGA den Hubmagneten mittels Pulsweitenmodulation an. Die entsprechende Schnittstelle des VHDL-Moduls ist in Abbildung 6.1 zu sehen.

Schnittstelle:



VALUE:	Einschaltdauer von 0 bis 10000.
clk:	Taktsignals.
ce:	Taktfreischaltung.
PWM_EN:	Pulsweitenmodulation Ein/Aus
PWM_OUT:	PWM-Signal.

Abbildung 6.1: PWM-Modul

Funktionsweise:

Durch Vorgaben von 0 bist 10000 am VALUE-Eingang wird die Einschaltdauer im Verhältnis zur Gesamtperiode vorgegeben. 0 entspricht dabei einer Einschaltdauer von 0% und 10000 einer Einschaltdauer von 100%. Das pulsweitenmodulierte Signal liegt am PWM_OUT Ausgang an. Dieser wird einem Pin des FPGAs zugeordnet.
Um eine Periodendauer von 5 kHz zu erhalten, muss die Taktfequenz am clk-Eingang 50MHz entsprechen.
Durch den PWM_EN Eingang kann der Pulsweitenmodulator zusätzlich ein- und ausgeschaltet werden.

Implementierung:

Die Realisierung eines PWMs in einem FPGA gestaltet sich einfach, wie folgendes VHDL-Programm, das aufgrund seiner Kürze hier aufgeführt ist, zeigt.

```
-- PWM.vhd

entity PWM is
  Port ( clk : in  STD_LOGIC; --The onboard clk of the Spartan3E(C9)
         ce: in STD_LOGIC;                -- needed for System Generator
         PWM_OUT : out STD_LOGIC;               -- Output of PWM Signal
         VALUE : in  STD_LOGIC_VECTOR (13 downto 0); -- 0?10000 => 0-100%
         PWM_EN: in STD_LOGIC);                      -- enable PWM
end PWM;
```

```
architecture RTL of PWM is
  signal clk_int: integer range 0 to 10000;
begin

process(clk, PWM_EN)
  variable PWM_int: std_logic;
begin
  if PWM_EN /= '1' then
      clk_int <= 0;
   PWM_int := '0';

  elsif clk'event and clk = '1' then
      if ce = '1' then
            clk_int <= clk_int + 1;
            if clk_int <= conv_integer(unsigned(VALUE)) then
                  PWM_int := '1';
            else
                  PWM_int := '0';
            end if;
            if clk_int = 10000 then
                  clk_int <= 0;
            end if;
      end if;
  end if;
  PWM_OUT <= PWM_int;
end process;
end RTL;
```

Es handelt sich um einen Zähler der von 0 bis 10000 zählt. Innerhalb der Zählerschleife findet ein Vergleich der Zählervariablen und dem VALUE-Eingang statt. Solange der Zählerwert kleiner oder gleich dem VALUE-Wert ist, wird PWM_OUT auf ‚1' geschaltet, ansonsten auf ‚0'. So erhält man ein PWM-Singal, mit einer, am Periodenanfang befindlichen, Einschaltphase.

6.1.2 Analog-Digital-Wandler über SPI-Bus

Das Spartan-3E Starter Kit verfügt über einen 2-Kanal 14-Bit Analog-Digital-Wandler-Chip (Linear Technology LTC1407A-1). Diesem vorgeschaltet ist ein Verstärkerbaustein (Linear Technology LTC6912-1) der eine Spannungsverstärkung um die Faktoren 0, 1, 2, 5, 10, 20, 50 oder 100 vornehmen kann. Sowohl der AD-Wandler als auch der Verstärker sind über Serial Peripheral Interface (SPI) Bus angeschlossen.

Beim SPI-Bus handelt es sich um ein einfaches serielles Bussystem von Motorola zum Verbinden von Digitalbausteinen. Am Bus befindet sich immer ein Master, der den Zugriff auf die Leitungen steuert, und theoretisch beliebig viele Peripheriebausteine, die über drei gemeinsam genutzte Leitungen mit dem Master verbunden sind, anbindet. Diese sind:

SDO (*Serial Data Out*) oder MOSI (*Master out Slave in):* Daten vom Master zu den Peripheriebausteinen
SDI (*Serial Data In*) oder MISO (*Master in Slave out*): Daten von den Peripheriebausteinen zum Master.
SCKL (*Serial Clock*) Gibt den Takt für die Datenübertragung vor.

Zudem führen zu den Peripheriebausteinen noch Chip Select-Leitungen (CS).

Der Anschluss kann auf mehrere Arten erfolgen. Eine Möglichkeit ist, alle Bausteine parallel an die drei Leitungen anzuschließen. Jeder Baustein erhält eine Chip Select-Leitung, die zum Master führt.
Eine andere Möglichkeit ist, die Bausteine hintereinander zu schalten, SDO des einen Bausteins zu SDI des nächsten usw.. Alle Bausteine liegen an einer Chip Select-Leitung.
Auch eine Kombination aus beiden Anschlussarten ist denkbar, aber nur die Erstgenannte funktioniert für jeden Baustein.
Die Kommunikation ist keinem strikten Protokoll unterworfen. Das Funktionsprinzip ist zumeist folgendes. Der Master zieht die CS-Leitung der ausgewählten Peripherie auf „Low“ und startet dann ein Taktsignal auf die SCKL-Leitung. Gleichzeitig sendet er mit jedem Impuls auf der SCKL-Leitung ein Datenbit auf der SDO Leitung, bzw. empfängt Datenbits auf der SDI-Leitung.

Der AD-Wandler und der Vorverstärker des Spartan-3E Starter Kit sind parallel am SPI-Bus angeschlossen. Abbildung 6.2 zeigt die VHDL-Schnittstelle des AD-Wandler-Moduls.

Schnittstelle:

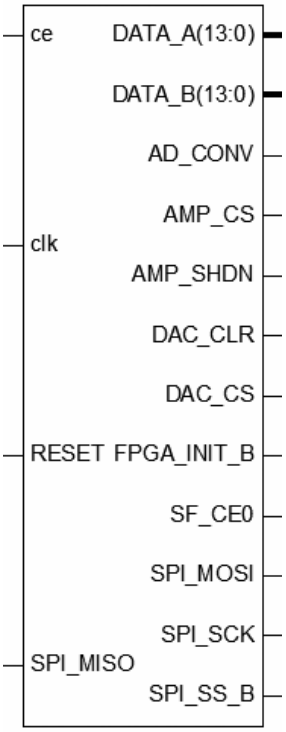


Abbildung 6.2: ADC_OVER_SPI-Modul

clk: Taktsigal
ce: Taktfreigabe
RESET: Rücksetzen des Moduls
SPI_MISO: FPGA SPI-Dateneingangsleitung
SPI_MOSI: FPGA SPI-Datenausgangsleitung
SPI_SCK: SPI-Taktsignal
DATA_A, DATA_B: Ergebnis der AD-Wandlung
AD_CONV:
AMP_SHDN: Rücksetzeingang für Vorverstärker
AMP_CS: Chip-Select Vorverstärker

FPGA_INIT_B:
SF_CE0: Leitungen zu nicht genutzten
DAC_CS: SPI-Bausteinen
DAC_CLR:
SPI_SS_B:

Funktionsweise:

Um den AD-Wandler nutzten zu können, muss zuerst der Verstärkerbaustein einmalig programmiert werden. Da der AD-Wandler für eine Eingangsspannung von 0,4V-2,9V ausgelegt ist und der verwendete Infrarot-Distanzsensor einen Spannungswert in diesem Bereich liefert, wird ein Verstärkungsfaktor von 1 eingestellt. Nach der Initialisierung des Vorverstärkers liegt Spannung am Eingang des AD-Wandlers an.
Die AD-Wandlung wird durch einen Puls an der AD_CONV-Leitung ausgelöst. Im Folgenden werden die beiden Kanäle des Wandlers ausgelesen, wobei die Daten immer Zeitversetzt um einen Messzyklus ausgegeben werden(Abbildung 6.3).

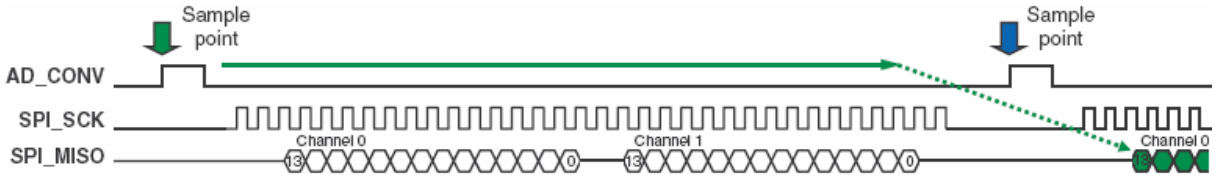


Abbildung 6.3: SPI-Kommunikation des AD-Wandlers [7]

Implementierung:

Der VHDL-Code des Moduls ist im Anhang D3 zu finden. Es besteht aus den vier Dateien `ADC_OVER_SPI.vhd`, `ADC_CONV.vhd`, `ADC_AMP.vhd` und `WAIT_CLK.vhd`.

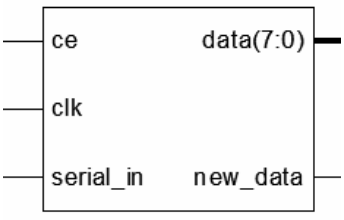
Die Schnittstelle des Moduls nach außen wird in VHDL als Toplevel-Entity bezeichnet. In diesem Modul heißt diese ADC_OVER_SPI und ist in gleichnamiger Datei zu finden.
In Ihr werden zuerst die Chip Select-Signale der nicht verwendeten SPI-Bausteine fest auf ‚1' gesetzt. Damit nehmen sie nicht an der Buskommunikation teil.
Die Hauptfunktion ist in einem Zustandsautomaten implementiert, der den Ablauf der SPI-Kommunikation koordiniert. Nach einer Wartezeit von 128 Taktzyklen bei 50 MHz wird der Vorverstärker mit dem Faktor 1 initialisiert. Nach dessen Abarbeitung wird die AD-Wandlung angestoßen und kontinuierlich wiederholt.
Da die Leitungen SPI_SCK und SPI_MISO vom Vorverstärker und AD-Wandler gemeinsam genutzt werden, werden sie im Hauptmodul durch einen Multiplexer umgeschaltet, und dem momentan aktiven Modul zugeordnet.

6.1.3 Serielle Schnittstellen

Auch bei der FPGA Implementierung des Reglers wurde die serielle Schnittstelle zur Übertragung der Führungsgröße vom PC ausgewählt.
Für den FPGA Regler sind nur 10 Stufen für die Hubhöhe des Ankers vorgesehen. Andere Konfigurationsparameter für den Regler gibt es nicht.
Deshalb kann das Übertragungsprotokoll auf eine Ein-Byteübertragung reduziert werden. Die Übertragung erfolgt als Zahl im ASCII-Format. Die Zahl gibt dabei die Hubhöhe in Zentimetern vor.

Anders als bei den vorhergehenden Modulen wurde dieses nicht komplett selbst programmiert, sondern großteils vom PicoBlaze Softcore-Prozessor entliehen. Dieser enthält bereits ein entsprechendes Modul, das lediglich um eine Paritätsprüfung und einen Ein-Byte-Buffer ergänzt wurde.

Schnittstelle:



clk:	Takt
ce:	Taktfreigabe
serial_in:	FPGA-Pin des seriellen Eingangs
data:	Byte-Ausgang
new_data:	Byte-Empfangen

Abbildung 6.4: UART-Modul

Funktionweise:

Für die serielle Kommunikation müssen Einstellungen gewählt:

Baudrate:	38400
Datenbits:	8
Parität:	gerade
Stopbits:	1
Flußsteuerung:	keine

Wird ein Byte über serial_in Eingang empfangen und ist dessen Parität korrekt, so wird es an den data-Ausgang weitergegeben. Das eintreffen von Daten wird mit einem Impuls am new_data-Ausgang bekannt gegeben.

Implementierung:

Die VHDL-Dateien für das UART-Modul sind im Anhang D3 zu finden. Dies sind `uart.vhd` und `clkDiv.vhd`.
Das verwendete VHDL-Modul `kcuart.vhd` des PicoBlaze-Softcores ist in [8] näher beschrieben. Es benötigt lediglich eine Taktquelle, die Impulse die dem 16-fachen der gewünschten Baudrate entsprechen, erzeugt. Diese werden vom clkDiv-Modul geliefert.
Wurden Daten vom kcuart-Modul empfangen, wird dies am data_strob-Ausgang des Moduls angezeigt. Die Daten werden dann an den data-Ausgang des eigenen Moduls weitergegeben, wenn die Paritätsprüfung erfolgreich war.
Die Paritätsprüfung erfolgt mittels XOR auf die empfangenen Datenbits und Vergleich mit dem höchstwertigen Bit am data_out-Ausgang des kcuart-Moduls.

6.2 HDL-Coder kompatibles Reglermodell

Die erste Implementierung des Reglermodells auf dem Xilinx FPGA erfolgt unter Verwendung des HDL-Coders. Dieser erweitert Matlab/Simulink um die Fähigkeit aus Modellen synthetisierbaren VHDL oder Verilog-Code zu generieren.
Die Verwendung des HDL-Coders ist allerdings mit Einschränkungen der Blockvielfalt von Simulink verbunden, weshalb eine Modellanpassung erfolgen muss.

Anders als beim System Generator, kann mit dem HDL-Coder kein externer VHDL-Code eingebunden werden. Ist dies gewünscht, muss dies nach der Codegenerierung manuell erfolgen.

6.2.1 Anpassung des Reglermodells

Modelle, aus denen mit dem HDL-Coder Code generiert werden soll, müssen einigen Konventionen entsprechen. So muss es sich um ein diskretes Modell handeln. Alle im Modell verwendeten Elemente müssen mit Fixed-Point Datentypen arbeiten. Nicht alle Blöcke aus der Simulink Standardbibliothek werden unterstützt, deshalb müssen nicht unterstützte Blöcke nachgebaut werden. Ein- und Ausgänge des Modells müssen dem Umfeld, in das sie eingebettet werden, entsprechen.

Die erste Forderung, dass es sich beim Modell des Reglers um ein diskretes Modell handeln muss, ist bereits durch den Regler aus Abschnitt 4.3 erfüllt.

Um ein Modell in Fixed-Point Datendarstellung umzuwandeln, muss die Fixed-Point Toolbox installiert sein. Diese erweitert Matlab/Simulink um die Typen sfix() für Festkommazahlen in Zweierkomplementdarstelltung und ufix() für vorzeichenlose Festkommazahlen.
Die Bitlänge und Position des Dezimalpunkts ist bei diesem Datentypen frei wählbar.

Für das Reglermodell wurde eine 32-Bit Zahlendarstellung im Zweierkomplement mit Komma nach dem sechzehnten Bit, ausgehend vom hochwertigsten Bit, gewählt (Abbildung 6.5).

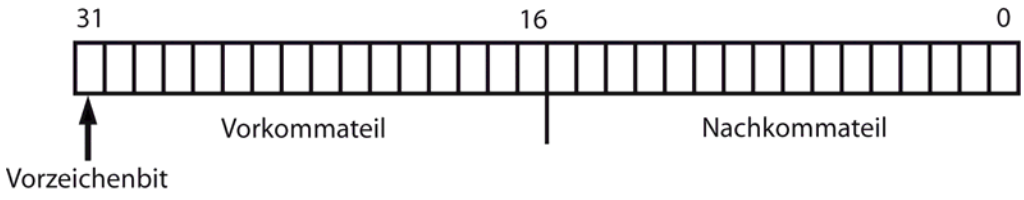


Abbildung 6.5: Verwendete Zahlendarstellung

Dadurch ergibt sich eine Wertebereich -32768 bis +32768 bei einer Auflösung von $2^{-16} \approx 15,259 \cdot 10^{-6}$.

Die Festlegung der Datentypen innerhalb des Modells erfolgt explizit im Konfigurationsdialog (Abbildung 6.6) der einzelnen Blöcke oder durch implizite Regeln.

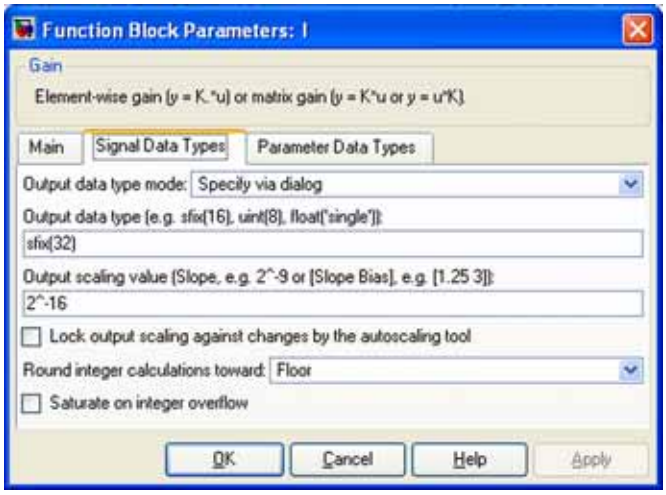


Abbildung 6.6: Explizite Festlegung der Zahlendarstellung

Innerhalb des Reglermodells werden beide Varianten verwendet. Für den ersten Block hinter einem Eingangs-Block und für Konstanten wurde die Zahlendarstellung explizit festgelegt. Alle folgenden Blöcke erhielten als Regel, die Zahlendarstellung des Blockeingangs zu übernehmen. Das Modell arbeitet somit durchgehend mit der gleichen Zahlendarstellung.

Die Eingänge selbst wurden so konfiguriert, dass sie mit den Schnittstellen der VHDL-Module aus Abschnitt 6.1 übereinstimmen. Dies erfolgt durch die Parameter der Ein- und Ausgangs-Blöcke des Modells (Abbildung 6.7).
Aus sfix(14) am Istwert-Eingang wird beispielsweise `std_logic_vector(13 downto 0)` im generierten VHDL-Code.

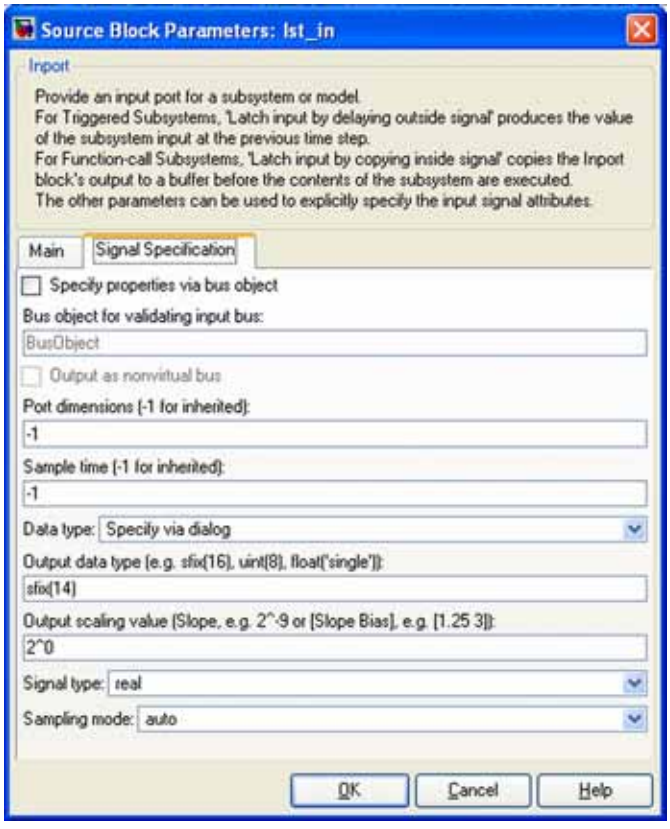


Abbildung 6.7: Konfiguration des Input-Blocks

Um das Modell HDL-Coder-konform zu gestalten, müssen alle nicht unterstützten Blöcke ersetzt werden. Zu diesen Blöcken gehören alle, die eine zeitabhängige Komponente beinhalten.
Beim Reglermodell sind dies der Integrator- und der Derivative-Block aus der Kategorie *Discrete* der Simulink Bibliothek. Beide Blöcke müssen also nachgebaut werden.

Die Nachbildung des Integrator-Blocks (Abbildung 6.8) besteht aus einem Addierer mit einem, um eine Abtastperiode verzögerten, rückgekoppelten Zweig. Dadurch werden alle Eingangswerte aufsummiert. Durch Multiplikation mit der Abtastzeit erhält man näherungsweise das Integral des Eingangswerts über die Zeit.

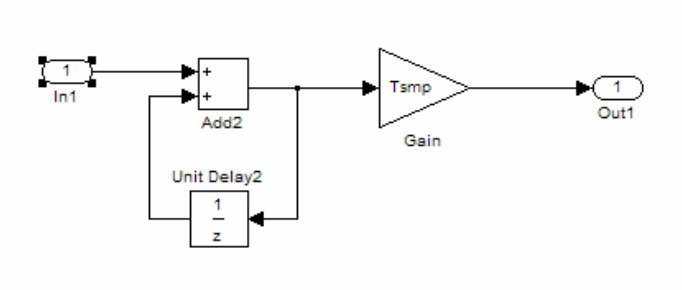


Abbildung 6.8: Nachbildung Integrator-Block

Bei der Nachbildung des Derivative-Blocks (Abbildung 6.9) wird vom aktuellen Eingangssignal, das um eine Abtastperiode verzögerte, subtrahiert. Durch Multiplikation mit dem Kehrwert der Abtastperiode erhält man die durchschnittliche Steigung zwischen dem letzten und dem aktuellen Eingangswert.

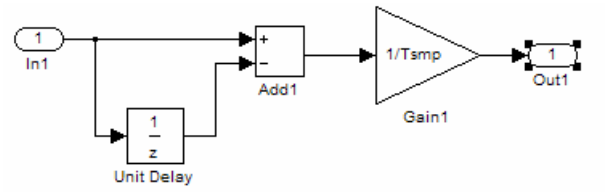


Abbildung 6.9: Nachbildung Derivative-Block

Obwohl der Saturation-Block keine speichernden Elemente enthält, wird er trotzdem nicht vom HDL_Coder unterstützt. Der Nachbau (Abbildung 6.10) erfolgt mit 2 Switch-Blöcken, die in Abhängigkeit des Eingangswerts schalten. Liegt dieser außerhalb des Wertebereichs, wird die entsprechende Konstante durchgeschaltet.

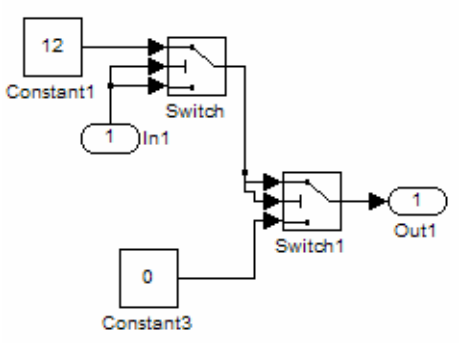


Abbildung 6.10: Nachbildung Saturation-Block

Auch der Division-Block von Simulink wird nicht unterstützt. Dieser wird im Reglermodell bei der Umrechnung des Spannungswerts des AD-Wandler in die Objektdistanz benötigt.

Eine Möglichkeit zur Lösung des Problems wäre es, den Dividierer mit Grundelementen nachzubauen. Die bei dieser Implementierung gewählte Methode ist jedoch, die Umrechnung über eine Lookup-Table durchzuführen. Für die Spannungen von 0,1V bis 3,3V liegen in 0,1V-Schritten die jeweiligen Distanz am Ausgang vor. Bei der Codegenerierung hat sich gezeigt, dass die Lookup-Table im VHDL-Code keine Zwischenwerte interpoliert, sondern bei der Umrechnung auf den nächst niederen Wert rundet. Da die Messwerte im Bereich von 0,8V bis 2,6V liegen, wären nur 19 unterschiedliche Ausgangswerte möglich. Dies sind zu wenige um eine präzise Regelung durchzuführen.

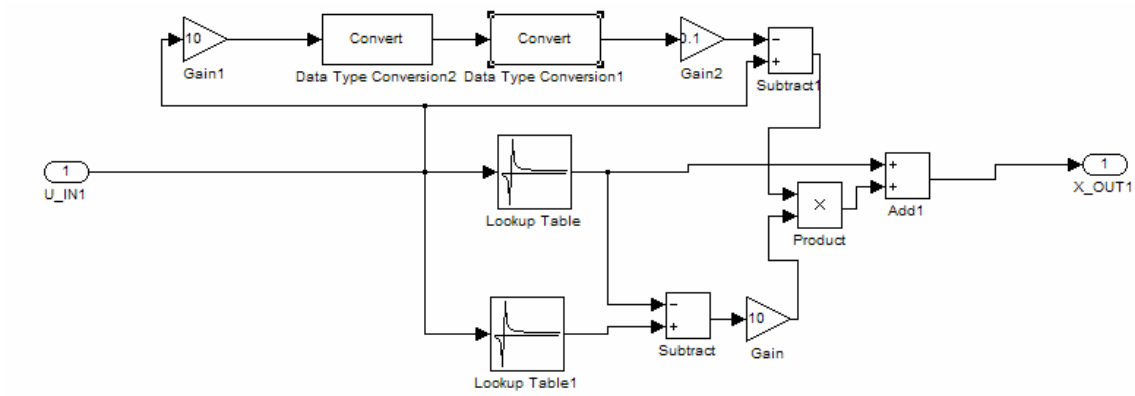


Abbildung 6.11: Lookup-Table mit Interpolation

Deshalb wurde ein Subsystem im Modell erstellt, das die vorhandene Lookup-Table benutzt, um aus benachbarten Eingangswerten, genauere Ausgangswerte zu interpolieren (Abbildung 6.11).
Dazu werden zwei Instanzen der Lookup-Table verwendet. Bei der ersten wird der nächst niedrige Eingangswert zur Ermittlung des Ausgangswerts benutzt, bei der Zweiten der nächst höhere. Die beiden unterschiedlichen Ergebnisse werden voneinander subtrahiert. Mit dem Kehrwert des Abstands der Tabelleneinträge multipliziert, erhält man die Steigung für den Abschnitt zwischen den zwei Tabelleneinträgen.
Die so ermittelte Steigung wird nun mit dem Abstand des Eingangswerts zum nächst niedrigen Lookup Table-Eintrag multipliziert und zum Eintrag der ersten Lookup Table addiert. Mit diesem Verfahren wird die Lookup Table stückweise linearisiert.

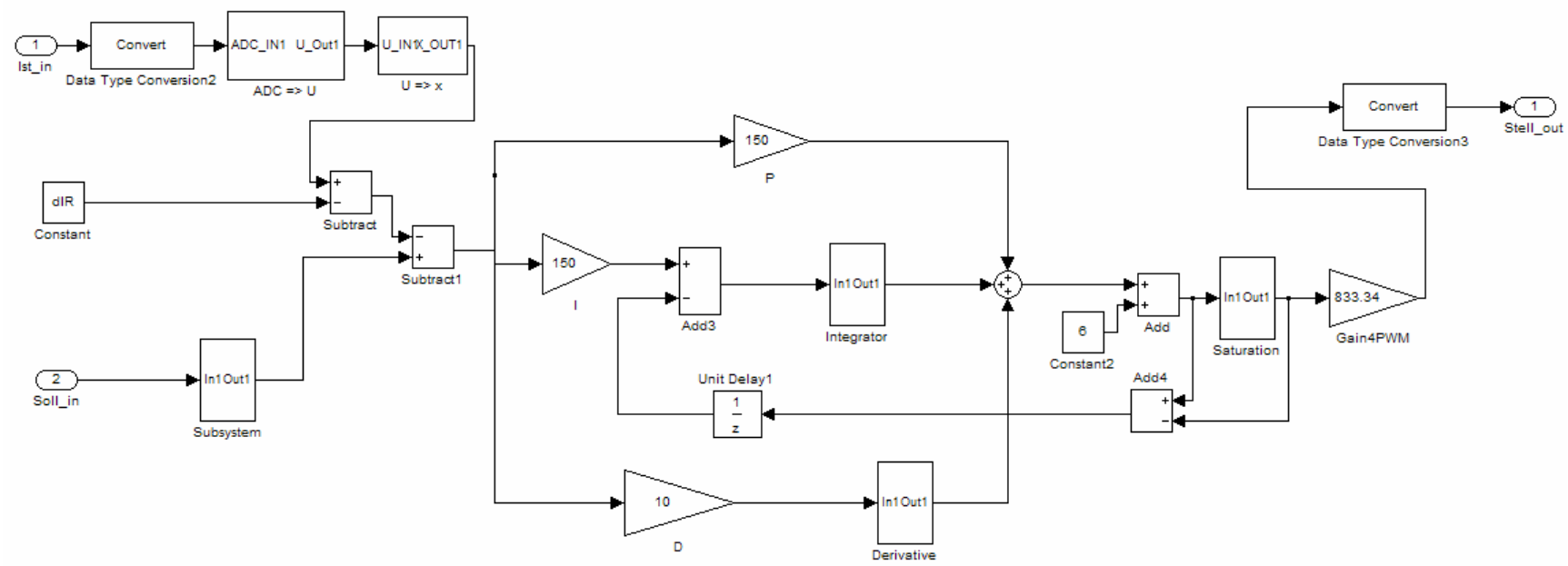


Abbildung 6.12: HDL-Coder-konformes Reglermodell

6.2.2 Codegenerierung

Die Codegenerierung kann, wie schon beim Mikrocontroller, sowohl für ein gesamtes Modell, als auch für ein Subsytem durchgeführt werden.
Die dazu nötigen Einstellungen erfolgen im Dialogfeld der Modellkonfiguration (*Menü/Simulation/Configuration Parameter…*).
Ist der HDL-Coder installiert, ist ein entsprechender Auswahlpunkt „HDL-Coder“ vorhanden (Abbildung 6.13).
Dort kann das zu generierende Modell oder Subsystem ausgewählt werden. Außerdem kann festgelegt werden ob der Code in VHDL oder Verilog generiert werden soll und in welchem Zielverzeichnis in dem die Dateien erstellt werden sollen.
Mit der Schaltfläche „Run Compatibilitiy Checker“, kann überprüft werden ob das Modell HDL-Coder-konform ist.
Die Codegenerierung wird durch Betätigung der „Generate“-Schaltfläche eingeleitet.

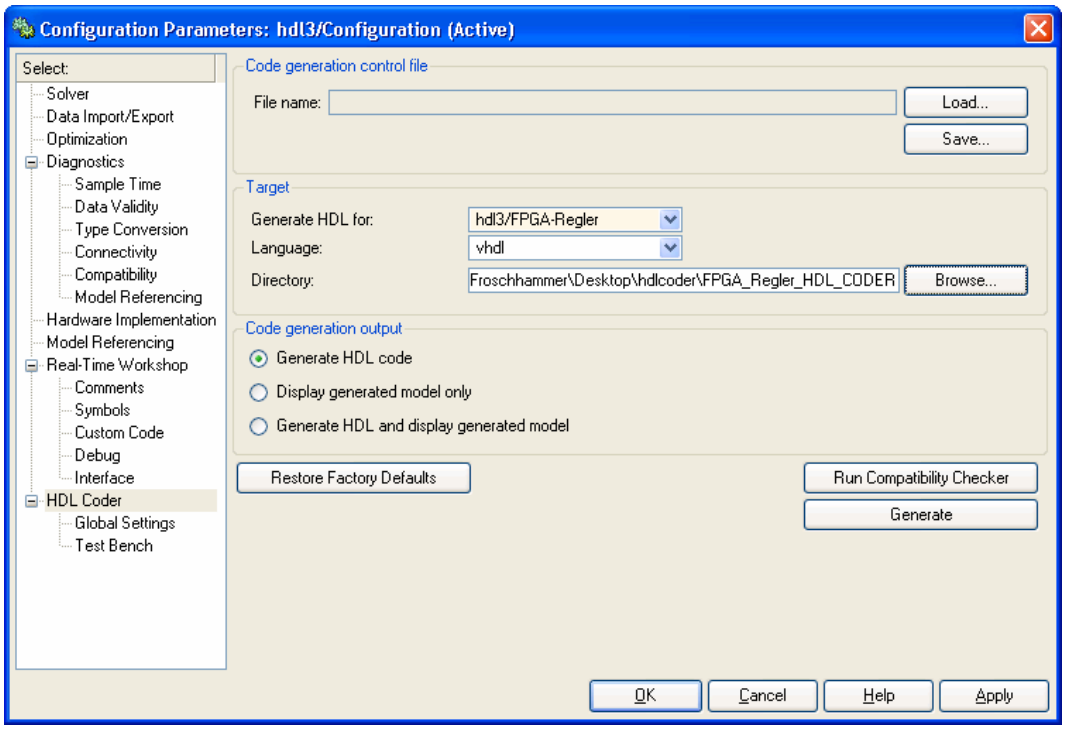


Abbildung 6.13: HDL-Coder Konfigurationsdialog

Generierungsergebnis:

Das Ergebnis der Codegenerierung sind mehrere VHDL-Dateien. Aus jedem Subsystem des Modells wir eine Datei generiert. Alle erzeugten Dateien werden als Komponenten in das Modul, das den Namen des Modells trägt, eingefügt.
In den VHDL-Dateien werden nach der Schnittstellenbeschreibung (entity) alle Signale definiert. Sie entsprechen den Signalleitungen im Modell oder Subsystem und tragen den Namen der Modellblöcke, aus denen sie hervorgehen.

Konvertierungen und Umwandlungsfunktionen stammen aus den VHDL-Standard-Bibliotheken `IEEE.std_logic_1164` und `IEEE.numeric_std`.
Der Großteil des Modells ist in kombinatorischer Logik umgesetzt. Die Anzahl der erreichten Schaltungsstufen bei einem komplexen Modell, wie dem des Reglers, ist daher erheblich.
Speicher- oder Verzögerungsglieder werden in sequentiellen VHDL-Code umgewandelt.
Das Beispiel zeigt das Verzögerungsglied aus dem nachgebildeten Derivative-Block:

```
Unit_Delay1_process : PROCESS (clk, reset)
 BEGIN
   IF reset = '1' THEN
     Unit_Delay1_out1 <= (OTHERS => '0');
   ELSIF clk'event AND clk = '1' THEN
     IF enb = '1' THEN
       Unit_Delay1_out1 <= Add4_out1;
     END IF;
   END IF;
 END PROCESS Unit_Delay1_process;
```

Lookup Tables werden ebenfalls in sequentielle Logik umgewandelt.Der Prozess in dem sie sich befinden wird durch Änderung des Eingangswerts der Lookup Table angestoßen.
Insgesamt ist der erzeugte VHDL-Code trotz seiner Komplexität gut les- und nachvollziehbar.

6.2.3 Einbettung in ein VHDL-Projekt

Anders als beim Systemgenerator, unterstützt der HDL-Coder keine Einbettung von VHDL-Modulen ins Modell. Deshalb müssen diese manuell hinzugefügt werden.

Zudem muss dem generierten Modell eine externe Taktquelle zur Verfügung gestellt werden. Da es sich bei diesem Modell um ein Single-Rate-Modell handelt, ist nur ein Taktsignal in der Abtastrate des Systems notwendig.

Zur Integration aller VHDL-Module zu einem Projekt wurde die Software ISE von Xilinx verwendet. Einen Überblick über die VHDL-Module und deren Verknüpfung gibt Abbildung 6.14.

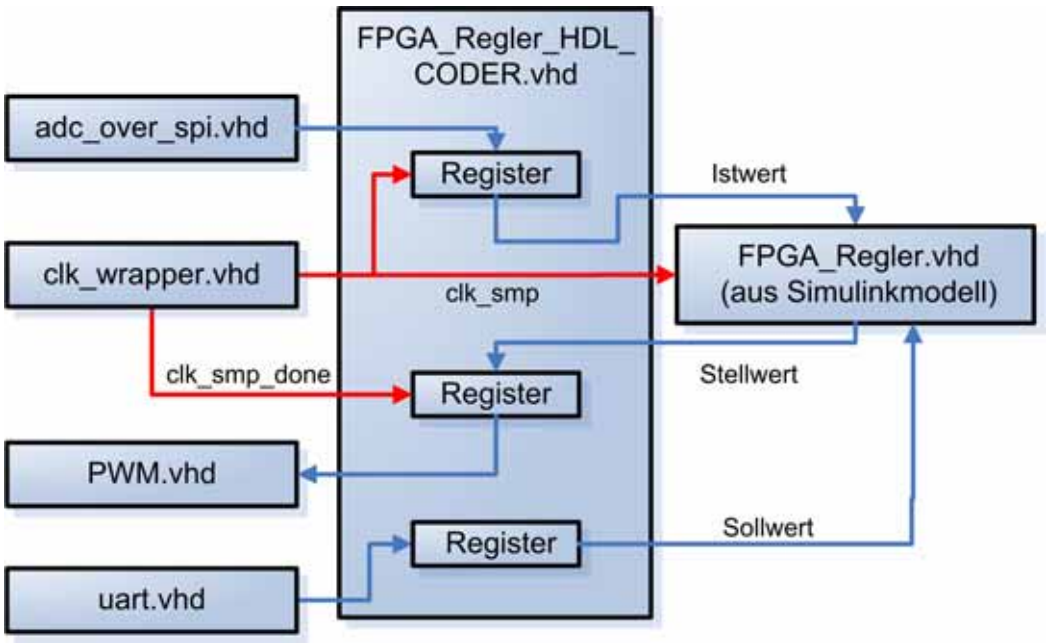


Abbildung 6.14: Einbettung des Regler-Moduls

Hauptmodul und Schnittstelle nach außen ist die Datei `FPGA_Regler_HDL_CODE.vhd` (Anhang D4). In ihm sind alle weiteren Module eingebunden. Die Ablaufsteuerung erfolgt über das Modul `clk_wrapper.vhd` (Anhang D4). Es stellt zwei Signale bereit. Zum einen das Taktsignal für den von Simulink generierten Regler, zum anderen ein Signal, das zeitverzögert zum Ersten dafür sorgt, dass der ermittelte Stellwert zum richtigen Zeitpunkt übernommen und an das PWM-Modul weitergegeben wird.

6.2.4 Schaltungssynthese

Die Schaltungssynthese wurde aus dem Xilinx ISE mit dem XST Synthesewerkzeug durchgeführt. Der Syntheseprozess des generierten Codes verläuft fehlerfrei. Eine Codeanpassung des Codes ist daher nicht notwendig. Lediglich einige Warnungen, die auf der Entkoppelung des Modells vom Taktsignal beruhen, werden gemeldet.
Auf das FPGA übertragen, erwies sich der Regler aber als nicht funktionsfähig. Die Pulsweite am PWM-Ausgang beträgt durchgängig 100%.
Bei der Untersuchung mit ChipScope, einem Programm zur schaltungsinternen Analyse, stellte sich heraus, dass das Eingangssignal des Reglers außerhalb des Wertebereichs der Lookup Tables lag. Der Regler nimmt also fälschlicherweise an, dass sich der Anker weit unterhalb des vorgegebenen Wertes befindet.
Dies weist auf einen Konvertierungsfehler oder die Missinterpretation eines Zahlenwerts hin. Leider konnte bis zur Abgabe der Diplomarbeit nicht ermittelt werden, ob der Fehler im Modell liegt oder bei der Codegenerierung aufgetreten ist.

6.3 Reglermodell mit Xilinx System Generator Blockset

Die zweite Umsetzung des Reglers für das Spartan-3E FPGA unterscheidet sich von den bisherigen Methoden dadurch, dass nicht das Simulink Standard-Blockset zur Modellbildung verwendet wurde, sondern das Xilinx System Generator Blockset.
Dieses ist laut Herstellerangaben zum Implementieren von Anwendungen der Digitalen Signalverarbeitung auf FPGAs von Xilinx konzipiert. Die Tatsache, dass das Blockset nicht nur DSP spezifische, sondern auch elementare Blöcke zur Verfügung stellt, lässt aber auch eine alternative Verwendung zu.

Nach Installation des Xilinx System Generators stehen im Simulink Library Browser neue Elemente zur Verfügung. Diese entsprechen in der Handhabung denen der normalen Simulink-Bausteine.
Einige wesentliche Unterschiede zum Standard-Blockset gibt es dennoch. Die System Generator-Komponenten sind hardwarebezogen und oft direkte Nachbildung digitaler Schaltelemente.
Anstelle von konventionellen Datentypen wie int, uint, double usw. werden beim System Generator-Blockset nur Festkommazahlen benutzt.
Es handelt sich aber dabei nicht um die Datentypen ufix und sfix, wie sie beim HDL-Coder-Modell verwendet werden, sondern um ein eigenes Format.

6.3.1 Nachbau des Reglers

Das mit dem System Generator erstellte Reglermodell ist das komplexeste der gezeigten. Es baut wieder auf dem im Abschnitt 4.3 vorgestellten diskreten PID-Regler auf. In diesem Fall wird aber das Modell nicht nur angepasst, sondern mit den Elementen des Xilinx Blocksets komplett neu aufgebaut.

Neben der, in der Bibliothek verwendeten Kategorisierung der Xilinx-Blocks, wie sie in ähnlicher Weise auch im Simulink-Standard-Blockset vorliegt und bereits in Abschnitt 4.1 erläutert wurde, ist es für den Nachbau des Reglers von Vorteil, das Xilinx-Blockset wie folgt einzuteilen:

> Systemblocks: Sie dienen zur Verwaltung des Modells an sich und werden benötigt, um HDL-Code zu generieren, Modelle zu analysieren und Schnittsellen zu andern, nicht auf dem System Generator beruhenden, Modellteilen zu realisieren.
>
> Elemente die in ähnlicher Form auch im Simulink Standard-Blockset zur Verfügung stehen: Dabei handelt es sich zumeist um elementare Komponenten aus dem Bereich mathematischer, logischer und relationaler Operatoren. Viele elementare Bestandteile sind nicht im System Generator enthalten. Meist ist dies der Fall, wenn speichernde Komponenten enthalten sind, die von der jeweiligen Bezugszeit des Modells abhängen. Diese müssen dann mit elementaren Blöcken und Registern nachgebildet werden.

Blöcke die auf spezielle Hardware des FPGAs zurückgreifen wie etwa Dual Port RAM, Shift-Register usw.

Darüber hinaus gibt es noch Blöcke, die speziell für die Digitale Signalverarbeitung bestimmt sind.

Nicht für alle Elemente der Simulink Standard-Bibliothek gibt es auch ein Äquivalent im System Generator. Speziell auf das Reglermodell bezogen sind die Blöcke Integrator und Derivative, sowie der Saturation-Block im System Generator nicht verfügbar. Der Nachbau dieser Komponenten wird später im Detail erläutert.

Für die Ansteuerung der Peripherie des FPGA werden die VHDL-Module für den Analog-Digital-Wandler, die serielle Schnittstelle und Pulsweitenmodulation aus Abschnitt 6.1 verwendet. Die Einbettung erfolgt unter Verwendung von Black Box-Blöcken.

Eine weitere Besonderheit dieses Modells besteht darin, dass der zeitliche Ablauf des Regelalgorithmus selbst koordiniert werden muss.
Dies ist notwendig, da bei der Codegenerierung so genannte Timing Constrains für den Syntheseprozess erstellt werden. In diesen Vorgaben wird die maximale Durchlaufzeit eines Signals festgelegt, die nicht größer als eine Taktperiode sein darf. Das führt dazu, dass nicht beliebig viele Schaltglieder hintereinander folgen dürfen, da sonst die Schaltungssynthese mit Fehlern abbricht. Um dem entgegenzuwirken, ist es bei mehrstufiger Logik möglich, Register zwischenzuschalten, die ihr Eingangssignal um einen Takt verzögert weiterleiten.

Um Ein- und Ausgänge des Reglermodells einem physikalischen Pin am FPGA zuzuweisen und um das System Generator-Modell von einem normalen Simulink-Modell abzugrenzen, stehen Gateway In- und Gateway Out-Blöcke zur Verfügung (Abbildung 6.15). In Ihnen kann einem Signal im Modell ein Input\Output Block (IOB) des FPGAs zugeordnet werden.
Ist die Signalleitung, die zum Gateway Block führt, ein Bitvektor, so muss eine entsprechende Anzahl an IOBs zur Verfügung gestellt werden.

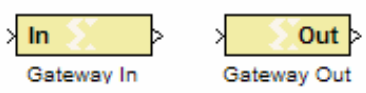


Abbildung 6.15: Gateway Blöcke

Das Modell des gesamten Reglers aus System Generator-Bausteinen ist Abbildung 6.15 zu sehen. Die einzelnen Elemente und die verwendeten Subsysteme werden im Folgenden, angefangen mit dem Reglereingang, detailliert beschrieben.

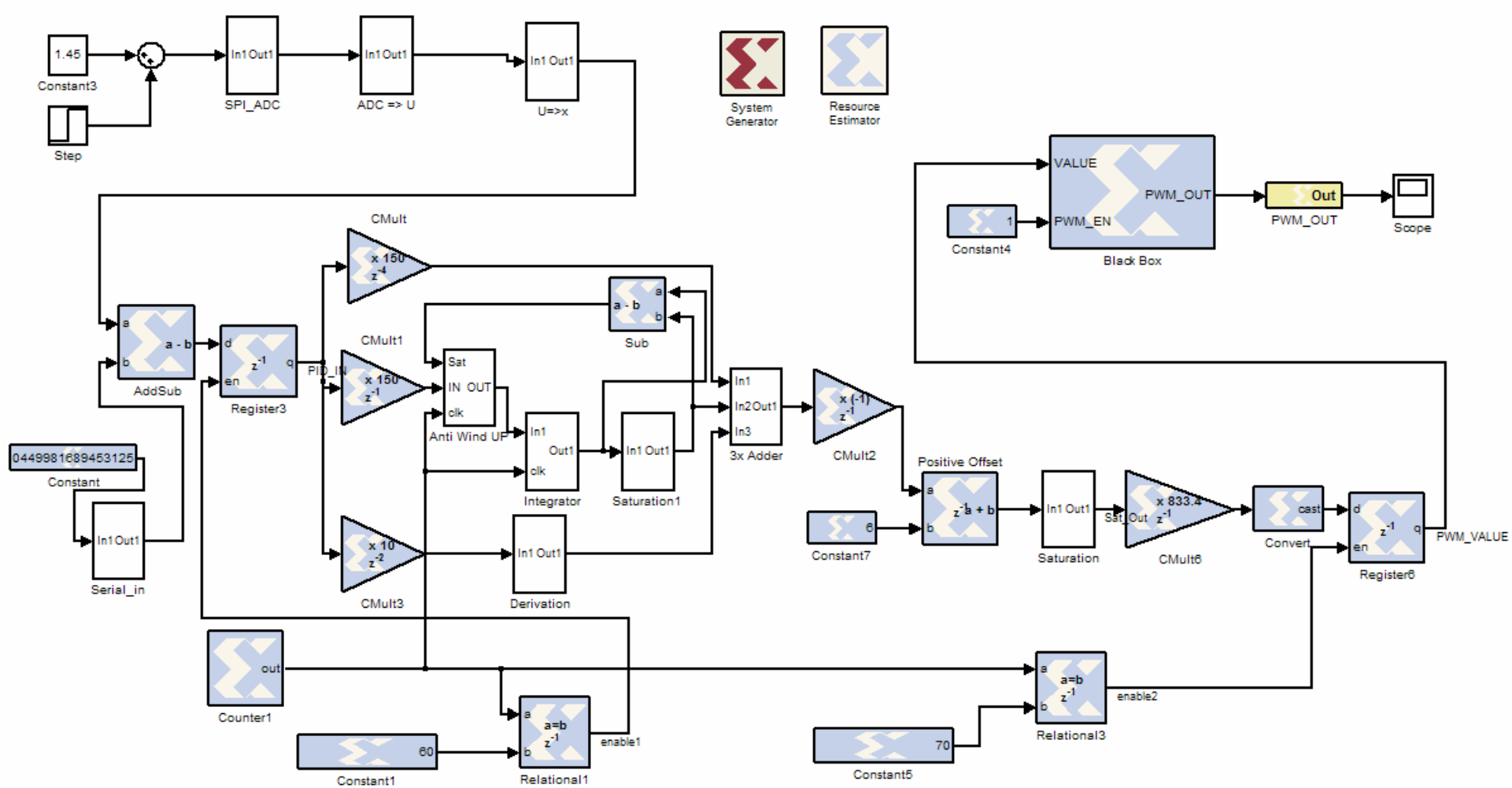


Abbildung 6.16: System Generator-Modell des Reglers

6.3.1.1 Reglereingang

Abbildung 6.17 zeigt den Eingangsabschnitt des FPGA-Reglers. Dieser ist im Gesamtmodell des Reglers (Abbildung 6.15) oben links zu finden. In der Simulation ist das Eingangssignal ein konstanter Wert, wie er vom Sensor stammen könnte. Durch Addition einer Sprungfunktion kann das Verhalten des Reglers bei Spannungsänderung betrachtet werden.

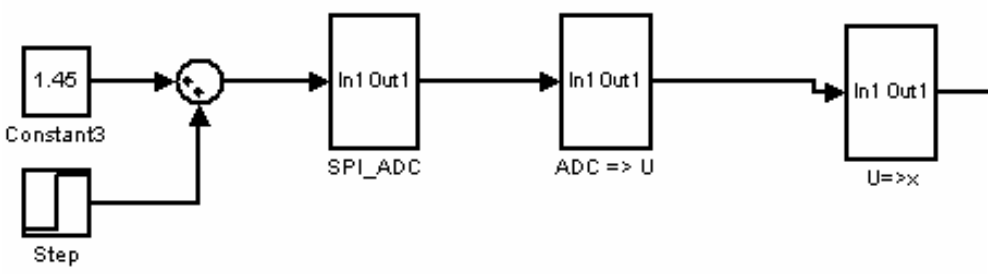


Abbildung 6.17: Reglereingang

Der Eingangsteil des Reglers besteht aus drei Subsystemen, die ihrerseits Teilfunktionen des Modells zusammenfassen. Das erste der drei Subsysteme beinhaltet das VHDL-Modul des AD-Wandlers. Im zweiten wird die Ausgabe des AD-Wandlers in einen Spannungswert umgerechnet und im dritten Subsystem wird aus der Spannung die Objektdistanz errechnet.

Die drei hintereinander geschalteten Subsyteme werden im Folgenden einzeln erläutert.

SPI_ADC-Subsystem: Schnittstelle zum Analog-Digital-Wandler

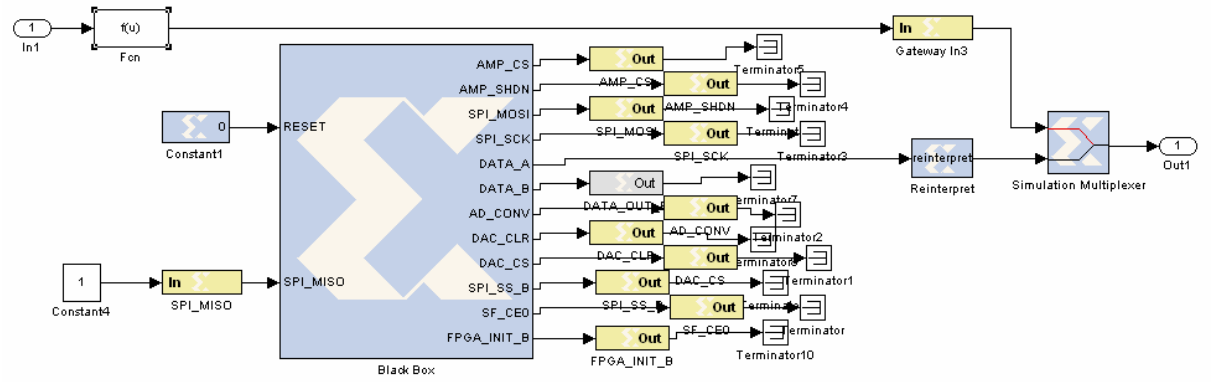


Abbildung 6.18: SPI_ADC Subsystem

Den Eingang des Reglers bildet das SPI_ADC Subsystem. Für Simulation und Codegenerierung existieren in diesem Subsystem zwei unterschiedliche Pfade. Durch den Simulationsmultiplexer wird jeweils der gültige Pfad gewählt.
Im Simulationspfad befindet sich ein fcn-Block, der das Verhalten des Analog-Digital-Wandlers während der Simulation nachahmt, indem er das Eingangssignal in 14-Bit Zweierkomplementdarstellung konvertiert.

Im zweiten Pfad ist das VHDL-Modul zur Ansteuerung des Analog-Digital-Wandlers über SPI eingebettet.
Alle Ein- und Ausgangspins, die zum Ansteuern des SPI-Busses notwendig sind, sind mit einem Gateway In- oder Gateway Out-Block verknüpft.
Innerhalb dieser Blöcke wird die Zuweisungen der Ein- und Ausgänge zu den entsprechenden physikalischen Anschlüssen des FPGAs vorgenommen.

Da die Black Box des AD-Wandlers nur Bitvektoren als Ausgang liefert, wird ein Reinterpret-Block zwischengeschaltet, so dass die Daten des Analog-Digital-Wandlers am Ausgang DATA_A als 14-Bit Zweierkomplement interpretiert werden.

Subsystem ADC=>U: Berechnung der Spannung aus dem Wert des AD-Wandlers

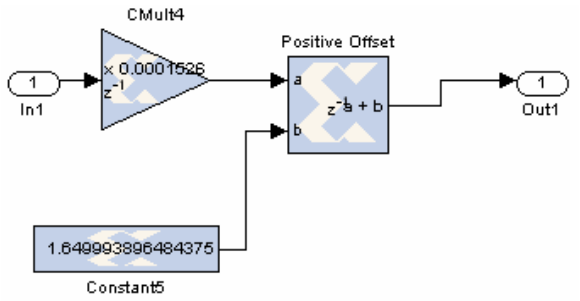


Abbildung 6.19: Subsystem ADC => U

Am Eingang dieses Subsystems liegt der 14-Bit Wert des ADWandlers in Zweierkomplementdarstellung. Bei einer Referenzspannung von 3,3V kann der AD-Wandler Werte von 1,65V $\pm$ 1,25V darstellen. 1,65 V am Eingang entspricht dabei dem Wert 0 am Ausgang des Wandlers.
Das Spartan-3E Benutzerhandbuch [7] liefert folgende Gleichung für diesen Zusammenhang:

$$D[13:0] = GAIN \cdot \frac{V_{IN} - 1{,}65V}{1{,}25V} \cdot 8192;$$

$D[13:0]$: Wert am Ausgang des AD-Wandlers
$GAIN$: Vorverstärkung
V_{IN}: Eingangsspannung [V]

Da für den Infrarot Distanzsensor nur der Zusammenhang zwischen Spannung und Entfernung bekannt ist wird die Gleichung nach V_{IN} umgestellt:

$$V_{IN} = \frac{D[13:0] \cdot 1{,}25V}{8192} + 1{,}65V;$$

Die Darstellung der Gleichung als Modell zeigt Abbildung 6.19. Um unnötige Berechungen während der Laufzeit zu vermeiden, sind alle Multiplikatoren und Divisoren mit konstanten Werten vorberechnet und in einem CMult-Block zusammengefasst.

Subsystem U=>x : Berechnung Objektdistanz aus Sensorspannung

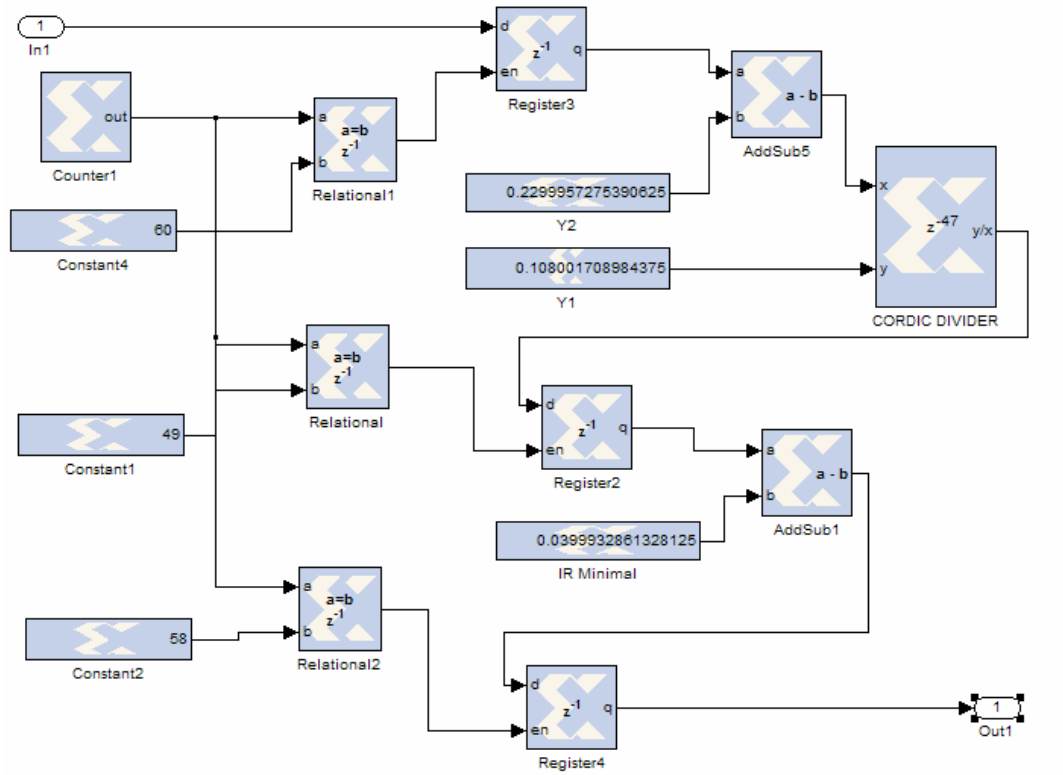


Abbildung 6.20: Subsystem U=>x

Aus dem Spannungswert, der vom letzten Subsystem geliefert wird, wird in diesem Subsystem die Objektdistanz errechnet. Dazu wird der aus Abschnitt 3.2.2 bekannte Zusammenhang zwischen Ausgangsspannung des Infrarot-Distanzsensors und Distanz zum Objekt in ein Modell umgesetzt.

$$x = \frac{Y_1}{U - Y_2}; \quad \text{für x >= 4 cm.}$$

x := Abstand [m]
U:= Spannung am Ausgang [V]
Y_1:= Steigungskonstante [Vm]
Y_2:= Offset [V]

Abbildung 6.20 zeigt die Gleichung als Modell im Subsystem U=>x.
Für die dazu nötige Division wird ein CORDIC Divider aus dem Xilinx Reference Blockset verwendet. CORDIC steht für **CO**ordinate **R**otation **DI**gital **C**omputer. Es handelt sich um iterative Algorithmen mit denen sich viele mathematisch Funktionen umsetzten lassen.
Der CORDIC Divider benötigt mehrere Taktzyklen um ein gültiges Ergebnis zu liefern. Zudem befinden sich zeitkritische Pfade in dem Subsystem.

Um bei der Codegenerierung und Schaltungssynthese keine Fehlermeldungen zu erhalten, müssen speichernde Elemente in Form von Registern zwischengeschaltet werden.

Die Register übernehmen die am Eingang liegenden Werte taktsynchron, wenn „1“ am Enable-Eingang anliegt. Diese Freigabe wird über einen Vergleichsoperator realisiert, der immer dann, wenn ein Zähler, der an einem Eingang des Relationalen Operators anliegt, mit dem konstanten Wert, der am andren Eingang anliegt, übereinstimmt. So lässt sich der zeitliche Ablauf gezielt steuern.
Am Ausgang des Subsystems werden von der ermittelten Distanz noch 0.04 m subtrahiert. Dies ist der Abstand des Ankers vom Sensor, wenn sich dieser am unteren Anschlag befindet.

Subsystem Serial_in:

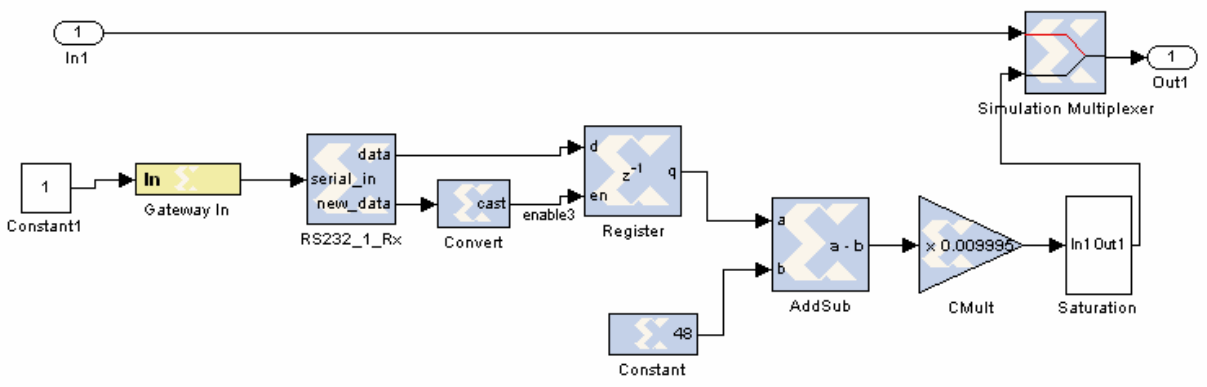


Abbildung 6.21: Subsystem Serial_in

Der Messwerteingang ist jedoch nicht der Einzige des Reglers.
Im Serial_in-Subsystem ist das VHDL-Modul für die serielle Schnittstelle, die den Sollwerteingang des Reglers bildet, eingebettet.
Das Black Box-Element RS232_1_Rx enthält dieses. Treffen Daten ein wird dies über den new_data Ausgang signalisiert. Die Daten werden in ein Register geschrieben. Da als serielle Eingabe die ASCII-Zeichen von 0 bis 9 festgelegt wurden erhält man durch Subtraktion von 48 den Wert in Zentimeter. Werte die nicht im Bereich von 0 bis 9 liegen, werden mit dem Saturation-Block zum nächst gültigen Wert gerundet.

6.3.1.2 PID-Regler

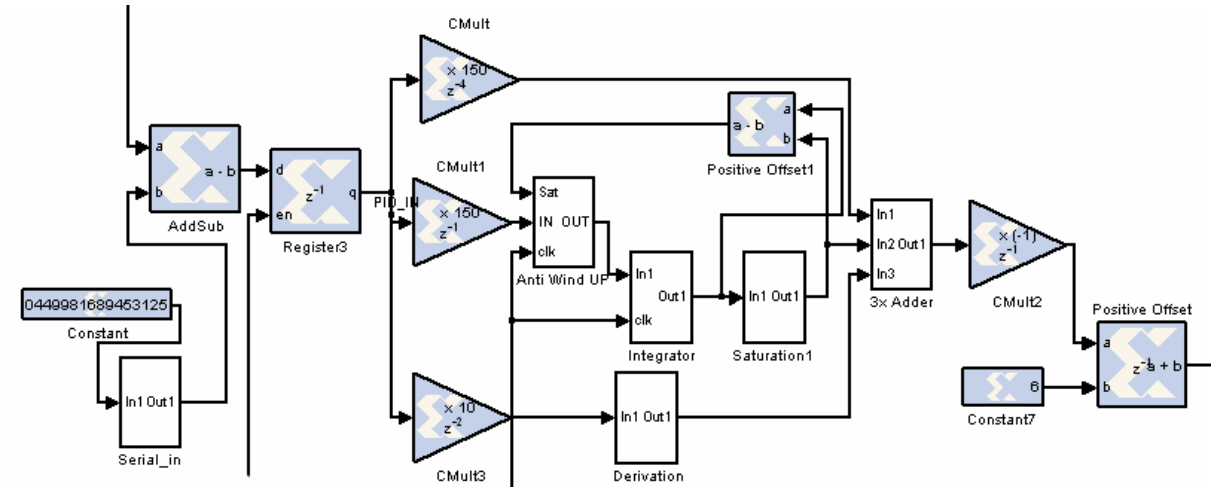


Abbildung 6.22: System Generator PID-Reglermodell

Wie schon die vorangegangenen Regelerimplementierungen, kommt auch hier wieder ein PID-Regelalgorithmus zum Einsatz. Die drei parallelen Pfade für Proportional, Integral- und Differenzialanteil befinden sich in der Mittle von Abbildung 6.22. Zuvor wird aber erst die Regeldifferenz zwischen dem Istwert der Hubhöhe und der Sollwertvorgabe aus dem Serial_in gebildet.

Für die Integration und das Differenzieren stehen keine Blöcke im System Generator zur Verfügung. Sie wurden aus diesem Grund in den Subsystemen Integrator (Abbildung 6.23) und Derivation (Abbildung 6.24) nachgebildet.
Für das Subsytem des Integrierers wurde das sehr einfache Verfahren der Vorwärtsintegration gewählt. Dieses basiert darauf, dass der aktuelle Wert der Regeldifferenz mit der Abtastzeit multipliziert und zum letzten Wert hinzuaddiert wird. So erhält man einen angenäherten Wert für das Integral der Regeldifferenz über die Zeit.

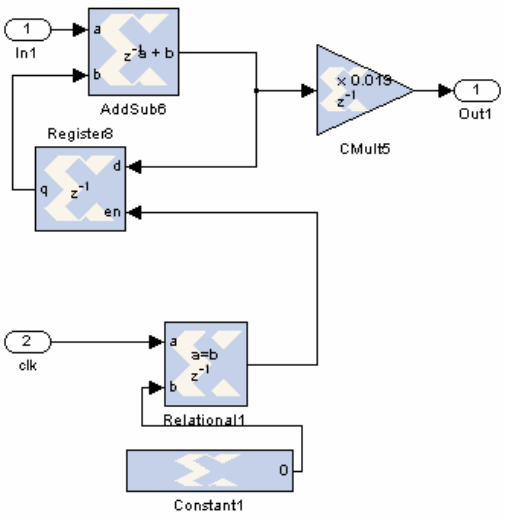


Abbildung 6.23: Integrator-Subsystem

Der Differenzialanteil verstärkt die Änderung der Regeldifferenz. Um die Höhe dieser Änderung zu ermitteln wird der Vorgängerwert in einem Register zwischengespeichert und mit Verzögerung an ein Subtraktionsglied weitergeleitet. Die Differenz aus aktuellem Eingabewert und dem der vorherigen Abtastung ergibt, multipliziert mit dem Kehrwert der Abtastperiode, die Steigung.

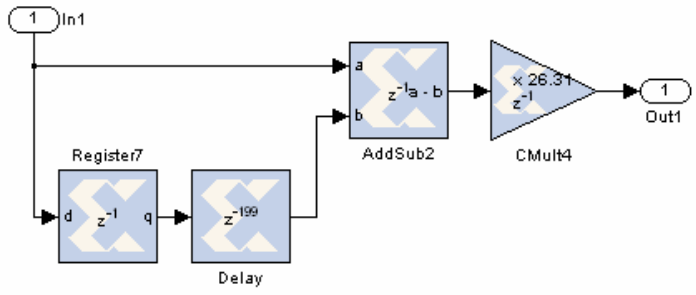


Abbildung 6.24: Derivation-Subsystem

Addierer mit Drei Eingängen:

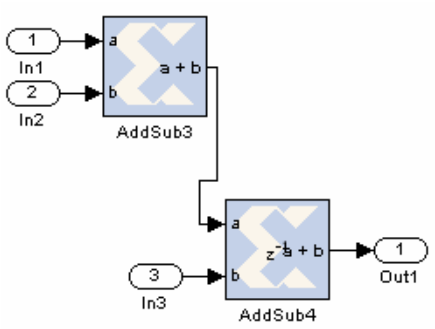


Abbildung 6.25: Subsytem Dreifach-Addierer

Die drei Zweige des PID-Reglers werden mit dem 3xAdder-Subsystem summiert. In ihm werden einfach zwei Addierer hintereinander geschaltet. Der letzte Addierer enthält ein Speicherglied, wodurch die unterschiedlichen Signallaufzeiten der Eingänge ausgeglichen werden. Das korrekte Ergebnis wird einen Takt verzögert ausgegeben.

Saturation -Subsystem:

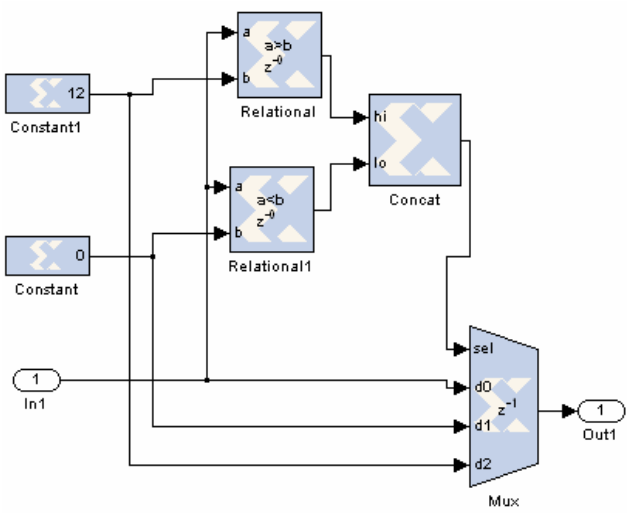


Abbildung 6.26: Subsystem Saturation

Das Saturation-Subsystem ist eine Nachbildung des gleichnamigen Blocks aus dem Simulink Standard-Blockset. Innerhalb des Reglermodells wird dieses mehrfach eingesetzt. Es sorgt dafür, dass der Eingangswert auf einen definierten Wertebereich beschränkt wird.

Die beiden Constant-Blocks bilden das Minimum und Maximum des Wertebereichs. Über relationale Operatoren werden diese mit dem Eingangswert verglichen. Deren Ausgänge werden im Concat-Block zusammengefügt und dienen als Steuersignal eines Multiplexers. Je nachdem, ob sich der Wert innerhalb der Grenzen befindet oder nicht, wird entweder die Eingangsleitung, das Minimum oder Maximum zum Ausgang durchgeschaltet.

Anti-Wind-Up - Algorithmus:

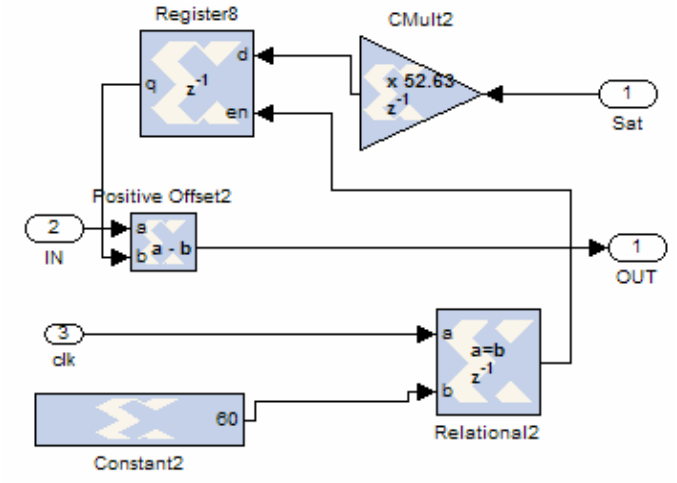


Abbildung 6.27: Anti-Wind-Up-Subsystem

Wie bei allen digitalen Reglern, die ein Integrationsglied enthalten, ist es auch hier notwendig, den maximalen Integrationsanteil zu beschränken. Eine Rückkopplung muss geschaffen werden.
Dazu werden die Werte vor und nach dem Saturation-Subsystem voneinander abgezogen. Die Differenz wird, mit einer Abtastperiode verzöget, vom Eingangswert des Integrator-Subsytem subtrahiert. Dies sorgt dafür, dass der gespeicherte Wert des Integrators innerhalb definierter Grenzen bleibt.

Synchronisation:

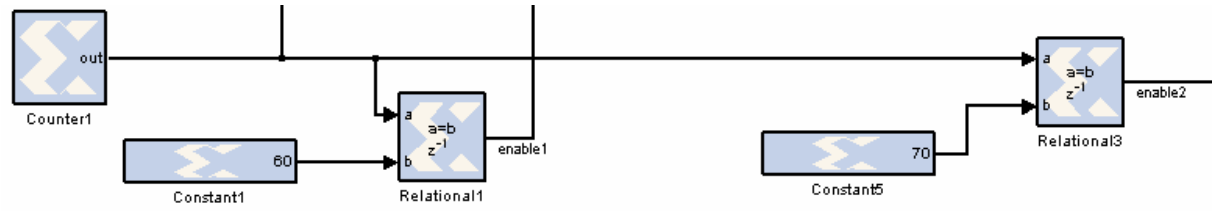


Abbildung 6.28: Synchronisation

Die Synchronisation der Reglerschaltung erfolgt ähnlich wie im U=>x-Subsystem, mit, durch einen Zähler freigeschaltete, Register. Der Zähler ist als Aufwärtszähler konfiguriert. Beim Erreichen des maximalen Zählerwerts springt dieser auf 0 zurück und beginnt erneut.
Der maximale Zählerwert ist maßgebend für die Abtastzeit des Reglers. Auch in diesem Fall dient die Abtastzeit des Sensors von 38 ms als Vorgabe für den Regler. Bei einem Taktzyklus von 20 ns entspricht dies 1900000 Takten.
Bei einem Zählerstand von 60 wird der aktuelle Wert der Regeldifferenz in das Eingangsregister des Reglers übernommen und das Stellsignal wird erneut berechnet. Beim Zählerstand 70 liegt das Ergebnis des Stellwerts vor und wird an den PWM-Eingang übergeben und dort gehalten, bis ein neuer Stellwert ermittelt wurde.

6.3.1.3 Regler-Ausgang

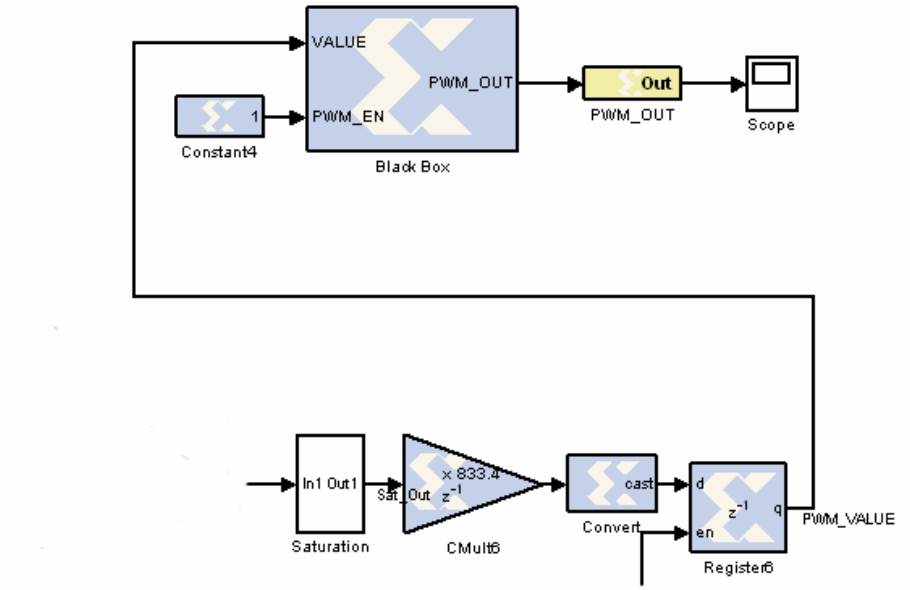


Abbildung 6.29: Reglerausgang

Das Erste Glied des Reglerausgangs bildet ein weiteres Saturation-Subsystem. Dieses begrenzt den durch den PID-Regler ermittelten Stellwert auf 0 bis 12 V. Durch Multiplikation mit 833.4 erhält man einen Wert von 0 bis 10000. Dieser wird in einen vorzeichenlosen 14-Bit Wert konvertiert. Dies entspricht den Vorgaben des VHDL-Moduls für die Pulsweitenmodulation, welches in dem Black Box-Element eingebettet ist. Die Gateway Out-Komponente weist dem Reglerausgang einen Kontakt am FPGA zu.

6.3.2 Codegenerierung

Um HDL-Code aus dem eben vorgestellten Reglermodell zu generieren, muss ein System Generator-Block (Abbildung 6.30) in dieses eingefügt werden. Durch Doppelklick auf diesen wird ein Dialogfenster (Abbildung 6.31) geöffnet, in dem unter anderem Einstellungen für die Codegenerierung vorgenommen werden können.

Abbildung 6.30: System-Generator Block

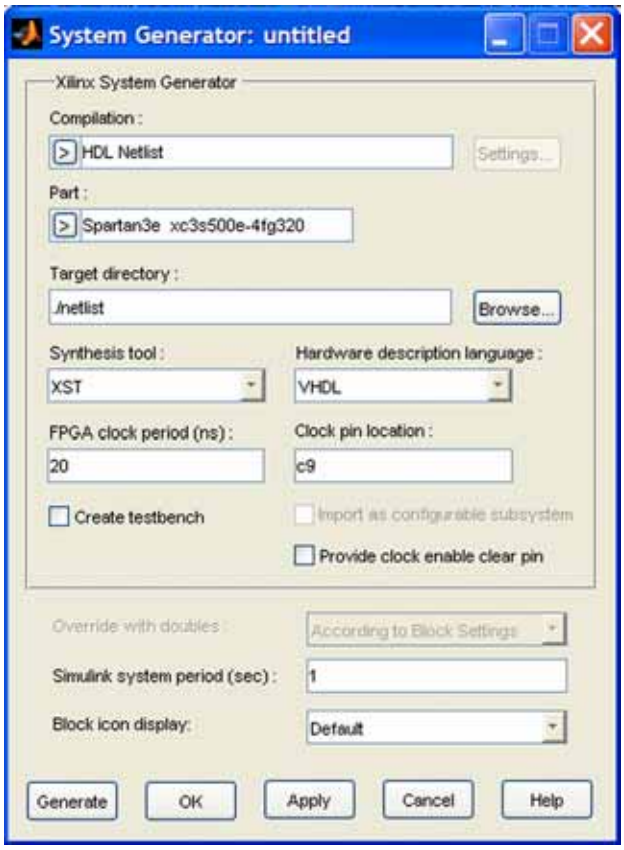


Abbildung 6.31: System Generator-Konfiguration

Es stehen verschiedene Optionen zur Codegenerierung zur Verfügung:

HDL- und NGC-Netlist: Durch die Codegenerierung wird ein ISE-Projekt erstellt, das das codierte Modell beinhaltet.

Timing Analysis: Codegenerierung und Schaltungssynthese wird durchgeführt. Anschließend wird das Zeitverhalten der erzeugten Schaltung untersucht.

Bitstream: Direkt aus Simulink heraus wird der vollständige Syntheseprozess durchlaufen. Ergebnis ist eine Bitsteam-Datei die nur noch auf den FPGA übertragen werden muss.

Weitere Einstellungen betreffen die Bauteilauswahl, das verwendete Synthesewerkzeug, sowie Einstellungen bezüglich des Taktsignals.
Mit der Schaltfläche „Generate“ wird die Codegenerierung bzw. Schaltungssynthese durchgeführt. Das Ergebnis wird im gewählten Verzeichnis hinterlegt.

Generierungsergebnis:

Wurde die Option „HDL-Netlist“ gewählt, wird eine Reihe von Dateien generiert, die in einem ISE-Projekt zusammengefasst sind:

`<modellname>.vhd`: Beinhaltet den in VHDL generierten Teil des Modells.

`<modellname>_cw.vhd`: Schnittstelle nach Außen.

`.edn` und `.ngc` Dateien: Neben der Generierung von VHDL-Code wird bei der Codegenerierung der CORE Generator verwendet um Teile des Modells zu implementieren. Diese Bausteine sind in den .edn und .ngc Dateien zu finden.

`<modellname>_cw.xcf`: Timing- und Area-Constrains für das Synthesewerkzeug

`<modellname>_cw.ise`: Die Xilinx ISE Projekt-Datei

`vcom.do` : Skriptdatei für Verhaltenssimulation in ModelSim

Lädt man das Projekt in Xilinx ISE und synthetisiert man die Schaltung, werden viele Warnungen erzeugt, die keine Auswirkung auf die Funktionsfähigkeit des generierten Codes haben. Das Problem hierbei ist jedoch, das Warnungen die wirklich zu unerwünschten Verhalten des FPGAs führen, in der Menge verloren gehen.

Weitere Probleme treten auf, wenn man versucht, die Zuordnung der FPGA-Pins erst im Xilinx ISE durchzuführen. Offensichtlich führt dies zu Konflikten zwischen der ISE Constrains-Datei und der vom System Generator erzeugten.

6.4 HDL-Coder und Xilinx System Generator im Vergleich

Sowohl HDL-Coder als auch Xilinx System Generator haben FPGAs als Zielsystem. Jedoch nur der HDL_Coder liefert generischen HDL-Code. Der System Generator ist an die FPGAs des Herstellers Xilinx gebunden.

Die Modellierungsbausteine, die der System Generator zur Verfügung stellt, sind näher an der Hardware. Dies lässt mehr Eingriffsmöglichkeiten bei der Modellbildung zu, hat aber auch zur Folge, dass bereits mehr Gedanken zur tatsächlichen Hardwareumsetzung ins Modell einfließen müssen.
Dafür liefert der System Generator bereits HDL-Code, der ohne Nachbearbeitung synthetisiert und auf den FPGA übertragen werden kann.
Zudem kann in System Generator-Modellen externer VHDL-Code eingebunden werden.

Der VHDL-Code, der vom HDL-Coder erzeugt wird, ist nicht direkt auf einen FPGA übertragbar. Er muss in eine Umgebung eingebettet werden, die ihm zumindest ein Taktsignal zur Verfügung stellt. Wenn zusätzlich noch VHDL-Module angesprochen werden sollen, müssen auch diese manuell eingebunden werden. Dafür kann aber das Standard-Blockset von Simulink verwendet werden und eine Umstellung ist nicht notwendig.

Hinsichtlich der Simulierbarkeit schneidet der HDL-Coder besser ab, da sich das Modell des System Generators nur schwer an ein Standard-Simulink-Modell, das beispielsweise ein physikalisches System beschreibt, anbinden lässt. Der Grund dafür ist die unterschiedliche Zeitbasis der beiden Systeme. Während ein physikalisches Modell beispielsweise im Sekundebereich abläuft, wird ein System Generator-Modell taktgenau simuliert.

Was die Resourcennutzung betrifft, so kann der System Generator die Besonderheiten der Xilinx FPGAs nutzen. Der HDL-Coder erzeugt sehr schlanken Code, kann aber nur auf Standardkonstrukte, die in VHDL definiert sind, zurückgreifen.

7 Zusammenfassung

In dieser Diplomarbeit wurde gezeigt, wie aus einem simulierbaren, graphischen Computermodell funktionsfähiger Code, sowohl für Mikrocontroller, als auch für FPGAs generiert werden kann.
Am Beispiel eines Digitalreglers wurde der gesamte Entwurfsprozess, von der Modellbildung bis zur Integration ins System, dargestellt. Begonnen wurde hierbei mit der Modellierung der Regelstrecke anhand physikalischer Gesetzmäßigkeiten. Das dabei entstandene Modell diente als Grundlage für den Reglerentwurf. Durch Modellanpassung und Verfeinerung wurde der Regler für die Zielplattformen, Mikrocontroller und FPGA, umgesetzt.

Die Codegenerierung für den verwendeten Mikrocontroller XC167 von Infineon erfolgte mit dem Embedded Real-Time Workshop von The Mathworks. Für die FPGA-Implementierung auf dem Spartan-3E von Xilinx wurde sowohl der HDL-Coder, ebenfalls von The Mathworks, als auch der Xilinx System Generator zur Erzeugung von HDL-Code verwendet.

Es sei an dieser Stelle erwähnt, dass in der Diplomarbeit bei weitem nicht alle Möglichkeiten von Matlab/Simulink ausschöpft wurden. Speziell das mächtige Stateflow-Werkzeug, mit dem komplexe Zustands- und Ablaufdiagramme erstellt werden können, blieb außen vor.
Mit Stateflow-Diagrammen können beispielsweise Programmablaufsteuerungen oder Nachrichtenparser realisiert werden.
Für den Anwendungsfall des Reglers stellt Matlab/Simulink, mit der Control System Toolbox, weitere Methoden zum Reglerentwurf bereit, die über das, in dieser Arbeit angewandte, empirische Verfahren hinausgehen.

Aber schon bei diesem Projekt haben sich die Vorteile der Codegenerierung mit Matlab /Simulink gezeigt.

Beachtlich ist, dass für die gesamte Reglerimplementierung auf dem Mikrocontroller, mit der Einbettung des generierten Codes in ein Programmgerüst, keine 100 Codezeilen von Hand programmiert werden mussten.

Bei der FPGA-Variante sieht dies, aufgrund der VHDL-Module für die Peripherie, deren Erstellung mit herkömmlicher Programmierung noch effizienter ist, anders aus.
Was die Einbettung solcher externer Module in ein Modell betrifft, herrscht beim HDL-Coder noch Nachholbedarf.

Insgesamt kann man sagen, dass der Entwurf von Algorithmen mit Simulink auf einer höheren Abstraktionsebene erfolgt. Typische Programmierfehler werden dadurch vermieden und der Entwickler kann sich auf das eigentliche Problem konzentrieren.

Ein wesentlicher Vorzug der modellgestützten Softwareentwicklung besteht darin, dass der Algorithmus, schon bevor er im realen System zum Einsatz kommt, in der Simulation erprobt werden kann. Bei sicherheitskritischen Systemen, in denen risikolose Tests nicht möglich sind, oder unter Umständen zur Zerstörung des Systems selbst führen, ist dies von großem Nutzen.

Jedoch muss darauf vertraut werden, dass der Codegenerator fehlerfreien Code aus dem Modell produziert. Bei generiertem C-Code kann dies nicht überprüft werden, da Simulink eine properitäre Bibliothek verwendet.

Gerade wegen der fehlenden Validierbarkeit der Software kommt der Simulink-eigene Codegenerator, z.B. zur Erstellung von Steuergerätesoftware für Automobile, zur Zeit noch nicht zum Einsatz.

Allerdings gibt es mittlerweile Drittanbieter, die Codegeneratoren für Simulink anbieten, die zertifizierten Code erzeugen.

Sicher ist in den nächsten Jahren noch eine interessante Entwicklung auf dem Sektor der modellbasierenden Codegenerierung zu beobachten. Nicht zuletzt, wenn sie mit andern modernen Methoden der Softwareentwicklung, wie Modell Driven Achitecture, kombiniert wird.

Anhang A: Literaturverzeichnis

[1] Fachhochschule Regensburg, EMS-Labor,
http://fbim.fh-regensburg.de/~ems_labor/

[2] SHARP: GP2D120 – General Purpose Type Distance Measuring Sensors- Datasheet

[3] Fabian Greif: Infrarot Entfernungsmesser
kreatives-chaos.com , Juli 2006
http://www.kreatives-chaos.com/artikel/infrarot-entferungsmesser

[4] Scherf H.E.: Modellbildung und Simulation dynamischer Systeme,
Oldenbourg Verlag, 2004

[5] Angermann A., et. al.: Matlab-Simulink-Stateflow
Oldenbourg Verlag, 2005

[6] Berger Manfred: Grundkurs der Regelungstechnik,
Books On Demand GmbH, 2001

[7] Spartan-3E Starter Kit Board User Guide,
Xilinx Ltd, 2006

[8] Ken Chapman: UART Transmitter and Receiver Macros,
Xilinx Ltd, 2003

Anhang B: Abbildungsverzeichnis

Anhang C: Abkürzungsverzeichnis

ADC	Analog-Digital Converter
ADU	Analog-Digital Umsetzer
API	Application Programming Interface
ASC	Asynchron / Synchron Communication
ASIC	Application Specific Integrated Circuit
CAN	Controller Area Network
CLB	Configurable Logic Blocks
COM	Serielle Schnittstelle
CPLD	Complex Programmable Logic Device
DAU	Digital Analog Umsetzter
DAvE	Digital Application virtual Engineer
EEPROM	Electrically Erasable Programmable Read-Only Memory
FPGA	Field Programmable Gate Array
GPIO	General Purpose Input/Output
GPT	General Purpose Timer
HDL	Hardware Description Language
IDE	Integrated Development Environment
IOB	Input/Output Block
IR	Infrarot
ISR	Interrupt Service Routine
LED	Light Emitting Diode
LIN	Local Interconnect Network
MAC	Multiply-and-Accumulate
MDA	Model Driven Achitecture
MOSFET	Metal Oxide Semiconductor Feld Effekt Transistor
PID	Proportional Integral Differenzial
PWM	Pulsweiten Modulation
RAM	Random Access Memory
RS-232	Serielle Schnittstele
SCI	Serial Communication Interface
SPI	Serial Pheripheral Interface
SRAM	Static Random Access Memory
tlc	Target Language Compiler
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus
VHDL	Very High Speed Integrated Circuit Hardware Description Language

Anhang D: Code-Listings

D1: .tlc-Dateien

xc167_adc.tlc

```
%% File : xc167_adc.tlc
%% Created: Sun Dec  3 22:35:54 2006
%%
%% Description:
%%   Real-Time Workshop wrapper functions interface generated for
%%   S-function "xc167_adc.c".
%%
%%         File generated by S-function Builder Block
%%
%%   For more information on using the Target Language with the
%%   Real-Time Workshop, see the Target Language Compiler manual
%%   (under Real-Time Workshop) in the "Inlining S-Functions"
%%   chapter under the section and subsection:
%%     "Writing Block Target Files to Inline S-Functions",
%%        "Function-Based or Wrappered Code".
%%
%implements  xc167_adc "C"
%% Function: BlockTypeSetup ===================================================
%%
%% Purpose:
%%      Set up external references for wrapper functions in the
%%      generated code.
%%
%function BlockTypeSetup(block, system) Output
  %openfile externs

  #include "ADC.h"

  %closefile externs
  %<LibCacheExtern(externs)>
  %%
%endfunction

%% Function: Start ============================================================
%%
%function Start(block, system) Output
   /* %<Type> Block: %<Name> */

  ADC_vInit();

%endfunction

%% Function: Outputs ==========================================================
%%
%% Purpose:
%%      Code generation rules for mdlOutputs function.
%%
%function Outputs(block, system) Output
   /* S-Function "xc167_adc_wrapper" Block: %<Name> */

  %assign py0 = LibBlockOutputSignal(0, "", "", 0)

  ADC_vStartConv(); //Wandlung starten
  while(ADC_ubBusy()==1); //Warten bis AD-Wandlung ausgeführt
  %<py0> = (ADC_uwReadConv() & 0x03ff); // Die untersten 10-Bit auslesen

  %%
%endfunction

%% Function: Terminate ========================================================
%%
%function Terminate(block, system) Output
   /* %<Type> Block: %<Name> */

%endfunction
%% [EOF] xc167_adc.tlc
```

xc167_pwm.tlc

```
%% File : xc167_pwm.tlc
%% Created: Mon Dec  4 12:28:30 2006
%%
%% Description:
%%   Real-Time Workshop wrapper functions interface generated for
%%   S-function "xc167_pwm.c".
%%
%%         File generated by S-function Builder Block
%%
%%   For more information on using the Target Language with the
%%   Real-Time Workshop, see the Target Language Compiler manual
%%   (under Real-Time Workshop) in the "Inlining S-Functions"
%%   chapter under the section and subsection:
%%     "Writing Block Target Files to Inline S-Functions",
%%        "Function-Based or Wrappered Code".
%%
%implements  xc167_pwm "C"
%% Function: BlockTypeSetup ===================================================
%%
%% Purpose:
%%      Set up external references for wrapper functions in the
%%      generated code.
%%
%function BlockTypeSetup(block, system) Output
  %openfile externs

  #include "CCU6.h"

  %closefile externs
  %<LibCacheExtern(externs)>
  %%
%endfunction


%% Function: Start ============================================================
%%
%function Start(block, system) Output
   /* %<Type> Block: %<Name> */

  CCU6_vInit();


%endfunction

%% Function: Outputs ==========================================================
%%
%% Purpose:
%%      Code generation rules for mdlOutputs function.
%%
%function Outputs(block, system) Output
   /* S-Function "xc167_pwm_wrapper" Block: %<Name> */

  %assign pu0 = LibBlockInputSignal(0, "", "", 0)

  CCU6_vLoadChannelShadowRegister_CCU6_CHANNEL_0(5000U - %<pu0>);
  CCU6_vEnableShadowTransfer_CCU6_TIMER_12();

  %%
%endfunction


%% Function: Terminate ========================================================
%%
%function Terminate(block, system) Output
   /* %<Type> Block: %<Name> */
   CCU6_vLoadChannelShadowRegister_CCU6_CHANNEL_0(5000U);
   CCU6_vEnableShadowTransfer_CCU6_TIMER_12();

%endfunction

%% [EOF] xc167_pwm.tlc
```

serial_IO.tlc

```
%% File : serial_IO.tlc
%% Created: Mon Jan  1 01:49:52 2007
%%
%% Description:
%%   Real-Time Workshop wrapper functions interface generated for
%%   S-function "serial_IO.c".
%%
%%         File generated by S-function Builder Block
%%
%%   For more information on using the Target Language with the
%%   Real-Time Workshop, see the Target Language Compiler manual
%%   (under Real-Time Workshop) in the "Inlining S-Functions"
%%   chapter under the section and subsection:
%%     "Writing Block Target Files to Inline S-Functions",
%%        "Function-Based or Wrappered Code".
%%
%implements  serial_IO "C"
%% Function: BlockTypeSetup ===================================================
%%
%% Purpose:
%%      Set up external references for wrapper functions in the
%%      generated code.
%%
%function BlockTypeSetup(block, system) Output
  %openfile externs

  #include "ASC1.h"
  %closefile externs
  %<LibCacheExtern(externs)>
  %%
%endfunction


%% Function: Start ============================================================
%%
%function Start(block, system) Output
   /* %<Type> Block: %<Name> */

  %assign sl_Soll = LibBlockOutputSignal(0, "", "", 0)
  %assign sl_P = LibBlockOutputSignal(1, "", "", 0)
  %assign sl_I = LibBlockOutputSignal(2, "", "", 0)
  %assign sl_D = LibBlockOutputSignal(3, "", "", 0)

%endfunction

%% Function: Outputs ==========================================================
%%
%% Purpose:
%%      Code generation rules for mdlOutputs function.
%%
%function Outputs(block, system) Output


  %assign sl_Soll = LibBlockOutputSignal(0, "", "", 0)
  %assign sl_P = LibBlockOutputSignal(1, "", "", 0)
  %assign sl_I = LibBlockOutputSignal(2, "", "", 0)
  %assign sl_D = LibBlockOutputSignal(3, "", "", 0)

  %<sl_Soll> = param_x;
  %<sl_P> = p_faktor;
  %<sl_I> = i_faktor;
  %<sl_D> = d_faktor;
  %%
%endfunction


%% Function: Terminate ========================================================
%%
%function Terminate(block, system) Output
   /* %<Type> Block: %<Name> */

%endfunction

%% [EOF] serial_IO.tlc
```

D2: Generierter C-Code

MAIN.C

```
/******************************************************************************
// @Module        Project Settings
// @Filename      MAIN.C
// @Project       DAVE-config.dav
//----------------------------------------------------------------------------
// @Controller    Infineon XC167CI-16F40
//
// @Compiler      Keil
//
// @Codegenerator 2.4
//
// @Description   This file contains the project initialization function.
//
//----------------------------------------------------------------------------
// @Date          21.12.2006 01:52:55
//
//******************************************************************************
//******************************************************************************
// @Project Includes
//******************************************************************************

#include "MAIN.H"
#include "PID.h"                       /* Model's header file */
#include "rtwtypes.h"                  /* MathWorks types */

static boolean_T OverrunFlag = 0;



//******************************************************************************
// @Function      void MAIN_vInit(void)
//
//----------------------------------------------------------------------------
// @Description   This function initializes the microcontroller.
//
//----------------------------------------------------------------------------
// @Returnvalue   None
//
//----------------------------------------------------------------------------
// @Parameters    None
//
//----------------------------------------------------------------------------
// @Date          21.12.2006
//
//******************************************************************************

void MAIN_vInit(void)
{

  MAIN_vUnlockProtecReg();       // unlock write security
  PLLCON         =  0x7889;      // load PLL control register


  //   ----------------------------------------------------------------------
  //   Initialization of the Peripherals:
  //   ----------------------------------------------------------------------

  //   initializes the Parallel Ports
  IO_vInit();
  //   initializes the General Purpose Timer Unit (GPT1)
  GPT1_vInit();
   //   initializes the Watchdog Timer (WDT)
  WDT_vInit();

  // USER CODE BEGIN (Init,3)
  // Defaultwerte für Reglereinstellungen
```

```
        param_x = 0.0;
        p_faktor = 150.0;
        i_faktor = 150.0;
        d_faktor = 10.0;

  // USER CODE END
} //  End of function MAIN_vInit


void rt_OneStep(void)
{
  /* Disable interrupts here */

  /* Check for overun */
  if (OverrunFlag++) {
    rtmSetErrorStatus(PID_M, "Overrun");
    return;
  }

  /* Save FPU context here (if necessary) */
  /* Re-enable timer or interrupt here */
  /* Set model inputs here */

  PID_step();
  WDT_vTriggerWDT();

  /* Get model outputs here */

  OverrunFlag--;

  /* Disable interrupts here */
  /* Restore FPU context here (if necessary) */
  /* Enable interrupts here */
}

void MAIN_vUnlockProtecReg(void)
{
  ubyte ubPASSWORD;

  if((SCUSLS & 0x1800) == 0x0800)      // if low protected mode
  {

    ubPASSWORD = (ubyte)SCUSLS & 0x00FF;
    ubPASSWORD = ~ubPASSWORD;
    SCUSLC = 0x8E00 | ubPASSWORD;      // command 4

  }  // end if low protected mode

  if((SCUSLS & 0x1800) == 0x1800)      // if write protected mode
  {
    SCUSLC = 0xAAAA;                   // command 0
    SCUSLC = 0x5554;                   // command 1

    ubPASSWORD = (ubyte)SCUSLS & 0x00FF;
    ubPASSWORD = ~ubPASSWORD;

    SCUSLC = 0x9600 | ubPASSWORD;      // command 2
    SCUSLC = 0x0800;                   // command 3; new PASSWOR is 0x00

    ubPASSWORD = (ubyte)SCUSLS & 0x00FF;
    ubPASSWORD = ~ubPASSWORD;
    SCUSLC = 0x8E00 | ubPASSWORD;      // command 4

  }  // end if write protected mode

} //  End of function MAIN_vUnlockProtecReg
```

```
//*****************************************************************************
// @Function      void main(void)
//-----------------------------------------------------------------------------
// @Description   This is the main function.
//-----------------------------------------------------------------------------
// @Returnvalue   None
//-----------------------------------------------------------------------------
// @Parameters    None
//-----------------------------------------------------------------------------
// @Date          21.12.2006
//
//*****************************************************************************

void main(void)
{
  // USER CODE BEGIN (Main,2)

  // USER CODE END

  MAIN_vInit();

  //   globally enable interrupts


  PID_initialize(1);

  /* Attach rt_OneStep to a timer or interrupt service routine with
   * period 0.04 seconds (the model's base sample time) here.  The
   * call syntax for rt_OneStep is
   *
   * rt_OneStep();
   */

   PSW_IEN        =  1;

  RSTCON  = RSTCON & 0x40;    //Configure ResetOut

  DP2 = 0xff00;
  P2 = ~0x100;

  while (rtmGetErrorStatus(PID_M) == NULL) {
    /*  Perform other application tasks here */

  }

} //  End of function main
```

PID.h

```
/*
 * File: PID.h
 *
 * Real-Time Workshop code generated for Simulink model PID.
 *
 * Model version                        : 1.92
 * Real-Time Workshop file version      : 6.3  (R14SP3)  26-Jul-2005
 * Real-Time Workshop file generated on : Sat Feb 10 17:14:46 2007
 * TLC version                          : 6.3 (Aug  5 2005)
 * C source code generated on           : Sat Feb 10 17:14:47 2007
 */

#ifndef _RTW_HEADER_PID_h_
#define _RTW_HEADER_PID_h_

#ifndef _PID_COMMON_INCLUDES_
# define _PID_COMMON_INCLUDES_
#include <math.h>
#include <float.h>
#include <stddef.h>
#include "rtwtypes.h"
#include "rtlibsrc.h"
#endif                                 /* _PID_COMMON_INCLUDES_ */

#include "PID_types.h"

/* Macros for accessing real-time model data structure  */

#ifndef rtmGetErrorStatus
# define rtmGetErrorStatus(rtm) ((void*) 0)
#endif

#ifndef rtmSetErrorStatus
# define rtmSetErrorStatus(rtm, val) ((void) 0)
#endif

/* Block signals (auto storage) */
typedef struct {
  real_T serial_IO_o1;                 /* '<S1>/serial_IO' */
  real_T serial_IO_o2;                 /* '<S1>/serial_IO' */
  real_T serial_IO_o3;                 /* '<S1>/serial_IO' */
  real_T serial_IO_o4;                 /* '<S1>/serial_IO' */
  real_T pwm;                          /* '<S1>/pwm' */
  uint16_T SFunction;                  /* '<S1>/S-Function' */
  uint16_T Gain;                       /* '<S1>/Gain' */
} BlockIO_PID;

/* Block states (auto storage) for system: '<Root>' */
typedef struct {
  real_T DiscreteTimeIntegrator_DSTATE; /* '<S1>/Discrete-Time Integrator' */
  real_T UD_DSTATE;                    /* '<S2>/UD' */
  real_T Soll;                         /* '<S1>/Data Store Memory' */
  real_T P;                            /* '<S1>/Data Store Memory1' */
  real_T I;                            /* '<S1>/Data Store Memory2' */
  real_T D;                            /* '<S1>/Data Store Memory3' */
} D_Work_PID;

/* External inputs (root inport signals with auto storage) */
typedef struct {
  real_T Ist;                          /* '<Root>/Ist' */
} ExternalInputs_PID;

/* External outputs (root outports fed by signals with auto storage) */
typedef struct {
  real_T Stell;                        /* '<Root>/Stell' */
} ExternalOutputs_PID;

/* Block signals (auto storage) */
extern BlockIO_PID PID_B;

/* Block states (auto storage) */
extern D_Work_PID PID_DWork;

/* External inputs (root inport signals with auto storage) */
extern ExternalInputs_PID PID_U;
```

```
/* External outputs (root outports fed by signals with auto storage) */
extern ExternalOutputs_PID PID_Y;

/* Model entry point functions */
extern void PID_initialize(boolean_T firstTime);
extern void PID_step(void);

/*
 * The generated code includes comments that allow you to trace directly
 * back to the appropriate location in the model.  The basic format
 * is <system>/block_name, where system is the system number (uniquely
 * assigned by Simulink) and block_name is the name of the block.
 *
 * Use the MATLAB hilite_system command to trace the generated code back
 * to the model.  For example,
 *
 * hilite_system('<S3>')    - opens system 3
 * hilite_system('<S3>/Kp') - opens and selects block Kp which resides in S3
 *
 * Here is the system hierarchy for this model
 *
 * '<Root>' : PID
 * '<S1>'   : PID/PID-Regler
 * '<S2>'   : PID/PID-Regler/Discrete Derivative
 * '<S3>'   : PID/PID-Regler/U => x
 */

#endif                                 /* _RTW_HEADER_PID_h_ */

/* File trailer for Real-Time Workshop generated code.
 *
 * [EOF]
 */
```

PID.c

```
/*
 * File: PID.c
 *
 * Real-Time Workshop code generated for Simulink model PID.
 *
 * Model version                        : 1.92
 * Real-Time Workshop file version      : 6.3  (R14SP3)  26-Jul-2005
 * Real-Time Workshop file generated on : Sat Feb 10 17:14:46 2007
 * TLC version                          : 6.3 (Aug  5 2005)
 * C source code generated on           : Sat Feb 10 17:14:47 2007
 */

#include "PID.h"
#include "PID_private.h"

/* Block signals (auto storage) */
BlockIO_PID PID_B;

/* Block states (auto storage) */
D_Work_PID PID_DWork;

/* External inputs (root inport signals with auto storage) */
ExternalInputs_PID PID_U;

/* External output (root outports fed by signals with auto storage) */
ExternalOutputs_PID PID_Y;

/* Model step function */
void PID_step(void)
{

  /* local block i/o variables*/
  real_T rtb_P;
  real_T rtb_TSamp;
  real_T rtb_Sum1;
  real_T rtb_Sum3;
  real_T rtb_DiscreteTimeIntegrator;
  real_T rtb_r64_temp27;

  /* S-Function "serial_IO_wrapper" Block: <S1>/serial_IO */

  PID_B.serial_IO_o1 = param_x;
  PID_B.serial_IO_o2 = p_faktor;
  PID_B.serial_IO_o3 = i_faktor;
  PID_B.serial_IO_o4 = d_faktor;

  /* DataStoreWrite: '<S1>/Data Store Write' */
  PID_DWork.Soll = PID_B.serial_IO_o1;

  /* DataStoreWrite: '<S1>/Data Store Write1' */
  PID_DWork.P = PID_B.serial_IO_o2;

  /* DataStoreWrite: '<S1>/Data Store Write2' */
  PID_DWork.I = PID_B.serial_IO_o3;

  /* DataStoreWrite: '<S1>/Data Store Write3' */
  PID_DWork.D = PID_B.serial_IO_o4;

  /* S-Function "xc167_adc_wrapper" Block: <S1>/S-Function */

  ADC_vStartConv();                     //Wandlung starten
  while(ADC_ubBusy()==1);               //Warten bis AD-Wandlung ausgeführt
  PID_B.SFunction = (ADC_uwReadConv() & 0x03ff); // Die untersten 10-Bit auslesen

  /* Sum: '<S1>/Sum3' incorporates:
   *  DataTypeConversion: '<S1>/Data Type Conversion'
   *  Gain: '<S3>/ADC => U'
   *  Sum: '<S3>/Subtract1'
   *  Product: '<S3>/Divide'
   *  Sum: '<S1>/Subtract'
   *  DataStoreRead: '<S1>/Data Store Read3'
   *  Constant: '<S1>/Constant'
   */
  rtb_Sum3 = PID_DWork.Soll - (0.1008 / ((real_T)PID_B.SFunction * 0.0029296875
    - 0.23) - 0.04);
```

```
/* Product: '<S1>/P' incorporates:
 *  DataStoreRead: '<S1>/Data Store Read'
 */
rtb_P = rtb_Sum3 * PID_DWork.P;

/* DiscreteIntegrator: '<S1>/Discrete-Time Integrator' */
rtb_DiscreteTimeIntegrator = PID_DWork.DiscreteTimeIntegrator_DSTATE;

/* Product: '<S1>/D' incorporates:
 *  DataStoreRead: '<S1>/Data Store Read2'
 */
rtb_r64_temp27 = rtb_Sum3 * PID_DWork.D;

/* S-Function (sfix_tsampmath): '<S2>/TSamp' */

/* Sample Time Math Block: '<S2>/TSamp'
 *
 * y = u * K     where     K = 1 / ( w * Ts )
 *
 * Input0  Data Type:  Floating Point real_T
 * Output0 Data Type:  Floating Point real_T
 * Saturation Mode: Saturate
 *
 * Parameter: WtEt == Weighted Elapsed Time
 *   Data Type:  Floating Point real_T
 *
 */
{
  rtb_TSamp = (rtb_r64_temp27*25.0);
}

{
  real_T rtb_dbl_rtsaturate_U0DataInY0Container;
  real_T rtb_dbl_tmp;

  /* Sum: '<S1>/Add' incorporates:
   *  UnitDelay: '<S2>/UD'
   *  Sum: '<S2>/Diff'
   *  Sum: '<S1>/Sum2'
   *  Product: '<S1>/P'
   */
  rtb_r64_temp27 = ((rtb_P + rtb_DiscreteTimeIntegrator) + (rtb_TSamp -
    PID_DWork.UD_DSTATE)) + 6.0;

  /* Saturate: '<S1>/Saturation' */
  rtb_dbl_rtsaturate_U0DataInY0Container = rt_SATURATE(rtb_r64_temp27, 0.0,
   12.0);

  /* Gain: '<S1>/Gain' */
  rtb_dbl_tmp = fmod(floor(rtb_dbl_rtsaturate_U0DataInY0Container *
    4.1667E+002), 65536.0);
  if(rtb_dbl_tmp < 0.0) {
    rtb_dbl_tmp += 65536.0;
  }
  PID_B.Gain = (uint16_T)rtb_dbl_tmp;

  /* Sum: '<S1>/Sum1' incorporates:
   *  DataStoreRead: '<S1>/Data Store Read1'
   *  Product: '<S1>/I'
   *  Sum: '<S1>/Sum'
   */
  rtb_Sum1 = rtb_Sum3 * PID_DWork.I - (rtb_r64_temp27 -
    rtb_dbl_rtsaturate_U0DataInY0Container);
}

/* S-Function "xc167_pwm_wrapper" Block: <S1>/pwm */

CCU6_vLoadChannelShadowRegister_CCU6_CHANNEL_0(5000U - PID_B.Gain);
CCU6_vEnableShadowTransfer_CCU6_TIMER_12();

/* Outport: '<Root>/Stell' */
PID_Y.Stell = PID_B.pwm;

/* Update for DiscreteIntegrator: '<S1>/Discrete-Time Integrator' */
PID_DWork.DiscreteTimeIntegrator_DSTATE = 0.04 * rtb_Sum1 +
  PID_DWork.DiscreteTimeIntegrator_DSTATE;
```

```
  /* Update for UnitDelay: '<S2>/UD' */
  PID_DWork.UD_DSTATE = rtb_TSamp;
}

/* Model initialize function */
void PID_initialize(boolean_T firstTime)
{

  if (firstTime) {

    /* S-Function Block: <S1>/serial_IO */

    /* S-Function Block: <S1>/S-Function */

    ADC_vInit();

    /* S-Function Block: <S1>/pwm */

    CCU6_vInit();

    /* Start for DataStoreMemory: '<S1>/Data Store Memory' */
    PID_DWork.Soll = 0.045;

    /* Start for DataStoreMemory: '<S1>/Data Store Memory1' */
    PID_DWork.P = 150.0;

    /* Start for DataStoreMemory: '<S1>/Data Store Memory2' */
    PID_DWork.I = 428.0;

    /* Start for DataStoreMemory: '<S1>/Data Store Memory3' */
    PID_DWork.D = 12.6;
  }
}

/* File trailer for Real-Time Workshop generated code.
 *
 * [EOF]
 */
```

D3: VHDL-Module

ADC_OVER_SPI.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity ADC_OVER_SPI is

PORT ( clk : in STD_LOGIC;  -- System Clock 50 MHz
               ce: in STD_LOGIC;
               RESET: in STD_LOGIC; -- Reset all SPI-Devices
               AMP_CS : out STD_LOGIC; -- AD Amp - ChipSelect
               AMP_SHDN: out STD_LOGIC; -- AD Amp - ShutDown
               SPI_MOSI: out STD_LOGIC; -- SPI FPGA out
               SPI_MISO: in STD_LOGIC; -- SPI FPGA in
               SPI_SCK: out STD_LOGIC; -- SPI FPGA clk
               DATA_A : out STD_LOGIC_VECTOR (13 downto 0); -- Output of ADC A;
               DATA_B : out STD_LOGIC_VECTOR (13 downto 0);  -- Output of ADC B;
               AD_CONV: out STD_LOGIC;  -- Trigger Output for ADC Conversion
               DAC_CLR: out STD_LOGIC; -- Reset DAC
               DAC_CS: out STD_LOGIC; -- ChipSelect Digital Analog Converter
               SPI_SS_B:out STD_LOGIC;
               SF_CE0:out STD_LOGIC;
               FPGA_INIT_B:out STD_LOGIC);
end ADC_OVER_SPI;


architecture Behavioral of ADC_OVER_SPI is

component ADC_AMP
       PORT ( CLK : in STD_LOGIC;
               RESET: in STD_LOGIC;
               ADC_AMP_EN : in STD_LOGIC;
               AMP_CS : out STD_LOGIC;
               AMP_SHDN: out STD_LOGIC;
               AMP_DONE: out STD_LOGIC;
               SPI_MOSI: out STD_LOGIC;
               SPI_SCK: out STD_LOGIC);
end component;

component wait_clk
    Port ( clk : in  STD_LOGIC;
           WAIT_EN : in  STD_LOGIC;
                       WAIT_DONE: out STD_LOGIC);
end component;


component ADC_CONV
   Port ( CLK : in  STD_LOGIC;
               RESET: in STD_LOGIC;
               ADC_CONV_EN : in STD_LOGIC;
               DATA_A : out STD_LOGIC_VECTOR (13 downto 0);
               DATA_B : out STD_LOGIC_VECTOR (13 downto 0);
               CONV_DONE: out STD_LOGIC;
               AD_CONV: out STD_LOGIC;
               SPI_MISO: in STD_LOGIC;
               SPI_SCK: out STD_LOGIC
               );
end component;

signal AMP_SPI_CLK : STD_LOGIC;
signal CONV_SPI_CLK : STD_LOGIC;
signal ADC_CONV_EN : STD_LOGIC;
signal CONV_DONE: STD_LOGIC;
signal ADC_AMP_EN : std_logic;
signal WAIT_EN:std_logic;
```

```
signal AMP_DONE: STD_LOGIC;
TYPE STATE_TYPE IS (Z0,Z1,Z2,Z3);
SIGNAL STATE, NEXT_STATE : STATE_TYPE;
signal clk2: STD_logic;
signal wait_done: std_logic;

signal TMP: STD_LOGIC_VECTOR(1 downto 0);

begin

-- This are the chip select ports for the other spi devices on the
-- Spartan 3E board.
-- Setting CS 'High' for disable them
SPI_SS_B <= '1';
SF_CE0 <= '1';
FPGA_INIT_B <= '1';
DAC_CLR <= '1';
DAC_CS <= '1';


TMP <= ADC_AMP_EN & ADC_CONV_EN;

        with TMP select
        SPI_SCK <= AMP_SPI_CLK when "10",
                                CONV_SPI_CLK when "01",
                                '0' when others;

-- Moore FSM

--      SPI_SCK <= AMP_SPI_CLK;

        PROCESS (clk2, NEXT_STATE)
        BEGIN
                IF CLK2='1' AND CLK2'event THEN
                        STATE <= NEXT_STATE;
                END IF;
        END PROCESS;


        PROCESS (STATE,WAIT_DONE,AMP_DONE,RESET,CONV_DONE)
        BEGIN


                ADC_AMP_EN <= '0';
                ADC_CONV_EN <= '0';

                NEXT_STATE<=Z0;
                WAIT_EN <= '0';

                IF ( RESET='1' ) THEN
                        NEXT_STATE<=Z0;
                ELSE
                        CASE STATE IS
                                WHEN Z0 =>

                                        IF ( WAIT_DONE='1' ) THEN
                                                NEXT_STATE<=Z1;
                                        ELSE
                                                NEXT_STATE<=Z0;
                                                WAIT_EN <= '1';
                                                end if;

                                WHEN Z1 =>
                                        IF ( AMP_DONE='1' ) THEN
                                                NEXT_STATE<=Z2;
                                        ELSE
                                           NEXT_STATE<=Z1;
                                                ADC_AMP_EN <= '1';
                                        END IF;
                                --WHEN Z2 =>
                                --              NEXT_STATE<=Z2;

                                                WHEN Z2 =>
                                        IF ( CONV_DONE='1' ) THEN
                                                NEXT_STATE<=Z3;
                                        ELSE
                                                NEXT_STATE<=Z2;
                                                ADC_CONV_EN <= '1';
```

```
                                END IF;
                            WHEN Z3 =>
                                        NEXT_STATE<=Z2;


                        END CASE;
                END IF;
        END PROCESS;

process(clk)
variable clkint: STD_LOGIC := '0';
begin
                if clk ='1' and clk'event then
                        if ce = '1' then
                                clkint := not clkint;
                                clk2 <= clkint;
                        end if;
                end if;
end process;




CONV1: ADC_CONV port map
                ( CLK2,
                 RESET,
                 ADC_CONV_EN,
                 DATA_A,
                 DATA_B,
                 CONV_DONE,
                 AD_CONV,
                 SPI_MISO,
                 CONV_SPI_CLK);

WAT1: wait_clk
   Port map (clk2,
           WAIT_EN,
                  WAIT_DONE);


AMP1: ADC_AMP  PORT map
          ( CLK2,
                 RESET,
                 ADC_AMP_EN,
                 AMP_CS,
                 AMP_SHDN,
                 AMP_DONE,
                 SPI_MOSI,
                 AMP_SPI_CLK );

end Behavioral;
```

ADC_CONV.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity ADC_CONV is
    Port ( CLK : in  STD_LOGIC;
                RESET : in STD_LOGIC;
                ADC_CONV_EN : in STD_LOGIC;
                DATA_A : out STD_LOGIC_VECTOR (13 downto 0);
                DATA_B : out STD_LOGIC_VECTOR (13 downto 0);
                CONV_DONE: out STD_LOGIC;
                AD_CONV: out STD_LOGIC;
                SPI_MISO: in STD_LOGIC;
                SPI_SCK: out STD_LOGIC);
end ADC_CONV;

architecture ADC_CONV_RTL of ADC_CONV is
signal SPI_CLOCK_EN: STD_LOGIC;
signal COUNT: integer range 0 to 36 ;

begin

SPI_SCK <= CLK when SPI_CLOCK_EN = '1' else '0';

process (CLK)

variable TEMP: STD_LOGIC_VECTOR (0 to 29);
variable AD_CONV_INT: STD_LOGIC;
variable AD_CONV_DONE_INT: STD_LOGIC;
begin
       if CLK='0' and CLK'event then

              AD_CONV_INT := '0';
              AD_CONV_DONE_INT := '0';

              if RESET = '1' then
                     SPI_CLOCK_EN <= '0';
                     count <= 0;
              elsif ADC_CONV_EN  = '1' then

                     count <= count + 1;
                 case count is
                     when 0 =>  AD_CONV_INT := '1';
                                       SPI_CLOCK_EN <= '0';
                                       CONV_DONE <= '0';
                     when 1 =>  SPI_CLOCK_EN <= '1';
                     when 5 to 34 =>
                                       TEMP(count-5) := SPI_MISO;
                     when 35 => SPI_CLOCK_EN <= '0';
                                       DATA_A <= TEMP(0 to 13);
                                       DATA_B <= TEMP(16 to 29);
                     when 36 => AD_CONV_DONE_INT := '1';
                                       count <= 0;
                     when others =>
                     end case;
              else
                     count <= 0;
              end if;
       end if;

       CONV_DONE <= AD_CONV_DONE_INT;
       AD_CONV <= AD_CONV_INT;
end process;
end ADC_CONV_RTL;
```

Wait_clk.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity wait_clk is
    Port ( clk : in  STD_LOGIC;
           WAIT_EN : in  STD_LOGIC;
                       WAIT_DONE: out STD_LOGIC);
end wait_clk;

architecture Behavioral of wait_clk is

signal cnt : integer range 0 to 128;

begin

process(clk, WAIT_EN)
begin

       if clk'event and clk='1' then
              WAIT_DONE <= '0';

              if     WAIT_EN ='1' then
                     cnt<= cnt + 1;
                     if cnt = 128 then
                            WAIT_DONE<='1';
                            cnt <= 0;
                     end if;
              end if;
       end if;

end process;

end Behavioral;
```

ADC_AMP.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity ADC_AMP is
        PORT ( CLK : in STD_LOGIC;
                RESET: in STD_LOGIC;
                ADC_AMP_EN : in STD_LOGIC;
                AMP_CS : out STD_LOGIC;
                AMP_SHDN: out STD_LOGIC;
                AMP_DONE: out STD_LOGIC;
                SPI_MOSI: out STD_LOGIC;
                SPI_SCK: out STD_LOGIC);
end ADC_AMP;

architecture ADC_AMP_RTL of ADC_AMP is
signal clk_100ns : STD_LOGIC;
signal DATA_OUT : STD_LOGIC_VECTOR(0 to 7);
signal CLK_OUT: STD_LOGIC;
signal DATA: std_logic;
signal cnt: integer range 0 to 10 :=0;

begin

        DATA_OUT <= "00010001";
        SPI_SCK <= CLK_100ns when CLK_OUT = '1' else '0';

        SPI_MOSI <= DATA when ADC_AMP_EN = '1' else '0';

process (CLK_100ns, ADC_AMP_EN)

        variable amp_done_int: std_logic;
        variable amp_shdn_int: std_logic;
        variable amp_cs_int: std_logic;

        begin

        if CLK_100ns = '0' and CLK_100ns'event then

        amp_shdn_int := '0';
        amp_done_int := '0';

        DATA <= '0';
        AMP_CS_INT := '1';


                if RESET = '1' then

                        AMP_SHDN_INT := '1';
                        cnt<= 0;
                        CLK_OUT <= '0';

                elsif ADC_AMP_EN = '1' then
                        cnt <= cnt +1;
                        case cnt is

                        when 0 => AMP_CS_INT := '0';
                        when 1 to 8 => CLK_OUT <= '1';
                                                                DATA <= DATA_OUT(cnt-1);
                                                                AMP_CS_INT := '0';
                        when 9 => CLK_OUT <= '0';
                                                AMP_CS_INT := '0';
                        when 10 => amp_done_int := '1';
                        end case;
           else
                        cnt <= 0;
                end if;
        end if;
```

```
        AMP_DONE <= amp_done_int;
        AMP_SHDN <= AMP_SHDN_INT;
        AMP_CS <= AMP_CS_INT;

end process;

process(CLK)
variable cnt: integer range 0 to 5 := 0;
variable clk_100ns_int: STD_LOGIC := '0';
begin
                if CLK ='1' and CLK'event then
                        cnt := cnt +1;
                        if cnt = 5 then
                                clk_100ns_int := not clk_100ns_int;
                                cnt:= 0;
                        end if;
                        clk_100ns <= clk_100ns_int;
                end if;
end process;

end ADC_AMP_RTL;
```

PWM.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity PWM is
  Port ( clk : in  STD_LOGIC; --The onboard clk of the Spartan3E(C9)
                       ce: in STD_LOGIC; -- needed for SystemGenerator
          PWM_OUT : out STD_LOGIC;    -- Output of PWM Signal
         VALUE : in  STD_LOGIC_VECTOR (13 downto 0); -- 0?10000 => 0-100%
          PWM_EN: in STD_LOGIC);     -- enable PWM
end PWM;

architecture RTL of PWM is
  signal clk_int: integer range 0 to 10000;
begin

process(clk, PWM_EN)
    variable PWM_int: std_logic;
begin
  if PWM_EN /= '1' then
       clk_int <= 0;
   PWM_int := '0';

  elsif clk'event and clk = '1' then
       if ce = '1' then
              clk_int <= clk_int + 1;
              if clk_int <= conv_integer(unsigned(VALUE)) then
                     PWM_int := '1';
              else
                     PWM_int := '0';
              end if;
              if clk_int = 10000 then
                     clk_int <= 0;
              end if;

       end if;
  end if;

  PWM_OUT <= PWM_int;

end process;

end RTL;
```

Uart.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity uart is
       Port( clk: in std_logic;
                      ce: in std_logic;
                      serial_in : in std_logic;
                      data: out std_logic_vector(7 downto 0);
                      new_data: out std_logic);
end uart;

architecture Behavioral of uart is

component clkDiv is
    Port ( clk : in  STD_LOGIC;
           ce : in  STD_LOGIC;
                       clk_2 : out  STD_LOGIC;
           clk_16 : out  STD_LOGIC);
end component;

component kcuart9_rx is
   Port (      serial_in : in std_logic;
                 data_out : out std_logic_vector(8 downto 0);
               data_strobe : out std_logic;
             en_16_x_baud : in std_logic;
                      clk : in std_logic);
    end  component;

signal parity: std_logic;
signal data_out: std_logic_vector( 8 downto 0 );
signal data_strobe: std_logic;
signal clk_38400: std_logic;
signal clk_2: std_logic;

begin

parity <= (((data_out(7) xor data_out(6))xor( data_out(5) xor data_out(4))) xor ((data_out(3)
xor data_out(2))xor( data_out(1) xor data_out(0))));

process (data_strobe, clk_2)
begin
       new_data <='0';

       if data_strobe = '1' and parity = data_out(8) then
              data <= data_out(7 downto 0);
              new_data <='1';
       end if;
end process;

cdiv: clkDiv          Port map ( clk, ce, clk_2, clk_38400);
kcu:   kcuart9_rx  Port map (serial_in, data_out, data_strobe, clk_38400, clk_2);

end Behavioral;


clkDiv.vhd

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;
```

clkDiv.vhd

```
entity clkDiv is
    Port ( clk : in  STD_LOGIC;
           ce : in  STD_LOGIC;
                       clk_2: out STD_LOGIC;
           clk_16 : out  STD_LOGIC);
end clkDiv;

architecture Behavioral of clkDiv is

signal clk_2_int: std_logic;

begin

clk_2 <= clk_2_int;

process ( clk, ce)
variable clk_int: std_logic := '0';
begin

if clk'event and clk ='1' then
       if ce = '1' then
                      clk_int := not clk_int;
       end if;

       clk_2_int <= clk_int;
end if;

end process;

process ( clk_2_int )
variable cnt: integer range 0 to 41;
begin

if clk_2_int'event and clk_2_int='1' then
       clk_16 <= '0';
       cnt := cnt + 1;

       if cnt = 41 then
              clk_16 <= '1';
              cnt := 0;
       end if;

end if;

end process;

end behavioral;
```

D4: HDL-Coder- Zusatzmodule

FPGA_Regler.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity FPGA_Regler_HDL_CODER is
  port (
    clk: in std_logic;
    serial_in: in std_logic;
    --gateway_in3: in std_logic_vector(13 downto 0);
    spi_miso: in std_logic;
    ad_conv: out std_logic;
    amp_cs: out std_logic;
    amp_shdn: out std_logic;
    dac_clr: out std_logic;
    dac_cs: out std_logic;
    fpga_init_b: out std_logic;
    pwm_out: out std_logic;
    sf_ce0: out std_logic;
    spi_mosi: out std_logic;
    spi_sck: out std_logic;
    spi_ss_b: out std_logic
  );
end FPGA_Regler_HDL_CODER;

architecture rtl of FPGA_Regler_HDL_CODER is

component ADC_OVER_SPI
PORT ( clk : in STD_LOGIC;  -- System Clock 50 MHz
               ce: in STD_LOGIC;
               RESET: in STD_LOGIC; -- Reset all SPI-Devices
               AMP_CS : out STD_LOGIC; -- AD Amp - ChipSelect
               AMP_SHDN: out STD_LOGIC; -- AD Amp - ShutDown
               SPI_MOSI: out STD_LOGIC; -- SPI FPGA out
               SPI_MISO: in STD_LOGIC; -- SPI FPGA in
               SPI_SCK: out STD_LOGIC; -- SPI FPGA clk
               DATA_A : out STD_LOGIC_VECTOR (13 downto 0); -- Output of ADC A;
               DATA_B : out STD_LOGIC_VECTOR (13 downto 0);  -- Output of ADC B;
               AD_CONV: out STD_LOGIC;  -- Trigger Output for ADC Conversion
               DAC_CLR: out STD_LOGIC; -- Reset DAC
               DAC_CS: out STD_LOGIC; -- ChipSelect Digital Analog Converter
               SPI_SS_B:out STD_LOGIC;
               SF_CE0:out STD_LOGIC;
               FPGA_INIT_B:out STD_LOGIC);
end component;

component uart
       Port( clk: in std_logic;
                     ce: in std_logic;
                     serial_in : in std_logic;
                     data: out std_logic_vector(7 downto 0);
                     new_data: out std_logic);
end component;
component FPGA_Regler
  PORT( clk                               :   IN    std_logic;
        reset                             :   IN    std_logic;
        clk_enable                        :   IN    std_logic;
        Ist_in                            :   IN    std_logic_vector(13 DOWNTO 0);  -- ufix14
        Soll_in                           :   IN    std_logic_vector(7 DOWNTO 0);  -- ufix8
        ce_out                            :   OUT   std_logic;
        Stell_out                         :   OUT   std_logic_vector(13 DOWNTO 0)  -- ufix14
        );
END component;
component PWM
  Port ( clk : in  STD_LOGIC; --The onboard clk of the Spartan3E(C9)
```

```
                      ce: in STD_LOGIC; -- needed for SystemGenerator
          PWM_OUT : out STD_LOGIC;    -- Output of PWM Signal
         VALUE : in  STD_LOGIC_VECTOR (13 downto 0); -- 0?10000 => 0-100%
          PWM_EN: in STD_LOGIC);     -- enable PWM
end component;

component clk_wrapper
       Port ( clk : in  STD_LOGIC;
       ce : in  STD_LOGIC;
                clk_smp: out STD_LOGIC;
       smp_done: out  STD_LOGIC);
end component;

signal PWM_VALUE: STD_LOGIC_VECTOR ( 13 downto 0);
signal PWM_EN: STD_LOGIC;
signal clk_smp: STD_LOGIC;
signal smp_done: STD_LOGIC;
signal clk_smp_enable: STD_LOGIC;
signal Stell_out: std_logic_vector (13 downto 0);
signal serial_data: std_logic_vector ( 7 downto 0);
signal Soll_in: std_logic_vector (7 downto 0) := "00110000";
signal new_data: std_logic;
signal ADC_DATA_A: STD_LOGIC_VECTOR (13 downto 0); -- Output of ADC A;
signal ADC_DATA_B: STD_LOGIC_VECTOR (13 downto 0); -- Output of ADC A;
signal IST_IN: STD_LOGIC_VECTOR (13 downto 0); -- Output of ADC A;
signal RESET: STD_LOGIC;
signal ce: std_logic;
signal ce_out: std_logic;

begin
       PWM_EN <= '1';
       clk_smp_enable <= '1';
       reset <= '0';
       ce <= '1';

        process (new_data)
       begin
              if new_data'event and new_data = '1' then
                     Soll_in <= serial_data;
              end if;
       end process;
       process (smp_done)
       begin
              if smp_done'event and smp_done = '1' then
                     PWM_Value <= Stell_out;
              end if;
       end process;

       process (clk_smp)
       begin
              if clk_smp'event and clk_smp = '1' then
                     IST_in <= ADC_DATA_A;
              end if;
       end process;

pwm1: PWM      Port map ( clk, ce, PWM_OUT, PWM_VALUE, PWM_EN);
Regler1: FPGA_Regler PORT map( clk_smp, reset, clk_smp_enable, Ist_in, Soll_in, ce_out,
Stell_out );
ADC1: ADC_OVER_SPI PORT map ( clk,  -- System Clock 50 MHz
                              ce,
                              RESET,
                              AMP_CS,
                              AMP_SHDN,
                              SPI_MOSI,
                              SPI_MISO,
                              SPI_SCK,
                              ADC_DATA_A,
                              ADC_DATA_B,
                              AD_CONV,
                              DAC_CLR,
                              DAC_CS,
                              SPI_SS_B,
                              SF_CE0,
                              FPGA_INIT_B);
serial1: uart Port map( clk, ce, serial_in, serial_data, new_data);
clk_wr1: clk_wrapper Port map ( clk, ce, clk_smp, smp_done);

end rtl;
```

clk_wrapepr.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity clk_wrapper is
       Port ( clk : in  STD_LOGIC;
       ce : in  STD_LOGIC;
                clk_smp: out STD_LOGIC;
       smp_done: out  STD_LOGIC);
end clk_wrapper;

architecture rtl of clk_wrapper is


       signal cnt: integer range 0 to 1900000;

begin


process ( clk, ce)
       variable clk_smp_int: std_logic :='0';
begin

if clk'event and clk ='1' then
       if ce = '1' then
              cnt <=cnt+1;
              smp_done <= '0';
              if cnt = 0 then
                     clk_smp_int := not clk_smp_int;
              end if;
              if cnt = 1000 then
                     smp_done <= '1';
              end if;
              if cnt = 950000 then
                     clk_smp_int := not clk_smp_int;
              end if;
              if cnt = 1900000 then
                     cnt <= 0;
              end if;

              clk_smp <= clk_smp_int;
       end if;
end if;

end process;

end rtl;
```