Masterthesis

David Kellermann

Ansteuerkonzept für kaskadierte piezoelektrische Linearmotoren für den Sub-Nanometer-Bereich

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

David Kellermann

Ansteuerkonzept für kaskadierte piezoelektrische Linearmotoren für den Sub-Nanometer-Bereich

Masterthesis eingereicht im Rahmen der Masterprüfung
im Masterstudiengang Automatisierung
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Rasmus Rettig
Zweitgutachter : Prof. Dr.-Ing. Florian Wenck

Abgegeben am 18. September 2014

David Kellermann

Thema der Masterthesis

Ansteuerkonzept für kaskadierte piezoelektrische Linearmotoren für den Sub-Nanometer-Bereich

Stichworte

Ansteuerung, Konzept, Kaskade, Piezo, Linear, Motor, Nanometer, Regler, FPGA, VHDL, MATLAB, Simulink, Arduino, Verstärker

Kurzzusammenfassung

Diese Arbeit beschreibt die Entwicklung eines Ansteuerkonzepts für kaskadierte piezoelektrische Linearmotoren. Sie umfasst die Simulation eines Piezoaktors und -motors mit MATLAB/Simulink, die Entwicklung eines vierkanäligen Piezoverstärkers mit Serial Peripheral Interface, sowie die softwareseitige Ansteuerung des Motors mittels FPGA/Raspberry PI.

David Kellermann

Title of the paper

Control concept for cascaded piezoelectric linear motors for the sub-nanometer range

Keywords

Controlling, concept, cascade, piezo, linear, drive, motor, nanometer, controller, FPGA, VHDL, MATLAB, Simulink, Arduino, Amplifier

Abstract

In this report the control concept for cascaded piezoelectric linear motors is described. It includes the simulation of a piezoelectric actuator and motor with MATLAB/Simulink, the development of a four channel piezoamplifier with Serial Peripheral Interface, as well as the software-based control of the piezodrive using FPGA / Raspberry PI.

Inhaltsverzeichnis

Tabellenverzeichnis 6

Abbildungsverzeichnis 7

Listings 10

Abkürzungsverzeichnis 11

1. Einleitung 12
1.1. Einführung und Motivation 12
1.2. Stand der Technik 13
1.2.1. Wurm- und Schreitantriebe 13
1.2.1.1. Inchworm-Motor 13
1.2.1.2. Piezo-LEGS-Motor 14
1.2.2. Weitere Piezoaktoren 15
1.3. Nachteile der aktuellen Technik 16
1.4. Zielsetzung 17

2. Grundlagen 19
2.1. Direkter und inverser piezoelektrischer Effekt 19
2.2. Piezokeramik 19
2.3. Verwendeter Aktor 20
2.4. Mathematische Beschreibung des Piezoaktors 21
2.4.1. Nichtlinearität 21
2.4.2. Hysterese 22
2.4.3. Kriechen 23
2.4.4. Temperatureffekte 24

3. Der Piezomotor 25

4. Modellbildung 27
4.1. Simulation des Piezoaktors 27
4.1.1. Hystereseberechnung 29
4.1.2. Driftberechnung 33
4.2. Simulation des Piezomotors 38
4.2.1. Fahrlogik 39
4.2.2. Inverse Steuerung 41
4.2.3. Optimierung der Wellenform 43

5. Realisierung der Ansteuerung 45
5.1. Konzept 1: Ansteuerung mit Digitalverstärker 45
5.1.1. PWM-Signalerzeugung mittels Mikrocontroller 45
5.1.2. PWM-Signalerzeugung mittels FPGA 47
5.1.3. MOSFET-Verstärkerschaltung 50
5.2. Konzept 2: Ansteuerung mit Analogverstärker 52
5.2.1. Signalerzeugung mittels FPGA 53
5.2.2. Raspberry Pi Logic Controller 55
5.2.3. Analog-Verstärkerschaltung 57
5.2.3.1. Prototyp 1 57
5.2.3.2. Prototyp 2 63
5.2.4. Leistungsbetrachtung des Verstärkers und Aktors 66
5.2.4.1. Gesamtleistungsaufnahme des Aktors 66
5.2.4.2. Verlustleistung 68
5.2.5. Reglerentwurf 69
5.2.5.1. Erfassung der Regelgröße 69
5.2.5.2. Inverse Steuerung in VHDL 71
5.2.5.3. Positionsregelung in VHDL 73
5.2.5.4. Geschwindigkeitsregelung in VHDL 76

6. Messungen und Auswertung 80
6.1. Messungen des Gleichlaufs 81
6.1.1. Vergleich Standard- und optimierte Wellenform 82
6.1.2. Vergleich Motorprototyp mit Piezo-LEGS-Motor 83
6.1.3. Ursache der Gleichlaufunruhe 85
6.2. Messungen der Positioniergenauigkeit 87
6.2.1. Positionsoptimierung mittels inverser Steuerung 87
6.2.2. Positionsregelung 88

7. Zusammenfassung und Ausblick 90
7.1. Zusammenfassung 90
7.2. Ausblick 92

Literaturverzeichnis 94

A. Schaltpläne 98

B. Rechnungen 102
B.1. Mittlere Leistung 102
B.1.1. Rechtecksignal 102
B.1.2. Sägezahnsignal 103

C. Quellcode 104

D. CD 124

Tabellenverzeichnis

1.1. Weitere Piezoaktoren 16
1.2. Spezifikationen 18

5.1. Bestimmung des Counteroffsets zwischen Lade- und Entladekurve 71
5.2. PID-Parameter 76

6.1. Auswertung des Piezomotorenvergleichs 84

Abbildungsverzeichnis

1.1. Bewegungsablauf Inchworm-Motor 13
1.2. Ansteuerung Inchworm-Motor 14
1.3. Piezo-LEGS-Motor Bewegungsablauf 15
1.4. Wellenform des Piezo-LEGS-Motor 15
1.5. Positions- und Geschwindigkeitsplot eines Piezo-LEGS-Motor 16
1.6. Querschnitt durch einen Piezo-LEGS-Motor 17

2.1. Direkter und inverser Piezoeffekt 19
2.2. Elektromechanisches Schema eines Stapelaktors 20
2.3. Verwendeter Aktor 21
2.4. Elektrische Dipole in den Domänen 22
2.5. Hysterese des verwendeten Piezostacks 22
2.6. Positionskriechen eines Piezoaktors 23

3.1. CAD-Zeichnung des Piezomotors 25
3.2. Realer Piezomotor 25
3.3. Bewegungsablauf Piezomotorprototyp 26

4.1. Messung von Hystereseschleifen 27
4.2. Punkte für die Polynomberechnung 28
4.3. Berechnung eines Polynompunktes 29
4.4. Ablaufdiagramm der Hystereseberechnung 30
4.5. Fall 1: Berechnung des Ladeasts ohne Verlassen der aktuellen Schleife . 31
4.6. Fall 2: Berechnung des Ladeasts mit Verlassen der aktuellen Schleife . . 31
4.7. Herauslaufen mit Schneiden 32
4.8. Herauslaufen mit fehlerhafter Skalierung 32
4.9. Fall 3: Berechnung des Entladeasts ohne Verlassen der aktuellen Schleife 32
4.10. Fall 4: Berechnung des Entladeasts mit Verlassen der aktuellen Schleife . 33
4.11. Simulinkmodell Piezostack mit Hysterese- und Driftberechnung 33
4.12. Piezostackausdehnung 1 34
4.13. Piezostackausdehnung 2 35
4.14. Piezostackausdehnung 3 35
4.15. Entladekurve über Ladekurve 36
4.16. Offset bei negativen Spannungssprüngen 36
4.17. Vergleich von realem und simuliertem Aktor 37
4.18. Vorüberlegung Wellenform zur Ansteuerung des Motors 38
4.19. Simulinkmodell des gesamten Piezomotors 39

4.20. Simulinkmodell der Fahrlogik 40
4.21. Fehlerhafter Algorithmus 40
4.22. Korrekter Algorithmus 40
4.23. Vergleich der Differentiationsmethoden 41
4.24. Inverse Steuerung eines Übertragungssystems 41
4.25. Inverse Steuerung 42
4.26. Simulationsvergleich der Positionsänderung vom Motor mit und ohne inverser Steuerung 42
4.27. Optimierte Wellenform zur Ansteuerung des Piezomotors 44

5.1. VHDL-Beschreibung zur Wellenformgenerierung mittels PWM 48
5.2. Prinzip des Phasenakkumulators 50
5.3. PWM-Signalerzeugung durch Sinus-Dreieck-Modulation 51
5.4. Topologie des Schaltverstärkers 51
5.5. Gefilterte Wellenform 52
5.6. Sternförmige SPI-Busstruktur 53
5.7. VHDL Blocksymbol des SPI-Masters 54
5.8. SPI Übertragungsprotokoll 54
5.9. Wellenform des PiezoMotor PMD90 Controller 1 57
5.10. Wellenform des PiezoMotor PMD90 Controller 2 57
5.11. Prinzip des nichtinvertierenden Verstärkers 58
5.12. Erster Schaltungsentwurf auf einer Lochrasterplatine für einen Kanal . . . 59
5.13. Bestückte Verstärkerplatine Version 1.0 59
5.14. Fehlerhafte Verstärkung der Operationsverstärker 60
5.15. Bestückte Verstärkerplatine Version 1.1 61
5.16. Ausgangssignale des Verstärkers Version 1.1 61
5.17. Einbruch der Wellenform 62
5.18. Stromverlauf Class AB-Push-Pull-Verstärker 64
5.19. Bestückte Verstärkerplatine Version 2.0 65
5.20. Strom- und Spannungsverlauf des Piezoaktors 1 66
5.21. Strom- und Spannungsverlauf des Piezoaktors 1 66
5.22. Messung von Spannung, Strom und mittlere Leistung 67
5.23. Elektrisches Ersatzschaltbild eines Piezoelements 68
5.24. Zählweise des Graycodes 70
5.25. Quadratursignal mit Richtungswechsel 70
5.26. VHDL Blocksymbol des Encoders 70
5.27. Sprünge im Ausgangssignal beim Richtungswechsel durch falsche Counterwerte 71
5.28. Richtungswechsel bei inverser Steuerung 72
5.29. Gegenüberstellung der Piezoausdehnung 72
5.30. Prinzip des Positionsregelalgorithmus 73
5.31. Positionsregelalgorithmus in ModelSim 75
5.32. Position und Geschwindigkeitsänderung in ModelSim 78
5.33. Komplettes Blockschema der finalen FPGA-Beschreibung 79

6.1. Messaufbau 80
6.2. Diagramm des Messaufbaus 81
6.3. Positions- und Geschwindigkeitsplot 1 82
6.4. Positions- und Geschwindigkeitsplot 2 82
6.5. Standardwellenform 83
6.6. Optimierte Wellenform 83
6.7. Vergleich des Motorprototyps mit Piezo-LEGS-Motor 84
6.8. Vergleich der Positionsänderungen der Motoren für eine Wellenformperiode 85
6.9. Positionsänderung des Motorprototyp 85
6.10. Vergleich der Positionsänderung mit der Wellenform 86
6.11. Vergleich der Positionsänderung des Läufers bei Ansteuerung mit regulärer und inverser Wellenform 87
6.12. Positionsregelung 1 88
6.13. Positionsregelung 2 89

A.1. Platinenlayout Piezoverstärker Version 1.1 98
A.2. Schaltplan Piezoverstärker Version 1.1 99
A.3. Platinenlayout Piezoverstärker Version 2.0 100
A.4. Schaltplan Piezoverstärker Version 2.0 101

Listings

5.1. Auszug aus dem C-Quellcode des Raspberry Pi zum Steuern des FPGAs 56
5.2. Auszug aus dem VHDL-Quellcode zur Positionsregelung 74
5.3. Auszug aus dem VHDL-Quellcode des PID-Reglers 76

C.1. C-Quellcode des Raspberry Pi zum Steuern des FPGAs 104
C.2. VHDL-Quellcode der PWM 110
C.3. VHDL-Quellcode des Encoders 111
C.4. VHDL-Quellcode des PID-Reglers 112
C.5. VHDL-Quellcode des Zweipunktreglers 113
C.6. VHDL-Quellcode des Funktionsgenerators 1 114
C.7. VHDL-Quellcode des Funktionsgenerators 2 118
C.8. VHDL-Quellcode des SPI-Masters 122

Abkürzungsverzeichnis

CAD .... Computer-Aided Design
CS .... Chip Select
DAC .... Digital-Analog-Converter
DC .... Direct Current
DDS .... Direct Digital Synthesis
DESY .... Deutsches Elektronen-Synchrotron
FPGA .... Field Programmable Gate Array
GUI .... Graphical User Interface
HASYLAB .... Hamburger Synchrotronstrahlungslabor
I/O .... Input/Output
LUT .... Look-Up-Table
LVTTL .... Low-Voltage-TTL
MISO .... Master Input, Slave Output
MOSFET .... Metal-Oxide-Semiconductor Field-Effect-Transistor
MOSI .... Master Output, Slave Input
PID-Controller ... Proportional-Integral-Derivative-Controller
PiLC .... Raspberry Pi Logic Controller
PLL .... Phase-Locked-Loop
PTC .... Positive Temperature Coefficient
PWM .... Pulsweitenmodulation
PZT .... Plumbum-Zirkonat-Titanat
SCLK .... Serial Clock
SPI .... Serial Peripheral Interface
SS .... Slave Select
TTL .... Transistor-Transistor-Logik
UHV .... Ultrahochvakuum
USB .... Universal Serial Bus
VHDL .... Very High Speed Integrated Circuit Hardware Description Language

1. Einleitung

Die vorliegende Arbeit beschreibt die Entwicklung eines Ansteuerkonzepts für kaskadierte piezoelektrische Linearmotoren für den Sub-Nanometer-Bereich von der Planung bis zur Realisierung. Sie ist in Kooperation mit dem Deutschen Elektronen-Synchrotron DESY im Bereich der Forschung mit Synchrotronstrahlung HASYLAB an der Beamline *P11* angefertigt worden.
Die Arbeit ist in folgende Kapitel unterteilt: Im zweiten Kapitel wird im allgemeinen auf die Entdeckung und Funktion des piezoelektrischen Effekts und deren gegenwärtige Nutzung eingegangen. Nachdem Kapitel 3 den für diese Arbeit entwickelten Linearmotor vorstellt, erläutert Kapitel 4 den Vorgang sowohl die Piezokeramik als Solche als auch den gesamten Aktor als annähernd realistische Simulation umzusetzen. Die hieraus gewonnenen Erkenntnisse werden in Kapitel 5 angewendet und zwei verschiedene Konzepte für die Umsetzung in Hardware aufgezeigt. Messungen an der Hardware und deren Ergebnisse und Auswertungen werden in Kapitel 6 dargestellt. Die Arbeit schließt mit einer Zusammenfassung und einem Ausblick im letzten Kapitel.

1.1. Einführung und Motivation

Die Eigenschaft piezoelektrischer Materialien, elektrische Energie direkt in mechanische Energie umzusetzen, findet heute in vielen Bereichen Anwendung. Sei es in alltäglichen Produkten, wie Feuerzeugen oder Lautsprechern, als auch in Industrie und Forschung (z.B. piezogetriebene Einspritzventile in Dieselmotoren).
Speziell bei Experimenten mit Synchrotronstrahlung sind Piezoaktoren verstärkt im Einsatz, da hier stets höhere Positioniergenauigkeiten bis in den Sub-Nanometer-Bereich gefordert werden. Da eine direkte Abhängigkeit der Ausdehnung der Piezoaktoren von der angelegten Spannung besteht, wird die erreichte Auflösung nur durch die Spannungsauflösung begrenzt. Das Experimentedesign wird jedoch durch die Bauform der am Markt erhältlichen Aktoren oftmals eingeschränkt. Weiterhin ist die erreichbare Positioniergenauigkeit bestehender Systeme zwar meistens sehr gut ($< 1nm$), allerdings wäre eine höhere Gleichlaufruhe oftmals wünschenswert. Weitere nachteilige Aspekte sind Verschleiß und hohe Anschaffungskosten. Aus den genannten Punkten heraus entstand die Idee, im Rahmen dieser Masterthesis den Grundstein für eine eigene Entwicklung am DESY im Bereich der Nanostelltechnik mit Piezoaktoren zu legen.
Im folgenden Abschnitt werden zunächst einige der derzeit am Markt erhältlichen Aktoren mit ihren Vor- und Nachteilen vorgestellt.

1.2. Stand der Technik

Da es heute eine Fülle verschiedenster Piezoaktoren gibt, werden hier nur die am weitverbreitetsten und vorzugsweise im Forschungsbereich eingesetzten Aktoren vorgestellt. Den meisten dieser Aktoren liegen Stapel- oder Multilayeraktoren zu Grunde, auf die im Kapitel 2.2 noch näher eingegangen wird. Um den begrenzten Stellweg dieser Aktoren zu erhöhen, werden oftmals mechanische Wegvergrößerungssysteme wie z.B. Festkörpergelenke eingesetzt. Durch sie ist es möglich, Aktoren zu realisieren, die nicht von der Ausdehnung des Piezoelements begrenzt werden. Aus diesem Grund werden diese Aktoren auch Piezomotoren genannt.

1.2.1. Wurm- und Schreitantriebe

1.2.1.1. Inchworm-Motor

„Der bekannteste Vertreter des Wurmantriebs ist der Inchworm-Motor, der 1975 erstmals kommerziell angeboten wurde. Die Bezeichnung dieses Linearmotors rührt daher, dass sein Bewegungsablauf dem der Spannerraupe (engl. inchworm) ähnelt. Abbildung 1.1 beschreibt das Prinzip: Eine glatte Welle (Läufer), die axial positioniert werden soll, wird von drei Piezoelementen umschlossen [Aktor 1, Aktor 2, Aktor 3]. [...] Werden [...] [die äußeren] Elemente angesteuert, so klemmen sie die Welle fest. Der mittlere Hohlzylinder hat großes Spiel und dehnt sich beim Anlegen einer Spannung in axialer Richtung, d.h. dieses Element sorgt für den Vorschub“ (Janocha, 2013, S. 42).

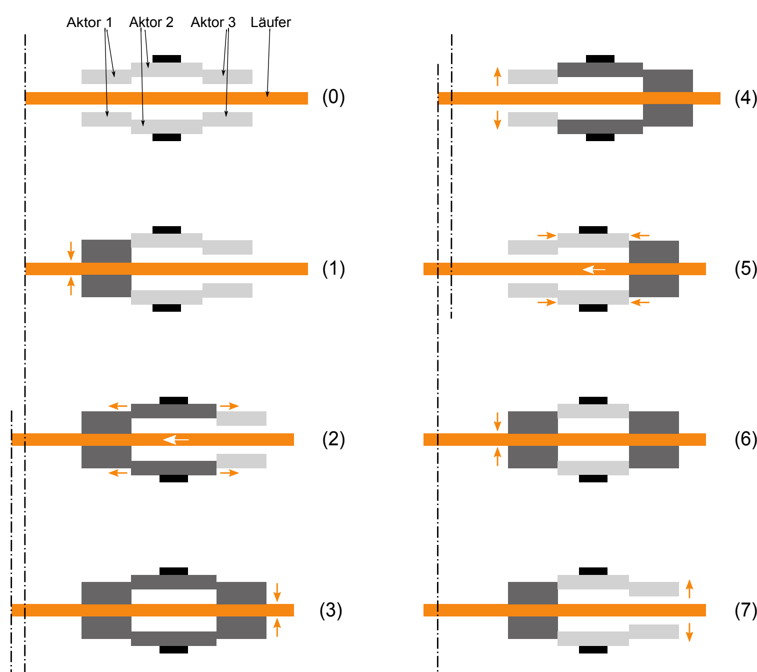


Abbildung 1.1.: Bewegungsablauf Inchworm-Motor (nach Janocha, 2013, S. 43)

1. Aktor 1 dehnt sich aus und klemmt so den Läufer ein.
2. Aktor 2 dehnt sich aus und verschiebt so den Läufer um einen Schritt nach links.
3. Aktor 3 dehnt sich aus und klemmt den Läufer ebenfalls ein.
4. Aktor 1 öffnet sich.
5. Aktor 2 entspannt sich und schiebt den Läufer dadurch um einen weiteren Schritt nach links.
6. Aktor 1 klemmt den Läufer erneut ein.
7. Aktor 3 öffnet sich. Der Ablauf beginnt von neuem.

Die beiden Klemmaktoren werden jeweils mit einem Rechtecksignal angesteuert, wohingegen der Vorschubaktor prinzipbedingt mit einem Dreiecksignal versorgt wird, dessen Stufenunterteilung die Schrittweite und dessen Steilheit der Treppe die Geschwindigkeit festlegt, wie auf Abbildung 1.2 zu sehen ist (vgl. Janocha, 2013, S. 42-43).

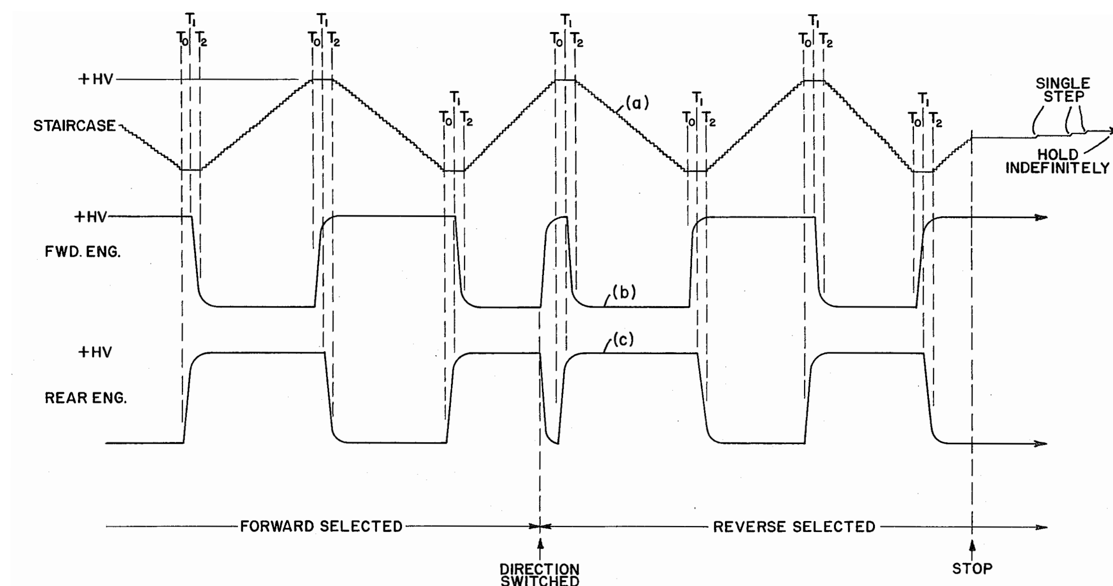


Abbildung 1.2.: Ansteuerung Inchworm-Motor der Fa. Burleigh (May, 1975)

1.2.1.2. Piezo-LEGS-Motor

Ein weiterer, weit verbreiteter Aktor ist der Piezo-LEGS-Motor der Firma PiezoMotor Schweden. Er ist einer der häufigsten Vertreter der Nanostelltechnik im Forschungsbereich mit Synchrotronstrahlung am DESY. „Monolithische Multilayer-Blöcke werden hier [...] mit jeweils zwei elektrisch isolierten Elektrodenbereichen versehen und bilden auf diese Weise ein sog. Bein (eng. leg)“ (Janocha, 2013, S. 43). Wird nun an diese Beine eine elektrische Spannung angelegt, lassen sich sowohl Längs- als auch Biegebewegungen erzeugen (vgl. Janocha, 2013, S. 43-44). Abbildung 1.3 veranschaulicht diesen

Vorgang. Die jeweils dunklere Seite erfährt eine höhere Spannung.

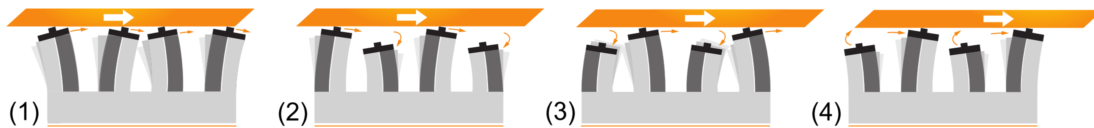


Abbildung 1.3.: Piezo-LEGS-Motor Bewegungsablauf (nach PiezoMotor, 2014b, S. 2)

Abbildung 1.4 stellt Messungen der zugehörigen Wellenform mit unterschiedlichen Auflösungen für jedes der vier Beinchen dar (Spannung über Zeit).

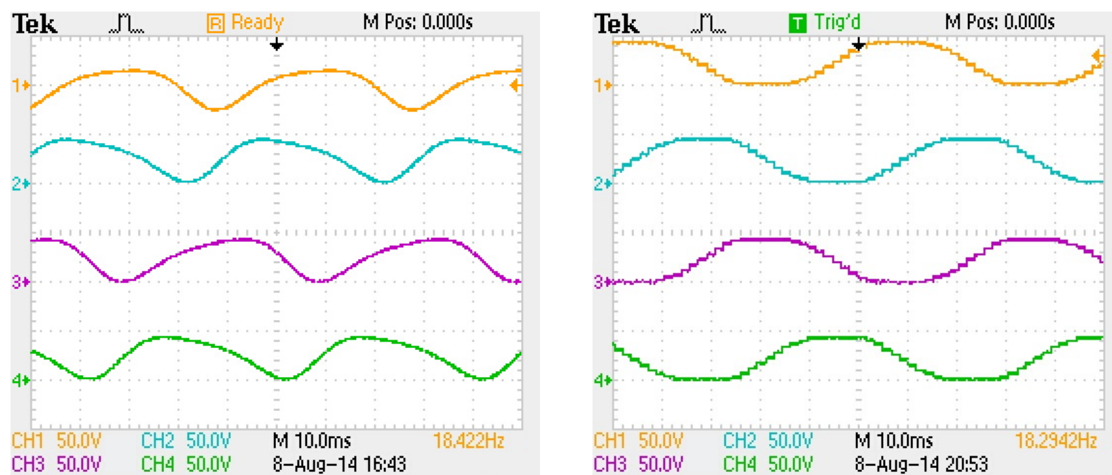


Abbildung 1.4.: Wellenform des Piezo-LEGS-Motor mit hoher (links) und niedriger (rechts) Auflösung

Der Läufer, der von einer Feder auf die Piezoblöcke gedrückt wird, wird also durch eine fortlaufende „Krabbelbewegung“ der Beine translatorisch bewegt. Laut Hersteller beträgt die Schrittweite $4\mu m$ und die maximale Geschwindigkeit $15mm/s$ bei einer Treiberspannung von $0 \ldots 48V$ (vgl. PiezoMotor, 2014b, S. 4).

1.2.2. Weitere Piezoaktoren

Neben den vorgestellten und für die vorliegende Arbeit wichtigsten Vertretern der Piezoaktoren, sind noch viele weitere Aktoren am Markt erhältlich, von denen einige nachfolgend tabellarisch aufgeführt sind. Für nähere Informationen sei auf weiterführende Literatur verwiesen, wie z.B. (Janocha, 2013, S. 45-54).

Produktname	Hersteller	Bewegungsart	Anwendung (Beispiele)
Elliptec-Motor	Elliptec	Translatorisch und rotatorisch	Stellbewegung, Verriegelung
Wandlerwellenmotor	Shinsei	Rotatorisch	Kameraobjektive
Picomotor	Newport	Translatorischer Vorschub	Laserstrahlstabilisierung
Piezo Actuator Drive (PAD)	Siemens/Noliac	Rotatorisch	Medizingerätetechnik, Luft- und Raumfahrt
Squiggle-Motor	Newscale	Translatorischer Vorschub	Positionierung der Kameralinse in Mobiltelefonen

Tabelle 1.1.: Weitere Piezoaktoren (Janocha, 2013, S. 45-54)

1.3. Nachteile der aktuellen Technik

Dieser Abschnitt betrachtet einige Kriterien des unter Abschnitt 1.2.1 vorgestellten und derzeit zum Einsatz kommenden Piezo-LEGS-Motors.

Kriterium 1: Gleichlauf
Verschiedene Messungen des LEGS-Motors haben gezeigt, dass der Gleichlauf nicht optimal verläuft, sprich: Nicht linear. Die Schwankungen sind aller Wahrscheinlichkeit nach dem mechanischem Aufbau des Motors geschuldet, da es durch die 4 Beine laufend zu einer „Umgreifproblematik“ kommt: Nachdem zwei Beine den Läufer um einen Schritt bewegt haben, lösen sie sich vom Läufer und die restlichen zwei Beine bewegen den Läufer um einen Schritt weiter. Im Moment des Umgreifens zwischen Beinpaar 1 und 2 kommt es zu einem Geschwindigkeitseinbruch. Auf Abbildung 1.5 ist sowohl die Positionsänderung als auch die Geschwindigkeit über die Zeit mit den Schwankungen zu sehen.

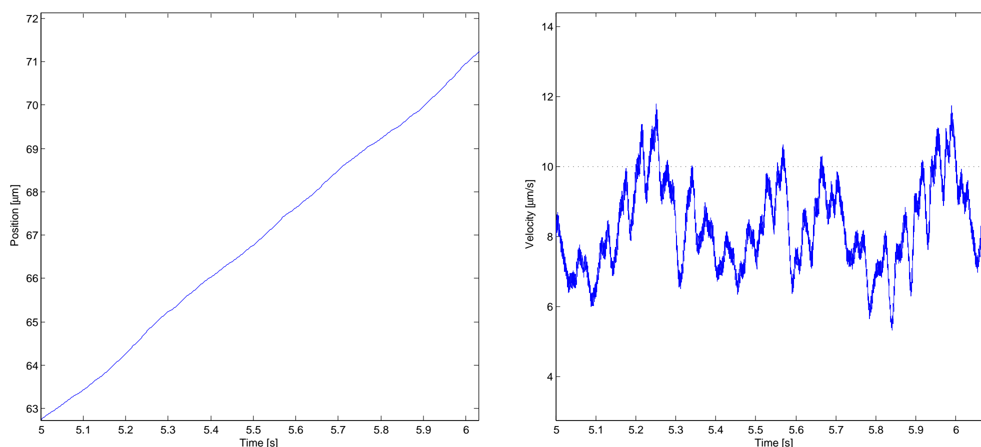


Abbildung 1.5.: Positions- und Geschwindigkeitsplot eines Piezo-LEGS-Motor

Die Messungen sind bewusst im open-loop-Betrieb, also ungeregelt, aufgenommen worden, um eine möglichst genaue Vorstellung von der Problematik zu bekommen. Dem Motor wurde eine Geschwindigkeit von $10\mu m/s$ vorgegeben (gestrichelte Linie), die auf Grund des ungeregelten Betriebs nicht ganz erreicht wird.

Kriterium 2: Verschleiß
Da der Läufer direkt auf den Piezoaktoren reibschlüssig aufliegt, besteht die Gefahr, durch zu große äußere Krafteinwirkung auf den Läufer, in oder entgegen der Bewegungsrichtung, die Aktoren zu beschädigen. Abbildung 1.6, die einen Querschnitt durch einen solchen Piezomotor darstellt, verdeutlicht die eben genannte Problematik.

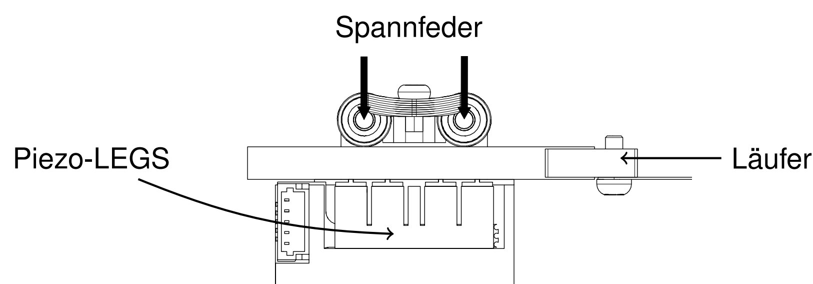


Abbildung 1.6.: Querschnitt durch einen Piezo LEGS Linear 6N (PiezoMotor, 2014a)

Kriterium 3: Kosten
Ein weiterer nicht zu vernachlässigender Punkt betrifft die Anschaffungskosten. Ein komplettes System in der einfachen Ausführung (wird als Referenz für diese Arbeit herangezogen), bestehend aus Motor, Ansteuerung und Software[1] beläuft sich derzeit auf ca. 1500-2000 Euro, je nach Ausführung des Motors und der Ansteuerung. Die Kosten für die professionelle Ausführung hingegen (höhere Genauigkeit, mehr Schnittstellen, bis zu 8 Motoren anschließbar) liegen um die 10000 Euro.

1.4. Zielsetzung

Nachdem in den vorangegangenen Abschnitten auf die Ausgangssituation sowie auf den Stand der Technik mit seinen Vor-und Nachteilen eingegangen wurde, soll in diesem Abschnitt das Ziel der Arbeit konkretisiert werden.
Den bereits im Abschnitt 1.1 erwähnten Nachteilen mit den derzeit im Einsatz befindlichen Piezomotoren wurde bisher, auf Grund mangelnder Alternativen, weitestgehend mit Akzeptanz begegnet. Die zu lösende Aufgabe besteht somit in der Entwicklung eines

[1] betrifft den ungeregelter Betrieb, da für die Regelung verschiedene Systeme anderer Hersteller verwendet werden können

Produkts, welches sowohl die Hardware (Motor, Verstärker) als auch die Software umfasst und dabei diese Punkte optimiert. Dabei liegt von Anfang an das größte Augenmerk auf einer sehr guten Gleichlaufruhe, gefolgt von einer hohen Schrittauflösung sowie einer Positioniergenauigkeit, die mindestens im Bereich $\pm 2nm$ liegt. Weitere Punkte sind ein insgesamt robusterer und variablerer Aufbau der Hardware im Vergleich zu bestehenden Systemen sowie günstigere Anschaffungskosten der Einzelkomponenten. So ergibt sich folgender Anforderungskatalog:

Größe	**Bereich**	**Einheit**	**Anmerkung**
Geschwindigkeit	$0 - 5$	*mm/s*	ohne Last
Schrittauflösung	$0.002 - 5$	μm	ohne Last
Positioniergenauigkeit	≤ 2	*nm*	Geregelter Betrieb
Frequenz[2]	$0 - 5000$	*Hz*	-
Treiberspannung	$0 - 120$	*V*	-
Kosten	< 5000	€	Motor, Verstärker, Ansteuersoftware

Tabelle 1.2.: Spezifikationen

Der Fahrbereich vor und zurück ist bedingt durch die Bauform theoretisch unendlich und wird nur durch die Länge des Läufers begrenzt.

[2]Wiederholfrequenz der Wellenform

2. Grundlagen

Dieses Kapitel gibt eine kurze Einführung in die physikalischen Grundlagen der Piezoelektrizität sowie die Phänomene der Nichtlinearität, Hysterese- und Kriecheffekte. Zudem wird der für die vorliegende Arbeit zum Einsatz kommende Piezoaktor vorgestellt, anhand dessen diese Eigenschaften verdeutlicht werden.

2.1. Direkter und inverser piezoelektrischer Effekt

Im Jahre 1880 veröffentlichten die Brüder Curie erstmals, dass das Ausüben von Druck auf Quartzkristalle an deren Oberfläche elektrische Ladung erzeugt. Vom griechischen Wort Piezo (altgr. *πιεζειν* *piezein* ‚drücken') abgeleitet, nannten sie dieses Phänomen Piezoeffekt. Genau genommen handelt es sich hierbei um den direkten piezoelektrischen Effekt (Abbildung 2.1 links). Diesem gegenüber steht der reziproke oder inverse piezoelektrische Effekt, bei dem es durch Anlegen einer elektrischen Spannung an einen Piezokristall zu einer Verformung desselben kommt (Abbildung 2.1 rechts) (Absatz nach Hegewald, 2007, S. 17).

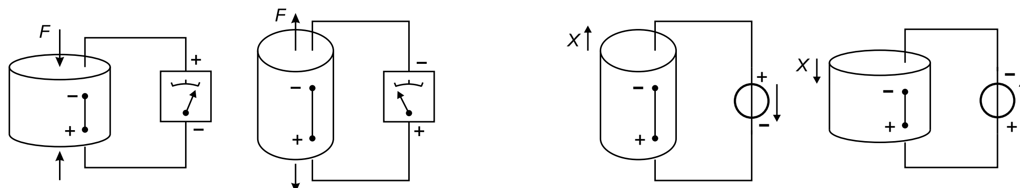


Abbildung 2.1.: Direkter (links) und inverser (rechts) Piezoeffekt (Wang, 2006, S. 7)

2.2. Piezokeramik

Erst 66 Jahre nach der Entdeckung des direkten und inversen Piezoeffekts wurde auch in Keramiken Piezoelektrizität beobachtet. Besonders stark ausgeprägt ist dieser Effekt bei Blei-Zirkonat-Titanat-Keramiken (PZT), weshalb diese auch bis heute am häufigsten für piezoelektrische Bauteile zum Einsatz kommen (nach Physik Instrumente, 2009, S.

2-177). Um mit den Keramiken größere Auslenkungen zu erzielen, werden sie zu sogenannten Stapelaktoren übereinandergeschichtet und bilden die Gruppe der Piezoaktoren mit begrenzter Auslenkung (als Pendant dazu vgl. Kapitel 1.2, Aktoren mit unbegrenzter Auslenkung). Abbildung 2.2 verdeutlicht, wie die piezoelektrischen Keramikscheiben paarweise mit entgegengesetzter Polarisation übereinandergeschichtet werden. „Auf diese Weise sind sie elektrisch parallel und mechanisch in Reihe geschaltet" (Janocha, 2013, S. 31).

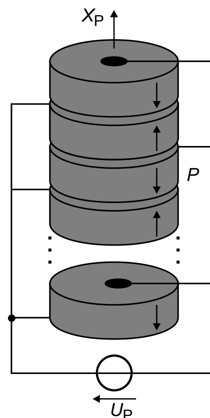


Abbildung 2.2.: Elektromechanisches Schema eines Stapelaktors (nach Wang, 2006, S. 12)

Durch Anlegen einer elektrischen Spannung U_P dehnen sich die Einzelelemente auf Grund der wechselnden Polarisation P in entgegengesetzter Richtung aus. Daraus resultiert die Längenänderung X_P des Aktors (nach Janocha, 2013, S. 32). Da Stapelaktoren nur Druckkräften ausgesetzt werden dürfen (auf Grund der Steifigkeit der Keramik), muss bei Anwendungen mit Zugkräften der Aktor mechanisch vorgespannt werden (vgl. Physik Instrumente, 2009, S. 2-192). Dieser Punkt ist bei der Realisierung eines Piezomotors, falls dieser auch „ziehen" soll, zu beachten.

2.3. Verwendeter Aktor

Der in dieser Arbeit verwendete Aktor ist ein Multilayer-Aktor der PICMA Serie der Firma Physik Instrumente. Er ist unter anderem gewählt worden, da er besonders gut für Positionierung im Sub-Nanometer-Bereich geeignet ist, bei gleichzeitig sehr hoher Ausdehnung im Verhältnis zur angelegten Spannung. Weiterhin ist er unempfindlich gegen Luftfeuchtigkeit und ultrahochvakuumkompatibel bis $10^{-9} hPa$, was ihn besonders für den Einsatz im Forschungsbereich mit Vakuumanforderungen (Forschung mit Synchrotronstrahlung) attraktiv macht (vgl. Physik Instrumente, 2014, S. 1). Der Stack des Typs P-883.11 weist folgende Daten auf:

- Abmessung (AxBxL) [mm]: 3x3x9
- Nominalstellweg [μm] (0-100V): 6,5 $\pm$ 20%
- Max. Stellweg [μm] (0-120V): 8 $\pm$ 20%
- Blockierkraft [N] (0-120V): 290
- Steifigkeit [$N/\mu m$]: 36
- Elektrische Kapazität [μF]: 0,21 $\pm$ 20%
- Resonanzfrequenz [kHz]: 135 $\pm$20%

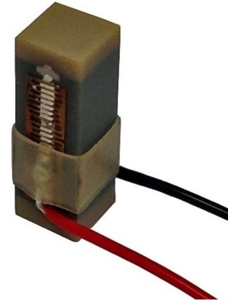


Abbildung 2.3.: PICMA Stack Multilayer-Piezoaktor P-883.11

2.4. Mathematische Beschreibung des Piezoaktors

Neben den vielen genannten Vorteilen wird in diesem Abschnitt auf den wesentlichen Nachteil der Nichtlinearität piezoelektrischer Aktoren eingegangen. Die chemischen und physikalischen Zusammenhänge, die zu diesem Effekt führen, werden dabei vereinfacht behandelt.

2.4.1. Nichtlinearität

Eine PZT-Keramik „[..] ist nach der Herstellung zunächst nicht piezoelektrisch“ (Hegewald, 2007, S. 24). Sie besteht aus einer Vielzahl von Kristallen, die sich bei sinkender Temperatur asymmetrisch verzerren und so eine spontane Polarisation aufweisen (vgl. Physik Instrumente, 2009, S. 2-181). Hierbei bilden sich in den Kristallen Dipole aus (vgl. Abb. 2.4 (1)). Bereiche gleicher Ausrichtung werden als Domänen bezeichnet. An dieser Stelle ist der piezoelektrische Effekt jedoch noch nicht nutzbar, da die Dipole unterschiedliche Ausrichtungen aufweisen und diesen Effekt daher neutralisieren. Erst durch Anlegen eines starken elektrischen Feldes ist es möglich, die Dipole in eine Richtung auszurichten und so die gewünschten piezoelektrischen Eigenschaften (Ausdehnung der Keramik) zu nutzen (vgl. Abb. 2.4 (2)). Es fällt auf, dass die Polarisation nach Entfernen des elektrischen Feldes weitestgehend erhalten bleibt (vgl. Abb. 2.4 (3)) (Absatz nach Gnad, 2005, S. 68-70). „Die Keramik besitzt jetzt piezoelektrische Eigenschaften und verändert beim Anlegen einer elektrischen Spannung ihre Dimension“ (Physik Instrumente, 2009, S. 2-181). Durch gegenseitige Behinderung bei der Ausrichtung der Domänen im Kristallgitter beim Lade- bzw. Entladevorgang des Aktors kommt es zu einer hysteresebehafteten Nichtlinearität.

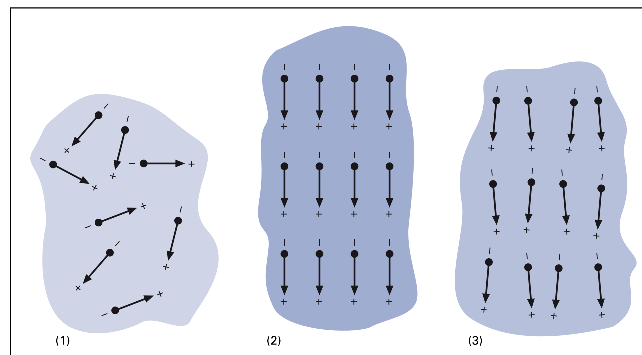


Abbildung 2.4.: Elektrische Dipole in den Domänen; unpolarisierte Keramik (1), während (2) und nach (3) der Polung (piezoelektrische Keramik) (Physik Instrumente, 2009, S. 2-181).

2.4.2. Hysterese

Da im ungeregelten Betrieb die genannte Nichtlinearität der Ausdehnungs-Spannungskurve sowohl beim Lade- als auch beim Entladevorgang auftritt, kommt es zur Hysteresebildung. Auf Abbildung 2.5 sind die gemessene Lade- und Entladekurve des verwendeten Aktors, die daraus resultierende Hysterese sowie eine Referenzkurve, die den idealen Ausdehnungsverlauf darstellt, zu sehen. Dafür ist im Abstand von 10 Volt jeweils ein Ausdehnungswert aufgenommen worden.

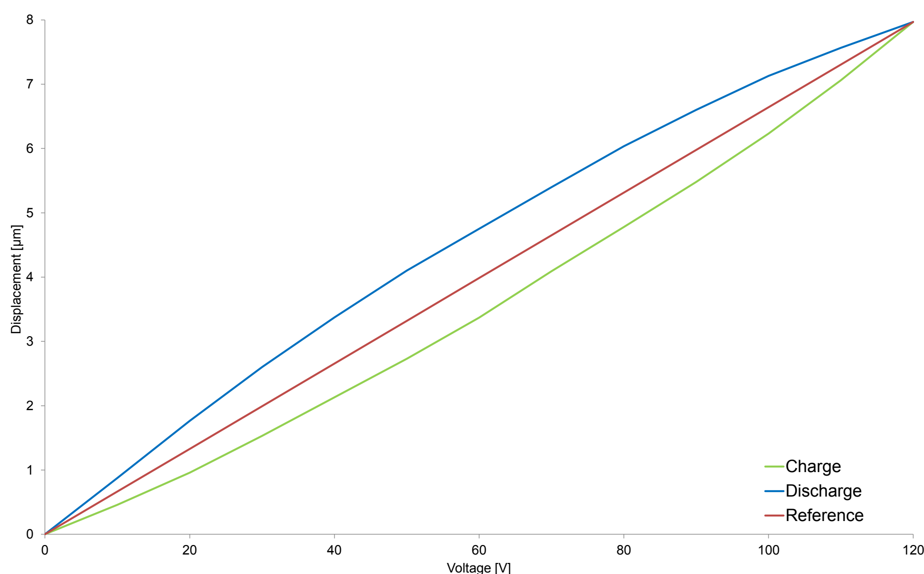


Abbildung 2.5.: Hysterese des P883.11 Stacks

Die Hysterese nimmt mit der Betriebsspannung (elektrisches Feld) des Aktors zu. Die „Öffnung“ der Spannungs-[...][Ausdehnungs]kurven beginnt typisch bei 2% (Kleinsignal) und erreicht unter Großsignalbedingungen ein Maximum von 10-15% (Physik Instrumente, 2009, S. 2-185).

Durch bestimmte Maßnahmen lassen sich die Nichtlinearitäten weitestgehend kompensieren. Hierfür stehen verschiedene Möglichkeiten zur Wahl: Die wohl verbreitetste Methode ist die Regelung der Ausgangsgröße des Aktors. Dabei wird die aktuelle Position gemessen und dem Regler zurückgeführt. Eine weitere Methode ist die inverse Steuerung. Bei dieser Methode wird der Aktor mit einem invertierten Signal seines eigenen Übertragungsverhaltens angesteuert. Dies geschieht in offener Wirkungskette, also sensorlos. Die größte Schwierigkeit dieser Methode ist, dass ein möglichst exaktes Modell der Nichtlinearitäten vorliegen muss (vgl. Janocha, 2013, S. 283). Auf diese Methode wird in Kapitel 4 noch näher eingegangen.
Weiterhin besteht die Möglichkeit der Hysteresekompensation durch Ladungssteuerung statt Spannungssteuerung, da bei Piezoelementen Ausdehnung und Ladung einen fast linearen Zusammenhang bilden. Der Nachteil dieser Methode ist, dass die permanente Integration des Lade- und Entladestroms fehlerbehaftet ist und dieser Fehler mit der Zeit zunimmt (vgl. Janocha, 2013, S. 60).

2.4.3. Kriechen

Neben der Hysteresebildung ist das Kriechen ein weiterer Effekt piezoelektrischer Aktoren. „Kriechen beschreibt die Positionsänderung über die Zeit bei unveränderter Steuerspannung“ (Physik Instrumente, 2009, S. 2-186). „Dieses Nachlaufen ist durch Nachpolarisation der Keramik begründet“ (Piezosystem Jena, 2014, S. 13). „Die Kriechgeschwindigkeit nimmt logarithmisch mit der Zeit ab [...]. In der Praxis erreicht die durch Kriechen bedingte Positionsänderung nach einigen Stunden einige Prozent der ursprünglichen Positionierung“ (Physik Instrumente, 2009, S. 2-186). Abbildung 2.6 verdeutlicht diesen Effekt. $s(T_s)$ gibt die Auslenkung an, die zum Zeitpunkt T_s entstanden ist.

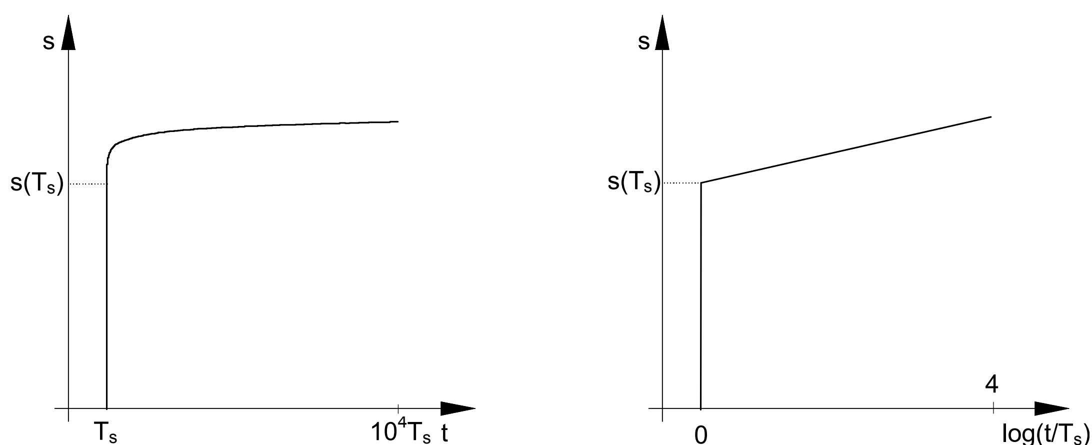


Abbildung 2.6.: Positionskriechen eines Piezoaktors mit linearer (links) und logarithmischer (rechts) Zeitachse (Kuhnen, 2001, S. 17)

2.4.4. Temperatureffekte

Zwei Effekte gilt es zu beachten: Die lineare thermische Ausdehnung und die Temperaturabhängigkeit des Piezoeffekts. Ersterer beschreibt das Verhalten verschiedener Piezokeramiken von ihrer thermischen Dehnung bei unterschiedlichen Temperaturen. Er ist also hauptsächlich relevant, wenn in einem System verschiedene Keramiken zum Einsatz kommen.
Die Temperaturabhängigkeit des Piezoeffekts beschreibt hingegen, in welchem Temperaturbereich Piezoaktoren stabil arbeiten. Dieser ist für gewöhnlich sehr breit, sodass der Piezoeffekt auch bis nahe 0 Kelvin auftritt, sich hier jedoch auf bis zu 20% des Raumtemperaturwertes reduziert (nach Physik Instrumente, 2009, S. 2-204).
Die Aktoren der PICMA Serie der Firma Physik Instrumente sind sehr temperaturstabil und können bis ca. 150 °C betrieben werden. Da die zum Einsatz kommenden Aktoren dieser Serie angehören und zunächst kein Einsatz unter extremen Temperaturbedingungen geplant ist, werden Temperatureffekte vernachlässigt.

3. Der Piezomotor

Der erste Prototyp des Piezomotors beruht auf dem Prinzip des Wurmantriebs (vgl. Abschnitt 1.2.1). Anders als der klassische Inchworm-Motor der Firma Burleigh (vgl. (May, 1975)), der für den Vorschub nur ein Piezoelement vorsieht, kommen bei diesem Motor zwei Vorschubelemente zum Einsatz, die kaskadiert angeordnet sind. Dies soll den kurzen Einbruch der Geschwindigkeit beim Umgreifen minimieren.

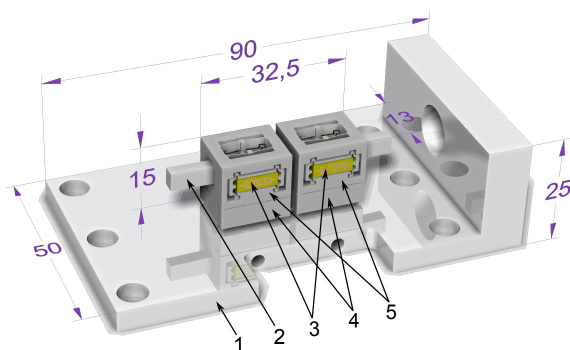


Abbildung 3.1.: CAD-Zeichnung des Piezomotors mit Bemaßung in Millimeter: Messvorrichtung (1), Läufer (2), Piezostacks (3), Vorschubeinheit (4), Klemmeinheit (5)

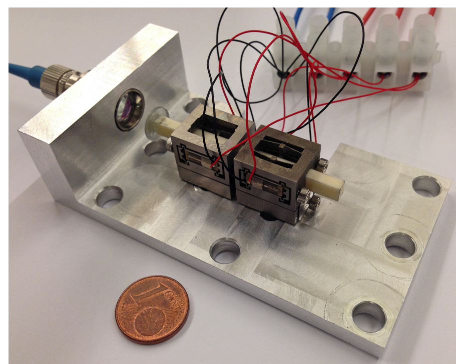


Abbildung 3.2.: Realer Piezomotor mit auf dem Läufer aufgeklebten Spiegel für den eingeschraubten Interferometermesskopf

Wie den Abbildung 3.1 und 3.2 entnommen werden kann, besteht ein Motor aus zwei Blöcken (hier auf einer Basisplatte (1) montiert, die zudem als Halterung für den Interferometermesskopf fungiert), welche wiederum aus Klemm- und Vorschubeinheit bestehen. Die Klemmeinheit (oberes Element (5)) beinhaltet zwei Piezoaktoren (3), die parallel in zwei Festkörpergelenke links und rechts vom Läufer (2) eingesetzt und vorgespannt werden. Beim Anlegen einer Spannung an die Aktoren bewirkt deren Ausdehnung ein Verformen der Gelenke, was wiederum zu einer mittigen Ausdehnung derselben führt, wodurch diese kraftschlüssig am Läufer anliegen. Die Vorschubeinheit (unteres Element (4)) besteht aus zwei Starrkörpern, die ebenfalls durch ein Festkörpergelenk miteinander verbunden sind. Der Piezostack wird mittig in den inneren Körper eingespannt, der wiederum an der Klemmeinheit festgeschraubt wird. Der äußere Körper hingegen wird an der Basisplatte befestigt. Dehnt sich nun der Aktor aus, verformt sich das Gelenk und sorgt für eine Positionsänderung des inneren Starrkörpers. Da dieser, wie bereits erwähnt, am oberen Element montiert ist, wird folglich die gesamte Klemmeinheit um die Piezoausdehnung (1:1 Übersetzung) verschoben. Auf Abbildung 3.3 ist in Analogie zu Abbildung 1.1 der Bewegungsablauf des Motorprototyps dargestellt.

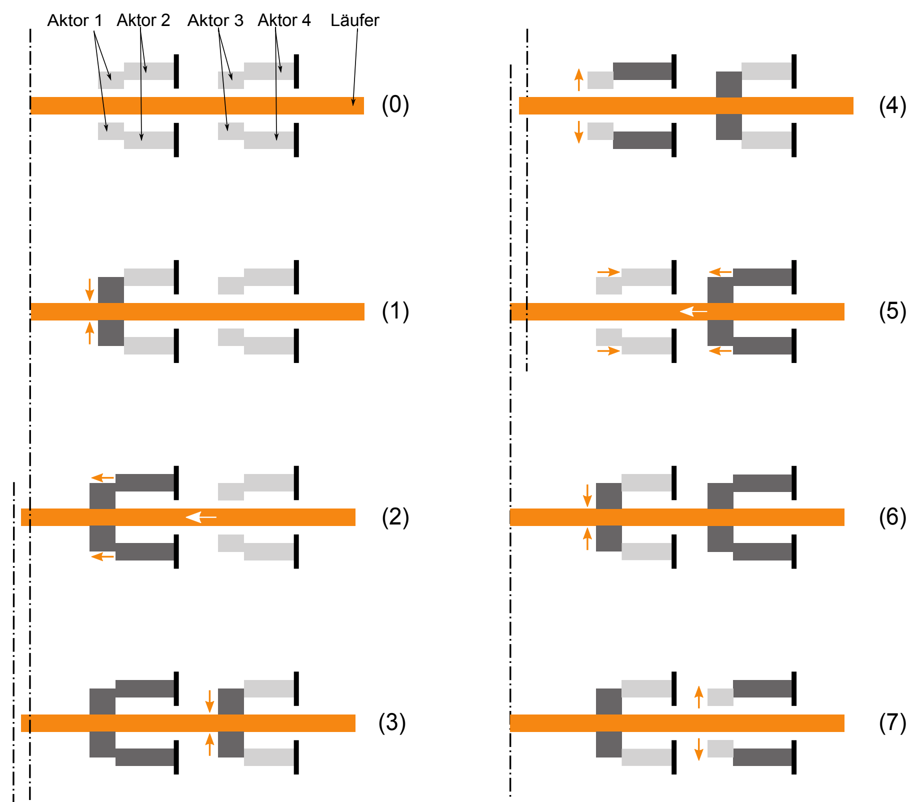


Abbildung 3.3.: Bewegungsablauf Piezomotorprototyp

1. Aktor 1 dehnt sich aus und klemmt so den Läufer ein.
2. Aktor 2 dehnt sich aus und verschiebt so den Läufer um einen halben Schritt nach links.
3. Aktor 3 dehnt sich aus und klemmt den Läufer ebenfalls ein.
4. Aktor 1 öffnet sich.
5. Aktor 2 entspannt sich, gleichzeitig dehnt sich Aktor 4 aus und verschiebt so den Läufer um einen weiteren halben Schritt nach links.
6. Aktor 1 klemmt den Läufer erneut ein.
7. Aktor 3 öffnet sich. Der Ablauf beginnt von Neuem.

4. Modellbildung

Um den realen Piezomotor aus Kapitel 3 mit einer für ihn optimalen Wellenform anzusteuern, wird zunächst mit der Modellbildung in MATLAB/Simulink begonnen. Die Modellierung der dabei wichtigsten Komponente, der Piezostack selbst (siehe 2.3), wird im folgenden Abschnitt beschrieben.
In der Literatur finden sich zahlreiche Ansätze, das Verhalten piezoelektrischer Aktoren, allen voran die Hysteresebildung, sowohl mathematisch als auch phänomenologisch nachzubilden. Die wohl bekanntesten mathematischen Beschreibungen sind hierbei die Modelle von Preisach, Duhem und Prandtl-Ishlinskii (Preisach, 1935; Duhem, 1897; Visintin, 1994). Die Herangehensweise an die Simulation in dieser Arbeit beruht auf der Idee von Gnad (Gnad, 2005). Dieser beschreibt in seiner Dissertation die Hysteresemodellierung beruhend auf der Skalierung der maximalen Hysterese (Hüllkurve).

4.1. Simulation des Piezoaktors

Die Modellbildung von Gnad sieht vor, dass sämtliche inneren Hystereseschleifen eines piezoelektrischen Aktors Skalierungen der Hüllkurve sind. Eine Messung am realen Aktor bestätigt diesen Sachverhalt: Die Schleifen 1-3-1 und 4-5-4 sind Skalierungen der Hüllkurve 1-2-1, wie Abbildung 4.1 illustriert.

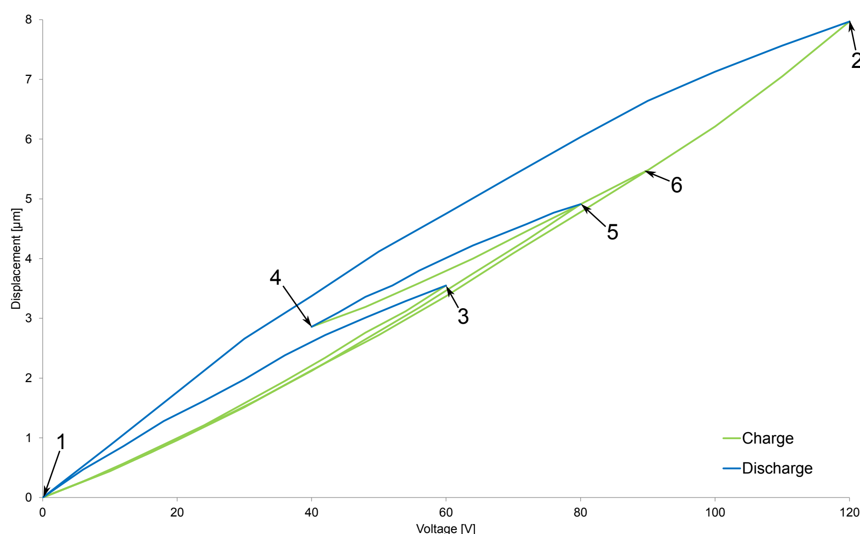


Abbildung 4.1.: Messung von Hystereseschleifen des Aktors P883.11

Weiterhin ist abzulesen, dass nach einem Richtungswechsel (von Lade- zu Entladevorgang) stets der letzte Punkt bzw. letzte Richtungswechsel angestrebt wird, sodass geschlossene Schleifen entstehen, beispielsweise wird nach dem Richtungswechsel am Punkt 2 wieder Punkt 1 oder nach dem Richtungswechsel an Punkt 5 wieder Punkt 1 angestrebt. Auch ist ein Herauslaufen von inneren Schleifen nur durch den letzten Wendepunkt möglich (4-5-6).
Mit diesem Wissen ist mit der Modellbildung begonnen worden: Für den Lade- und Entladeast ist ein Polynomansatz 3. Ordnung gewählt worden, da der Ladeast im oberen Viertel eine Sättigung erfährt und diese einen Wendepunkt bildet.

$$y = p_1 \cdot X^3 + p_2 \cdot X^2 + p_3 \cdot X + p_4 \tag{4.1}$$

Die Koeffizienten des Polynoms $p_1 - p_4$ werden aus vier Punkten auf den Lade- bzw. Entladeästen berechnet. Zwei von diesen bilden den Anfangs- und Endpunkt des Astes, die restlichen zwei werden rechnerisch ermittelt. Wie dies genau geschieht, wird im Folgenden erklärt:
Von den 12 gemessen Spannungs-Ausdehnungswertepaaren (x,y) des verwendeten Piezostacks im Abstand von 10 Volt werden 6 Wertepaare für die Modellbildung herangezogen. Diese sind der Ursprungs- und Maximalpunkt der Ausdehnung sowie jeweils zwei Wertepaare auf dem Lade- bzw. Entladeast (siehe Abbildung 4.2).

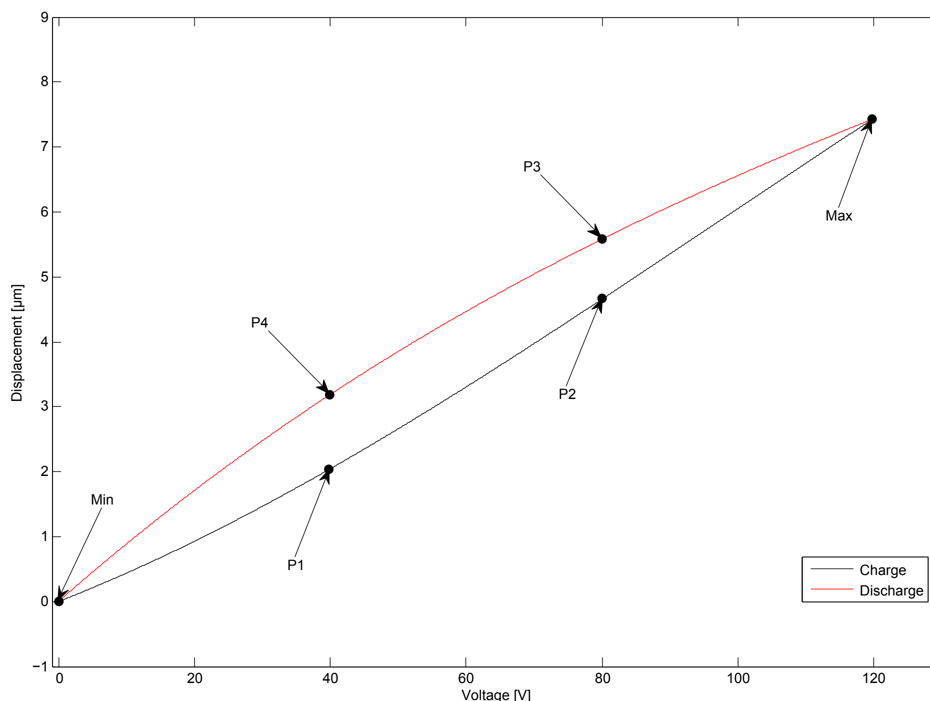


Abbildung 4.2.: Punkte für die Polynomberechnung

Wie die Skalierung der neuen Lade- und Entladeäste abläuft, wird beispielhaft an der Berechnung des Punktes $P3_{neu}$, der den oberen der zwei Punkte auf dem neuen Entladeast bildet, anhand Abbildung 4.3 erläutert.

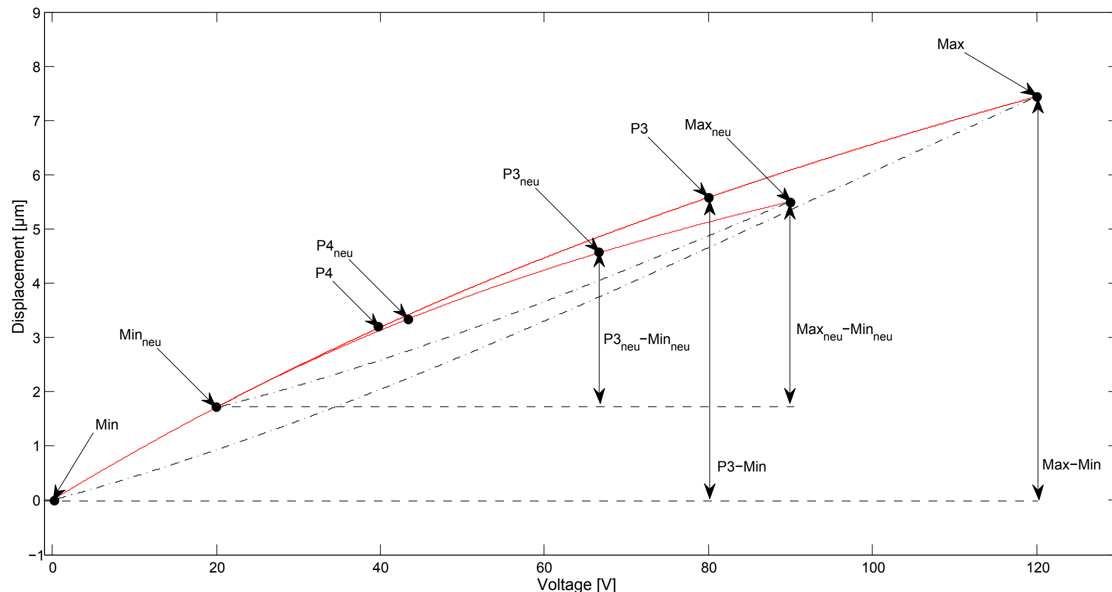


Abbildung 4.3.: Berechnung des Punktes $P3_{neu}$

Ausgehend von der Beziehung:

$$\frac{P3_{neu} - Min_{neu}}{Max_{neu} - Min_{neu}} = \frac{P3 - Min}{Max - Min} \tag{4.2}$$

kann der Punkt $P3_{neu}$ berechnet werden zu:

$$P3_{neu} = Min_{neu} + (Max_{neu} - Min_{neu}) \cdot \frac{P3 - Min}{Max - Min} \tag{4.3}$$

Analog gilt dies für Punkt $P4_{neu}$.

Die nachfolgende Simulation des Piezostacks und des Motors erfolgt in MATLAB/Simulink.

4.1.1. Hystereseberechnung

Zunächst wird das beschriebene Hystereseverhalten simuliert. Hierfür wird eine Embedded Matlab Function verwendet, die den Programmcode zur Berechnung enthält. Dieser soll im Folgenden erläutert werden:

Zu Beginn der Simulation muss für die Startspannung und Aktorausdehnung ein Initialisierungswert definiert werden, der bei Null liegt. Alle Spannungs-Ausdehnungswerte werden im Laufe der Simulation in einem Array abgelegt, welches unter anderem als Berechnungsgrundlage für die Hystereseschleifen dient. Grundsätzlich werden vier Fälle für die Berechnung der Schleifen unterschieden, wie Diagramm 4.4 zeigt.

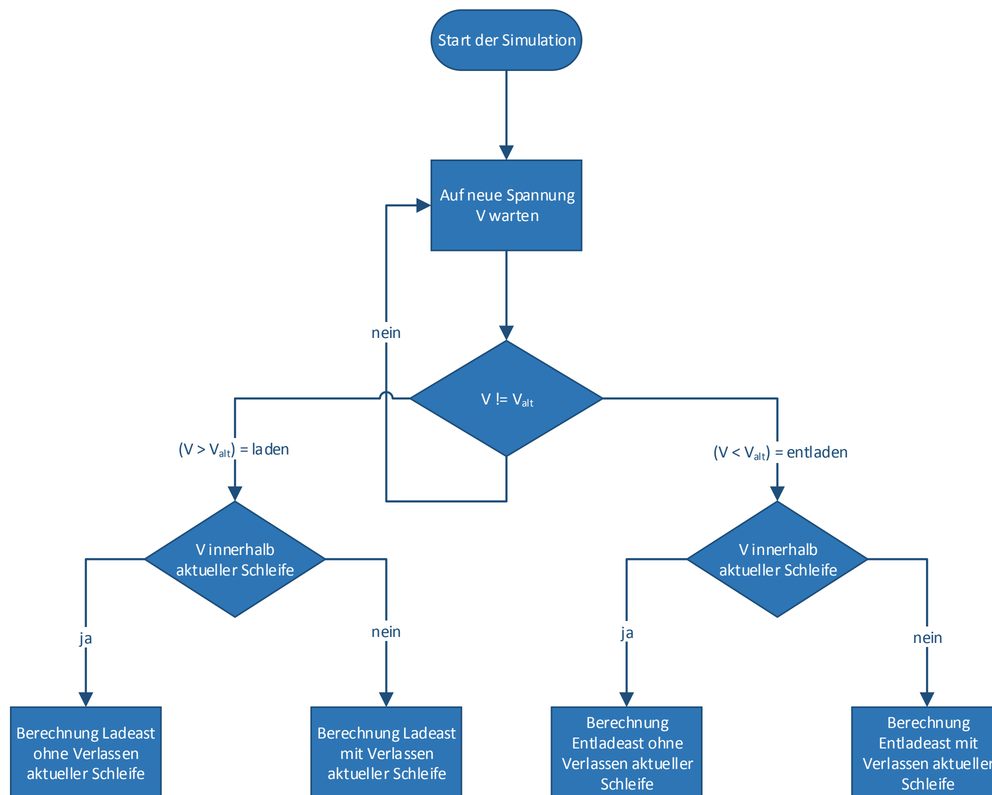


Abbildung 4.4.: Ablaufdiagramm der Hystereseberechnung in MATLAB/Simulink

Fall 1: Berechnung des Ladeasts ohne Verlassen der aktuellen Schleife
Es wird davon ausgegangen, dass der aktuelle Spannungswert größer ist als der vorherige, jedoch kleiner als der x-Wert des letzten maximalen Wendepunkts. Der neue Ladeast wird somit zwischen dem vorherigen minimalen und maximalen Wendepunkt berechnet, jedoch nur vom letzten minimalen Wendepunkt bis zum aktuellen Spannungswert gezeichnet. In Abbildung 4.5 wird dies deutlich: Der Ladeast wird zwischen den Wertepaaren $(10|0.84)$ und $(90|5.35)$ berechnet, gezeichnet allerdings nur zwischen $(10|0.84)$ und $(50|2.84)$, da Letzteres die Vorgabespannung von 50 Volt enthält (die gestrichelte Linie dient hier nur zur Verdeutlichung).

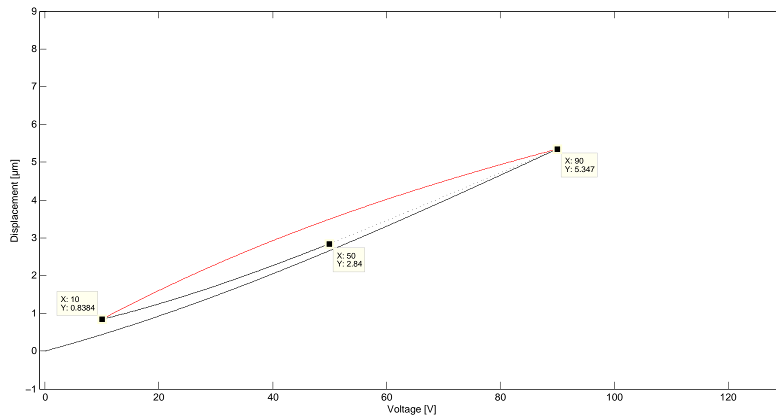


Abbildung 4.5.: Fall 1: Berechnung des Ladeasts ohne Verlassen der aktuellen Schleife

Fall 2: Berechnung des Ladeasts mit Verlassen der aktuellen Schleife

Ist der aktuelle Spannungswert größer als der vorherige und ebenfalls größer als der x-Wert des letzten maximalen Wendepunktes, so verlässt in diesem Fall der Ladeast seine aktuelle Schleife. Wie bereits in Abschnitt 4.1 erwähnt und auf Abbildung 4.1 gezeigt, wird eine Schleife nur durch den letzten Wendepunkt verlassen. Ein Schneiden der Schleife, wie im Punkt *P3* auf Abbildung 4.7 gezeigt, ist nicht möglich. Beispielhaft soll dieser Vorgang an Abbildung 4.6 erläutert werden. Nach Vorgabe der Spannungswerte 90 Volt und 10 Volt, steht mit 110 Volt der dritte Wert außerhalb der ersten Schleife. Diese wird zunächst geschlossen ($(10|0.84)$ bis $(90|5.35)$) und das Wertepaar $(10|0.84)$ verliert seine Gültigkeit. Da es keine weiteren äußeren Schleifen, ausgenommen der Hüllkurve gibt, liegt der neue Spannungswert auf letzterer Kurve. Nun darf nicht der gesamte Ladeast der Hüllkurve skaliert werden wie in Abbildung 4.8 (blauer Kreis: Für die Skalierung der Kurve zwischen *P1* und *P3* ist der gesamte Entladeast herangezogen worden, was dazu führt, dass der Nullpunkt nicht erreicht wird), sondern es wird nur das letzte Stück von $(90|5.35)$ bis $(120|7.43)$ berechnet und bis zum aktuellen Spannungs-Ausdehnungswertepaar $(110|6.74)$ gezeichnet.

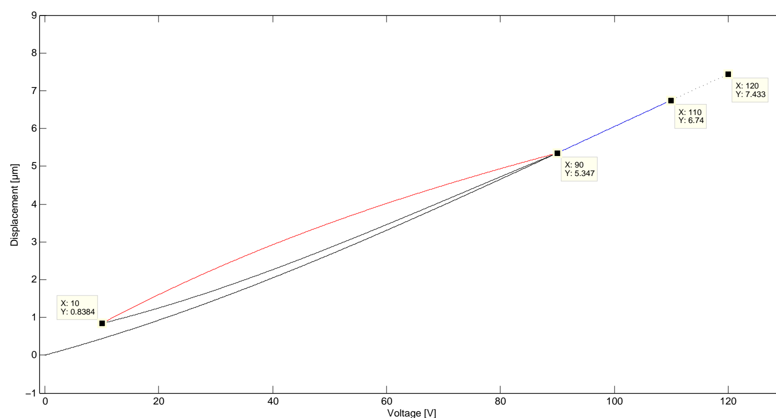


Abbildung 4.6.: Fall 2: Berechnung des Ladeasts mit Verlassen der aktuellen Schleife

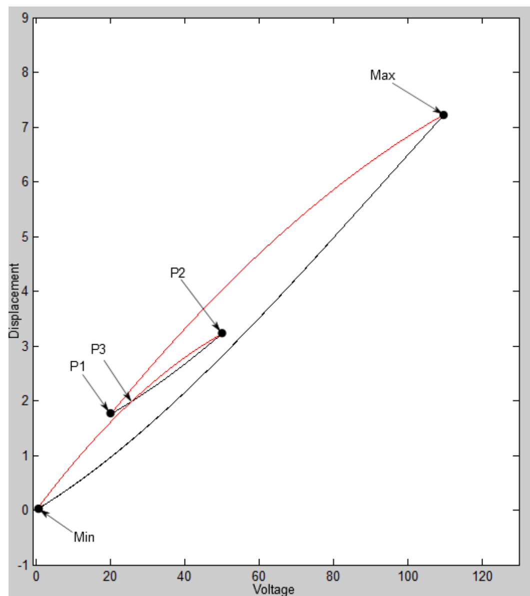


Abbildung 4.7.: Herauslaufen mit Schneiden an Punkt *P3*

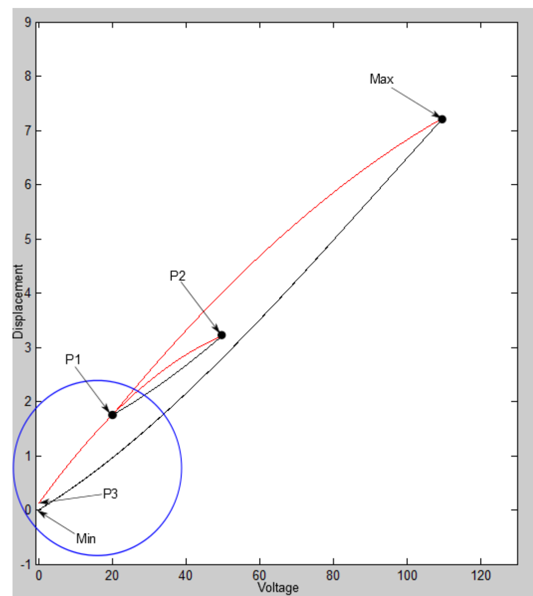


Abbildung 4.8.: Herauslaufen mit fehlerhafter Skalierung zwischen *P1* und *P3*

Auf Fall 3 (Berechnung des Entladeasts ohne Verlassen der aktuellen Schleife) und Fall 4 (Berechnung des Entladeasts mit Verlassen der aktuellen Schleife), soll an dieser Stelle, außer mit zwei Abbildungen (4.9 und 4.10), nicht weiter eingegangen werden, da sie sich von Fall 1 und 2 nur durch umgekehrte Richtung unterscheiden.

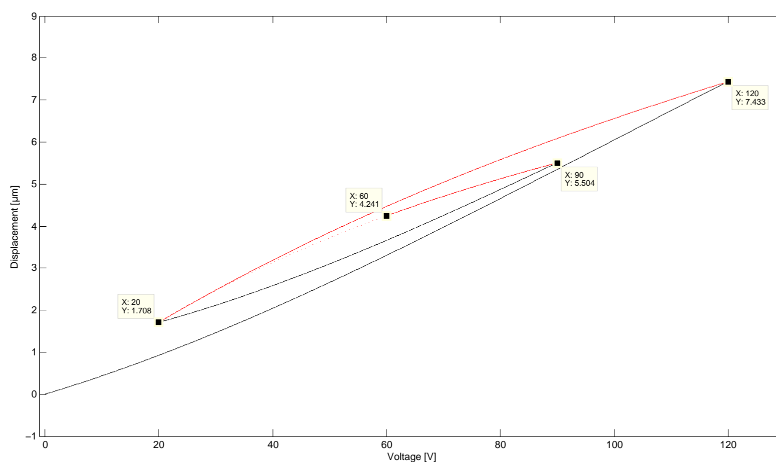


Abbildung 4.9.: Fall 3: Berechnung des Entladeasts ohne Verlassen der aktuellen Schleife

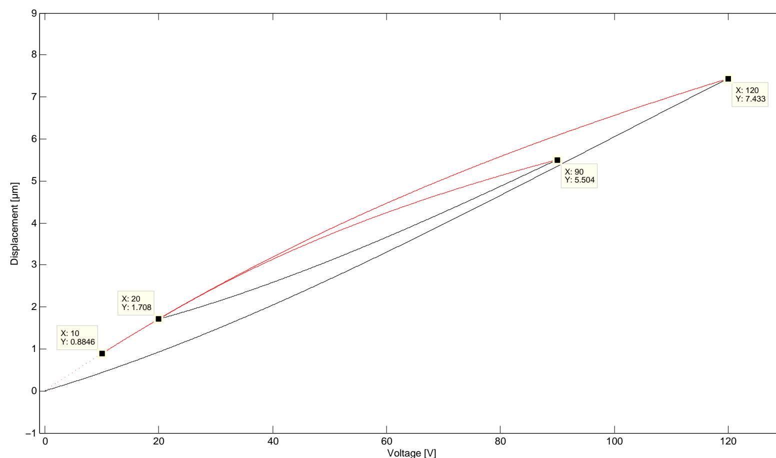


Abbildung 4.10.: Fall 4: Berechnung des Entladeasts mit Verlassen der aktuellen Schleife

4.1.2. Driftberechnung

In einem nächsten Schritt wird das Modell so erweitert, dass nun die sechs Koordinaten für die Berechnungen der Hüllkurve und skalierten Schleifen von externer Quelle bezogen werden. Dies macht das Simulationsmodell unabhängig von einer bestimmten Art Piezostacks: Es müssen jetzt lediglich 6 Wertepaare an einem beliebigen Stack gemessen und in die entsprechenden Blöcke *Loadcoords* und *Unloadcoords* eingetragen werden (vgl. Abbildung 4.11).

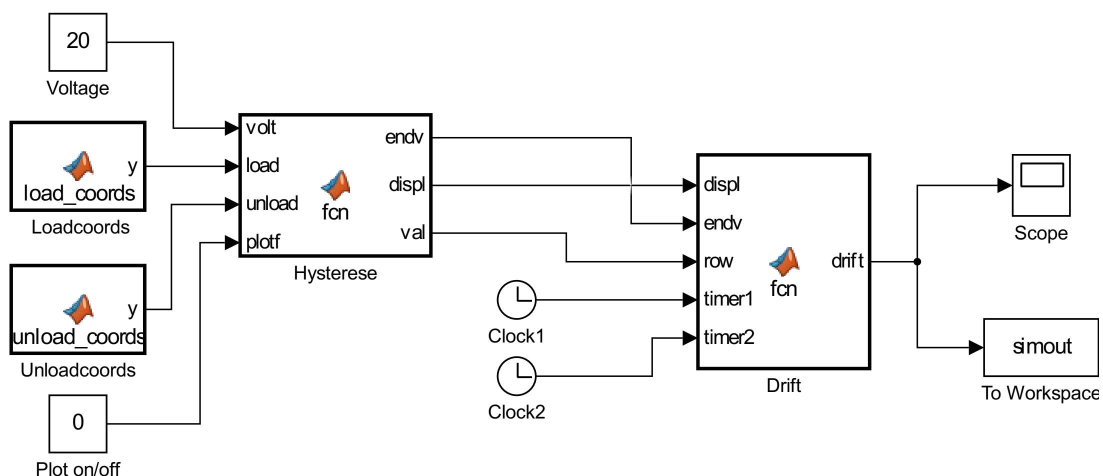


Abbildung 4.11.: Simulinkmodell Piezostack mit Hysterese- und Driftberechnung

Weiterhin wird ein Block hinzugefügt, der die Berechnung des Drifts übernimmt. Dieser basiert auf folgender Formel:

$$\Delta L(t) \approx \Delta L_{t=0.1} \left[1 + \gamma \cdot \lg \left(\frac{t}{0.1} \right) \right] \tag{4.4}$$

mit:

- t: Zeit $[s]$
- $\Delta L(t)$: Positionsänderung als Funktion der Zeit $[m]$
- $\Delta L_{t=0.1}$: Auslenkung 0.1 Sekunde nach Ende der Spannungsänderung $[m]$
- γ: Kriechfaktor, abhängig von den Materialeigenschaften

die jedoch etwas modifiziert werden muss. Zum einen sieht die Standardformel vor, dass der Drift erst zum Zeitpunkt $t = 0.1sec$ beginnt, was nicht dem realen Verhalten entspricht, zum anderen ist sie nur für positive Spannungssprünge geeignet (nach Physik Instrumente, 2009, S. 2-186).
Die Umsetzung in Simulink verläuft folgendermaßen: Dem Driftblock wird als Parameter jeweils der aktuelle Ausdehnungswert des Piezoaktors (*displ*) sowie das Array mit allen vorherigen Ausdehnungswerten übergeben (*endv*). Ändert sich nun der Ausdehnungswert, wird zunächst überprüft, ob dieser kleiner oder größer ist als der Vorherige. Je nachdem welcher Fall eintritt, wird ein Timer gestartet, der der Formel als Wert für t übergeben wird sowie bei positivem Spannungssprung die Standardformel, bei negativem Sprung eine modifizierte Version der Formel mit dem aktuellen Ausdehnungswert für ΔL ausführt. Sobald sich der Ausdehnungswert des Piezoelements erneut ändert, wird der Timer zurückgesetzt und der Ablauf beginnt von Neuem.
Um das bisherige Modell noch realitätsnäher zu gestalten, werden am realen Aktor Messreihen mit verschiedenen Signalverläufen aufgenommen, anhand derer das Simulationsmodell „getuned“ wird. Diese Messreihen stellen Spannungssprünge mit gewisser zeitlicher Verzögerung dar, um das Aktorverhalten besser verstehen zu können (Abbildung 4.12, 4.13 und 4.14).

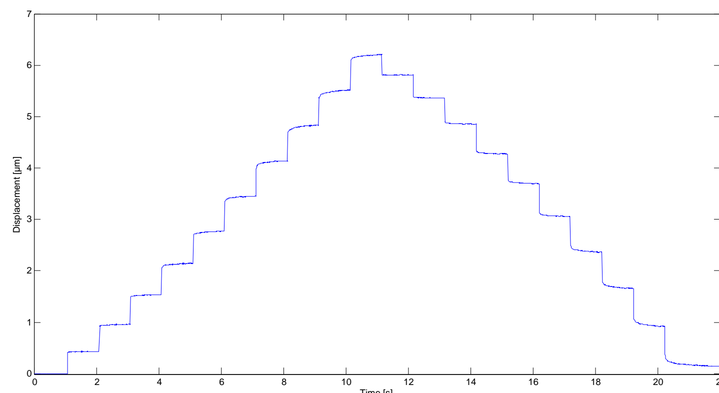


Abbildung 4.12.: Messung 1 der Piezostackausdehnung in 10 Volt Schritten

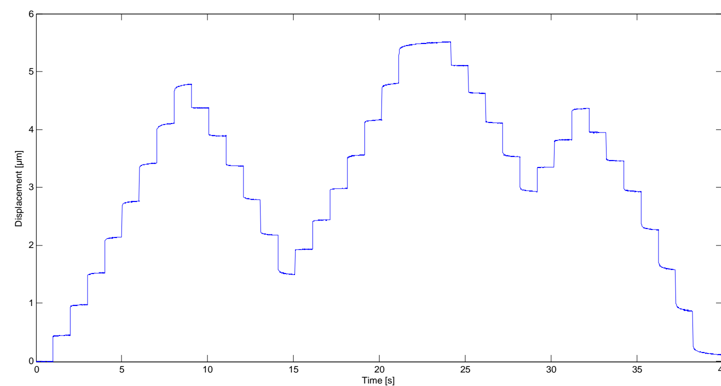


Abbildung 4.13.: Messung 2 der Piezostackausdehnung in 10 Volt Schritten

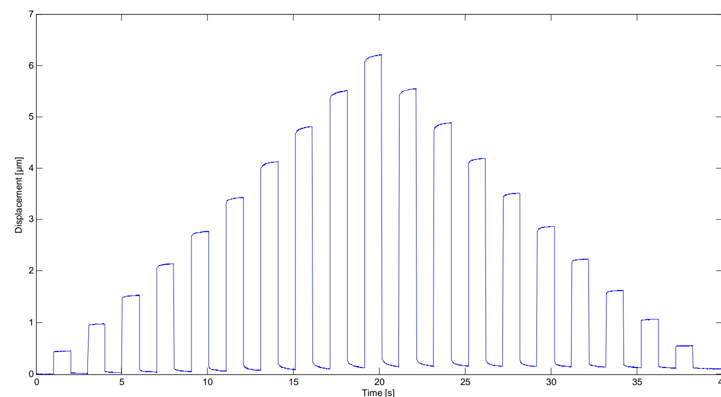


Abbildung 4.14.: Messung 3 der Piezostackausdehnung in 10 Volt Schritten

Bei diesen Messungen sind zwei interessante Beobachtungen gemacht worden, die maßgeblich für die Simulation waren:
Zum einen fällt bei Messung 1 (Abbildung 4.12) auf, dass die Entladekurve gespiegelt an Abszisse und Ordinate nahezu deckungsgleich zur Ladekurve ist (Abbildung 4.15). Dies ist wichtig für die negative Berechnung des Drifts. Würde man in der Standardformel 4.4 bei Berechnung von negativen Spannungssprüngen lediglich den $+$ - Operator durch ein $-$ ersetzen, würde zwar der Drift in die korrekte Richtung gehen, jedoch wäre die Ausprägung desselben falsch. Wie auf Abbildung 4.12 zu sehen, wird die Ausprägung des Drifts bei der Ladekurve mit steigender Ausdehnung des Piezostacks größer, bei der Entladekurve ist dies jedoch genau invertiert.
Gelöst wird diese Aufgabe, indem der Wert für ΔL vor der Berechnung neu gebildet wird: Der aktuelle Ausdehnungswert wird zunächst vom vorherigen subtrahiert, mit diesem Ergebnis die Formel ausgeführt und anschließend das Ergebnis vom vorherigen Ausdehnungswert subtrahiert.
Zum anderen ist bei Messung 3 (Abbildung 4.14) festgestellt worden, dass bei Spannungssprüngen zu 0 Volt ein Offset der Ausdehnung auftritt, welcher umso größer wird, je höher die vorherige Ausdehnung ist. Experimentell wird jeweils an den fallenden Flanken über die Punkte, an denen der Drift beginnt, eine Kurve gelegt, die ein nahezu lineares Verhalten aufweist (Abbildung 4.16).

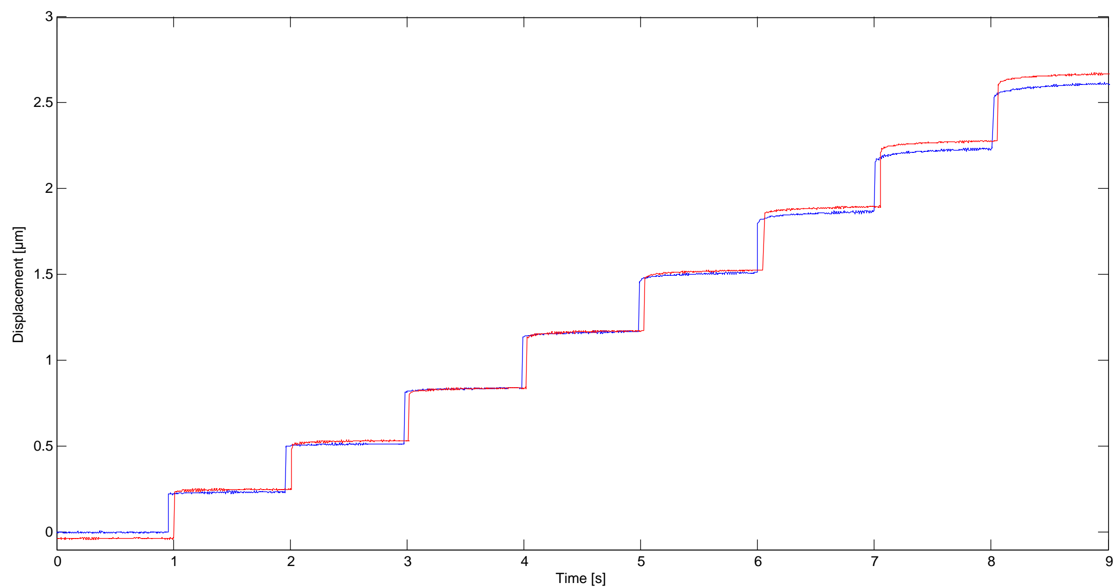


Abbildung 4.15.: Entladekurve (rot) an x- und y-Achse gespiegelt und über Ladekurve (blau) gelegt

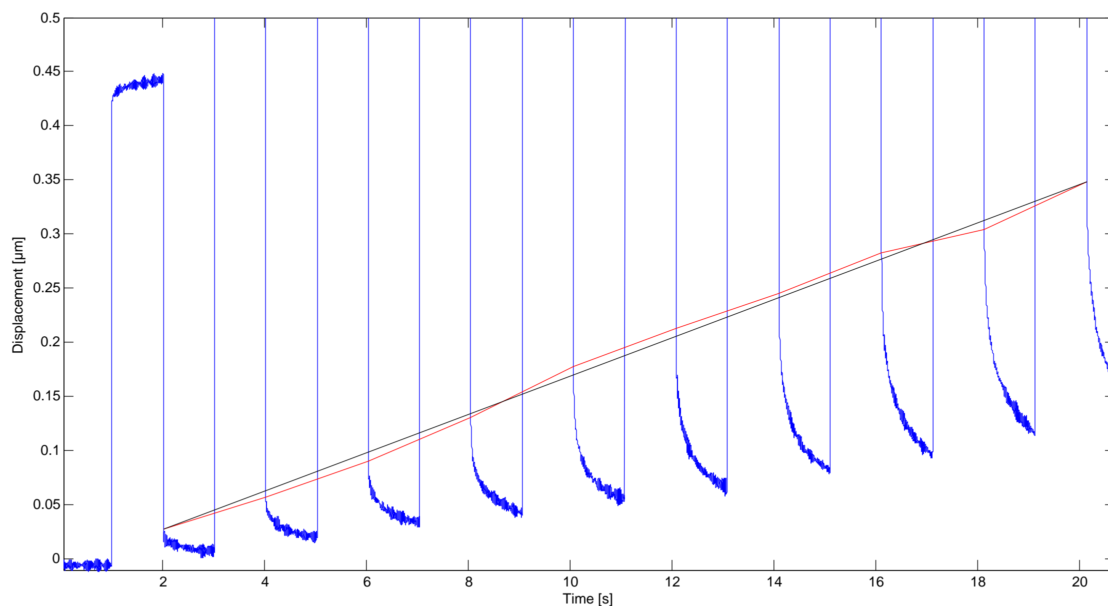


Abbildung 4.16.: Offset bei negativen Spannungssprüngen. Die rote Kurve liegt über den Startpunkten des jeweiligen Drifts, die schwarze Kurve stellt die Ausgleichsgerade dar.

Über die Geradengleichung $y = m \cdot x + b$ werden die Steigung m und der y-Achsenabschnitt b bestimmt und dem Funktionsblock der Hystereseberechnung hinzugefügt. Anschließend werden mehrere Vergleichsmessungen vorgenommen, bei denen in MATLAB/Simulink über einen Funktionsgenerator dieselben Wellenformen auf das System gegeben werden wie bei den realen Messungen. Abbildung 4.17 zeigt einen dieser Vergleiche, wobei die blaue Kurve den realen, die rote Kurve den simulierten Aktor darstellt.

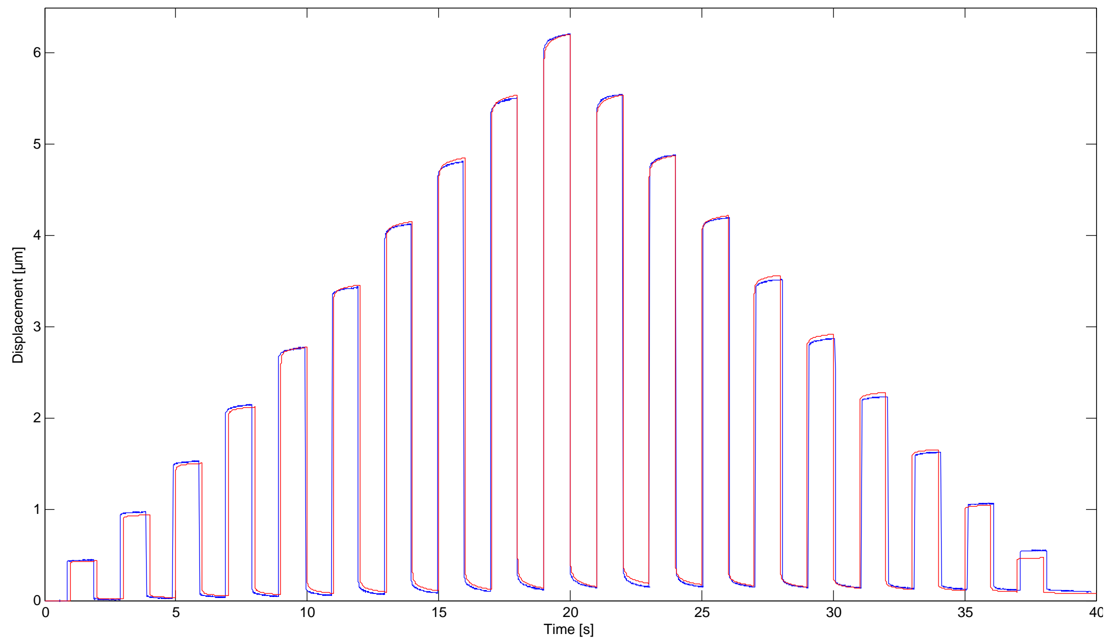


Abbildung 4.17.: Vergleich von realem (blau) und simuliertem Aktor (rot)

Da die simulierte Piezocharakteristik bis auf wenige Abweichungen sehr gut mit den Messergebnissen übereinstimmt, kann mit der Modellierung des Piezomotors begonnen werden.

4.2. Simulation des Piezomotors

Zu Beginn der Simulation des Motors sind einige Vorüberlegungen erforderlich. Zunächst wird die generelle Funktionsweise des Motors geklärt, damit die Verschaltung der Blöcke und die Programmierung der Fahrlogik korrekt durchgeführt werden können. Bekannt ist, dass es sich um einen Motor mit Wurmantrieb handelt, ebenso, dass vier (respektive sechs) Piezostacks zum Einsatz kommen (die Piezoaktoren für den Vorschub werden redundant ausgeführt, was auf die Simulation jedoch keinen Einfluss hat). Weiterhin ist die maximale Ausdehnung der verwendeten Stacks bekannt (siehe Abschnitt 2.3) sowie die gewünschte maximale Geschwindigkeit (siehe Abschnitt 1.2). Anders als beim klassischem Inchworm-Motor, der in einem Bewegungsablauf den Läufer „schiebt “ und „zieht “, wird dieser Motor den Sachverhalt durch Kaskadierung zweier reiner Vorschubelemente bewerkstelligen. Daraus lässt sich ableiten, dass es zu einem sich wiederholenden Bewegungsablauf vom Typ „klemmen, schieben“ (1. Motorelement), „klemmen, schieben“ (2. Motorelement) kommt. Grafisch ausgedrückt würde es wie auf Abbildung 4.18 aussehen, wobei der 1. und 3. Piezostack für das Klemmen und der 2. und 4. Stack für den Vorschub des Läufers verantwortlich wären.

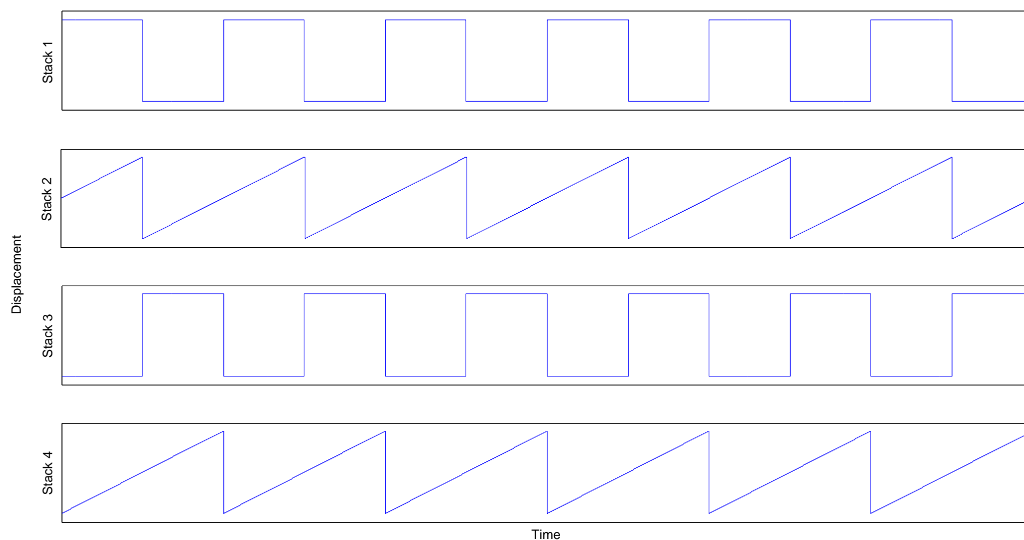


Abbildung 4.18.: Vorüberlegung Wellenform zur Ansteuerung des Motors

Diese Wellenform gilt es im weiteren Verlauf der Arbeit zu optimieren. Möglicherweise würde eine Ansteuerung mit kurvenförmiger Wellenform, ähnlich Abbildung 1.4, für einen ruhigeren Gleichlauf des Motors sorgen und gleichzeitig die Stacks mechanisch weniger stark beanspruchen. Aus diesem Grund ist es wichtig, den Schwellenwert, ab dem der Läufer von Stack 2 und 4 als gegriffen gilt, parametrierbar zu machen, da dieser von späteren, verschiedenen Wellenformen abhängt. Für die vorliegende Wellenform ist ein Defaultwert von $5\mu m$ gewählt worden, so dass der Läufer ab 100 Volt Steuerspannung definitiv als „gegriffen“ gilt.
Weiterhin hat die Fahrlogik dafür zu sorgen, dass die Ausdehnung von Stack 2 und 4, die

zunächst als unmittelbarer Vorschub des Läufers angenommen wird (ideale Mechanik), nach Erreichen der maximalen Ausdehnung den Endwert als neuen Startwert für den nächsten Stack heranzieht, damit es in der Summe zu einer gleichmäßigen Positionsänderung kommt.

Mit Hilfe der Vorüberlegungen wird ein Modell des Motors in Simulink aufgebaut. Dies lässt sich grob in drei Teile zerlegen: Erzeugung der Wellenform, die vier Piezoaktoren aus Abschnitt 4.1 und die Fahrlogik (siehe Abbildung 4.19).

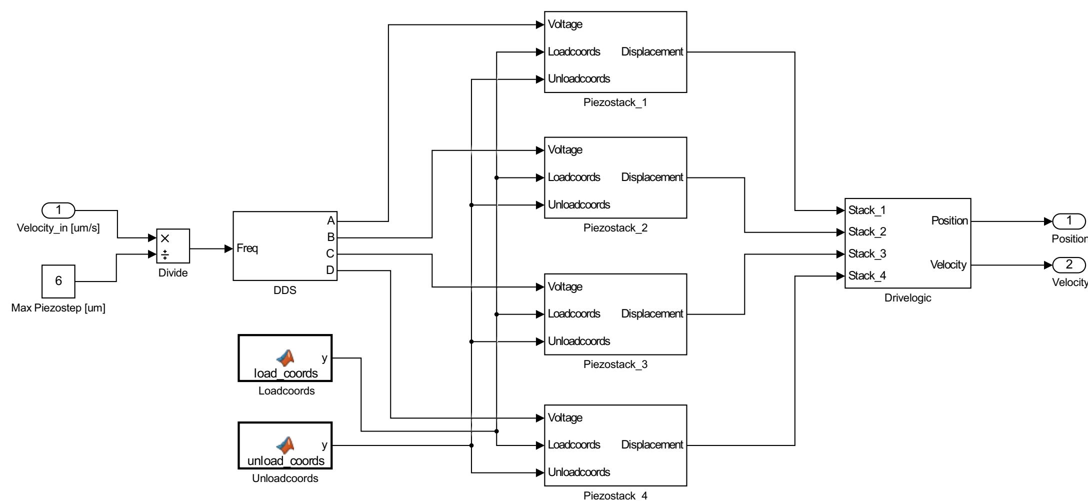


Abbildung 4.19.: Simulinkmodell des gesamten Piezomotors

Da die Piezoblöcke unverändert übernommen worden sind, wird an dieser Stelle nur noch auf den Block *Drivelogic* eingegangen. Die Signalerzeugung mittels Direct Digital Synthesis (Block *DDS*) wird in Abschnitt 5.1.2 näher erläutert.

4.2.1. Fahrlogik

Der Block *Drivelogic* übernimmt zweierlei Aufgaben: Zum einen die Berechnung des Fahralgorithmus, zum anderen die Differentiation der Positionsänderung im Hinblick auf eine spätere Geschwindigkeitsregelung (siehe Abbildung 4.20).

Linker Teil des *Drive Algorithm*-Blocks beschreibt den weiter oben erwähnten Schwellenwert, ab dem der Läufer vom Piezostack als geklemmt gilt. Ist beispielsweise die Ausdehnung von Stack 1 größer 5, würde Schalter *Switch_1* öffnen und die Ausdehnung von Stack 2 am Eingang *s1* vom Block *Drive Algorithm* anliegen. Andernfalls ist der Wert an *s1* Null.
Die Berechnung der fortlaufenden Positionsänderung übernimmt der Block *Drive Algorithm*. Wie weiter oben erwähnt, muss jeweils der letzte Ausdehnungswert gespeichert werden, damit dieser anschließend, beim Wechsel zwischen den Piezostacks, als neuer

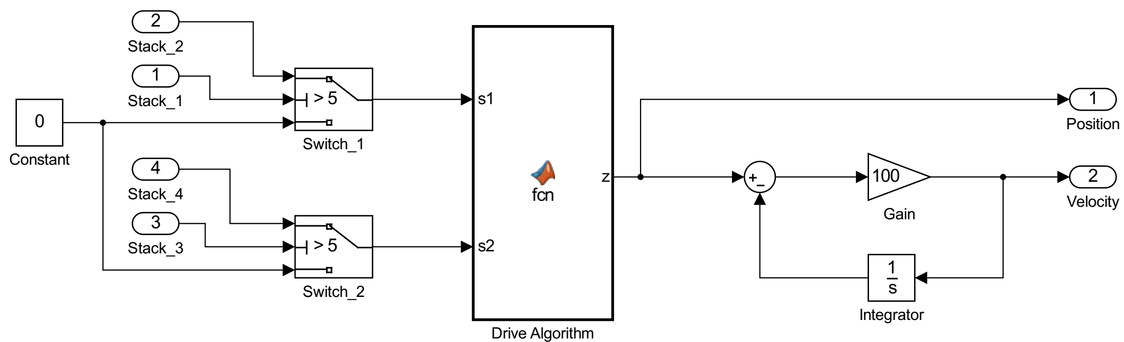


Abbildung 4.20.: Simulinkmodell der Fahrlogik

„Startwert“ für den nächsten Stack herangezogen werden kann. Abbildung 4.21 und 4.22 verdeutlichen sowohl den falschen als auch der korrekten Fall der Berechnung.

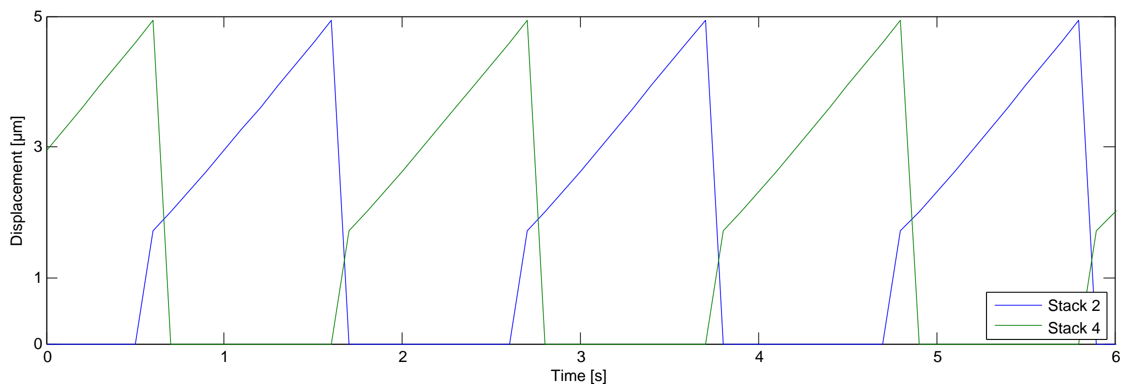


Abbildung 4.21.: Fehlerhafter Algorithmus

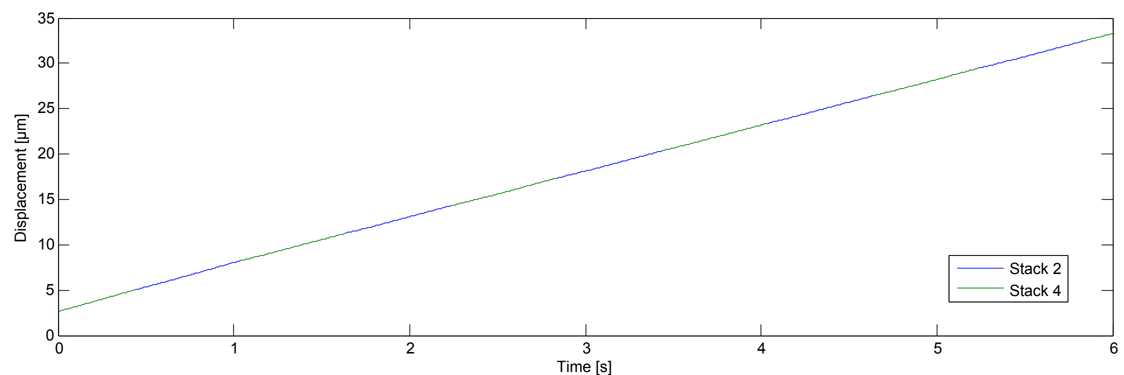


Abbildung 4.22.: Korrekter Algorithmus

Den letzten Teil der Fahrlogik bildet die Differentiation der Positionsänderung. Diese ist von Nöten, um zusätzlich zur aktuellen Position auch die Geschwindigkeit als Ausgabewert zu haben. Für das Bilden der Ableitung gibt es in MATLAB/Simulink mehrere Wege. Standardmäßig wird auf den *Derivative*-Block zurückgegriffen, der jedoch beim Verwenden des *Variable Step Solvers* sehr empfindlich reagiert und es dadurch zu unerwarteten Schwankungen im abgeleiteten Signal kommen kann. Eine Maßnahme wäre,

die Ergebnisse mit Hilfe eines Tiefpasses zu filtern, geeigneter ist es jedoch, generell auf *Derivative*-Blöcke zu verzichten und stattdessen mit Hilfe des *Integrator*-Blocks die Ableitung zu bilden, da dies dem Solver erlaubt, die Schrittweite anzupassen und somit die Genauigkeit der Simulation verbessert wird. Erreicht wird dies, indem eine Schleife modelliert wird, in deren Rückführzweig der Integrationsblock sitzt. Zusätzlich muss in den Vorwärtszweig ein Verstärkungsblock gesetzt werden, dessen Wert mit $G = 1/\tau$ (τ = Zeitkonstante) gegeben ist. Abbildung 4.23 verdeutlicht nochmal die beiden Differentiationstypen, wobei der obere Plot die Standard-, der untere die Integrationsmethode darstellt.

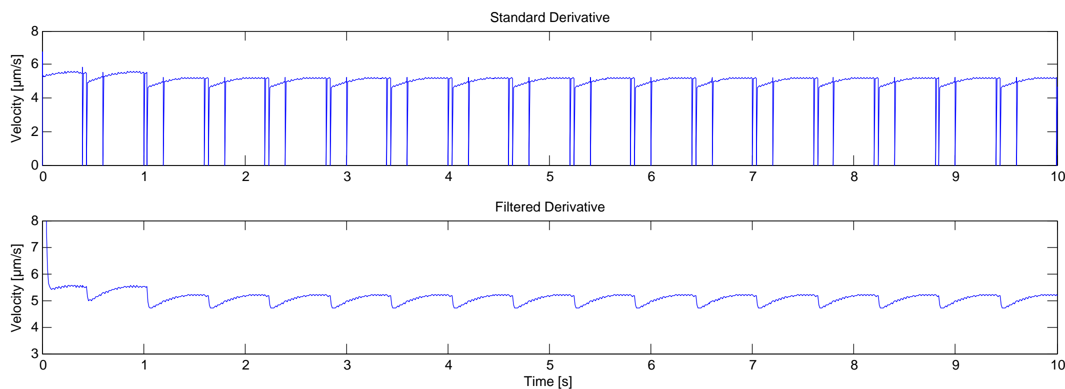


Abbildung 4.23.: Vergleich der Differentiationsmethoden

4.2.2. Inverse Steuerung

In einem nächsten Schritt wird das Motormodell um eine inverse Steuerung erweitert. Wie bereits in Abschnitt 2.4.2 erwähnt, besteht „ihre Aufgabe [...] darin, aus einem vorgegebenen Steuersignal y_{soll}, das dem gewünschten Ausgangssignal des realen Systems entspricht, ein Eingangssignal x für das reale System so zu erzeugen, dass das tatsächliche Ausgangssignal y des realen Systems mit dem vorgegebenen Steuersignal y_{soll} bis auf einen Proportionalitätsfaktor vollständig übereinstimmt“ (Janocha, 2013, S. 283).

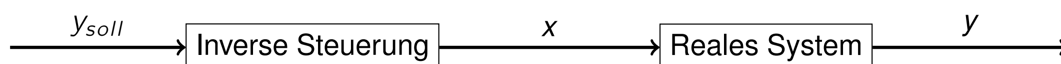


Abbildung 4.24.: Inverse Steuerung eines Übertragungssystems ($y \equiv y_{soll}$)

Für den Entwurf einer inversen Steuerung muss die Nichtlinearität des Aktors hinreichend bekannt sein, da sie in offener Wirkungskette geschieht. Dies ist durch die genaue Charakterisierung des Piezoaktors und dessen Simulationsmodell gegeben, sodass ohne weitere Vorarbeit mit der Umsetzung begonnen werden kann. Zunächst wird dafür dem Block, der für die Hystereseberechnung verantwortlich ist, ein weiterer Block

gleicher Art vorgeschaltet, dessen Ausgangssignal jedoch vom Sollsignal subtrahiert wird. Mit diesem, an der Winkelhalbierenden gespiegelten Signal wird schließlich der ursprüngliche Hystereseblock gespeist, dessen Ausgangssignal nun wie gewünscht mit dem Sollsignal übereinstimmt. Abbildung 4.25 veranschaulicht diesen Vorgang, wobei das grüne Signal das ungeregelte, das rote das invertierte und das blaue schließlich das gewünschte Ausgangssignal darstellt, was zugleich identisch mit dem Sollsignal ist.

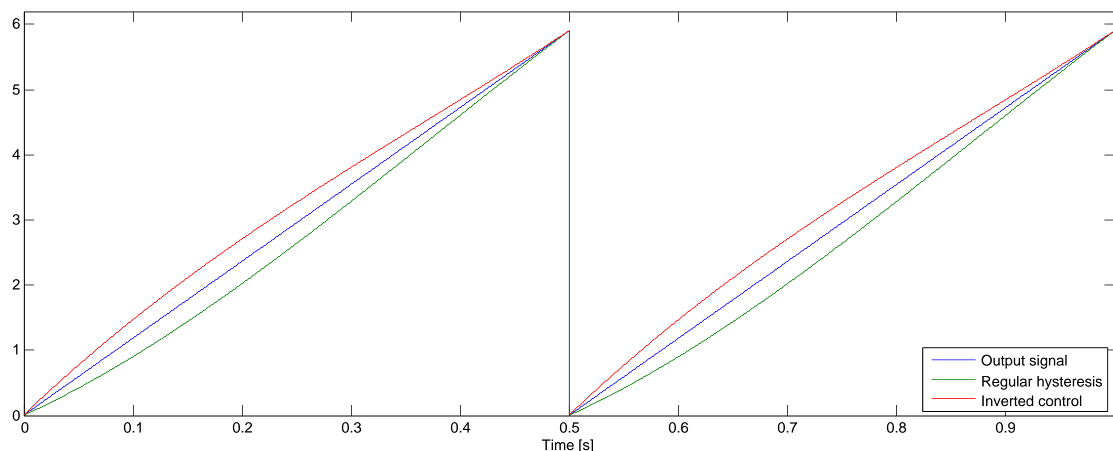


Abbildung 4.25.: Inverse Steuerung

Dass die inverse Steuerung zufriedenstellend funktioniert, kann schließlich mit Simulationsplot 4.26 bewiesen werden, der die Positionsänderung bei einer Sollgeschwindigkeit von $5\mu m/s$ darstellt: Nach 10 Sekunden Simulationszeit liegt die erreichte Position des Motors mit invertierter Steuerung bei erwarteten $50\mu m$, wohingegen die Position des ungeregelten Motors eine Abweichung von ca.10% aufweist.

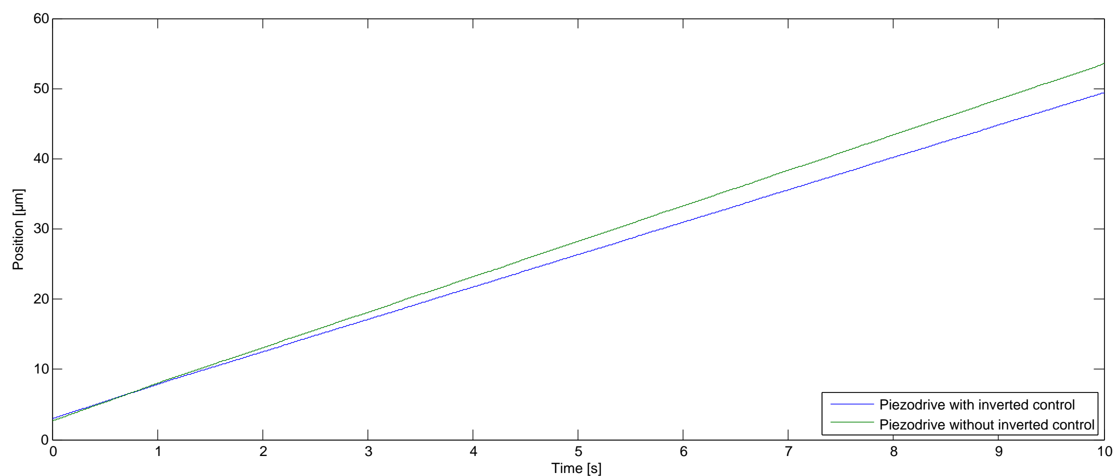


Abbildung 4.26.: Simulationsvergleich der Positionsänderung vom Motor mit und ohne inverser Steuerung

Da das Motormodell als ideal simuliert worden ist, also ohne Rücksicht auf Mechanik, Resonanzfrequenzen etc., würde es nicht ausreichen, das Ansteuersignal gemäß dem

Hystereseverhalten der einzelnen Piezostacks zu invertieren und auf den realen Motor anzuwenden. Hierfür müssen zunächst möglichst genaue Messungen am Motor vorgenommen werden, die die Nichtlinearitäten der Piezoaktoren in Kombination mit dem Motorgehäuse aufzeigen, um diese anschließend am Sollsignal zu spiegeln und als neues Steuersignal festzulegen.

4.2.3. Optimierung der Wellenform

Als letzter Schritt der Motorsimulation wird die Wellenform dahingehend optimiert, dass die Geschwindigkeitseinbrüche beim Umgreifen zwischen erstem und zweitem Motorelement minimiert werden um so eine hohe Gleichlaufruhe zu erzielen. Wie bereits zu Beginn von Kapitel 4.2 gezeigt, sieht die einfachste Wellenform für die Klemmaktoren 1 und 3 ein Rechteck-, für die Vorschubelemente 2 und 4 ein Sägezahnsignal vor. Durch Verschieben der Phasenlage der Signale zueinander ist schließlich eine vorläufige Wellenform festgelegt worden, deren Ergebnisse hinsichtlich der Position und Geschwindigkeit zufriedenstellend sind. Auf Abbildung 4.27, die diese Wellenform darstellt, ist eine Periode T markiert, um zu verdeutlichen, wie sich die Phasenlage der Signale im Gegensatz zur ursprünglichen Wellenform (Abbildung 4.18) geändert hat. Gleichzeitig steigende und fallende Flanken werden vermieden, da besonders diese durch endliche Flankensteilheit zu Geschwindigkeitseinbrüchen führen. Die Sägezähne 2 und 4 werden jeweils um 180 Grad zueinander verschoben, die Rechtecksignale 1 und 3 zu den Sägezähnen jeweils um 120 Grad. Die Pulsbreite der Rechtecksignale wird zudem auf 53% der Periodendauer T festgelegt, um zu gewährleisten, dass es beim Wechsel des Klemmvorgangs nicht zu Unterbrechungen kommt.

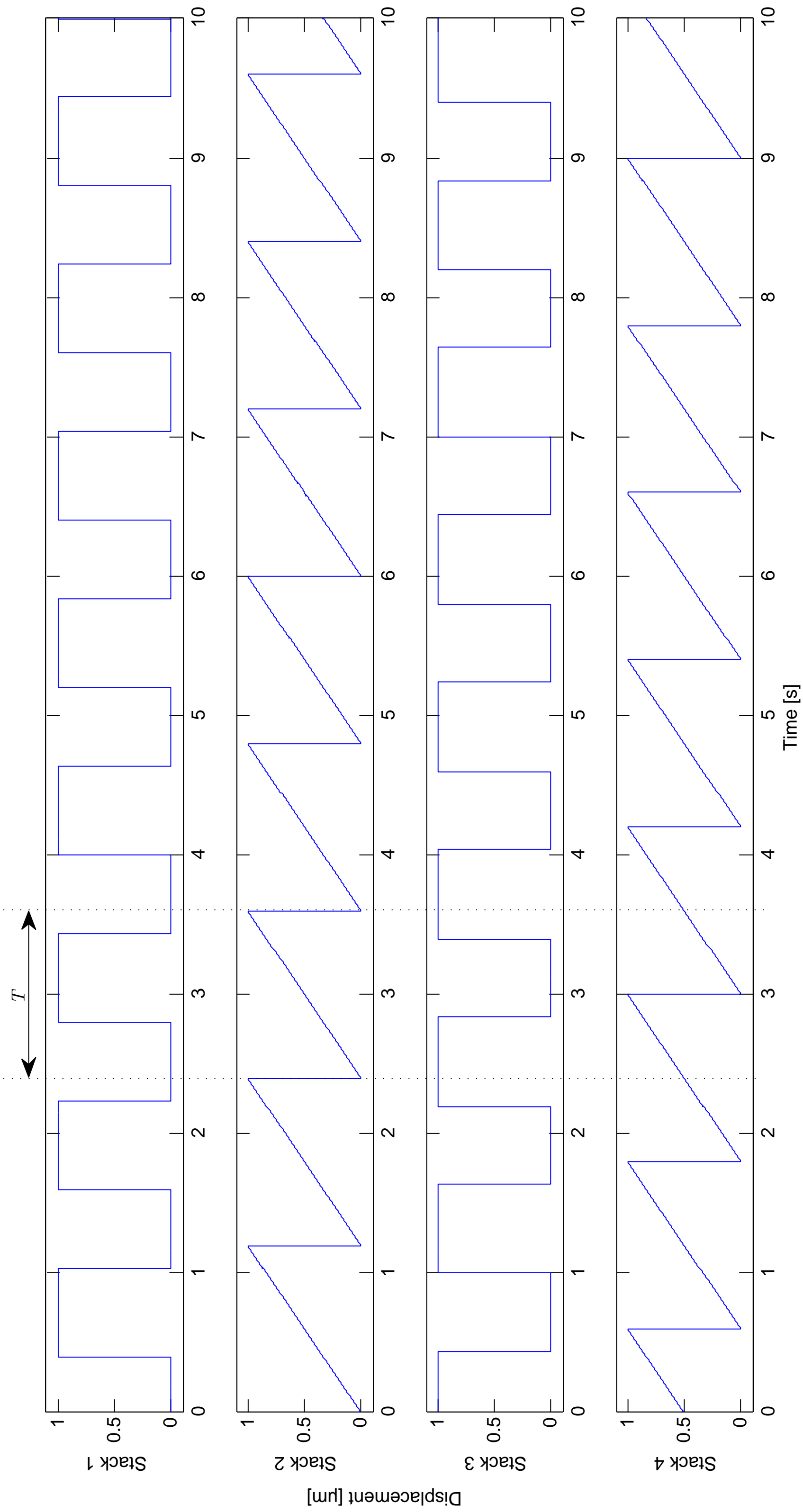


Abbildung 4.27.: Optimierte Wellenform zur Ansteuerung des Piezomotors

5. Realisierung der Ansteuerung

Dieses Kapitel umfasst die Umsetzung der Ansteuerung des realen Piezomotors, beginnend mit der Wellenformgenerierung mittels Mikrocontroller/FPGA und einer MOSFET-Verstärkerschaltung als erstes sowie der Signalerzeugung mittels FPGA in Kombination mit einem Analog-Piezoverstärker als zweites Konzept.

5.1. Konzept 1: Ansteuerung mit Digitalverstärker

5.1.1. PWM-Signalerzeugung mittels Mikrocontroller

In einem ersten Schritt ist ein Mikrocontroller gewählt worden, der die Wellenform erzeugen und diese mittels Pulsweitenmodulation (PWM) ausgeben soll. Hierfür kommt ein Arduino Due-Board zum Einsatz, dessen Hardware auf einem Atmel SAM3X8E ARM Cortex-M3 Prozessor basiert. Er stellt zur Zeit (Stand 2014) mit 84 MHz Taktfrequenz die leistungsstärkste Variante aller Arduino Boards dar und verfügt darüber hinaus über 12 PWM-fähige Ausgänge, deren Auflösung ab Werk 8 Bit (Werte zwischen 0 und 255) beträgt. Diese Auflösung kann auf maximal 12 Bit geändert werden, genau wie die PWM-Frequenz.
Die Programmierung des Mikrocontrollers erfolgt in einer Arduino eigenen Entwicklungsumgebung, einer plattformunabhängigen Java-Anwendung. Ein Programm muss mindestens aus den Methoden *setup()* und *loop()* bestehen. Erstere enthält typischerweise I/O-Zuweisungen, letztere den Programmcode, der fortlaufend ausgeführt wird. Soll nun ein Signal an einem der PWM-Ausgänge ausgegeben werden, so bietet es sich an, die Werte für dieses Signal in ein Look-Up-Table zu schreiben, die dann in der *loop*-Methode mittels *analogWrite*-Befehl auf den entsprechenden Ausgangspin geschrieben werden. Die Pulsweitenmodulation des Signals übernimmt dabei der Arduino Due. Da nach Tabelle 1.2 die gewünschte Auflösung eines Schrittes $2nm$ beträgt, wird nach Formel 5.1 eine PWM-Auflösung von 12 Bit gewählt. Daraus ergeben sich 4096 diskrete Werte, in die die Ausdehnung des Piezostacks unterteilt wird. Ausgehend von einer maximalen Ausdehnung desselben von $8\mu m$ (vgl. Abschnitt 2.3) ergibt sich so eine Auflösung von $1.95nm$ pro Schritt.

$$maxval = \frac{displ}{stepsize} \tag{5.1}$$

mit:

- $maxval$: Maximale Anzahl Werte, in die ein Schritt unterteilt wird
- $displ$: Ausdehnung des Piezostacks bei 100 Volt [μm]
- $stepsize$: Schrittweite des Motors [nm]

Damit ergibt sich folgendes Problem: Da die PWM-Frequenz mindestens doppelt so hoch sein muss wie das zu modulierende Signal, um später ein möglichst unverauschtes Ausgangssignal zu erhalten, kommt der Arduino Due mit den gewünschten Vorgaben an seine Grenzen. Gleichung 5.2 verdeutlicht diese Problematik: Indem man den Systemtakt durch die Auflösung teilt, bekommt man die maximale PWM-Frequenz zu:

$$\frac{84MHz}{4096} = 20.5kHz \tag{5.2}$$

Diese erfüllt zwar knapp das Nyquist-Shannon-Abtasttheorem ($f_{Abtast} \geq 2 \cdot f_{max}$), welches jedoch einen idealen Tiefpass fordert, um Aliasing, also störende Signalanteile des Trägersignals zu vermeiden. „In der Praxis gibt es keinen idealen Tiefpass, außerdem bewirken Tiefpässe mit hoher Flankensteilheit starke Phasenverzerrungen [...] und sind sehr aufwendig zu realisieren“ (León u. a., 2010, S. 234-235). Dieses Problem lässt sich mit Überabtastung umgehen, was bedeutet, dass das Ausgangssignal mit N -facher Abtastfrequenz abgetastet wird. Faktor N wird nachfolgend bestimmt.
Zunächst wird festgelegt, dass die spätere Signalfilterung mit Hilfe eines passiven Tiefpasses 1. Ordnung geschehen soll, der eine Flankensteilheit von ca. $20dB/Dekade$ aufweist und typischerweise aus einem Widerstand R und einem Kondensator C besteht. Die Grenzfrequenz von $10kHz$ an der $-3dB$-Grenze bestimmt R und C nach der Gleichung:

$$RC = \frac{1}{2 \cdot \pi \cdot f_g} \tag{5.3}$$

zu

$$R = 1.6kOhm$$
$$C = 0.01\mu F$$

Nach (Palacherla, 1997) wird nun bestimmt, um wieviel Dezibel das PWM-Signal bei $20kHz$ gedämpft wird:

$$\begin{aligned} (dB)20kHz &= -10 \cdot \log\left[1 + (2 \cdot \pi \cdot f_{PWM} \cdot R \cdot C)^2\right] \\ (dB)20kHz &\approx -7dB \end{aligned} \tag{5.4}$$

Diese Dämpfung ist mit einem Tiefpass 1. Ordnung nicht realisierbar, was sich mit der vorhergehenden Überlegung deckt. Um die benötigte PWM-Frequenz bei einer Dämpfung von ca. $-20dB$ zu bekommen, wird Formel 5.4 wie folgt umgestellt:

$$f_{PWM} = \frac{\sqrt{10^{\frac{-20dB}{-10}} - 1}}{2 \cdot \pi \cdot R \cdot C}$$
$$f_{PWM} \approx 100kHz \tag{5.5}$$

Hieraus ergibt sich für N ein Wert von 10, die PWM-Frequenz muss also um Faktor 10 höher sein als die Frequenz des Ausgangssignals. Weiterhin muss jetzt die benötigte Taktfrequenz des Mikrocontrollers berechnet werden. Da mit Formel 5.1 eine Auflösung von 12 Bit festgelegt ist, muss die maximale PWM-Frequenz mit 4096 multipliziert werden, woraus sich ein Systemtakt von mindestens $410MHz$ ergibt. Spätestens jetzt wird deutlich, dass dieses Vorhaben nicht mit dem Arduino Due in Kombination mit einem Tiefpass 1. Ordnung realisiert werden kann.
Aus diesem Grund wird an dieser Stelle ein FPGA eingeführt, dessen Parallelität hohe Verarbeitungsgeschwindigkeiten erlaubt und daher für die Anforderungen geeigneter ist. Die Umsetzung wird im folgenden Abschnitt beschrieben.

5.1.2. PWM-Signalerzeugung mittels FPGA

Ein Field-Programmable-Gate-Array (FPGA) ist ein rekonfigurierbarer Logikbaustein. „Die wiederprogrammierbaren Siliziumchips sind genauso flexibel wie Software, die auf einem Prozessor ausgeführt wird. Ihre Leistungsfähigkeit wird jedoch nicht von der Anzahl der verfügbaren Prozessorkerne eingeschränkt. Im Unterschied zu Prozessoren bieten FPGAs echte Parallelität, sodass verschiedene Verarbeitungsoperationen nicht auf die gleiche Ressource angewiesen sind. Jeder einzelne Verarbeitungs-Task wird einem dedizierten Bereich auf dem Chip zugewiesen und kann so autonom und ohne Beeinflussung anderer Logikblöcke ausgeführt werden. Daher wird die Leistungsfähigkeit der Anwendung nicht eingeschränkt, wenn weitere Verarbeitungs-Tasks hinzugefügt werden“ (National Instruments, 2013).
Bei dem hier verwendeten FPGA handelt es sich um den Cyclone IV der Firma Altera mit 22320 Logikelementen. Dieser ist auf einem Entwicklungsboard, dem DE0-Nano-Board der Firma Terasic, untergebracht und kann über eine USB-Schnittstelle programmiert werden. Weitere Spezifikationen des Boards sind dem Datenblatt (Terasic Technologies Inc., 2012) zu entnehmen.
Die Konfiguration des FPGAs geschieht mit Hilfe der Hardwarebeschreibungssprache VHDL (Very High Speed Integrated Circuit Hardware Description Language) in der Programmierumgebung Quartus II Subscription Edition Software der Firma Altera, die ebenfalls eine grafische Konfiguration mit Hilfe von Funktionsblöcken erlaubt.
Das folgende Programm besteht aus sieben Funktionsblöcken, welche den VHDL-Quellcode enthalten und wie folgt verschaltet sind (Abbildung 5.1):

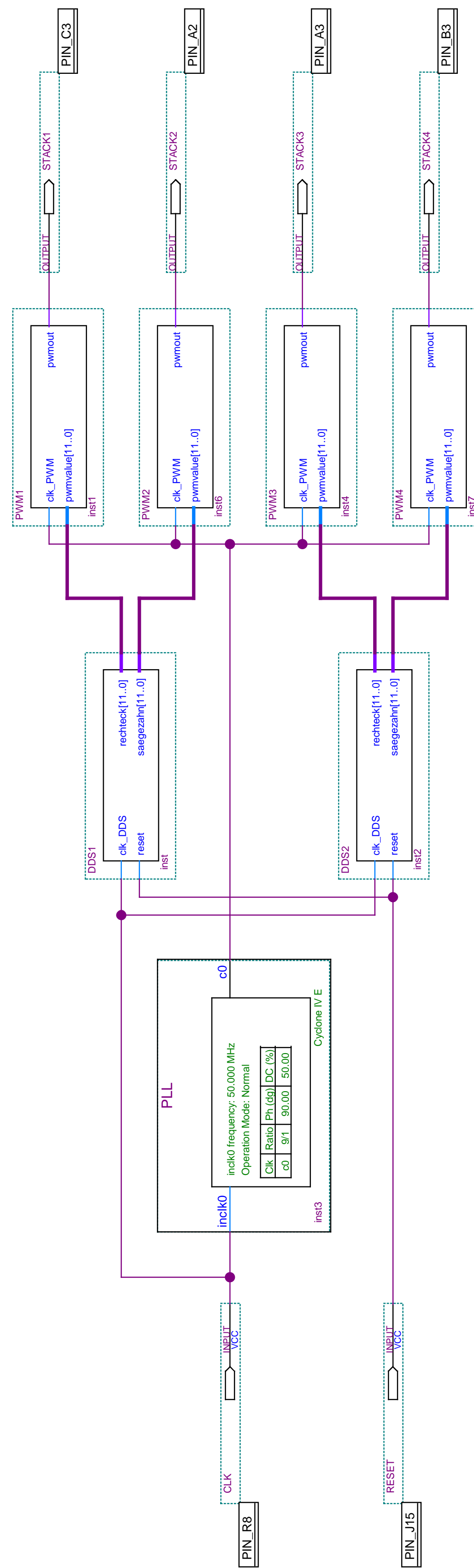


Abbildung 5.1.: VHDL-Beschreibung zur Wellenformgenerierung mittels PWM

PLL-Block

In der Digitaltechnik wird üblicherweise zur Vervielfältigung oder Teilung der Referenzfrequenz eine Phasenregelschleife (PLL) verwendet. Quartus II bietet schon fertige PLL-Blöcke an, mit deren Hilfe man komfortabel gewünschte Taktfrequenzen erzeugen kann. Wie aus Abschnitt 5.1.1 bekannt ist, muss die PWM-Frequenz mindestens $410MHz$ betragen. Da der FPGA auf dem DE0-Nano-Board mit $50MHz$ Grundfrequenz getaktet ist, wird in der PLL ein Multiplikator von 9 gewählt, sodass für die PWM $450MHz$ zur Verfügung stehen.

DDS-Blöcke

Die zwei DDS-Blöcke sind für die Erzeugung der Wellenformen verantwortlich. Sie machen sich dafür die Direkte Digitale Synthese (DDS) zu nutze, ein Verfahren, das bei digitalen Funktionsgeneratoren häufig zum Einsatz kommt. Im einfachsten Fall besteht die DDS aus einem Phasenregister (Akkumulator) und einem Look-Up-Table (LUT), in dem beispielsweise die Werte einer Sinusperiode gespeichert sind. Mit jedem Takt wird nun das Phasenregister um einen fest eingestellten Wert *cnt* inkrementiert. Sobald der Höchstwert des Registers erreicht ist, springt dieses wieder auf Null und der Zählvorgang beginnt erneut. Zu jedem Wert, den der Akkumulator annimmt, wird dann der zugehörige Wert aus dem Look-Up-Table an den Ausgang gegeben. Abbildung 5.2 veranschaulicht diesen Vorgang, wobei der Akkumulator als Phasenrad mit 7 Bit, also 128 Stützstellen dargestellt wird und die grauen Sektoren die abgespeicherten Sinuswerte bilden. Variable *cnt* hat hier den Wert 3, d.h. das Phasenregister wird bei jedem Takt um 3 inkrementiert und gibt den zugehörigen Sinuswert aus dem LUT aus. Das ausgangsseitige Ergebnis dieser Synthese würde dann wie mittig dargestellt aussehen. Da sich die Ausgangsfrequenz einer DDS proportional zum *cnt*-Wert verhält (vergleiche Formel 5.6), kann es bei zu großen Schrittweiten zu unsauberen Signalen kommen, da Speicherstellen im LUT übersprungen werden (siehe Abbildung 5.2 rechts). „Deshalb ist man in der Praxis bestrebt, nicht nur eine möglichst große Anzahl von Funktionswerten zu speichern, sondern auch die Wiederholungsrate des Additionsvorgangs möglichst hoch zu wählen, damit es bei hohen Frequenzen nicht zu Störungen kommt“ (Schwarz, 2003, S. 8).

$$f_{out} = cnt \cdot \frac{clk}{2^n} \tag{5.6}$$

(Analog Devices Inc., 1999) mit:

- f_{out}: Ausgangsfrequenz $[Hz]$
- cnt: Schrittweite
- clk: Systemtakt $[Hz]$
- n: Anzahl der Bits des Phasenregisters

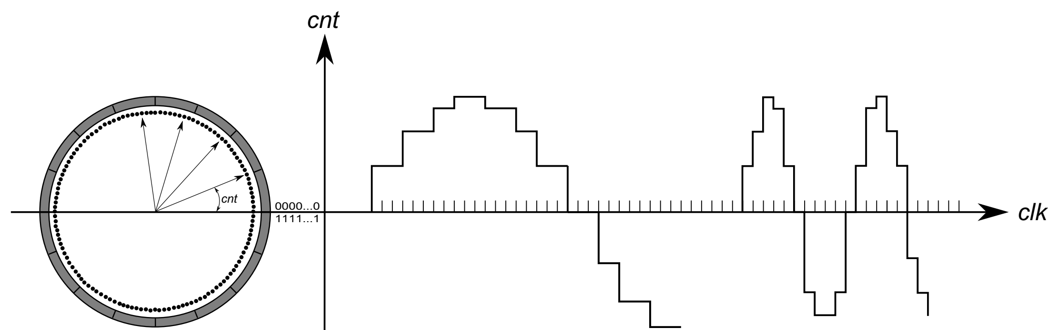


Abbildung 5.2.: Phasenakkumulator (Links), Sinus mit Schrittweite *cnt* $= 3$ (Mitte), Sinus mit Schrittweite *cnt* $= 12$ (Rechts) (nach Schwarz, 2003, S. 7)

Nicht immer ist die Generierung eines Sinussignals oder ähnlicher harmonischer Signale erwünscht, sondern, wie auch im vorliegenden Fall, ein Sägezahn-, bzw. Rechtecksignal. Dafür reicht es, den Wert, den der Phasenzähler annimmt, direkt auszugeben (Sägezahn) und für das Rechtecksignal den Ausgang ab einem bestimmten Wert des Phasenzählers auf 1 bzw. 0 zu setzen. Im ersten VHDL-Programm (Abbildung 5.1) ist diese Umsetzung für die DDS-Blöcke gewählt worden, zudem ist der *cnt*-Wert für eine Ausgangsfrequenz von $1kHz$ fix eingestellt worden. Der einzige Unterschied zwischen den Blöcken *DDS1* und *DDS2* besteht darin, dass die Wellenformen gemäß Abschnitt 4.2.3 in ihrer Phasenlage zueinander verschoben werden.

PWM-Blöcke

Die Erzeugung der PWM-Signale übernehmen die PWM-Blöcke 1-4, in dem sie das Eingangssignal (Sägezahn, Rechteck, Sinus etc.) mit einem symmetrischen Trägersignal (hier Dreieck) modulieren. Dabei zählt ein Counter abwechselnd von 0 bis 4095 (12 Bit) und zurück und vergleicht seinen aktuellen Wert mit dem Wert des Eingangssignals. Liegt dieser Wert unter dem des Counters, wird der Ausgang auf logisch 0 (LOW), andernfalls auf logisch 1 (HIGH) geschaltet. Abbildung 5.3 verdeutlicht diesen Sachverhalt: Im oberen Graphen sind das zu modulierende Sinussignal *u* sowie das Trägersignal *u** und im unteren Graphen das aus dem Vergleich resultierende PWM-Signal zu sehen.

5.1.3. MOSFET-Verstärkerschaltung

Die im vorangegangenen Abschnitt beschriebene Signalerzeugung und Modulierung mittels PWM wird nun über einen geeigneten Verstärker an die Spannungsbedürfnisse des Piezomotors angepasst. Dafür wird zunächst ein einfacher Schaltverstärker mit Hilfe eines MOSFETs entwickelt (siehe Abbildung 5.4). Für den MOSFET (*Q1*) steht ein N-Kanal High Speed Switch MOSFET der Firma NTE zur Verfügung (NTE Electronics Inc., 2014), der in der Lage ist, Spannungen bis 200 Volt zu schalten. Des Weiteren wird die Eingangsseite mittels Optokoppler (*OK1*) galvanisch vom Rest der Schaltung

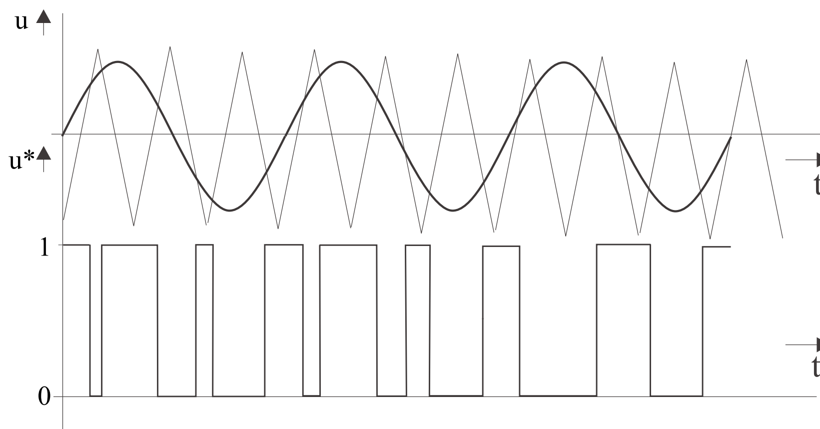


Abbildung 5.3.: PWM-Signalerzeugung durch Sinus-Dreieck-Modulation (nach Kennel, 2014, S. 4)

getrennt, um den FPGA vor Zerstörung durch zu hohe Spannungen zu schützen. Ein an $5V$-Versorgungsspannung liegender Pull-Up-Widerstand (*R1*) sorgt am MOSFET für einen definierten High-/Lowpegel, indem bei geschlossenem Transistor im Optokoppler am Gate des MOSFETs 5 Volt bzw. beim Durchschalten des Optokopplers 0 Volt am Gate anliegen. Am Drain des MOSFETs werden 120 Volt angelegt, die dann beim Durchschalten des FETs über dem Tiefpass bzw. Piezomotor anliegen. Für den Tiefpass 1. Ordnung werden die Werte für *R* und *C* aus Abschnitt 5.1.1 gewählt.

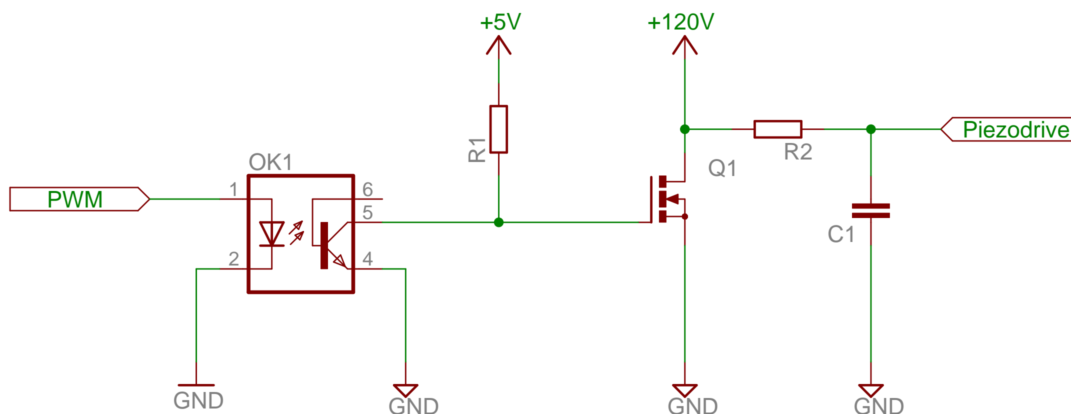


Abbildung 5.4.: Topologie des Schaltverstärkers mit Tiefpass 1. Ordnung

Abbildung 5.5 zeigt einen Plot der gefilterten Wellenform bei einer Frequenz von ca. $300Hz$. Der Grund für die niedrige Frequenz liegt in der Tatsache begründet, dass die Schaltzeiten des MOSFETs (Turn-On-Time $\approx 20ns$) deutlich über den geforderten Schaltzeiten ($\approx 2ns$ bedingt durch $450MHz$ PWM-Frequenz) liegen und somit sowohl die Grundfrequenz der Wellenform als auch die PWM-Frequenz reduziert werden muss. Die Problematik der hohen PWM-Frequenzen, der unsauberen Signalflanken und dauerhaft präsenten PWM-Jitter im Ausgangssignal, welches sich besonders negativ auf die zu Beginn der Arbeit gewünschte Gleichlaufruhe des Piezomotors auswirkt, führt zu der

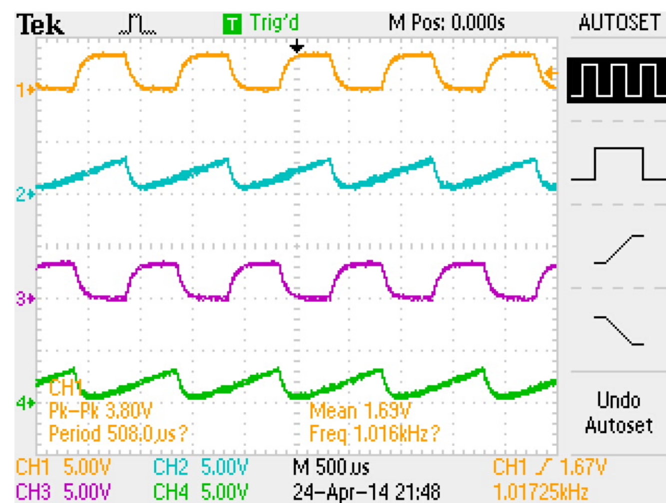


Abbildung 5.5.: Gefilterte Wellenform der Kanäle 1, 2, 3 und 4

Überlegung, die Signalverstärkung in einem zweiten Schritt rein analog aufzubauen. Die Umsetzung dieses Vorhabens wird im folgenden Abschnitt beschrieben.

5.2. Konzept 2: Ansteuerung mit Analogverstärker

Will man einen analogen Piezoverstärker entwickeln, lohnt sich zunächst ein Blick auf am Markt vorhandene Produkte und deren Funktionsweise. So bieten nicht nur Firmen wie PI, Piezosystem Jena oder Noliac Verstärker an, sondern auch Firmen wie Apex Microtechnology, die für Piezoaktoren geeignete Treiberschaltkreise im Bereich von einigen Milliampere bis zu 40 Ampere herstellen[1]. Größter Nachteil dieser Verstärker und Bauteile ist zweifelsfrei der Preis, der im Bereich von 400 Euro für einkanälige Verstärker bis zu mehreren tausend Euro für mehrkanälige Verstärker liegt. Da auch die Kosten der Komponenten ein Kriterium des Projekts sind (siehe Abschnitt 1.4), kommen diese Verstärker nicht in Betracht.
Trotzdem ist deren Funktionsweise von Interesse, auch im Hinblick auf vorhandene Schnittstellen. So bieten beispielsweise PI und Noliac reine Spannungsverstärker (analog/digital) in unterschiedlichen Leistungsbereichen an. Piezosystem Jena hat Verstärker mit digitalen Schnittstellen, wie RS-232 oder USB im Portfolio. PiezoMotor hat ebenfalls zwei Verstärker im Angebot (1- und 6-Kanal), die jedoch nur auf die hauseigenen Motoren (siehe Abschnitt 1.2.1.2) ausgelegt sind.
Da die Verwendung des FPGA aus dem vorherigen Abschnitt beibehalten werden soll, fällt die Wahl auf einen Verstärkertypen, der intern die Signalwandlung mittels Digital/Analogwandler (DAC) übernimmt und über das Serial Peripheral Interface (SPI) mit der „Außenwelt“ kommuniziert.

[1](Physik Instrumente, 2013; Piezosystem Jena, 2013; Noliac, 2014; Microtechnology, 2014)

5.2.1. Signalerzeugung mittels FPGA

Zunächst gilt es, das vorhandene VHDL-Programm dahingehend zu verändern, dass eine Kommunikation über SPI möglich ist. Dafür muss das SPI-Protokoll näher betrachtet werden:
SPI ist ein serielles Kommunikationsprotokoll, das von Motorola entwickelt wurde. Es ist ein Vollduplex-Protokoll, d.h. die Daten können gleichzeitig gesendet und empfangen werden, basierend auf einem Master-Slave-Prinzip. SPI sieht in der Standardkonfiguration vier Signale vor, zwei Steuer- und zwei Datensignale:

- Vom Master generierter Takt: SCLK
- Wahl des Slaves: SS oder CS
- Datenleitung vom Master zum Slave: MOSI
- Datenleitung vom Slave zum Master: MISO

SCLK, MOSI und MISO werden von allen Busteilnehmern verwendet. Die Datenübertragung wird durch den SPI-Takt zwischen den Teilnehmern synchronisiert, während die Signale MOSI und MISO die Daten zwischen Master und Slave übertragen. Es ist nicht zwingend notwendig, immer beide Datenleitungen zu verwenden. Hat z.B. ein Slave keinen MISO Ausgang, funktioniert die Datenübertragung auch im Simplexbetrieb. Mit der Slave-Select Leitung wird vom Master der entsprechende Slave ausgewählt. Somit ist es möglich, von einem Master mehrere Slaves anzusteuern. Für gewöhnlich liegt das SS-Signal auf HIGH und wird zu Beginn der Datenübertragung vom Master auf LOW gezogen. Abbildung 5.6 stellt eine SPI-Beispielkonfiguration dar, in der die Teilnehmer sternförmig miteinander verbunden sind (Absatz nach Miller, 2009).

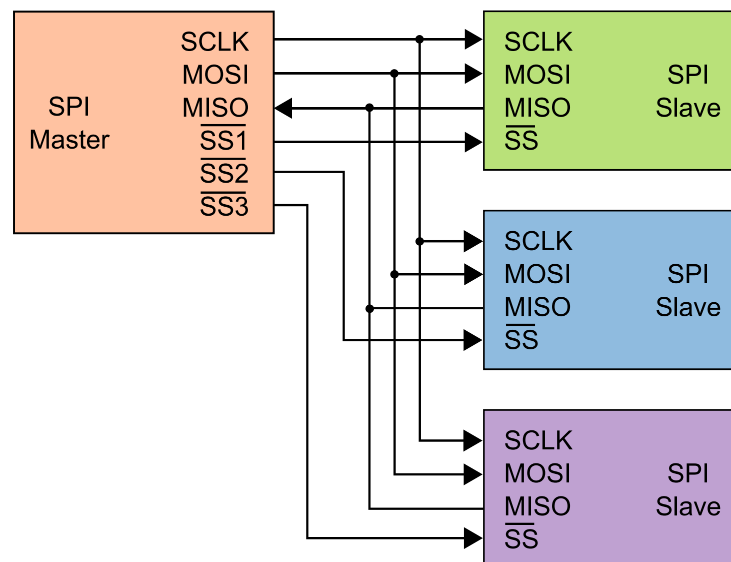
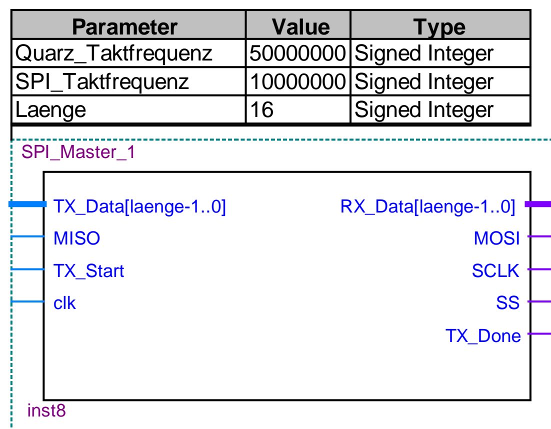


Abbildung 5.6.: Sternförmige SPI-Busstruktur (Burnett, 2006)

Da das SPI-Protokoll nie standardisiert worden ist, gibt es weder ein definiertes Übertragungsprotokoll, noch sind der Takt oder die SS-Einstellung festgelegt. Für gewöhnlich entnimmt man diese Einstellungen dem Datenblatt des zu verwendenden Bauteils (vgl. Schwerdtfeger, 2000).
Im vorliegenden Fall dient der FPGA also als SPI-Master und der DAC als Slave. Das FPGA-Programm muss dahingehend verändert werden, dass die von der DDS erzeugten Signale nicht mehr pulsweitenmoduliert, sondern einem SPI-Master zugeführt werden, welcher diese dann den Einstellungen entsprechend „versendet". Der zum Einsatz kommende VHDL-Quellcode basiert auf dem SPI-Master von Lothar Miller (vgl. Miller, 2009) und ist als VHDL-Blocksymbol auf Abbildung 5.7 dargestellt.



Parameter	Value	Type
Quarz_Taktfrequenz	50000000	Signed Integer
SPI_Taktfrequenz	10000000	Signed Integer
Laenge	16	Signed Integer

Abbildung 5.7.: VHDL Blocksymbol des SPI-Masters (nach Miller, 2009)

An Abbildung 5.8 wird nachfolgend die Funktionsweise des SPI-Masters für die Übertragung eines 16 Bit Datenpakets erläutert.

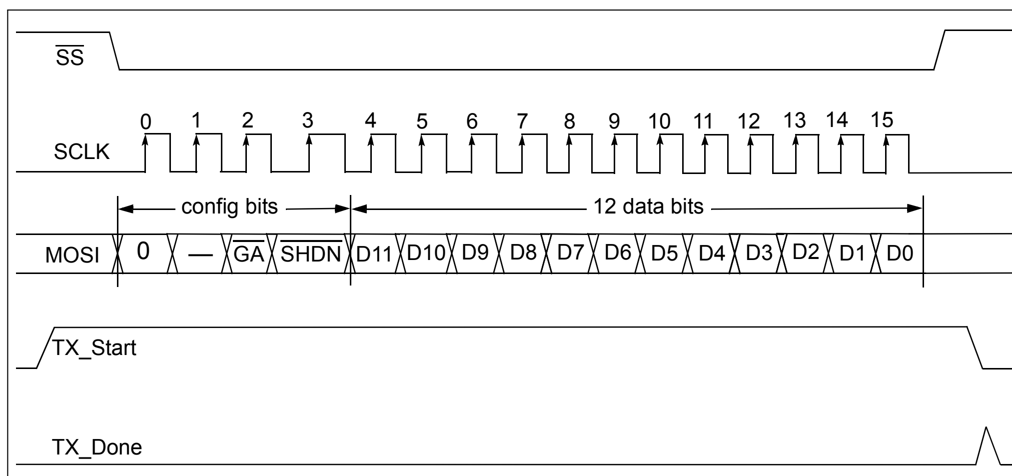


Abbildung 5.8.: SPI Übertragungsprotokoll eines 16 Bit Datenpakets (nach Microchip, 2010, S. 23)

„An den Port TX_Data werden die [...] [16] zu übertragenden Bits angelegt. Danach wird mit TX_Start die Übertragung gestartet. Erst wird SS aktiviert und dann die [...] [16]

Datenbits über das Schieberegister tx_reg an MOSI ausgegeben und gleichzeitig der Datenstrom von MISO in das Schieberegister rx_reg eingelesen. Nach der Übertragung wird SS deaktiviert, und danach TX_Done solange aktiviert, bis TX_Start inaktiv wird. Die empfangenen Daten sind jetzt am Port RX_Data abholbereit" (Miller, 2009).
Von den zur Verfügung stehenden DAC ist der MCP4821 der Firma Microchip gewählt worden, der sowohl über eine Auflösung von 12 Bit verfügt (vgl. Tabelle 1.2) als auch eine SPI-Schnittstelle aufweist (Microchip, 2010). Da dieser DAC lediglich Daten empfängt, kann auf den Eingang MOSI sowie den Ausgang RX_Data verzichtet werden. Die SPI-Frequenz des Masters wird auf $10MHz$ festgelegt (der DAC schafft maximal $20MHz$), und die Datenübertragung TX_Start mit $200kHz$ getaktet, was bedeutet, dass bis zu 50 Bit übertragen werden können. Da nur 16 Bit breite Datenpakete übertragen werden müssen, kann durch die so entstandene zeitliche Reserve von $3.4\mu s$ ebenfalls auf das Signal TX_Done verzichtet werden, da die Übertragung definitiv vor dem nächsten Start beendet ist. Aus dem Datenblatt des DAC geht auch hervor, dass zu den 12 Datenbits noch weitere 4 Bits hinzukommen, die verschiedene Betriebsmodi desselben erlauben. Daraus resultieren die bereits oben erwähnten 16 Bit (Microchip, 2010, S. 22).

Eine weitere Änderung im VHDL-Programm erfahren die DDS-Blöcke. Ist bei diesen zunächst die Ausgangsfrequenz auf $1kHz$ fix eingestellt, so ist es jetzt möglich, diese in Stufen von $0.012Hz$ zu verändern. Erreicht wird das, indem das Phasenregister von 17 auf 32 Bit erhöht wird sowie der *cnt*-Wert nun von außen einstellbar ist. Zudem wird die Möglichkeit der Richtungsumkehr und des Anhaltens auf dem aktuellen Wert der Wellenform hinzugefügt.

Da die Werte eines VHDL-Programms zur Laufzeit nicht ohne Weiteres verändert werden können, soll im Folgenden auf ein Interface eingegangen werden, das für die hier vorliegende Anwendung hinzugezogen wird.

5.2.2. Raspberry Pi Logic Controller

Der Raspberry Pi Logic Controller (PiLC) ist ein von der DESY Gruppe FS-EC entwickeltes Modul, welches in erster Linie die Kommunikation mit einem FPGA erlaubt (Spitzbart und Zink, 2013). Die Schnittstelle bildet dabei ein Raspberry Pi (RP Foundation, 2014), der via SPI mit dem ebenfalls verbauten FPGA (Terasic DE0-Nano Board) kommuniziert. Darüber hinaus bietet das Modul 16 frei belegbare I/Os, die vom Raspberry Pi gelesen und mit Daten vom FPGA beschrieben werden können. Eigene VHDL-Beschreibungen können so auf den FPGA implementiert und etwaige Ein- und Ausgänge im VHDL-Programm über den Raspberry Pi genutzt werden. Das erspart den Entwurf eigener Anschlussmöglichkeiten. Der Raspberry Pi wiederum kann bequem über Ethernet angesprochen werden und es obliegt dem Nutzer, ob er das Modul in bestehende Serverstrukturen einbindet oder, wie auch im vorliegenden Fall, das auf dem Raspberry Pi laufende C-Programm so zu modifizieren, dass die Eingabe von Werten über die Kommandozeile geschehen kann. Listing 5.1 zeigt einen Auszug des auf dem Raspberry

Pi laufenden C-Programms, in welchem die Eingabe der Werte für die Erzeugung der Wellenform auf dem FPGA abgefragt werden[2].

```
if (kbhit()){
    printf("----------------------------------------------\n\n");
    printf("Run (0 = Stop, 1 = Forward, 2 = Backward): ");
    scanf("%d",&run);
    printf("You entered:%d.\n\n",run);
    if (run != 0){
      printf("Speed [ums/s]: ");
      scanf("%d",&speed);
      printf("You entered:%d um/s.\n\n",speed);
      printf("Micrometer per Waveform [um]: ");
      scanf("%d",&umperwave);
      printf("You entered:%d um/wfm.\n\n",umperwave);
      cnt = ((speed/umperwave) * pow(2,32))/50000000;
    }
    else{
      cnt = 0;
    }
    printf("----------------------------------------------\n\n");
    char c = getch();
  }
  FPGA(I0_1_IN_Register, cnt);
  FPGA(I0_2_IN_Register, run);
```

Listing 5.1: Auszug aus dem C-Quellcode des Raspberry Pi zum Steuern des FPGAs

Nach dem Betätigen der Returntaste der Tastatur wird das Programm zur Abfrage der Werte unterbrochen. Zunächst wird in Zeile 3/4 abgefragt, ob der Motor anhalten oder nach links/rechts fahren soll (`run`). Diese Abfrage steht am Anfang, damit später der Motor sofort gestoppt werden kann, ohne andere Eingaben gemacht zu haben. Wird entweder 1 für Rechts- oder 2 für Linkslauf eingegeben, springt das Programm in die if-Anweisung und liest in Zeile 7/8 die gewünschte Geschwindigkeit (`speed`) und in Zeile 10/11 die Ausdehnung des Piezostacks ein (`umperwave`). Diese Abfrage erfolgt insofern, als dass die Ausdehnung des Stacks lastabhängig ist und somit nicht immer von der maximalen Ausdehnung des Datenblatts (vgl. Abschnitt 2.3) ausgegangen werden kann. Weiterhin kann diese Variable zum „tunen“ verwendet werden, sollte die vorgegebene Geschwindigkeit nicht mit der gemessenen übereinstimmen. In Zeile 13 wird dann der *cnt*-Wert nach Formel 5.6 berechnet, der in jeden Takt des FPGA zum Phasenakkumulator addiert wird. Schließlich werden die Werte in Zeile 21/22 an die 32 Bit Register des FPGA übergeben.

Mit Hilfe des PiLC ist es nun möglich, die SPI-Schnittstelle zum DAC komfortabel zu testen. Dabei werden nicht nur die Richtungsumkehr, sondern auch die Frequenzen bis $10kHz$ mit zufriedenstellendem Ergebnis geprüft. Da das Phasenregister auf 32 Bit erweitert worden ist, stehen nun bei allen Frequenzen bis einschließlich $10kHz$ 4096 Stützstellen zur Verfügung. Dies bedeutet, dass der Motor bei jeder Geschwindigkeit mit der höchsten Auflösung fahren kann. Das ist ein großer Vorteil gegenüber den bisher eingesetzten Motoren der Firma PiezoMotor, bei denen eine direkte Abhängigkeit zwischen Frequenz und Auflösung der Wellenform besteht: Je schneller der Motor fahren

[2] Der vollständige Quellcode ist Anhang C zu entnehmen.

soll, desto niedriger muss die Auflösung gewählt werden (Abbildung 5.9). Das führt bei hohen Frequenzen dazu, dass die Wellenform unsauber wird (Abbildung 5.10). Dieser Sachverhalt liegt vermutlich in der Ansteuerung begründet, die bei hohen Frequenzen nicht mehr alle im LUT abgelegten Werte ausgeben kann.

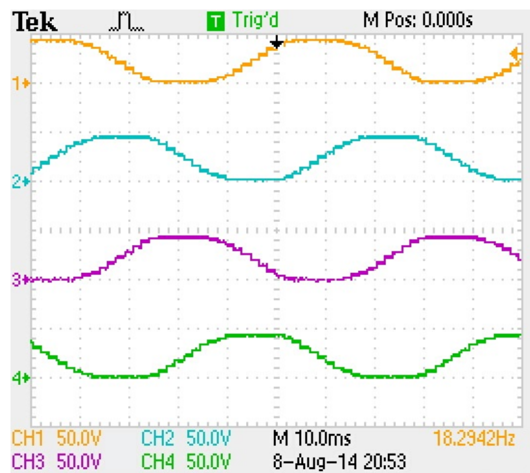


Abbildung 5.9.: Wellenform des PiezoMotor PMD90 Controller mit 5 Bit Schrittauflösung bei ca. 20 Hz

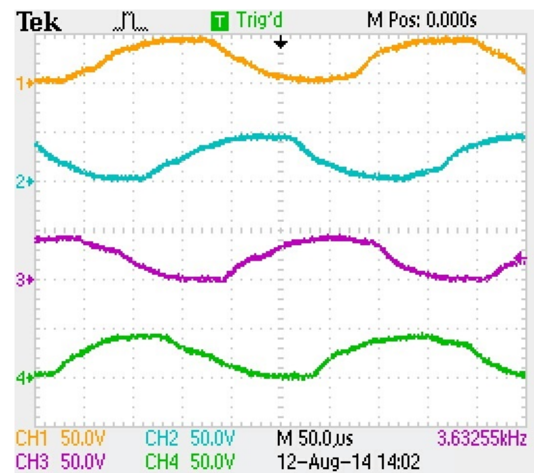


Abbildung 5.10.: Wellenform des Piezo-Motor PMD90 Controller mit 5 Bit Schrittauflösung bei ca. 3.6 kHz.

5.2.3. Analog-Verstärkerschaltung

Die folgenden zwei Abschnitte beschreiben die Entwicklung des analogen Piezoverstärkers. Der in 5.2.3.1 vorgestellte Verstärker ist für Frequenzen bis maximal $300Hz$ ausgelegt. Die zweite Version baut schließlich auf der ersten auf und erweitert diese so, dass sie den Anforderungen aus 1.2 gerecht wird.

5.2.3.1. Prototyp 1

Die Grundidee bei der ersten Verstärkerschaltung beruht auf einem nichtinvertierenden Verstärker, einer Grundschaltung des Operationsverstärkers. Diese Schaltung wird anhand Abbildung 5.11 erklärt.

Für einen idealen Operationsverstärker gelten die „drei goldenen Regeln“:

1. Die Spannungsverstärkung des Operationsverstärkers ist unendlich.
2. Es fließen nahezu keine Ströme in die beiden Eingänge.
3. Der Innenwiderstand des Ausgangs ist nahezu 0 Ohm.

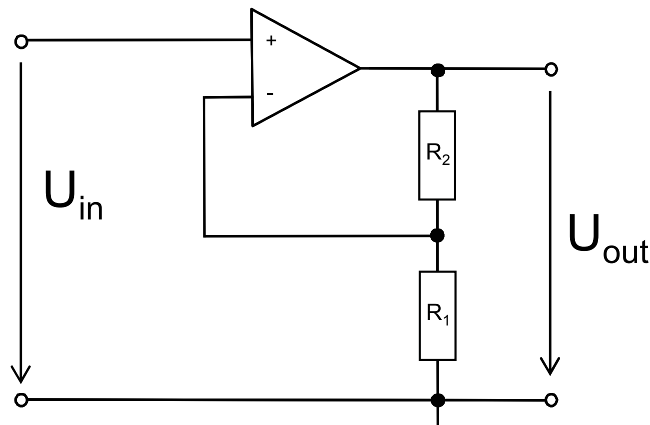


Abbildung 5.11.: Prinzip des nichtinvertierenden Verstärkers

Da der invertierende Eingang sehr hochohmig ist, fließt in ihn fast kein Strom. Folglich fließt durch *R2* und *R1* der gleiche Strom. Weiterhin befindet sich der nichtinvertierende Eingang auf Grund der hohen Leerlaufverstärkung des Operationsverstärkers praktisch auf dem gleichen Potential wie der invertierende Eingang. Folglich fällt an *R1* die Eingangsspannung U_{in} und an *R2+R1* die Ausgangsspannung U_{out} ab. Da nun durch *R1* und *R2* die gleichen Ströme fließen, ist der Spannungsabfall an ihnen proportional zu den Widerstandswerten, und man kann schreiben (Absatz nach Lange-Janson, 2012):

$$\begin{aligned} V &= \frac{U_{out}}{U_{in}} \\ V &= 1 + \frac{R2}{R1} \end{aligned} \tag{5.7}$$

Da die Ausgangsspannung des bereits im vorherigen Abschnitt erwähnten DAC im Bereich von $0V$ bis $2.048V$ liegt, wird für den Verstärkungsfaktor *V* der Wert 58 gewählt, um die vollen 120 Volt der Piezostacks aussteuern zu können. Bei der Wahl der Widerstandswerte ist darauf zu achten, dass sowohl der Ausgang des Operationsverstärkers nicht zu stark belastet wird (*R1* und *R2* zu klein) als auch der Strom sich am invertierenden Eingang nicht bemerkbar macht (*R1* und *R2* zu groß). Durch die Wahl von *R1* zu $8.2k\Omega$ und *R2* zu $470k\Omega$ ergibt sich ein Maximalstrom von $0.3mA$, welcher einen guten Kompromiss darstellt. Für den Operationsverstärker steht zunächst ein LTC6090 der Firma Linear Technology zur Verfügung (Linear Technology, 2012). Dieser zeichnet sich in erster Linie durch die hohe Versorgungsspannung bis 140 Volt aus, die sowohl symmetrisch ($\pm 70V$) als auch unsymmetrisch ($140V$) beschaltet werden kann. Durch die rail-to-rail-Fähigkeit reicht die Ausgangsspannung bis fast an die Versorgungsspannung heran. Daher erscheint dieser Operationsverstärker bestens geeignet für die vorliegende Anwendung (vgl. Linear Technology, 2012).
Das letzte Bauelement im „Signalweg“ ist ein Optokoppler, der die Eingangsseite (Ansteuerung/FPGA) vom Rest der Schaltung galvanisch trennt. Bei der Wahl des Optokopplers gilt es darauf zu achten, dass dieser die hochfrequenten SPI-Signale verzö-

gerungsfrei übertragen kann. Dies wird mit dem ausgewählten Avago HCPL-090J gewährleistet, der eine Laufzeitverzögerung von maximal 15ns und eine Datenrate von 100MBaud erlaubt (Avago, 2013).
Die Spannungsversorgung der Bauteile wird durch DC/DC-Wandler realisiert, die zum einen die 120 Volt für die OpAmps (Recom International, 2010) und 5 Volt für den DAC und den Optokoppler (Lineage Power, 2008) zur Verfügung stellen. Beide Wandlertypen werden eingangsseitig mit 12 Volt gespeist, sodass die Grundversorgung der Schaltung auf diese Spannung festgelegt werden kann. Abbildung 5.12 zeigt den ersten Schaltungsentwurf auf einer Lochrasterplatine für einen Piezokanal. Das Signal (schwarze Pfeile) durchläuft dabei folgende, durch farbige Kästen hervorgehobene Bauteile:

- Optokoppler (blau)
- Pull-Up-Widerstände (grün)
- DAC (rot)
- Operationsverstärker (gelb)
- Spannungsteiler, der den Verstärkungsfaktor bestimmt (magenta)

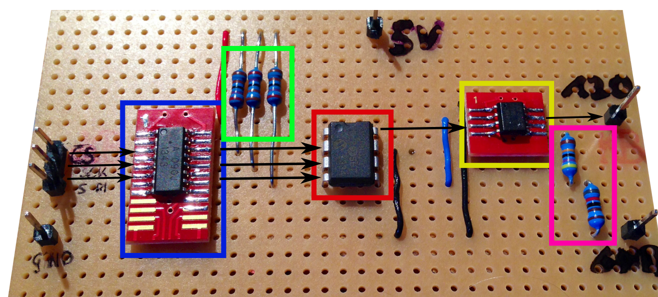


Abbildung 5.12.: Erster Schaltungsentwurf auf einer Lochrasterplatine für einen Kanal

Anschließend ist die Schaltung mit Hilfe der CAD-Software EAGLE (CadSoft, 2014) als Schaltplan gezeichnet und daraus mit der selben Software ein Platinenlayout entworfen worden. Die fertig bestückte Platine ist auf Abbildung 5.13 zu sehen.

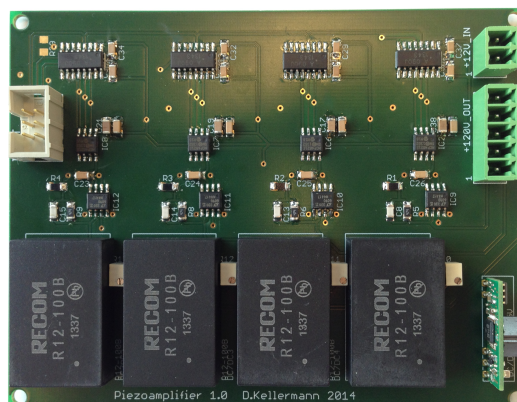


Abbildung 5.13.: Bestückte Verstärkerplatine Version 1.0

Nach ausführlichen Messungen an der fertigen Platine hat sich herausgestellt, dass die Operationsverstärker nicht korrekt arbeiten, gut zu sehen auf Abbildung 5.14.

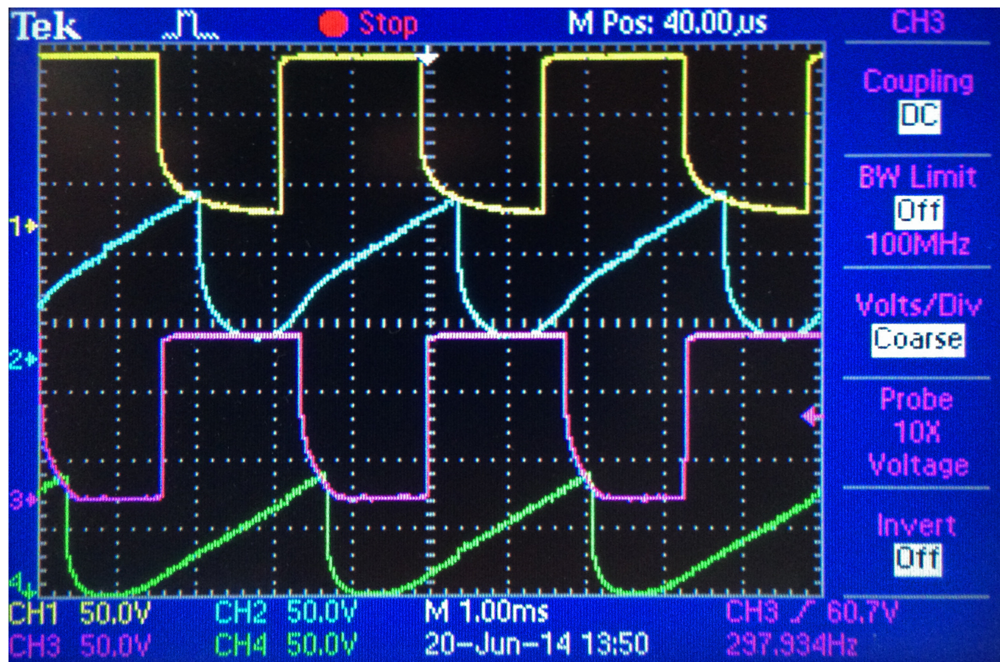


Abbildung 5.14.: Fehlerhafte Verstärkung der Operationsverstärker der Kanäle 1 (gelb), 2 (türkis), 3 (violett), 4 (grün)

Wie unschwer zu erkennen ist, kommt es bei jeder fallenden Flanke zu einer Kurvenbildung, die einer kapazitiven Entladekurve gleichkommt. Die Ursache hierfür ist die *Input Common Mode Range* des OpAmps. Dieser Bereich, der laut Datenblatt mit 3 Volt angegeben ist, gibt die Differenz der Eingangsspannung zur Versorgungsspannung an. Da in der vorliegenden Schaltung am Operationsverstärker unsymmetrische Versorgungsspannung anliegt, liegt folglich der Eingang der negativen Versorgungsspannung auf Masse, also 0 Volt. Da der Spannungsbereich der Eingangsspannung zwischen 0 und 2 Volt liegt, wird somit im unteren Bereich die *Common Mode Range* verletzt (keine 3 Volt-Differenz zwischen Versorgungs- und Eingangsspannung), der Operationsverstärker arbeitet nicht mehr korrekt und die zu sehende Entladekurve ist die restliche Ladung, die über den Widerstand des Spannungsteilers nach Masse fließt. Dieser Umstand kann ebenfalls in einer Simulation mit LTSpice beobachtet werden.

Dieser Fehler wird behoben, indem der Anschluss der negativen Versorgungsspannung mit -5 Volt beschaltet wird, um zu jedem Zeitpunkt die geforderte Spannungsdifferenz zu gewährleisten. Weiterhin wird der 5 Volt DC/DC-Wandler von Prototyp 1.0 gegen zwei isolierte Wandler getauscht, um sowohl die Primärseite als auch die Sekundärseite des Optokopplers mit galvanisch getrennten 5 Volt zu versorgen. Der zugehörige Schaltplan als auch das Platinenlayout von Prototyp 1.1 sind im Anhang A zu finden. Die fertig bestückte Platine ist auf Abbildung 5.15 zu sehen sowie, Bezug nehmend auf Abbildung 5.14, eine Messung aller vier Ausgangskanäle, die nun korrekt dargestellt werden, zu sehen auf Abbildung 5.16.

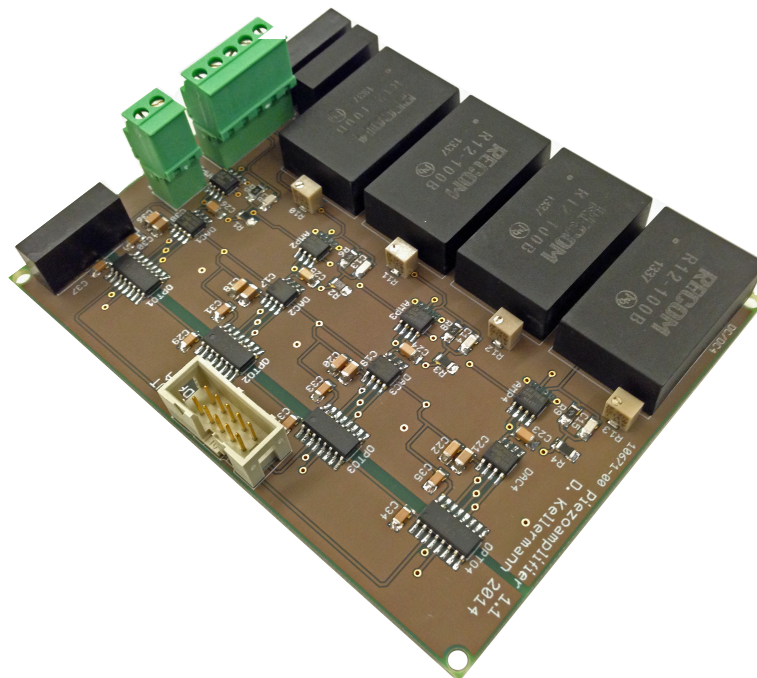


Abbildung 5.15.: Bestückte Verstärkerplatine Version 1.1

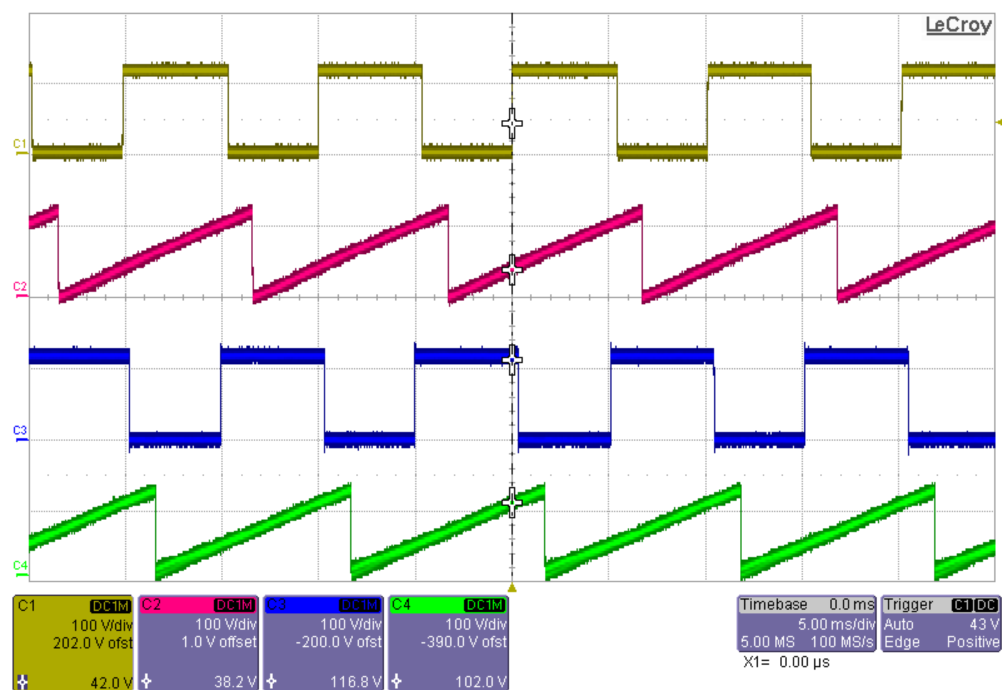


Abbildung 5.16.: Ausgangssignale der Kanäle 1 (gelb), 2 (rot), 3 (blau) und 4 (grün) des Verstärkers Version 1.1 bei 100Hz

Wie bereits zu Beginn des Kapitels erwähnt, ist dieser Verstärker nicht für die geforderte Frequenz nach Tabelle 1.2 geeignet. Wie dem Datenblatt des Operationsverstärkers entnommen werden kann, ist dieser bis 200pF Kapazität am Ausgang *unity gain stable*, was bedeutet, dass er mit Gegenkopplungsfaktor 1 nicht ins Schwingen gerät. Wie Messungen zeigen, lassen sich auch höhere Kapazitäten treiben, allerdings auf Kosten der Signalstabilität: Ab ca $300Hz$ kommt es zu Einbrüchen in der Signalform, die durch den Überhitzungschutz des Operationsverstärkers ausgelöst werden. Da in der Schaltung der *TFLAG*- und *OD*-Pin des Bauteils miteinander verbunden worden sind, schaltet sich dieses ab $150\,^{\circ}\mathrm{C}$ selbst ab. Die Belastung ist mit einem Rechtecksignal höher als mit einem Sägezahn, daher kommt es dementsprechend häufiger zu einer Abschaltung, wie man Abbildung 5.17 entnehmen kann.

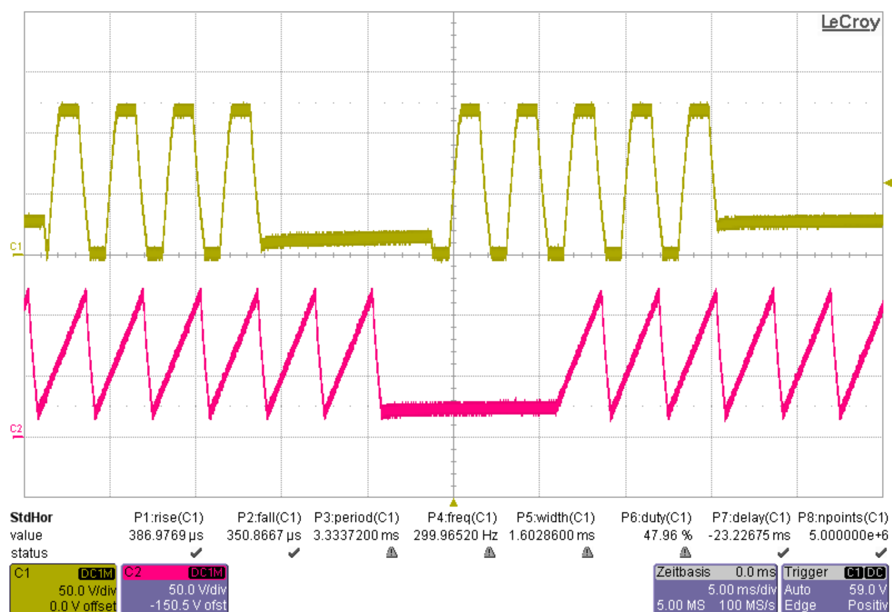


Abbildung 5.17.: Einbruch der Wellenform bei ca. 300Hz mit 210nF Last am Verstärkerausgang durch thermische Abschaltung des OpAmps

Zusätzlich ist zu sehen, dass die Flanken der Signale nicht mehr senkrecht, sondern schräg verlaufen ebenso die Bildung eines Offsets, insbesondere beim Sägezahnsignal. Beides lässt sich durch den zu niedrigen Ausgangsstrom erklären, was folgende Rechnung belegt: Nimmt man den im Datenblatt des Operationsverstärkers angegebenen Kurzschlussstrom von 50mA am Ausgang, so lässt sich die Ausgangsimpedanz mit 120 Volt Ausgangsspannung berechnen zu:

$$
\begin{aligned}
Z_R &= \frac{u(t)}{i(t)} \\
Z_R &= 2.4k\Omega
\end{aligned}
\tag{5.8}
$$

Stellt man nun Gleichung 5.3 auf die Frequenz um und setzt für R die zuletzt ausgerechnete Impedanz Z_R und für $C = 210nF$ (Kapazität des Piezostacks) ein, so ergibt sich eine Grenzfrequenz von:

$$f_g = \frac{1}{2 \cdot \pi \cdot 2.4k\Omega \cdot 210nF}$$
$$f_g = 315.8Hz \tag{5.9}$$

Die Anforderungen an einen zweiten Prototyp sind damit eindeutig: Das Schaltungsdesign muss dahingehend verändert werden, dass der Verstärker sowohl die in Abschnitt 1.4 geforderten $10kHz$ Maximalfrequenz als auch genügend Ausgangsstrom für die gewählten Piezokapazitäten liefern kann. Die Umsetzung desselben wird im folgenden Abschnitt erläutert.

5.2.3.2. Prototyp 2

Das Schaltungskonzept des zweiten Prototyp baut auf dem Prinzip eines *Class AB*-Push-Pull-Verstärkers auf. Dieser Schaltungstyp vereint die Vorteile der nachfolgend aufgeführten *Class A*- und *Class B*-Verstärker und ist daher der effizienteste Typ der Analogverstärker.

Class A-Verstärker
Der *Class A*-Verstärker besteht aus einem aktiven Bauteil, welches sowohl die positive als auch negative Halbwelle des Eingangssignals verstärkt. Er zeichnet sich durch geringe Verzerrungen im Nulldurchgangsbereich aus. Da das Bauelement hierfür jedoch ständig in der Mitte des linearen Teils seiner Kennlinie gehalten werden muss, weist dieser Verstärkertyp einen relativ schlechten Wirkungsgrad (im Idealfall bei 50%, üblicherweise bei 25%) auf (nach ITWissen, 2014).

Class B-Verstärker
„Wesentlich effektiver als die nach *Class A* aufgebauten Endstufen sind *Class B*-Verstärker mit einer theoretischen Effektivität von fast 80 Prozent. Hier übernehmen zwei getrennte aktive Bauteile jeweils eine Halbwelle des eingehenden Wechselspannungssignals“ (Muthig, 2010). Somit ist immer nur ein Bauteil zur Zeit eingeschaltet, wodurch der Ruhestrom des jeweils anderen Bauteils wegfällt. Dies ergibt zwar einen höheren Wirkungsgrad, die Linearität ist jedoch durch den Übergang der beiden Kennlinien deutlich ungenauer und es kommt zu Übernahmeverzerrungen.

Class AB-Push-Pull-Verstärker
Wie eingangs erwähnt, ist dieser Verstärkertyp eine Kombination aus den beiden vorangegangenen. Genau wie beim *Class B*-Verstärker kommen auch hier zwei aktive Bauelemente zum Einsatz, mit dem Unterschied, dass diese ständig mit einem geringen Ruhestrom versorgt werden und so das Übernahmeverzerren vermieden werden kann. Auch liegt der Wirkungsgrad mit 40-60% weit über dem des *Class A*-Verstärkers. Der Begriff des Push-Pull-Verstärkers bezieht sich dabei sprichwörtlich auf die Funktionsweise: Ein Bauelement „drückt" den Stromfluss in die Last, wohingegen das andere denselben aus der Last „zieht".

Abbildung 5.18 verdeutlicht diesen Sachverhalt: Zum Zeitpunkt t_1 wird der Aktor mit maximalem Strom aufgeladen (*push*), zudem fließt ein geringer Ruhestrom. „Auf Grund des kapazitiven Charakters wird zum Halten einer bestimmten Position ohne mechanische Laständerung keine Leistung benötigt" (Stiebel u. a., 2014, S. 1), somit fließt zum Zeitpunkt t_2 lediglich der Ruhestrom. Zum Zeitpunkt t_3 wird der Aktor dann mit maximalem Strom entladen (*pull*), weiterhin fließt ein geringer Ruhestrom.

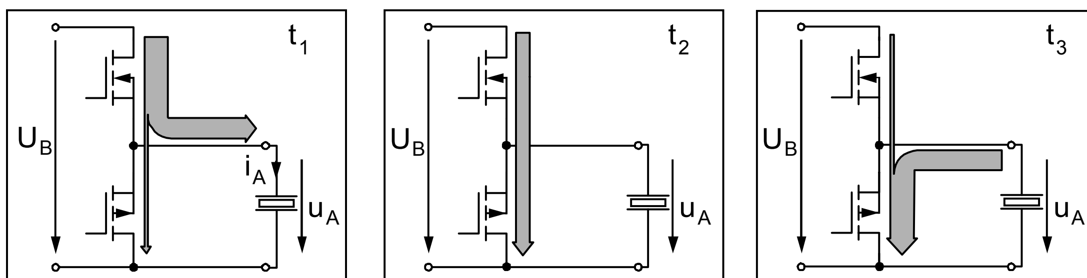


Abbildung 5.18.: Stromverlauf Class AB-Push-Pull-Verstärker (nach Stiebel u. a., 2014, S. 3)

Üblicherweise kommen in dem zuletzt genannten Verstärkertyp komplementäre Transistoren oder MOSFETs zum Einsatz (jeweils für die positive und negative Halbwelle des Eingangssignals), je nachdem, ob die Basis, bzw. das Gate mit Strom oder Spannung getrieben werden soll. Für den vorliegenden Verstärker werden MOSFETs gewählt, da diese im Vergleich zu Bipolartransistoren sehr kostengünstig sind und sich im Betrieb durch ihr Kaltleiterverhalten (PTC) auszeichnen: Bei steigender Temperatur im Bauteil nimmt dessen Leitfähigkeit ab, dadurch wiederum auch die Wärmeproduktion, und es findet sich ein stabiler Arbeitspunkt (Vishay, 2014a,b).
Die Wahl des Gatetreibers ist auf den LM4702, einen Audio-Leistungsverstärker der Firma Texas Instruments gefallen (Texas Instruments, 2013), der bis zu einer Spannung von 200 Volt ($\left|V^+\right| + \left|V^-\right|$) beschaltet werden kann und gleich zwei Kanäle in einem Gehäuse unterbringt (Einsatz als Stereoverstärker). Weiterhin kann dieser Treiber nur symmetrisch betrieben werden, was jedoch der aus den MOSFETs bestehenden Gegentaktendstufe zupass kommt und das Einstellen des Ruhestroms vereinfacht.
Da es bei dieser Beschaltungsart zwangsläufig zu einer negativen Halbwelle der Ausgangsspannung kommt, der Spannungsbereich des verwendeten Piezoaktors jedoch

rein positiv verläuft, wird als Massepotential für den Piezoaktor am Ausgang V^- gewählt. Somit „sieht“ dieser den vollen, für ihn rein positiven Spannungsbereich von 0 Volt bis $V^- + V^+$.
Die restliche Schaltung ist weitestgehend identisch mit der des ersten Prototyps (Optokoppler, Pull-Up-Widerstände, DAC), bis auf zwei Potentiometer, die sowohl DAC-ausgangsseitig als auch in der Gegenkopplung des Audiotreibers hinzukommen. Über Ersteres lässt sich die Dämpfung des Eingangssignals, über das Zweite der Offset des Ausgangssignals des Verstärkers regulieren, was eine spätere Feinjustierung ermöglicht. Weiterhin sind sowohl eingangs- als auch ausgangsseitig Sicherungen hinzugefügt worden, um die Bauteile vor Kurzschlüssen und Überlastung zu schützen. Ebenso zwei Glättungskondensatoren, die die Restwelligkeit der gleichgerichteten Eingangsspannung minimieren sollen (Panasonic, 2014).

Nachdem die Schaltung auf einem Breadboard erfolgreich getestet worden ist, ist das Platinenlayout mit Hilfe der Software EAGLE umgesetzt worden (siehe Anhang A). Abbildung 5.19 zeigt schließlich den fertigen Verstärker. Die Datenblätter aller verwendeten Bauteile sind Anhang D zu entnehmen.

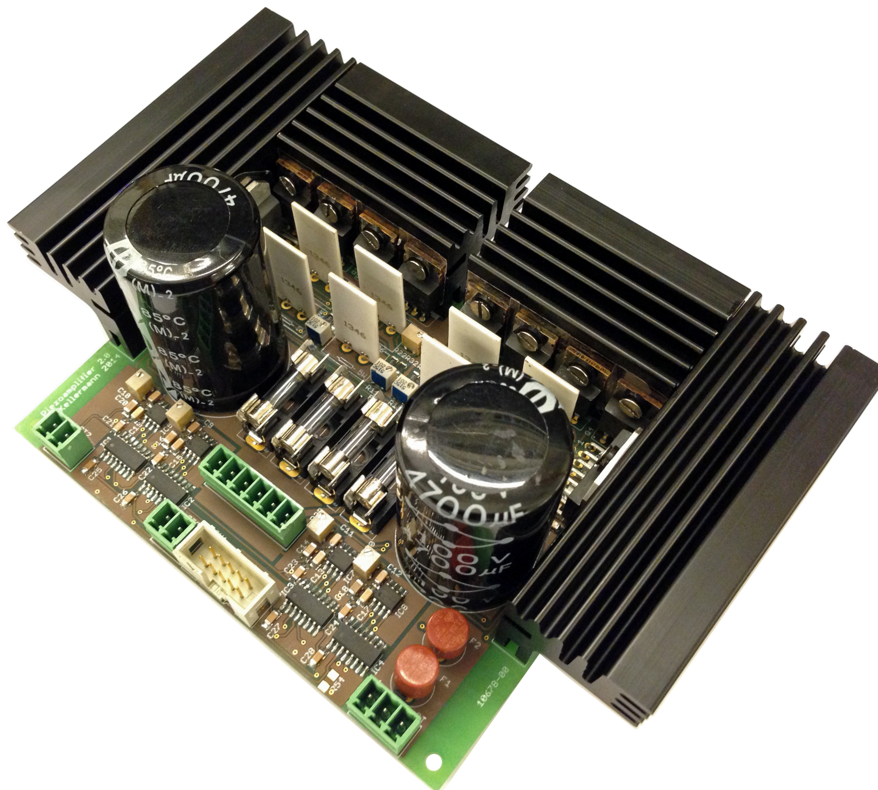


Abbildung 5.19.: Unten: SPI-Eingang, Optokoppler und DAC; Mitte: Ausgangssicherungen und Glättungskondensatoren; Oben: An Kühlkörper geschraubte Audiotreiber und MOSFETs sowie Stabilisierungswiderstände (weiß)

5.2.4. Leistungsbetrachtung des Verstärkers und Aktors

Dieser Abschnitt stellt eine Leistungsbetrachtung des Verstärkerprototypen 2 und des Aktors dar und soll zudem Aufschluss über das elektromechanische Modell eines Piezoelements geben.

5.2.4.1. Gesamtleistungsaufnahme des Aktors

Für die Messung der Leistung, die ein Piezostack in einer Periode aufnimmt, werden zeitgleich die Spannung über den Aktor und der Strom in Reihe über einen 1Ω Widerstand aufgenommen. Abbildung 5.20 und 5.21 stellen jeweils für $f = 1kHz$ den Strom-Spannungsverlauf für das Rechteck- und Sägezahnsignal dar.

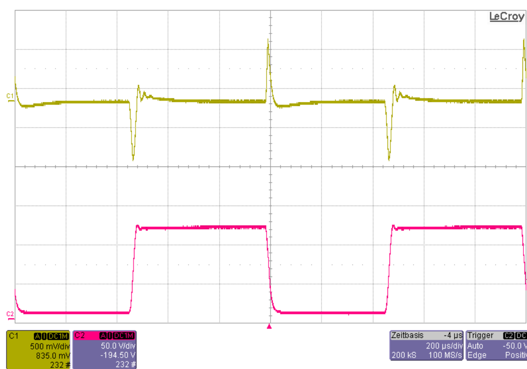


Abbildung 5.20.: Strom- (grün) und Spannungsverlauf (magenta) des Piezoaktors bei Ansteuerung mit einem $f = 1kHz$ Rechtecksignal

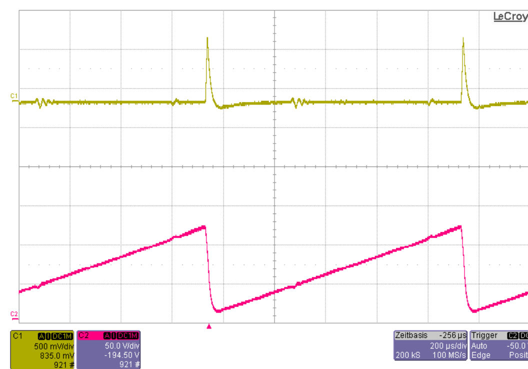


Abbildung 5.21.: Strom- (grün) und Spannungsverlauf (magenta) des Piezoaktors bei Ansteuerung mit einem $f = 1kHz$ Sägezahnsignal

Da sich der Stromverlauf wie die erste Ableitung des Spannungsverlaufs verhält und sich für den Stromfluss im Kondensator aus der Gleichung $Q = C \cdot U$ die Beziehung zwischen Strom und Spannung zu

$$I = \frac{\mathrm{d}Q}{\mathrm{d}t} = C \cdot \frac{\mathrm{d}U}{\mathrm{d}t}$$

ergibt, bestätigt sich die Vermutung, dass ein Piezoelement kapazitives Verhalten aufweist. Mit Hilfe von MATLAB wird die mittlere Leistung einer Periode nach folgender Formel berechnet:

$$\overline{|p(t)|} = \frac{1}{T} \int_0^T |u(t) \cdot i(t)| \, \mathrm{d}t \tag{5.10}$$

Daraus ergeben sich für Rechteck- und Sägezahnsignal folgende mittlere Leistungen:

$$\overline{|p(t)_{Square}|} = 2.59W$$
$$\overline{|p(t)_{Saw}|} = 1.29W$$

Eine schriftliche Überprüfung der Rechnung[3] ergibt folgende Werte:

$$\overline{|p(t)_{Square}|} = 3.02W$$
$$\overline{|p(t)_{Saw}|} = 1.51W$$

Diese Werte weichen zwar um ca. 16% von denen mit MATLAB errechneten ab, das Verhältnis der Ergebnisse untereinander stimmt jedoch überein, sodass angenommen werden kann, dass die Abweichungen auf grafische Abschätzung zurückzuführen sind. Abbildung 5.22 stellt nochmal die MATLAB-Ergebnisse gegenüber:

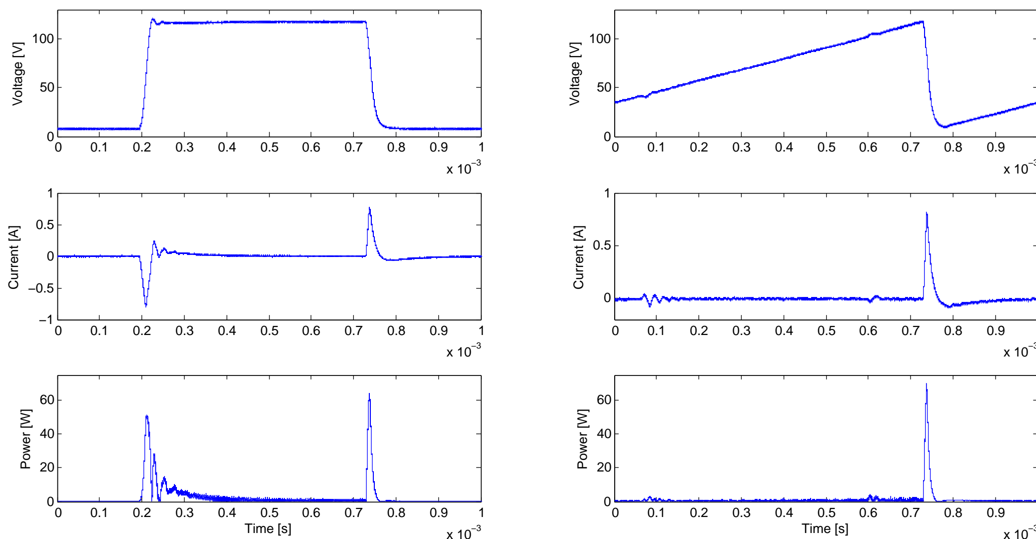


Abbildung 5.22.: Spannung, Strom und mittlere Leistung von Rechteck- (links) und Sägezahnsignal (rechts)

Es fällt auf, dass die größte Leistungsaufnahme vom Piezoaktor an den Spannungsflanken stattfindet, die restliche Zeit ist die Leistung nahezu 0 Watt. Diese Beobachtung deckt sich mit dem Zitat von Stiebel u. a. von Seite 64, dass zum Halten einer Position keine Leistung benötigt wird. Weiterhin verdeutlicht die Abbildung, dass der Strombedarf und die daraus resultierende Leistungsaufnahme eines Piezoaktors bei Ansteuerung mit Wellenformen, die steile Signalflanken enthalten, weit über dem Strombedarf liegt, der mit harmonischen Signalen anfallen würde: Mit Formel 5.11 (vgl. Physik Instrumente,

[3] Die vollständige Rechnung ist im Anhang B zu finden.

2009, S. 2-196), die den Spitzenstrombedarf bei Sinusbetrieb darstellt, ergibt sich mit $f = 1kHz$, $C = 210nF$ und $U_{pp} = 120V$ ein Strombedarf von:

$$\begin{aligned} i_{max} &\approx f \cdot \pi \cdot C \cdot U_{pp} \\ i_{max} &\approx 80mA \end{aligned} \tag{5.11}$$

wohingegen Abbildung 5.22 ein Spitzenstrom von ca. $800mA$ entnommen werden kann, also ein um Faktor 10 höherer Strom. Es lässt sich somit festhalten, dass bei der Wahl von Piezoverstärkern darauf geachtet werden sollte, welche Signalformen verstärkt werden sollen, da theoretisch die Spitzenströme bei unendlicher Flankensteilheit auch unendlich hoch werden.

5.2.4.2. Verlustleistung

Für die Betrachtung der Verlustleistung wird zunächst das elektrische Ersatzschaltbild eines Piezoelements hinzugezogen.

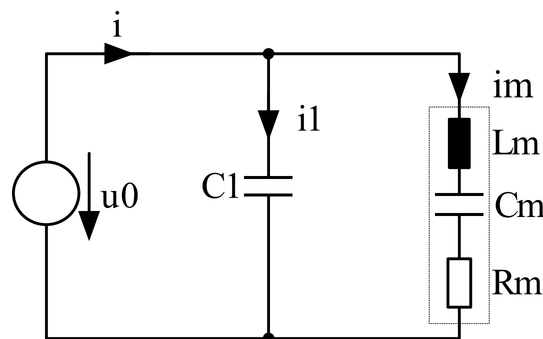


Abbildung 5.23.: Elektrisches Ersatzschaltbild eines Piezoelements (nach Kessler, 2014)

Dieses besteht aus einer mit Strom $i1$ durchflossenenen Kapazität $C1$ sowie parallel dazu einem „mechanischen “ Teil (gestrichelter Kasten), bestehend aus einem Reihenschwingkreis mit Induktivität Lm, Kapazität Cm und Widerstand Rm, durchflossen vom Strom im. Für die Herleitung dieses Ersatzschaltbildes sei an dieser Stelle auf (Kessler, 2014) verwiesen. Der „mechanische“ Teil basiert auf dem Feder-Masse-Modell und ist hauptverantwortlich für die Verluste eines Piezoelements. Diese Verluste werden durch den dielektrischen Verlustfaktor $\tan\delta$ ausgedrückt. „Bei Aktor-Piezokeramik liegt der Verlustfaktor [...] in der Größenordnung von 0.01-0.02. Dadurch werden bis zu 2% der elektrischen Leistung, die in den Aktor fließen, in Wärme umgewandelt“ (Physik Instrumente, 2009, S. 2-198). Somit ergibt sich für die unter Abschnitt 5.2.4.1 bestimmten Werte für die Gesamtleistung ein Verlustleistungsanteil für Rechteck- und Sägezahnsignal von:

$$\begin{aligned} P_{V,Square} &\approx 52mW \\ P_{V,Saw} &\approx 26mW \end{aligned}$$

Die Leistung, die wiederum in den MOSFETs des Verstärkers in Wärme umgesetzt wird, beträgt näherungsweise das Doppelte von der Gesamtleistung, die der Piezoaktor aufnimmt: Beim Laden des Aktors wird prinzipbedingt vom N-Kanal-MOSFET genauso viel Leistung in Wärme umgesetzt wie der Aktor aufnimmt. Beim Entladen wandelt schließlich der P-Kanal-MOSFET die restliche vom Aktor abgegebene Energie in Wärme um. Demnach muss nach Formel 5.12 (vgl. Stiebel u. a., 2014, S. 2):

$$P_V \approx 2 \cdot C \cdot {U_{pp}}^2 \cdot f \tag{5.12}$$

der hiesige Verstärker eine mittlere Gesamtleistung für Rechteck- und Sägezahnsignal pro Periode von:

$$P_{V,Square} \approx 5.2W$$
$$P_{V,Saw} \approx 2.6W$$

aufbringen. Da kein potentialfreies Oszilloskop zur Verfügung steht und somit Spannung und Strom über den MOSFETs nicht messbar sind, können die oben genannten Werte nicht überprüft werden. Durch die Erwärmung der Kühlkörper können die Werte jedoch als realistisch betrachtet werden.

5.2.5. Reglerentwurf

Der Piezomotor soll sowohl geschwindigkeits- als auch positionsgeregelt sowie die Nichtlinearitäten des Aktors durch eine inverse Steuerung ausgeglichen (vgl. 4.2.2) werden. Von einer Regelung mittels Ladungsansteuerung statt Spannungsansteuerung, die bei Piezoaktoren häufig Verwendung findet, wird an dieser Stelle abgesehen. Nachfolgend werden die Entwürfe in VHDL betrachtet, ebenso die Erfassung der Regelgröße mittels Interferometer.

5.2.5.1. Erfassung der Regelgröße

Für die Verwendung eines klassischen Reglers bedarf es neben der Sollwertvorgabe eines Istwerts des zu regelnden Systems. Die Position des Läufers des Piezomotors ist in diesem Fall die Regelgröße, dessen Istwert zunächst über ein Interferometer erfasst wird. Besagtes Interferometer des Typs AttoFPSX010 der Firma Attocube ist ein vollautomatischer Ausdehnungssensor mit einer Auflösung von $1pm$, dessen Messwerte als Quadratursignal ausgegeben werden können (Attocube Systems, 2013). Als Schnittstelle zwischen Interferometer und FPGA dient ein weiteres Mal der PiLC, der die Encodersignale weitergibt. Das Einlesen des Signals in VHDL geschieht wie folgt:

Ein Quadratursignal besteht in der Regel aus zwei 90 Grad zueinander phasenverschobenen Rechteckimpulsen mit 5 Volt (TTL) oder 3.3 Volt (LVTTL) Ausgangspegel. Werden die jeweiligen Signale zusammengefasst, so ergibt sich ein einfacher 2 Bit-Graycode,

dessen Bitänderung einen Zählschritt bedeutet. Tabelle 5.24 stellt so einen Graycode dar.

Position	Graycode
0	00
1	01
2	10
3	11

Abbildung 5.24.: Zählweise des Graycodes

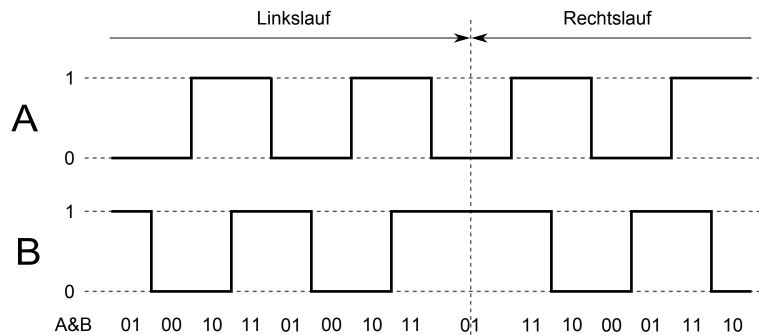


Abbildung 5.25.: Quadratursignal mit Richtungswechsel

Ein Vergleich des aktuellen Zustandes mit dem vorherigen gibt zudem Aufschluss über die Bewegungsrichtung. Folgt beispielsweise dem Zustand „00“ der Zustand „01“, so liegt eine Bewegung in positiver Richtung vor, wohingegen in diesem Fall der Zustand „10“ eine negative Richtung anzeigt. Nach genau diesem Prinzip arbeitet auch der VHDL-Code. Die Signale werden zunächst zusammengefasst und anschließend in einer *case*-Anweisung ausgewertet. Die vier Fälle der *case*-Anweisung stellen dabei den vorherigen Signalzustand dar. Je nach Folgezustand wird dann ein Zähler in- oder dekrementiert und ausgegeben. Auf Abbildung 5.26 ist der Encoderblock aus Quartus II zu sehen mit den Quadratursignaleingängen *A* und *B* sowie dem Systemtakt und einem *Reset*-Eingang, über den sich die Position auf Null zurücksetzen lässt. Dieser ist insofern von Nöten, da sich der gemessene Positionswert des Interferometers in dessen Software ebenfalls zurücksetzen lässt. Das Zurücksetzen sollte somit möglichst zeitgleich passieren, da es sonst zu falschen Positionswerten kommen würde.

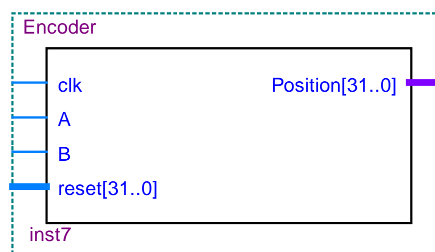


Abbildung 5.26.: VHDL Blocksymbol des Encoders

Die Umrechnung von Zählschritten in absolute Positionen gibt letztendlich der *Conversionfactor* an. Beträgt dieser beispielsweise $1nm$, dann entspricht das 1000 Zählschritten. Läuft der ausgegebene Counterwert z.B. von 0 bis 5000, dann hat sich der Läufer des Motors um 5000 Nanometer, also $5\mu m$ bewegt. Die gezählten Schritte gibt der Encoder an den Komparator und PID-Regler weiter, denen diese Werte als Istwert dienen.

5.2.5.2. Inverse Steuerung in VHDL

Um die Nichtlinearität der Piezoaktoren laufend auszugleichen, wird die Ansteuerung durch die bereits in Abschnitt 4.2.2 eingeführte inverse Steuerung ersetzt. Für die Implementierung in VHDL werden in einem ersten Schritt frühe Ausdehnungs-Spannungsmessungen am Aktor herangezogen, die mit Hilfe von MATLAB an der Winkelhalbierenden (Sollsignal) gespiegelt, anschließend auf 4096 Werte normiert und in zwei LUTs, jeweils für Lade- und Entladevorgang, abgelegt werden. Um diese LUTs wiederum werden die bestehenden Signalgeneratoren in VHDL erweitert, sodass jetzt nicht mehr einfach der Phasenakkumulator direkt ausgegeben wird, sondern dieser bei Rechtslauf des Motors das LUT der invertierten Ladekurve und bei Linkslauf das LUT der invertierten Entladekurve durchgeht. Dabei kommt es zu folgendem Problem: Da die Zahlenwerte in den LUTs auf Grund des Hysteresecharakters nicht den selben Counterwerten des Phasenzählers entsprechen (Beispiel: Counter = 5 entspricht dem Wert 11 im Lade-LUT und 6 im Entlade-LUT), kommt es folglich beim Richtungswechsel zu einem Sprung im Ausgangssignal. Abbildung 5.27 stellt diesen Fehler anhand eines ModelSim Simulationsplots dar, der sowohl das Ausgangssignal (oben) mit den Sprüngen (rote Markierungen) als auch den Phasenzähler (unten) darstellt.

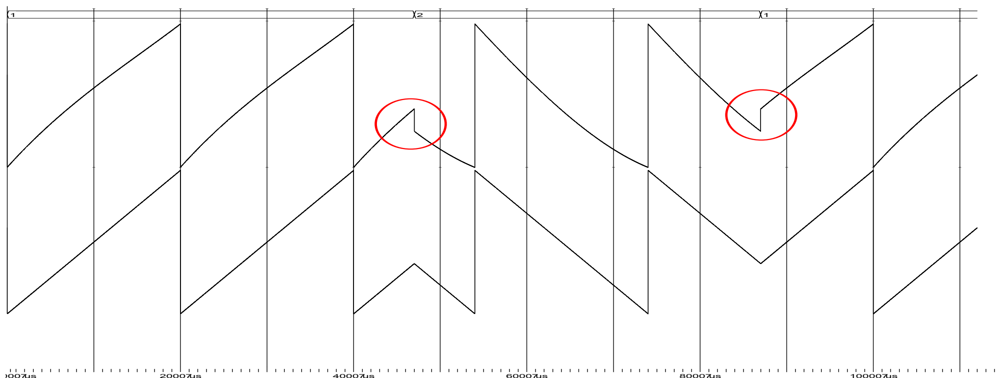


Abbildung 5.27.: Sprünge im Ausgangssignal beim Richtungswechsel durch falsche Counterwerte

Gelöst wird dieses Problem, indem zwei neue LUTs berechnet werden, die jeweils den Offset des Counterwerts zu jedem Signalwert beinhalten: Sowohl für den Sprung vom Lade- zum Entladeast als auch für den Sprung vom Entlade- zum Ladeast. Tabelle 5.1 stellt ein Beispiel für die ersten 15 Werte dar.

Counter	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Ladekurve	0	1	2	3	5	6	7	9	10	11	13	14	15	17	18
Entladekurve	0	1	1	2	3	3	4	5	5	6	7	8	8	9	10
Counteroffset	0	0	1	2	4	4	4	6	7	7	9	10	10	12	13

Tabelle 5.1.: Bestimmung des Counteroffsets zwischen Lade- und Entladekurve

Kommt es beispielsweise zu einem Richtungswechsel und der Ladeast hat den Wert 9, so wird nun auf den Counterwert (8) der zugehörige Wert aus der Offsettabelle (6) addiert. Da dieser neue Counterwert (14) im Entladeast dem Wert 9 entspricht, kommt es zu einem nahtlosen Richtungswechsel, wie Abbildung 5.28 verdeutlicht (oberer Graph).

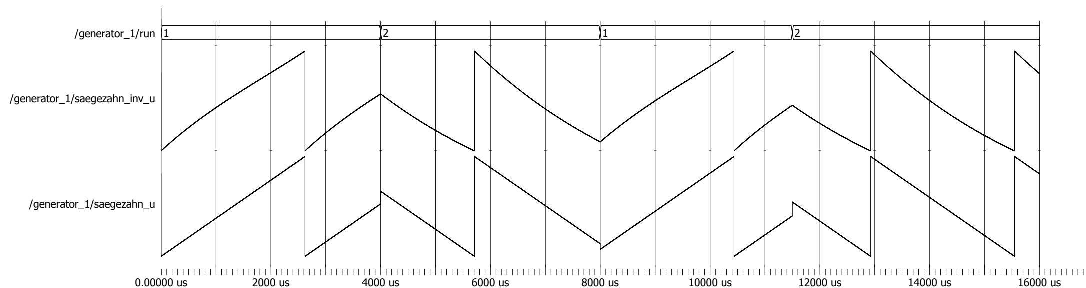


Abbildung 5.28.: Richtungswechsel mit nahtlosen Übergängen (oben) und Counter mit Sprüngen (unten)

Mit der fertigen inversen Ansteuerung sind anschließend einige Messungen durchgeführt worden, die gut die Auswirkung dieser Steuerung darstellen. Abbildung 5.29 zeigt den direkten Vergleich der Piezoausdehnung bei Ansteuerung mit dem herkömmlichen Sägezahnsignal und der bei inverser Steuerung. Die rote gestrichelte Linie verdeutlicht die jeweiligen Abweichungen, wobei sie im rechten Plot absichtlich um einige Millimeter parallelverschoben worden ist, um das Signal nicht zu verdecken.

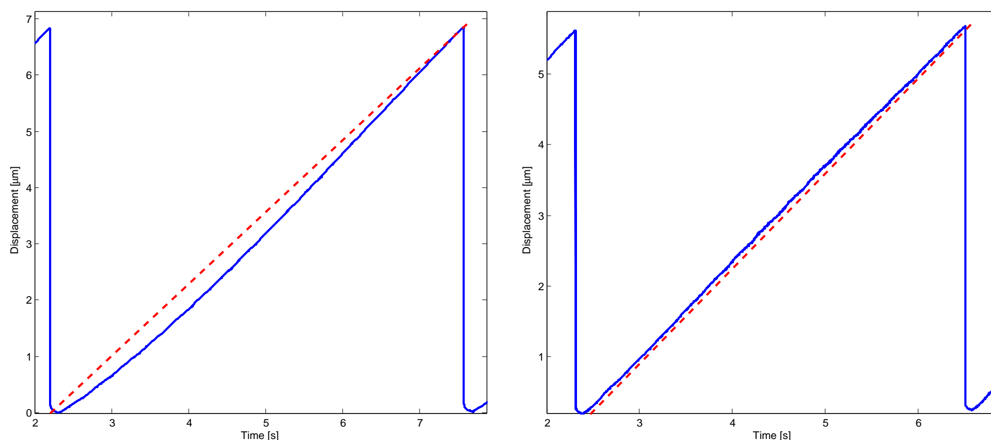


Abbildung 5.29.: Gegenüberstellung der Piezoausdehnung bei Ansteuerung mit regulärem Sägezahn (links) und inverser Steuerung (rechts)

Ein Vergleich der Messergebnisse mit denen aus der Simulation auf Seite 42 beweisen, dass diese Ansteuerungsart auch in der Realität sehr gut funktioniert.

5.2.5.3. Positionsregelung in VHDL

Für die Regelung der Position kommt ein Zweipunktregler zum Einsatz. Dieser wird in Form eines Komparators in VHDL implementiert, dem jeweils die Soll-und Istposition zugeführt wird. Die Ausgangsgröße des Komparators (Regelabweichung) wiederum dient den Signalgeneratoren als Richtungsindikator: Ist die Regelabweichung positiv, wird die Wellenform für den Rechtslauf, andernfalls für den Linkslauf mit der vom Benutzer vorgegebenen Frequenz erzeugt. Um eine etwaige zu hohe Geschwindigkeitsvorgabe durch den Benutzer und eine damit einhergehende Gefahr des „Überfahrens“ der Sollposition zu verhindern, wird kurz vor der Endposition die Vorgabe außer Kraft gesetzt. Der genaue Ablauf wird nachfolgend erklärt:
Um das Ausregeln der Sollposition ruhiger zu gestalten, empfiehlt es sich, dieses rein spannungsgesteuert, also nur von der Ausdehnung eines Vorschubpiezoaktors bewerkstelligen zu lassen. Anhand Abbildung 5.30 wird dieser Sachverhalt erläutert. Betrachtet wird an dieser Stelle nur ein Vorschubpiezoaktor, in der Umsetzung werden beide Aktoren auf diese Weise ausgeregelt.

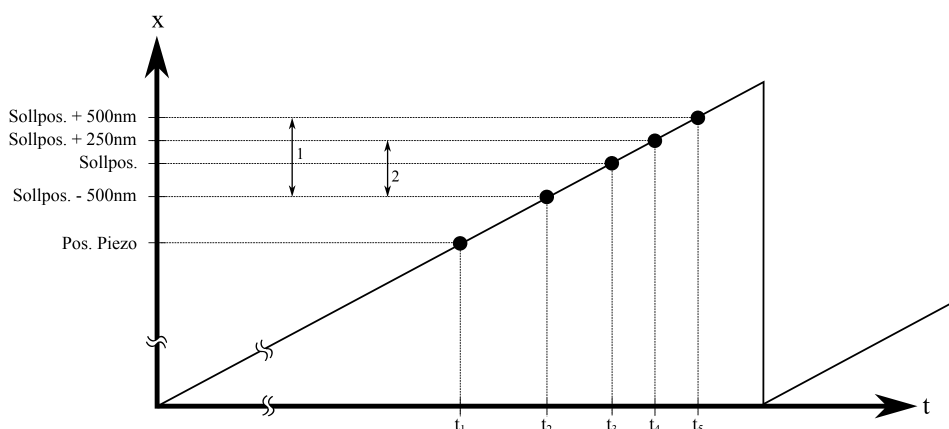


Abbildung 5.30.: Prinzip des Positionsregelalgorithmus

Zum Zeitpunkt t_1 befindet sich der Piezoaktor noch vor der Sollposition und wird mit der vom Benutzer vorgegebenen Frequenz angesteuert. Zum Zeitpunkt t_2 wird der Bereich $\pm 500nm$ um die Endposition überfahren (Pfeil 1) und die Vorgabegeschwindigkeit abgeschaltet. Jetzt gilt es zu prüfen, welcher der beiden Vorschubaktoren die Sollposition innerhalb seiner maximalen Ausdehnung erreichen könnte. Dafür wird festgelegt, dass sich der Piezoaktor mindestens bis zur Sollposition plus $250nm$ ausdehnen können muss (Pfeil 2) und dass $2\mu m$ Piezoausdehnung einer Auflösung von 4096 Schritten entspricht (12 Bit). Daraus ergeben sich durch Proportionalität für die $750nm$ Stellweg 1536 Schritte. Die Piezoausdehnung wird bewusst auf $2\mu m$ begrenzt, um sicher zu gehen, dass sie wirklich bis zur Sollposition reicht[4]. Die Abfrage lautet somit:

[4]Beispiel: Bei $6\mu m$ Ausdehnung entsprächen $750nm$ 512 Schritten. Da aber meistens davon ausgegangen werden kann, dass die tatsächliche Ausdehnung kürzer ist (z.B. durch externe Belastung etc.), würde sich die Anzahl der Schritte erhöhen und der Aktor die Sollposition so nie erreichen können.

```
 1   if z = Z0 then --Rechtslauf
 2     if (l_index + 1536) <= 4096 then
 3       ...
 4     end if;
 5   elsif z = Z1 then -- Linkslauf
 6     if (l_index - 1536) >= 0 then
 7       ...
 8     end if;
 9   else
10   ...
11   end if;
```

Listing 5.2: Auszug aus dem VHDL-Quellcode zur Positionsregelung

Die Variable *l_index* (Zeile 2 und 6) stellt dabei den letzten Ausdehnungswert des Piezoaktors dar. Liegt das Ergebnis der jeweiligen Abfrage innerhalb des erlaubten Bereichs, ist der betroffene Aktor in der Lage, die Endposition zu erreichen, ohne dass dabei das Phasenregister überlaufen und es zu einer Spannungsflanke kommen würde. In diesem Fall wird nur noch der Vorschubpiezo weiter angesteuert, jetzt jedoch mit einem Dreiecksignal. Weiterhin werden die Klemmaktoren permanent auf *HIGH* gesetzt und der jeweils andere Motorblock (Klemm- und Vorschubeinheit) wird komplett abgeschaltet. Abbildung 5.31 stellt den Regelalgorithmus für einen Vorschubaktor in ModelSim dar.

Erklärung der einzelnen Zeitpunkte (rote Markierungslinien):

- $0\mu s < t < 7000\mu s$: Abstand zur Sollposition *comp* $= 501nm$, normaler Linkslauf mit $\approx 370Hz$
- $t = 7000\mu s$: Eintritt in den $500nm$-Bereichs, dargestellter Piezoaktor kann Sollposition nicht erreichen (*l_index* = 1349) und öffnet den Klemmaktor, damit das zweite Motorelement frei regeln kann
- $t = 10500\mu s$: Abstand zur Sollposition wieder *comp* $= 501nm$, normaler Linkslauf mit $\approx 370Hz$
- $t = 14000\mu s$: Unterschreitung des $500nm$-Bereichs, dargestellter Piezoaktor kann Sollposition erreichen (*l_index* = 4073), schaltet den zweiten Vorschubaktor ab (*trigger* = 1) und auf Dreiecksignal mit $\approx 100Hz$ um
- $t = 18000\mu s$: Sollposition ist erreicht (*comp* $= 0nm$), Vorschubaktor hält die aktuelle Spannung
- $t = 19000\mu s$: Sollposition wird unterschritten (*comp* $= -1nm$), Vorschubaktor schaltet auf Rechtslauf und regelt dagegen
- $t = 20500\mu s$: Sollposition wird erneut erreicht (*comp* $= 0nm$), Vorschubaktor hält die aktuelle Spannung

Messungen der Positionsregelung werden im Kapitel 6.2.2 betrachtet.

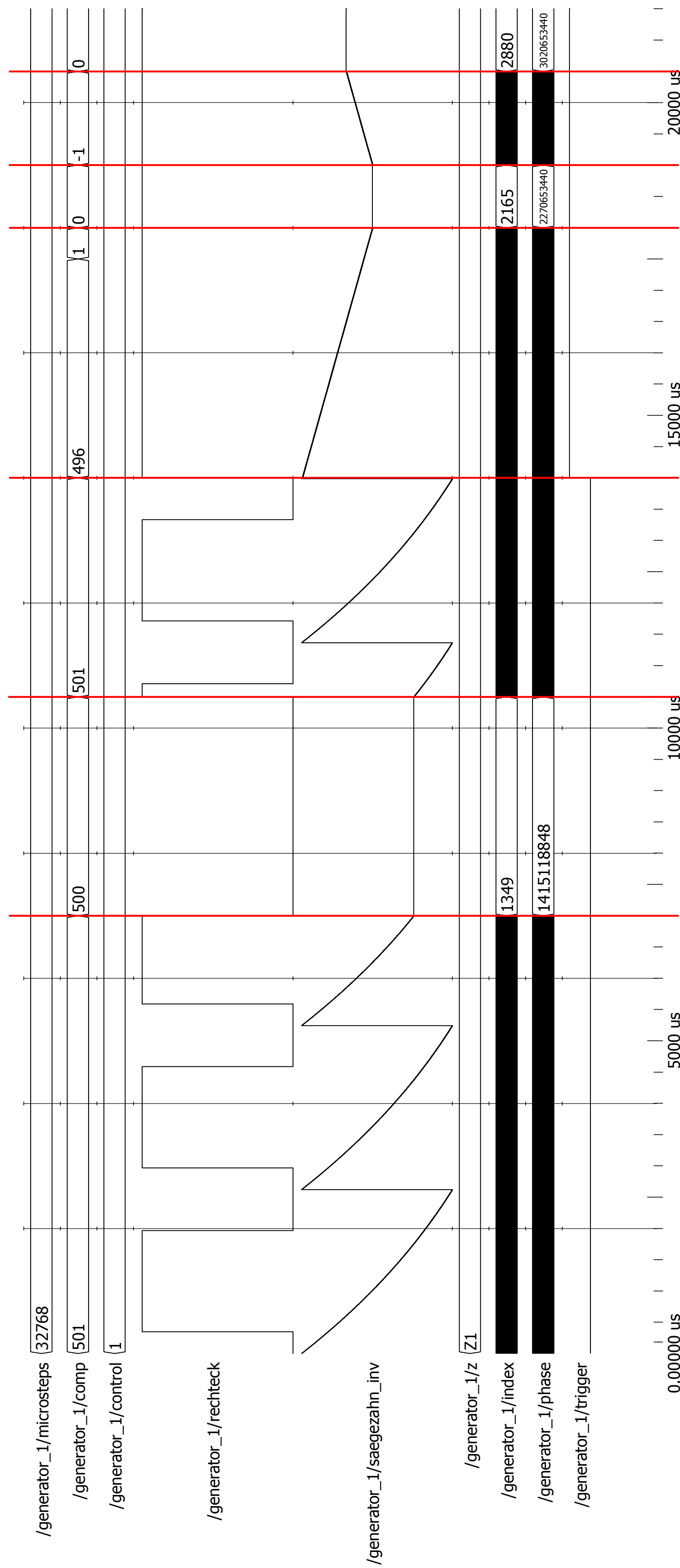


Abbildung 5.31.: Positionsregelalgorithmus in ModelSim

5.2.5.4. Geschwindigkeitsregelung in VHDL

Für die Regelung der Geschwindigkeit des Läufers kommt ein klassischer PID-Regler zum Einsatz, dessen P-, I- und D-Anteile sich zur Laufzeit ändern lassen. Nachfolgend wird der VHDL-Entwurf vorgestellt. Dafür wird zunächst die Differenzialgleichung des kontinuierlichen PID-Reglers in Parallelstruktur betrachtet:

$$y(t) = K_p \cdot e(t) + K_i \int_0^t e(\tau)\,\mathrm{d}\tau + K_d \frac{\mathrm{d}e(t)}{\mathrm{d}t} \tag{5.13}$$

Da es sich um einen digitalen Regler handelt, muss der Regler gemäß der Differenzialgleichung für den zeitdiskreten PID-Regler umgesetzt werden (Wenck, 2013, S. 9):

$$y_k = K_p \cdot e_k + K_i \cdot T_a \sum_{i=0}^{k} e_i + \frac{K_d}{T_a}(e_k - e_{k-1}) \tag{5.14}$$

Tabelle 5.2 führt die Variablen der Gleichung 5.14 auf und stellt das jeweilige Pendant der VHDL-Quelltext-Umsetzung diesen gegenüber.

Variablenname Gleichung	Variablenname VHDL	Erklärung
K_p	k_p	Proportionalbeiwert
K_i	k_i	Integrierbeiwert
K_d	k_d	Differenzierbeiwert
T_a	-	Abtastzeit
k	-	Nummer der Abtastung
e_k	e	Regeldifferenz
e_{k-1}	e_alt	Regeldifferenz vom vorherigen Abtastzeitpunkt
$\sum_{i=0}^{k}$	e_sum	Summenbildung

Tabelle 5.2.: PID-Parameter

Die fertige Umsetzung der Gleichung 5.14 in VHDL sieht dann wie folgt aus:

```
1  process begin
2    wait until rising_edge(clk);
3      e  <= w - x;      --Regeldifferenz
4      e_sum <= e_sum + e; --Summenbildung
5      --Anti-Windup
6      if e_sum < x"FFFFC568" then -- e_sum < -15000
7        e_sum <= x"FFFFC568";
8      elsif e_sum > x"00003A98" then --e_sum > 15000
9        e_sum <= x"00003A98";
10     end if;
11     y_i <= resize(((signed(k_p)*signed(e))+(signed(k_i)*signed(e_sum))+(signed(k_d)*
12     (signed(e)-signed(e_alt)))),32);
13     e_alt <= e;
14 end process;
```

Listing 5.3: Auszug aus dem VHDL-Quellcode des PID-Reglers

Zu Beginn eines jeden Zyklus wird zunächst die Regeldifferenz gebildet und in Variable `e` gespeichert (Zeile 3). Mit dieser Regeldifferenz wird anschließend die Integration mit Hilfe der Trapezregel approximiert, indem zu jedem Abtastzeitpunkt die Regeldifferenz aufsummiert wird (Zeile 4). In Zeile 11 wird schließlich die Stellgröße `y_i` nach Gleichung 5.14 berechnet. Da Divisionen in VHDL sehr viele Taktzyklen benötigen (ca. 1 Rechenschritt pro Bit), wird angestrebt, diese zu vermeiden. Aus diesem Grund taucht die Abtastzeit T_a, die für die Bildung der Obersummenapproximation des Integralanteils und des Rückwärtsdifferenzenquotienten des Differentialanteils benötigt wird, im Code auch nicht auf, sondern wird bei der Bestimmung von K_i und K_d über die Ausdrücke $K_i \cdot T_a$ und K_d/T_a vorab „verrechnet“.
Als letzter Vorgang wird in Zeile 13 die Regeldifferenz für die Bildung des Differenzenquotienten im nächsten Taktzyklus gespeichert.

Die Bestimmung der Regelparameter wird dem Benutzer überlassen, der diese vom jeweiligen Einsatz abhängig neu bestimmen muss, da das System Schwankungen unterliegt (z.B. Temperatur, externe Lasten etc.) und eine feste Parametrierung zu ungenauem Regelverhalten führen würde. Die übliche Vorgehensweise wäre somit, die Parameter empirisch zu bestimmen oder, sofern möglich, die Einstellregeln nach Ziechler-Nichols, Takahashi oder das Wendetangentenverfahren heranzuziehen (Wenck, 2013, S. 15-17).

Anti-Wind-Up-Algorithmus
Bei digitalen Reglern mit integralem Anteil ist ein Anti-Wind-Up-Algorithmus unverzichtbar. Der Wind-Up-Effekt tritt auf, wenn der Regler die Regeldifferenz nicht ausgleichen kann (z.B. durch Stellgrößenbeschränkung oder Blockade der Regelstrecke), sich der Integralwert jedoch weiter aufsummiert. Verringert sich nun die Regeldifferenz, entsteht eine ungewollte Verzögerung der Stellgröße, da der Integralanteil eine gewisse Zeit benötigt, um auf einen Pegel zu kommen, der die Stellgröße wieder aus der Begrenzung bringt (nach Froschhammer, 2007, S. 44). In VHDL wird dieser Problematik mit einer Begrenzung des Integralanteils begegnet, die diesen bei Überschreitung des Wertebereichs $> \pm 15000 \mu m$ auf diesen Wert „einfriert“, wie Listing 5.3 in Zeile 6-10 entnommen werden kann.
Da der Regler, wie eingangs erwähnt, für die Geschwindigkeitsregelung eingesetzt wird, müssen Maßnahmen ergriffen werden, um ihn in die bestehende Codeumgebung zu integrieren: Zunächst wird der Encoderbaustein (Seite 70) dahingehend erweitert, dass er neben der Position auch die Geschwindigkeit ausgibt. Dafür wird eine Zählvariable *cnt* eingeführt, die alle 20ns um 1 inkrementiert wird. Wenn sie den Wert 50 erreicht hat, was $1 \mu s$ entspricht, wird vom aktuellen Positionswert der $1 \mu s$ zurückliegende Positionswert subtrahiert und der Zähler wieder auf Null gesetzt. So ergibt sich die Geschwindigkeit in $nm/\mu s$, was wiederum $\mu m/s$, also der Geschwindigkeitsvorgabe des Benutzers entspricht. Dieser Wert und der des Benutzers werden dann dem PID-Regler als Regel- und Führungsgröße zugeführt. Abbildung 5.32 zeigt die Geschwindigkeitsänderung pro Mikrosekunde bei willkürlicher Quadratursignalvorgabe und daraus resultierender Positionsänderung.

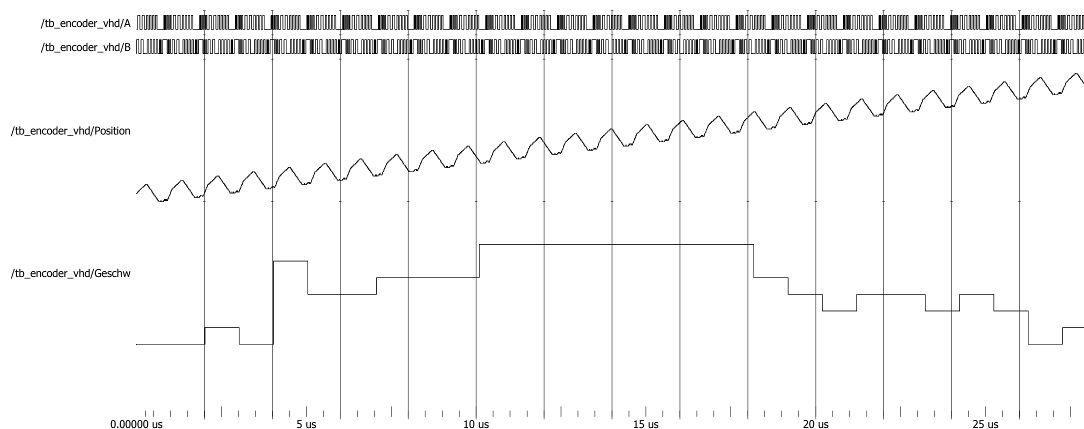


Abbildung 5.32.: Position und Geschwindigkeitsänderung in ModelSim

Eine weitere Veränderung erfahren die Wellenformgeneratoren. Diese werden um zwei zusätzliche Inputports erweitert, einen für die Stellgröße des Reglers (Geschwindigkeitsabweichung) und einen für die Ausdehnung des Piezostack (`umperwave`, vgl. Seite 56). Intern wird die Abweichung in einem neuen Prozess verarbeitet. Dabei wird zunächst geprüft, ob vom Benutzer eine Regelung gewünscht ist. Dann wird bei jeder Änderung der Stellgröße der `cnt`-Wert gemäß Formel 5.6 berechnet. Anschließend wird mit diesem Wert, je nach Laufrichtung des Motors, der Phasenakkumulator in- oder dekrementiert.
Als letztes werden die neuen I/O-Ports dem Raspberry Pi zugewiesen und das auf dem PiLC laufende C-Programm (Listing 5.1) wird um eine Abfrage für die Regelung erweitert. Der komplette Quelltext ist in Anhang C zu finden.

Zum Schluss des Kapitels sei der Vollständigkeit halber das finale Quartus II Blockschema der kompletten FPGA-Beschreibung auf Abbildung 5.33 dargestellt.

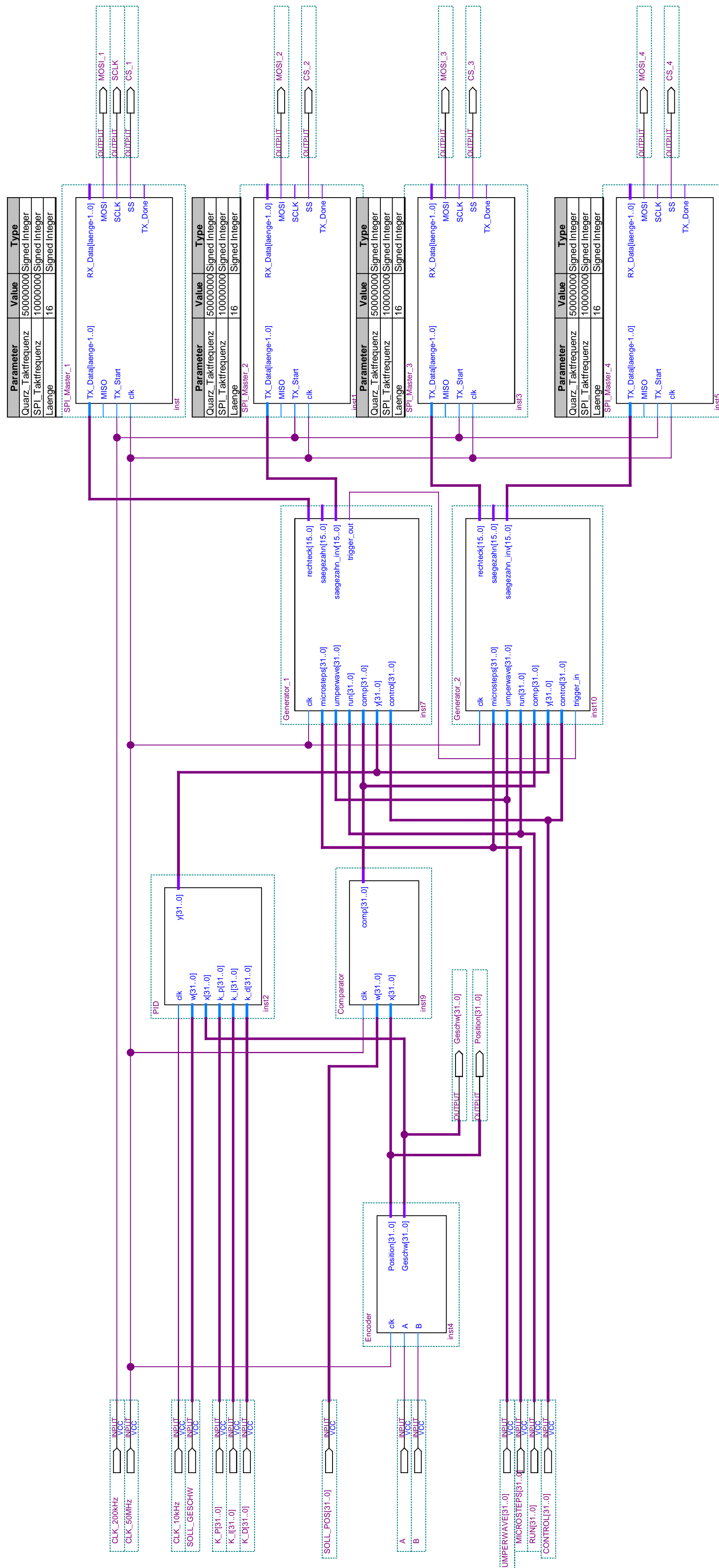


Abbildung 5.33.: Komplettes Blockschema der finalen FPGA-Beschreibung

6. Messungen und Auswertung

Dieses Kapitel führt einige Messungen auf, die sowohl an Einzelkomponenten als auch am kompletten Versuchsaufbau durchgeführt worden sind. Zunächst ein Bild des Messaufbaus an der Beamline P11:

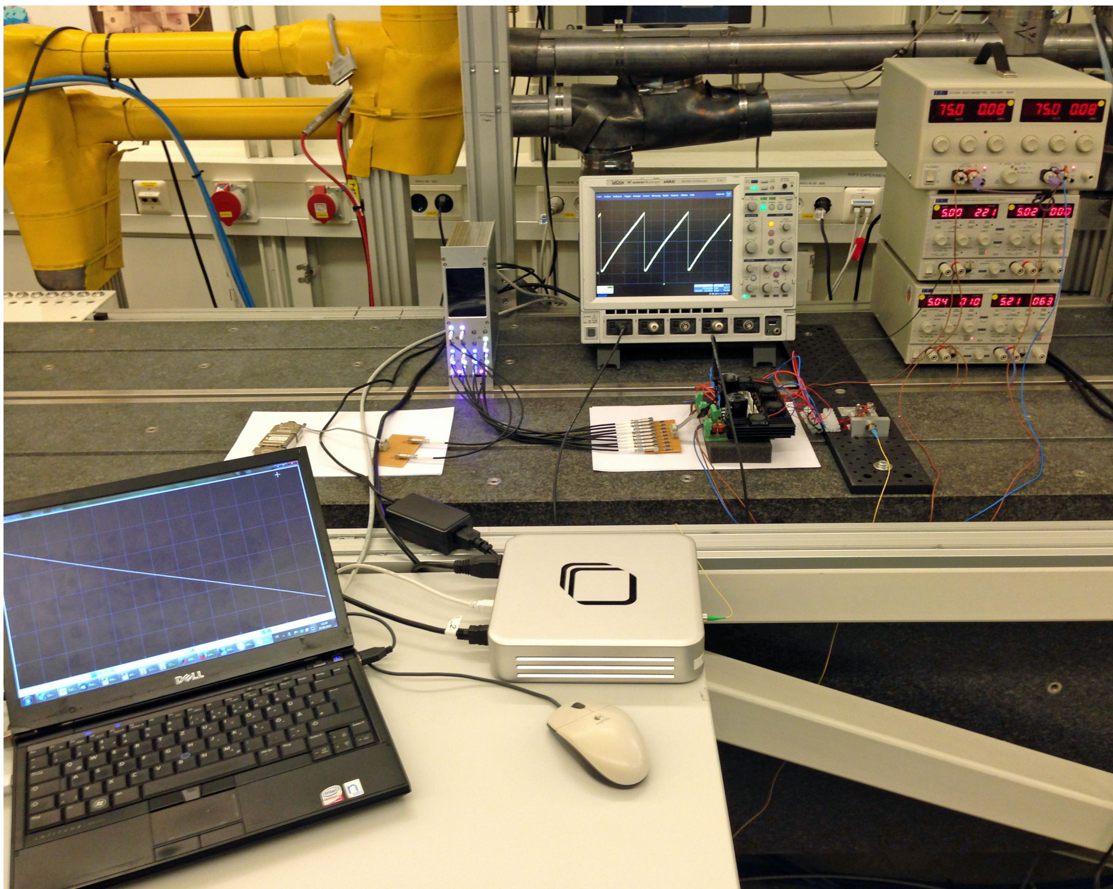


Abbildung 6.1.: Von links nach rechts: Hinten: PiLC, Oszilloskop, Spannungsversorgungen; Mitte: TTL-Adapter, Piezoverstärker, Piezomotor; Vorne: PC, Interferometer

Zum besseren Verständnis wird der Messaufbau mittels Abbildung 6.2 noch einmal im Detail erklärt: Um auf den im PiLC verbauten Raspberry Pi zugreifen zu können, muss zunächst eine Remote Desktop Verbindung vom PC zum PiLC hergestellt werden. Sodann kann das C-Programm gestartet werden, welches die Kommunikation mit dem FPGA herstellt. Die vom Benutzer eingegebenen Parameter werden über diese Schnittstelle vom FPGA verarbeitet und mittels SPI über die Ausgänge des PiLC an den Piezoverstärker weitergegeben. Dieser wandelt die digitalen in analoge Signale um, verstärkt

sie und gibt sie schließlich an den Piezomotor weiter. Das Interferometer übernimmt die Aufgabe, die Bewegung des Läufers zu erfassen und die Positionswerte über USB sowohl an den PC (Interferometersoftware) als auch als Quadratursignal an den PiLC (zur Regelung) zu senden. Das Oszilloskop dient der Messung der verschiedenen Signale und die Netzteile stellen die Spannungsversorgung für den Piezoverstärker zur Verfügung.

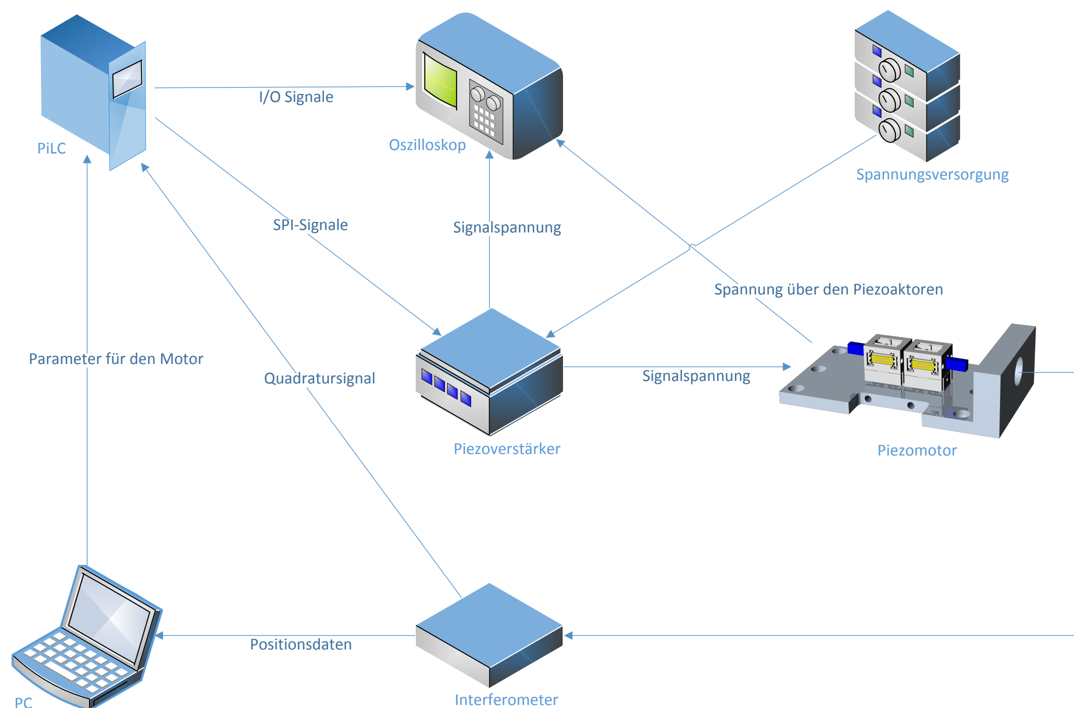


Abbildung 6.2.: Diagramm des Messaufbaus

Die nachfolgenden Messreihen werden aufgeteilt in Messungen, die den Gleichlauf und die Positioniergenauigkeit betrachten.

6.1. Messungen des Gleichlaufs

Gemäß der Vorgabe in Abschnitt 1.4 ist eine hohe Gleichlaufruhe des Motors wünschenswert, im Idealfall sogar besser als die der bisherigen Systeme. Aus diesem Grund sind am Motor Messungen mit verschiedenen Geschwindigkeiten vollzogen worden, die nachfolgend dargestellt und ausgewertet werden.

6.1.1. Vergleich Standard- und optimierte Wellenform

Die erste Messung stellt einen Vergleich zwischen der Motoransteuerung mit der Standardwellenform von Seite 38 mit der mit Simulink optimierten Wellenform von Seite 44 dar.

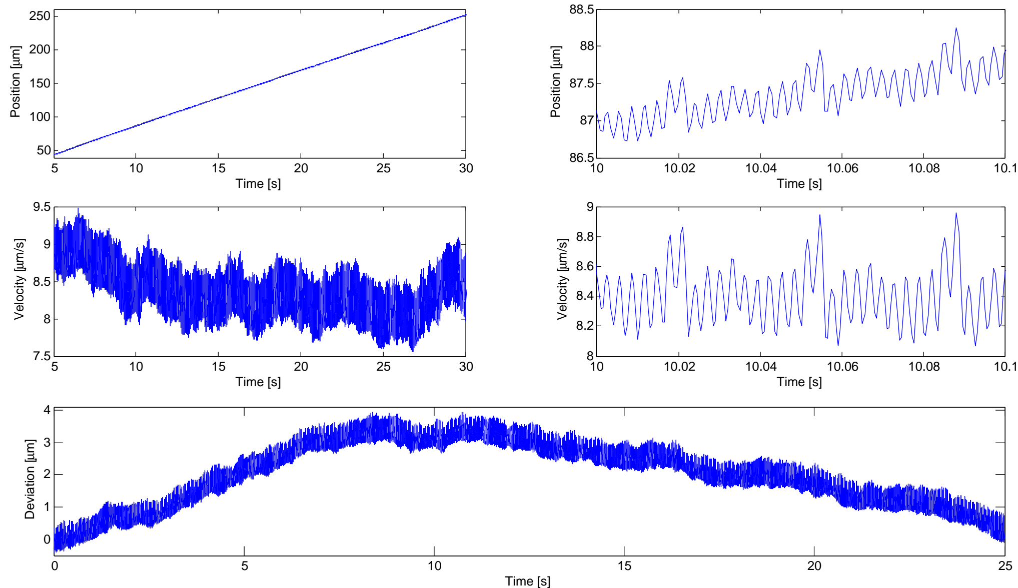


Abbildung 6.3.: Positions- (oben), Geschwindigkeits- (mitte) und Positionsabweichungsplot (unten) mit Ansteuerung durch Standardwellenform

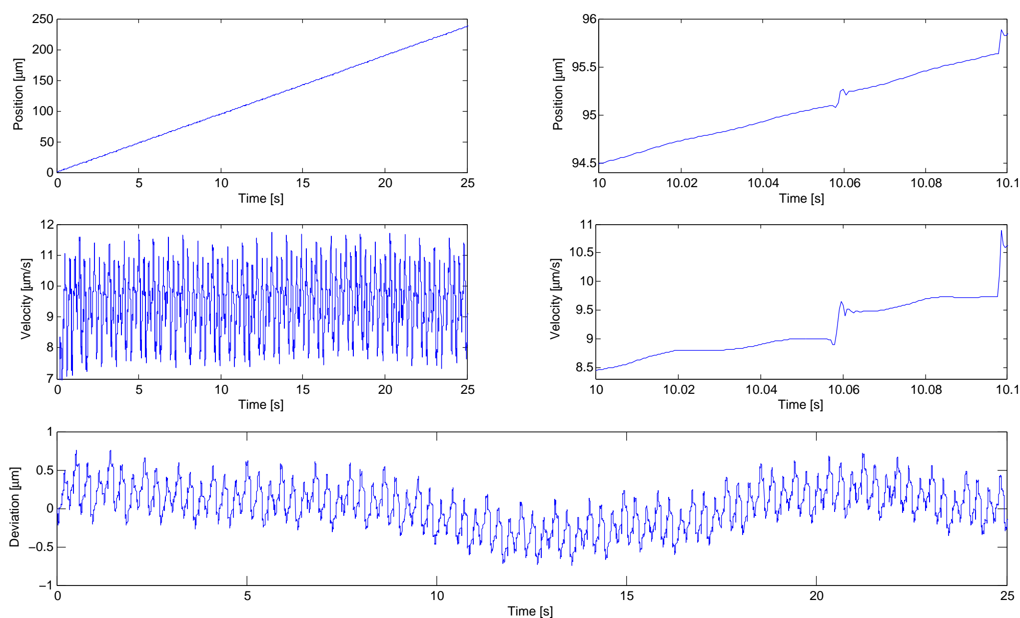


Abbildung 6.4.: Positions- (oben), Geschwindigkeits- (mitte) und Positionsabweichungsplot (unten) mit Ansteuerung durch optimierte Wellenform

Die Plots zeigen jeweils links den kompletten Zeitbereich und rechts einen Ausschnitt von $100ms$, sowie unten die Positionsabweichung über den gesamten Zeitraum. Bei beiden Messungen wird eine Geschwindigkeit von $10\mu m/s$ vorgegeben, die, wie die linken mittleren Plots darstellen, bei der optimierten Wellenform deutlich besser eingehalten wird. Die Durchschnittsgeschwindigkeit über die komplette Fahrzeit ergibt für die Standardwellenform $8.4\mu m/s$, für die optimierte $9.5\mu m/s$, was einer Verbesserung von 11% entspricht. Zurückzuführen ist dies auf die höhere Gleichlaufruhe: Vergleicht man an dieser Stelle noch einmal beide Wellenformen, so ist zu erkennen, dass bei der Standardform in einer Periode stets drei Signalflanken auf einen gemeinsamen Zeitpunkt fallen (Abbildung 6.5), bei der optimierten Wellenform hingegen wird dieser Umstand vermieden (Abbildung 6.6). Die Auswirkungen dieses Sachverhalts sind in den jeweils unteren Positionsabweichungsplots zu sehen: Liegt die maximale Abweichung bei Ansteuerung mit der normalen Wellenform noch bei $4.38\mu m$, so reduziert sie sich bei Ansteuerung mit der optmierten Wellenform auf $1.54\mu m$. Dies entspricht einer Verbesserung um Faktor ≈ 2.85.

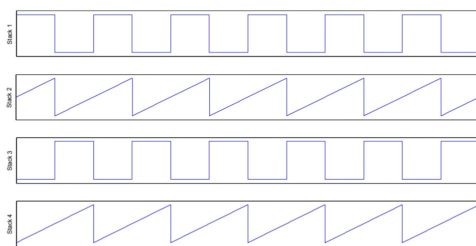


Abbildung 6.5.: Standardwellenform

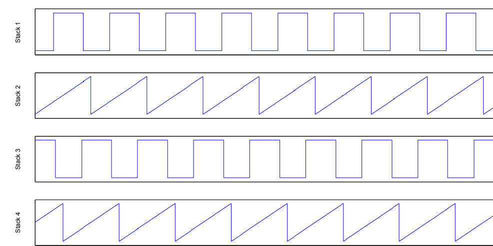


Abbildung 6.6.: Optimierte Wellenform

6.1.2. Vergleich Motorprototyp mit Piezo-LEGS-Motor

Um die Eigenschaften des Piezomotors besser einschätzen zu können, wird im Folgenden der Vergleich zu einem Piezo-LEGS-Motor durchgeführt, der in diesem Fall als Referenz herangezogen wird. Auch bei dieser Messung wird beiden Motoren eine Sollgeschwindigkeit von $10\mu m/s$ vorgegeben und sie findet nach wie vor im open-loop-Betrieb (ungeregelt) statt.

Für die Vergleichsmessung wird wieder die Positionsänderung über 25 Sekunden aufgenommen, daraus mittels Differentiation die Geschwindigkeit ermittelt und zusätzlich die Abweichung des Läufers von der idealen Position bestimmt. Die Ergebnisse werden nachfolgend tabellarisch aufgeführt.

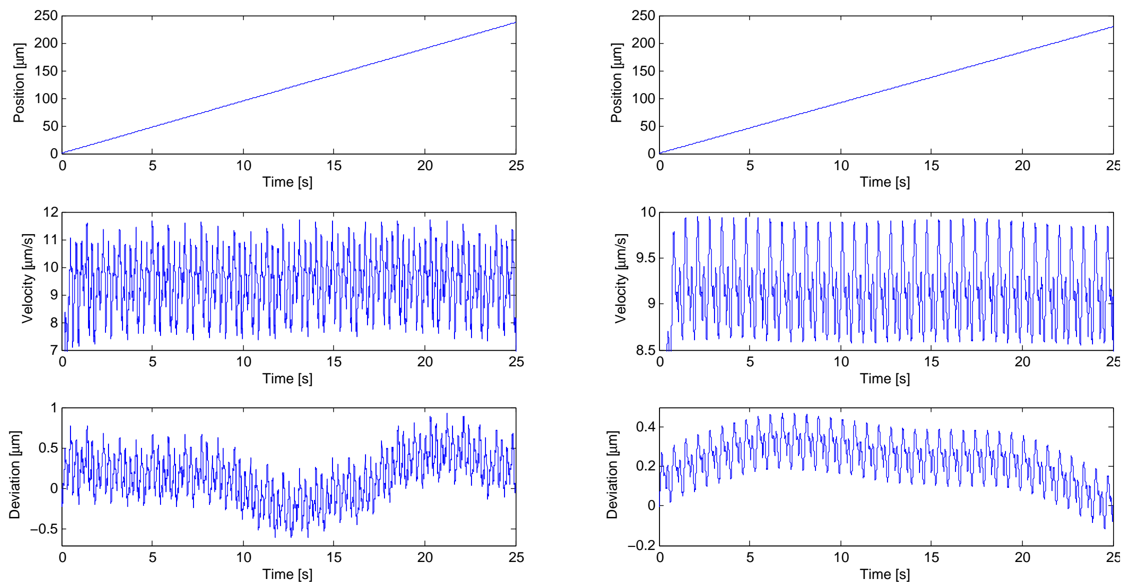


Abbildung 6.7.: Vergleich des Motorprototyp (links) mit Piezo-LEGS-Motor (rechts)

	Erwartet	Motor-prototyp	Abweich-ung [%]	Piezo-LEGS-Motor	Abweich-ung [%]
Geschw. [$\mu m/s$]	10	9.5	5	9.2	8
Position nach 25 sec. [μm]	250	237	5.2	229	8.4
Max. Abweichung der Position [μm]	0	1.54	-	0.59	-

Tabelle 6.1.: Auswertung des Piezomotorenvergleichs

Wie der Tabelle entnommen werden kann, liegt der Prototyp in puncto gleichmäßige Geschwindigkeit und der damit erreichten Endpositionen leicht vor dem Piezo-LEGS-Motor (5% zu 8% Abweichung), dieser weist jedoch eine höhere Gleichlaufruhe auf: Wie den unteren Plots von Abbildung 6.7 und Tabelle 6.1 entnommen werden kann, liegt der Abweichungsfehler zur Sollposition über den gesamten Zeitraum mit $0.59\mu m$ deutlich unter dem des Motorprototyps ($1.54\mu m$). Da die Läufer beider Motoren bei der Messung jedoch nicht durch eine zusätzliche Mechanik geführt worden sind (in der Praxis würde ein Motor so nie eingesetzt werden), müssen die Schwankungen für eine Periode bei gleichförmiger Geschwindigkeit betrachtet werden, um eine wirkliche Aussage über den Gleichlauf zu treffen. Abbildung 6.8 stellt diesen Vergleich dar, wobei der obere Plot die Positionsänderung des Läufers des Prototyps und der untere die Positionsänderung des Läufers des Piezo-LEGS-Motors über eine Wellenformperiode darstellt.

Wird die Differenz des größten und kleinsten Positionswertes gebildet, ergibt sich jeweils für den Prototyp ein Fehler von $\Delta x = 730nm$, beim Piezo-LEGS-Motor $\Delta x = 270nm$. Dies entspricht einem Verhältnis von $0.73\mu m/0.27\mu m \approx 2.7$, welches mit dem Verhältnis der Positionsabweichungswerte aus Tabelle 6.1, die über 25 Sekunden Fahrt

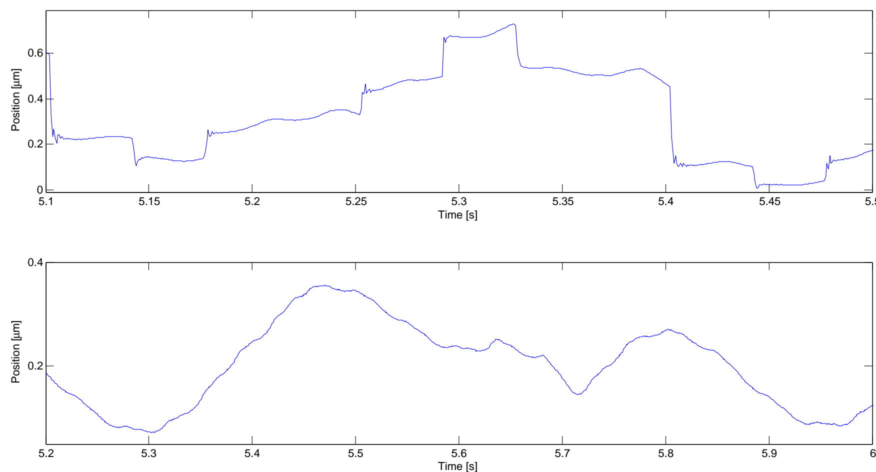


Abbildung 6.8.: Vergleich der Positionsänderung des Läufers des Motorprototyp (oben) mit der des Läufers des Piezo-LEGS-Motors (unten) für eine Periode

aufgenommen worden sind, $(1.54\mu m/0.59\mu m \approx 2.6)$ nahezu identisch ist. Es kann somit gesagt werden, dass die Positionsgenauigkeit des für diese Arbeit entwickelten Piezomotors, im Mittel um den Faktor ≈ 2.65 geringer ist als der des bisher zum Einsatz kommenden Piezo-LEGS-Motors.
Wird weiterhin die Geschwindigkeitsänderung beider Motoren betrachtet, lässt sich feststellen, dass diese beim Motorprototyp in einer Periode um $\approx 4.03\mu m/s$, beim LEGS-Motor um $\approx 1.30\mu m/s$ schwankt. Dies entspricht einem Faktor von ≈ 3.1. Um einen idealen Gleichlauf zu erreichen, müsste die Geschwindigkeitsänderung $v = 0\mu m/s$ betragen. Somit besteht hier Optimierungsbedarf, im Folgenden werden daher die Ursachen der Gleichlaufunruhe näher analysiert.

6.1.3. Ursache der Gleichlaufunruhe

Um die Schwankungen der Position besser untersuchen zu können, wird zunächst die Läuferposition bei einer gleichförmigen Geschwindigkeit von $1\mu m/s$ betrachtet.

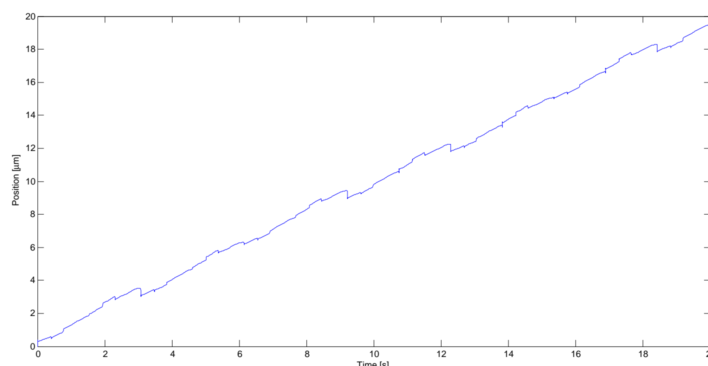


Abbildung 6.9.: Positionsänderung des Motorprototyp bei $v = 1\mu m/s$

Legt man diese Kurve zusammen mit der Wellenform in ein Koordinatensystem, so wird offensichtlich, woher die Flanken im Positionssignal stammen.

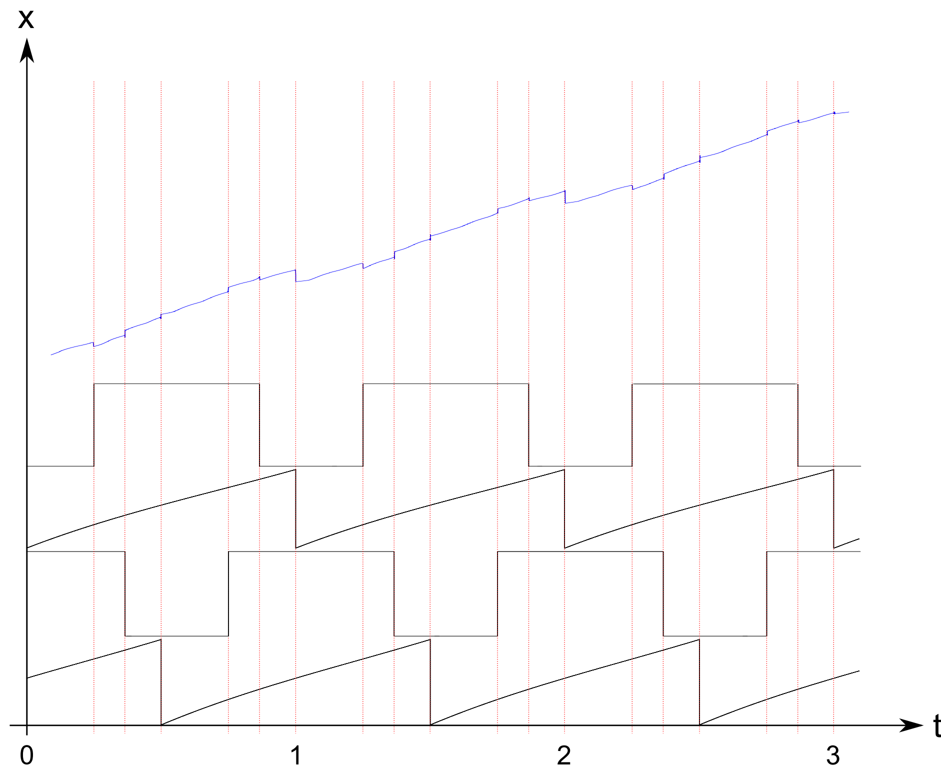


Abbildung 6.10.: Vergleich der Positionsänderung mit der Wellenform über die Zeit *t*

Wie Abbildung 6.10 entnommen werden kann, ist jede Flanke der Wellenform im Positionssignal zu erkennen, da mechanisch bedingt die Backen der Klemmeinheit selbst im entspannten Zustand immer leichte Reibung zum Läufer haben und sich weiterhin die Vorspannung der Piezostacks sehr empfindlich auf die Klemmkraft auswirkt. Somit wird jede Flanke der Wellenform mehr oder weniger stark ausgeprägt auf den Läufer übertragen (je nachdem wie stark die einzelnen Piezostacks im jeweiligen Motorblock vorgespannt sind). Diese Problematik zu beheben bedeutet in erster Linie, die Mechanik so weit zu verbessern, dass ausgeschlossen werden kann, dass die Klemmbacken im offenen Zustand den Läufer berühren. Damit wären zumindest die durch das Sägezahnsignal verursachten Flanken eliminiert. Eine weitere Maßnahme zur Verbesserung des Gleichlaufs wäre, die oberen Ecken des Rechtecksignals für die Klemmpiezos abzurunden, um so einen weicheren Übergang beim Greifen des Läufers zu erreichen. Erst wenn diese und weitere Optimierungsmaßnahmen erfolgreich durchgeführt worden sind, kann abgewogen werden, inwiefern die bereits implementierte PID-Regelung für die Geschwindigkeit noch sinnvoll ist.

6.2. Messungen der Positioniergenauigkeit

Dieser Abschnitt stellt einige Messungen vor, die zum Thema Positioniergenauigkeit gemacht worden sind. Die Genauigkeit, mit der eine gewünschte Position angefahren und gehalten wird, ist neben dem Wunsch nach einem hohe Gleichlauf ein weiterer wichtiger Punkt des Anforderungskatalogs (vgl. Abschnitt 1.4).

6.2.1. Positionsoptimierung mittels inverser Steuerung

Ergänzend zu Abschnitt 5.2.5.2 wird an dieser Stelle noch eine Messung aufgeführt, die den direkten Vergleich der Positionsänderung des Motors bei Ansteuerung mit regulärer und inverser Wellenform bei einer gleichförmigen Geschwindigkeit von $10\mu m/s$ zeigt (Abbildung 6.11).
Es bestätigt sich die bereits auf Abbildung 5.29 gezeigte positive Auswirkung der inversen Steuerung: Nach 20 Sekunden Fahrzeit liegt die Läuferposition bei $202.3\mu m$, was einer Abweichung von 1.15% entspricht (zur Sollposition von $200\mu m$). Im Gegensatz dazu liegt die Abweichung der Endposition des Läufers bei regulärer Ansteuerung bei 5.05%. Die inverse Steuerung verbessert somit das Anfahren von Positionen um ca. 4%. Ob dieser relativ geringe Wert im Verhältnis zur aufwändigen Programmierung steht, ist fraglich. Da für jede neue Wellenform zunächst eine invertierte Version dieses Signals sowie für den Richtungswechsel zwei weitere Offsetwertetabellen z.B. mit MATLAB berechnet werden müssen, findet diese Art der Ansteuerung am ehesten Einsatz, wenn eine sensorlose Regelung erfolgen soll.

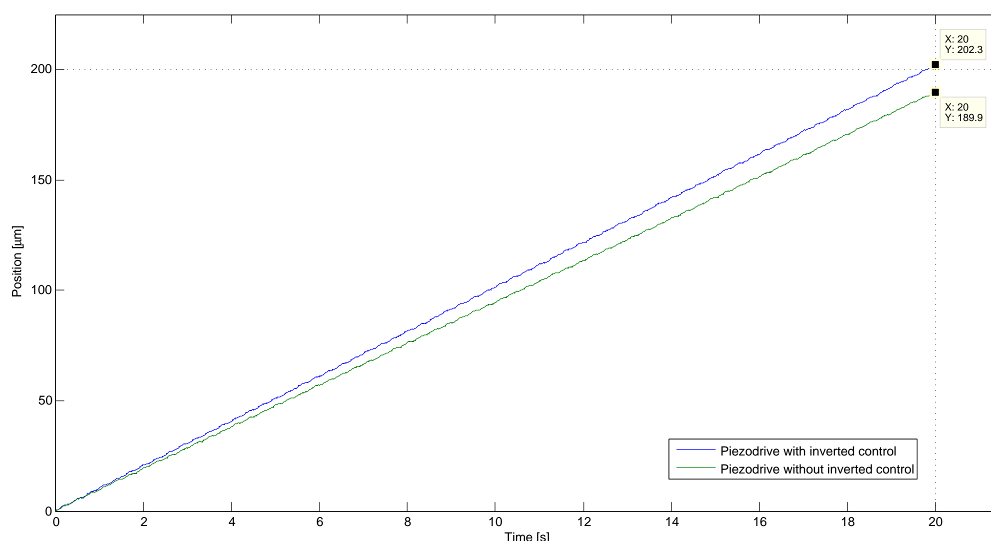


Abbildung 6.11.: Vergleich der Positionsänderung des Läufers bei Ansteuerung mit regulärer und inverser Wellenform

6.2.2. Positionsregelung

Der in Abschnitt 5.2.5.3 vorgestellte Positionsalgorithmus ist in einigen Versuchen getestet worden, von denen nachfolgend die prägnantesten vorgestellt werden. Abbildung 6.12 stellt eine Messung dar, bei der die Position $p = 6\mu m$ angefahren und gehalten wird. Bei dieser Messung kommt ein älterer Regelalgorithmus zum Einsatz, der nur ein Motorelement für die Regelung einsetzt und auch noch nicht von dem Sägezahn- auf das Dreiecksignal umschaltet.

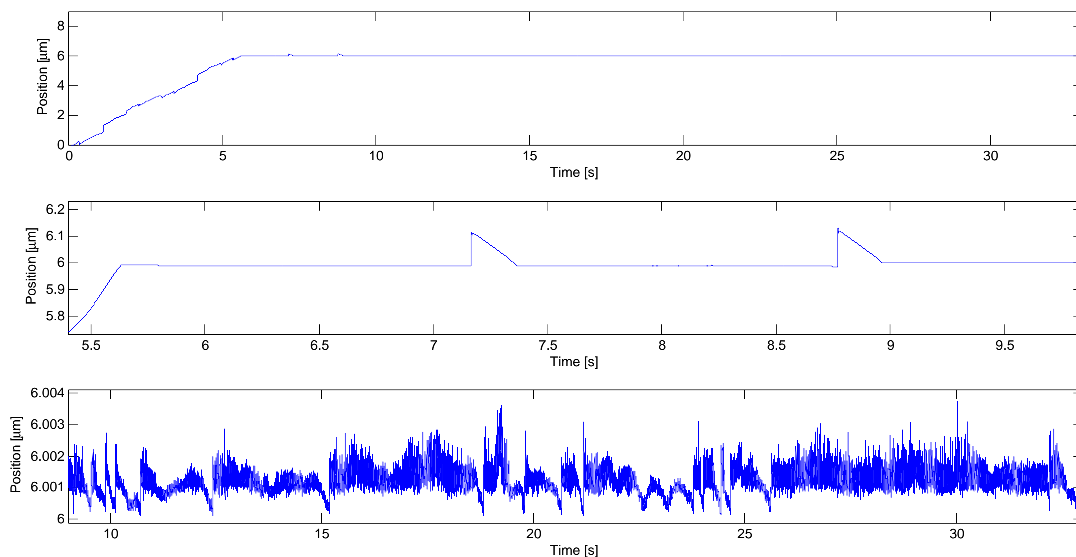


Abbildung 6.12.: Positionsregelung auf $p = 6\mu m$

Der obere Plot stellt die komplette Übersicht der Messung dar. Auf dem mittleren Plot sind die Auswirkungen des alten Regelalgorithmus zu erkennen: Die zu regelnde Position befindet sich zwischen Sekunde 7 und 9 genau im dem Bereich, in dem das Phasenregister des betroffenen Aktors und damit einhergehend seine Ausdehnung vor dem Über- bzw. in diesem Fall Unterlauf (Flanke) steht. Nach den zwei Sprüngen hat sich die Mechanik so verschoben, dass die Aktorausdehnung reicht, um die Position zu halten. Im unteren Plot schließlich ist die Positionsregelung über ca. 25 Sekunden zu sehen, die in diesem Zeitbereich auf $\pm 1.8nm$ gehalten wird. Die Abweichung zur Sollposition ist in diesem Fall auf ungenauen Abgleich der Nullposition vom Interferometer mit dem PiLC zurückzuführen.
Bei der nächsten Messung (Abbildung 6.13) wird ausgehend von $p = 0\mu m$ die Position $p = 500\mu m$ angefahren und über ca. 20 Sekunden gehalten. Dabei kommt der überarbeitet Regelalgorithmus zum Einsatz (vgl. Abschnitt 5.2.5.3).

Der obere Plot stellt wieder die komplette Übersicht der Messung dar. Für den kurzen Ausschlag bei ca. 5.5 Sekunden ist ein externer Einfluss verantwortlich (Stoß gegen den Messaufbau) und für die Betrachtung somit nicht weiter relevant. Ein Ausschnitt der eigentlichen Positionsregelung ist im unteren Plot zu sehen. Über einen Zeitbereich von

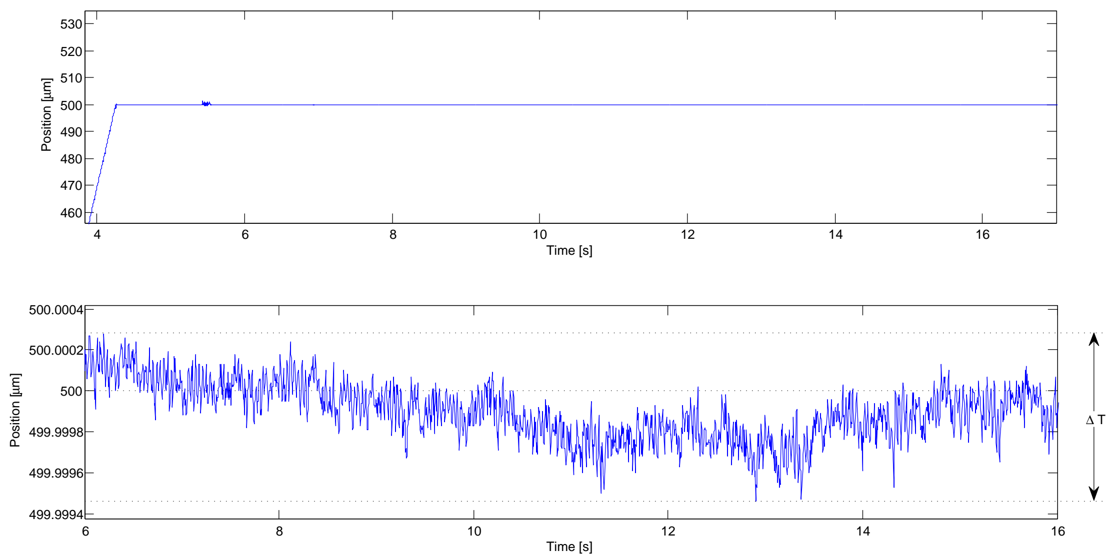


Abbildung 6.13.: Positionsregelung auf $p = 500\mu m$

10 Sekunden wird die Position mit einer Abweichung von $\Delta T = \pm 410pm$ gehalten. Fairerweise muss an dieser Stelle erwähnt werden, dass die Regelfrequenz (Frequenz, mit der das Phasenregister in- und dekrementiert wird), bei dieser Messung weit unterhalb der Frequenz von der vorherigen Messung liegt (50MHz zu 1kHz). Da die Erfassung der Regelgröße und Ausgabe des Quadratursignals durch das Interferometer sehr hochfrequent erfolgt ($f = 25MHz$), der FPGA ebenfalls die Daten mit dieser Frequenz verarbeitet und auch sonst die Latenzen in der Regelstrecke minimal sind (ein gutes Beispiel bildet hier Abbildung 6.10), muss für eine ruhigere Schaltfrequenz eine Hysterese ähnlich einem Schmitt-Trigger zugeordnet werden, die die Regelgröße zur Stellgröße zeitlich verzögert. Der Ansatz, den der Regelalgorithmus in dieser Messung verfolgt (niedrige Frequenz zum Einstellen der Stellgröße), erzielt zwar eine ruhigere, aber dafür auch sehr viel langsamere Regelung der Position. Für die Zukunft gilt es somit, einen Kompromiss aus hoher Schalt- und Stellgrößenänderungsfrequenz durch eine einstellbare Hysterese zu finden.

7. Zusammenfassung und Ausblick

Das letzte Kapitel fasst die vorliegende Arbeit zusammen und stellt in einem Ausblick mögliche Projekt betreffende Lösungsansätze für noch bestehende Problematiken dar. Nachdem in Kapitel 2 die Grundlagen des Piezoeffekts sowie der verwendete Aktor und die Eigenschaften von Piezokeramiken erläutert wurden, stellte Kapitel 3 den Piezomotor vor. Kapitel 4 widmete sich der Modellbildung sowohl eines einzelnen Aktors als auch der des gesamten Motors. Diese Erkenntnisse wurden in Kapitel 5 für die Realisierung der Ansteuerung herangezogen, bevor diese in Kapitel 6 getestet und ausgewertet worden sind. Abschnitt 7.1 fasst die Ergebnisse der Arbeit zusammen, indem sie diese den gesetzten Zielen gegenüber stellt. Die Arbeit schließt mit einem Ausblick in Abschnitt 7.2.

7.1. Zusammenfassung

Um die erreichten Ergebnisse besser bewerten zu können, werden an dieser Stelle noch einmal die gesetzten Ziele aus Abschnitt 1.4 in der Reihenfolge ihrer Priorität aufgeführt und anschließend mit den Ergebnissen der Arbeit verglichen.

Gleichlaufruhe
Einer der wichtigsten Punkte und ausschlaggebender Grund für dieses Projekt war der Wunsch einer höheren Gleichlaufruhe eines Piezomotors. Dieses Ziel konnte bis zur Fertigstellung der vorliegenden Arbeit aus folgenden, die Mechanik betreffenden Gründen, nicht erreicht werden:

- Da für das Einklemmen des Läufers zwei Piezostacks benötigt werden, müssten diese mit der gleichen Kraft vorgespannt werden. Selbiges gilt für die Vorspannung der Vorschubpiezoaktoren.
- Die Ausrichtung der Motorelemente zueinander müsste für eine ideale Fahrbewegung des Läufers absolut identisch sein.
- Die Backen der Klemmeinheit berühren selbst im spannungslosen Zustand des Piezostacks den Läufer, wodurch es zwangsläufig zu einer unsauberen Laufbewegung kommt.

Zu den mechanischen Nachteilen kommen die der Ansteuerung hinzu: Die finalen Messungen in Kapitel 6 haben gezeigt, dass die steilen Flanken in der Wellenform noch mehr dazu beitragen, dass der Gleichlauf unruhiger wird. Ein ebenfalls nicht zu vernachlässigender Punkt ist die Beeinträchtigung der Lebensdauer der Piezokeramiken. Wie in Abschnitt 5.2.4 erwähnt, ist die Leistungsaufnahme bei Signalen mit steilen Flanken und damit einhergehend die Erwärmung des Aktors bedeutend höher als bei harmonischen Signalen. Eine Überarbeitung der Wellenform ist daher unabdingbar.

Schrittauflösung

Der Bereich der Schrittauflösung (= komplette Ausdehnung eines Piezostacks) wurde laut Anforderungskatalog auf $2nm - 0.5\mu m$ festgelegt, da ursprünglich davon ausgegangen wurde, dass die Auflösung bei höheren Geschwindigkeiten gröber werden müsste. Tatsächlich werden im endgültigen Ansteuerkonzept (Konzept 2) keine Geschwindigkeiten unterschieden, vielmehr fährt der Motor bei jeder Frequenz mit der „höchsten“ Auflösung von 12 Bit. Da die Stackausdehnung variiert (Vorspannung, mechanische und externe Last), skaliert somit auch die Auflösung. Da jedoch bei der größten Ausdehnung des Stacks die geforderten $2nm$ schon unterschritten werden, kann die Auflösung bei sinkender Ausdehnung nur höher werden (z.B. $244pm$ pro Schritt bei $1\mu m$ Maximalausdehnung). Die Vorgabe konnte somit mehr als zufriedenstellend erfüllt werden.

Positioniergenauigkeit

Der dritte wichtige Punkt des Projekts betrifft die Positioniergenauigkeit, die mindestens das Niveau der bestehenden Systeme oder ein höheres erreichen sollte ($\leq 2nm$). Obwohl vorläufig die Regelung der Position über einen Zweipunktregler erfolgt, konnten mit diesem schon sehr gute Ergebnisse verzeichnet werden, die die Spezifikation des Anforderungskatalogs erfüllen (vgl. Abschnitt 6.2.2).

Geschwindigkeit und Frequenz

Die Anforderungen an die maximale Geschwindigkeit und Frequenz liegen laut Tabelle 1.2 bei $v = 5mm/s$ bzw. $f = 5kHz$. Sowohl der Piezomotor als auch der Verstärker wurden für diese Spezifikationen konzipiert und mittels Messungen konnte nachgewiesen werden, dass sie diese auch erfüllen (der Piezoverstärker sogar für Frequenzen bis $20kHz$). Es hat sich jedoch gezeigt, dass durch die verwendete Wellenform sowohl die Piezostacks als auch die Bauteile des Verstärkers bei hohen Frequenzen ($> 4kHz$) sehr stark belastet werden (Wärmeentwicklung), weshalb vorerst keine längeren Messungen mit diesen Frequenzen gemacht wurden.

Weitere Anforderungen

Weitere Punkte im Anforderungskatalog betreffen die Beschaffenheit des Motors und die Kostenreduktion. Wie bereits auf Seite 17 erwähnt, besteht bei den LEGS-Motoren die

Gefahr der Aktorbeschädigung durch zu große Krafteinwirkung. Dies kann beim vorliegenden Motorprototyp nicht eintreten, da keine direkte Verbindung zwischen Läufer und Piezoaktoren besteht.
Als letzter Punkt werden an dieser Stelle die Anschaffungs-/Herstellungskosten betrachtet, die einen nicht unerheblichen Einfluss auf die Attraktivität des Systems haben. Werden die reinen Materialkosten aller nötigen Komponenten[1] aufaddiert (ausgenommen sei das Interferometer, da auch andere weit günstigere Systeme zur Regelgrößenerfassung zum Einsatz kommen könnten), ergibt sich eine Summe von ca. 2500 Euro, die die Anforderung von < 5000 Euro zufriedenstellend einhält. Diese, im Verhältnis zum bestehenden System (vgl. Kriterium 3 auf Seite 17) dennoch hohen Kosten, kommen in erster Linie durch die für diese Arbeit entwickelte Einzelanfertigung zustande. Jedoch könnten bei höheren Stückzahlen die Kosten erheblich reduziert und somit auch die Anschaffungskosten bestehender Systeme unterschritten werden.

7.2. Ausblick

Neben vielen kleinen Verbesserungen, die die Ansteuerung betreffen, z.B. Programmierung einer grafischen Benutzerschnittstelle (GUI) zur komfortableren Bedienung des Piezomotors oder der Entwicklung eines Gehäuses für den Piezoverstärker, sind vor allem die Punkte erstrebenswert zu verbessern, die maßgeblich für die Genauigkeit des Systems verantwortlich sind. Nachfolgend werden einige dieser Punkte aufgeführt.

Mechanik
Wie schon mehrfach benannt, ist die Verbesserung oder Veränderung der Mechanik erforderlich. Eine Möglichkeit wäre hier, die Oberflächenrauheit der Greifbacken durch Läppen zu verbessern. Weiterhin könnte durch Überarbeitung der Hebelwirkung der Festkörpergelenke die Klemmkraft des Läufers erhöht werden.

Wellenform
Neben einer Überarbeitung der Wellenform wäre es ebenfalls denkbar, mehrere verschiedene Wellenformen für unterschiedliche Frequenzen in LUTs auf dem FPGA abzulegen, um sie dann entweder manuell oder automatisch auszuwählen. Ebenfalls wäre eine Parametrierung der Phasenlage der Signale zueinander wünschenswert, um beim Testen von Wellenform und Mechanik schneller auf Aussetzer der Laufbewegung reagieren zu können.

Regelung
Der in Abschnitt 5.2.5.4 vorgestellte PID-Regler für die Geschwindigkeit konnte bis zum Schluss nicht mehr real getestet werden. Da jedoch die mechanischen Probleme

[1] PiLC, Piezoverstärker mit Spannungsversorgung, Piezomotor

so gravierend auftraten, gilt es erst diese zu beheben, bevor versucht wird, mit einem Regler diese Fehler zu kompensieren. Denkbar wäre auch der Einsatz für die Positionsregelung. Bei dieser Regelung müsste jedoch erst die Auswirkung einer Schalthysterese (vgl. Abschnitt 6.2.2) getestet und als unzureichend bewertet werden, um den Implementierungsaufwand für einen PID-Regler zu rechtfertigen.

Abschließend kann gesagt werden, dass die Entwicklung des Projekts zum Zeitpunkt der Fertigstellung dieses Schriftstücks so weit fortgeschritten war, dass es durch weitere Optimierungen das Potential hat, als ernst zu nehmende Alternative zu bestehenden Systemen eingesetzt zu werden.

Literaturverzeichnis

[Analog Devices Inc. 1999] ANALOG DEVICES INC.: *A Technical Tutorial on Digital Signal Synthesis*. 1999. – URL http://www.analog.com/static/imported-files/tutorials/450968421DDS_Tutorial_rev12-2-99.pdf. – Zugriffsdatum: 02.08.2014

[Attocube Systems 2013] ATTOCUBE SYSTEMS: *AttoFPSX010 User Manual*. 2013

[Avago 2013] AVAGO: *HCPL-090J*. 2013. – URL http://www.avagotech.com/docs/AV02-0137EN. – Zugriffsdatum: 09.08.2014

[Burnett 2006] BURNETT, Colin M.L.: *SPI-Bus Example*. 2006. – URL http://engimusing.com/blog/modules-and-busses. – Zugriffsdatum: 05.08.2014

[CadSoft 2014] CADSOFT: *Eagle PCB-Software*. 2014. – URL http://www.cadsoft.de/?language=de. – Zugriffsdatum: 14.09.2014

[Duhem 1897] DUHEM, P.: Die dauernden Änderungen und die Thermodynamik. In: *Zeitschrift für Physikalische Chemie* (1897), Nr. 22, S. 543–589

[Froschhammer 2007] FROSCHHAMMER, Martin: *Codegenerierung mit Matlab/-Simulink für Mikrocontroller und FPGAs*, Fachhochschule Regensburg, Diplomarbeit, 2007. – URL http://fbim.fh-regensburg.de/~ems_labor/HSC/docu/Diplomarbeit.pdf. – Zugriffsdatum: 01.09.2014

[Gnad 2005] GNAD, Gunnar: *Ansteuerkonzept für piezoelektrische Aktoren*, Otto-von-Guericke-Universität Magdeburg, Dissertation, 2005

[Hegewald 2007] HEGEWALD, Thomas: *Modellierung des nichtlinearen Verhaltens piezokeramischer Aktoren*, Universität Erlangen-Nürnberg, Dissertation, 2007

[ITWissen 2014] ITWISSEN: *Verstärker-Klassen*. 2014. – URL http://www.itwissen.info/definition/lexikon/Verstaerker-Klasse-amplifier-class.html. – Zugriffsdatum: 11.08.2014

[Janocha 2013] JANOCHA, Hartmut: *Unkonventionelle Aktoren*. Oldenbourd Verlag, 2013

[Kennel 2014] KENNEL, Prof.Dr.-Ing. R.: *PWM*. 2014. – URL https://www.eal.ei.tum.de/fileadmin/tueieal/www/courses/UEEML/tutorial/PWM.pdf. – Zugriffsdatum: 02.08.2014

[Kessler 2014] KESSLER, Prof. Dr. R.: *Simulation der Dynamik eines Piezoelements*. 2014. – URL http://www.home.hs-karlsruhe.de/~kero0001/piezo/piezo4.pdf. – Zugriffsdatum: 23.08.2014

[Kuhnen 2001] KUHNEN, Klaus: *Inverse Steuerung piezoelektrischer Aktoren mit Hysterese-, Kriech- und Superpositionsoperatoren*, Universität des Saarlandes, Dissertation, 2001

[Lange-Janson 2012] LANGE-JANSON, Volker: *Nichtinvertierender Verstärker mit einem Operationsverstärker*. 2012. – URL http://elektronikbasteln.pl7.de/nichtinvertierender-verstaerker-mit-einem-operationsverstaerker.html. – Zugriffsdatum: 09.08.14

[León u. a. 2010] LEÓN, Fernando P. ; KIENCKE, Uwe ; JÄKEL, Holger: *Signale und Systeme*. Oldenbourg Verlag, 2010. – ISBN 3486597485

[Lineage Power 2008] LINEAGE POWER: *Austin MinilynxTM 12V SIP Non-isolated Power Modules*. 2008. – URL http://docs-europe.electrocomponents.com/webdocs/0b74/0900766b80b74739.pdf. – Zugriffsdatum: 09.08.2014

[Linear Technology 2012] LINEAR TECHNOLOGY: *LTC6090*. 2012. – URL http://cds.linear.com/docs/en/datasheet/6090fc.pdf. – Zugriffsdatum: 09.08.2014

[May 1975] MAY, J.W.G.: *Piezoelectric electromechanical translation apparatus*. August 26 1975. – URL http://www.google.com/patents/US3902084. – Zugriffsdatum: 15.06.2014. – US Patent 3,902,084

[Microchip 2010] MICROCHIP: *MCP4801/4811/4821 8/10/12-Bit Voltage Output DAC Converter*. 2010. – URL http://ww1.microchip.com/downloads/en/DeviceDoc/22244B.pdf. – Zugriffsdatum: 05.08.2014

[Microtechnology 2014] MICROTECHNOLOGY, Apex: *Power Operational Amplifiers*. 2014. – URL http://www.apexanalog.com/products/power-operational-amplifiers/. – Zugriffsdatum: 11.09.2014

[Miller 2009] MILLER, Lothar: *Einfacher SPI-Master / Mode 0*. 2009. – URL http://www.lothar-miller.de/s9y/categories/45-SPI-Master. – Zugriffsdatum: 05.08.2014

[Muthig 2010] MUTHIG, Armin: *A class of its Own - Audio-Endstufen*. 2010. – URL http://blog.steinigke.de/wissenswertes/a-class-of-its-own-audio-endstufen/. – Zugriffsdatum: 11.08.2014

[National Instruments 2013] NATIONAL INSTRUMENTS: *Wie funktionieren FPGAs?* 2013. – URL www.ni.com/white-paper/6983/de/pdf. – Zugriffsdatum: 02.08.2014

[Noliac 2014] NOLIAC: *Drivers for piezoelectric actuators*. 2014. – URL http://www.noliac.com/Files/Billeder/02%20Standard/Drivers/NDR_leaflet.pdf. – Zugriffsdatum: 11.09.2014

[NTE Electronics Inc. 2014] NTE ELECTRONICS INC.: *NTE2388 MOSFET*. 2014. – URL http://www.nteinc.com/specs/2300to2399/pdf/nte2388.pdf. – Zugriffsdatum: 03.08.2014

[Palacherla 1997] PALACHERLA, Amar: *Using PWM to Generate Analog Output*. Microchip Technology Inc. 1997. – URL http://ww1.microchip.com/downloads/en/AppNotes/00538c.pdf. – Zugriffsdatum: 29.07.2014

[Panasonic 2014] PANASONIC: *Aluminum Electrolytic Capacitors*. 2014. – URL http://docs-europe.electrocomponents.com/webdocs/1272/0900766b81272d7d.pdf. – Zugriffsdatum: 14.08.2014

[Physik Instrumente 2009] PHYSIK INSTRUMENTE: *Grundlagen der Nanostelltechnik*. 2009. – URL http://www.physikinstrumente.de/de/pdf_extra/2009_PI_Katalog_Grundlagen_der_Nanostelltechnik-Tutorial.pdf. – Zugriffsdatum: 09.07.2014

[Physik Instrumente 2013] PHYSIK INSTRUMENTE: *PICMA® Stack Multilayer-Piezoaktoren*. 2013. – URL http://www.physikinstrumente.de/de/pdf/P882_Datenblatt.pdf. – Zugriffsdatum: 09.07.2014

[Physik Instrumente 2014] PHYSIK INSTRUMENTE: *Piezoverstärker*. 2014. – URL http://www.physikinstrumente.de/de/produkte/piezo_steuerung/index.php. – Zugriffsdatum: 11.09.2014

[PiezoMotor 2014a] PIEZOMOTOR: *Installation Guidelines for Piezo LEGS*. 2014. – URL http://www.piezomotor.com/app/content/uploads/150099_Installation_Guidelines_for_Piezo_LEGS.pdf. – Zugriffsdatum: 15.06.2014

[PiezoMotor 2014b] PIEZOMOTOR: *Piezo LEGS Linear 6N*. 2014. – URL http://www.piezomotor.com/app/content/uploads/150010_LL10.pdf. – Zugriffsdatum: (15.06.2014)

[Piezosystem Jena 2013] PIEZOSYSTEM JENA: *Piezofibel*. 2013. – URL http://www.piezosystem.de/fileadmin/redakteure/bilder/Sonstige/Piezofibel/Piezofibel_Deutsch_Website.pdf. – Zugriffsdatum: 09.07.2014

[Piezosystem Jena 2014] PIEZOSYSTEM JENA: *Piezoverstärker*. 2014. – URL http://www.piezosystem.de/piezo_nanopositionierung/piezo_verstaerker_steuerelektronik/. – Zugriffsdatum: 11.09.2014

[Preisach 1935] PREISACH, F.: Über die magnetische Nachwirkung. In: *Zeitschrift für Physik* (1935), Nr. 94, S. 277–302

[Recom International 2010] RECOM INTERNATIONAL: *INNOLINE DC/DC-Converter*. 2010. – URL http://www.recom-international.com/pdf/Innoline/Rxx-B.pdf. – Zugriffsdatum: 09.08.2014

[RP Foundation 2014] RP FOUNDATION: *Raspberry Pi*. 2014. – URL http://www.raspberrypi.org/. – Zugriffsdatum: 06.08.2014

[Schwarz 2003] SCHWARZ, Andreas: *Bau eines digitalen Funktionsgenerators*, Gymnasium Alexandrinum Coburg, Facharbeit, 2003. – URL http://www.mikrocontroller.net/wikifiles/5/51/Dds.pdf. – Zugriffsdatum: 02.08.2014

[Schwerdtfeger 2000] SCHWERDTFEGER, Martin: *SPI - Serial Peripheral Interface*. 2000. – URL http://www.mct.de/faq/spi.html. – Zugriffsdatum: 05.08.2014

[Spitzbart und Zink 2013] SPITZBART, Tobias ; ZINK, Horst: *PiLC - Raspberry PI Logic Controller Datasheet*. 2013

[Stiebel u. a. 2014] STIEBEL, Chr. ; WÜRTZ, Th. ; H.JANOCHA: *Leistungsverstärker für piezoelektrische Aktoren*. 2014. – URL http://www.lpa.uni-saarland.de/pdf/Elektronik.PDF. – Zugriffsdatum: 14.06.2014

[Terasic Technologies Inc. 2012] TERASIC TECHNOLOGIES INC.: *DE0-Nano User Manual*. 2012. – URL http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=593&PartNo=4. – Zugriffsdatum: 02.08.2014

[Texas Instruments 2013] TEXAS INSTRUMENTS: *LM4702 Audio Power Amplifier*. 2013. – URL http://www.ti.com/lit/ds/symlink/lm4702.pdf. – Zugriffsdatum: 11.08.2014

[Vishay 2014a] VISHAY: *Power MOSFET IRF620*. 2014. – URL http://www.vishay.com/docs/91027/sihf620.pdf. – Zugriffsdatum: 14.08.2014

[Vishay 2014b] VISHAY: *Power MOSFET IRF9620*. 2014. – URL http://www.vishay.com/docs/91082/91082.pdf. – Zugriffsdatum: 14.08.2014

[Visintin 1994] VISINTIN, A.: *Differential Models of Hysteresis*. Springer-Verlag, Berlin, 1994

[Wang 2006] WANG, Qi: *Piezoaktoren für Anwendungen im Kraftfahrzeug, Messtechnik und Modellierung*, Ruhr-Universität Bochum, Dissertation, 2006. – URL http://www-brs.ub.ruhr-uni-bochum.de/netahtml/HSS/Diss/WangQi/diss.pdf. – Zugriffsdatum: 06.07.2014

[Wenck 2013] WENCK, Prof. Dr.-Ing. F.: *Digitale Regelungstechnik*. Vorlesungsscript 7. Entwurf quasikontinuierlicher Regler, 2013

A. Schaltpläne

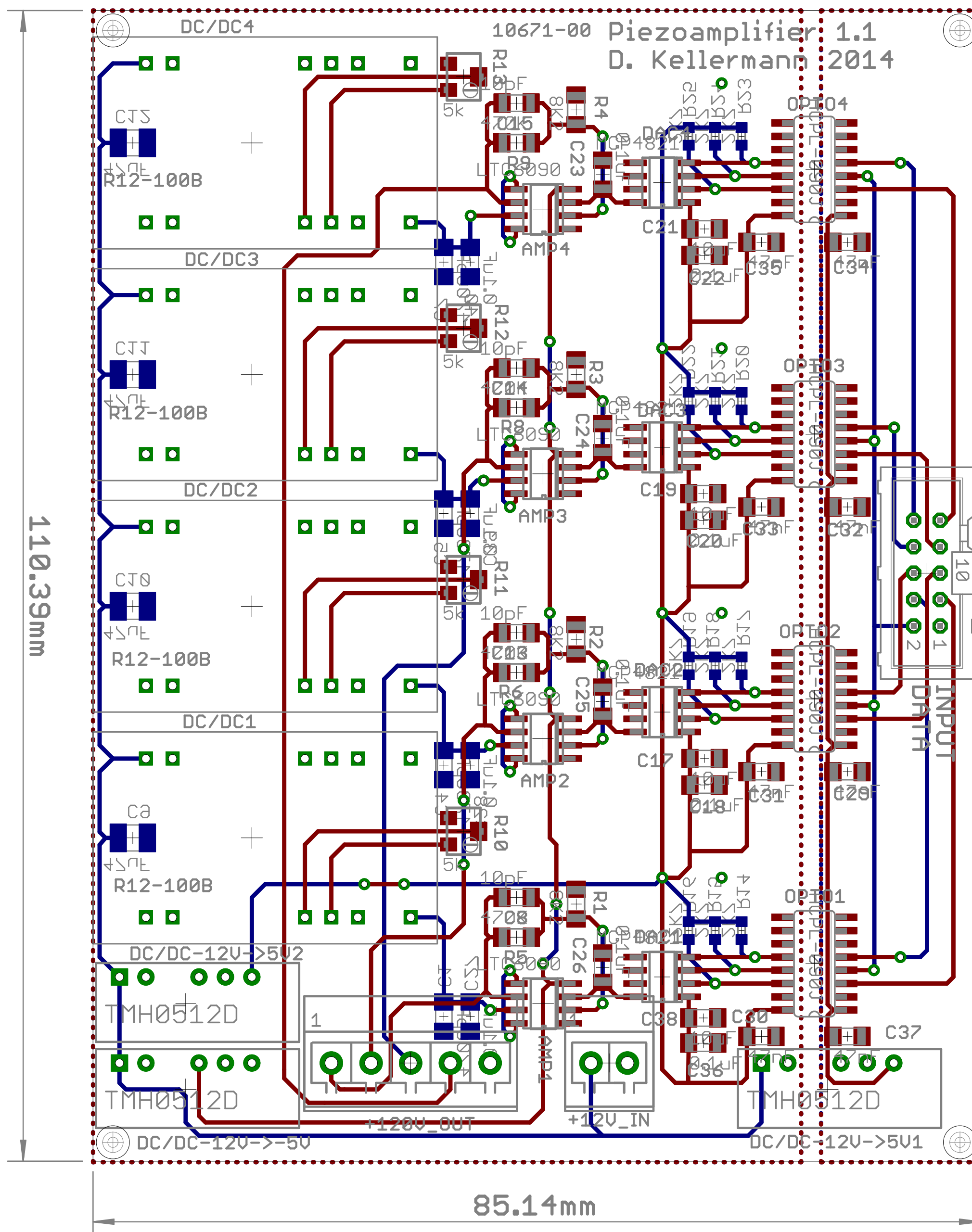


Abbildung A.1.: Platinenlayout Piezoverstärker Version 1.1

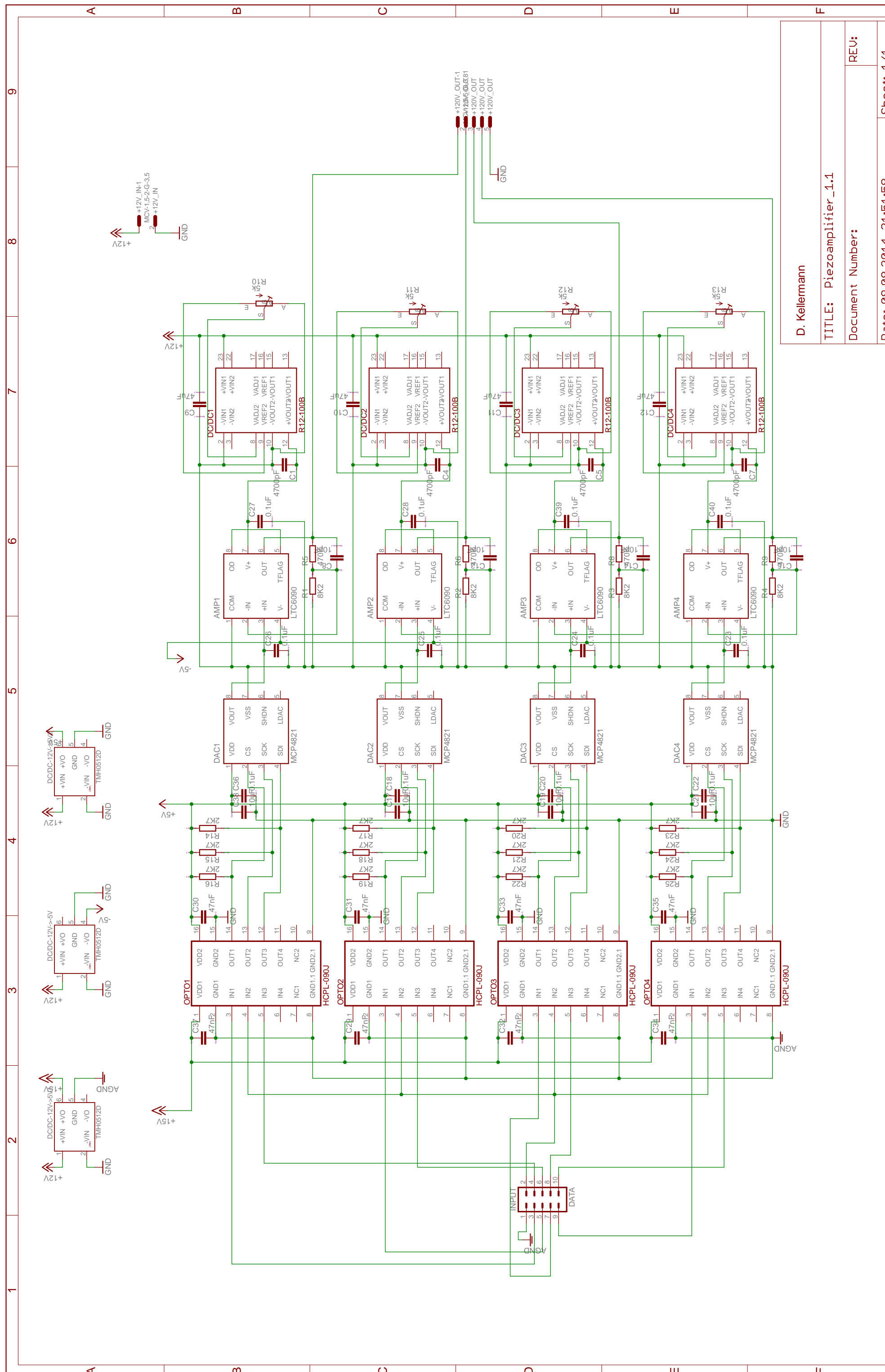


Abbildung A.2.: Schaltplan Piezoverstärker Version 1.1

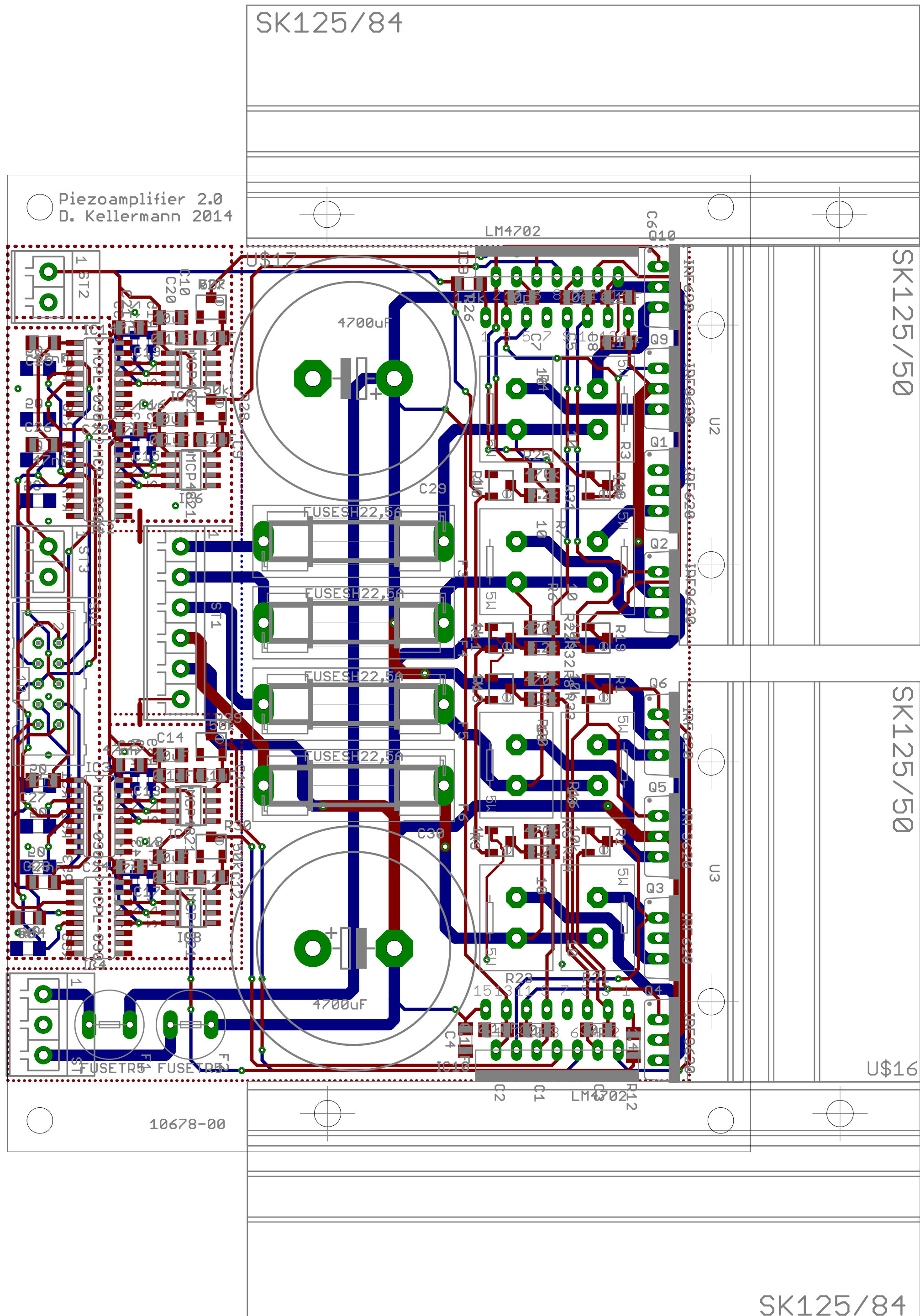


Abbildung A.3.: Platinenlayout Piezoverstärker Version 2.0

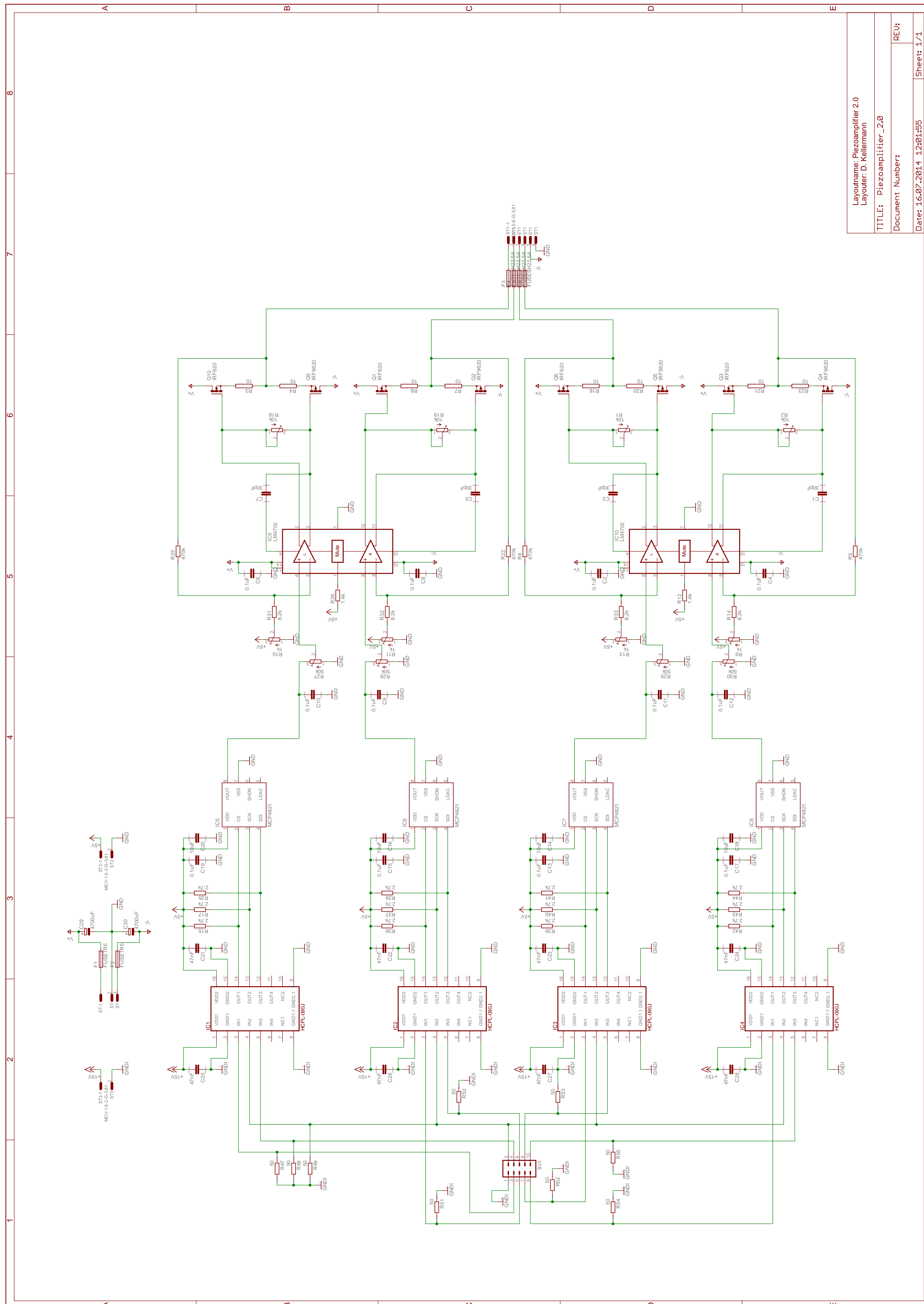


Abbildung A.4.: Schaltplan Piezoverstärker Version 2.0

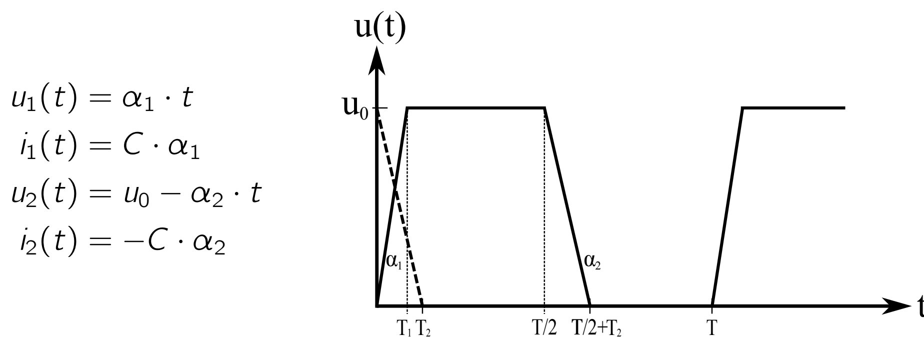
B. Rechnungen

B.1. Mittlere Leistung

Schriftliche Überprüfung der mittleren Leistung die vom Piezoaktor in einer Periode umgesetzt wird für:

B.1.1. Rechtecksignal

$$
\begin{aligned}
u_1(t) &= \alpha_1 \cdot t \\
i_1(t) &= C \cdot \alpha_1 \\
u_2(t) &= u_0 - \alpha_2 \cdot t \\
i_2(t) &= -C \cdot \alpha_2
\end{aligned}
$$



$$
\begin{aligned}
\overline{\left|p(t)\right|} &= \frac{1}{T}\int_0^T \left|u(t)\cdot i(t)\right| \mathrm{d}t \\
&= \frac{1}{T}\left(\int_0^{T_1} (C\cdot {\alpha_1}^2 \cdot t)\,\mathrm{d}t + \int_0^{T_2} (u_0 \cdot C \cdot \alpha_2 - C\cdot {\alpha_2}^2 \cdot t)\,\mathrm{d}t\right) \\
&= \frac{1}{T}\left(C\cdot {\alpha_1}^2 \cdot \left[\frac{t^2}{2}\right]_0^{T_1} + u_0 \cdot C \cdot \alpha_2 \cdot [t]_0^{T_2} - C\cdot {\alpha_2}^2 \cdot \left[\frac{t^2}{2}\right]_0^{T_2}\right)
\end{aligned}
$$

mit:

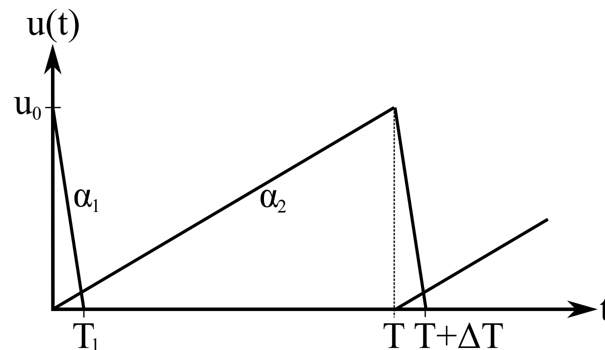
$u_0 = 120V$, $C = 210nF$, $\alpha_1 = \frac{120V}{20\mu s}$, $\alpha_2 = \frac{120V}{50\mu s}$, $T = 1ms$, $T_1 = 20\mu s$, $T_2 = 50\mu s$

ergibt sich für die mittlere Leistung:

$$\overline{|p(t)|} = 3.02W$$

B.1.2. Sägezahnsignal

$$u_1(t) = u_0 - \alpha_1 \cdot t$$
$$i_1(t) = -C \cdot \alpha_1$$
$$u_2(t) = \alpha_2 \cdot t$$
$$i_2(t) = C \cdot \alpha_2$$



$$\begin{aligned}\overline{|p(t)|} &= \frac{1}{T}\int_0^T \left|u(t)\cdot i(t)\right| \mathrm{d}t \\ &= \frac{1}{T}\left(\int_0^{T_1}(u_0\cdot C\cdot\alpha_1 - C\cdot{\alpha_1}^2\cdot t)\,\mathrm{d}t + \int_0^T (C\cdot{\alpha_2}^2\cdot t)\,\mathrm{d}t\right) \\ &= \frac{1}{T}\left(u_0\cdot C\cdot\alpha_1\cdot[t]_0^{T_1} - C\cdot{\alpha_1}^2\cdot\left[\frac{t^2}{2}\right]_0^{T_1} + C\cdot{\alpha_2}^2\cdot\left[\frac{t^2}{2}\right]_{T_1}^{T}\right)\end{aligned}$$

mit:

$u_0 = 120V$, $C = 210nF$, $\alpha_1 = \frac{120V}{30\mu s}$, $\alpha_2 = \frac{120V}{1ms}$, $T = 1ms$, $T_1 = 30\mu s$

ergibt sich für die mittlere Leistung:

$$\overline{|p(t)|} = 1.51W$$

C. Quellcode

Listing C.1: C-Quellcode des Raspberry Pi zum Steuern des FPGAs

```
#include <stdio.h>
#include <unistd.h>      //Used for UART
#include <termios.h>
#include <math.h>
#include <stdlib.h>
#include <pthread.h>
#include <errno.h>
#include <string.h>
#include "bcm2835.h"
#include "spi.h"
#define TTL_Input   0
#define NIM_Input   1
#define TTL_Output  3
#define NIM_Output  7
#define OR      0
#define NOR     1
#define AND     2
#define NAND    3
#define DELAY   1
#define I0_1_IN_Register  1
#define I0_1_OUT_Register   2
#define I0_2_IN_Register  3
#define I0_2_OUT_Register   4
#define I0_3_IN_Register  5
#define I0_3_OUT_Register   6
#define I0_4_IN_Register  7
#define I0_4_OUT_Register   8
#define I0_5_IN_Register  9
#define I0_5_OUT_Register   10
#define I0_6_IN_Register  11
#define I0_6_OUT_Register   12
#define I0_7_IN_Register  13
#define I0_7_OUT_Register   14
#define I0_8_IN_Register  15
#define I0_8_OUT_Register   16
#define I0_9_IN_Register  17
#define I0_9_OUT_Register   18
#define I0_10_IN_Register   19
#define I0_10_OUT_Register  20
#define I0_11_IN_Register   21
#define I0_11_OUT_Register  22
#define I0_12_IN_Register   23
#define I0_12_OUT_Register  24
#define I0_13_IN_Register   25
#define I0_13_OUT_Register  26
#define I0_14_IN_Register   27
#define I0_14_OUT_Register  28
#define I0_15_IN_Register   29
#define I0_15_OUT_Register  30
#define I0_16_IN_Register   31
#define I0_16_OUT_Register  32
#define FPGA_IO_DIR_Register  33
```

```
#define FPGA_IO_STATUS_Register 34
#define  FPGA_Power     RPI_V2_GPIO_P1_15

float speed = 0;
float umperwave = 0;
int microsteps = 0;
int control= 0;
int run = 0;
float freq = 0;
int posfromfpga = 0;
int speedfromfpga = 0;
float position = 0;
int reset = 1;
int kp = 0;
int ki = 0;
int kd = 0;

unsigned char Version=0;
unsigned long FPGA_IO_DIR=0;
unsigned char FPGA_IO_DIR_Array[17];
unsigned long FPGA_IO_STATUS=0;
unsigned char FPGA_IO_STATUS_Array[17];
unsigned char NIMTTL_STATUS_Array[17];


//kbhit()-Befehl einbinden
int kbhit(void);
int kbhit(void){
  struct termios term, oterm;
  int fd = 0;
  int c = 0;
  tcgetattr(fd, &oterm);
  memcpy(&term,&oterm,sizeof(term));
  term.c_lflag = term.c_lflag & (!ICANON);
  term.c_cc[VMIN] = 0;
  term.c_cc[VTIME] = 1;
  tcsetattr(fd,TCSANOW,&term);
  c = getchar();
  tcsetattr(fd,TCSANOW,&oterm);
  if(c!=-1)
  ungetc(c,stdin);
  return((c!=-1)?1:0);
}
//getch()-Befehl einbinden
int getch();
int getch()
{
  static int ch = -1, fd = 0;
  struct termios neu, alt;
  fd = fileno(stdin);
  tcgetattr(fd,&alt);
  neu = alt;
  neu.c_lflag &= ~(ICANON|ECHO);
  tcsetattr(fd,TCSANOW,&neu);
  ch = getchar();
  tcsetattr(fd,TCSANOW,&alt);
  return ch;
}

void Thread_1 (void *name)
{
printf("Thread_1 gestartet\n");

unsigned char  Blink=0;
unsigned int t250ms=0;
```

```
unsigned int  test=0;

for(unsigned char i=1;;++i){
// FPGA_IO_STATUS_Register auslesen
    FPGA_IO_STATUS=FPGA(FPGA_IO_STATUS_Register,0);
// FPGA_IO_STATUS_Register informationen in das FPGA_IO_STATUS_Array speichern
  for (unsigned char i=0;i!=16; i++){
    FPGA_IO_STATUS_Array[i]=((FPGA_IO_STATUS)>>i);
    FPGA_IO_STATUS_Array[i]=((FPGA_IO_STATUS_Array[i]&0b00000001)+48);
    FPGA_IO_STATUS_Array[i+1]=0;
    }
    t250ms++;
  if (t250ms==500){
    t250ms=0;
    test++;

  // Irgendeine Taste drücken unterbricht das Programm für die Abfrage
  if (kbhit()){
    printf("--------------------------------------------\n\n");

    printf("Run (0 = Stop, 1 = Forward, 2 = Backward): ");
    scanf("%d",&run);
    printf("You entered:%d.\n\n",run);

    if (run != 0){

      control = 0;

      printf("Speed [ums/s] : ");
      scanf("%f",&speed);
      printf("You entered:%f um/s.\n\n",speed);

      printf("Micrometer per Waveform [um]: ");
      scanf("%f",&umperwave);
      printf("You entered:%f um/wfm.\n\n",umperwave);

      freq = speed/umperwave;

      printf("Frequency: %f Hz.\n\n",freq);

      microsteps = (freq * pow(2,32))/50000000;
      printf("microsteps: %d.\n\n",microsteps);
    }
    else{
      microsteps = 0;
      reset = 1;
      position = 0;

      printf("Control (0 = No, 1 = Yes): ");
      scanf("%d",&control);
      printf("You entered:%d.\n\n",control);

      if (control == 1){
        printf("Speed [ums/s] : ");
        scanf("%f",&speed);
        printf("You entered:%f um/s.\n\n",speed);

        printf("Micrometer per Waveform [um]: ");
        scanf("%f",&umperwave);
        printf("You entered:%f um/wfm.\n\n",umperwave);

        printf("Position [um]: ");
        scanf("%f",&position);
        printf("You entered position: %f .\n\n",position);
        position = position * 1000;
```

```
        printf("Proportional gain: ");
        scanf("%i",&kp);
        printf("You entered: %i .\n\n",kp);

        printf("Integral gain: ");
        scanf("%i",&ki);
        printf("You entered: %i .\n\n",ki);

        printf("Derivative gain: ");
        scanf("%i",&kd);
        printf("You entered: %i .\n\n",kd);

      }
      else{
        position = 0;
        microsteps = 0;
        reset = 1;
        kp = 0;
        ki = 0;
        kd = 0;
      }

    }
    printf("-------------------------------------------\n\n");

    char c = getch();
  }

  FPGA(IO_1_IN_Register, microsteps);
  FPGA(IO_2_IN_Register, run);
  FPGA(IO_3_IN_Register, reset);
  FPGA(IO_4_IN_Register, control);
  FPGA(IO_5_IN_Register, position);
  FPGA(IO_6_IN_Register, speed);
  FPGA(IO_7_IN_Register, umperwave);
  FPGA(IO_8_IN_Register, kp);
  FPGA(IO_9_IN_Register, ki);
  FPGA(IO_10_IN_Register, kd);

  posfromfpga=FPGA(IO_7_OUT_Register,0);
  speedfromfpga=FPGA(IO_8_OUT_Register,0);


  printf("Ist-Position: %f \n",posfromfpga * 0.001);
  printf("Soll-Position: %f \n",position * 0.001);

  printf("Ist-Geschwindigkeit: %f \n",speedfromfpga);
  printf("Soll-Geschwindigkeit: %f \n",speed);

  printf("%i :ID\n",FPGA(35,0));
// LEDS je nach zustand beschreiben
  for (unsigned char i=0;i!=16; i++){
    if  (NIMTTL_STATUS_Array[i]=='1'){
      if (FPGA_IO_DIR_Array[i]=='1'){
        if (FPGA_IO_STATUS_Array[i]=='1'){

          }
        if (FPGA_IO_STATUS_Array[i]=='0'){
          LedFront (i+1,30,30,3);
          }
        }
      if (FPGA_IO_DIR_Array[i]=='0'){
        if (FPGA_IO_STATUS_Array[i]=='1'){
          LedFront (i+1,0,30,0);
```

```
          }
        if (FPGA_IO_STATUS_Array[i]=='0'){
          LedFront (i+1,30,0,30);
          }
        }
      }
    }
// LED 17 und 18 togglen lassen
    Blink=!Blink;
  if (Blink==0){
    //((FPGA(I0_1_IN_Register,(test<<4)+(DELAY<<3)+AND);
    //((printf("%d :test\n",(test<<4)+(DELAY<<3)+AND);
    LedFront (17,30,0,0);
    LedFront (18,0,0,30);
  }
  if (Blink==1){
    //FPGA(I0_1_IN_Register,(test<<4)+AND);
    //printf("%d :test\n",(test<<4)+AND);
    LedFront (17,0,0,30);
    LedFront (18,30,0,0);
    }
  }
usleep(1000);
  }
}

int main() {
  printf("Prog. Start\n");

  bcm2835_init();
  spi_init();
// FPGA Reset
  bcm2835_gpio_fsel(FPGA_Power, BCM2835_GPIO_FSEL_OUTP);
  bcm2835_gpio_write(FPGA_Power, LOW);
  usleep(1000000UL);
  bcm2835_gpio_write(FPGA_Power, HIGH);
  usleep(1000000UL);
// Front LEDs ausschalten
  for (unsigned char i=1;i!=19; i++){
    LedFront (i,0,0,0);
    }
  usleep(100);
// NIMTTL Karten abfragen
  for (unsigned char i=0;i!=16; i++){
    //Version=NIMTTL(i,0x09,0x00,0x00);
    Version=NIMTTL(i+1,0x09,0x00,0x00);
    Version=NIMTTL(i+1,0x09,0x01,0x00);
    if (Version==8){
      printf("%lu: %lu NIMTTL Modul Version 1\n" ,i+1,NIMTTL(i+1,0x09,0x01,0x00));
      NIMTTL_STATUS_Array[i]='1';
      NIMTTL(i+1,0,0,0xF8);
      NIMTTL(i+1,0x06,0,0x07);
      NIMTTL(i+1,0x09,0,TTL_Input);
      }
    else{
      printf("%lu: %lu No Moldul insert\n" ,i+1,NIMTTL(i+1,0x09,0x01,0x00));
      NIMTTL_STATUS_Array[i]='0';
      }
    NIMTTL_STATUS_Array[i+1]=0;
    }
// FPGA ID Register abfragen

// FPGA IO Richtungs Register abfragen
  FPGA_IO_DIR=FPGA(FPGA_IO_DIR_Register,0);
  FPGA_IO_DIR=FPGA(FPGA_IO_DIR_Register,0);
```

```
    for (unsigned char i=0;i!=16; i++){
    FPGA_IO_DIR_Array[i]=((FPGA_IO_DIR)>>i);
    FPGA_IO_DIR_Array[i]=((FPGA_IO_DIR_Array[i]&0b00000001)+48);
    FPGA_IO_DIR_Array[i+1]=0;
    }
  printf("%s :FPGA IO Dir\n",FPGA_IO_DIR_Array);

// NIMTTL Karten konfigurieren <-NUR TTL, NIM TTL umschaltung fehlt noch
  for (unsigned char i=0;i!=16; i++){
    if (FPGA_IO_DIR_Array[i]=='0'){
      NIMTTL(i+1,0x09,0,TTL_Input);
      printf("%lu: TTL_Input\n" ,i+1);
      }
    if (FPGA_IO_DIR_Array[i]=='1'){
      NIMTTL(i+1,0x09,0,TTL_Output);
      printf("%lu: TTL_Output\n" ,i+1);
      }
    }
// Threads starten
   pthread_t t1;
  if(pthread_create(&t1, NULL, (void *)&Thread_1, (void *)"Thread 1") != 0)
    {
      fprintf(stderr, "Fehler bei Thread 1......\n");
      exit(0);
    }
    pthread_join(t1, NULL);
    return 0;
}
```

Listing C.2: VHDL-Quellcode der PWM

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity PWM is
port(
      clk_PWM   : in  STD_LOGIC;
      pwmvalue  : in  STD_LOGIC_VECTOR (11 downto 0);
      pwmout    : out STD_LOGIC := '0');
end PWM;
architecture Behavioral of PWM is
signal cnt  : integer range 0 to 4095 := 0;
signal dir  : std_logic := '0'; -- 0=up
begin
   process begin
      wait until rising_edge(clk_PWM);
      -- Zähler
      if(dir = '0') then -- up
      if (cnt < 4095) then
        cnt <= cnt+1;
      else
        dir <= '1';
        cnt <= cnt-1;
         end if;
    else               -- down
      if (cnt > 0) then
        cnt <= cnt-1;
         else
        dir <= '0';
        cnt <= cnt+1;
         end if;
      end if;
-- Vergleicher
      if (cnt>=to_integer(unsigned(pwmvalue))) then
        pwmout <= '0';
      else
        pwmout <= '1';
      end if;
  end process;
end Behavioral;
```

Listing C.3: VHDL-Quellcode des Encoders

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Encoder is
port(
      clk       : in STD_LOGIC;
      A         : in STD_LOGIC;
      B         : in STD_LOGIC;
      Position  : out STD_LOGIC_VECTOR(31 downto 0);
      Geschw    : out STD_LOGIC_VECTOR(31 downto 0));
end Encoder;
architecture Behavioral of Encoder is
  type zustaende is (Z00, Z01, Z11, Z10);
  signal z      : zustaende := Z00;
  signal p      : integer := 0;
  signal g      : integer := 0;
  signal p_alt  : integer := 0;
  signal i      : std_logic_vector(1 downto 0);
  signal e      : std_logic_vector(1 downto 0);
  signal cnt    : integer range 0 to 50 := 0;
begin
  process begin
    wait until rising_edge(clk);
      i <= A & B;  -- Zusammenfassen der Eingänge A und B
         e <= i;
   end process;
  process      -- Weiterschalten und Zählen
  variable cu, cd : std_logic := '0';
  begin
    wait until rising_edge(clk);
      cu := '0'; -- lokale Werte
      cd := '0';
       case z is
        when Z00 => if    (e = "01") then z <= Z01; cu := '1';
                elsif (e = "10") then z <= Z10; cd := '1';
                end if;
        when Z01 => if    (e = "11") then z <= Z11; cu := '1';
                elsif (e = "00") then z <= Z00; cd := '1';
                end if;
        when Z11 => if    (e = "10") then z <= Z10; cu := '1';
                elsif (e = "01") then z <= Z01; cd := '1';
                end if;
        when Z10 => if    (e = "00") then z <= Z00; cu := '1';
                elsif (e = "11") then z <= Z11; cd := '1';
                end if;
       end case;
      if    (cu='1') then
        p <= p+1;
      elsif (cd='1') then
        p <= p-1;
      end if;
   --Geschwindigkeit alle 1us ermitteln
      if (cnt = 50) then
        g <= ((p) - (p_alt));
        p_alt <= p;
        cnt <= 0;
      else
        cnt <= cnt+1;
      end if;
  end process;
  Position <= std_logic_vector(to_signed(p,32)); -- Position ausgeben
  Geschw <= std_logic_vector(to_signed(g,32)); -- Geschwindigkeit ausgeben
end Behavioral;
```

Listing C.4: VHDL-Quellcode des PID-Reglers

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use ieee.math_real.all;

entity PID is
port(
      clk : in std_logic;
      w   : in signed(31 downto 0)  := (others => '0');
      x   : in signed(31 downto 0)  := (others => '0');
      k_p : in signed(31 downto 0)  := (others => '0');
      k_i : in signed(31 downto 0)  := (others => '0');
      k_d : in signed(31 downto 0)  := (others => '0');
      y   : out signed(31 downto 0) := (others => '0')
  );
end PID;
architecture Behavioral of PID is
signal e     : signed(31 downto 0) := (others => '0');
signal e_alt : signed(31 downto 0) := (others => '0');
signal e_sum : signed(31 downto 0) := (others => '0');
signal y_i   : signed(31 downto 0) := (others => '0');
signal y_ii  : signed(31 downto 0) := (others => '0');
begin
process begin
  wait until rising_edge(clk);
    e  <= w - x;     --Regeldifferenz
    e_sum <= e_sum + e; --Summenbildung
    --Anti-Windup
    if e_sum < x"FFFFC568" then -- e_sum < -15000
      e_sum <= x"FFFFC568";
    elsif e_sum > x"00003A98" then --e_sum > 15000
      e_sum <= x"00003A98";
    end if;
    y_i <= resize(((signed(k_p)*signed(e))+(signed(k_i)*signed(e_sum))+(signed(k_d)*(
        signed(e)-signed(e_alt)))),32);
    e_alt <= e;
end process;
process begin --Stellgrößenbeschränkung zwischen 0 und 15000 um
  wait until rising_edge(clk);
    if y_i > x"00003A98" then -- y_i > 15000
      y_ii <= x"00003A98";
    elsif y_i < x"00000000" then -- y_i < 0
      y_ii <= x"00000000";
    else
      y_ii <= y_i;
    end if;
end process;
y <= y_ii;
end Behavioral;
```

Listing C.5: VHDL-Quellcode des Zweipunktreglers

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use ieee.math_real.all;

entity Comparator is
port(
      clk   : in std_logic;
      w     : in signed(31 downto 0) := (others => '0');
      x     : in signed(31 downto 0) := (others => '0');
      comp  : out signed(31 downto 0) := (others => '0')
  );
end Comparator;
architecture Behavioral of Comparator is
signal e  : signed(31 downto 0) := (others => '0');
begin
  process begin
    wait until rising_edge(clk);
    e  <= w - x;     --Differenz
  end process;
comp <= e;
end Behavioral;
```

Listing C.6: VHDL-Quellcode des Funktionsgenerators 1

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Generator_1 is
port(
      clk           : in std_logic; -- Systemclock (50 MHz)
      microsteps    : in unsigned(31 downto 0) := (others => '0');
      umperwave     : in unsigned(31 downto 0) := (others => '0');
      run           : in unsigned(31 downto 0) := (others => '0'); -- 0 = steht, 1 =
          links, 2 = rechts
      comp          : in signed(31 downto 0) := (others => '0');
      y             : in signed(31 downto 0) := (others => '0');
      control       : in unsigned(31 downto 0) := (others => '0'); -- Regelung: 0 =
          nein, 1 = ja
      rechteck      : out std_logic_vector(15 downto 0) := (others => '0');
      saegezahn     : out std_logic_vector(15 downto 0) := (others => '0');
      saegezahn_inv : out std_logic_vector(15 downto 0) := (others => '0');
      trigger_out   : out std_logic := '1'
      );

end Generator_1;

architecture bhv of Generator_1 is
type saw_load_array_t is array (0 to 4095) of integer range 0 to 4095;
constant saw_load_array_c : saw_load_array_t :=
(--WERTETABELLE AUS PLATZGRÜNDEN ENTFERNT
);

type saw_unload_array_t is array (0 to 4095) of integer range 0 to 4095;
constant saw_unload_array_c : saw_unload_array_t :=
(--WERTETABELLE AUS PLATZGRÜNDEN ENTFERNT
);

type offset_load_unload_array_t is array (0 to 4095) of integer range 0 to 4095;
constant offset_load_unload_array_c : offset_load_unload_array_t :=
(--WERTETABELLE AUS PLATZGRÜNDEN ENTFERNT
);

type offset_unload_load_array_t is array (0 to 4095) of integer range 0 to 4095;
constant offset_unload_load_array_c : offset_unload_load_array_t :=
(--WERTETABELLE AUS PLATZGRÜNDEN ENTFERNT
);

type zustaende is (Z0, Z1);
signal z                  : zustaende := Z0;
signal index              : integer range 0 to 4095;
signal newindex           : integer range 0 to 4095 := 0;
signal phase              : unsigned(31 downto 0) := (others => '0');
signal saegezahn_inv_u    : unsigned(11 downto 0) := (others => '0');
signal saegezahn_u        : unsigned(11 downto 0) := (others => '0');
signal rechteck_u         : unsigned(11 downto 0) := (others => '0');
signal dir                : std_logic := '0'; -- 0 = up
signal cnt_e              : integer range 0 to 4095 := 0;
signal flag               : std_logic := '0';
signal trigger            : std_logic := '0';
signal microsteps_control : unsigned(31 downto 0) := (others => '0');

begin
  process begin --Microsteps bei Geschwindigkeitsregelung neu berechnen
    wait until rising_edge(clk);
      if control = x"00000001" then
        microsteps_control <= resize(((to_integer(y)/umperwave)*x"FFFFFFFF")/x"02FAF080
            ",32);
```

```
      end if;
  end process;
  process
  variable l_index : integer range 0 to 4095;
  begin
    wait until rising_edge(clk);
----------------------------ohne Regelung (control = 0)-------------------------
    if control = x"00000000" then
      flag <= '0';
      if run = x"00000001" then
        z <= Z0;
        if newindex = 0 then
          phase <= (phase + microsteps);
        elsif  newindex  /= 0 then
          phase <= to_unsigned(newindex,12) & x"00000";
        end if;
      elsif run = x"00000002" then
        z <= Z1;
          if newindex = 0 then
            phase <= (phase - microsteps);
          elsif  newindex  /= 0 then
            phase <= to_unsigned(newindex,12) & x"00000";
          end if;
      end if;
----------------------------mit Regelung (control = 1)-------------------------
    elsif control = x"00000001" then
      if comp < x"00000000" and comp < x"FFFFFE0C" then -- comp < 500nm
        flag <= '0';
        trigger <= '0';
        z <= Z0;
        l_index := index; -- Index merken für spätere Berechnung
        if newindex = 0 then
          phase <= (phase + microsteps_control);
        elsif  newindex  /= 0 then
          phase <= to_unsigned(newindex,12) & x"00000";
        end if;
      elsif comp > x"00000000" and comp > x"000001F4" then -- comp > 500nm
        flag <= '0';
        trigger <= '0';
        z <= Z1;
        l_index := index; -- Index merken für spätere Berechnung
        if newindex = 0 then
          phase <= (phase - microsteps_control);
        elsif  newindex  /= 0 then
           phase <= to_unsigned(newindex,12) & x"00000";
        end if;
      else -- Regelung ab <= 500nm
        if z = Z0 then --Rechtslauf
          if (l_index + 1536) <= 4096 then -- Prüfen, ob Piezo die Sollposition
              erreichen könnte (1536 = Ergebnis aus Dreisatz)
            trigger <= '1';
            flag <= '1';
            if comp < x"00000000" then -- up
              if phase(31 downto 20) <= 4095 then
                phase <= (phase + x"00002710");
              end if;
            elsif comp > x"00000000" then -- down
              if phase(31 downto 20) >= 0 then
                phase <= (phase - x"00002710");
              end if;
            end if;
          else
            trigger <= '0';
            flag <= '1';
          end if;
```

```
      elsif z = Z1 then -- Linkslauf
        if (l_index - 1536) >= 0 then -- Prüfen, ob Piezo die Sollposition erreichen
            könnte (1536 = Ergebnis aus Dreisatz)
          trigger <= '1';
          flag <= '1';
          if comp < x"00000000" then -- up
            if phase(31 downto 20) <= 4095 then
              phase <= (phase + x"00002710");
            end if;
          elsif comp > x"00000000" then -- down
            if phase(31 downto 20) >= 0 then
              phase <= (phase - x"00002710");
            end if;
          end if;
        else
          trigger <= '0';
          flag <= '1';
        end if;
      end if;
    end if;
  end if;
  end process;
 process -- invertierte Steuerung
 variable dir : std_logic := '0';
 begin
  wait until falling_edge(clk);
  case z is
    when Z0 => --Rechtslauf
      if (dir = '0') then
        saegezahn_inv_u <= to_unsigned( saw_load_array_c (index), saegezahn_inv_u'
            length);
        newindex <= 0;
        dir := '0';
      elsif (dir = '1') then
        newindex <= index - offset_unload_load_array_c(index);
        dir := '0';
      end if;
    when Z1 => --Linkslauf
      if (dir = '1') then
        saegezahn_inv_u <= to_unsigned( saw_unload_array_c (index), saegezahn_inv_u'
            length);
        newindex <= 0;
        dir := '1';
      elsif (dir = '0') then
        newindex <= index + offset_load_unload_array_c(index);
        dir := '1';
      end if;
  end case;
 end process;
  process begin -- Rechtecksignal für die Klemmpiezos je nach Modus an/abschalten
  wait until rising_edge(clk);
  if flag = '0' then
    if (saegezahn_u >= x"400" and saegezahn_u <= x"DDD")then
      rechteck_u <= x"FFF";
    else
      rechteck_u <= x"000";
    end if;
  elsif flag = '1' and trigger = '1' then
      rechteck_u <= x"FFF";
  elsif flag = '1' and trigger = '0' then
      rechteck_u <= x"000";
  end if;
  end process;

 index <= to_integer(phase(31 downto 20));
```

```
    saegezahn_u <= phase(31 downto 20);

  -- Outputs umwandeln
  process begin
    wait until rising_edge(clk);
    if flag = '0' then
      saegezahn_inv <= "0011" & std_logic_vector(saegezahn_inv_u);
      saegezahn <= "0011" & std_logic_vector(saegezahn_u);
      rechteck  <= "0011" & std_logic_vector(rechteck_u);
    elsif flag = '1' then
      saegezahn_inv <= "0011" & std_logic_vector(saegezahn_u);
      rechteck  <= "0011" & std_logic_vector(rechteck_u);
    end if;
  end process;
  trigger_out <= trigger; -- Anderen Generator abschalten, wenn nötig
end bhv;
```

Listing C.7: VHDL-Quellcode des Funktionsgenerators 2

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Generator_2 is
port(
      clk           : in std_logic; -- Systemclock (50 MHz)
      microsteps    : in unsigned(31 downto 0) := (others => '0');
      umperwave     : in unsigned(31 downto 0) := (others => '0');
      run           : in unsigned(31 downto 0) := (others => '0'); -- 0 = steht, 1 =
          links, 2 = rechts
      comp          : in signed(31 downto 0) := (others => '0');
      y             : in signed(31 downto 0) := (others => '0');
      control       : in unsigned(31 downto 0) := (others => '0'); -- Regelung: 0 =
          nein, 1 = ja
      trigger_in    : in std_logic := '0';
      rechteck      : out std_logic_vector(15 downto 0) := (others => '0');
      saegezahn     : out std_logic_vector(15 downto 0) := (others => '0');
      saegezahn_inv : out std_logic_vector(15 downto 0) := (others => '0')
      );

end Generator_2;

architecture bhv of Generator_2 is

type saw_load_array_t is array (0 to 4095) of integer range 0 to 4095;
constant saw_load_array_c : saw_load_array_t :=
(--WERTETABELLE AUS PLATZGRÜNDEN ENTFERNT
);

type saw_unload_array_t is array (0 to 4095) of integer range 0 to 4095;
constant saw_unload_array_c : saw_unload_array_t :=
(--WERTETABELLE AUS PLATZGRÜNDEN ENTFERNT
);

type offset_load_unload_array_t is array (0 to 4095) of integer range 0 to 4095;
constant offset_load_unload_array_c : offset_load_unload_array_t :=
(--WERTETABELLE AUS PLATZGRÜNDEN ENTFERNT
);

type offset_unload_load_array_t is array (0 to 4095) of integer range 0 to 4095;
constant offset_unload_load_array_c : offset_unload_load_array_t :=
(--WERTETABELLE AUS PLATZGRÜNDEN ENTFERNT
);

type zustaende is (Z0, Z1);
signal z                  : zustaende := Z0;
signal index              : integer range 0 to 4095;
signal newindex           : integer range 0 to 4095 := 0;
signal phase              : unsigned(31 downto 0) := x"7FFFFFFF";
signal saegezahn_inv_u    : unsigned(11 downto 0) := (others => '0');
signal saegezahn_u        : unsigned(11 downto 0) := (others => '0');
signal rechteck_u         : unsigned(11 downto 0) := (others => '0');
signal dir                : std_logic := '0'; -- 0 = up
signal cnt_e              : integer range 0 to 4095 := 0;
signal flag               : std_logic := '0';
signal trigger            : std_logic := '0';
signal microsteps_control : unsigned(31 downto 0) := (others => '0');

begin
    process begin --Microsteps bei Geschwindigkeitsregelung neu berechnen
    wait until rising_edge(clk);
      if control = x"00000001" then
```

```
      microsteps_control <= resize(((to_integer(y)/umperwave)*x"FFFFFFFF")/x"02FAF080
          ",32);
    end if;
end process;
process
variable l_index : integer range 0 to 4095;
begin
  wait until rising_edge(clk);
  if control = x"00000000" then -- ohne Regelung
    flag <= '0';
    if run = x"00000001" then
      z <= Z0;
      if newindex = 0 then
        phase <= (phase + microsteps);
      elsif  newindex  /= 0 then
        phase <= to_unsigned(newindex,12) & x"00000";
      end if;
    elsif run = x"00000002" then
      z <= Z1;
        if newindex = 0 then
          phase <= (phase - microsteps);
        elsif  newindex  /= 0 then
          phase <= to_unsigned(newindex,12) & x"00000";
        end if;
    end if;
  elsif control = x"00000001" then -- mit Regelung
    if comp < x"00000000" and comp < x"FFFFFE0C" then -- comp < 500nm
      flag <= '0';
      trigger <= '0';
      z <= Z0;
      l_index := index; -- Index merken für spätere Berechnung
      if newindex = 0 then
        phase <= (phase + microsteps);
      elsif  newindex  /= 0 then
        phase <= to_unsigned(newindex,12) & x"00000";
      end if;
    elsif comp > x"00000000" and comp > x"000001F4" then -- comp > 500nm
      flag <= '0';
      trigger <= '0';
      z <= Z1;
      l_index := index; -- Index merken für spätere Berechnung
      if newindex = 0 then
        phase <= (phase - microsteps);
      elsif  newindex  /= 0 then
         phase <= to_unsigned(newindex,12) & x"00000";
      end if;
    else -- Regelung ab 500nm Restdifferenz
      if z = Z0 then --Rechtslauf
        if ((l_index + 1536) <= 4096) and (trigger_in /= '1') then -- Prüfen, ob
            Piezo die Sollposition erreichen könnte (1536 = Ergebnis aus Dreisatz)
          trigger <= '1';
          flag <= '1';
          if comp < x"00000000" then -- up
            if phase(31 downto 20) <= 4095 then
              phase <= (phase + x"000D1B71");
            end if;
          elsif comp > x"00000000" then -- down
            if phase(31 downto 20) >= 0 then
              phase <= (phase - x"000D1B71");
            end if;
          end if;
        else
          trigger <= '0';
        end if;
      elsif z = Z1 then -- Linkslauf
```

```
        if ((l_index - 1536) >= 0) and (trigger_in /= '1') then -- Prüfen, ob Piezo
            die Sollposition erreichen könnte (1536 = Ergebnis aus Dreisatz)
          trigger <= '1';
          flag <= '1';
          if comp < x"00000000" then -- up
            if phase(31 downto 20) <= 4095 then
              phase <= (phase + x"000D1B71");
            end if;
          elsif comp > x"00000000" then -- down
            if phase(31 downto 20) >= 0 then
              phase <= (phase - x"000D1B71");
            end if;
          end if;
        else
          trigger <= '0';
        end if;
      end if;
    end if;
  end if;
  end process;
 process
 variable dir : std_logic := '0';
 begin
  wait until falling_edge(clk);
  case z is
    when Z0 =>
      if (dir = '0') then
        saegezahn_inv_u <= to_unsigned( saw_load_array_c (index), saegezahn_inv_u'
            length);
        newindex <= 0;
        dir := '0';
      elsif (dir = '1') then
        newindex <= index-offset_unload_load_array_c(index);
        dir := '0';
      end if;
    when Z1 =>
      if (dir = '1') then
        saegezahn_inv_u <= to_unsigned( saw_unload_array_c (index), saegezahn_inv_u'
            length);
        newindex <= 0;
        dir := '1';
      elsif (dir = '0') then
        newindex <= index+offset_load_unload_array_c(index);
        dir := '1';
      end if;
  end case;
 end process;
  process begin -- Rechtecksignal für die Klemmpiezos je nach Modus an/abschalten
  wait until rising_edge(clk);
  if (flag = '0' and control = x"00000000") then
    if (saegezahn_u >= x"400" and saegezahn_u <= x"DDD")then
      rechteck_u <= x"FFF";
    else
      rechteck_u <= x"000";
    end if;
  elsif flag = '1' and trigger = '1' then
      rechteck_u <= x"FFF";
  elsif (comp >= x"FFFFFE0C" and trigger = '0') or (comp <= x"000001F4" and trigger =
       '0') then
      rechteck_u <= x"000";
  end if;
  end process;
 index <= to_integer(phase(31 downto 20));
  saegezahn_u <= phase(31 downto 20);
-- convert output values
```

```
  process begin
    wait until rising_edge(clk);
    if flag = '0' then
      saegezahn_inv <= "0011" & std_logic_vector(saegezahn_inv_u);
      saegezahn <= "0011" & std_logic_vector(saegezahn_u);
      rechteck  <= "0011" & std_logic_vector(rechteck_u);
    elsif flag = '1' then
      saegezahn_inv <= "0011" & std_logic_vector(saegezahn_u);
      rechteck  <= "0011" & std_logic_vector(rechteck_u);
    end if;
  end process;
end bhv;
```

Listing C.8: VHDL-Quellcode des SPI-Masters

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity SPI_Master_1 is  -- SPI-Modus 0: CPOL=0, CPHA=0
    Generic ( Quarz_Taktfrequenz : integer   := 50000000;  -- Hertz
              SPI_Taktfrequenz   : integer   := 10000000;  -- Hertz / zur Berechnung
                  des Reload-Werts für Taktteiler
              Laenge             : integer   := 16        -- Anzahl der zu
                  übertragenden Bits
             );
    Port ( TX_Data  : in  STD_LOGIC_VECTOR (Laenge-1 downto 0); -- Sendedaten
           RX_Data  : out STD_LOGIC_VECTOR (Laenge-1 downto 0); -- Empfangsdaten
           MOSI     : out STD_LOGIC;
           MISO     : in  STD_LOGIC;
           SCLK     : out STD_LOGIC;
           SS       : out STD_LOGIC;
           TX_Start : in  STD_LOGIC;
           TX_Done  : out STD_LOGIC;
           clk      : in  STD_LOGIC
         );
end SPI_Master_1;

architecture Behavioral of SPI_Master_1 is
  signal   delay       : integer range 0 to (Quarz_Taktfrequenz/(2*SPI_Taktfrequenz));
  constant clock_delay : integer := (Quarz_Taktfrequenz/(2*SPI_Taktfrequenz))-1;

  type   spitx_states is (spi_stx,spi_txactive,spi_etx);
  signal spitxstate    : spitx_states := spi_stx;

  signal spiclk    : std_logic;
  signal spiclklast: std_logic;

  signal bitcounter    : integer range 0 to Laenge; -- wenn bitcounter = Laenge -->
      alle Bits uebertragen
  signal tx_reg        : std_logic_vector(Laenge-1 downto 0) := (others=>'0');
  signal rx_reg        : std_logic_vector(Laenge-1 downto 0) := (others=>'0');

begin
  ------ Verwaltung --------
  process begin
     wait until rising_edge(CLK);
     if(delay>0) then delay <= delay-1;
     else             delay <= clock_delay;
     end if;
     spiclklast <= spiclk;
     case spitxstate is
       when spi_stx =>
             SS          <= '1'; -- slave select disabled
             TX_Done     <= '0';
             bitcounter  <= Laenge;
             spiclk      <= '0'; -- SPI-Modus 0
             if(TX_Start = '1') then
                spitxstate <= spi_txactive;
                SS         <= '0';
                delay      <= clock_delay;
             end if;

       when spi_txactive =>  -- Daten aus tx_reg uebertragen
             if (delay=0) then -- shift
                spiclk <= not spiclk;
                if (bitcounter=0) then -- alle Bits uebertragen -> deselektieren
                   spiclk     <= '0';  -- SPI-Modus 0
                   spitxstate <= spi_etx;
```

```
                end if;
                if(spiclk='1') then    -- SPI-Modus 0
                   bitcounter <= bitcounter-1;
                end if;
             end if;

       when spi_etx =>
             SS      <= '1'; -- disable Slave Select
             TX_Done <= '1';
             if(TX_Start = '0') then -- Handshake: warten, bis Start-Flag geloescht
               spitxstate <= spi_stx;
             end if;
    end case;
  end process;

  ---- Empfangsschieberegister -----
  process begin
     wait until rising_edge(CLK);
     if (spiclk='1' and  spiclklast='0') then -- SPI-Modus 0
        rx_reg <= rx_reg(rx_reg'left-1 downto 0) & MISO;
     end if;
  end process;

  ---- Sendeschieberegister -------
  process begin
     wait until rising_edge(CLK);
     if (spitxstate=spi_stx) then   -- Zurücksetzen, wenn SS inaktiv
        tx_reg <= TX_Data;
     end if;
     if (spiclk='0' and  spiclklast='1') then -- SPI-Modus 0
        tx_reg <= tx_reg(tx_reg'left-1 downto 0) & tx_reg(0);
     end if;
  end process;

  SCLK    <= spiclk;
  MOSI    <= tx_reg(tx_reg'left);
  RX_Data <= rx_reg;

end Behavioral;
```

D. CD

Diese Anhänge sind auf der beigelegten CD, mit folgender Ordnerstruktur zu finden:

- **Datenblätter** (enthält die Datenblätter aller verwendeter Bauteile)
- **Masterthesis** (enthält die Abschlussarbeit als PDF-Dokument)
- **Quartus Projekt** (enthält das komplette VHDL-Projekt)
- **Simulink Projekte** (enthält alle MATLAB/Simulink Projekte)

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Hamburg, 18. September 2014	
Ort, Datum	Unterschrift