Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Joshua Supra

Entwicklung einer Handsteuerung für Schrittmotoren mit Mikrocontroller und FPGA

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

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

Joshua Supra

Entwicklung einer Handsteuerung für Schrittmotoren mit Mikrocontroller und FPGA

Bachelorthesis eingereicht im Rahmen der Bachelorprüfung
im Studiengang Informations- und Elektrotechnik
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr.-Ing. Karl-Ragmar Riemschneider
Zweitgutachter : Prof. Dr. Pawel Buczek

Abgegeben am 14. September 2018

Joshua Supra

Thema der Bachelorthesis

Entwicklung einer Handsteuerung für Schrittmotoren mit Mikrocontroller und FPGA

Stichworte

Handsteuerung, Mikrocontroller, FPGA, STM32F4, Cyclone IV, 2-Phasen-Schrittmotoren, Hardware, Software, Firmware

Kurzzusammenfassung

Im Rahmen dieser Bachelorthesis wird eine Handsteuerung für 2-Phasen-Schrittmotoren entwickelt. Mit dieser Handsteuerung wird es möglich sein, experimentelle Aufbauten am DESY im Vorwege zu überprüfen. Es werden die Konzepte für die Hardware, die Software im Mikrocontroller und die Firmware für den FPGA skizziert und erläutert. Basierend auf den Konzepten wird die Handsteuerung realisiert. Abschließend werden Funktionstests durchgeführt und ein Ausblick auf die zukünftigen Entwicklungen gegeben.

Joshua Supra

Title of the paper

Development of a hand control for stepper motors with microcontroller and FPGA

Keywords

Hand control, microprocessor, FPGA, STM32F4, Cyclone IV, 2-phase-stepper-motors, hardware, software, firmware

Abstract

In this bachelor thesis a hand control for 2-phase-stepper-motors will be developed. With this hand control it will be possible to validate experimental construction at DESY in advance. Concepts for the hardware, the software of the microprocessor and firmware for the FPGA will be described. Based on these concepts the hand control will be realised. Eventually the functionality will be tested and a prospect on future developments will be given.

Inhaltsverzeichnis

1. Einführung **7**
1.1. DESY . . . 7
1.2. Großanlagen am DESY in Hamburg . . . 8
1.3. Experiment Control Gruppe . . . 9
1.4. Schrittmotoren . . . 9
1.5. Aufgabe . . . 13

2. Konzeption und Vorstudie **14**
2.1. Rahmenbedingungen . . . 14
2.1.1. Kontrolleinheit . . . 14
2.1.2. Endstufe . . . 14
2.1.3. Spannungsversorgung . . . 15
2.1.4. Backplane . . . 15
2.1.5. Gehäuse . . . 16
2.1.6. Encoder . . . 17
2.2. Anforderungen . . . 18
2.2.1. Hardware Anforderungen . . . 18
2.2.2. Software Anforderungen . . . 19
2.2.3. Firmware Anforderungen . . . 19
2.3. Konzept der Handsteuerung . . . 20
2.3.1. Konzept der Hardware . . . 21
2.3.2. Konzept der Software im Mikrocontroller . . . 22
2.3.3. Konzept der Firmware im FPGA . . . 23

3. Realisierung der Konzepte **25**
3.1. Auswahl der Komponenten . . . 25
3.1.1. Mikrocontroller . . . 27
3.1.2. FPGA . . . 28
3.2. Aufbau der Leiterplatte . . . 30
3.3. Die Software im Mikrocontroller . . . 31
3.3.1. Menüsteuerung . . . 31
3.3.2. Speicherung der eingestellten Endstufenparameter . . . 31

3.3.3. Schnittstelle für die Kommunikation zum Display 32
3.3.4. Schnittstelle für die Kommunikation zum FPGA 33
3.3.5. Parametrierung der Endstufe 33
3.3.6. Programmablauf 34
3.4. Aufbau der Firmware im FPGA 41
3.4.1. Auswerten von Encodern 41
3.4.2. Verarbeitung der Kommunikation über SPI im FPGA 41
3.4.3. Datenverwaltung 45
3.4.4. Frequenzerzeugung im FPGA 46
3.4.5. Impulserzeugung 48
3.4.6. Impulszähler 52

4. Test und Validierung **53**
4.1. Erstinbetriebnahme 53
4.2. Validierung einzelner Funktionen 56
4.2.1. Impulserzeugung 56
4.2.2. Lesen von Encoder-Signalen 57
4.2.3. Kommunikation mit der Endstufe 58
4.3. Funktionstest der ZMX-Handsteuerung für 2-Phasen-Schrittmotoren 59

5. Auswertung und Bewertung **63**
5.1. Auswertung 63
5.2. Ausblick 64
5.2.1. EtherCAT Anbindung 64
5.2.2. Positionsregelung 64

Abkürzungsverzeichnis **65**

Tabellenverzeichnis **67**

Abbildungsverzeichnis **69**

Literaturverzeichnis **71**

A. Schaltpläne und Zeichnungen **75**

B. VHDL-Quellcode der Firmware vom FPGA **85**
B.1. DDS.vhd 87
B.2. SPISlave.vhd 88
B.3. Registerverwaltung.vhd 93
B.4. Impulszaeler.vhd 97
B.5. Encoder.vhd 98

B.6. Stepgeneration.vhd 100

C. Quellcode der Software des Mikrocontrollers: Header Dateien 110
C.1. main.h 110
C.2. encoder.h 112
C.3. ea_dip203B.h 113
C.4. various.h 115
C.5. spi_fpga.h 116
C.6. I2C.h 117
C.7. gpio.h 118
C.8. ZMX.h 119

D. Quellcode der Software des Mikrocontrollers: Source Dateien 121
D.1. main.c 121
D.2. encoder.c 154
D.3. ea_dip203B.c 158
D.4. various.c 168
D.5. spi_fpga.c 173
D.6. I2C.c 177
D.7. gpio.c 179
D.8. ZMX.c 181

1. Einführung

In diesem Kapitel wird das Forschungsinstitut DESY[1] vorgestellt. Es werden die wichtigsten, sich im Betrieb befindlichen Anlagen dargestellt und eine kurze Übersicht über die Fachgruppe „Experiment Control“ gegeben. Im Anschluss wird auf die Aufgabenstellung der Bachelorthesis eingegangen.

1.1. DESY

Das Deutsche Elektronen-Synchrotron ist ein mit öffentlichen Mitteln finanziertes Forschungszentrum der Helmholtz-Gemeinschaft. Es wurde am 18. Dezember 1959 in Hamburg gegründet. Heute hat das DESY zwei Standorte in Deutschland, den Hauptstandort in Hamburg und einen weiteren in Zeuthen in Brandenburg [1].

Der Bau des ersten Ringbeschleunigers „DESY“ begann Anfang 1959 und wurde 1964 abgeschlossen. Bis Ende 1978 wurden dort Experimente mit Synchrotron-Strahlung durchgeführt. Im Anschluss wurde die Anlage mehrfach umgebaut und erneuert, so dass sie heute noch als Vorbeschleuniger für PETRA[2] III genutzt wird [2, p. 10,19].

Die Forschung am DESY ist in drei Schwerpunkte aufgeteilt [3].

- Entwicklung, Bau und Betrieb von Beschleunigern
- Forschung mit Photonen
- Teilchen- und Astrophysik

[1] Deutsches Elektronen-Synchrotron
[2] Positron-Elektron-Tandem-Ring-Anlage

1.2. Großanlagen am DESY in Hamburg

Zur Zeit werden auf dem Campus in Hamburg drei Großanlagen betrieben: die beiden Linearbeschleuniger FLASH[3] und der European XFEL[4], sowie der Ringbeschleuniger PETRA III.

FLASH

Der Linearbeschleuniger FLASH wurde 2005 fertigstellt und erzeugt seitdem ultrakurz gepulste Röntgenlaserblitze. Auf einer Länge von 315 Metern werden Elektronen auf fast Lichtgeschwindigkeit beschleunigt und durch Magnete abgelenkt. Die dabei entstehenden Röntgenblitze können genutzt werden, um chemische Reaktionen auf atomarer Ebene sichtbar zu machen. FLASH bietet insgesamt fünf Messplätze für Experimente. 2016 wurde eine zweite Experimentierhalle in Betrieb genommen, die Platz für weitere sechs Messplätze zur Verfügung stellt [4].

European XFEL

Wie FLASH ist der European XFEL ein Linearbeschleuniger. Für den Bau und Betrieb ist die eigens gegründete Gesellschaft European XFEL GmbH verantwortlich. Nach sieben Jahren Bauzeit wurde das europäische Gemeinschaftsprojekt 2016 fertiggestellt. Seit 2017 erzeugt der ca. 3,4 Kilometer lange Beschleuniger extrem brillante Laserstrahlung im Röntgenbereich, mit deren Hilfe die Reaktionen von Atomen miteinander aufgenommen werden können. Zurzeit stellt eine unterirdische Experimentierhalle zehn Messplätze zur Verfügung. Bei Bedarf kann der European XFEL um eine weitere Experimentierhalle mit der gleichen Anzahl an Messplätzen erweitert werden [5].

PETRA III

Mit PETRA III stellt DESY eine sehr brillante Lichtquelle zur Verfügung, welche Forscher aus der ganzen Welt nutzen können. In dem 2,3 Kilometer langen Speicherring werden Elektronen auf nahezu Lichtgeschwindigkeit beschleunigt. Durch Ablenkung der Elektronen in Magneten entsteht dabei Synchrotronstrahlung. Diese lässt sich bündeln und erzeugt Strahlen die bis zu 5000-mal feiner sind als ein menschliches Haar. Hierdurch lassen sich Proben bis in den atomaren Bereich analysieren.

[3] Freie-Elektronen-Laser in Hamburg
[4] X-Ray Free-Electron Laser

Seit 2010 liefert PETRA III die Röntgenstrahlung für 30 Messtationen, die jedoch aufgrund der hohen Nachfrage regelmäßig ausgebucht sind. Deshalb wurden zwei neue Experimentierhallen, PETRA III Erweiterung Nord und Ost gebaut. Durch den derzeitigen Ausbau beider Hallen, entstehen zehn weitere Messplätze [6].

1.3. Experiment Control Gruppe

Die Gruppe FS-EC[5] ist für die Unterstützung der Nutzer an den Messplätzen von PETRA III zuständig. Der Support lässt sich in die Bereiche Hardware und Software einteilen. Erstellung von Steuerungssoftware, Datenerfassung und Speicherung von Forschungsdaten gehören zum Aufgabengebiet des Bereichs Software. Entwicklung von nutzerspezifischer Hardware, Reparatur von Geräten oder Anfertigen von speziellen Kabeln, sind Aufgabe des Hardware bzw. Elektronikbereichs der Gruppe [7].

1.4. Schrittmotoren

Schrittmotoren entsprechen in Aufbau und Funktion dem Synchronmotor. Im Prinzip bestehen sie aus einen Läufer und einem Stator mit der Erregerwicklung. Durch Anlegen einer rechteckförmigen Gleichspannung kann eine Bewegung in beide Richtungen erfolgen. Dabei werden je nach Drehrichtung die einzelnen Wicklungen umgepolt, oder abgeschaltet. Der Läufer folgt daraufhin dem geänderten Magnetfeld, bis der maximale magnetische Fluss zwischen Nord- und Südpol vorhanden ist. Der Motor bleibt stehen, bis sich das Feld wieder ändert. Die Anzahl der Polpaare im Läufer und der Phasen im Stator gibt dabei den Winkel vor, um den sich der Motor bei einem Schritt dreht. Ein Motor mit zwei Phasen und einem Polpaar bewegt sich bei einem Schritt um 90 Grad. Dementsprechend benötigt der Läufer für eine Umdrehung vier Schritte (vgl. Abb. 1.1) [12, p. 380-383].

[5] Forschung Synchrotron - Experiment Control

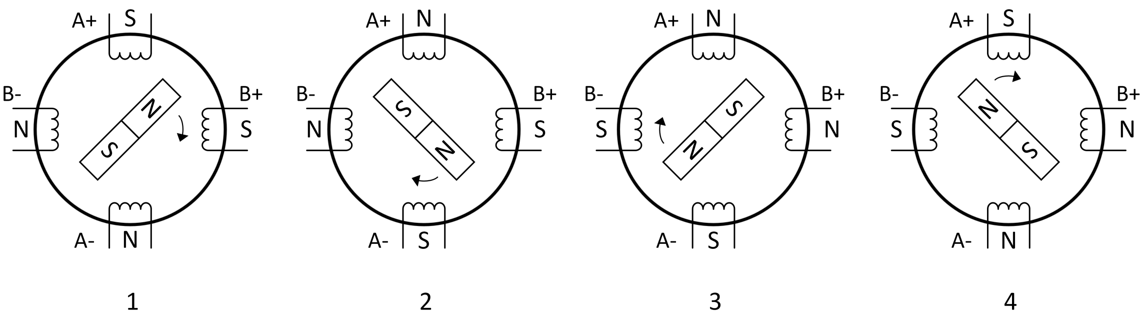


Abbildung 1.1.: *Im Vollschrittbetrieb werden beide Motorphasen bestromt. Bei jedem Schritt bewegt sich der Läufer um 90 Grad im Uhrzeigersinn. Werden die Schritte im umgekehrter Reihenfolge ausgeführt, dreht sich der Läufer gegen den Uhrzeigersinn.*

In Schritt (1) fließt der Strom in den Motorphasen A und B von A+ nach A- bzw. von B+ nach B-. Im folgenden Schritt (2) wird die Motorphase A umgepolt. Der Strom fließt nun dort von A- nach A+. Das Feld dreht sich um 90 Grad im Uhrzeigersinn und somit auch der Läufer. Im nächsten Schritt wird die Motorphase B umgepolt (3), das Feld und somit der Läufer bewegen sich um 90 Grad weiter. Im letzten Schritt (4) wird wieder die Motorphase A umgepolt. Der Strom in der Phase A fließt von A+ nach A- und in der Phase B von B- nach B+. Nach insgesamt vier Schritten hat sich der Läufer um 360 Grad gedreht. In Tabelle 1.1 ist die Stromrichtung bei den jeweiligen Schritten zusammengefasst.

Schritt	Stromrichtung	
	A	B
1	+	+
2	-	+
3	-	-
4	+	-

Tabelle 1.1.: *Stromrichtung in den Motorphasen im Vollschrittbetrieb.*

Es ist möglich den Läufer um kleinere Winkel als 90 Grad zu bewegen. Hierfür werden Betriebsarten wie der Halbschritt- oder der Mikroschrittbetrieb genutzt. Im Halbschrittbetrieb (vgl. Abb. 1.2) werden im Gegensatz zum Vollschrittbetrieb Motorphasen bei jedem zweiten Schritt abgeschaltet [12, p. 383]. Dies hat zur Folge, dass sich der zweiphasige Motor mit jedem Schritt um 45 Grad bewegt.

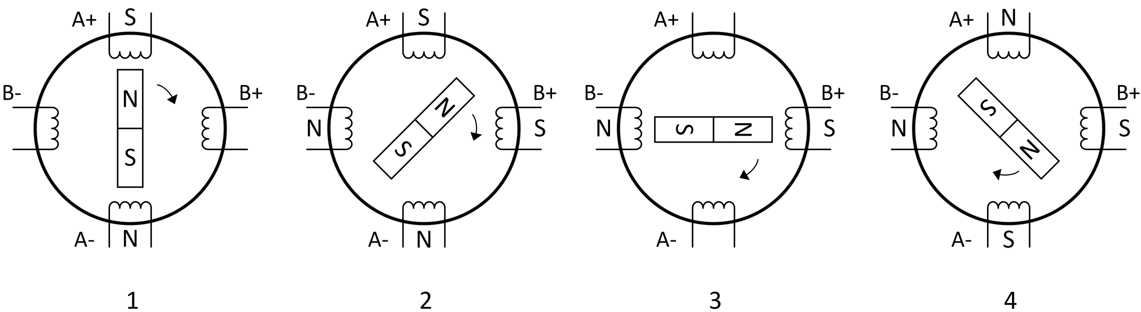


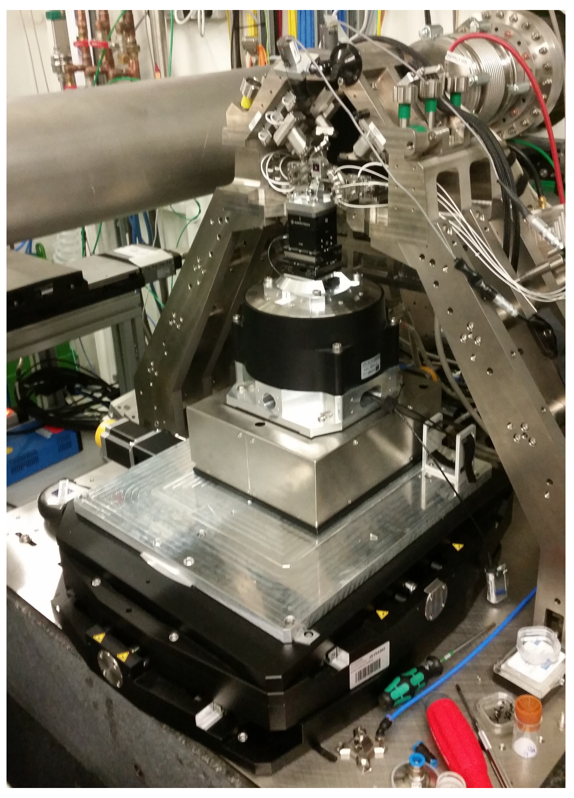
Abbildung 1.2.: *Durch das Zu- und Abschalten der Motorphasen ist es möglich den Läufer um 45 Grad zu bewegen. Die Abbildung zeigt die ersten vier Schritte im Halbschrittbetrieb, dabei dreht sich der Läufer im Uhrzeigersinn.*

Für eine ganze Umdrehung benötigt der Motor acht Schritte. Die Stromrichtung in den Motorphasen im Halbschrittbetrieb ist in Tabelle 1.2 dargestellt.

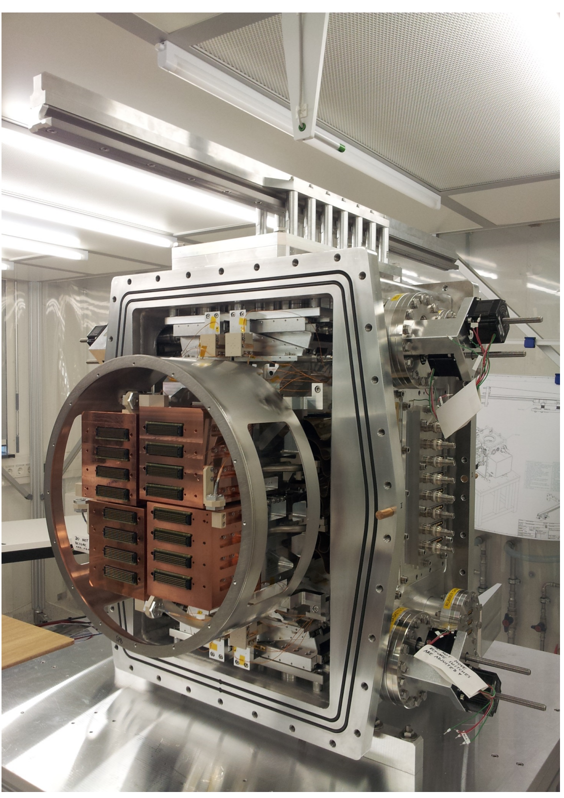
Schritt	Stromrichtung	
	A	B
1	+	0
2	+	+
3	0	+
4	-	+
5	-	0
6	-	-
7	0	-
8	+	-

Tabelle 1.2.: *Stromrichtung in den Motorphasen im Halbschrittbetrieb.*

Um kleinere Schrittauflösungen zu erreichen ist es möglich neben dem Ab- und Zuschalten der Motorphasen, auch den Strom in den Phasen Schrittweise zu verändern. Wird der Strom in einem bestimmten Verhältnis zueinander verändert, kann eine Auflösung von z. B. 128 Mikroschritten je Vollschritt erreicht werden. Für eine volle Umdrehung würde ein Motor mit zwei Phasen und einem Polpaar 512 Schritte benötigen. Somit ist eine Auflösung von 0.7 Grad je Schritt möglich [12, p. 383-384].



(a) Die Probe (oben im Bild) befindet sich auf einem Schrittmotor der in Z Richtung bewegt werden kann. Darunter zu sehen ist ein Tisch für die Bewegung in X und Y Richtung.



(b) Vakuumkammer des AGIPD Detektors. Die Schrittmotoren (außerhalb der Kammer) dienen zum Verfahren der einzelnen Quadranten (mitte Bild) in X und Y Richtung. Foto: Annette Delfs ©DESY

Abbildung 1.3.: *Die beiden Abbildungen zeigen zwei Einsatzgebiete von Schrittmotoren am DESY. Der Aufbau in Abbildung a befindet sich an einem Experiment an einem Messplatz von PETRA III. Die Abbildung b zeigt einen Aufbau in einem Labor auf dem Gelände des DESY.*

1.5. Aufgabe

Es soll eine neue Serie von Handsteuerungen für 2-Phasen-Schrittmotoren gebaut werden. Hierfür wird eine neue Leiterplatte zur Ansteuerung der Endstufe für die Schrittmotoren, sowie Anzeige- und Bedienelemente benötigt. Auf einem Display sollen Motordaten, sowie wichtige Daten im Betrieb dargestellt werden. Nicht mehr lieferbare Teile sollen durch kompatible Komponenten ersetzt werden. Der vorhandene 8-Bit Mikrocontroller soll durch ein 32-Bit Mikrocontroller mit ARM[6]-Architektur ersetzt werden. Um Encoder-Signale auswerten zu können, soll ein passendes System entwickelt werden. Für die Steuerung muss eine Software für den Mikrocontroller entwickelt werden. Nach erfolgreicher Entwicklung der Hard- und Software, muss durch ein Test die Funktionalität der Handsteuerung belegt werden.
Im Folgenden wird die Handsteuerung für 2-Phasen-Schrittmotoren „ZMX-Handsteuerung“[7] genannt.

[6] Advanced RISC Machines
[7] Durch DESY gewählter Name

2. Konzeption und Vorstudie

In diesem Kapitel werden die Rahmenbedingungen zur Entwicklung der ZMX-Handsteuerung beschrieben. Es werden die Anforderungen an die ZMX-Handsteuerung definiert und die Vorgängerversion analysiert. Für die Anforderungen werden passende Lösungsmöglichkeiten ermittelt und erläutert.

2.1. Rahmenbedingungen

Im Folgenden werden die Rahmenbedingungen für die Entwicklung der ZMX-Handsteuerung beschrieben. Es werden die wichtigsten Baugruppen skizziert, sowie das Funktionsprinzip der Encoder dargestellt.

2.1.1. Kontrolleinheit

Die Kontrolleinheit bildet das Herzstück der ZMX-Handsteuerung. Ihre Aufgabe besteht darin, die durch den Nutzer eingestellten Parameter zur Endstufe zu übertragen. Des Weiteren wird die Kontrolleinheit benötigt, um die Impulse und das Richtungssignal vorzugeben. Das Auswerten von Endlagen und Inkrementalencodern gehört außerdem zum Funktionsumfang. Auf ihr befinden sich alle Anzeige- und Bedienelemente. Um in das Gehäuse zu passen, muss sie die gleichen Maße besitzen, wie die Vorgängerversion. Das Konzeptionieren, Realisieren und Testen ist Inhalt dieser Bachelorthesis. Auf den Quellcode wird dabei nicht speziell eingegangen.

2.1.2. Endstufe

Für die Ansteuerung von 2-Phasen-Schrittmotoren wird eine Endstufe benötigt. Die Firma Phytron bietet mit der ZMX+ eine 19" Schrittmotor-Endstufe (vgl. Abb. 2.1) zur bipolaren Ansteuerung von Zweiphasen-Schrittmotoren an. Über den ServiceBus [8, p. 5-6] wird die Mög-

lichkeit geboten die Endstufenparameter wie z.B. Schrittauflösung, Lauf[1]- oder Stopstrom[2] einzustellen. Es handelt sich hierbei um eine serielle Kommunikation mit einem vorgegebenen Telegrammformat. Durch vielfachen Einsatz am DESY hat sich diese Endstufe bewährt und wird ebenfalls in der Vorgängerversion der ZMX-Handsteuerung eingesetzt. Im Folgenden wird die 2-Phasen-Schrittmotor Endstufe „ZMX-Endstufe genannt“ [9].

Abbildung 2.1.: *Gezeigt ist die ZMX$^+$ Endstufe der Firma Phytron. Sie wird zur Ansteuerung der Schrittmotoren verwendet.*

2.1.3. Spannungsversorgung

Die gesamte Spannungsversorgung der ZMX-Handsteuerung wird durch ein AC-DC Netzteil von N2Power realisiert. Über den Kaltgeräteanschluss in der Rückplatte werden 230V AC eingespeist. Daraus generiert das Netzteil 56V DC für die Versorgung der ZMX-Endstufe, 12V DC für die Endlagen und Encoder, sowie 5V DC für die Versorgung der Kontrolleinheit [10]. Der Einsatz dieses Netzteils hat sich in der Vorgängerversion bewährt und wird deshalb wieder eingesetzt.

2.1.4. Backplane

Im hinteren Teil des Gehäuses befindet sich die Backplane[3]. Sie bildet die Schnittstelle zwischen ZMX-Endstufe, Kontrolleinheit, Spannungsversorgung und Motor. Mit Hilfe von Steck-

[1]Strom in den Motorphasen während des Betriebs
[2]Haltestrom in den Motorphasen im Stillstand des Motors
[3]Frei gewählter Name

verbindern werden die für den Betrieb notwendigen Signale von und zur Rückplatte geführt. Die Backplane wurde durch die Fachgruppe FS-EC zur Verfügung gestellt.[4]

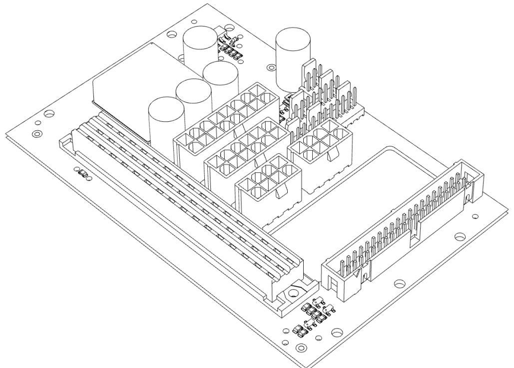


Abbildung 2.2.: *Die Backplane ist im hinteren Teil des Gehäuses verbaut. Sie bildet die Schnittstelle zwischen Motor, Endstufe und Kontrolleinheit.*

2.1.5. Gehäuse

Bei dem Gehäuse handelt es sich um ein Komplettgehäuse[5] mit EMV[6]-Schirmung von der Firma Schroff. Es besitzt Lüftungsschlitze um die durch das Netzteil, die ZMX-Endstufe oder der Elektronik erzeugte Wärme abzutransportieren. Das Gehäuse hat sich in der Vorgängerversion bewährt und wird aus diesem Grund wieder eingesetzt [11, p. 4].

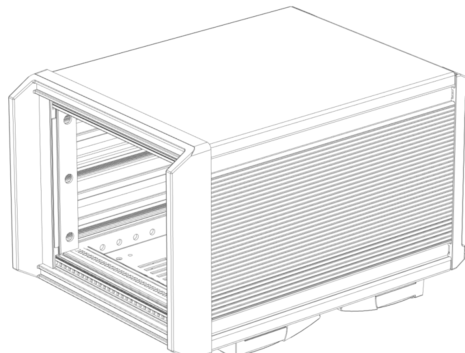


Abbildung 2.3.: *Das Gehäuse bietet Platz für die Kontrolleinheit, Backplane, ZMX-Endstufe und Spannungsversorgung.*

[4]Schaltplan siehe Anhang A, „ZMX Handsteuerung Backplane“
[5]Tischgehäuse PropacPRO: 24576-103
[6]Elektromagnetische Verträglichkeit

2.1.6. Encoder

Encoder bieten die Möglichkeit eine Rückmeldung über die Position oder Geschwindigkeit eines sich drehenden bzw. bewegenden Objekts wie z.B. einen Motor, zu geben. Das Objekt ist mit einem Raster verbunden, das z.B optisch durch einen Lesekopf abgetastet wird. Es gibt die Absolut- und die Inkremental-Encoder. Absolute Encoder verarbeiten ein codiertes Raster, ähnlich wie ein Lineal, wodurch die Position ermittelt werden kann. Vorteil bei dieser Methode ist, dass bei einem Verlust der Versorgungsspannung die Information der Position erhalten bleibt. Ein Nachteil hingegen ist die geringere Auflösung [12, p. 333-336].
Inkremental-Encoder besitzen ein gleichmäßig aufgeteiltes Raster. Sie stellen zwei Signale zur Verfügung, A und B, die eine Phasenverschiebung von 90 Grad vorweisen. Je nachdem ob zuerst A und dann B den Pegel ändert, zählt eine Logik auf- bzw. abwärts (vgl. Abb. 2.4) [13, p. 619]. In diesem Fall wird ein Rechtecksignal dargestellt. An den Experimenten bei PETRA III werden hauptsächlich Encoder mit Quadratursignal verwendet.

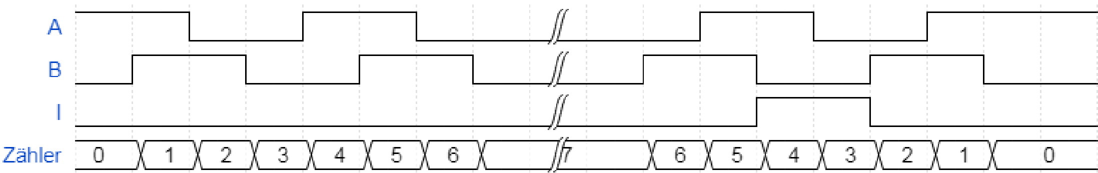


Abbildung 2.4.: *Encodersignale A und B mit zugehörigem Zählerverhalten. Eilt das Signal A dem Signal B um 90 Grad voraus wird inkrementiert, eilt es nach wird dekrementiert.*

Die Drehrichtung eines angeschlossenen Motors kann ebenfalls aus den Encodern ermittelt werden. Dafür muss bekannt sein, wie die beiden Signale bei einer bestimmten Drehrichtung zueinander phasenverschoben sind. Zusätzlich zu den beiden Ausgängen stellen einige Encoder ein dritten Ausgang, den Index I zur Verfügung. Erreicht der Motor diese Position, wird das Signal auf den Pegel „high“ gesetzt. Dies kann z.B. der Fall sein, wenn der Motor seine 0 Grad Position erreicht hat [13, p. 619]. Dadurch könnte der Zähler zurückgesetzt werden. Ein großer Vorteil gegenüber den Absolut-Encodern ist, dass eine deutlich höhere Auflösung bis in den Nanometer Bereich erreicht werden kann. Ein Ausfall der Versorgungsspannung führt zum Verlust der gemessenen Position [12, p. 336].

2.2. Anforderungen

Im Folgenden werden die Anforderungen an die ZMX-Handsteuerung definiert. Hierbei ist darauf zu achten, dass die Bedienbarkeit der Vorgängerversion[7] entspricht. Die Menüführung soll identisch bleiben und nur um eventuelle neue Funktionalitäten erweitert werden. Für die Impulserzeugung und das Auswerten von Encodern soll ein FPGA[8] verwendet werden. FPGAs können im Vergleich zum Mikrocontroller Aufgaben parallel ausführen und somit schneller auf Ereignisse reagieren. Die Verwendung des FPGA ist als Anforderung, um zukünftige Funktionalitäten umsetzten zu können, zu betrachten. Die Anforderungen wurden durch das Elektroniklabor der Fachgruppe FS-EC festgelegt.

2.2.1. Hardware Anforderungen

Die Anforderungen an die Hardware beziehen sich auf den Aufbau der Kontrolleinheit, sie sind im folgenden aufgelistet:

- Steuerung der Endstufe über die Eingänge: Takt, Drehrichtung, Entregen, Reset [9, p. 25].
- An den Motor angeschlossene Endlagen sollen ausgewertet werden.
- LEDs visualisieren den Zustand der Endlagen [14, p. 6].
- Eine LED visualisiert den Betriebszustand der Endstufe: Rot signalisiert den Zustand aktiviert, grün signalisiert den Zustand deaktiviert.
- An den Motor angeschlossene Encoder sollen ausgewertet werden.
- Über Taster und Drehencoder auf der Leiterplatte soll die ZMX-Handsteuerung bedient werden.
- Ein Display visualisiert die wichtigsten Endstufenparameter (siehe Software Anforderungen).
- Ein Mikrocontroller mit 32-Bit ARM-Architektur soll die Bedien- und Anzeigeelemente auswerten und ansteuern.
- Die Leiterplatte darf nicht größer als 112 mm x 98,5 mm sein.
- Die Ein- und Ausgabeelemente sollen gemäß der Zeichnung der Frontplatte[9] angeordnet werden.

[7] ZMX-Handsteuerung Serie 2
[8] Field Programmable Gate Array
[9] Zeichnung siehe Anhang A, „Frontplatte ZMX-Handsteuerung“

- Die Leiterplatte wird mit Schrauben an der Frontplatte befestigt.

2.2.2. Software Anforderungen

Bei der Programmierung des Mikrocontrollers sollen die folgenden Punkte umgesetzt werden:

- Parametrierung der Endstufe über den ServiceBus [8]: Laufstrom und Stoppstrom einstellbar von 0A bis 5A in 0,1A Schritten, Schritteinstellung einstellbar [9, p. 24].
- Endstufenparameter sollen im Display visualisiert werden.
- Der Zustand der Endlagen soll im Display angezeigt werden.
- Visualisierung der gezählten Encoder-Schritte vom Motor im Display.
- Visualisierung der Anzahl der erzeugten Impulse für die ZMX-Endstufe.
- Die Kommunikation mit dem FPGA soll realisiert werden.

2.2.3. Firmware Anforderungen

Für die Firmware des FPGA gelten die folgenden Vorgaben:

- Impulserzeugung für die ZMX-Endstufe: 1Hz bis 100kHz mit einer minimalen Impulsbreite von 1µs [9, p. 41].
- Endlageneinstellung CW[10], CCW[11]: „active-high“, „active-low“und deaktiviert.
- Auslösen einer Endlagen führt zum Stopp des Motors, keine Bewegung in diese Richtung mehr möglich.
- Auswerten von Encoder-Signalen mit Reset Funktion.
- Schnittstelle zur Kommunikation mit dem Mikrocontroller.
- Einstellen der Motordrehrichtung und Aktivierung der ZMX-Endstufe.

[10] Clockwise

[11] Counter Clockwise

2.3. Konzept der Handsteuerung

Im Folgenden wird auf das Konzept der ZMX-Handsteuerung bzw. der Kontrolleinheit eingegangen. Abbildung 2.5 zeigt dieses Konzept. Auf Darstellung der Backplane zur Signalverteilung wurde aus Gründen der Übersicht verzichtet.
Auf der Kontrolleinheit befinden sich die Taster und Drehencoder (1) zur Bedienung der ZMX-Handsteuerung. Außerdem befinden sich dort LEDs zur Anzeige. Der Mikrocontroller (2) wertet die Taster und die Drehencoder aus und steuert die LEDs an. Eingestellte Parameter wie Lauf- und Stoppstrom werden im Display (3) dargestellt. Gleichzeitig erfolgt die Parametrierung der Endstufe (5) über den ServiceBus. Die Ansteuerung der ZMX-Endstufe, d.h. Impulserzeugung, die Aktivierung oder die Auslösung des Signals Reset erfolgt über den FPGA (4). Die dafür benötigten Werte werden ebenfalls über eine Serielle Schnittstelle von dem Mikrocontroller gesendet. Ist die Endstufe aktiviert und erhält Impulse, steuert sie gemäß der Parametrierung den Motor (6) an. Angeschlossene Encoder und Endlagen werden vom FPGA ausgewertet und die Informationen an den Mikrocontroller übertragen.

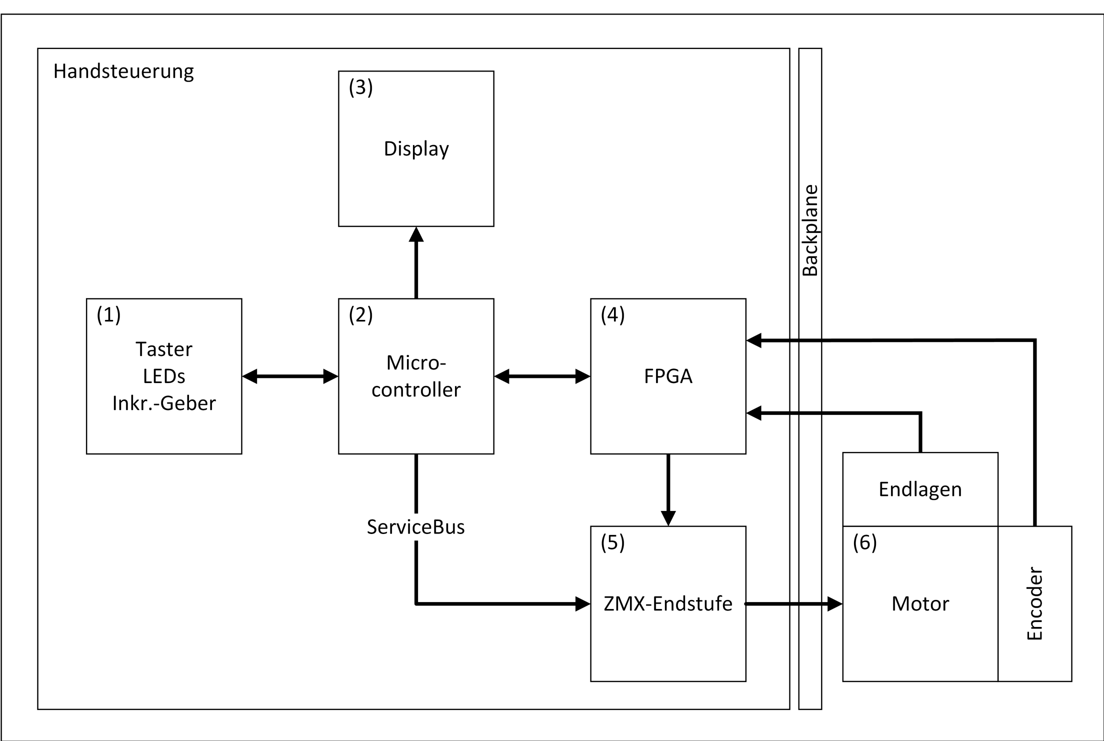


Abbildung 2.5.: *Konzept für den Aufbau der Handsteuerung. Die Kontrolleinheit besteht aus den Ein- und Ausgabeelementen, sowie Mikrocontroller und FPGA. Die Endstufe wird über den ServiceBus parametriert, während der FPGA für die Impulserzeugung und das Auslesen von Endlagen und Encoder zuständig ist.*

2.3.1. Konzept der Hardware

Das Hardware Konzept in Abbildung 2.6 stellt die Anordnung der Baugruppen auf der Leiterplatte dar. Im unteren Teil befinden sich die Ein- und Ausgabeelemente (1), wie die Taster und Drehencoder. Der Mikrocontroller (3) wird zentral auf der Leiterplatte positioniert. Der Takt wird ihm von einem einen Oszillator (2) mit 8MHz zur Verfügung gestellt. Das Display (5) wird über SPI[12] vom Mikrocontroller angesteuert. Um den Kontrast einstellen zu können, wird ein digitales Potentiometer eingesetzt, das über I^2C[13] angesprochen wird.

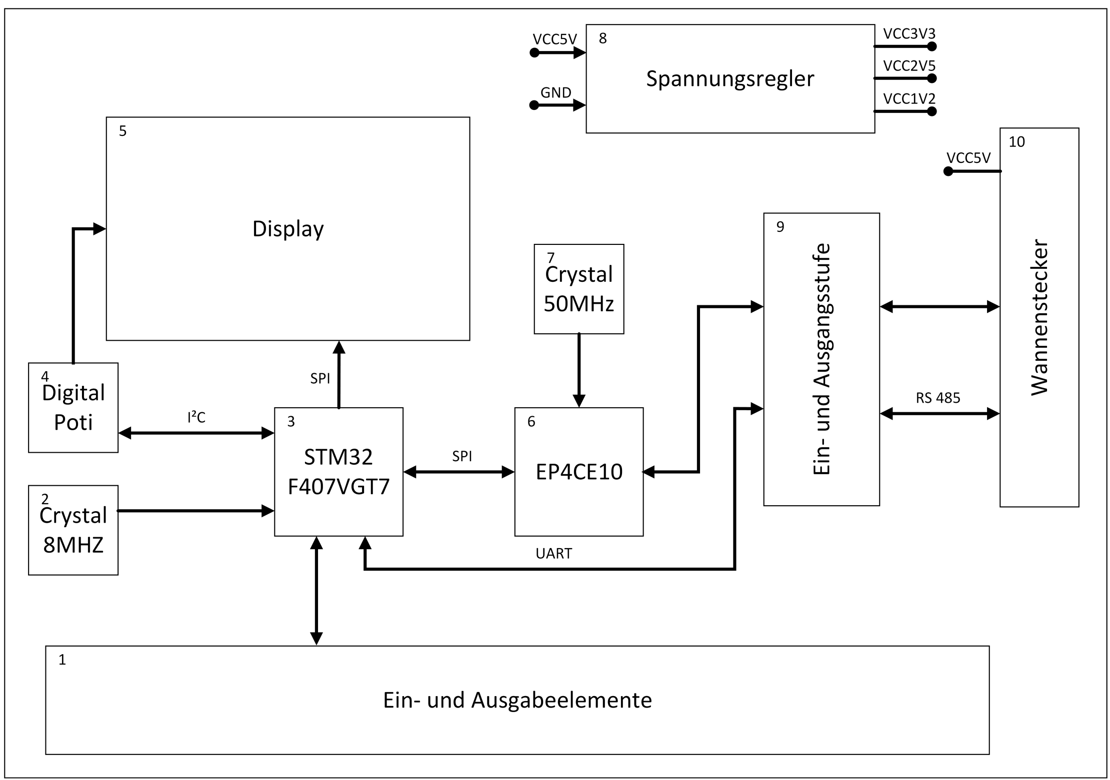


Abbildung 2.6.: *Hardware-Konzept der Kontrolleinheit: Geplante räumliche Anordnung der einzelnen Teilschaltungen/Komponenten auf der Leiterplatte.*

Der FPGA (6) bekommt seinen Takt ebenfalls über einen Oszillator (7) mit einer Frequenz von 50MHz. Die Parametrierung der Endstufe erfolgt über eine UART[14]-Schnittstelle vom Mikrocontroller und wird mit Hilfe von Treibern in der Ein- und Ausgangsstufe (9) auf RS485 gewandelt. Die Pegel der Signale zwischen dem FPGA und der Endstufe werden angepasst. Es findet eine Wandlung der Pegel von 3,3V auf 5V statt. Die Endlagen werden mit 15V

[12] Serial Peripheral Interface
[13] Inter-Integrated Circuit
[14] Universal Asynchronous Receiver Transmitter

von der Backplane versorgt. Um sie mit dem FPGA auswerten zu können, wird der Pegel auf 3,3V angepasst. Über den Wannenstecker (10) wird die Verbindung zur Backplane und somit zu Endstufe hergestellt. Hierüber werden ebenfalls die 5V zur Verfügung gestellt. Die Spannungsregler (8) wandeln 5V auf 3,3V. Diese dienen zur Versorgung der ICs und den I/Os des FPGAs. Außerdem stellen sie 2,5V für die analoge und 1,2V für die digitale PLL-Spannungsversorgung des FPGAs bereit.

2.3.2. Konzept der Software im Mikrocontroller

Der Mikrocontroller bildet die zentrale datenverarbeitende Baugruppe. In Abbildung 2.7 wird dieses Konzept skizziert. Mit Hilfe des Menü-Tasters und dem Drehencoder in der Frontplatte, wird das Menü (1) gesteuert. Die Menüstruktur (2) wird über die Kommunikationsschnittstelle (3) an das Display übertragen und dort visualisiert. Über das Menü lässt sich die ZMX-Endstufe konfigurieren (4), d.h. es lassen sich Stopp- und Laufstrom sowie Schrittauflösung einstellen. Die Parameter werden über UART (5) an die ZMX-Endstufe übertragen. Frequenzeinstellung für die Impulserzeugung, aktivieren der Endstufe und Laufrichtungseinstellungen werden in der Konfiguration für den FPGA (7) ausgewertet.

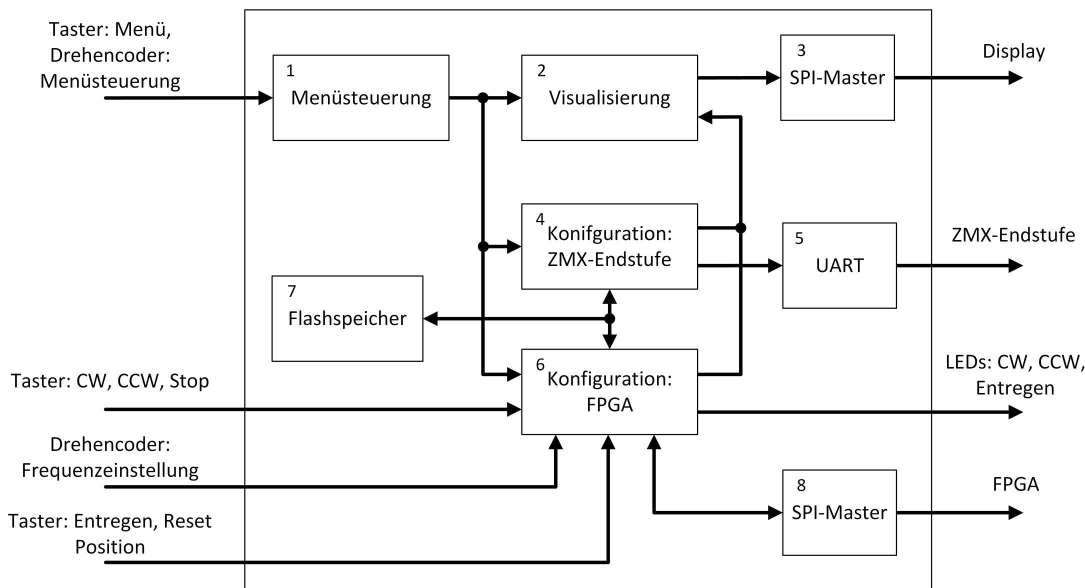


Abbildung 2.7.: *Dargestellt ist das Konzept der Software im Mikrocontroller. Einstellungen erfolgen durch Eingabeelemente in der Frontplatte. Parameter werden im Display visualisiert und an die entsprechende Baugruppe übertragen.*

Über SPI (8) werden die Daten zum FPGA übertragen. Im Flash-Speicher (6) werden Endstufenparameter und Einstellungen gespeichert, damit sie bei der nächsten Nutzung der

Handsteuerung wieder zur Verfügung stehen. Informationen über den Zustand der Endlagen erhält der Mikrocontroller ebenfalls über SPI vom FPGA und steuert die LEDs in der Frontplatte dementsprechend an.

2.3.3. Konzept der Firmware im FPGA

Der FPGA erhält seinen Takt von einem externen 50MHz Oszillator und wird intern mit Hilfe einer PLL[15] auf 100MHz geregelt. Dieser globale Takt versorgt alle Teilschaltungen in der Logik. Sollte der Takt aufgrund geänderter Anforderungen nicht mehr ausreichen, ist es möglich die Frequenz zu erhöhen. Den limitierenden Faktor stellt hierbei die Komplexität der Schaltung und die damit verbundene Gatterlaufzeit dar.

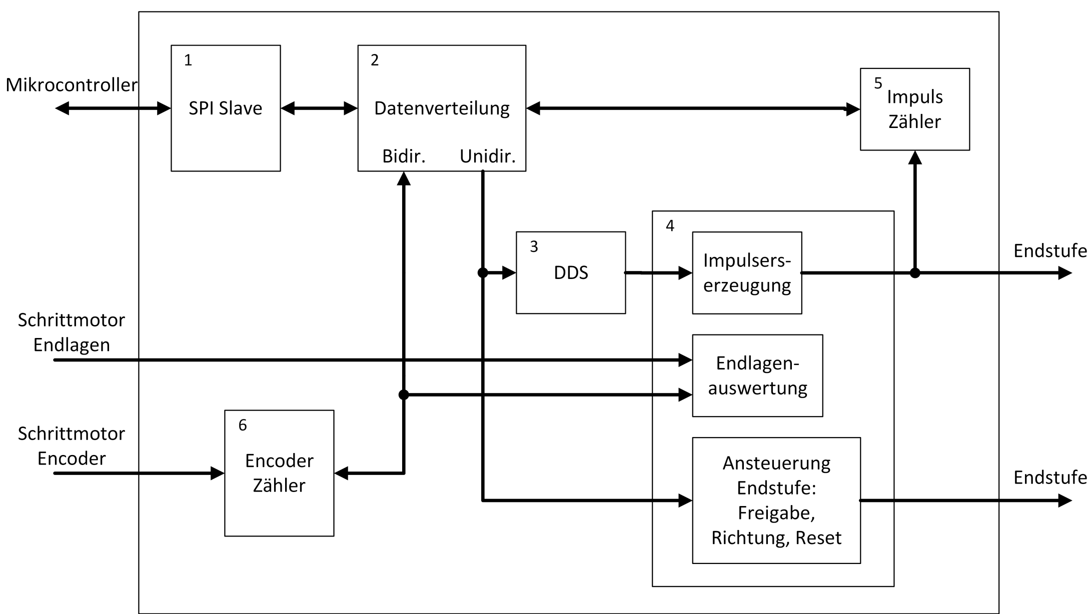


Abbildung 2.8.: *Das Konzept zeigt den Aufbau der Firmware innerhalb des FPGAs. Hierbei wird zwischen uni- und bidirektionalem Datenaustausch unterschieden. Je nach Zugriffsmodus können Register beschrieben oder gelesen werden.*

In Abbildung 2.8 wird das Hardware-Konzept für die Firmware vom FPGA dargestellt. Über das Modul zur SPI Kommunikation (1) findet der Datenaustausch zwischen FPGA und Mikrocontroller statt. Die Schnittstelle ist als Slave definiert, d.h. es können nur Daten gesendet und empfangen werden, wenn der Mikrocontroller den Schiebetakt bereitstellt. Werden Daten empfangen oder sollen gesendet werden, stellt das Modul für die Datenverteilung (2) diese bereit. Hierbei wird zwischen uni- und bidirektionaler Datenverteilung unterschieden. Un-

[15] Phase Locked Loop

idirektional bedeutet, es werden nur Daten in Register geschrieben, bidirektional es können die Register auch ausgelesen werden. Eine Beschreibung der Register und Zugriffsmöglichkeiten befinden sich in Tabelle 2.1. Die einstellbare Frequenz zur Taktung der Impulse wird im Modul DDS[16] (3) erzeugt und in den Baustein für die Impulserzeugung (4) eingespeist. Hier findet auch die Auswertung der Endlagen und die Ansteuerung der Endstufe statt. Hat eine Endlage ausgelöst, fällt die Freigabe zur Impulserzeugung weg und der Motor bleibt stehen. Der Baustein Impulszähler (5) zählt die generierten Pulse, die vom Mikrocontroller ausgelesen werden können. Das Modul Encoder-Zähler (6) wertet die Encoder vom Schrittmotor aus. Bei Bedarf kann der Zählerstand dem Mikrocontroller bereitgestellt werden.

Register	**Zugriff**	**Beschreibung**
Firmware	R	Lese aktuelle Firmware Version aus
Inkrementalgeber	R	Lese Encoder Wert aus
	W	Reset Encoder
Steuerregister	R	Lese Steuerregister und Endlagen
	W	Schreibe Steuerregister
n-Schritte Sollwert	W	Schreibe n-Schritte Sollwert
Impulsfrequenz	W	Schreibe Impulsfrequenz
Impulszähler	R	Lese Impulszähler
	W	Reset Impulszähler

Abkürzungen: R = Read, W = Write

Tabelle 2.1.: *Register im FPGA und deren Zugriffsmodus*

[16]Direct Data Synthesis

3. Realisierung der Konzepte

In diesem Kapitel wird auf die Umsetzung der Konzepte eingegangen. Als Erstes wird die Realisierung der Hardware beschrieben. Anschließend die Implementierung der Konzepte im Mikrocontroller bzw. FPGA.

3.1. Auswahl der Komponenten

Die neue ZMX-Handsteuerung soll vom Aufbau und der Bedienung der Vorgängerversion entsprechen. Aus diesem Grund wird die Vorgängerversion in Bezug auf die dort verwendeten Bauteile analysiert[1]. Hierbei wird explizit nur auf die Bedienelemente und den Mikrocontroller eingegangen. Das Ergebnis ist in Tabelle 3.1 dargestellt.

Bauteil	**Serie 2**	**Status**[a]	**Serie 3**
Taster	Schurter: 1241.1612	OK	Schurter: 1241.1612
Drehencoder	Grayhill: 62P22-H	Obs.	Bourns: PEC11R-4215F-S0024
Potentiometer	Vishay: MOD 335, 10k	N/A	Bourns: PEC11R-4215F-S0024
Display	EA: Electronic Assembly: DIP204B-6NLW	Obs.	Electronic Assembly: EA DIP203B-6NLW
LEDs	Osram: LSG T676	OK	Osram: LSG T676
Mikrocontroller	Atmel: ATmega64-16AU	N/A	STMicroelectronics: STM32Fxx

a) Abkürzungen: OK: wird verwendet, N/A: nicht mehr verwendet, Obs: nicht mehr lieferbar

Tabelle 3.1.: *Ausgewählte Bauteile aus der Handsteuerung Serie 2 und deren Status, die in der Serie 3 wiederverwendet werden sollen bzw. mögliche Ersatztypen*

Die Taster zur Bedienung der Handsteuerung sind lieferbar und können somit weiter verwendet werden. Der optische Drehencoder von Grayhill ist nicht mehr lieferbar und wird durch

[1] Schaltplan siehe Anhang A, „ZMX-Handsteuerung Frontplatine“

einen mechanischen Drehencoder von der Firma Bourns ersetzt. Das zuvor verwendete Potentiometer zur Einstellung der Impulsfrequenz, wird ebenfalls durch einen Drehencoder ausgetauscht. Die LEDs sind verfügbar und können wieder eingesetzt werden. Das Display der Serie 2 ist nicht mehr lieferbar, stattdessen wird das Nachfolgemodell vom selben Hersteller verbaut. Das Display kann entweder über SPI oder einen 4/8-Bit parallel Bus angesteuert werden. Der einzige relevante Unterschied besteht dabei in der Änderung der Versorgungsspannung von 5V DC auf 3,3V DC [15].
Die größte Änderung wird der Ersatz des 8-Bit Mikrocontrollers durch einen 32-Bit ARM-Prozessor. Nach Rücksprache mit der Fachgruppe FS-EC wurde entschieden einen ARM Cortex-M4 32 Bit MCU[2] von STM[3] zu nutzen. Durch den wiederholten Einsatz der Mikrocontroller von STM innerhalb der Gruppe, kann auf ein fundiertes Wissen im Bereich der Programmierung zurückgegriffen werden. Einen weiteren Vorteil bietet die identische Versorgungsspannung von Display und Mikrocontroller. Eine Anpassung der Pegel für eine Kommunikation untereinander entfällt. Die sich aus den Anforderungen und den Konzepten ergebenen Schnittstellen sind in Tabelle 3.2 zusammengefasst.

Teilnehmer A	Teilmehmer B	Schnittstelle	Eigenschaft
Mikrocontroller	Display	SPI	Simplex, Mikrocontroller: Master
Mikrocontroller	Digitales Potentiometer	I^2C	Half-Duplex, Mikrocontroller: Master
Mikrocontroller	FPGA	SPI	Duplex, Mikrocontroller: Master
Mikrocontroller	ZMX-Endstufe	UART / RS485	TX, RX
Mikrocontroller	Taster und LEDs	GPIO	Menü, Entregen, CW, CCW, Stop, Pos. Reset
FPGA	ZMX-Endstufe	GPIO[4]	Reset, Richtung, Entregen
FPGA	Motor: Encoder	GPIO	A, B
FPGA	Motor: Endlagen	GPIO	CW, CCW

Tabelle 3.2.: *Schnittstellen zur Kommunikation zwischen den einzelnen Baugruppen*

[2]Microcontroller Unit
[3]STMicroelectronics
[4]General Purpose Input/Output

3.1.1. Mikrocontroller

Um einen geeigneten Mikrocontroller für die Steuereinheit zu finden, wird die ARM Cortex-M4 Familie von STM betrachtet. STM bietet hierbei vier Anwendungsgebiete: Wireless, Ultra-low-power, Mainstream und High Performance. Mit bis zu 168 I/Os[5] sollten alle nötigen Pins für das Projekt abgedeckt sein. Darüber hinaus existiert Raum für Änderungen oder Erweiterungen. Die wichtigsten Aspekte in dieser spezifischen Anwendung sind in Tabelle 3.3 dargestellt. Die ZMX-Handsteuerung wird weder über eine WLAN-Anbindung verfügen, noch wird auf besondere Energieeffizienz Wert gelegt, so dass die Typen STM32-WB und STM32-L4/L4+ aus der Auswahl herausfallen. Finanzielle Aspekte bei der Beschaffung von Bauteilen spielen am DESY eine untergeordnete Rolle, wichtig sind vor allem Leistung und Verfügbarkeit.

Anwendung	Wireless	Ultra-low-power	Mainstream	High performance
Typ	STM32-WB	STM32-L4/L4+	STM32F3	STM32F4
Takt	64MHz Dual-Core	80/120 MHz max.	72MHz max.	180MHz max.
Quadratur-Eingang	Nein	Ja	Ja	Ja
SPI[a]	2	3	4	6
I²C[a]	2	4	3	4
UART[a]	1	6	3	4
IOs[a]	72	140	115	168

a) Maximale Anzahl die innerhalb der Familie zur Verfügung steht.

Tabelle 3.3.: *ARM Cortex-M4 Mikrocontroller von STM im Vergleich in ausgewählten Kategorien*

Insgesamt werden 40 Pins für I/Os und Kommunikation verwendet (vgl. Tabelle 3.4). Die geplante Kommunikation zwischen den einzelnen Baugruppen besteht aus zwei SPI Schnittstellen und einer UART Schnittstelle. Außerdem werden zwei Timer im Encodermode benötigt. Dies ermöglicht die direkte Auswertung von Drehencodern.
Aus diesem Grund wurde sich für die High performance Serie STM32F4 entschieden. Mit einer Geschwindigkeit bis zu 180MHz sollten keine Engpässe bei dem Programmablauf im Mikrocontroller entstehen, auch wenn mehrere Schnittstellen zeitnah betrieben werden [16]. In der Vergangenheit hat der Verfasser bereits mit dem Mikrocontroller STM32F407 gearbeitet und gute Erfahrungen gemacht. Aus diesem Grund wird überprüft, ob der Mikrocontroller für die Umsetzung der Anforderungen geeignet ist.

[5] Input/Outputs

Tabelle 3.4 zeigt die genutzten und verfügbaren Pins des STM32F407. Es ist deutlich zu erkennen, dass die gesetzten Anforderungen mit dem Mikrocontroller umgesetzt werden können. Wichtig für mögliche Änderungen ist zu bedenken, dass nicht jede alternative Funktion gleichzeitig implementiert werden kann, da GPIOs teilweise doppelt belegt sind [17, p. 62-70].

Signal	Anzahl d. Pins	Funktion
Taster	7	GPIO
Drehencoder	4	AF, Timer, Encodermode
Taster Drehencoder	2	GPIO
Komm. Display SPI	4	AF, SPI
Komm. Display 8 Bit (Reserve)	7	GPIO
Komm. FPGA	4	AF, SPI
Komm. dig. Potentiometer	2	AF, I2C
Komm. ZMX	2	AF, UART
Debug PC	2	AF, UART
LED	6	GPIO
Summe	40	AF + GPIO

AF: Alternate Function des GPIO

Tabelle 3.4.: *Pin-Nutzung des Mikrocontrollers*

Mit einer Taktfrequenz von 168MHz, bis zu 1MB Speicher und 140 I/Os stellt der STM32F407 eine optimale Lösung dar [17, p. 1]. Des Weiteren bietet STM mit dem STM32F4DISCOVERY ein günstiges und verfügbares Evaluations-Board für erste Tests und Entwicklungen.

Funktion	Genutzt	Verfügbar
AF, Timer, Encodermode	2	4
AF, SPI	2	3
AF, I2C	1	3
AF, UART	2	2

Tabelle 3.5.: *Ressourcennutzung des Mikrocontrollers STM32F407*

3.1.2. FPGA

Das Auslesen von Encodern, Auswerten der Endlagen und das Erzeugen der Impulse für die Endstufe soll durch einen FPGA realisiert werden. Im Folgenden wird eine passende Auswahl für ein Modell getroffen.

Aufgrund von Erfahrungen aus anderen Projekten, wird die Nutzung der FPGAs der Cyclone IV Familie von Intel vorgeschlagen. Intel bietet generell zwei Typen an: die Cyclone IV G FPGAs und die Cyclone IV E FPGAs. Während die G-Typen nur in der Bauform BGA[6] verfügbar sind, können einige E-Typen auch in der Bauform QFP[7] erworben werden. Allgemein ist die Bestückung von ICs[8] in BGA-Bauform deutlich aufwändiger als die der QFP-Bauform. In der Anfangsphase des Projekts erleichtern zudem die seitlich heraus geführten Pins das Aufnehmen von Signalen mit dem Oszilloskop zwecks der Fehlersuche. Aus diesen Gründen, wird sich für die E-Typen in QFP-Bauform entschieden. Mögliche verwendbare FPGAs sind in Tabelle 3.6 zusammengetragen.

Product	EP4CE6	EP4CE10	EP4CE15	EP4CE22
Logic Elements (k)	6	10	15	22
M9K memory blocks	30	46	56	66
Embedded memory (kb)	270	414	504	594
18 x 18 multipliers	15	23	56	66
Global clock networks	10	10	20	20
PLLs	2	2	4	4
I/O voltage levels supported (V)	1.2, 1.5, 1.8, 2.5, 3.3			
User I/O Pins	91	91	81	79
Pin compatible	yes	yes	yes	yes

Tabelle 3.6.: *Vergleich der Cyclone IV E in der Bauform QFP [18].*

Ein Vergleich der Typen untereinander zeigt, dass die Hauptunterschiede in der Speichergröße und in der Anzahl der Multiplizierer liegt. Da die endgültige Größe des Programms nicht bekannt ist und Tools zum Debuggen im FPGA recht speicherintensiv sind, wird der Typ EP4CE10 festgelegt. Mit 10.000 Logikelementen und 46 Memory Blöcken stellt der FPGA eine gute Grundlage für den Einstieg dar. Die vier Typen sind Pin kompatibel, d.h. sollte der Speicher zu klein oder deutlich zu groß sein, können sie untereinander ersetzt werden. Ein wichtiger Vorteil ist die I/O Versorgungsspannung von 3,3 Volt. Um mögliche Daten zwischen FPGA und Mikrocontroller auszutauschen, muss keine Pegelanpassung vorgenommen werden.

[6]Ball Grid Array
[7]Quad Flat Package
[8]Integrated Circuits

3.2. Aufbau der Leiterplatte

Die folgenden zwei Abbildung zeigen die Leiterplatte der Kontrolleinheit. Zusätzlich sind die wichtigsten Bauteile gekennzeichnet.

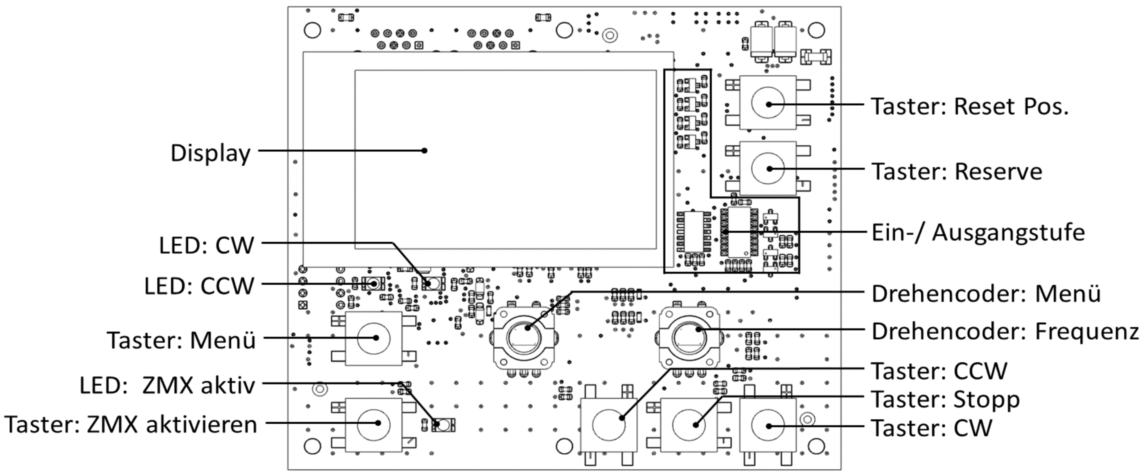


Abbildung 3.1.: *Top-Ansicht der bestückten Leiterplatte von der Kontrolleinheit. Die wichtigsten Bauteile sind gekennzeichnet.*

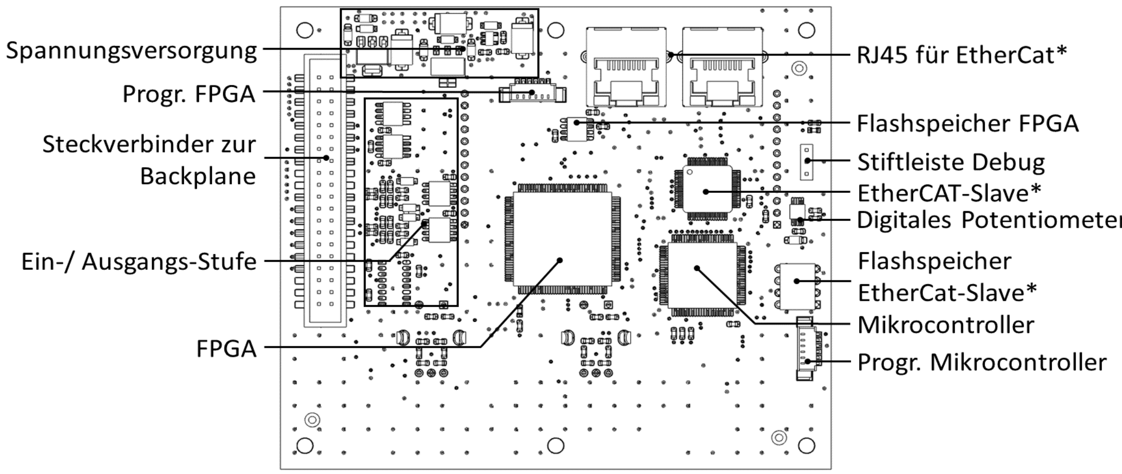


Abbildung 3.2.: *Bottom-Ansicht der bestückten Leiterplatte von der Kontrolleinheit. Die wichtigsten Bauteile sind gekennzeichnet. Für Informationen über Bauteile die mit einem * markiert sind, siehe Abschnitt 5.2.*

3.3. Die Software im Mikrocontroller

Im Folgenden wird auf die Realisierung des Konzepts der Software im Mikrocontroller eingegangen. Es werden die einzelnen Schnittstellen und Protokolle beschrieben.

3.3.1. Menüsteuerung

Die Bedienung der ZMX-Handsteuerung erfolgt über ein Menü, dass auf dem Display visualisiert wird. Um im Menü zu navigieren, wird der Drehencoder „Menü “ auf der Kontrolleinheit benötigt. Für die Auswertung wird der Timer 1 als Encodereingang konfiguriert [19, p. 550-552]. Die dafür benötigten Leitungen sind in Tabelle 3.7 dargestellt.

Bezeichnung	**Richtung**	**Funktion**
SW1_A	Eingang	Rechtecksignal A
SW1_B	Eingang	Rechtecksignal B
SW1_T	Eingang	Taster dient zum Auswählen des Parameters

Tabelle 3.7.: *Leitungen vom Drehinkrementalgeber zum Mikrocontroller*

Die Encodereingänge sind zusätzlich mit einem digitalen Tiefpass-Filter konfiguriert, um dem mechanischen Prellen entgegen zu wirken [20, p. 13].

3.3.2. Speicherung der eingestellten Endstufenparameter

Damit zuvor eingestellte Parameter der ZMX-Endstufe auch nach einem Neustart der ZMX-Handsteuerung zur Verfügung stehen, wird ein nichtflüchtiger Speicher benötigt. STM stellt für die Mikrocontroller der Familie STM32F40x eine EEPROM[9] Emulation im Flash-Speicher bereit [21, p. 1]. Gespeichert werden die Werte:

- Laufstrom
- Stoppstrom
- Schrittauflösung
- Endlageneinstellungen
- Impulsfrequenz

[9] Electrically Erasable Programmable Read-Only Memory

Sollten noch keine Werte beim Start der ZMX-Handsteuerung gespeichert sein, werden Default-Werte geladen.

3.3.3. Schnittstelle für die Kommunikation zum Display

Die Kommunikation zum Display wird über eine Hardware-SPI-Schnittstelle realisiert, d.h. der Mikrocontroller bietet GPIOs an die passend konfiguriert werden können [19, p. 882]. Die dafür benötigten Leitungen sind in Tabelle 3.8 zusammengefasst.

Bezeichnung	Richtung	Funktion
SCK[10]	Eingang	Schiebetakt für den Slave
NSS[11]	Eingang	Aktiviert den Datenempfang beim Slave
MOSI[12]	Ausgang	Datenleitung vom Master zum Slave

Tabelle 3.8.: *Leitungen der SPI-Schnittstelle vom Mikrocontroller zum Display.*

Das Display bietet die Möglichkeit Daten über SPI auszulesen, wird jedoch im Rahmen dieser Arbeit nicht eingesetzt. Der Ablauf der Kommunikation erfolgt gemäß Abbildung 3.3. Als erstes wird ein Start Byte gesendet das aus Synchronisationsbits, R/W[13] Anweisung und RS[14]-Bit besteht. Aus den zu sendenden Daten werden mit Hilfe einer Funktion ein „upper “ und „lower “ Byte erzeugt. Anschließend werden die Daten an das Display gesendet.

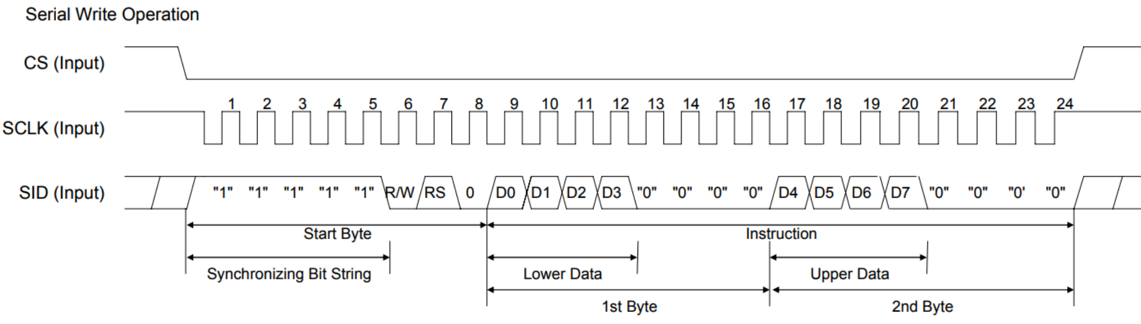


Abbildung 3.3.: *Timing-Diagramm der Kommunikation vom Mikrocontroller zum Display [22, p. 27].*

Nach jeder gesendeten Anweisung wird die geforderte Ausführungszeit gewartet. Diese entspricht beim Senden eines Zeichens mindestens 43µs und beim Löschen des gesamten

[10]Shift Clock
[11]Active Low Slave Select
[12]Master Out Slave In
[13]Read/Write
[14]Register Select

Displays mindestens 1.53ms. Erst danach darf eine neue Anweisung gesendet werden [15, p. 29-32].

3.3.4. Schnittstelle für die Kommunikation zum FPGA

Die Kommunikation zum FPGA wird über eine SPI-Schnittstelle realisiert. Der Mikrocontroller ist als Master und der FPGA als Slave definiert. Für die Schnittstelle sind vier Leitungen notwendig (vgl. Tabelle 3.9).

Bezeichnung	**Richtung**	**Funktion**
SCK	Eingang	Schiebetakt für den Slave
NSS	Eingang	Aktiviert den Datenempfang beim Slave
MISO[15]	Eingang	Datenleitung vom Slave zum Master
MOSI	Ausgang	Datenleitung Vom Master zum Slave

Tabelle 3.9.: *Leitungen der SPI-Schnittstelle zwischen FPGA und Mikrocontroller aus Sicht des Masters.*

Es ist möglich im Full-Duplex Betrieb zu arbeiten, d.h. es können gleichzeitig Daten gesendet und empfangen werden. Bei der Datenübertragung über SPI kann ein Slave den Takt zur Kommunikation nicht selber generieren. Das bedeutet, dass der Master beim Auslesen der Daten aus dem Slave den Takt bereit stellen muss. Um eine hinreichend schnelle Kommunikation zu gewährleisten, ist die Frequenz für den Schiebetakt auf 5.25MHz eingestellt. Eine genaue Beschreibung der Datenübertragung zum FPGA befindet sich im Kapitel 3.4.2.

3.3.5. Parametrierung der Endstufe

Die Anbindung der ZMX-Endstufe erfolgt über eine RS-485-Schnittstelle. Dabei handelt es sich um eine asynchrone serielle Datenübertragung. Hierfür kann der UART-Baustein im Mikrocontroller verwendet werden. Der Mikrocontroller stellt GPIOs bereit, die entsprechend der Anforderung konfiguriert werden können [19, p. 1010]. Für die Kommunikation werden zwei Pins benötigt, die laut Tabelle 3.10 parametriert sind.

Bezeichnung	**Richtung**	**Funktion**
Rx	Eingang	Leitung für den Empfang von Daten
Tx	Ausgang	Leitung für das Senden von Daten

Tabelle 3.10.: *Leitungen der UART-Schnittstelle vom Mikrocontroller zur ZMX-Endstufe.*

[15] Master In Slave Out

Für den gesicherten Datenaustausch zwischen Mikrocontroller und ZMX-Endstufe, hat die Firma Phytron ein Protokoll vorgegeben [8, p. 6]. Die Einstellung für die UART-Schnittstelle sind wie folgt:

- Asynchrone Übertragung, 8 Bits/Byte, 1 Stoppbit, 1 Pritybit
- Übertragungsgeschwindigkeit: 57600 Baud.

Das Telegrammformat muss der folgenden Form entsprechen:

<STX> <Adresse_H> <Adresse_L> <Befehl> <Wert> : <csh> <csl> <ETX>

Bezeichner	**Bedeutung**
STX	Startbyte, 0x02
Adresse_H	Höherwertiges Byte der Geräteadresse
Adresse_L	Niederwertiges Byte der Geräteadresse
Befehl	Befehlsbyte
Wert	Datenbyte/s
:	Begrenzer
csh	„High“ -Nibble der Prüfsumme
csl	„Low“ -Nibble der Prüfsumme
ETX	Endzeichen, 0x03

Tabelle 3.11.: *Erläuterung des Protokolls zur Kommunikation mit der ZMX-Endstufe [8, p. 6].*

Im Mikrocontroller werden die Daten entsprechend aufbereitet und an die ZMX-Endstufe gesendet.

3.3.6. Programmablauf

Die Software des Mikrocontrollers besteht aus zwei Teilen. Die Initialisierung und dem Hauptprogramm.
Zuerst wird der Mikrocontroller initialisiert, d.h. es werden die GPIOs, der Timer zum Senden von Daten an das Display und die Kommunikationsschnittstellen konfiguriert (vgl. Abb.3.4). Es werden selbst definierte Zeichen[16] [15, p. 3] an das Display übertragen, um sie später nutzen zu können.

[16] c und w negiert, c und w invertiert und C und W durchgestrichen

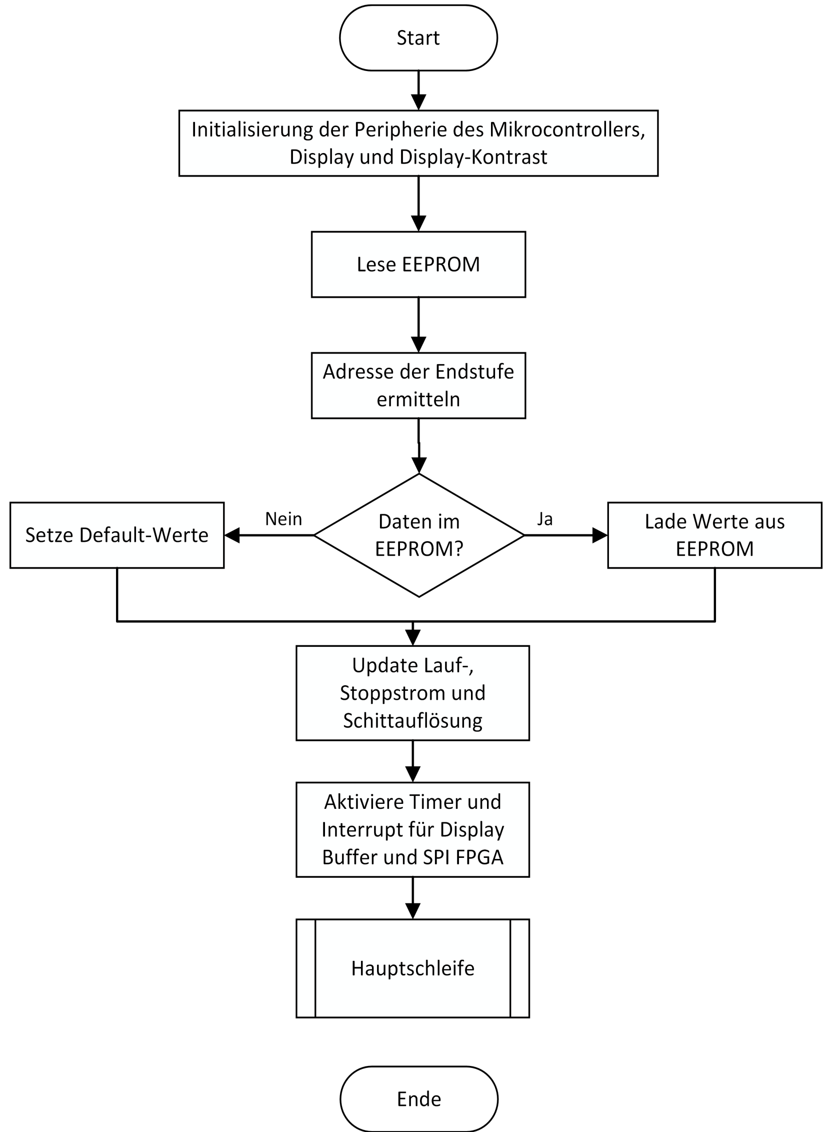


Abbildung 3.4.: *Programmablauf der Initialisierung im Mikrocontroller. Es werden die genutzten Funktionen konfiguriert und die ZMX-Handsteuerung in einen sicheren Anfangszustand gebracht.*

Anschließend werden die gespeicherten Parameter aus dem emulierten EEPROM geladen und für die Übertragung zur ZMX-Endstufe bereit gestellt. Die Zähler der Encoder und der Schritte im FPGA werden zurückgesetzt. Um einen definierten Anfangszustand zu erhalten, wird die Impulserzeugung gestoppt und die ZMX-Endstufe deaktiviert. Als letzter Punkt werden die Timer und Interrupts für das Schreiben von Daten in den Buffer für das Display und für die Datenübertragung zum FPGA initialisiert. Das Programm wird in der Hauptschleife (vgl. Abb. 3.5) fortgesetzt.

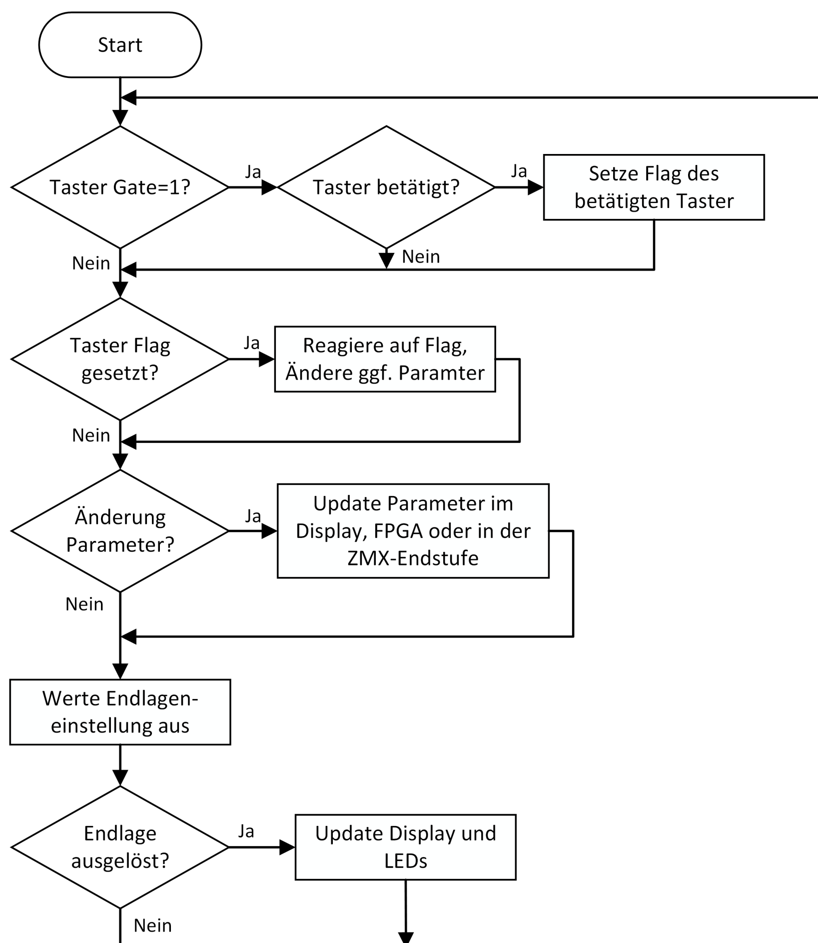


Abbildung 3.5.: *Programmablauf innerhalb der Hauptschleife im Mikrocontroller. Dieser Programmteil reagiert auf Änderungen durch Eingabe des Nutzers: Betätigung der Taster CW, CCW, Stopp und Entregen (dieser Taster dient der Aktivierung und Deaktivierung der Endstufe).*

Der Programmablauf im Betrieb ist in zwei Teile unterteilt, der Hauptschleife und dem Interrupt gesteuerten Teil. Innerhalb der Hauptschleife wird auf die Betätigung der Taster reagiert. Resultiert daraus eine Änderung eines Parameters, wird das passende Bit gesetzt. Löst ein Interrupt aus, z.B. für die Aktualisierung der Parameter in dem Display, werden die dazugehörigen Bits überprüft. Aktualisierte Parameter werden in den Sendebuffer für das Display geschrieben. Daten die zum FPGA übertragen werden sollen, werden ebenfalls in Variablen gespeichert. Erst mit dem Interrupt zum Senden von Daten, werden die Daten in das adressierte Register geschrieben. Daten an die ZMX-Endstufe werden direkt bei Änderung des Werts übertragen.

Insgesamt werden fünf Timer gesteuerte Interrupts genutzt. Mit Hilfe der Gleichung 3.3.6.1 [23, p. 10] können Prescaler bzw. Periode bei geeigneter Annahme berechnet werden.

$$f_{up} = \frac{f_{clk}}{(Periode + 1) \cdot (Prescaler + 1)} \tag{3.3.6.1}$$

Ein Interrupt z.B. soll alle 20 ms ausgelöst werden. Das bedeutet eine Frequenz von 50Hz, die der Timer 5 vorgeben wird. Laut Datenblatt ist die dort anliegende Frequenz 84MHz [19, p. 217]. Für den Prescaler wird der Wert 499 angenommen, so dass die Periode mit der umgestellten Formel berechnet werden kann:

$$Periode = \left(\frac{f_{clk}}{(Prescaler + 1) \cdot f_{up}} \right) - 1 \tag{3.3.6.2}$$

$$= \left(\frac{84MHz}{(499 + 1) \cdot 20MHz} \right) - 1 \tag{3.3.6.3}$$

$$= 4999 \tag{3.3.6.4}$$

In Tabelle 3.12 sind die genutzten Interrupts mit zugehöriger Konfiguration und Anwendung dargestellt.

Name	EEPROM	SPI-Display	SPI-FPGA	Display-Buffer	Drehencoder
Timer	2	3	4	5	6
Zeit	250ms	1us	20ms	20ms	50ms
Periode	4199	41	499	499	499
Prescaler	4999	1	3359	3359	8399
Priorität	4	0	2	1	3
Subpriorität	0	0	0	0	0

Tabelle 3.12.: *Konfiguration der Interrupts im Mikrocontroller.*

Löst der Interrupt für die Datenübertragung an das Display aus, wird überprüft ob Daten im Buffer vorhanden sind. Ist das der Fall, wird die Übertragung gestartet. Das Display wird selektiert und die erste Anweisung (Adresse) gesendet. Nach jeder Anweisung muss die Ausführungszeit im Display beachtet werden. Während dieser Zeit darf keine neue Anweisung an das Display erfolgen. Wird der Interrupt nach einer Mikrosekunde wieder aufgerufen, wird als erstes überprüft ob die Ausführungszeit abgelaufen ist. Ist das nicht der Fall, springt der Mikrocontroller zurück in das Hauptschleife. Auf diese Weise können während dieses Zeitraums andere Anweisungen ausgeführt werden. Abbildung 3.6 zeigt den Programmablauf im Interrupt.

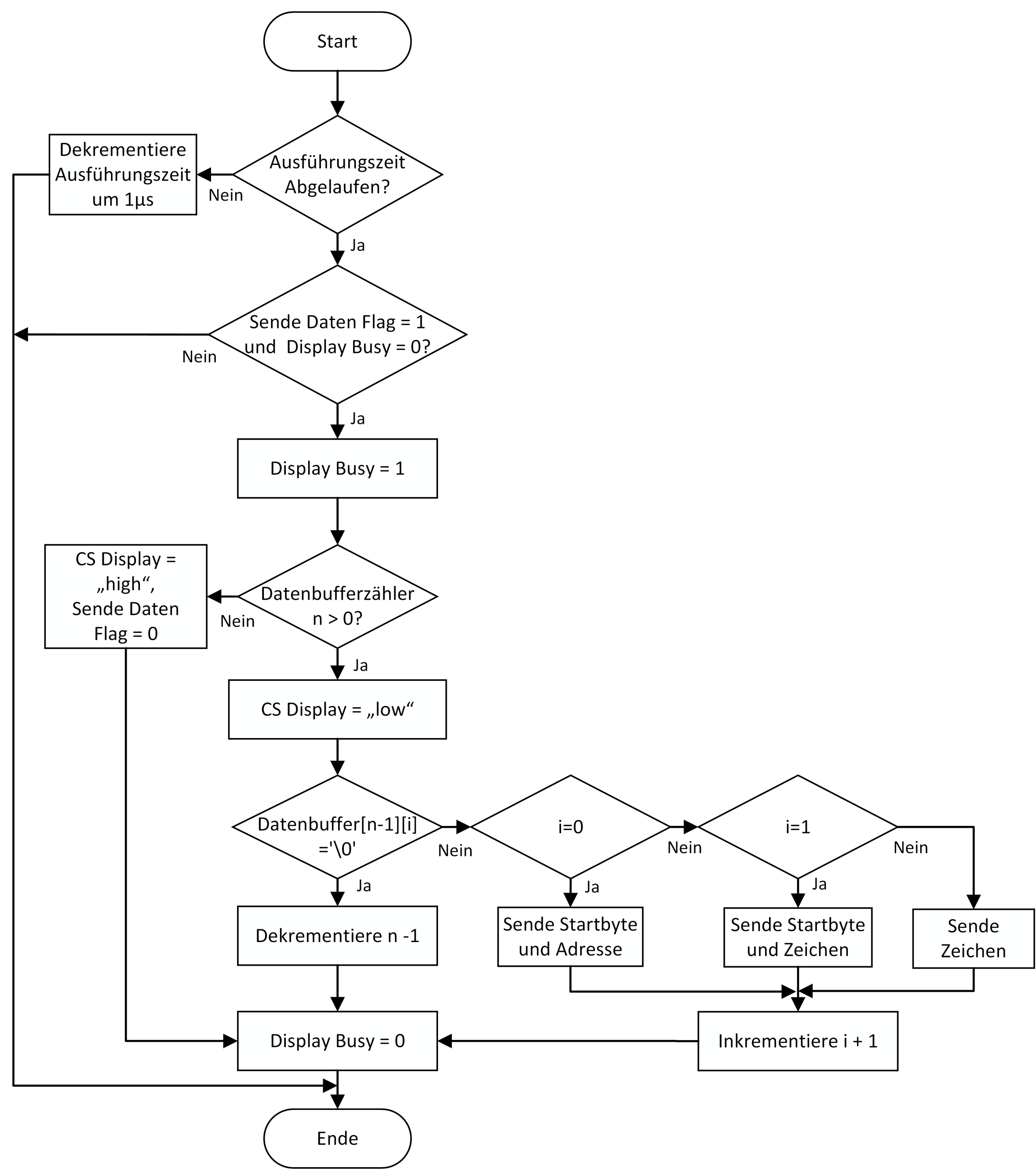


Abbildung 3.6.: *Dargestellt ist der Programmablauf für den Interrupt zum Senden von Daten an das Display. Erst wenn die Ausführungszeit nach dem Senden einer Anweisung an das Display abgelaufen ist, darf eine neue Anweisung gesendet werden. Nach jeder Übertragung wird durch die Funktion für das Senden von Daten über SPI der Zähler für die Ausführungszeit zurückgesetzt. So können während dieser Zeit andere Programmteile ausgeführt werden.*

Der Display-Buffer arbeitet nach dem FIFO[17]-Prinzip. Die Daten die zuerst in den Buffer geschrieben werden, werden auch als erstes an das Display übertragen. Abbildung 3.7 zeigt den Aufbau des Sendebuffers. Geänderte Anzeigewerte werden im Timer 5 Interrupt aufbereitet und in den Buffer geschrieben. Wird dort die Flag zum Senden von Daten gesetzt, startet beim nächsten Sende-Interrupt die Datenübertragung zum Display.

Buffer[n]	Adresse	D[0]	D[1]	D[2]	...	D[i-1]	D[i]
0	0x80	I	R	u	...	n	\0
1	0xA0	I	S	t	...	p	\0
2	0xCB	+	1	\0	...		
...	...	...	...	...	...	...	...
n	0xEB	1	5	0	...	z	\0

Abbildung 3.7.: *Aufbau des Display-Buffers: Zuerst wird die Adresse gespeichert, gefolgt von den Daten und dem Nullterminator. Zuerst in den Buffer geschriebene Daten, werden auch als erstes an das Display übertragen.*

Die Auswertung des FPGAs wird zyklisch alle 20ms ausgeführt. Es wird der Zustand der Endlagen ausgelesen, sowie die Werte des Encoder- und Schrittzählers. Ist das Flag zum senden von Daten gesetzt, werden sie in das passende Register im FPGA geschrieben. Der Programmablauf ist in Abbildung 3.8 dargestellt.

[17] First In First Out

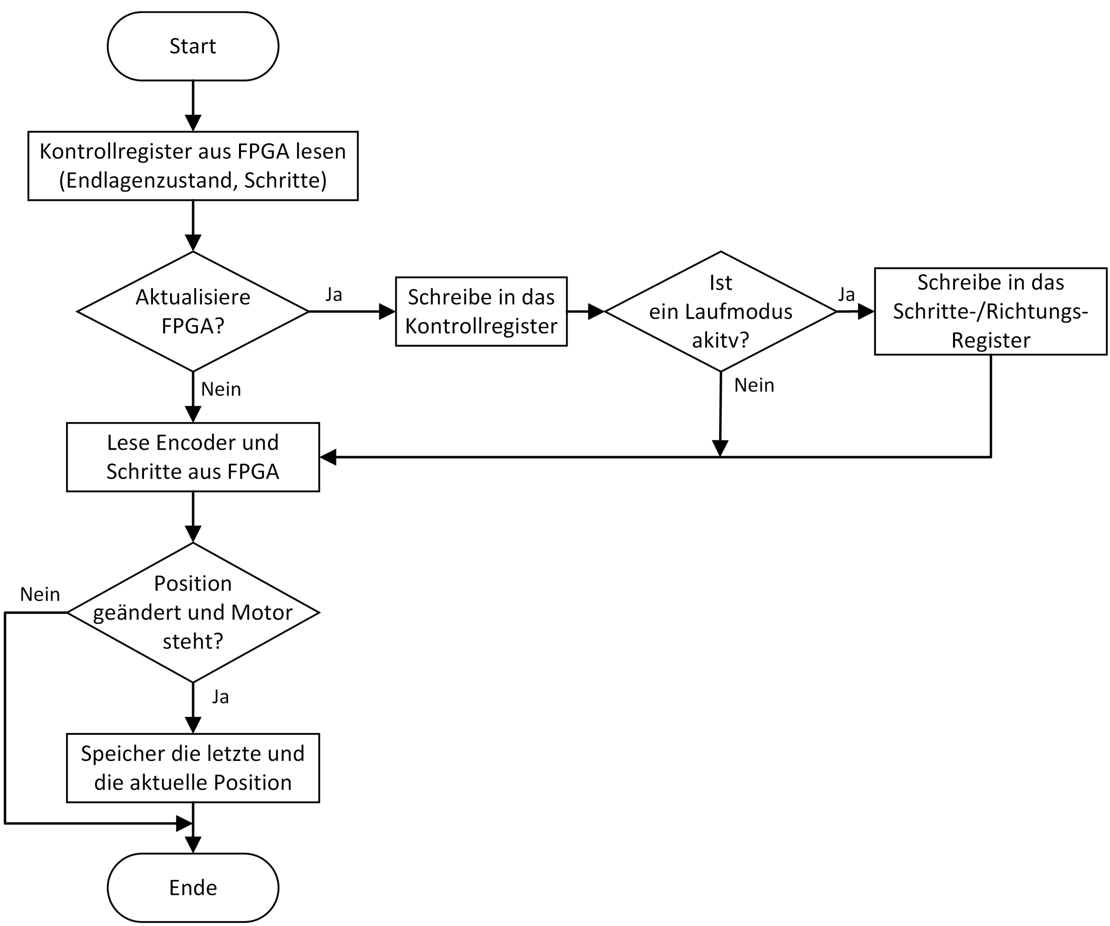


Abbildung 3.8.: *Programmablauf für den Interrupt zum Senden von Daten zum FPGA. Löst der Interrupt aus, wird als erstes Kontrollregister ausgelesen. Anschließend werden aktualisierte Daten an den FPGA übertragen. Als letztes werden die Encoder und gezählten Schritte ausgelesen.*

Der Timer 2 Interrupt ist für das Speichern von Parametern im EEPROM zuständig. Wird für 30s die Impulsfrequenz nicht geändert, oder das Menü verlassen, werden die Werte gespeichert. Im Interrupt der durch den Timer 6 ausgelöst wird, findet alle 50ms die Auswertung der Drehencoder statt. Bei Änderung wird das passende Flag gesetzt und neue Werte zum FPGA, zu der ZMX-Endstufe oder an das Display übertragen.

3.4. Aufbau der Firmware im FPGA

Im Folgenden wird die Realisierung der Firmware im FPGA beschrieben.

3.4.1. Auswerten von Encodern

Abbildung 3.9 zeigt das Konzept zum Auswerten von Encodern. Die Eingänge A und B, jeweils Bits, werden auf einen zwei Bit Vektor x abgebildet. Ein Zustandsautomat reagiert auf die Änderung in dem Vektor. Ist der Zustand "00" aktiv, heißt dies das beide Eingänge „low“ sind. Wird nun angenommen das A „high “ und B „low“ ist, so nimmt x den Wert „10“ an. Der Zustand wechselt nach "10" . Folglich hat sich der Motor um einen Encoder-Schritt bewegt und der Zähler wird um ein inkrementiert [24][18].

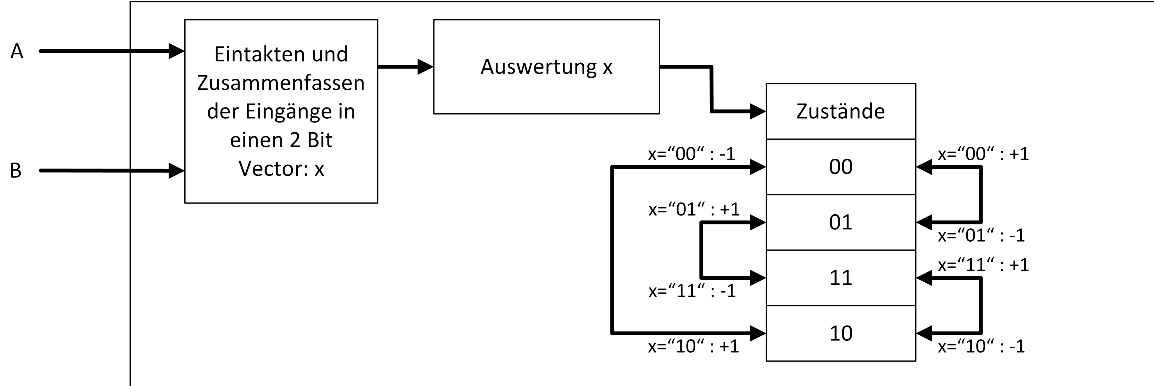


Abbildung 3.9.: *Auswertung von Quadratur-Encodern im FPGA. Die Eingänge A und B werden zusammengefasst und ausgewertet. Anhängig davon welcher Zustand aktiv ist, wird inkrementiert (+1) oder dekrementiert (-1) und der Zustand entsprechend gewechselt.*

3.4.2. Verarbeitung der Kommunikation über SPI im FPGA

Um die Kommunikation über die SPI-Schnittstelle zu beginnen, muss das Signal NSS auf den Pegel „low “ geschaltet werden. Mit dem Takt auf der SCK-Leitung werden die Daten von der MOSI-Leitung übernommen, bzw. auf die MISO-Leitung geschoben. Sobald das Signal NSS auf den Pegel „high“ gesetzt wird, ist die Kommunikation beendet. Abbildung 3.10 zeigt die Kommunikation zwischen FPGA und Mikrocontroller über SPI. Hierbei wird ein festes Format verwendet, welches aus insgesamt 6 Bytes (vgl. Tabelle 3.13) besteht: Als erstes wird die

[18]Der Quellcode wurde von [24] übernommen und angepasst. Änderungen sind im Quellcode vermerkt.

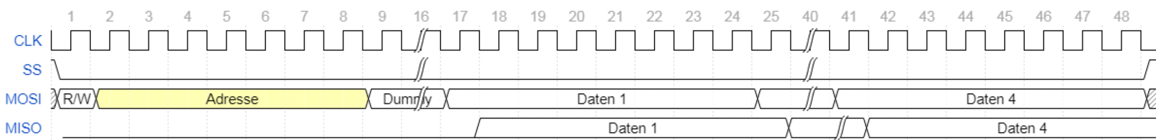


Abbildung 3.10.: *Timing-Diagramm der Kommunikation zwischen FPGA und Mikrocontroller. Es wird die Read-/Write-Anweisung mit Adresse und vier Byte Daten übertragen.*

R/W Anweisung übertragen, gefolgt von der Adresse des Registers, das beschrieben oder gelesen werden soll. Das erste Byte muss den Aufbau in Tabelle 3.14 entsprechen.

Byte 1	**Byte 2**	**Byte 3**	**Byte 4**	**Byte 5**	**Byte 6**
R/W, Adresse	Dummy	Daten 1	Daten 2	Daten 3	Daten 4

Tabelle 3.13.: *Datenframe zur Kommunikation mit dem FPGA über SPI.*

Bit	7	6	5	4	3	2	1	0
Anweisung	R/W	Adresse						
	= 0 : Lese Register = 1 : Schreibe Register							

Tabelle 3.14.: *Aufbau des ersten Bytes, das die R/W Anweisung und die Adresse enthält.*

Anschließend folgt ein Dummy-Byte. Beim Ausführen einer Leseanweisung müssen die Daten aus dem passenden Register in das Ausgangsregister geladen werden. Das Dummy-Byte sorgt dafür, dass genügend Zeit für den Vorgang zur Verfügung steht. Im Anschluss können vier Bytes an Daten geschrieben oder gelesen werden. Tabelle 3.15 zeigt die im FPGA implementierten Register mit Adresse, Zugriffsmodus und Beschreibung der Funktion.

Adresse	**Register**	**Zugriff**	**Beschreibung**
0x00	Firmware	R	Lese aktuelle Firmware Version aus
0x01	Inkrementalgeber	R	Lese Encoder Wert aus
		W	Reset Encoder
0x02	Steuerregister	R	Lese Steuerregister und Endlagen
		W	Schreibe Steuerregisterr
0x03	n-Schritte Sollwert	W	Schreibe n-Schritte Sollwert
0x04	Impulsfrequenz	W	Schreibe Impulsfrequenz
0x05	Impulszähler	R	Lese Impulszähler
		W	Reset Impulszähler

Tabelle 3.15.: *Register im FPGA mit Adresse und deren Zugriffsmodus*

Das Empfangen bzw. Versenden von Daten ist in zwei Programmteile unterteilt. Der erste Teil ist für die Auswertung des Schiebetakts zuständig (vgl. Abb. 3.11). Es werden die Takte gezählt, die durch den Mikrocontroller generiert werden. Mit steigender Flanke werden Daten von der MOSI Leitung übernommen, bzw. mit fallender Flanke auf die MISO Leitung gelegt. Das ist zwingend notwendig, damit dem Mikrocontroller die Daten rechtzeitig zur Verfügung stehen. Als erstes wird das MSB[19] gesendet bzw. empfangen.

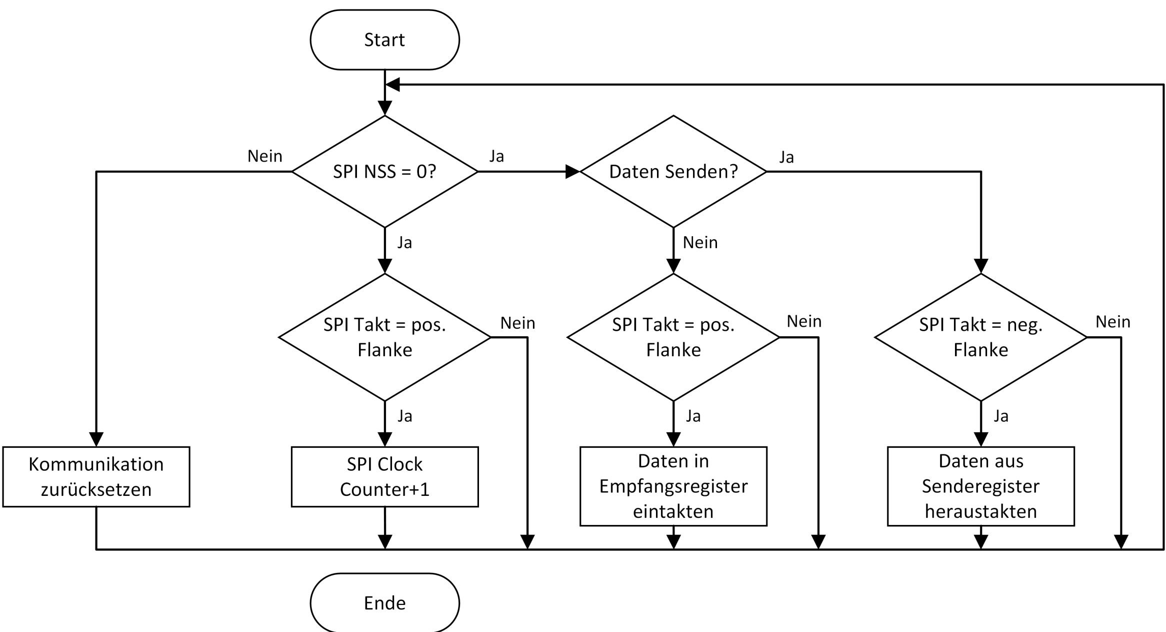


Abbildung 3.11.: *Programmablauf für das Empfangen bzw. Senden von Daten über SPI. Der FPGA ist als Slave aufgebaut und benötigt dementsprechend einen externen Takt. Daten werden mit steigender Flanke empfangen und bei fallender Flanke gesendet.*

Der zweite Programmteil befasst sich mit der Aufbereitung empfangener Daten. Mit Hilfe einer State-Machine werden sie entsprechend aufbereitet. Dabei werden dem Modul für die Datenverarbeitung die Lese- Schreibanweisung, die Registeradresse und ggf. die Daten einzeln zur Verfügung gestellt. Abbildung 3.12 zeigt den Ablauf, welcher im FPGA implementiert wurde.

[19]Most Significant Bit

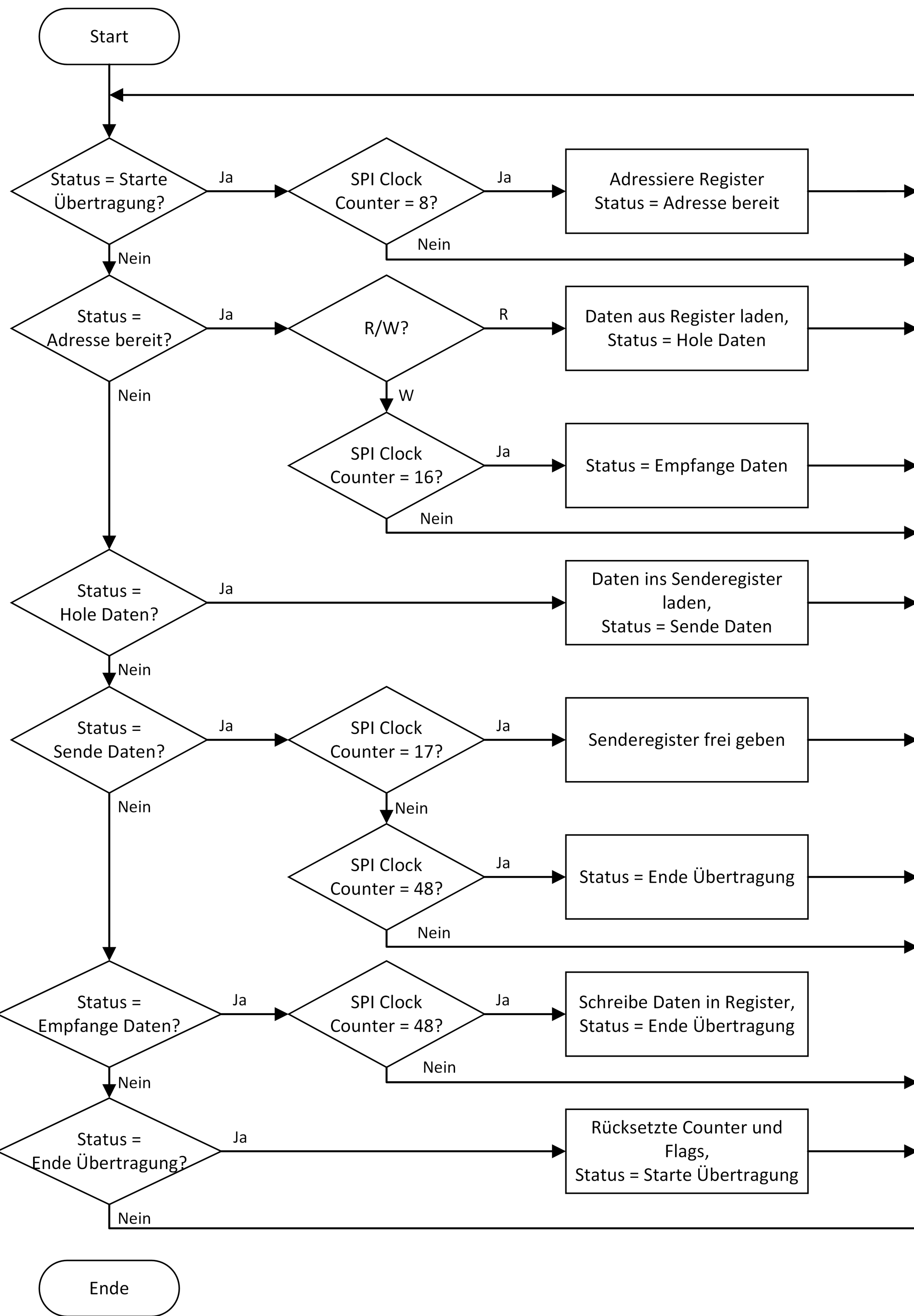


Abbildung 3.12.: *Programmablauf der State-Machine für die Kommunikation über SPI.*

3.4.3. Datenverwaltung

Stellt die SPI-Kommunikation Daten bereit wird überprüft, ob es sich dabei um neue Daten handelt. Dies ist der Fall, wenn sich die Adresse, die R/W-Anweisung oder die Daten geändert haben. Handelt es sich dabei um Schreibzugriff, werden die empfangenen Daten in das adressierte Register geschrieben. Sollen Daten gelesen werden, stellt die Datenverwaltung sie der Schnittstelle zur Kommunikation über die Schnittstelle SPI zur Verfügung. Der Programmablauf ist in Abbildung 3.13 veranschaulicht.

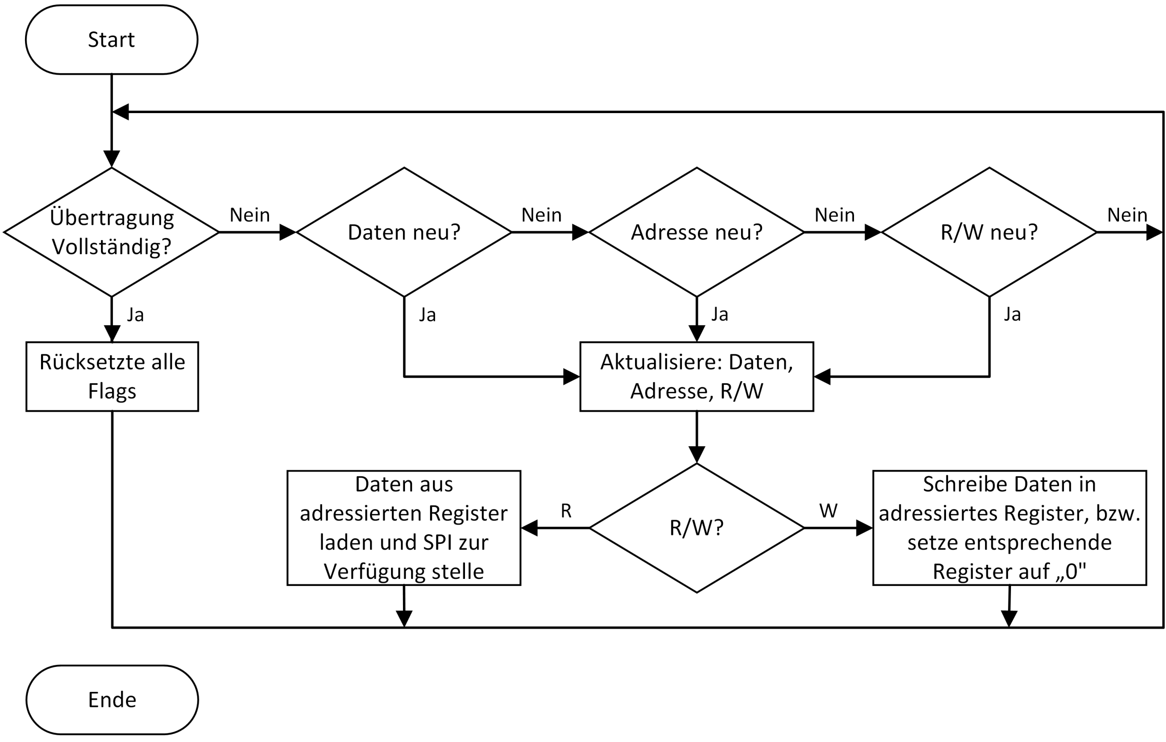


Abbildung 3.13.: *Programmablauf zur Datenverwaltung im FPGA: Werden neue Daten vom SPI-Slave bereit gestellt, findet die Verteilung an das passende Register statt.*

3.4.4. Frequenzerzeugung im FPGA

Um die Impulsfrequenz für die ZMX-Endstufe im FPGA zu erzeugen, wird eine DDS genutzt. Hierbei wird je nach eingestellter Ausgangsfrequenz ein Zähler um ein Vielfaches von Cnt_{min} inkrementiert. Erreicht der Zähler $2^{n-1}/2$, wird der Ausgang auf „high“ geschaltet. Entspricht der Zähler 2^n, kommt es zu einen Überlauf und er wird wieder auf 0 gesetzt. Der Ausgang wird „low“ gesetzt und die Periode beginnt von neuem [25, p. 5]. Der Programmablauf ist in Abbildung 3.14 dargestellt.

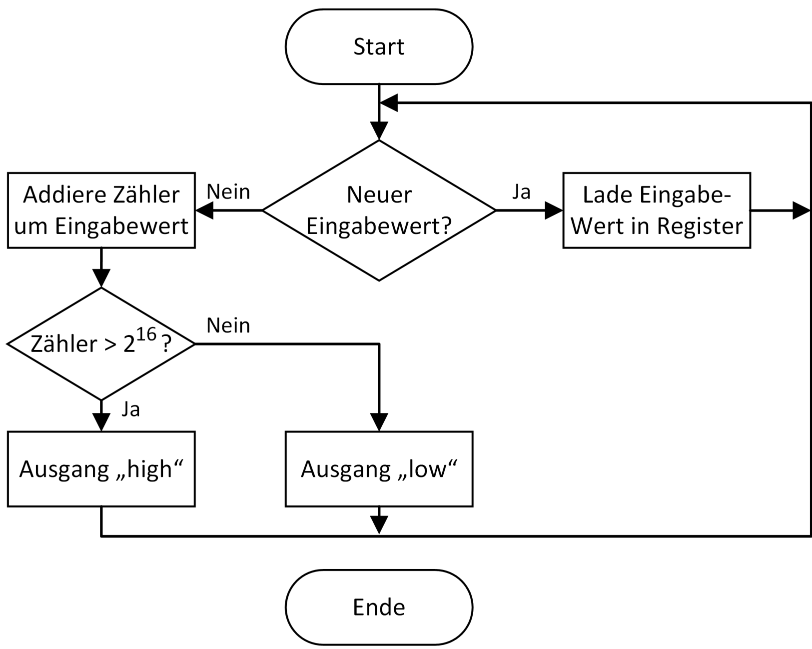


Abbildung 3.14.: *Programmablauf vom DDS. Bei jedem Durchlauf wird der Zähler um den Eingabewert erhöht und verglichen, ob der Zählerstand größer als 2^{32} ist und dementsprechend der Ausgang geschaltet.*

Mit Hilfe der Gleichung 3.4.4 [25, p. 14] lässt sich der minimale Wert, um den der Zähler erhöht werden muss berechnen.

$$Cnt_{min} = \frac{2^n \cdot fout_{min}}{f_{Takt}} \tag{3.4.4.1}$$

In diesem Fall soll sich die Frequenz von 1Hz bis 100kHz einstellen lassen. Die Taktfrequenz vom FPGA beträgt 100MHz und der Zähler ist 32-Bit groß.

$$Cnt_{min} = \frac{2^n \cdot fout_{min}}{f_{Takt}} \tag{3.4.4.2}$$

$$= \frac{2^{32} \cdot 1Hz}{100MHz} \tag{3.4.4.3}$$

$$= 42,9496 \tag{3.4.4.4}$$

Der minimale Wert um den der Zähler inkrementiert werden darf, beträgt laut Berechnung 42,9496. Zur Erzeugung einer Frequenz von beispielsweise 1.5 kHz wird in Gleichung 3.4.4 f_{out}=1500 Hz gesetzt.

$$= \frac{2^{32} \cdot 1500Hz}{100MHz} \tag{3.4.4.5}$$

$$= 64424,509 \tag{3.4.4.6}$$

Diese Rechenoperation wird im Mikrocontroller ausgeführt. Es wird an den FPGA der auf den nächsten ganzzahligen Wert gerundete *Cnt* übertragen.

3.4.5. Impulserzeugung

Innerhalb der Impulserzeugung findet sowohl das Generieren der Pulse für die ZMX-Endstufe, als auch das Auswerten der Endlagen statt. Der Programmablauf ist in Abbildung 3.15 dargestellt. Als erstes wird das Schritt- und Richtungsregister ausgewertet. Die Drehrichtung des Motors wird somit festgelegt. Das Register enthält ebenfalls die Anzahl der Schritte, welche in eine Richtung erzeugt werden sollen. Durch das Kontrollregister wird die ZMX-Endstufe aktiviert oder deaktiviert. Im Kontrollregister werden ebenfalls die Endlageneinstellungen und der Laufmodus festgelegt.

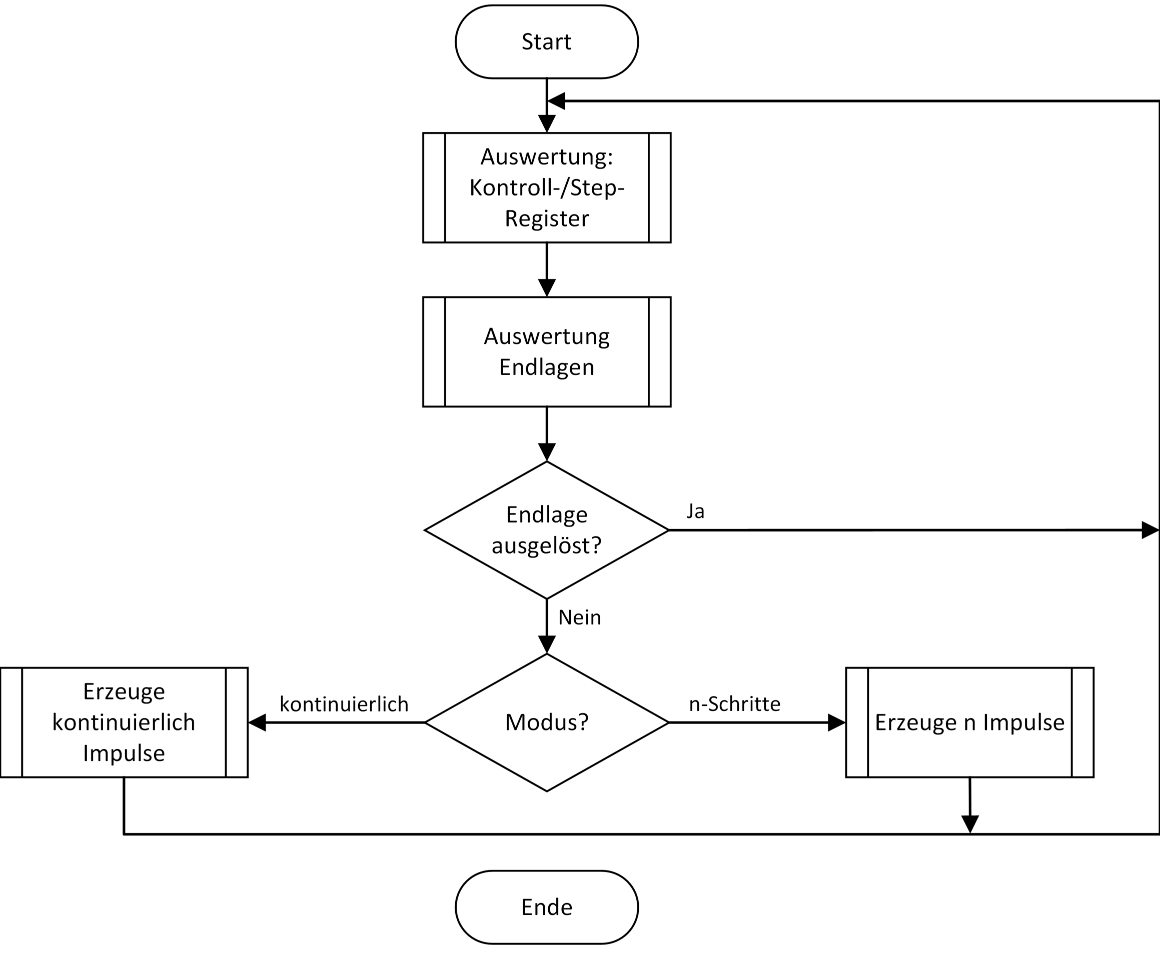


Abbildung 3.15.: *Programmablauf des Bausteins für die Impulserzeugung. Nachdem die Kontrollregister und Endlagen ausgewertet wurden, werden je nach Einstellung die Impulse für die ZMX-Endstufe erzeugt.*

Name	Bit	Wert [dezimal]	Funktion
Mode	0 - 1	0	Stop Impulserzeugung
		1	kontinuierlicher Modus
		2	n-Schritt-Modus
CCW	8 - 9	0	Reset Einstellungen
		1	active high
		2	acitve low
		3	deaktivert
CW	12 - 13	0	Reset Einstellungen
		1	active high
		2	acitve low
		3	deaktivert
ZMX	24	0	Endstufe deaktiviert
		1	Endstufe aktiviert
Reset	25	0	Endstufe Reset deaktiviert
		1	Endstufe Reset aktiviert

Tabelle 3.16.: *Bedeutung der einzelnen Bits in dem Kontrollregister mit den einstellbaren Funktionen.*

Name	Bit	Wert [dezimal]	Funktion
Steps	0 - 23	0 bis 16777216	Anzahl der Schritte
Dir	31	0	Laufrichtung CCW
		1	Laufrichtung CW

Tabelle 3.17.: *Bedeutung der einzelnen Bits in dem Schritt- und Richtungsregister mit den einstellbaren Funktionen.*

Nachdem der Laufmodus und die Richtung parametriert wurden, wird überprüft, ob eine Endlage ausgelöst hat. Dies dient als direkte Schaltbedingung für das Erzeugen der Impulse für die ZMX-Endstufe. Hat keine Endlage ausgelöst, wird mit der Impulserzeugung fortgefahren. Der kontinuierliche Modus (vgl. Abb. 3.16) wird so lange durchlaufen, bis dieser entweder vom Nutzer, oder durch das Auslösen einer Endlage gestoppt wird. Der Unterschied zwischen kontinuierlichem und n-Schritte Modus (vgl. Abb. 3.17) besteht darin, dass der Programmablauf im n-Schritte Modus n-mal durchlaufen werden muss und die Impulserzeugung danach beendet wird.

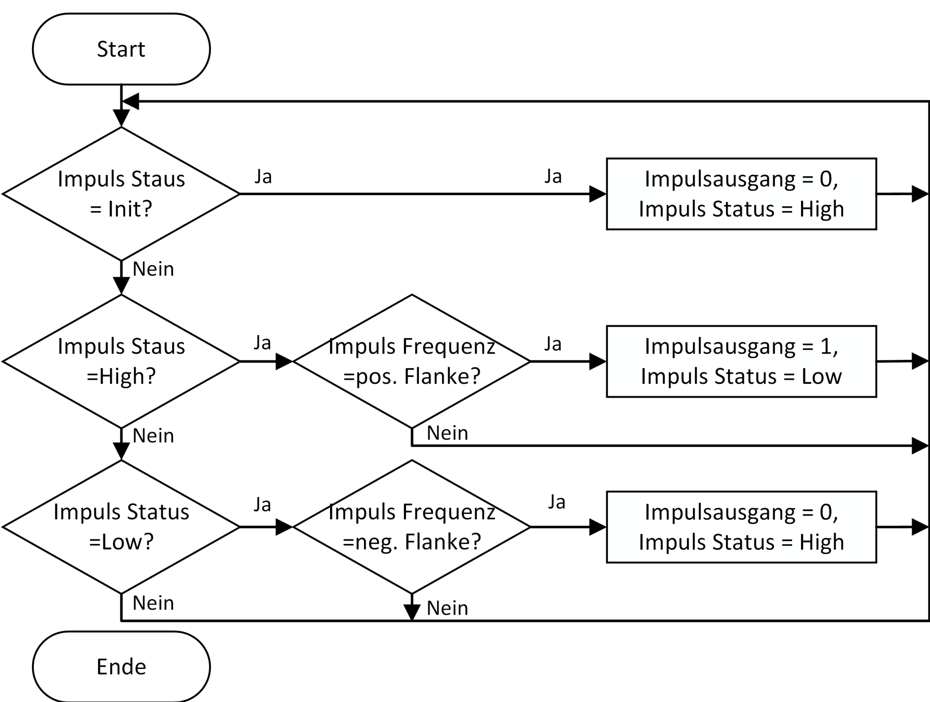


Abbildung 3.16.: *Programmablauf für die kontinuierliche Erzeugung von Impulsen. Der Modus wird so lange durchlaufen bis die Impulserzeugung gestoppt wird. Dies kann entweder durch den Nutzer, oder durch das Auslösen einer Endlage geschehen.*

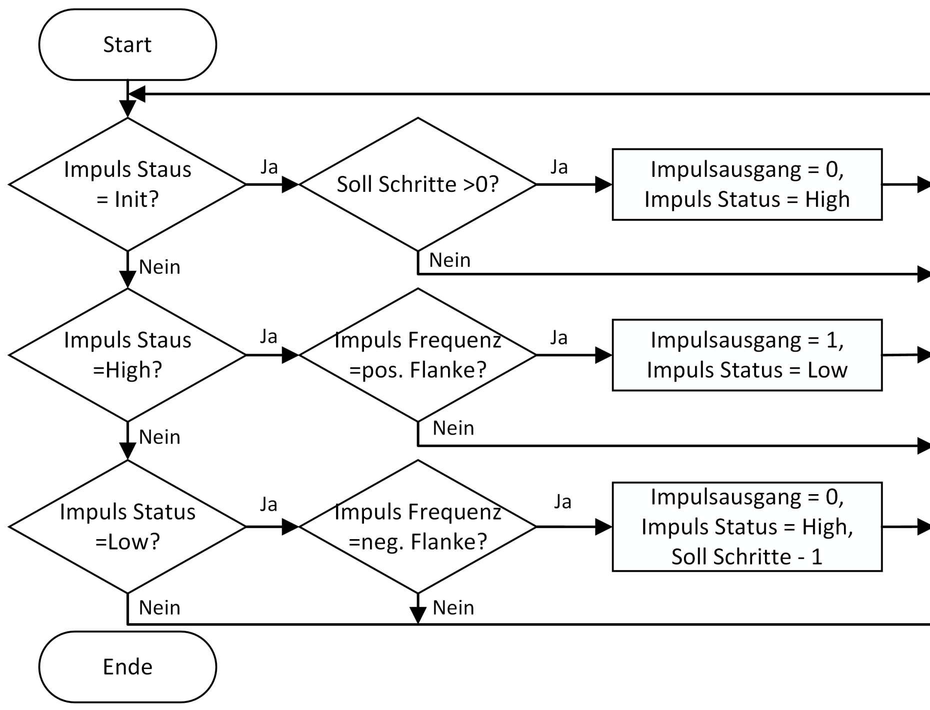


Abbildung 3.17.: *Programmablauf für die Erzeugung von n-Schritten. Wird der Modus n-mal durchlaufen, hört die Impulserzeugung auf. Das Auslösen einer Endlage, oder der Nutzer kann die Impulserzeugung vorzeitig beenden.*

3.4.6. Impulszähler

Wird ein Impuls im FPGA erzeugt, so detektiert der Impulszähler die steigende Flanke. Zusätzlich findet eine Überprüfung der Drehrichtung statt, um dann entsprechend mit einem Zähler aufwärts (im Uhrzeigersinn drehend) oder abwärts (gegen den Uhrzeigersinn drehend) zu zählen. In Abbildung 3.18 ist der Programmablauf des Impulszählers dargestellt.

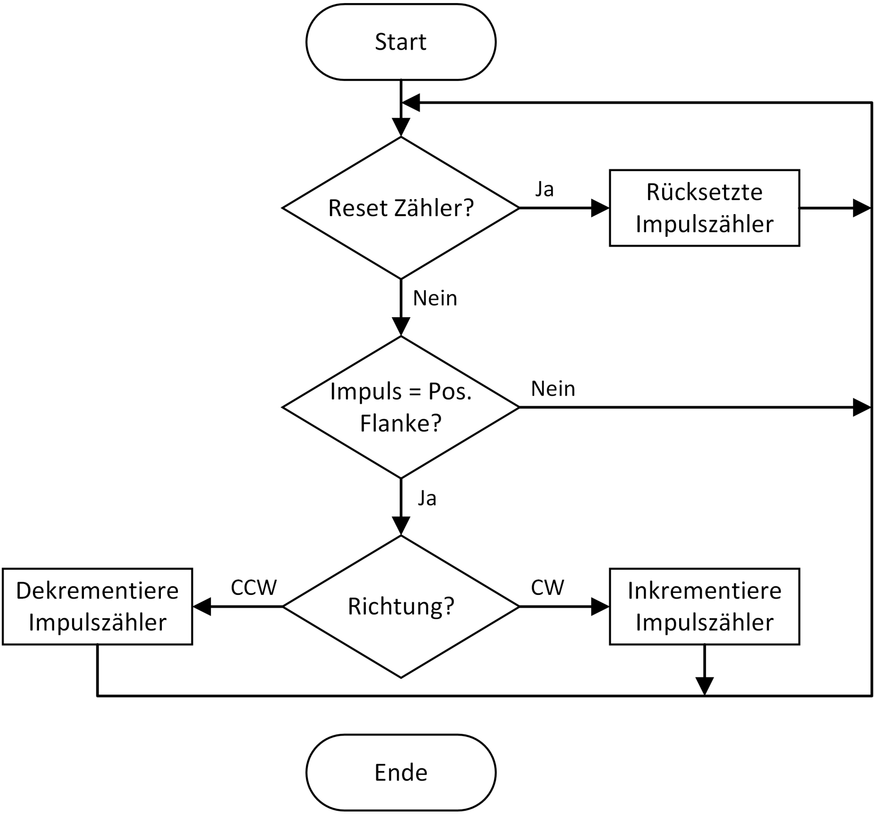


Abbildung 3.18.: *Programmablauf des Impulszählers. Wird eine steigende Flanke detektiert wird je nach Drehrichtung aufwärts oder abwärts gezählt.*

4. Test und Validierung

In diesem Kapitel wird auf die Inbetriebnahme und den Funktionstest der ZMX-Handsteuerung eingegangen. Als erstes wird die Hardware der Kontrolleinheit auf Funktion geprüft. Gefolgt vom Testen einzelner Komponenten in Soft- und Firmware. Im Anschluss folgt der Gesamttest der ZMX-Handsteuerung mit Hilfe eines Schrittmotors und mit angeschlossenem Encoder.

4.1. Erstinbetriebnahme

Für die Erstinbetriebnahme wurde die Leiterplatte in folgender Reihenfolge bestückt:

1. Spannungsregler: 3.3V, 2.5V, 1.2V
2. FPGA und Peripherie
3. Mikrocontroller und Peripherie
4. Display und Peripherie
5. Ein- und Ausgangsstufe
6. Ein- und Ausgabeelemente

Nach jedem Punkt wurde die Leiterplatte an einen einstellbaren Netzteil[1] angeschlossen und mit 5V DC versorgt. Die Strombegrenzung wurde auf 200mA eingestellt, um mögliche Schäden an den Bauteilen durch einen Kurzschluss auf der Leiterplatte zu verhindern.
Im ersten Testlauf wurde die Spannungsversorgung getestet. Nach Einschalten des Netzgeräts, lag der Stromverbrauch bei 1,9mA. Die Ausgangsspannungen der Linearregler wurden mit einem Multimeter[2] gemessen und in die Tabelle 4.1 eingetragen.
Anschließend wurde der FPGA bestückt und mit einer Firmware beschrieben. Mit der Fehlermeldung das der Chip nicht gefunden wurde, konnte das Beschreiben des FPGAs nicht abgeschlossen werden. Eine Überprüfung der Spannungsversorgung an den Versorgungspins ergab keine Abweichung zum Datenblatt. Nach genauer Studie des Datenblatts wurde

[1] HAMEG Porgrammable Power Supply HMP2030
[2] FLUKE 289 True RMS Multimeter

Spannung Sollwert [V]	3.3	2.5	1.2
Spannung Istwert [V]	3.28	2.51	1.19

Tabelle 4.1.: *Soll- und Istwert der Spannungsversorgung auf der Kontrolleinheit*

ersichtlich, dass das Pad unterhalb des Gehäuses angeschlossen werden muss [26, p. 5]. Zuvor wurde fälschlicherweise davon ausgegangen, dass es sich dabei um eine rein thermische und nicht elektrische Anbindung handelt. Um für die Erstbestückung den Aufwand gering zu halten, wurde auf die Anbindung des Pads verzichtet. Nachdem das Pad angeschlossen wurde, ließ sich der FPGA programmieren.
Der bestückte Mikrocontroller wurde direkt vom Programmieradapter erkannt und ließ sich programmieren.
Um das Display zu testen, wurde auf dem Mikrocontroller der Initialisierungsvorgang für die Anzeige durchgeführt [22, p. 49-50] und anschließend ein Teststring gesendet. Dieser Test blieb erfolglos. Die Fehlersuche ergab, dass MISO- und MOSI-Leitung der SPI-Kommunikation vertauscht waren. Nach Beheben des Fehlers konnte das Display erfolgreich initialisiert und beschrieben werden.
Zuletzt wurden die Ein- und Ausgangsstufe, sowie die Ein- und Ausgabeelemente bestückt. Um die Taster und die Drehenencoder zu testen, wurden jeweils die Eingänge am Mikrocontroller abgefragt und bei Betätigung eine LED zum Leuchten gebracht.
Die Funktionalität der differentiell zu single-ended Wandlung der Encodersignale wurde mit Hilfe eines Motors mit Encodern und einem Oszilloskop getestet (vgl. Abb. 4.1). Der Motor wurde von Hand bewegt, dabei war das zu erwartende Rechtecksignal auf dem Oszilloskop zu erkennen.

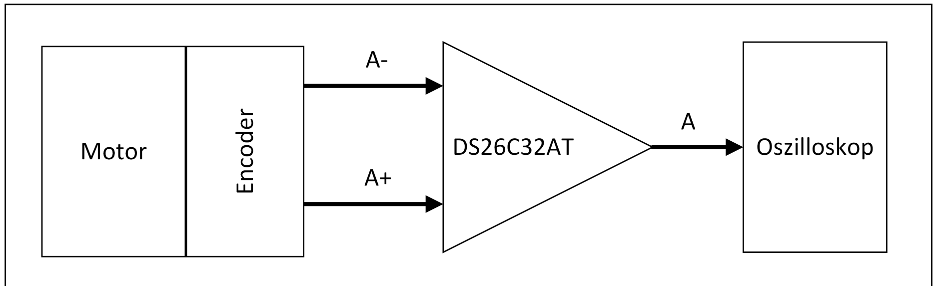


Abbildung 4.1.: *Testaufbau für die Wandlung der differentiellen Encodersignale in single ended. Das single ended Signal wird mit einem Oszilloskop überprüft.*

Löst eine Endlage aus, soll dies vom FPGA erkannt werden. Damit die 15V der Endlagen den FPGA nicht beschädigen, muss die Spannung auf 3.3V gewandelt werden. Um die Baugruppe für die Pegelanpassung zu testen, wurden am Eingang 15V bzw. 0V angelegt und die Spannung gemessen (vgl. Abb. 4.2). Die maximale Ausgangsspannung beträgt

3,3V, somit wird der Eingang vom FPGA nicht beschädigt.

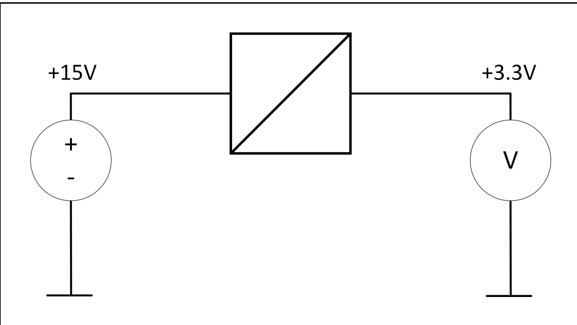


Abbildung 4.2.: *Testaufbau für Auswertung der Endlagen. Es werden 15V auf der Eingangsseite eingespeist und die Spannung auf der Ausgangsseite gemessen.*

Die Erstinbetriebnahme konnte erfolgreich beendet werden. Es kann nun die Programmierung des Mikrocontrollers und des FPGAs erfolgen. Für weitere Tests wird die ZMX-Handsteuerung zusammengebaut, um die gesamte Funktionalität zu überprüfen.

4.2. Validierung einzelner Funktionen

Im Folgenden wird die Überprüfung der Realisierung der Anforderungen dargestellt. Es wird auf grundlegende Funktionen wie Impulserzeugung im FPGA und Kommunikation zur ZMX-Endstufe eingegangen.

4.2.1. Impulserzeugung

Um die Richtigkeit der Impulserzeugung zu überprüfen, wurde mit einem Oszilloskop[3] direkt an dem Steckverbinder X2 Pin 6 der erzeugte Impuls aufgenommen (vgl. Abbildung 4.3). Die Impulsparameter wurden im Mikrocontroller vorgegeben und über SPI zum FPGA übertragen.

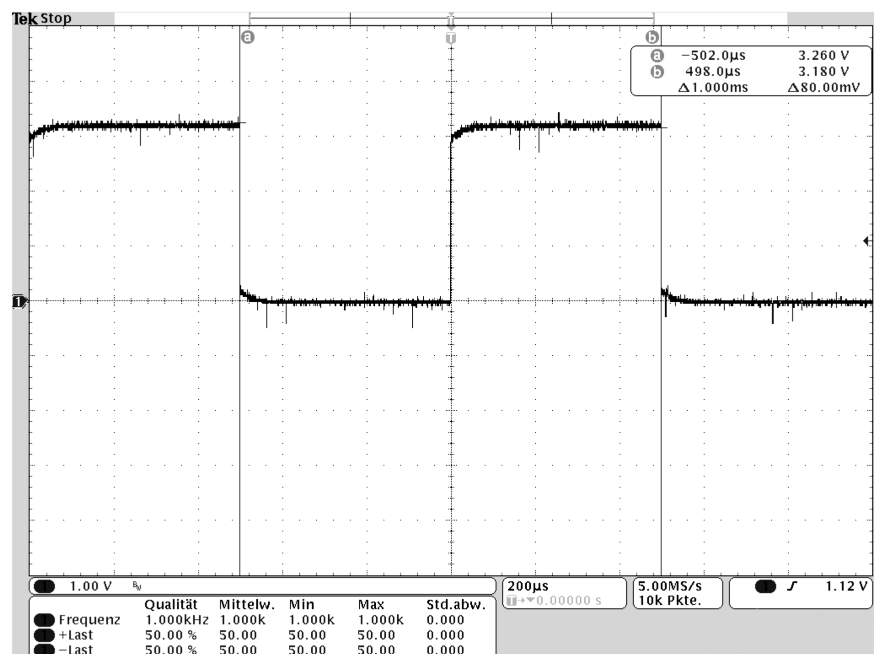


Abbildung 4.3.: *Wiederholungsrate von 1kHz der erzeugten Impulse für die ZMX-Endstufe gemessen an X2 Pin 6.*

[3] Tektronix MSO4104

Die Dauer der Impulse hängt von der Wiederholungsrate ab. Ein Maximum von 100kHz bedeutet, das die Minimale Impulsdauer

$$t_{min} = \frac{1}{f_{max} \cdot 2} = \frac{1}{100kHz \cdot 2} = 5\mu s \tag{4.2.1.1}$$

fünf Mikrosekunden entspricht. Daraus ist ersichtlich, dass die gestellte Anforderung von mindestens einer Mikrosekunde eingehalten wird. Des Weiteren wurde die Impulsrate überprüft. Dafür wurde eine Impulsfrequenz von 1000Hz vorgegeben. Die Abbildung 4.3 zeigt ganz deutlich, dass die Impulse mit genau 1kHz Wiederholungsrate erzeugt werden. Ebenfalls wurde die minimal und maximal einstellbare Frequenz überprüft. Mit 1Hz Minimum und 100kHz Maximum sind die Anforderungen erfüllt. Alle Messungen sind in Tabelle 4.2 dargestellt.

Bezeichnung	**gemessen**	**Sollwert**
Impulsbreite	min. 5µs	>1µs
f_{min}	1Hz	1Hz
f_{1000Hz}	1000Hz	1000Hz
f_{max}	100kHz	100kHz

Tabelle 4.2.: *Gemessene und vorgegebene Werte für die Impulserzeugung.*

4.2.2. Lesen von Encoder-Signalen

Um die Auswertung von Encodern zu testen, wurde ein Schrittmotor mit Encoder an die ZMX-Handsteuerung angeschlossen. Die Impulsfrequenz wurde auf 700Hz eingestellt. Altera bietet den SignalTap II Logic Analyzer, ein Tool das es ermöglicht, die internen Signale eines FPGA zu untersuchen [27, p. 1]. Mit diesem Tool wurde das Verhalten der Encoder-Auswertung analysiert (vgl. Abbildung 4.4). Nach dem Aktivieren des Signals Reset, wurde der Zähler auf 0 gesetzt. Jede Zustandsänderung an den Eingangssignalen A oder B bewirkt eine Änderung am Zählerstand. In diesem Fall dreht sich der Motor im Uhrzeigersinn und somit wird auch der Zählerstand erhöht.

Abbildung 4.4.: *Mit dem SignalTap II Logic Analyser aufgenommenes Verhalten der Encoder-Auswertung im FPGA. Die Auslösung des Signal Reset setzt den Zählerstand auf Null. Mit jeder Zustandsänderung der Eingänge wird der Zähler inkrementiert.*

4.2.3. Kommunikation mit der Endstufe

Um die Kommunikation mit der ZMX-Endstufe zu testen, wurde die Softwareversion der Endstufe ausgelesen. Über den ServiceBus wurde der Befehl „B " gesendet, der zum Auslesen der Softwareversion dient. Die Endstufe antwortete mit dem String „ZMX+ V1.01" [8, p. 17]. Als weiterer Test wurde der Laufstrom auf 1A eingestellt. Die Endstufe antwortete mit dem String „r100" [8, p. 18]. Über den ServiceBus kann die ZMX-Endstufe jetzt den Anforderungen entsprechend parametriert werden.

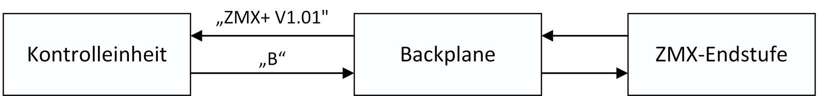


Abbildung 4.5.: *Test zur Kommunikation mit der Endstufe. Der Befehl „B" dient zum Auslesen der Firmware Version. Die Endstufe antwortet mit dem String „ZMX+ V1.01" .*

4.3. Funktionstest der ZMX-Handsteuerung für 2-Phasen-Schrittmotoren

Für den folgenden Funktionstest wurde ein Präzisions-Lineartisch mit angeschlossenen inkrementellen Linearencoder der Firma PI verwendet [28].

Eingabe

Im ersten Schritt wurde die Funktion der Eingabeelemente überprüft. Bei Betätigung des jeweiligen Bedienelements wird der Zustand der dazugehörigen Funktion getestet. Das Ergebnis ist in Tabelle 4.3 dargestellt.

Bedienelement	Funktion	Zustand
Menu	Aktiverung des Menü zur Parametrierung	OK
Entregen	Aktiverung der Endstufe	OK
Inkr. Menu	Navigation im Menü[a], Single-Step[b]	OK
	Single-Step[b]	OK
Taster Inkr. Menu	Ansichtswechsel Encoder/Steps	OK
Pos. Reset	Reset Encoder/ Steps[c]	OK
Inkr. Frequenz	Einstellen der Impulsfrequenz[b]	OK
CW	Motor dreht sich in Richtung CW[b]	OK
CCW	Motor dreht sich in Richtung CCW[b]	OK
Stop	Stoppt den Motor[b]	OK

a) Das Menü muss aktiviert sein.
b) Die Endstufe muss aktiv sein.
c) Die jeweils durch den Taster Inkr. Menu ausgewählte Ansicht wird zurückgesetzt.

Tabelle 4.3.: *Test der Eingabeelemente und dazugehöriger Funktion.*

Ausgabe

Der Test der Ausgabeelemente wird analog zum Test der Eingabeelemente durchgeführt. Die Ergebnisse sind in Tabelle 4.4 zusammengetragen. Um die Endlagen zu überprüfen, wurden sie entsprechend der Anforderungen, d.h. „active high“ , „active low“ sowie deaktiviert eingestellt und durch Betätigung überprüft.

Anzeige	Funktion	Zustand
LED CW	Zeigt den Zustand der Endlage CW an	OK
LED CCW	Zeigt den Zustand der Endlage CCW an	OK
LED ZMX	Zeigt den Zustand der ZMX-Endstufe an: rot aktiv, grün nicht aktiv	OK
Display	Darstellung diverser Parameter (vgl. Tabelle 4.5)	OK
Display	Anzeige Encoder[a] Anzeige Schritte[a]	OK OK

a) Die jeweils durch den Taster Inkr. Menu ausgewählte Ansicht wird angezeigt.

Tabelle 4.4.: *Test der Ausgabeelemente*

Parametrierung

Im Zuge des Test der Ein- und Ausgabeelemente wurde ebenfalls die Parametrierung der ZMX-Handsteuerung überprüft. Wie in den Anforderungen gefordert, konnten alle Parameter eingestellt werden. Das Resultat ist in Tabelle 4.5 dargestellt.

Parameter	Einstellung	Zustand
Laufstrom	0A bis 5A in 0,1A Schritte	OK
Stoppstrom	0A bis max. Laufstrom in 0,1A Schritte Auto-Modus: Stoppstrom= 0,5·Laufstrom	OK OK
Schrittauflösung	1/1 bis 1/512 in 14 Stufen	OK
Endlagenauswertung CW	Active high, active low, deaktiviert	OK
Endlagenauswertung CCW	Active high, active low, deaktiviert	OK
Impulsfrequenz	1Hz bis 100kHZ	OK

Tabelle 4.5.: *Test der Parametrierung der ZMX-Handsteuerung.*

Gesamttest anhand eines Testszenarios

Zum Abschluss wurde die Funktion ZMX-Handsteuerung in einem Testszenario validiert. Hierfür wurde der in Abbildung 4.6 gezeigte Testaufbau verwendet. Der Schrittmotor wurde direkt über eine Leitung mit der ZMX-Handsteuerung verbunden. Das erzeugte Signal vom linearen Inkrementalencoder wurde über eine Interpolationseinheit und „Encoder Power Supply“ mit der ZMX-Handsteuerung verbunden. Die zusätzliche Spannungsversorgung diente zur Versorgung der Interpolationseinheit und des Encoders. Mit dem Testaufbau

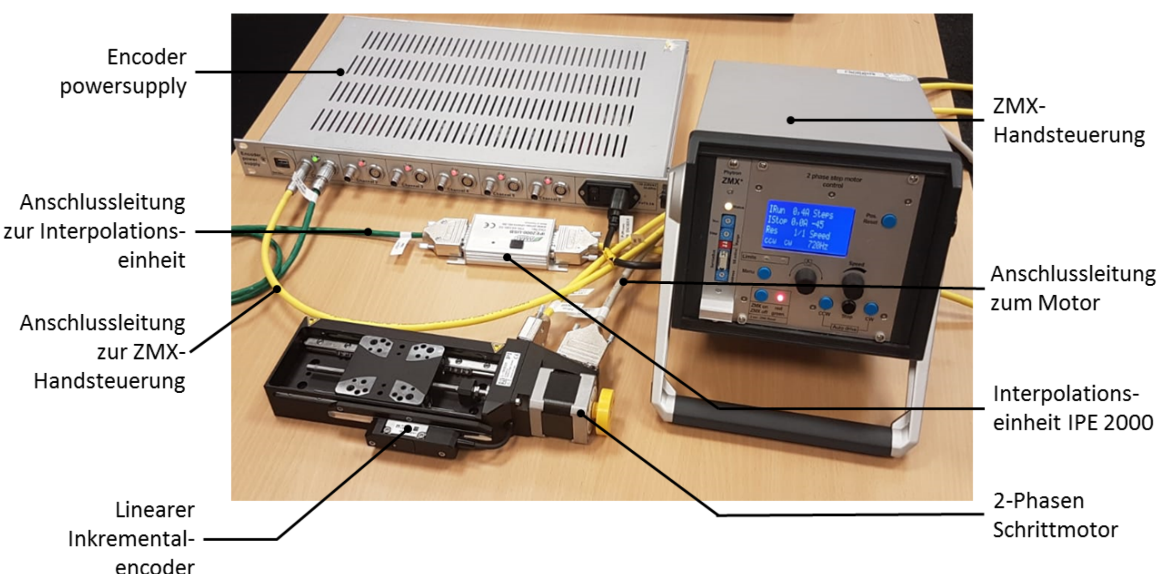


Abbildung 4.6.: *Testaufbau für die ZMX-Handsteuerung.*

wurden die in Tabelle 4.6 aufgelisteten Schritte durchgeführt, um die Funktionalität der ZMX-Handsteuerung zu belegen. Hierbei wurde ein besonderes Augenmerk auf die korrekte Bedienung und Anzeige im Display gelegt, sowie auf mögliches Verhalten durch den Nutzer (Schritt 11).

Des Weiteren wurde überprüft, ob die Anzahl der erzeugten Impulse und gezählten Encodern zu den Motordaten passen, um somit deren korrekte Funktion zu validieren. Laut Datenblatt benötigt der angeschlossene Schrittmotor 200 Vollschritte für eine ganze Umdrehung [28, p. 4]. Dabei wurden 100.015 Encoderschritte gezählt. Bei einer Auflösung von 10nm pro Encoderschritt ergibt sich daraus laut Gleichung 4.3.0.1 1mm Vorschub.

$$s = 100.015 \cdot 10nm \approx 1mm \tag{4.3.0.1}$$

Dies entspricht der Angabe aus dem Datenblatt [28, p. 3]. Die minimale Abweichung von 150nm kann durch mechanische Ungenauigkeiten in der Spindel oder durch Temperaturänderungen der Umgebung entstehen. Es kann angenommen werden, dass die Impulserzeugung und das Auswerten von Encodern die korrekte Funktion aufweisen.

Schritt	Beschreibung	Verhalten
1	Laufstrom auf 1.5A einstellen	Display: IRun 1.5A
2	Stoppstrom auf 0.7A einstellen	Display: IStop 0.7A
3	Schrittauflösung auf 1/1 einstellen	Display: Res 1/1
4	Endlagen auf active high	Display: Einstellung Endlage, LEDs Endlage aus
5	Endstufe aktivieren	Display: ZMX Enabled, LED rot
6	Frequenz auf 400Hz einstellen	Display: Speed 400Hz[a]
7	Fahre Motor kontinuierlich Richtung CW	Motor dreht sich CW mit 400Hz[b]
8	Erreichen der Endlage CW	Motor stoppt, keine Fahrt in Richtung CW mehr möglich, Display und LED zeigen die Aktivierung der Endlage an
9	Fahre Motor kontinuierlich Richtung CCW	Motor dreht sich CCW mit 400Hz[b]
10	Erreichen der Endlage CCW	Motor stoppt, keine Fahrt in Richtung CCW mehr möglich, Display und LED zeigen das Auslösen der Endlage an
11	Deaktivere Endlage CCW	Motor bleibt weiterhin im Stopp
12	Reset gezählte Schritte und Encoder	Display: Steps/Enc. 0
13	Autodrive auf 200 Schritte einstellen	Display: Autodrive ±200
14	Fahre 200 Schritte in Richtung CW	Display: Steps 200 / Enc. 100015
15	Betätige CW und CCW gleichzeitig	Motor fährt 200 Schritte in Richtung CCW, Display: Steps 0
16	Betätige CW und CCW gleichzeitig	Motor fährt 200 Schritte in Richtung CW, Display: Steps 200
17	Single Step Faktor auf 10 Fach einstellen	Display: Factor 10x
18	Drehe Menü um eine Rastung in Richtung CW	Motor fährt 10 Schritte in Richtung CW, Display: Steps 210
19	Drehe Menü um zwei Rastung in Richtung CCW	Motor fährt 20 Schritte in Richtung CCW, Display: Steps 190
20	Endstufe deaktivieren	Display: ZMX Disabled, LED grün

a) Impulsfrequenz für die ZMX-Endstufe

b) Der Motor benötigt 200 Vollschritte für eine Umdrehung [28, p. 4]. Daraus ergibt sich bei eingestellter Impulswiederholungsrate eine Frequenz von 2Hz.

Tabelle 4.6.: *Testszenario zur Validierung der Funktion der ZMX-Handsteuerung*

5. Auswertung und Bewertung

In diesem Kapitel wird die Bachelorthesis reflektiert und ein Ausblick für mögliche Erweiterungen gegeben.

5.1. Auswertung

Das Konzept für die ZMX-Handsteuerung konnte erfolgreich realisiert werden. In dem anschließenden Test bestand die ZMX-Handsteuerung allen Anforderungen.

Das erstellte Hardware Konzept, ermöglichte eine schnelle und zielführende Entwicklung der Kontrolleinheit für die ZMX-Handsteuerung. Das Konzept für den FPGA gestattete es die Firmware zügig zu entwickeln und zu implementieren. Der modulare Aufbau der Firmware erlaubte es, die einzelnen Bausteine wie die SPI-Slave-Schnittstelle oder die Impulserzeugung separat und dann im Gesamten erfolgreich zu testen. Das Konzept der Software für den Mikrocontroller erwies sich als gut. Die Verteilung der Kommunikation zwischen Mikrocontroller und FPGA, sowie Mikrocontroller und Display auf unterschiedliche Interrupts erlaubte das Priorisieren einzelner Schnittstellen. Der Einsatz eines Buffers für die Datenübertragung zu dem Display erwies sich als sinnvoll. So ist es möglich während der Ausführungszeit der Schreiboperationen im Display, andere Programmteile im Mikrocontroller auszuführen, bevor ein weiteres Zeichen gesendet wird. Das Konzept der Software konnte in Tests überzeugen.
Während der Inbetriebnahme tauchten Fehler in der Hardware auf, wie z.B. das Vertauschen der Datenleitungen in der Schnittstelle zur Kommunikation mit dem Display über SPI. Diese Fehler konnten schnell behoben werden. Die Aufgabenaufteilung auf FPGA und Mikrocontroller erwies sich als vorteilhaft. Der FPGA übernimmt die Impulserzeugung, Überwachung der Endlagen und das Auswerten der Inkrementalencoder. Während der Mikrocontroller für Visualisierung und Kommunikation zuständig ist. So wird sicher gestellt, dass das Auslösen der Endlagen immer rechtzeitig detektiert wird. Durch das Auswerten von Inkrementalencodern bietet die ZMX-Handsteuerung eine deutlich genauere Möglichkeit an, einen Motor zu positionieren. Grenzen werden hierbei nur noch bedingt durch die Mechanik des Motors oder die Auflösung der Encoderschritte gesetzt.

5.2. Ausblick

Im Folgenden wird ein Ausblick auf Erweiterungen gegeben, die in der ZMX-Handsteuerung implementiert werden können. Durch die Aufteilung der Aufgaben auf Mikrocontroller und FPGA stehen nun zwei Bausteine für Erweiterungen zur Verfügung. Sowohl FPGA als auch Mikrocontroller bieten unterschiedliche Möglichkeiten für die Realisierung neuer Anforderungen.

5.2.1. EtherCAT Anbindung

Als eine sehr interessante Möglichkeit für die Zukunft bietet sich die Anbindung der ZMX-Handsteuerung in ein EtherCAT Netz an. Die Abbildung 3.2 und der Schaltplan Seite 6 im Anhang A zeigen, dass die Grundvoraussetzung dafür bereits geschaffen wurden. IC11 ist ein EtherCAT Slave Controller, der über eine Schnittstelle zur Kommunikation über SPI von einem Mikrocontroller angesteuert werden kann. Über EtherCAT ist eine echtzeitfähige Anbindung der ZMX-Handsteuerung an ein Netzwerk möglich.

5.2.2. Positionsregelung

Angeschlossene Encoder bieten den Vorteil, dass die Position relativ zu einem Punkt bestimmt werden kann. Diese Eigenschaft kann genutzt werden, um z.B. die Position eines Motors über ein PID-Regler zu Regeln. Eine Möglichkeit wäre Proportional-, Integral und Differentialanteil über das Menü einstellen zu können. Bei Vorgabe des Encoderwerts, fährt der Motor die Position an und hält diese.

Abkürzungsverzeichnis

DESY Deutsches Elektronen-Synchrotron

PETRA Positron-Elektron-Tandem-Ring-Anlage

FLASH Freie-Elektronen-Laser in Hamburg

XFEL X-Ray Free-Electron Laser

FS-EC Forschung Synchrotron - Experiment Control

ARM Advanced RISC Machines

EMV Elektromagnetische Verträglichkeit

FPGA Field Programmable Gate Array

CW Clockwise

CCW Counter Clockwise

SPI Serial Peripheral Interface

I^2C Inter-Integrated Circuit

UART Universal Asynchronous Receiver Transmitter

PLL Phase Locked Loop

DDS Direct Data Synthesis

MCU Microcontroller Unit

STM STMicroelectronics

GPIO General Purpose Input/Output

BGA Ball Grid Array

QFP Quad Flat Package

IC Integrated Circuit

I/O Input/Output

EEPROM Electrically Erasable Programmable Read-Only Memory

R/W Read/Write

RS Register Select

NSS Active Low Slave Select

SCK Shift Clock

MOSI Master Out Slave In

MISO Master In Slave Out

Rx Receiver

Tx Transmitter

MSB Most Significant Bit

FIFO First In First Out

Tabellenverzeichnis

1.1. Stromrichtung in den Motorphasen im Vollschrittbetrieb. 10
1.2. Stromrichtung in den Motorphasen im Halbschrittbetrieb. 11

2.1. Register im FPGA und deren Zugriffsmodus 24

3.1. Ausgewählte Bauteile aus der Handsteuerung Serie 2 und deren Status, die in der Serie 3 wiederverwendet werden sollen bzw. mögliche Ersatztypen . . 25
3.2. Schnittstellen zur Kommunikation zwischen den einzelnen Baugruppen . . . 26
3.3. ARM Cortex-M4 Mikrocontroller von STM im Vergleich in ausgewählten Kategorien 27
3.4. Pin-Nutzung des Mikrocontrollers 28
3.5. Ressourcennutzung des Mikrocontrollers STM32F407 28
3.6. Vergleich der Cyclone IV E in der Bauform QFP 29
3.7. Leitungen vom Drehinkrementalgeber zum Mikrocontroller 31
3.8. Leitungen der SPI-Schnittstelle vom Mikrocontroller zum Display. 32
3.9. Leitungen der SPI-Schnittstelle zwischen FPGA und Mikrocontroller aus Sicht des Masters. 33
3.10. Leitungen der UART-Schnittstelle vom Mikrocontroller zur ZMX-Endstufe. . . 33
3.11. Erläuterung des Protokolls zur Kommunikation mit der ZMX-Endstufe 34
3.12. Konfiguration der Interrupts im Mikrocontroller. 37
3.13. Datenframe zur Kommunikation mit dem FPGA über SPI. 42
3.14. Aufbau des ersten Bytes, das die R/W Anweisung und die Adresse enthält. . 42
3.15. Register im FPGA mit Adresse und deren Zugriffsmodus 42
3.16. Bedeutung der einzelnen Bits in dem Kontrollregister mit den einstellbaren Funktionen. 49
3.17. Bedeutung der einzelnen Bits in dem Schritt- und Richtungsregister mit den einstellbaren Funktionen. 49

4.1. Soll- und Istwert der Spannungsversorgung auf der Kontrolleinheit 54
4.2. Gemessene und vorgegebene Werte für die Impulserzeugung. 57
4.3. Test der Eingabeelemente und dazugehöriger Funktion. 59
4.4. Test der Ausgabeelemente 60
4.5. Test der Parametrierung der ZMX-Handsteuerung. 60

4.6. Testszenario zur Validierung der Funktion der ZMX-Handsteuerung 62

Abbildungsverzeichnis

1.1. Schrittmotor im Vollschrittbetrieb 10
1.2. Schrittmotor im Halbschrittbetrieb 11
1.3. Aufbauten mit Schrittmotoren am DESY 12

2.1. Phytron ZMX$^+$ Endstufe 15
2.2. Backplane 16
2.3. Gehäuse der ZMX-Handsteuerung 16
2.4. Beispiel Encodersignal 17
2.5. Konzept der Handsteuerung 20
2.6. Konzept der Hardware der Kontrolleinheit 21
2.7. Konzept der Software im Mikrocontroller 22
2.8. Konzept der Firmware im FPGA 23

3.1. Leiterplatte Top-Anischt 30
3.2. Leiterplatte Bottom-Ansicht 30
3.3. Timing Diagramm SPI Mikrocontroller-Display 32
3.4. Programmablauf der Initialisierung im Mikrocontroller 35
3.5. Programmablauf innerhalb der Hauptschleife im Mikrocontroller 36
3.6. Programmablauf für den Interrupt zum Senden von Daten an das Display. . . 38
3.7. Display-Buffer 39
3.8. Programmablauf für den Interrupt zum Senden von Daten zum FPGA 40
3.9. Auswertung von Quadratur-Encodern im FPGA 41
3.10. Timing-Diagramm SPI Mikrocontroller-FPGA 42
3.11. SPI-Slave im FPGA 43
3.12. SPI-Slave State-Machine im FPGA 44
3.13. Datenverwaltung im FPGA 45
3.14. Frequenzerzeugung im FPGA 46
3.15. Impulserzeugung im FPGA 48
3.16. Kontinuierliche Impulserzeugung im FPGA 50
3.17. Erzeugung n-Impulsen im FPGA 51
3.18. Impulszähler im FPGA 52

4.1. Testaufbau diffentielle zu single ended Wandlung 54

4.2. Testaufbau für Auswertung der Endlagen 55
4.3. Impulserzeugung mit 1kHz 56
4.4. Verhalten des Encoderzählers 57
4.5. Test zur Kommunikation mit der Endstufe 58
4.6. Testaufbau für die ZMX-Handsteuerung 61

Literaturverzeichnis

[1] DESY, “Wir machen Erkenntnis möglich,” Webseite:, 2018, [Zugriff am 02.08.2018]. [Online]. Available: http://www.desy.de/ueber_desy/desy/index_ger.html

[2] ——, “50 Jahre DESY,” Firmenschrift: 50 Jahre DESY, 2009.

[3] ——, “Forschung für die Zukunft,” Webseite:, 2018, [Zugriff am 02.08.2018]. [Online]. Available: http://www.desy.de/ueber_desy/desy/forschung_fuer_die_zukunft/index_ger.html

[4] ——, “FLASH,” Webseite:, 2018, [Zugriff am 02.08.2018]. [Online]. Available: http://www.desy.de/forschung/anlagen__projekte/flash/index_ger.html

[5] ——, “European XFEL,” Webseite:, 2018, [Zugriff am 02.08.2018]. [Online]. Available: http://www.desy.de/forschung/anlagen__projekte/european_xfel/index_ger.html

[6] ——, “PETRA III,” Webseite:, 2018, [Zugriff am 02.08.2018]. [Online]. Available: http://www.desy.de/forschung/anlagen__projekte/petra_iii/index_ger.html

[7] ——, “Experiment Control Taks,” Webseite:, 2018, [Zugriff am 02.08.2018]. [Online]. Available: http://photon-science.desy.de/research/technical_groups/experiment_control/index_eng.html

[8] Phytron, “Befehlssatz für Schrittmotor-Endstufen mit ServiceBus,” Manual: MA 1246-A004 D, 2008, [Zugriff am 03.07.2018]. [Online]. Available: https://www.phytron.de/fileadmin/user_upload/produkte/kommunikation_programmierung/pdf/ma-servicebus-befehlssatz-de.pdf

[9] ——, “ZMX+ Schrittmotorenendstufe mit ServiceBus, Original Betriebsanleitung,” Manual: MA 2109-A015 D, 2018, [Zugriff am 03.07.2018]. [Online]. Available: https://www.phytron.de/fileadmin/user_upload/produkte/endstufen_controller/pdf/ma-zmxplus-de.pdf

[10] N2Power, “Ultra small, high efficiency power supllies, XL275 AC-DC Series,” Datenblatt: Rev: 07-14-17, 2017, [Zugriff am 03.07.2018]. [Online]. Available: https://www.qualstar.com/n2p-datasheets/XL275_AC-DC_datasheet.pdf

[11] nvent Schroff, "CONNECT AND PROTECT, PropacPRO," Poduktkatalog: SCHROFF PRODUKTKATALOG 05/2018 ELECTRONICS PROTEC, 2018, [Zugriff am 20.07.2018]. [Online]. Available: https://schroff.nvent.com/wcsstore/ExtendedSitesCatalogAssetStore/Attachment/SchroffAttachments/Documents/5_2_propacPRO_d.pdf

[12] Elmar Schrüfer, Leonhard Reindl und Bernhard Zagar, *Elektrische Messtechnik - Messung elektrischer und nichtelektrischer Größen*. Carl Hanser Verlag München, 2012, ISBN: 978-3-446-43079-2.

[13] Europa Lehrmittel, *Industrieelektronik und Informationstechnik*. Verlag Europa Lehrmittel, 2009, ISBN: 978-3-8085-3250-8.

[14] DESY, "2-Phasen Schrittmotor Handsteuerung," Manual, 2012.

[15] Electronic Assembly, "LCD-Modul 4x20 - 6,45mm," Manual:LCD-Modul 4x20 - 6,45mm, Inkl. Kontroller SSD1803, 2013, [Zugriff am 12.07.2018]. [Online]. Available: https://www.lcd-module.de/fileadmin/pdf/doma/dip203-6.pdf

[16] STMicroelectronics, "STM32 32-bit MCU family Leading supplier of Arm® Cortex®-M microcontrollers," Firmenschrift: BRSTM320218, 2018, [Zugriff am 29.06.2018]. [Online]. Available: https://www.st.com/content/ccc/resource/sales_and_marketing/promotional_material/brochure/f0/93/da/5c/6b/31/4a/96/brstm32.pdf/files/brstm32.pdf/jcr:content/translations/en.brstm32.pdf

[17] ——, "STM32F405xx and STM32F407xx Datasheet - production data," Datenblatt: DocID022152 Rev 8, 2016, [Zugriff am 14.08.2018]. [Online]. Available: https://www.st.com/resource/en/datasheet/dm00037051.pdf

[18] Intel, "Cyclone® IV FPGAs Product Table," Firmenschrift: Gen-1036-1.2, 2017, [Zugriff am 10.07.2018]. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/pt/cyclone-iv-product-table.pdf

[19] STMicroelectronics, "RM0090 Reference manual, STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced ARM®-based 32-bit MCUs," Manual: DocID018909 Rev 15, 2017, [Zugriff am 20.06.2018]. [Online]. Available: https://www.st.com/content/ccc/resource/technical/document/reference_manual/3d/6d/5a/66/b4/99/40/d4/DM00031020.pdf/files/DM00031020.pdf/jcr:content/translations/en.DM00031020.pdf

[20] ——, "AN4776, General-purpose timer cookbook," Application Note: DocID028459 Rev 2, 2011, [Zugriff am 09.07.2018]. [Online]. Available: https://www.st.com/content/ccc/resource/technical/document/application_note/group0/91/01/84/3f/7c/67/41/3f/DM00236305/files/DM00236305.pdf/jcr:content/translations/en.DM00236305.pdf

[21] ——, “AN3969, EEPROM emulation in STM32F40x/STM32F41x microcontrollers,” Application Note: Doc ID 022108 Rev 1, 2011, [Zugriff am 25.07.2018]. [Online]. Available: https://www.st.com/content/ccc/resource/technical/document/application_note/ec/dd/8e/a8/39/49/4f/e5/DM00036065.pdf/files/DM00036065.pdf/jcr:content/translations/en.DM00036065.pdf

[22] SOLOMONS YSTECH, “SOLOMON SYSTECH SEMICONDUCTOR TECHNICAL DATA, SSD1803, Advance Information,” Datenblatt: SSD1803 Rev 2.0 Feb 2010, 2010, [Zugriff am 12.07.2018]. [Online]. Available: https://www.lcd-module.de/fileadmin/eng/pdf/zubehoer/ssd1803_2_0.pdf

[23] STMicroelectronics, “AN4013, STM32 cross-series timer overview,” Application Note: DocID022500 Rev 6, 2016, [Zugriff am 09.08.2018]. [Online]. Available: https://www.st.com/content/ccc/resource/technical/document/application_note/54/0f/67/eb/47/34/45/40/DM00042534.pdf/files/DM00042534.pdf/jcr:content/translations/en.DM00042534.pdf

[24] L. Miller, “Drehgeber, Encoder, Quadraturdecoder,” Website, 2009, [Zugriff am 21.06.2018]. [Online]. Available: http://www.lothar-miller.de/s9y/categories/46-Encoder

[25] Analog Devices, “A Technical Tutorial on Digital Signal Synthesis,” Firmenschrift: A Technical Tutorial on Digital Signal Synthesis, 1999, [Zugriff am 03.07.2018]. [Online]. Available: http://www.analog.com/media/cn/training-seminars/tutorials/450968421DDS_Tutorial_rev12-2-99.pdf

[26] ALTERA, “Cyclone IV FPGA Device Family Overview,” Manual: Cyclone IV Device Handbook, Volume 1 March 2016, 2016, [Zugriff am 10.07.2018]. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/cyclone-iv/cyiv-51001.pdf

[27] ——, “Design Debugging Using the SignalTap II Logic Analyzer,” Manual: Quartus II Handbook Version 13, Volume 3: Verification, 2013, [Zugriff am 30.07.2018]. [Online]. Available: https://www.altera.com/en_US/pdfs/literature/hb/qts/qts_qii53009.pdf

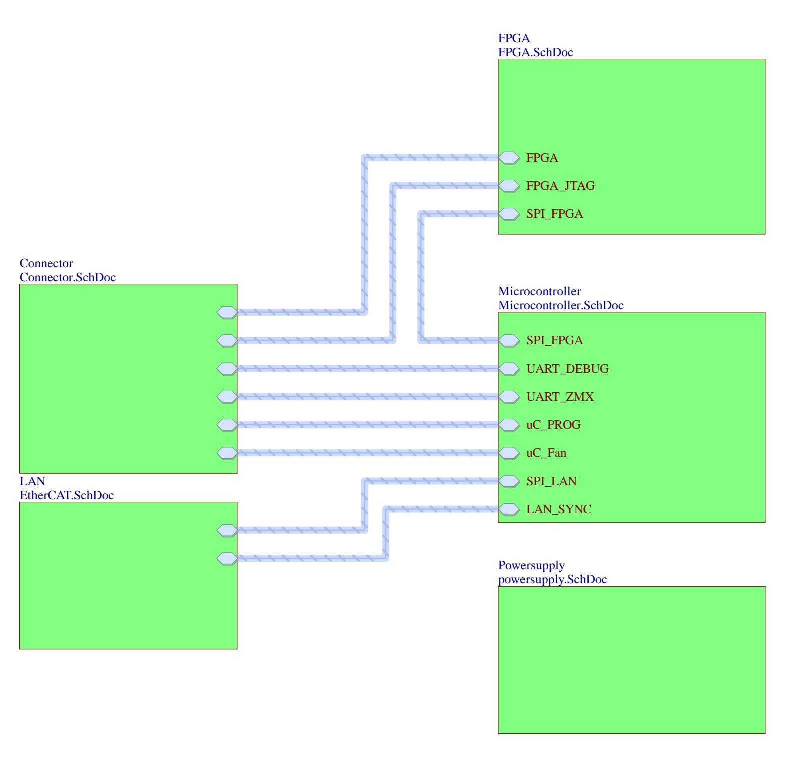
[28] PI, “Präzisions-Lineartisch, Kompakte Bauform, für Lasten bis 10 kg,” Manual:Präzisions-Lineartisch L-509, 2018, [Zugriff am 02.08.2018]. [Online]. Available: https://www.physikinstrumente.de/fileadmin/user_upload/physik_instrumente/files/datasheets/L-509-Datenblatt.pdf

Anhang

Im Anhang befinden sich die Schaltpläne und Zeichnungen (Anhang A) der ZMX-Handsteuerung. Des weiteren ist der Quellcode der Firmware im FPGA (Anhang B) und der Quellcode der Software im Mikrocontroller (Anhang C) enthalten. Zusätzlich befinden sich unter Anhang A auf dem beigefügten Datenträger die Gerber-Dateien der Leiterplatte. Außerdem sind unter Anhang D auf dem Datenträger die Abbildungen, sowie unter Anhang E die digitalen Quellen verfügbar.

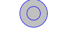
A. Schaltpläne und Zeichnungen

Der Anhang A enthält den Schaltplan der Kontrolleinheit und der Backplane der ZMX-Handsteuerung Serie 3. Des weiteren ist die Zeichnung der Frontplatte, sowie der Schaltplan der Vorgängerversion enthalten.



REFB1 REFT1

REFB2 REFT2

REFB3 REFT3 

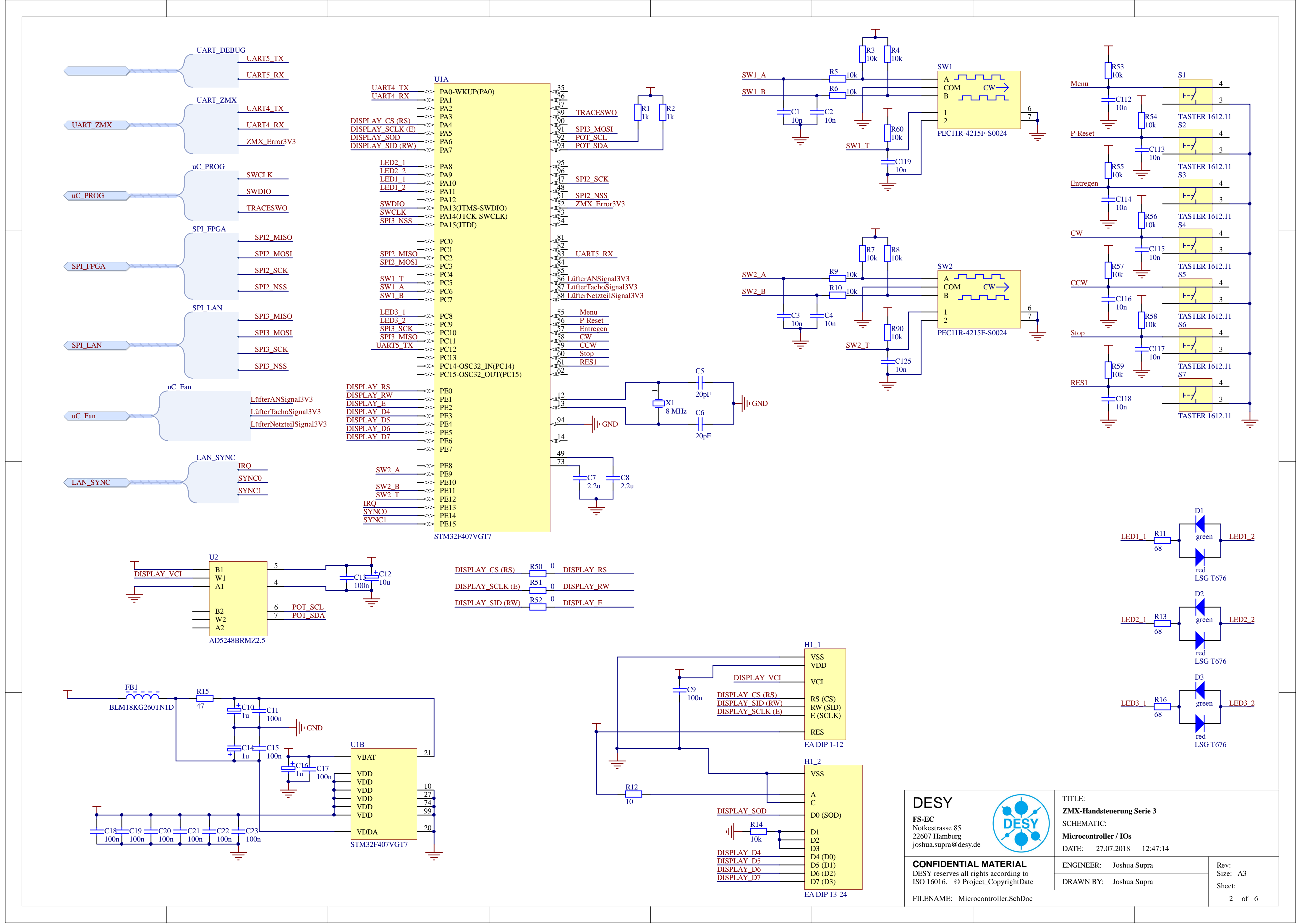
DESY
FS-EC
Notkestrasse 85
22607 Hamburg
joshua.supra@desy.de

TITLE:
ZMX-Handsteuerung Serie 3
SCHEMATIC:
Overview
DATE: 27.07.2018 12:47:14

CONFIDENTIAL MATERIAL
DESY reserves all rights according to ISO 16016. © Project_CopyrightDate

ENGINEER: Joshua Supra
DRAWN BY: Joshua Supra
FILENAME: Control.SchDoc

Rev:
Size: A3
Sheet: 1 of 6

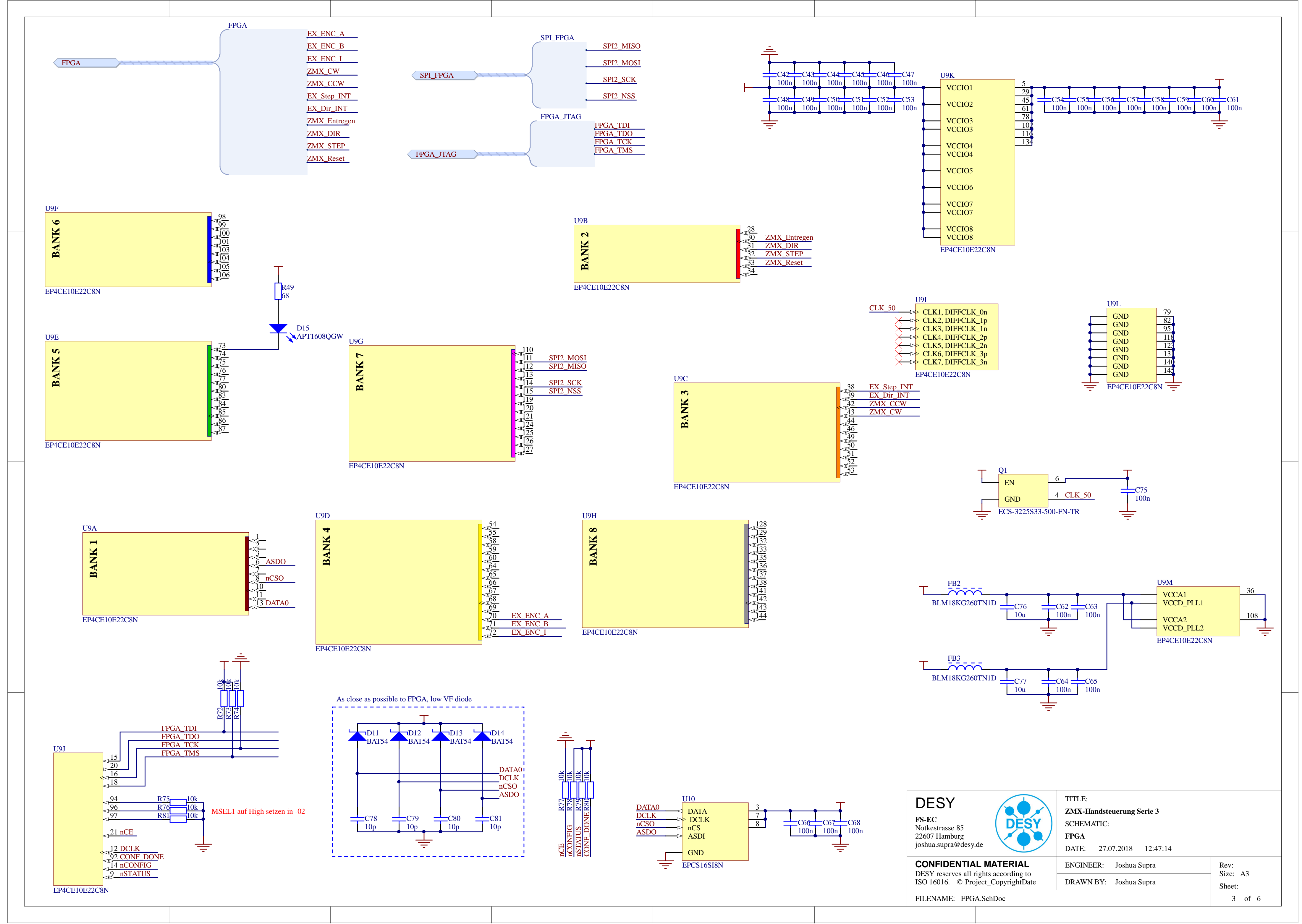


DESY
FS-EC
Notkestrasse 85
22607 Hamburg
joshua.supra@desy.de

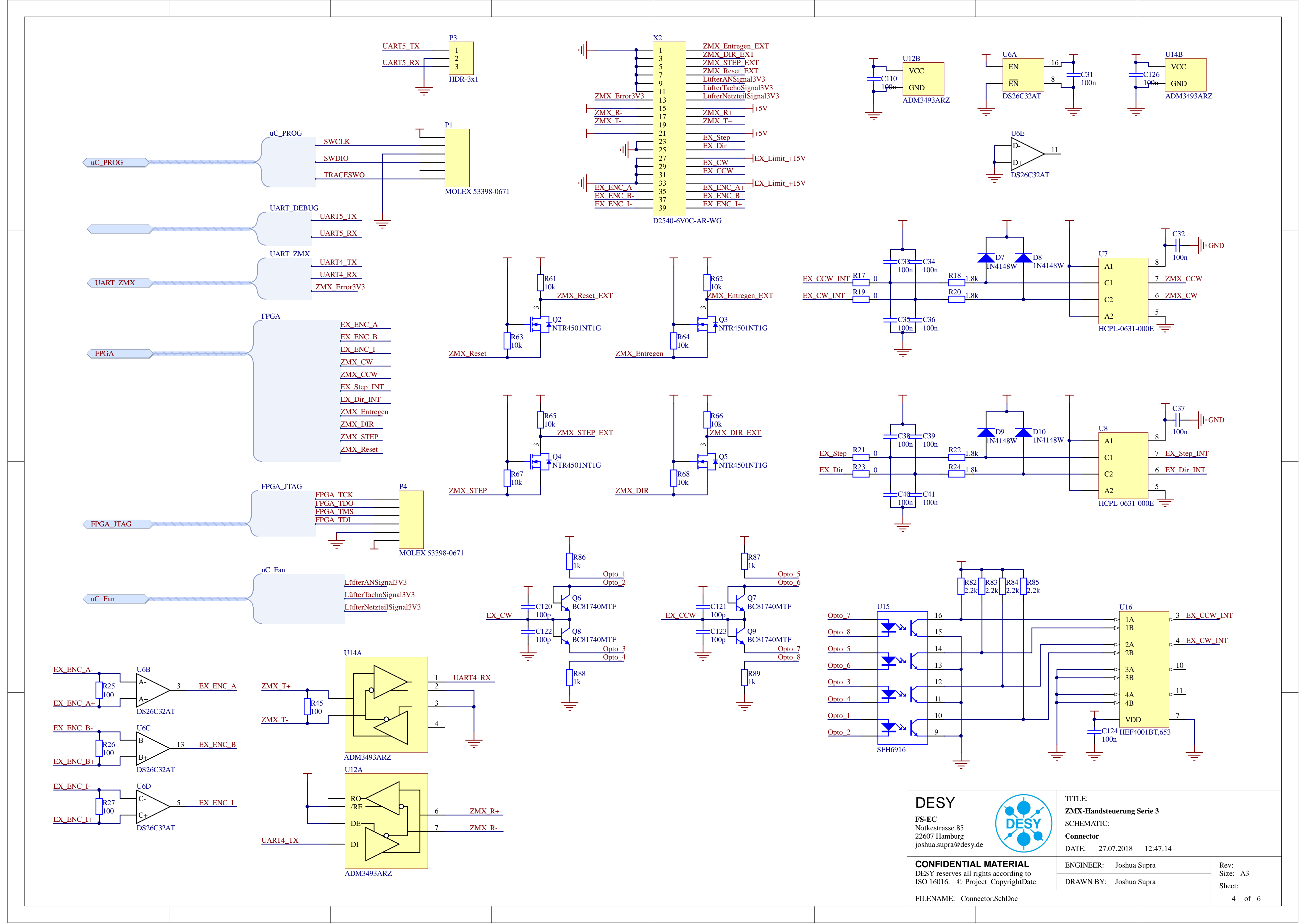
TITLE: ZMX-Handsteuerung Serie 3
SCHEMATIC: Microcontroller / IOs
DATE: 27.07.2018 12:47:14
ENGINEER: Joshua Supra
DRAWN BY: Joshua Supra
Rev:
Size: A3
Sheet: 2 of 6

CONFIDENTIAL MATERIAL
DESY reserves all rights according to ISO 16016. © Project_CopyrightDate

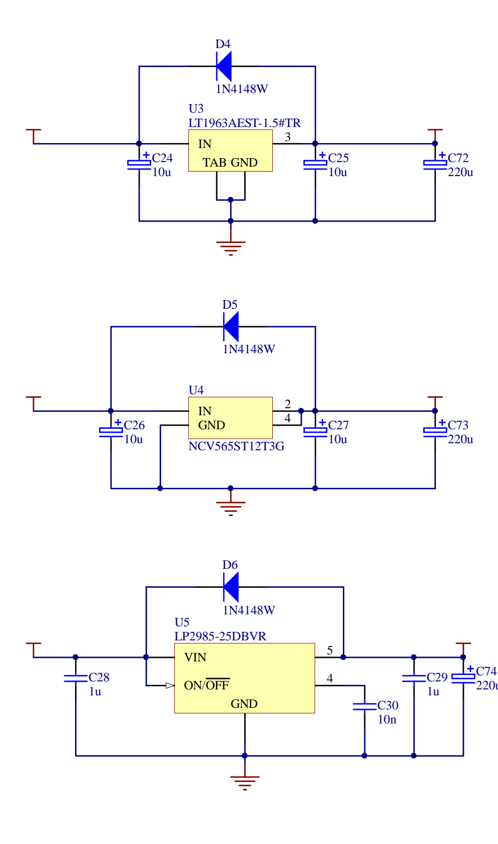
FILENAME: Microcontroller.SchDoc



FPGA
EX_ENC_A
EX_ENC_B
EX_ENC_I
ZMX_CW
ZMX_CCW
EX_Step_INT
EX_Dir_INT
ZMX_Entregen
ZMX_DIR
ZMX_STEP
ZMX_Reset
SPI_FPGA
SPI2_MISO
SPI2_MOSI
SPI2_SCK
SPI2_NSS
FPGA_JTAG
FPGA_TDI
FPGA_TDO
FPGA_TCK
FPGA_TMS
U9K
VCCIO1
VCCIO2
VCCIO3
VCCIO4
VCCIO5
VCCIO6
VCCIO7
VCCIO8
EP4CE10E22C8N
U9F
BANK 6
U9E
BANK 5
R49
68
D15
APT1608QGW
U9G
BANK 7
U9B
BANK 2
U9I
CLK_50
CLK1, DIFFCLK_0n
CLK2, DIFFCLK_1p
CLK3, DIFFCLK_1n
CLK4, DIFFCLK_2p
CLK5, DIFFCLK_2n
CLK6, DIFFCLK_3p
CLK7, DIFFCLK_3n
U9L
GND
U9C
BANK 3
Q1
EN
GND
ECS-3225S33-500-FN-TR
C75
100n
U9A
BANK 1
ASDO
nCSO
DATA0
U9D
BANK 4
U9H
BANK 8
FB2
BLM18KG260TN1D
FB3
U9M
VCCA1
VCCD_PLL1
VCCA2
VCCD_PLL2
U9J
As close as possible to FPGA, low VF diode
D11
D12
D13
D14
BAT54
DCLK
C78
C79
C80
C81
10p
MSEL1 auf High setzen in -02
nCE
CONF_DONE
nCONFIG
nSTATUS
R72
R73
R74
R75
R76
R81
R77
R78
R79
R80
10k
U10
DATA
nCS
ASDI
EPCS16SI8N
C66
C67
C68
DESY
FS-EC
Notkestrasse 85
22607 Hamburg
joshua.supra@desy.de
CONFIDENTIAL MATERIAL
DESY reserves all rights according to
ISO 16016. © Project_CopyrightDate
TITLE:
ZMX-Handsteuerung Serie 3
SCHEMATIC:
FPGA
DATE: 27.07.2018 12:47:14
ENGINEER: Joshua Supra
DRAWN BY: Joshua Supra
Rev:
Size: A3
Sheet:
3 of 6
FILENAME: FPGA.SchDoc


uC_PROG
SWCLK
SWDIO
TRACESWO
P1
MOLEX 53398-0671
UART_DEBUG
UART5_TX
UART5_RX
UART_ZMX
UART4_TX
UART4_RX
ZMX_Error3V3
FPGA
EX_ENC_A
EX_ENC_B
EX_ENC_I
ZMX_CW
ZMX_CCW
EX_Step_INT
EX_Dir_INT
ZMX_Entregen
ZMX_DIR
ZMX_STEP
ZMX_Reset
FPGA_JTAG
FPGA_TCK
FPGA_TDO
FPGA_TMS
FPGA_TDI
P4
MOLEX 53398-0671
uC_Fan
LüfterANSignal3V3
LüfterTachoSignal3V3
LüfterNetzteilSignal3V3
P3
HDR-3x1
X2
D2540-6V0C-AR-WG
ZMX_Entregen_EXT
ZMX_DIR_EXT
ZMX_STEP_EXT
ZMX_Reset_EXT
ZMX_R+
ZMX_T+
ZMX_R-
ZMX_T-
+5V
EX_Step
EX_Dir
EX_CW
EX_CCW
EX_Limit_+15V
EX_ENC_A+
EX_ENC_B+
EX_ENC_I+
EX_ENC_A-
EX_ENC_B-
EX_ENC_I-
U12B
ADM3493ARZ
U6A
DS26C32AT
U14B
U6E
Q2
Q3
Q4
Q5
NTR4501NT1G
Q6
Q7
Q8
Q9
BC81740MTF
EX_CW
EX_CCW
Opto_1
Opto_2
Opto_3
Opto_4
Opto_5
Opto_6
Opto_7
Opto_8
U7
U8
HCPL-0631-000E
EX_CCW_INT
EX_CW_INT
EX_Step
EX_Dir
D7
D8
D9
D10
1N4148W
GND
U15
SFH6916
U16
HEF4001BT,653
U6B
U6C
U6D
DS26C32AT
U14A
U12A
ADM3493ARZ
UART4_TX
RO
/RE
DE
DI
DESY
FS-EC
Notkestrasse 85
22607 Hamburg
joshua.supra@desy.de
TITLE:
ZMX-Handsteuerung Serie 3
SCHEMATIC:
Connector
DATE: 27.07.2018 12:47:14
CONFIDENTIAL MATERIAL
DESY reserves all rights according to
ISO 16016. © Project_CopyrightDate
ENGINEER: Joshua Supra
DRAWN BY: Joshua Supra
Rev:
Size: A3
Sheet:
4 of 6
FILENAME: Connector.SchDoc

L1
C71
1n
C69
220u
C70
220u



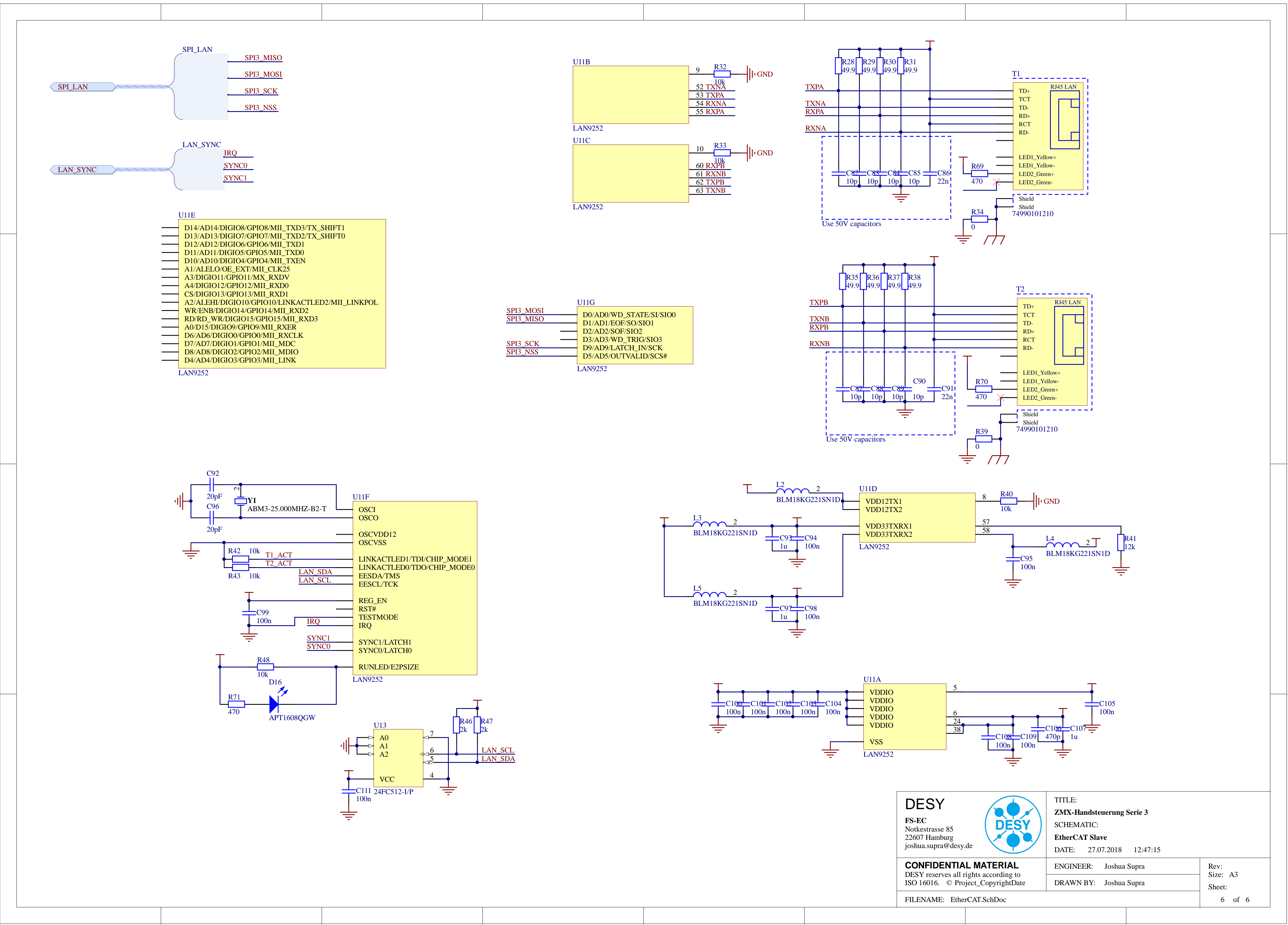
DESY
FS-EC
Notkestrasse 85
22607 Hamburg
joshua.supra@desy.de

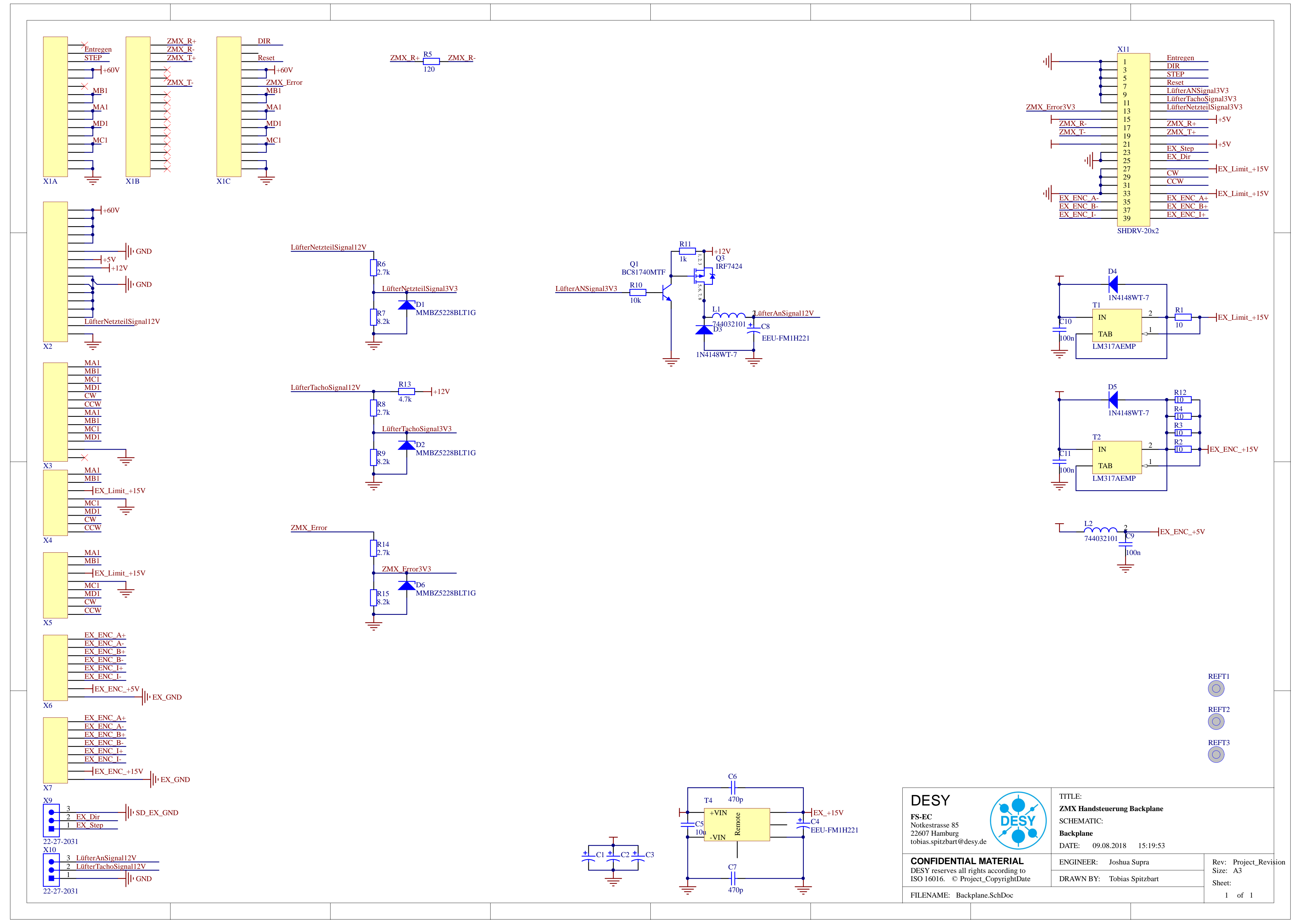
TITLE:
ZMX-Handsteuerung Serie 3
SCHEMATIC:
Powersupply
DATE: 27.07.2018 12:47:15

CONFIDENTIAL MATERIAL
DESY reserves all rights according to
ISO 16016. © Project_CopyrightDate

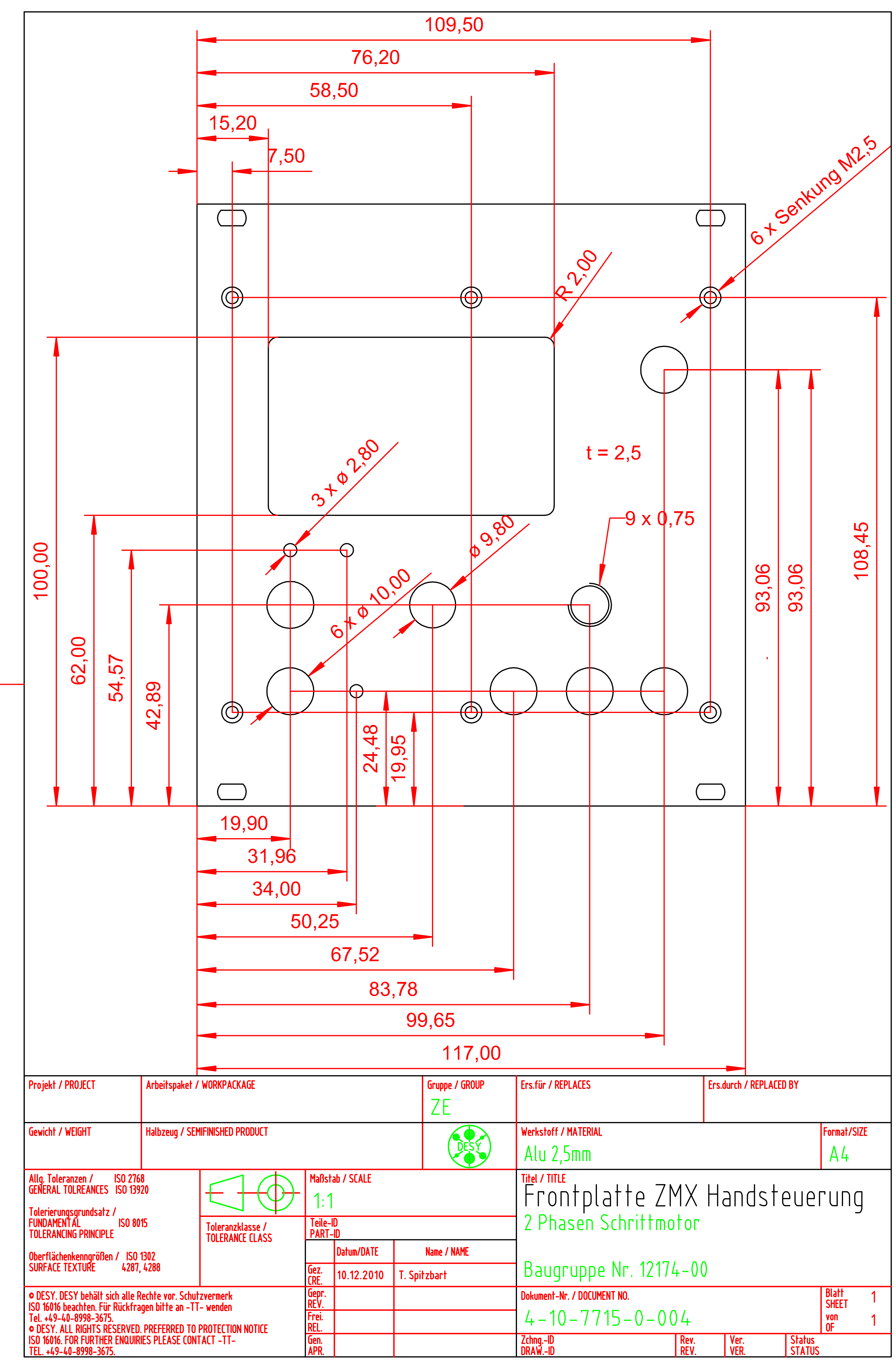
ENGINEER: Joshua Supra
DRAWN BY: Joshua Supra
Rev:
Size: A3
Sheet:
5 of 6

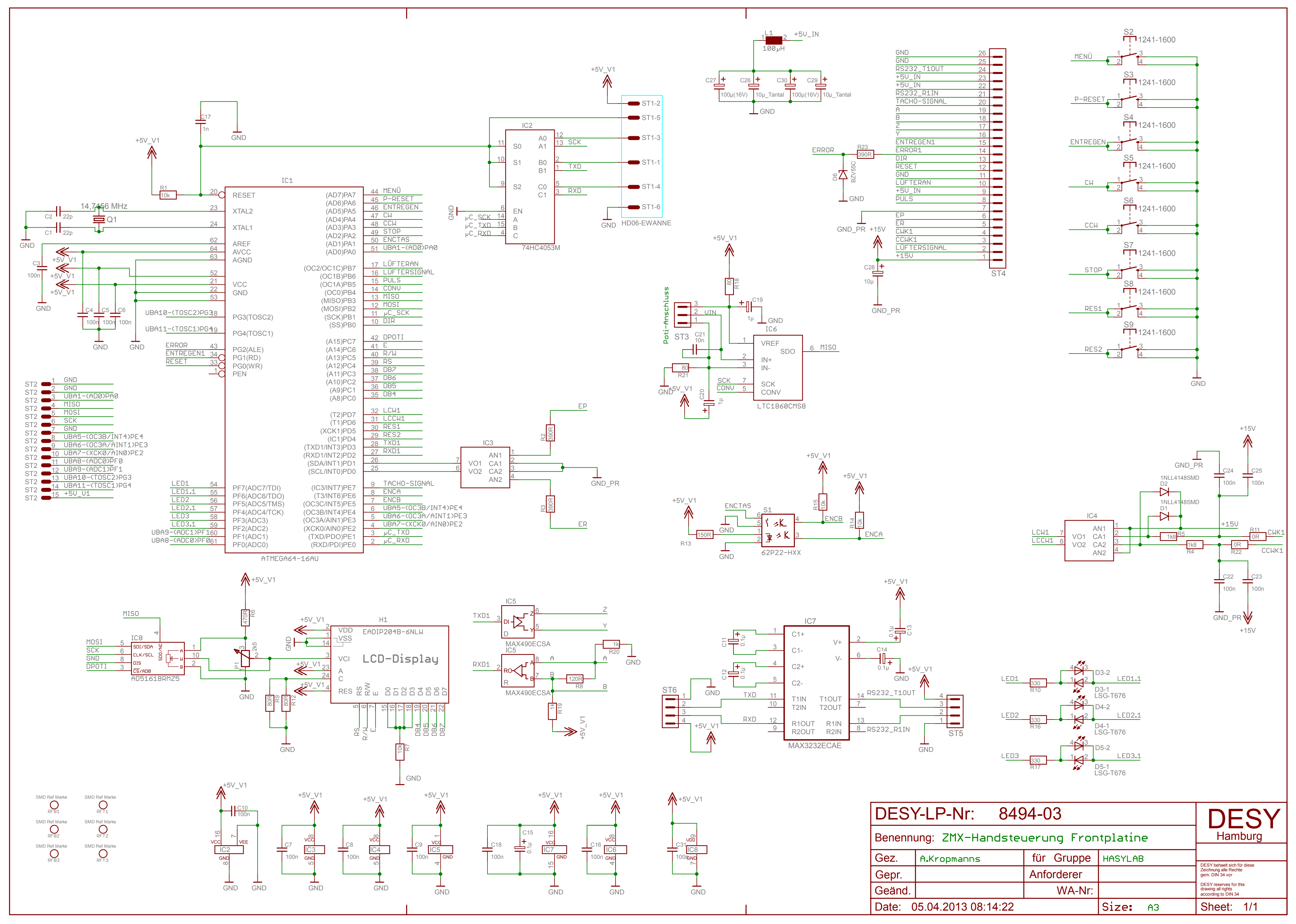
FILENAME: powersupply.SchDoc


SPI_LAN
SPI3_MISO
SPI3_MOSI
SPI3_SCK
SPI3_NSS
LAN_SYNC
IRQ
SYNC0
SYNC1
U11E
D14/AD14/DIGIO8/GPIO8/MII_TXD3/TX_SHIFT1
D13/AD13/DIGIO7/GPIO7/MII_TXD2/TX_SHIFT0
D12/AD12/DIGIO6/GPIO6/MII_TXD1
D11/AD11/DIGIO5/GPIO5/MII_TXD0
D10/AD10/DIGIO4/GPIO4/MII_TXEN
A1/ALELO/OE_EXT/MII_CLK25
A3/DIGIO11/GPIO11/MX_RXDV
A4/DIGIO12/GPIO12/MII_RXD0
CS/DIGIO13/GPIO13/MII_RXD1
A2/ALEHI/DIGIO10/GPIO10/LINKACTLED2/MII_LINKPOL
WR/ENB/DIGIO14/GPIO14/MII_RXD2
RD/RD_WR/DIGIO15/GPIO15/MII_RXD3
A0/D15/DIGIO9/GPIO9/MII_RXER
D6/AD6/DIGIO0/GPIO0/MII_RXCLK
D7/AD7/DIGIO1/GPIO1/MII_MDC
D8/AD8/DIGIO2/GPIO2/MII_MDIO
D4/AD4/DIGIO3/GPIO3/MII_LINK
LAN9252
U11B
R32 10k GND
52 TXNA
53 TXPA
54 RXNA
55 RXPA
LAN9252
U11C
R33 10k GND
60 RXPB
61 RXNB
62 TXPB
63 TXNB
LAN9252
U11G
SPI3_MOSI
SPI3_MISO
SPI3_SCK
SPI3_NSS
D0/AD0/WD_STATE/SI/SIO0
D1/AD1/EOF/SO/SIO1
D2/AD2/SOF/SIO2
D3/AD3/WD_TRIG/SIO3
D9/AD9/LATCH_IN/SCK
D5/AD5/OUTVALID/SCS#
LAN9252
R28 49.9
R29 49.9
R30 49.9
R31 49.9
T1
TXPA
TXNA
RXPA
RXNA
TD+
TCT
TD-
RD+
RCT
RD-
RJ45 LAN
LED1_Yellow+
LED1_Yellow-
LED2_Green+
LED2_Green-
Shield
Shield
74990101210
C82 10p
C83 10p
C84 10p
C85 10p
C86 22n
R69 470
R34 0
Use 50V capacitors
R35 49.9
R36 49.9
R37 49.9
R38 49.9
T2
TXPB
TXNB
RXPB
RXNB
C87 10p
C88 10p
C89 10p
C90 10p
C91 22n
R70 470
R39 0
Use 50V capacitors
C92 20pF
C96 20pF
Y1
ABM3-25.000MHZ-B2-T
U11F
OSCI
OSCO
OSCVDD12
OSCVSS
R42 10k
R43 10k
T1_ACT
T2_ACT
LAN_SDA
LAN_SCL
LINKACTLED1/TDI/CHIP_MODE1
LINKACTLED0/TDO/CHIP_MODE0
EESDA/TMS
EESCL/TCK
REG_EN
RST#
TESTMODE
IRQ
C99 100n
SYNC1
SYNC0
SYNC1/LATCH1
SYNC0/LATCH0
R48 10k
RUNLED/E2PSIZE
LAN9252
D16
R71 470
APT1608QGW
U13
A0
A1
A2
VCC
24FC512-I/P
R46 2k
R47 2k
LAN_SCL
LAN_SDA
C111 100n
L2 BLM18KG221SN1D
U11D
VDD12TX1
VDD12TX2
VDD33TXRX1
VDD33TXRX2
LAN9252
R40 10k GND
L3 BLM18KG221SN1D
C93 1u
C94 100n
L4 BLM18KG221SN1D
C95 100n
R41 12k
L5 BLM18KG221SN1D
C97 1u
C98 100n
U11A
VDDIO
VDDIO
VDDIO
VDDIO
VDDIO
VSS
LAN9252
C100 100n
C101 100n
C102 100n
C103 100n
C104 100n
C105 100n
C108 100n
C109 100n
C106 470p
C107 1u
DESY
FS-EC
Notkestrasse 85
22607 Hamburg
joshua.supra@desy.de
CONFIDENTIAL MATERIAL
DESY reserves all rights according to
ISO 16016. © Project_CopyrightDate
TITLE:
ZMX-Handsteuerung Serie 3
SCHEMATIC:
EtherCAT Slave
DATE: 27.07.2018 12:47:15
ENGINEER: Joshua Supra
DRAWN BY: Joshua Supra
Rev:
Size: A3
Sheet:
6 of 6
FILENAME: EtherCAT.SchDoc



X1A
Entregen
STEP
+60V
MB1
MA1
MD1
MC1
X1B
ZMX_R+
ZMX_R-
ZMX_T+
ZMX_T-
X1C
DIR
Reset
ZMX_Error
R5
120
ZMX_R+
ZMX_R-
X2
GND
+5V
+12V
LüfterNetzteilSignal12V
X3
MA1
MB1
MC1
MD1
CW
CCW
X4
EX_Limit_+15V
X5
X6
EX_ENC_A+
EX_ENC_A-
EX_ENC_B+
EX_ENC_B-
EX_ENC_I+
EX_ENC_I-
EX_ENC_+5V
EX_GND
X7
EX_ENC_+15V
X9
SD_EX_GND
EX_Dir
EX_Step
22-27-2031
X10
LüfterAnSignal12V
LüfterTachoSignal12V
R6
2.7k
LüfterNetzteilSignal3V3
D1
MMBZ5228BLT1G
R7
8.2k
LüfterTachoSignal12V
R13
4.7k
R8
2.7k
LüfterTachoSignal3V3
D2
R9
8.2k
ZMX_Error
R14
2.7k
ZMX_Error3V3
D6
R15
8.2k
Q1
BC81740MTF
LüfterANSignal3V3
R10
10k
R11
1k
Q3
IRF7424
L1
744032101
D3
1N4148WT-7
C8
EEU-FM1H221
X11
SHDRV-20x2
LüfterANSignal3V3
LüfterTachoSignal3V3
LüfterNetzteilSignal3V3
ZMX_Error3V3
EX_Step
EX_Dir
EX_ENC_+15V
D4
1N4148WT-7
T1
IN
TAB
LM317AEMP
C10
100n
R1
10
D5
T2
C11
R12
R4
R3
R2
L2
C9
EX_ENC_+5V
REFT1
REFT2
REFT3
C1
C2
C3
C6
470p
T4
+VIN
-VIN
Remote
C5
10u
C4
EX_+15V
C7
DESY
FS-EC
Notkestrasse 85
22607 Hamburg
tobias.spitzbart@desy.de
CONFIDENTIAL MATERIAL
DESY reserves all rights according to ISO 16016. © Project_CopyrightDate
TITLE:
ZMX Handsteuerung Backplane
SCHEMATIC:
Backplane
DATE: 09.08.2018 15:19:53
ENGINEER: Joshua Supra
DRAWN BY: Tobias Spitzbart
Rev: Project_Revision
Size: A3
Sheet: 1 of 1
FILENAME: Backplane.SchDoc

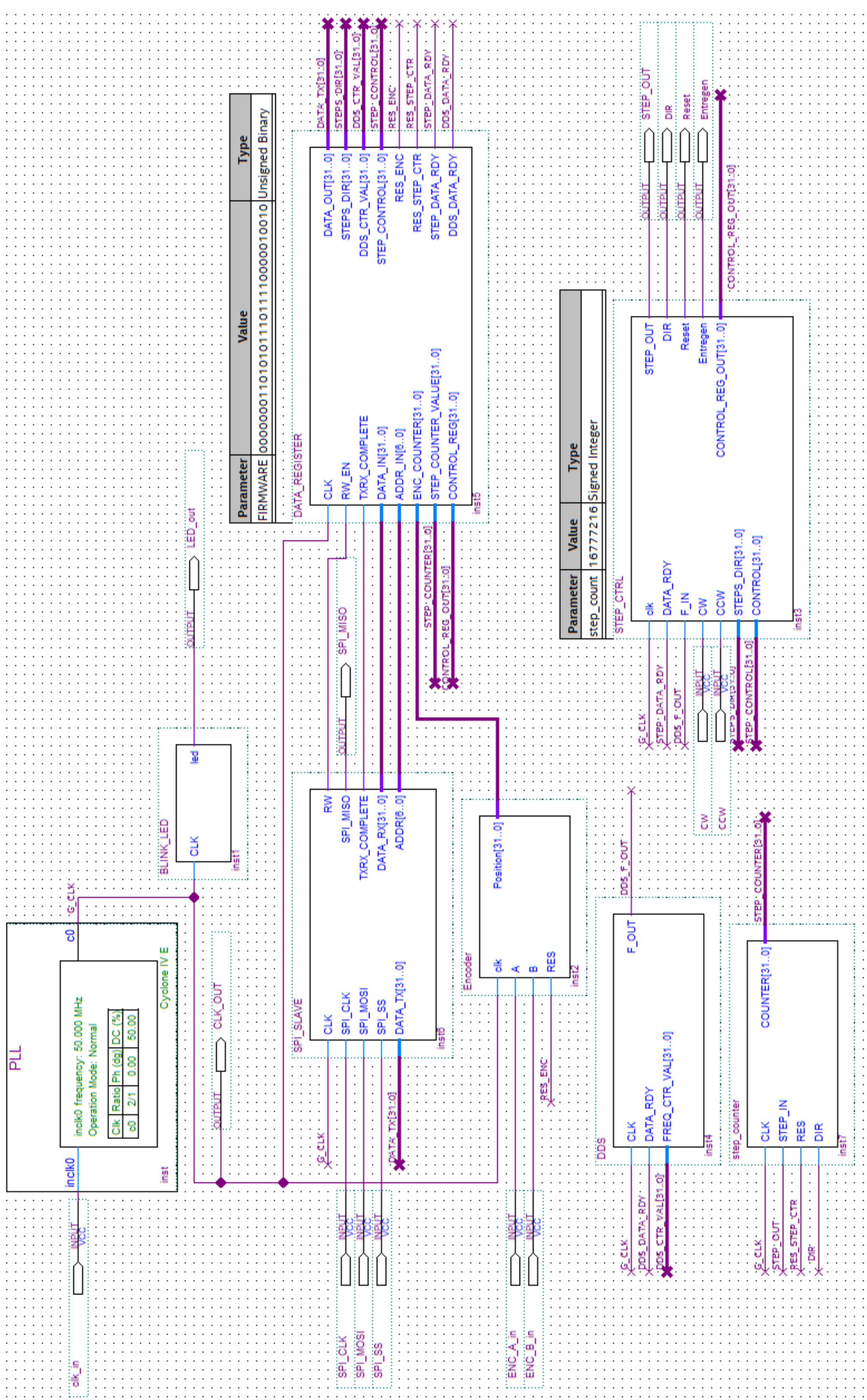

109,50
76,20
58,50
15,20
7,50
6 x Senkung M2,5
R 2,00
t = 2,5
3 x ø 2,80
ø 9,80
9 x 0,75
6 x ø 10,00
100,00
62,00
54,57
42,89
24,48
19,95
93,06
93,06
108,45
19,90
31,96
34,00
50,25
67,52
83,78
99,65
117,00
Projekt / PROJECT
Arbeitspaket / WORKPACKAGE
Gruppe / GROUP
ZE
Ers.für / REPLACES
Ers.durch / REPLACED BY
Gewicht / WEIGHT
Halbzeug / SEMIFINISHED PRODUCT
DESY
Werkstoff / MATERIAL
Alu 2,5mm
Format/SIZE
A4
Allg. Toleranzen / GENERAL TOLREANCES ISO 2768 ISO 13920
Tolerierungsgrundsatz / FUNDAMENTAL TOLERANCING PRINCIPLE ISO 8015
Oberflächenkenngrößen / SURFACE TEXTURE ISO 1302 4287, 4288
Toleranzklasse / TOLERANCE CLASS
Maßstab / SCALE
1:1
Teile-ID PART-ID
Datum/DATE
Name / NAME
Gez. CRE.
10.12.2010
T. Spitzbart
Gepr. REV.
Frei. REL.
Gen. APR.
Titel / TITLE
Frontplatte ZMX Handsteuerung
2 Phasen Schrittmotor
Baugruppe Nr. 12174-00
Dokument-Nr. / DOCUMENT NO.
4-10-7715-0-004
Blatt SHEET 1
von OF 1
Zchng.-ID DRAW.-ID
Rev. REV.
Ver. VER.
Status STATUS
© DESY. DESY behält sich alle Rechte vor. Schutzvermerk ISO 16016 beachten. Für Rückfragen bitte an -TT- wenden Tel. +49-40-8998-3675.
© DESY. ALL RIGHTS RESERVED. PREFERRED TO PROTECTION NOTICE ISO 16016. FOR FURTHER ENQUIRIES PLEASE CONTACT -TT- TEL. +49-40-8998-3675.



DESY-LP-Nr: 8494-03
Benennung: ZMX-Handsteuerung Frontplatine
Gez. A.Kropmanns
für Gruppe HASYLAB
Gepr.
Anforderer
Geänd.
WA-Nr:
Date: 05.04.2013 08:14:22
Size: A3
Sheet: 1/1
DESY
Hamburg
DESY behaelt sich für diese Zeichnung alle Rechte gem. DIN 34 vor
DESY reserves for this drawing all rights according to DIN 34
IC1
ATMEGA64-16AU
IC2
74HC4053M
HD06-EWANNE
ST4
IC3
IC4
IC5
MAX490ECSA
IC6
LTC1860CMS8
IC7
MAX3232ECAE
IC8
AD5161BRMZ5
H1
EADIP204B-6NLW
LCD-Display
Poti-Anschluss
ST3
ST5
ST6
62P22-HXX
14,7456 MHz
Q1
L1
100µH
1241-1600
LSG-T676
1NLL4148SMD
BZV55C
SMD Ref Marke

B. VHDL-Quellcode der Firmware vom FPGA

Der Anhang B enthält den VHDL-Quellcode des FPGAs. Am Anfang jeder Datei befindet sich eine Kurzbeschreibung, die einen Überblick über den Inhalt bietet. Abbildung **??** zeigt die Übersicht der Firmware des FPGA innerhalb der Entwicklungsumgebung (Quartus Prime).



Übersicht der Firmware im FPGA.

B.1. DDS.vhd

```
--Firma: DESY Hamburg
--Autor: Joshua Supra
--Projekt:   ZMX-Handsteuerung Serie 3
--Letzte Aenderung am: 10.07.2018
--Kurzbeschreibung: DDS zur Erzeugung der Frequenz von Pulsen fuer
    die ZMX-Endstufe
--(c) Copyright 2018 DESY
--/*********************************************************/
--/ * Kurzbeschreibung: DDS
--/ * Frequenz von 1Hz bis 100kHz
--/*********************************************************/
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
--/*********************************************************/
--/ * Port Deklaration DDS
--/*********************************************************/
entity DDS is
    port(
        CLK, DATA_RDY   : in STD_LOGIC;
        FREQ_CTR_VAL    : in STD_LOGIC_VECTOR (31 downto 0);
        F_OUT       : out STD_LOGIC
        );
end DDS;

architecture Verhalten of DDS is
--/*********************************************************/
--/ * Signal Deklarationen und ggf. Initialisierungen vom DDS
--/*********************************************************/
signal DDS_VALUE    : unsigned (31 downto 0) := x"00000000";--
   Wert mit dem DDS Counter bei jeden Takt inkrementiert wird
signal DDS_CTR      : unsigned (31 downto 0) := x"00000000";-- DDS
    Counter
signal F_OUT_TEMP   : STD_LOGIC := '0';

begin

process (CLK)
begin
--/*********************************************************/
--/ * Sequentielle Umgebung
--/*********************************************************/
```



```
if(CLK'event and CLK = '1') then
    if( DATA_RDY = '1') then    -- wenn neuer Zaehler Wert bereit
       liegt
        DDS_VALUE <= unsigned(FREQ_CTR_VAL);--in den Counter laden
    end if;
        DDS_CTR<=DDS_CTR + DDS_VALUE;-- DDS Counter um Zaehlerwert
            erhoehen

    if (DDS_CTR>(x"7FFFFFFF"))then  --wenn der Zaehler groesser
       als 2^16 ist
        F_OUT_TEMP<='0';            --Ausgang low setzen
    else
        F_OUT_TEMP<='1';            --ansosnten Ausgang high
           setzen
    end if;
end if;
end process;
--/*********************************************************/
--/ * Nebenlaeufige Anweisungen
--/ * Zuweisung der Signale an die Ports
--/*********************************************************/
F_OUT <= F_OUT_TEMP;
end Verhalten;
```

B.2. SPISlave.vhd

```
--Firma: DESY Hamburg
--Autor: Joshua Supra
--Projekt:   ZMX-Handsteuerung Serie 3
--Letzte Aenderung am: 20.06.2018
--Kurzbeschreibung: SPI Slave Schnittstelle fuer die Kommunikation
    zwischen Mikrocontroller und FPGA
--(c) Copyright 2018 DESY
--/*********************************************************/
--/ * Kurzbeschreibung: 6-Byte SPI Datenuebertragung:
--/     * 1 Byte: MSB: R/W, 6 bis 0: Adresse
--/     * 2 Byte: Dummy Byte damit FPGA genug Zeit hat Daten
   bereit zu stellen
--/     * 3 - 6 Byte: 32 Bit Daten, MSB zuerst
--/*********************************************************/
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
--/*********************************************************/
```

```
--/ * Port Deklaration des SPI Slaves
--/*********************************************************/
entity SPI_SLAVE is
    port(   CLK, SPI_CLK, SPI_MOSI, SPI_SS : in STD_LOGIC;
            RW, SPI_MISO,TXRX_COMPLETE    : out STD_LOGIC;
            DATA_RX : out STD_LOGIC_VECTOR (31 downto 0);   --
               Daten die an die Register gesendet werden
            DATA_TX: in STD_LOGIC_VECTOR (31 downto 0);     --
               Daten die aus Register gelesen werden
            ADDR: out STD_LOGIC_VECTOR (6 downto 0)         --
               Register Adresse
            );
end SPI_SLAVE;

architecture Verhalten of SPI_SLAVE is
--/*********************************************************/
--/ * Signal Deklarationen und ggf. Initialisierungen des SPI
   Slaves
--/*********************************************************/
signal SPI_CLK_SR   : STD_LOGIC_VECTOR(3 downto 0);
signal SPI_SS_SR    : STD_LOGIC_VECTOR(3 downto 0);

signal ADDRDY, DUMMYRDY, DATAINRDY, RW_TEMP, MISO_EN, SEND_RDY
       : STD_LOGIC := '0';
signal WRITE_TO_REG, READ_FROM_REG,TXRX_COMPLETE_TEMP :STD_LOGIC
   := '0';

signal SPI_CLK_CTR  : integer range 0 to 63 := 0;
signal DATA_BUFFER  : STD_LOGIC_VECTOR (31 downto 0) := x"00000000
   ";
signal ADDR_DATA    : STD_LOGIC_VECTOR (6 downto 0) := b"0000000";
signal MISO_BUFFER  : STD_LOGIC_VECTOR (31 downto 0) :=x"00000000"
   ;
--/*********************************************************/
--/ * Type Deklaration der SPI Slave State-Machine
--/*********************************************************/
type SPI_TRANSFER is(
    START_TRANS,        -- Anfang der Uebertragung
    ADDR_READY,         -- Adresse empfangen
    GET_DATA,           -- Daten werden aus Uebertragung entnommen
    RX_DATA,            -- Status zum Empfangen der Daten
    TX_DATA,            -- Status zum Senden der Daten
    EOT                 -- Ende der uebertragung
);
```



```
signal SPI_STATE : SPI_TRANSFER;

begin
process(CLK)
begin
--/*********************************************************/
--/ * Sequentielle Umgebung, taktsynchron
--/ * Eintakten vom SPI-CLOCK und CHIP-SELECT (active low)
--/ * 4 Bit Schieberegister um falsche Zustaende durch Stoerung
   auf den Leitungen zu verhindern
--/*********************************************************/
if(CLK'EVENT and CLK='1') then
                   -- Steigende Flanke am FPGA CLOCK
    SPI_CLK_SR <= SPI_CLK_SR (2) & SPI_CLK_SR (1) & SPI_CLK_SR (0)
        & SPI_CLK;   -- SPI-CLOCK wird eingetaktet
    SPI_SS_SR <= SPI_SS_SR (2) & SPI_SS_SR (1) & SPI_SS_SR (0) &
       SPI_SS;        -- CHIP-SELECT wird eingetaktet

    if(SPI_CLK_SR ="0111" and SPI_SS_SR ="0000") then   --
       Steigenden Flanke am SPI-CLOCK und low am CHIP-SELECT
        SPI_CLK_CTR <= SPI_CLK_CTR + 1;                 --
           Inkrementierung des CLOCK counters um 1
        DATA_BUFFER <= DATA_BUFFER (30 downto 0) & SPI_MOSI;    --
            Einlesen des jeweiligen MOSI Signals
    end if;
    if(MISO_EN = '1') then
        if(SPI_CLK_SR ="1000" and SPI_SS ='0') then -- fallende
           Flanke am SPI-CLOCK und CHIP-SELECT low
            MISO_BUFFER <= MISO_BUFFER(30 downto 0) & '0';  --
               MISO_BUFFER heraus takten
        end if;
    end if;

    if(SPI_SS_SR = "0011") then          -- steigende Flanke am
       CHIP-SELECT
        TXRX_COMPLETE_TEMP <= '1';       -- Datenuebertragungs-Flag
            wird 1 gesetzt
--/*********************************************************/
--/ * CHIP-SELECT high fuehrt zum Reset der SPI-Kommunikation,
   egal ob die
--/ * Datenuebertragung vollstaendig ist oder nicht.
--/ * Alle Flags und Counter werden zurueck gesetzt
--/*********************************************************/
```

```
    elsif(SPI_SS_SR = "1111") then
        TXRX_COMPLETE_TEMP <= '0';
        SPI_CLK_CTR <=0;
        SPI_STATE <= START_TRANS;
        ADDRDY <= '0';
        DUMMYRDY <= '0';
        DATAINRDY <= '0';
        SEND_RDY <= '0';
        MISO_EN <= '0';
        WRITE_TO_REG <= '0';
        READ_FROM_REG <= '0';
        DATA_BUFFER <= x"00000000";
    end if;
--/************************************************************/
--/ * Type Deklaration des SPI Slave Clock-Zaehlers
--/************************************************************/
    case SPI_CLK_CTR is
        when    8   =>  ADDRDY <= '1';
        when    16  =>  DUMMYRDY <= '1';
        when    17  =>  SEND_RDY <= '1';
        when    48  =>  DATAINRDY <= '1';
        when others =>
    end case;
--/************************************************************/
--/ * SPI Slave State-Machine
--/************************************************************/
    case SPI_STATE is
        when    START_TRANS     =>  if(ADDRDY = '1') then
                                        ADDR_DATA <= DATA_BUFFER
                                           (6 downto 0);  -- lese
                                           Adresse aus Daten
                                        RW_TEMP <= DATA_BUFFER (7)
                                           ;              -- lese R
                                           /W aus Daten
                                        SPI_STATE <= ADDR_READY;
                                                          -- gehe
                                            zum State Adresse
                                           liegt bereit
                                    end if;
        when    ADDR_READY      =>  if(RW_TEMP = '1' and DUMMYRDY
           = '1') then   -- Register soll geschrieben werden
                                        SPI_STATE <= RX_DATA;
                                                          --
```



```
                                            weiter im State
                                            empfange Daten
                                    elsif (RW_TEMP ='0') then
                                                        -- Register
                                        soll gelesen werden
                                        SPI_STATE <= GET_DATA;
                                                          --
                                            weiter im State hole
                                            Daten aus Register
                                    end if;
        when    GET_DATA        =>  READ_FROM_REG <= '1';
                           -- Register lese Flag wird high gesetzt
                                    if(READ_FROM_REG = '1') then
                                                        -- wenn
                                       Daten bereit liegen
                                        MISO_BUFFER <= DATA_TX;
                                                          --
                                            Register Daten in
                                            MISO_BUFFER kopieren
                                        READ_FROM_REG <='0';
                                                          --
                                            Register lese Flag wird
                                             low gesetzt
                                        SPI_STATE <= TX_DATA;
                                                          --
                                            weiter im State Sende
                                            Daten
                                    end if;
        when    RX_DATA         =>  if(DATAINRDY = '1') then
                           -- wenn Adresse und Daten empfangen
           wurden
                                        WRITE_TO_REG <= '1';
                                                          --
                                            setze Register schreib
                                            Flag high
                                        SPI_STATE <= EOT;
                                                          --
                                            weiter im State Ende
                                            der uebertragung
                                    end if;
        when    TX_DATA         =>  if(SEND_RDY = '1') then
                           -- wenn Daten zum Senden bereit liegen
                                        MISO_EN <= '1';
```

```
                                                              -- gebe
                                                MISO frei
                                        elsif (DATAINRDY = '1') then
                                                          -- wenn 48
                                             SPI Clocks gezaehlt wurden
                                            SPI_STATE <= EOT;
                                                              --
                                               weiter im State Ende
                                               der uebertragung
                                        end if;
        when    EOT             =>  ADDRDY <= '0';
                                -- Ruecksetze Clock-Counter Flags
                                        DUMMYRDY <= '0';
                                        DATAINRDY <= '0';
                                        SEND_RDY <= '0';
                                        READ_FROM_REG <= '0';
                                        SPI_CLK_CTR <=0;
                                                          -- Reset
                                             Clock-Counter
    end case;
end if;
end process;
--/*********************************************************/
--/ * Nebenlaeufige Anweisungen
--/ * Zuweisung der Signale an die Ports
--/*********************************************************/
ADDR <= ADDR_DATA when WRITE_TO_REG ='1' or READ_FROM_REG = '1'
   else (others => 'Z');--Adresse schreiben
RW <= RW_TEMP when WRITE_TO_REG ='1' or READ_FROM_REG = '1' else '
   Z';-- R/W schreiben
DATA_RX <= DATA_BUFFER when WRITE_TO_REG ='1' else (others => 'Z')
   ; -- Daten an Register uebergeben
SPI_MISO <= MISO_BUFFER(31) when SPI_CLK_CTR > 16 else '0';--
   Daten aus Register auf MISO legen
TXRX_COMPLETE <= TXRX_COMPLETE_TEMP;-- uebertragung abgeschlossen
   Flag
end Verhalten;
```

B.3. Registerverwaltung.vhd

```
--Firma: DESY Hamburg
--Autor: Joshua Supra
--Projekt:   ZMX-Handsteuerung Serie 3
```



```
--Letzte Aenderung am: 05.06.2018
--Kurzbeschreibung: Datenregister, Verteilung der Daten im FPGA,
   bzw. Auslesen der
--                  einzelnen Register und Rueckgabe an SPI-Slave,
    max Adressen
--                  7 Bit = 128 Adressen,
--(c) Copyright 2018 DESY
--/*********************************************************/
--/ Adresse:        Mode:       Beschreibung:
--/ 0x00            R           Lese aktuelle Firmware Version aus
--/ 0x01            R           Lese Encoder Wert aus
--/                 W           Reset Encoder
--/ 0x02            R           Lese Step Control Register und
   Limits
--/                 W           Schreibe Step Control Register
--/ 0x03            W           Schreibe Step Sollwert
--/ 0x04            W           Schreibe DDS Zaehler Wert
--/ 0x05            R           Lese Step-Counter Wert
--/                 W           Reset Step-Counter
--/*********************************************************/
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
--/*********************************************************/
--/ * Port Deklaration des Datenregisters
--/*********************************************************/
entity DATA_REGISTER is
    generic(    FIRMWARE: STD_LOGIC_VECTOR(31 downto 0) :=x"01
       ABBC12");--28032018
    port(   CLK, RW_EN,TXRX_COMPLETE: in STD_LOGIC;
            DATA_OUT : out STD_LOGIC_VECTOR (31 downto 0);
            STEPS_DIR : out STD_LOGIC_VECTOR (31 downto 0);
            DDS_CTR_VAL: out STD_LOGIC_VECTOR (31 downto 0);
            DATA_IN: in STD_LOGIC_VECTOR (31 downto 0);
            ADDR_IN: in STD_LOGIC_VECTOR (6 downto 0);
            ENC_COUNTER : in STD_LOGIC_VECTOR(31 downto 0);
            STEP_CONTROL: out STD_LOGIC_VECTOR(31 downto 0);
            STEP_COUNTER_VALUE: in STD_LOGIC_VECTOR(31 downto 0);
            CONTROL_REG: in STD_LOGIC_VECTOR(31 downto 0);
            RES_ENC, RES_STEP_CTR, STEP_DATA_RDY, DDS_DATA_RDY:
               out STD_LOGIC
            );
end DATA_REGISTER;
architecture Verhalten of DATA_REGISTER is
--/*********************************************************/
```

```
--/ * Signal Deklarationen und ggf. Initialisierungen des
   Datenregisters
--/**********************************************************/
signal ADDRESS: STD_LOGIC_VECTOR (6 downto 0) := "1111111";
signal DATA :   STD_LOGIC_VECTOR (31 downto 0) := x"FFFFFFFF";
signal DATA_IN_TEMP :   STD_LOGIC_VECTOR (31 downto 0) := x"
   FFFFFFFF";
signal RW_TEMP, RES_ENC_TEMP, STEP_CTRL_TEMP, DDS_DR_TEMP,
   STEP_GEN_CTR_DR_TEMP, STEP_CTR_DR_TEMP, RES_STEP_CTR_TEMP :
   STD_LOGIC := '0';

signal STEP_MODE: STD_LOGIC_VECTOR (31 downto 0) := x"FFFFFFFF";
signal ENC_COUNTER_VALUE: STD_LOGIC_VECTOR (31 downto 0) :=x"
   FFFFFFFF";
signal DDS_CTR_VAL_TEMP: STD_LOGIC_VECTOR (31 downto 0) :=x"
   00000000";
signal STEP_CTR: STD_LOGIC_VECTOR (31 downto 0) :=x"00000000";
begin

process(CLK, ADDRESS, RW_EN, DATA_IN)
begin
--/**********************************************************/
--/ * Sequentielle Umgebung, taktsynchron
--/ * eine abgeschlossene SPI uebertragung setzt alle Flags
   zurueck
--/**********************************************************/
if(CLK'event and CLK ='1') then
    if(TXRX_COMPLETE = '0') then
        RES_ENC_TEMP <= '0';
        STEP_CTRL_TEMP <= '0';
        DDS_DR_TEMP <= '0';
        STEP_GEN_CTR_DR_TEMP <= '0';
        STEP_CTR_DR_TEMP <= '0';
        RES_STEP_CTR_TEMP <= '0';
    end if;
--/**********************************************************/
--/ * Nur bei aenderung der Daten, Adresse oder R/W werden neue
   Werte vom Bus
--/ * uebernommen, bei Idle ist der Bus x"00000000"
--/**********************************************************/
    if((((DATA_IN /= DATA_IN_TEMP) and DATA_IN /= x"00000000")  or
         (ADDR_IN /= ADDRESS) or (RW_EN /= RW_TEMP))) then
        ADDRESS <= ADDR_IN;
        RW_TEMP <= RW_EN;
```



```
        DATA_IN_TEMP <= DATA_IN;
    elsif( not ((DATA_IN /= DATA_IN_TEMP) or (ADDR_IN /= ADDRESS)
       or (RW_EN /= RW_TEMP))) then
    end if;
--/**********************************************************/
--/ * Auswertung der Addresse (siehe oben) und R/W
--/**********************************************************/
case ADDRESS is
    when    "0000000"   =>  if(RW_TEMP = '0') then
                                DATA <= FIRMWARE;
                            end if;
    when    "0000001"   =>  if (RW_TEMP = '0') then
                                DATA <= ENC_COUNTER;
                            elsif (RW_TEMP = '1' and TXRX_COMPLETE
                                ='1' and RES_ENC_TEMP = '0') then
                                RES_ENC_TEMP <= '1';
                            end if;
    when    "0000010"   =>  if(RW_TEMP = '0') then
                                DATA <= CONTROL_REG;
                            elsif (RW_TEMP = '1' and TXRX_COMPLETE
                                ='1' and STEP_CTRL_TEMP = '0')
                               then
                                STEP_MODE <= DATA_IN_TEMP;
                                STEP_CTRL_TEMP <= '1';
                            end if;
    when    "0000011"   =>  if (RW_TEMP = '1' and TXRX_COMPLETE
       ='1' and STEP_GEN_CTR_DR_TEMP = '0') then
                                STEP_CTR <= DATA_IN_TEMP;
                                STEP_GEN_CTR_DR_TEMP <= '1';
                            end if;
    when    "0000100"   =>  if (RW_TEMP = '1' and TXRX_COMPLETE
       ='1' and DDS_DR_TEMP = '0') then
                                DDS_DR_TEMP <= '1';
                                DDS_CTR_VAL_TEMP <= DATA_IN_TEMP;
                            end if;
    when    "0000101"   =>  if (RW_TEMP = '1' and TXRX_COMPLETE =
       '1' and STEP_CTR_DR_TEMP = '0') then
                                RES_STEP_CTR_TEMP <= '1';
                                STEP_CTR_DR_TEMP <= '1';
                            elsif(RW_TEMP = '0') then
                                DATA <= STEP_COUNTER_VALUE;
                            end if;
    when    others      =>
end case;
```

```
end if;
end process;
--/**********************************************************/
--/ * Nebenlaeufige Anweisungen
--/ * Zuweisung der Signale an die Ports
--/ * Daten werden an die eizelnen Bausteine verteilt
--/**********************************************************/
DATA_OUT <= DATA;-- Daten fuer SPI TX
RES_ENC <= RES_ENC_TEMP;-- Reset Encoder Value
STEP_CONTROL <= STEP_MODE;-- Step generation Kontrollregister
STEP_DATA_RDY <= STEP_GEN_CTR_DR_TEMP;-- Data Ready Flag fuer Step
    generation
DDS_CTR_VAL <= DDS_CTR_VAL_TEMP;-- Counter Wert fuer DDS
DDS_DATA_RDY <= DDS_DR_TEMP;-- Data Ready Flag fuer DDS
STEPS_DIR <= STEP_CTR;-- Schritte und Richtungsregister fuer Step
   generation
RES_STEP_CTR <= RES_STEP_CTR_TEMP;-- Reset Step Counter
end Verhalten;
```

B.4. Impulszaeler.vhd

```
--Firma: DESY Hamburg
--Autor: Joshua Supra
--Projekt:   ZMX-Handsteuerung Serie 3
--Letzte Aenderung am: 29.06.2018
--Kurzbeschreibung: Zaehler um erzeugte Pulse zu zaehlen
--/**********************************************************/
--/ *
--/**********************************************************/
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.ALL;
--/**********************************************************/
--/ * Port Deklaration des Step-Counter
--/**********************************************************/
entity step_counter is
    port(CLK, STEP_IN, RES, DIR: in STD_LOGIC;
        COUNTER: out STD_LOGIC_VECTOR (31 downto 0)
        );
end step_counter;

architecture Verhalten of step_counter is
--/**********************************************************/
```



```
--/ * Signal Deklarationen und ggf. Initialisierungen des Step-
   Counters
--/**********************************************************/
signal STEP_SR: STD_LOGIC_VECTOR (1 downto 0) := "00";
signal COUNTER_TEMP: integer := 0;
begin
process(CLK, STEP_IN)
begin
--/**********************************************************/
--/ * Sequentielle Umgebung, taktsynchron
--/ * Eintakten von STEP_IN um Flanke zu detektieren
--/**********************************************************/
if(CLK'event and CLK ='1') then
    STEP_SR <= STEP_SR(0) & STEP_IN;
    if(RES = '1') then  -- wenn RES high
        COUNTER_TEMP <= 0;  -- Zaehler zuruecksetzen
    elsif(STEP_SR = "01") then-- wenn "01" d.h. steigende Flanke
       an STEP_IN
        if(DIR = '1') then--Richtung CW
            COUNTER_TEMP <= COUNTER_TEMP + 1;-- Counter
               inkrementieren
        elsif(DIR = '0') then--Richtung CCW
            COUNTER_TEMP <= COUNTER_TEMP - 1;--Counter
               dekrementieren
        end if;
    end if;
end if;
end process;
--/**********************************************************/
--/ * Nebenlaeufige Anweisungen
--/ * Zuweisung der Signale an die Ports
--/**********************************************************/
COUNTER <= std_logic_vector(to_signed(COUNTER_TEMP, 32));
end Verhalten;
```

B.5. Encoder.vhd

```
-- Der folgende Quellcode stammt wenn nicht anders Vermerkt von
   [1].
-- Es wurden Anpassungen vorgenommen, die entweder mit
-- (hinzugefuegt) = dem original Quellcode hinzugefuegter Code
-- oder mit
-- (angepasst) = aenderung am original Quellcode vorgennommen
```

```
-- kenntlich gemacht wurden.
-- [1] Lothar Miller, 21.6.2018, http://www.lothar-miller.de/s9y/
   categories/46-Encoder
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;
            --Reset Eingang fuer Ruecksetzen des Zaehlers (
               hinzugefuegt)
            RES : in     STD_LOGIC;
            --
               ------------------------------------------------------

           Position : 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 i : std_logic_vector(1 downto 0);
signal e : std_logic_vector(1 downto 0);
begin
  process begin -- Eintakten der asynchronen Signale
    wait until rising_edge(clk);
    i <= A & B;  -- Zusammenfassen der Eingaenge A und B
    e <= i;
  end process;

  process(clk)      -- Weiterschalten und Zaehlen
  variable cu, cd : std_logic := '0';
  begin
    -- Reset "high" bedeutet Ruecksetzen des Zaehlers (
       hinzugefuegt)
    if(RES = '1') then
        p <= 0;
    ------------------------------------------------------------
    elsif(clk = '1' and clk'event) then
    cu := '0'; -- lokale Werte
    cd := '0';
```



```
    case z is
    -- Zaehlerverhalten wurde der Vorzugsdrehrichtung fuer die ZMX
       -Endstufe angepasst,
    -- so dass CW drehend inkrementiert und CCW drehend
       dekremeniert (angepasst)
      when Z00 => if    (e = "01") then z <= Z01; cd := '1';
                  elsif (e = "10") then z <= Z10; cu := '1';
                  end if;
      when Z01 => if    (e = "11") then z <= Z11; cd := '1';
                  elsif (e = "00") then z <= Z00; cu := '1';
                  end if;
      when Z11 => if    (e = "10") then z <= Z10; cd := '1';
                  elsif (e = "01") then z <= Z01; cu := '1';
                  end if;
      when Z10 => if    (e = "00") then z <= Z00; cd := '1';
                  elsif (e = "11") then z <= Z11; cu := '1';
                  end if;
    --
       ----------------------------------------------------------------------

    end case;
    if    (cu='1') then
        p <= p+1;
    elsif (cd='1') then
        p <= p-1;
    end if;
    end if;
  end process;

  Position <= std_logic_vector(to_signed(p,32)); -- Position
     ausgeben
end Behavioral;
```

B.6. Stepgeneration.vhd

```
--Firma: DESY Hamburg
--Autor: Joshua Supra
--Projekt:   ZMX-Handsteuerung Serie 3
--Erstellt am: 28.08.2018
--Kurzbeschreibung: Impulserzeugung fuer die ZMX-Endstufe
--(c) Copyright 2018 DESY
--/**********************************************************/
--/ * Register: Step Control Register
```

```
--/              Step Count & direction
--/*********************************************************/
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE ieee.numeric_std.ALL;
--/*********************************************************/
--/ * Port Deklaration
--/ * Statische Variablen
--/*********************************************************/
entity STEP_CTRL is
    generic (step_count : integer := 16777216);-- maximale
       moegliche Pulse
    Port ( clk, DATA_RDY, F_IN, CW, CCW :   in  STD_LOGIC;--
       Eingangsbits: Clock, Daten stehen bereit, Frequenz vom DDS,
        Endlagen CW und CCW
            STEPS_DIR   :   in STD_LOGIC_VECTOR(31 downto 0);--
               Eingangsvector Anzahl der Schritte und Drehrichtung
            CONTROL     :   in STD_LOGIC_VECTOR(31 downto 0);--
               Eingangsvector Kontrollregister
            STEP_OUT, DIR, Reset, Entregen :    out STD_LOGIC;--
               Ausgangsbits: Impulsausgang, Drehrichtung, Reset
               ZMX-Endstufe, Entregen ZMX-Endstufe
            CONTROL_REG_OUT: out STD_LOGIC_VECTOR(31 downto 0));--
                Ausgangsvector: Ausgabe vom Kontrollregister,
               enthaelt Zustand der Endlagen
end STEP_CTRL;

architecture Verhalten of STEP_CTRL is
--/*********************************************************/
--/ * Type Deklaration der Impulserzeugung
--/*********************************************************/
type Impuls_Gen is (
    Init_ctr,-- Status Initialisierung
    Impuls_high,-- Status fuer Impuls "igh" Zeit
    Impuls_low-- Status fuer Impuls "low" Zeit
);
--/*********************************************************/
--/ * Signal Deklarationen und ggf. Initialisierungen
--/*********************************************************/
signal Impuls : Impuls_Gen;-- Erzeuge Signal vom Typ Impuls_Gen
signal output, INIT :  STD_LOGIC := '0';

signal DIR_TEMP: STD_LOGIC := '0';-- Richtung Temp
```



```
signal STEPS: STD_LOGIC_VECTOR (23 downto 0);-- Register fuer
   Schritte
signal CONTROL_TEMP: STD_LOGIC_VECTOR (1 downto 0);--Register fuer
    einstellen des Laufmodus
signal CONTROL_LIMITS_TEMP_CW, CONTROL_LIMITS_TEMP_CCW:
   STD_LOGIC_VECTOR (3 downto 0);--Register fuer einstellen der
   Endlagen
signal F_IN_SR  : STD_LOGIC_VECTOR (1 downto 0) := "00";--
   Schieberegister zum eintakten der Frequenz aus dem DDS,
   Flankenerkennung
signal CW_IN_SR : STD_LOGIC_VECTOR (3 downto 0) := "0000";--
   Schieberegister zum eintakten der CW Endlage fuer Entprellung
signal CCW_IN_SR : STD_LOGIC_VECTOR (3 downto 0) := "0000";--
   Schieberegister zum eintakten der CCW Endlage fuer Entprellung
signal LIMIT_DISABLE : STD_LOGIC := '0';--Limits deaktiviert Bit
signal CONTROL_REG_OUT_TEMP: STD_LOGIC_VECTOR(31 downto 0);--
   Register zum zwischenspeichern der Ausgabe vom Kontrollregister
signal get_counter: STD_LOGIC := '0';--counter Flag

signal step_ctr: integer range 0 to step_count;-- Zaehler fuer
   Schritte im n-Schritte Modus

signal MODE_N_STEP, MODE_CONT: STD_LOGIC := '0';-- Modi fuer
   Laufmodus, n-Schritte oder kontinuierlich
signal CW_ACT_HIGH, CW_ACT_LOW, CW_DIS, CCW_ACT_HIGH, CCW_ACT_LOW,
    CCW_DIS: STD_LOGIC := '0';-- Zustaende der moeglichen
   Endlageneinstellung, "high", "low" oder "disabled"
signal CW_LIMIT, CCW_LIMIT: STD_LOGIC := '0';-- Bit fuer
   ausgeloeste Endlage
signal Reset_temp, Entregen_temp: STD_LOGIC := '0';--
   Zwischenspeicher Reset ZMX-Endstufe, Entregen ZMX-Endstufe
signal IO_temp: STD_LOGIC_VECTOR (2 downto 0):="001";--
   Zwischenspeicher fuer Auswertung von Reset und Entregen im
   Kontrollregister
begin
CREATE_IMPULS: process (clk, F_IN)
begin
    if(clk'event and clk = '1')then--wenn steigender Flanke am
       Clock wird
        F_IN_SR <= F_IN_SR (0) & F_IN;-- Zustand des DDS
           eingetaktet
        CW_IN_SR <= CW_IN_SR (2) & CW_IN_SR (1) & CW_IN_SR (0) &
           CW;--Zustand der Endlage CW eingetaktet
```

```
        CCW_IN_SR <= CCW_IN_SR (2) & CCW_IN_SR (1) & CCW_IN_SR (0)
            & CCW;--Zustand der Endlage CCW eingetaktet
        if(DATA_RDY = '1' and get_counter= '0') then-- wenn Daten
           im Step_Dir Register bereit liegen wird
            get_counter <= '1';
            DIR_TEMP <= STEPS_DIR(31);--die Drehrichtung
               ausgelesen
            step_ctr <= step_ctr + to_integer(unsigned(STEPS_DIR
               (23 downto 0)));-- die Anzahl der auszufuehrenden
               Schritte in den Zaehler fuer Schritte geladen
        elsif(DATA_RDY = '0' and get_counter= '1')then
            get_counter<= '0';
        end if;
--/*********************************************************/
--/ * Auswertung Reset und Entregen
--/*********************************************************/
        IO_temp <= CONTROL (26 downto 24);--Zwischenspeicher fuer
           Auswertung von Reset und Entregen im Kontrollregister
           wird beschrieben
        case IO_temp is
        -- Eingaenge sind active low
            when    "000"   =>  Reset_temp <= '1';-- wenn "00"
               kein reset der Endstufe
                                Entregen_temp <= '1';-- Endstufe
                                   ist inaktiv
            when    "001"   =>  Reset_temp <= '1';-- wenn "01"
               kein reset der Endstufe
                                Entregen_temp <= '0';-- Endstufe
                                   ist aktiv
            when    "010"   =>  Reset_temp <= '0';-- wenn "10"
               reset der Endstufe
                                Entregen_temp <= '1';-- Endstufe
                                   ist inaktiv
            when    "011"   =>  Reset_temp <= '0';--wenn "11"
               reset der Endstufe
                                Entregen_temp <= '0';--Endstufe
                                   ist aktiv
            when    others  =>
        end case;
--/*********************************************************/
--/ * Auswertung Limit-switch Kontrollregister, Endlagen werden
   entsprechend
--/     gesetzt
--/ * Reset "000" senden
```



```
--/*********************************************************/
        CONTROL_LIMITS_TEMP_CW <= CONTROL (15 downto 12);
        case CONTROL_LIMITS_TEMP_CW is
            -- Ruecksetzen der Limit Einstellungen
            when    "0000"  =>  CW_ACT_HIGH <= '0';
                                CW_ACT_LOW <= '0';
                                CW_DIS <= '0';
                                -- CW ist active low
            when    "0010"  =>  CW_ACT_LOW <= '1';
                                CW_ACT_HIGH <= '0';
                                CW_DIS <= '0';
                                -- CW ist active high
            when    "0001"  =>  CW_ACT_LOW <= '0';
                                CW_ACT_HIGH <= '1';
                                CW_DIS <= '0';
                                -- CW ist disabled
            when    "0011"  =>  CW_ACT_LOW <= '0';
                                CW_ACT_HIGH <= '0';
                                CW_DIS <= '1';
            when    others  =>
        end case;
        CONTROL_LIMITS_TEMP_CCW <= CONTROL (11 downto 8);
        case CONTROL_LIMITS_TEMP_CCW is
            when    "0000"  =>  CCW_ACT_HIGH <= '0';
                                CCW_ACT_LOW <= '0';
                                CCW_DIS <= '0';
                                -- CCW ist active low
            when    "0010"  =>  CCW_ACT_LOW <= '1';
                                CCW_ACT_HIGH <= '0';
                                CCW_DIS <= '0';
                                -- CCW ist active high
            when    "0001"  =>  CCW_ACT_LOW <= '0';
                                CCW_ACT_HIGH <= '1';
                                CCW_DIS <= '0';
                                -- CCW ist disabled
            when    "0011"  =>  CCW_ACT_LOW <= '0';
                                CCW_ACT_HIGH <= '0';
                                CCW_DIS <= '1';
            when    others  =>
        end case;
--/*********************************************************/
--/ * Auswertung Endlagen
--/*********************************************************/
```

```
        if (CW_ACT_LOW = '1' and CW_IN_SR="0000") then-- wenn CW
           active low ist und CW Schieberegister komplett low
            CW_LIMIT <= '1';-- wird CW Endlage als betaetigt
               erkannt
        elsif (CW_ACT_HIGH = '1' and CW_IN_SR = "1111") then--
           wenn CW active high ist und CW Schieberegister komplett
            high
            CW_LIMIT <= '1';-- wird CW Endlage als betaetigt
               erkannt
        end if;
        if (CCW_ACT_LOW = '1' and CCW_IN_SR = "0000") then-- wenn
           CCW active low ist und CCW Schieberegister komplett low
            CCW_LIMIT <= '1';-- wird CCW Endlage als betaetigt
               erkannt
        elsif (CCW_ACT_HIGH = '1' and CCW_IN_SR = "1111") then--
           wenn CCW active high ist und CCW Schieberegister
           komplett high
            CCW_LIMIT <= '1';-- wird CCW Endlage als betaetigt
               erkannt
        end if;
        if (CW_ACT_LOW = '1' and CW_IN_SR= "1111") then-- wenn CW
           active low ist und CW Schieberegister komplett high
            CW_LIMIT <= '0';--wird CW Endlage als NICHT betaetigt
               erkannt
        elsif (CW_ACT_HIGH = '1' and CW_IN_SR = "0000") then--
           wenn CW active high ist und CW Schieberegister komplett
            low
            CW_LIMIT <= '0';--wird CW Endlage als NICHT betaetigt
               erkannt
        end if;
        if (CCW_ACT_LOW = '1' and CCW_IN_SR = "1111") then-- wenn
           CCW active low ist und CCW Schieberegister komplett
           high
            CCW_LIMIT <= '0';--wird CCW Endlage als NICHT
               betaetigt erkannt
        elsif (CCW_ACT_HIGH ='1' and CCW_IN_SR = "0000") then--
           wenn CCW active high ist und CCW Schieberegister
           komplett low
            CCW_LIMIT <= '0';--wird CCW Endlage als NICHT
               betaetigt erkannt
        end if;
--/**********************************************************/
--/ * Auswertung Endlagen LIMIT_DISABLE
--/**********************************************************/
```



```
        if (CW_ACT_LOW = '1' and CW_IN_SR="0000" and DIR_TEMP =
           '1' ) then-- hat die Endlage CW ausgeloest und die
           Drehrichtung ist CW (CW act low)
            LIMIT_DISABLE <= '1';--wird der Vorschub verhindert
        elsif (CW_ACT_HIGH = '1' and CW_IN_SR = "1111" and
           DIR_TEMP = '1') then-- hat die Endlage CW ausgeloest
           und die Drehrichtung ist CW (CW act high)
            LIMIT_DISABLE <= '1';--wird der Vorschub verhindert
        elsif (CCW_ACT_LOW = '1' and CCW_IN_SR = "0000" and
           DIR_TEMP = '0' ) then-- hat die Endlage CCW ausgeloest
           und die Drehrichtung ist CCW (CCW act low)
            LIMIT_DISABLE <= '1';--wird der Vorschub verhindert
        elsif (CCW_ACT_HIGH = '1' and CCW_IN_SR = "1111" and
           DIR_TEMP = '0') then-- hat die Endlage CCW ausgeloest
           und die Drehrichtung ist CCW (CCW act low)
            LIMIT_DISABLE <= '1';--wird der Vorschub verhindert
        end if;

        if(LIMIT_DISABLE = '1') then
            step_ctr <= 0;-- Soll Schrittzaehler wird auf 0
               gesetzt
        end if;

--/**********************************************************/
--/ * Auswertung Kontrollregister
--/**********************************************************/
        CONTROL_TEMP <= CONTROL (1 downto 0);
            case CONTROL_TEMP is
            --Stop Impuls Erzeugung, reset aller Endlagen, reset
               des Run-Mode
                when    "00"    =>  MODE_CONT <= '0';
                                    MODE_N_STEP <= '0';
                                    Impuls <= Init_ctr;-- Impuls
                                       State-Machine wird auf init
                                        zurueck gesetzt
                                    output <= '0';-- Impulsausgang
                                        wird auf low gesetzt
                                    step_ctr <= 0;-- Soll
                                       Schrittzaehler wird auf 0
                                       gesetzt
                                    LIMIT_DISABLE <= '0';
                when    "01"    =>  MODE_CONT <= '1';--
                   einschalten des kontinuierlichen Modus
                                    MODE_N_STEP <= '0';--
```

```
                                   ausschalten des n-Schritte
                                   Modus
               when    "10"    =>  MODE_N_STEP <= '1';--
                  einschalten des n-Schritte Modus
                                MODE_CONT <= '0';--
                                   ausschalten des
                                   kontinuierlichen Modus

               when    others  =>
           end case;
--/*********************************************************/
--/ * Ausgabe des Control Registers, Endlagen, sowie Einstellungen
--/*********************************************************/
       CONTROL_REG_OUT_TEMP(1 downto 0) <= CONTROL_TEMP;
       CONTROL_REG_OUT_TEMP(4) <= DIR_TEMP;
       CONTROL_REG_OUT_TEMP(11 downto 8) <=
          CONTROL_LIMITS_TEMP_CCW;
       CONTROL_REG_OUT_TEMP(15 downto 12) <=
          CONTROL_LIMITS_TEMP_CW;
       CONTROL_REG_OUT_TEMP(16) <= CW_LIMIT;
       CONTROL_REG_OUT_TEMP(17) <= CCW_LIMIT;
       if(step_ctr >0)then-- wenn noch Schritte ausgefuehrt
          werden sollen
           CONTROL_REG_OUT_TEMP(19) <= '1';
       else
           CONTROL_REG_OUT_TEMP(19) <= '0';
       end if;
--/*********************************************************/
--/ * Kontinuierlicher Modus, Frequenz ueber DDS einstellbar
--/*********************************************************/
       if(MODE_CONT = '1' and LIMIT_DISABLE = '0') then--solange
          der Modus kontinuierlich ist und keine Endlage
          ausgeloest hat
           case Impuls is--wird ein Impuls erzeugt
               when Init_ctr       =>  output <= '0';-- Im init
                  wird der Impulsausgang an die Endstufe auf low
                  gesetzt
                                    Impuls <= Impuls_high;--
                                       weiter im State fuer
                                       Impuls high
               when Impuls_high    =>  if(F_IN_SR ="01") then--
                  wenn eine steigende Flanke vom DDS kommt
                                        output <= '1';--wird
                                           der Impulsausgang
```



```
                                               an die Endstufe auf
                                                high gesetzt
                                            Impuls <= Impuls_low;
                                               --wird in den
                                               Zustand Impuls low
                                               gewechselt
                                        end if;
               when Impuls_low     =>  if(F_IN_SR ="10") then--
                  wenn eine fallende Flanke vom DDS kommt
                                            output <= '0';--wird
                                               der Impulsausgang
                                               an die Endstufe auf
                                                low gesetzt
                                            Impuls <= Impuls_high;
                                               -- und der Zustand
                                               wechselt wieder zum
                                                Impuls high
                                        end if;
           end case;
--/*********************************************************/
--/ * n-Puls Modus, Frequenz durch DDS vorgegeben (einstellbar),
   erzeugt bis
--/   zu 16777216 aufeinander folgende Pulse
--/*********************************************************/
       elsif(MODE_N_STEP = '1' and LIMIT_DISABLE = '0') then--
          wenn der Modus n-Step ist und keine Endlage ausgeloest
          hat
           case Impuls is
               when Init_ctr       =>  output <= '0';-- Im init
                  wird der Impulsausgang an die Endstufe auf low
                  gesetzt
                                        if(step_ctr >0)then-- wenn
                                            noch Schritte
                                           ausgefuehrt werden
                                           sollen
                                            Impuls <= Impuls_high;
                                               -- weiter im State
                                               fuer Impuls high
                                        end if;
               when Impuls_high    =>  if(F_IN_SR ="01") then--
                  wenn eine steigende Flanke vom DDS kommt
                                            output <= '1';--wird
                                               der Impulsausgang
```

```
                                                    an die Endstufe auf
                                                     high gesetzt
                                                 Impuls <= Impuls_low;
                                                    --wird in den
                                                    Zustand Impuls low
                                                    gewechselt
                                             end if;
                when Impuls_low     =>  if(F_IN_SR ="10") then--
                   wenn eine fallende Flanke vom DDS kommt
                                                 output <= '0';--wird
                                                    der Impulsausgang
                                                    an die Endstufe auf
                                                     low gesetzt
                                                 step_ctr <= step_ctr -
                                                     1;-- Schritt
                                                    Sollwert um eins
                                                    dekrementiert
                                                 Impuls <= Init_ctr;--
                                                    und in den Init
                                                    Zustand gewechselt
                                             end if;
            end case;
        end if;
        end if;
    end process CREATE_IMPULS;
--/**********************************************************/
--/ * Nebenlaeufige Anweisungen
--/ * Zuweisung der Signale an die Ports
--/**********************************************************/
    Entregen <= Entregen_temp;
    Reset <= Reset_temp;
    STEP_OUT <= output;
    DIR <= DIR_TEMP;
    CONTROL_REG_OUT <= CONTROL_REG_OUT_TEMP;
end Verhalten;
```

C. Quellcode der Software des Mikrocontrollers: Header Dateien

Der Anhang C enthält die Header Dateien der Software des Mikrocontrollers. Am Anfang jeder Datei befindet sich eine Kurzbeschreibung, die einen Überblick über den Inhalt bietet.

C.1. main.h

```
/***********************************************************
 * Firma: DESY Hamburg
 * Autor: Joshua Supra
 * Projekt: ZMX-Handsteuerung Serie 3
 * Datei: main.h
 * Letzte Aenderung: 30.08.2018
 * Kurzbeschreibung: Hauptprogramm: Auswertung der Eingabeelemente
   , Ansteuerung
 *      LEDs, Visualisierung der Parameter im Display,
   Parametrierung der
 *      ZMX-Endstufe und Parametrierung des FPGAs.
 * (c) Copyright 2018 DESY
***********************************************************/
#ifndef MAIN_H_
#define MAIN_H_

/*Defines für den Autodrive */
#define     AUTODRIVE_FAKTOR 500 //Multiplikator
#define     AUTODRIVE_MIN    0
#define     AUTODRIVE_MAX    60000

/*Defines für den Singlestep*/
#define     SINGLESTEP_MIN   1
#define     SINGLESTEP_MAX   100
/***********************************************************
 * Globale Variablen
```

```
***********************************************************/
/*Übertragung zum Display*/
volatile uint8_t display_tx_data;//Sende Daten Flag
volatile uint8_t display_tx_busy;//Daten werden gerade gesendet
   Flag
volatile uint16_t display_tx_wait;//Wartezeit nach übertragung
volatile uint8_t display_tx_data_ctr;//Anzahl der Datensätze im
   Display Buffer
/*Display Flags, Wert der aktualisiert werden soll*/
volatile uint8_t update_speed;
volatile uint8_t update_Irun;
volatile uint8_t update_Istop;
volatile uint8_t update_res;
volatile uint8_t update_cw;
volatile uint8_t update_ccw;
volatile uint8_t update_enc_speed;
volatile uint8_t update_single_step;
volatile uint8_t update_autodrive;
/*Taster Auswertung*/
volatile uint8_t button_get_state;//gate zum Auswerten der Taster
/*Endstufenparameter*/
volatile int8_t Irun;//Laufstrom
volatile int8_t Irun_old;
volatile int8_t Istop;//Stoppstrom
volatile int8_t Istop_old;
volatile int8_t res;//Schrittauflösung
volatile int8_t res_old;
/*Menüführung im Display*/
volatile uint8_t menu_state;
volatile uint8_t menu_state_old;
volatile uint8_t menu_print_select;//Parameter ausgewählt
volatile uint8_t menu_select;

volatile uint8_t Blink_ctr;//Zähler für blinkenden Char
/*FPGA Variablen und Flags*/
volatile uint8_t fpga_steps_enc;
volatile uint8_t fpga_steps_enc_old;
volatile uint8_t fpga_update;
//Empfangsbuffer
volatile int32_t fpga_receive_data;
volatile int32_t fpga_receive_enc;
volatile int32_t fpga_receive_steps;
//Werte für/aus Kontrollregister
volatile int8_t CCW_Limit;//Zustand CCW Endlage
```



```
volatile int8_t CW_Limit;//Zustand CW Endlage
volatile uint8_t zmx_entregen;//ZMX Zustand
volatile uint8_t zmx_entregen_old;
volatile uint8_t zmx_reset;
volatile uint8_t fpga_limit_cw;//Einstellung CW Endlage
volatile uint8_t fpga_limit_cw_old;
volatile uint8_t fpga_limit_ccw;//Einstellung CW Endlage
volatile uint8_t fpga_limit_ccw_old;
volatile uint8_t fpga_ctrl_mode;//Buffer für sende
   Kontrollregister
//Richtung CW/CCW
volatile uint8_t fpga_cw_drive;
volatile uint8_t fpga_cw_drive_old;
volatile uint8_t fpga_ccw_drive;
volatile uint8_t fpga_ccw_drive_old;
//Autodrive
volatile uint8_t fpga_cw_autodrive;
volatile uint8_t fpga_ccw_autodrive;
volatile uint32_t fpga_autodrive_steps;
//Single Step
volatile uint8_t fpga_single_step_cw;
volatile uint8_t fpga_single_step_ccw;
volatile int8_t fpga_single_step_ctr;
volatile int16_t fpga_single_step_factor;
//Last Pos. Dirve
volatile uint8_t fpga_cw_last_pos_drive;
volatile uint8_t fpga_ccw_last_pos_drive;

volatile uint8_t fpga_steps_left;//Noch Schritte im Register Flag
volatile int32_t fpga_pos_1;//Speicher aktuelle Position
volatile int32_t fpga_pos_2;//Speicher letzte Position
/*Leds*/
volatile uint8_t LED_ctr_high;//Zähler wenn Endlag act. high
volatile uint8_t LED_ctr_dis;//Zähler wenn Endlage deaktivert
/*EEPROM*/
volatile uint8_t eeprom_save_ctr;
volatile uint8_t eeprom_save;
/*Sonstige*/
volatile uint8_t startup_ctr;
#endif
```

C.2. encoder.h

```
/*********************************************************
 * Firma: DESY Hamburg
 * Autor: Joshua Supra
 * Projekt: ZMX-Handsteuerung Serie 3
 * Datei: encoder.h
 * Letzte Aenderung: 30.08.2018
 * Kurzbeschreibung: Initialiserung fuer Drehencoder Menue und
   Frqueunzeinstellung
 * (c) Copyright 2018 DESY
*********************************************************/

#ifndef ENCODER_H_
#define ENCODER_H_

#define     SPEED   TIM1
#define     MENU    TIM8

#define     ENCODER_PERIOD  0xFF
/*********************************************************
 * Prototyp Funktionen
*********************************************************/
/*Initialisierung GPIOs */
void init_GPIO_encoder(void);
void init_menu_encoder(void);
/*Initialisierung Timer */
void init_adjust_encoder(void);
void Timer6_Init(void);
/*********************************************************
 * Encoder Variablen
*********************************************************/
/*Auswertung Menue*/
volatile int16_t encoder_menu_value;
volatile int16_t encoder_menu_value_old;
/*Auswertung Frequenz*/
volatile int16_t encoder_speed_value;
volatile int16_t encoder_speed_value_old;

#endif
```

C.3. ea_dip203B.h

```
/*********************************************************
Firma: DESY Hamburg
```



```
Autor: Joshua Supra
Projekt: ZMX-Handsteuerung Serie 3
Datei: ea_dip203B.h
Letzte Aenderung: 30.08.2018
Kurzbeschreibung: Initialiserung fuer das Display, Funktionen für
   das Senden
und Aufbereiten von Daten an das Display
(c) Copyright 2018 DESY
*********************************************************/
#ifndef EA_DIP203B_H_
#define EA_DIP203B_H_

/*Chip Select SPI Display*/
#define     CS_DISP_LOW         GPIO_ResetBits(GPIOA, GPIO_Pin_4)
#define     CS_DISP_HIGH        GPIO_SetBits(GPIOA, GPIO_Pin_4)

/*Befehle fuer das Display. Siehe Manual*/
#define     START_INSTRUC       0x1F //Start Byte: Anweisung
#define     START_WRITE         0x5F //Start Byte: Schreibe
   Display

#define     EXTENSION_REGISTER_OFF      0x30
#define     INIT_ENTRY_MD_SET           0x06
#define     EXTENSION_REGISTER_ON       0x36
#define     INIT_4_LINES_SET            0x09
#define     SEGRAM_ADR_0                0x40
#define     EMPTY_CHARACTER             0x00
#define     DISPLAY_ON                  0x0C
#define     CLEAR_DISPLAY               0x01

/*Display Buffer Einstellungen*/
#define     DISPLAY_BUFFER_SIZE 30//Buffer Groesse
#define     STRING_SIZE         20//Buffer Datenlaenge
/*********************************************************
Prototyp Funktionen
*********************************************************/
/*Initialisierung Display*/
void init_display_SPI1(void);
void init_display(void);
void Timer5_Init(void);
void Timer3_Init(void);

/*Senden an das Display*/
void Write_Display(unsigned char data, uint16_t mode);
```

```
void display_puts(int y, int x,  char *data);
void display_putc(int y, int x,  uint8_t data);
void display_puts_frame(uint8_t y, uint8_t x, uint32_t value, char
    *data, uint8_t max_digits);
void display_puts_frame_s(uint8_t y, uint8_t x, uint32_t value,
   char *data, uint8_t max_digits);
void display_put_ASCII_Current(uint8_t y, uint8_t x, unsigned int
   Wert);
void Write_Display_CG(unsigned char addr, char *zeichen);
/*sonstige Funktionen*/
void SPI1_send(unsigned char data);
void User_symbol(char *zeichen);
int convert_addr(int y, int x);
void ItoA_point(uint16_t u, char* Buffer);
void blink_cursor(char *data, uint8_t y, uint8_t x);
/*********************************************************
Variablen
*********************************************************/
char databuffer[DISPLAY_BUFFER_SIZE][STRING_SIZE];//Datenbuffer
   fuer SPI Kommunikation zum Display
uint8_t old_x;//Cursor x-Koordinate Zwischenspeicher
uint8_t old_y;//Cursor y-Koordinate Zwischenspeicher
#endif
```

C.4. various.h

```
/*********************************************************
Firma: DESY Hamburg
Autor: Joshua Supra
Projekt: ZMX-Handsteuerung Serie 3
Datei: various.h
Letzte Aenderung: 30.08.2018
Kurzbeschreibung: Diverse Funktionen
(c) Copyright 2018 DESY
*********************************************************/
#ifndef VARIOUS_H_
#define VARIOUS_H_
/*********************************************************
Prototyp Funktionen
*********************************************************/
/*UART5 für Debug mit PC*/
void uart5_putc(unsigned char c);
void uart5_puts(const char *s);
```



```
void UART5_Init(void);
void gpio_UART5_Init(void);
void debug(char* strn, int value);//debug mit Putty

/*Blink LEDs, Startup*/
void Timer2_Init(void);

/*Wandlung */
char ItoA_unsigned(uint32_t u, char* Buffer);
char ItoA_signed(int32_t u, char* Buffer);
char ASCII_Spiegeln(char* Buffer,int i);
char HtoA(char HEX_Wert);

int32_t Grenzwert(int32_t Wert,int32_t min,int32_t max);
#endif
```

C.5. spi_fpga.h

```
/*********************************************************
Firma: DESY Hamburg
Autor: Joshua Supra
Projekt: ZMX-Handsteuerung Serie 3
Datei: spi_fpga.h
Letzte Aenderung: 30.08.2018
Kurzbeschreibung: Initialiserung der Kommunikation mit dem FPGA
   über SPI,
Funktionen zum Senden und Empfangen von Daten
(c) Copyright 2018 DESY
*********************************************************/
#ifndef COM_FPGA_H_
#define COM_FPGA_H_

#define     DDS_K       42.94967296  //DDS Faktor

#define     CS_FPGA_LOW         GPIO_ResetBits(GPIOB, GPIO_Pin_12)
#define     CS_FPGA_HIGH        GPIO_SetBits(GPIOB, GPIO_Pin_12)

/*Zugriffsmodi*/
#define     R                   0x0
#define     W                   0x1

/*Registeradressen*/
#define     FPGA_ENCODER_REG    0x1
```

```
#define     FPGA_STEP_MODE_REG  0x2
#define     FPGA_DIR_STEP_REG   0x3
#define     FPGA_DDS_REG        0x4
#define     FPGA_STEP_CTR_REG   0x5

/*Endlageneinstellungen*/
#define     FPGA_LIMIT_LOW      0x1
#define     FPGA_LIMIT_HIGH     0x2
#define     FPGA_LIMIT_DIS      0x3
#define     FPGA_LIMITS_RESET   0x0

/*Laufrichtung*/
#define     FPGA_RUN_CW         0x80000000
#define     FPGA_RUN_CCW        0x00000001

/*Laufmodi*/
#define     FPGA_MODE_RES       0x0
#define     FPGA_MODE_CONT      0x1
#define     FPGA_MODE_STEP      0x2
/*********************************************************
 * Prototyp Funktionen
*********************************************************/
void init_fpga_SPI2(void);
void Timer4_Init(void);
uint8_t SPI2_send(unsigned char data);
int32_t send_FPGA(uint8_t rw, uint8_t address ,uint32_t data);
uint32_t calc_DDS_count(uint32_t f);
#endif
```

C.6. I2C.h

```
/*********************************************************
Firma: DESY Hamburg
Autor: Joshua Supra
Projekt: ZMX-Handsteuerung Serie 3
Datei: I2C.h
Letzte Aenderung: 30.08.2018
Kurzbeschreibung: Initialiserung fuer das digitale Potentiometer
   zum Einstellen
des Kontrasts vom Display
(c) Copyright 2018 DESY
*********************************************************/
#ifndef I2C_H_
```



```
#define I2C_H_
/*********************************************************
Prototyp Funktionen
*********************************************************/
void init_I2C(void);
void Set_Display_contrast(uint8_t wert);
void I2C_Write(uint8_t data);

#endif
```

C.7. gpio.h

```
/*********************************************************
Firma: DESY Hamburg
Autor: Joshua Supra
Projekt: ZMX-Handsteuerung Serie 3
Datei: gpio.h
Letzte Aenderung: 30.08.2018
Kurzbeschreibung: Initialiserung der GPIOs für Taster und LEDs
(c) Copyright 2018 DESY
*********************************************************/
#ifndef DISPLAY_H_
#define DISPLAY_H_

/*---------------  Button define ---------------*/
#define     BUT_MENU            GPIO_ReadInputDataBit(GPIOD,
   GPIO_Pin_8)
#define     BUT_P_RESET         GPIO_ReadInputDataBit(GPIOD,
   GPIO_Pin_9)
#define     BUT_ENTREGEN        GPIO_ReadInputDataBit(GPIOD,
   GPIO_Pin_10)
#define     BUT_CW              GPIO_ReadInputDataBit(GPIOD,
   GPIO_Pin_11)
#define     BUT_CCW             GPIO_ReadInputDataBit(GPIOD,
   GPIO_Pin_12)
#define     BUT_STOP            GPIO_ReadInputDataBit(GPIOD,
   GPIO_Pin_13)
#define     BUT_RES1            GPIO_ReadInputDataBit(GPIOD,
   GPIO_Pin_14)

#define     BUT_ENC1            GPIO_ReadInputDataBit(GPIOC,
   GPIO_Pin_5)
```

```
#define     BUT_ENC2            GPIO_ReadInputDataBit(GPIOE,
   GPIO_Pin_12)

/*---------------  LED define ---------------*/
#define     LED_CW_OFF()          GPIO_ResetBits(GPIOA, GPIO_Pin_8
   )
#define     LED_CW_ON()           GPIO_SetBits(GPIOA, GPIO_Pin_8)
#define     LED_CCW_OFF()         GPIO_ResetBits(GPIOA,
   GPIO_Pin_10)
#define     LED_CCW_ON()          GPIO_SetBits(GPIOA, GPIO_Pin_10)
#define     LED_ZMX_R_ON()        GPIO_SetBits(GPIOC, GPIO_Pin_8)
#define     LED_ZMX_R_OFF()       GPIO_ResetBits(GPIOC, GPIO_Pin_8
   )
#define     LED_ZMX_G_ON()        GPIO_SetBits(GPIOC, GPIO_Pin_9)
#define     LED_ZMX_G_OFF()       GPIO_ResetBits(GPIOC, GPIO_Pin_9
   )
/*********************************************************
Prototyp Funktionen
*********************************************************/
void init_GPIOS(void);

#endif
```

C.8. ZMX.h

```
/*********************************************************
Firma: DESY Hamburg
Autor: Joshua Supra
Projekt: ZMX-Handsteuerung Serie 3
Datei: ZMX.h
Letzte Aenderung: 30.08.2018
Kurzbeschreibung: Initialiserung der Kommunikation mit der ZMX-
   Endstufe über
UART, Funktionen zum Senden und Empfangen von Daten
(c) Copyright 2018 DESY
*********************************************************/
#ifndef _ZMX_H_
#define _ZMX_H_

#define ZMX_MAX_DATA  97
/*********************************************************
Prototyp Funktionen
*********************************************************/
```



```
void init_uart_ZMX(void);
void send_ZMX(char *command, uint16_t value);
void uart4_putc(unsigned char data);
void uart4_puts(char *data);
void ZMX_get_address(void);
/*********************************************************
Variablen
*********************************************************/
uint8_t ZMX_Adresse_L;//Low Byte ZMX Adresse
uint8_t ZMX_Adresse_H;//High Byte ZMX Adresse

char ZMX_Daten_Empfangen[20];// Empfangsbuffer
volatile uint8_t ZMX_Datensatz_Empfangen;//Flag
volatile uint8_t ZMX_Daten_gesendet;//Flag

#endif
```

D. Quellcode der Software des Mikrocontrollers: Source Dateien

Der Anhang C enthält die Source Dateien der Software des Mikrocontrollers. Am Anfang jeder Datei befindet sich eine Kurzbeschreibung, die einen Überblick über den Inhalt bietet.

D.1. main.c

```
/***********************************************************
 * Firma: DESY Hamburg
 * Autor: Joshua Supra
 * Projekt: ZMX-Handsteuerung Serie 3
 * Datei: main.c
 * Letzte Aenderung: 30.08.2018
 * Kurzbeschreibung: Hauptprogramm: Auswertung der Eingabeelemente
    , Ansteuerung
 *      LEDs, Visualisierung der Parameter im Display,
    Parametrierung der
 *      ZMX-Endstufe und Parametrierung des FPGAs.
 * (c) Copyright 2018 DESY
***********************************************************/
#include "stm32f4xx_conf.h"
#include "main.h"
#include "encoder.h"
#include "various.h"
#include "I2C.h"
#include "ea_dip203B.h"
#include "spi_fpga.h"
#include "gpio.h"
#include "ZMX.h"
#include "eeprom.h"
/***********************************************************
* EEPROM Arrays
***********************************************************/
```



```
/*Adress-Array, virtuelle EEPROM Adressen*/
uint16_t eeprom_addr[NB_OF_VAR] = {0x0000, 0x0001, 0x0002, 0x0003,
    0x0004, 0x0005};
/*Daten-Array zum lesen und schreiben in das EEPROM*/
uint16_t eeprom_data[NB_OF_VAR] = {0};

int main(void)
{
/***********************************************************
* Initialisierung Peripherie und Mikrocontroller
***********************************************************/
    SystemInit();//CPU Clock, PLLs etc.
    UART5_Init();
    gpio_UART5_Init();
    init_I2C();//Kommunikation Potentiometer f. Display Kontrast
    Set_Display_contrast(14);//Lege Display
    init_GPIO_encoder();//GPIOs für die Drehencoder
    init_menu_encoder();// Menü Drehencoder
    init_adjust_encoder();// Frequenz Drehencoder
    init_display_SPI1();//SPI Display
    init_uart_ZMX();//UART ZMX-Endstufe
    init_GPIOS();// GPIOs (Taster, LEDs)
    init_fpga_SPI2();//SPI FPGA
    Timer3_Init();//Interrupt zum senden des Buffers an das
       Display
    Timer6_Init();//Init. Timer 6: Auswertung Drehencoder
    FLASH_Unlock();//Flash zugriff
    EE_Init();//EEPROM im Flash speicher
/***********************************************************
* Variablen und Flags
***********************************************************/
    /*Taster Auswertung*/
    uint8_t button_flag=0;//Taster betätigt flag
    uint8_t button_cw_flag=0;//Taster cw betätigt flag
    uint8_t button_ccw_flag= 0;//Taster ccw betätigt flag

    /*Stop betätigt*/
    uint8_t fpga_stop=0;
    uint8_t fpga_stop_old=0;
    /*Position Reset betätigt*/
    uint8_t fpga_pos_reset=0;
    uint8_t fpga_pos_reset_old=0;

    /*Last Pos drive betätigt*/
```

```
    uint8_t fpga_last_pos_drive=0;
    uint8_t fpga_last_pos_drive_old=0;
    uint8_t fpga_last_pos_drive_flag=0;

    /*Drehencoder Speed Taster betätigt*/
    uint8_t enc2_button=0;
    uint8_t enc2_button_old=0;

    /*Zustand Endlage alt, nicht ausgelöst*/
    uint8_t cw_limit_old=0;
    uint8_t ccw_limit_old=0;

    /*Autodrive und single Step Schritte/Faktor*/
    uint8_t fpga_single_step_factor_old=0;
    uint16_t fpga_autodrive_steps_old=0;


    uint8_t fpga_steps_left_old=0;//Schritte im Step Register flag

    uint8_t eeprom_checksum=0;//Checksum für EEPROM
/*********************************************************
* Initial-Werte für globale Variablen
*********************************************************/
    /*Menü*/
    menu_state= 0;//Menü ist deaktiviert
    menu_state_old=0;
    menu_select=0;//Kein Parameter ausgewählt
    menu_print_select=0;////Kein Parameter aktiv

    /*Update Flags, Änderung der Anzeige im Display wenn update_xx
       ==1 */
    update_Irun=0;//Laufstrom
    update_Istop=0;//Stoppstrom
    update_res=0;//Schrittauflösung
    update_cw=0;//Endlage CW
    update_ccw=0;//Endlage CCW
    update_enc_speed=0;//Impulsfrequenz
    update_single_step=0;//Single Step Faktor
    update_autodrive=0;//Autodrive Faktor

    /*Fahre ccw bis Endlage deaktiviert*/
    fpga_ccw_drive=0;
    fpga_ccw_drive_old=0;
```



```
    /*Fahre cw bis Endlage deaktiviert*/
    fpga_cw_drive=0;
    fpga_cw_drive_old=0;

    /*Autodrive cw/ccw deaktiviert*/
    fpga_cw_autodrive=0;
    fpga_ccw_autodrive=0;

    /*Fahre zu letzter Position deaktiviert*/
    fpga_cw_last_pos_drive=0;
    fpga_ccw_last_pos_drive=0;

    /*EEPROM Speicher Flags deaktiviert*/
    eeprom_save=0;
    eeprom_save_ctr=121;

    /*FPGA default werde beim Start*/
    zmx_reset = 0;//kein Reset
    zmx_entregen=1;//Endstufe deaktivert
    zmx_entregen_old = 1;//Endstufe deaktivert
    fpga_limit_cw_old=0;// Endlage CW low aktiv
    fpga_limit_ccw_old=0;//Endlage CCW low aktiv
    fpga_ctrl_mode =FPGA_MODE_RES;// Impulserzeugung deaktivert
    fpga_single_step_factor = 1;//Single Step Faktor 1x
    fpga_pos_1=0;//Letzte Position =0
    fpga_pos_2=0;//Letzte Position =0
    fpga_steps_left=0;////Schritte im Step Register flag

    /*Initialiserung Drehencoder default Werte*/
    encoder_menu_value=0;
    encoder_menu_value_old=1;
    encoder_speed_value=100;//100Hz Impulsfrequenz
    encoder_speed_value_old=0;

    /*Zeige Encoder im Display an*/
    fpga_steps_enc=0;
    fpga_steps_enc_old=0;

    /*Rücksetzte single Step und Autodrive*/
    fpga_single_step_ctr=0;
    fpga_autodrive_steps=0;
    fpga_single_step_cw=0;
    fpga_single_step_ccw=0;
```

```
    fpga_update = 1;//Update FPGA

    /*Vorherige Cursorposition im Display*/
    old_x = 6;
    old_y = 1;

    /*Endstufenparameter*/
    Irun_old=1;//Laufstrom 0,1A
    Istop_old=0;//Stoppstrom 0,0A
    res_old=1;//Schrittauflösung 1/2
/************************************************************
* Lade gespeicherte Parameter aus dem EEPROM
************************************************************/
/*Lade Werte aus EEPROM und schreibe in Daten Array*/
for(uint8_t eeprom_counter=0; eeprom_counter < NB_OF_VAR;
   eeprom_counter++){
        EE_ReadVariable(eeprom_addr[eeprom_counter], &eeprom_data[
           eeprom_counter]);
}
/*XOR alle Werte*/
eeprom_checksum = eeprom_data[0]^eeprom_data[1]^eeprom_data[2]^
   eeprom_data[3]^eeprom_data[4]^eeprom_data[5];
if(eeprom_checksum!= 0){//Wenn der Speicher nicht leer ist Lade
    Irun = eeprom_data[0];//Laufstrom
    Istop = eeprom_data[1];//Stoppstrom
    res = eeprom_data[2];//Schrittauflösung
    fpga_limit_ccw = eeprom_data[3];//Endlageneinstellung CCW
    fpga_limit_cw = eeprom_data[4];//Endlageneinstellung CW
    encoder_speed_value = eeprom_data[5];//Impulsfrequenz
/*Wenn der Speicher Leer ist Schreibe Default Werte*/
}else{
    Irun = 5;//Laustrom 0,5A
    Istop = 2;//Stoppstrom 0,2A
    res = 0;//Schrittauflösung 1/1
    fpga_limit_ccw = FPGA_LIMIT_LOW;//Endlage CCW low aktiv
    fpga_limit_cw = FPGA_LIMIT_LOW;//Endlage CW low aktiv
    encoder_speed_value = 50;//Impulsfrequenz 50Hz
    }

    /*Display Transfer*/
    display_tx_busy = 0;//Busy Flag
    display_tx_data = 1;//????????????
    display_tx_data_ctr= 0;//Display Buffer Counter
    display_tx_wait = 0;//Wait flag
```



```
    button_get_state = 0;//Taster Auswertung Flag
    /*Aktiviere LEDs*/
    LED_CW_ON();
    LED_CCW_ON();
    LED_ZMX_G_ON();
    LED_ZMX_R_OFF();

    init_display();//Initiealiere Display

    Timer2_Init();//Init. Timer 2: 250us; LEDs, EEPROM save
    ZMX_Adresse_L=0x30;//ZMX-Low Startadresse
    ZMX_Adresse_H=0x30;//ZMX-High Startadresse
    ZMX_get_address();//Endstufenadresse herausfinden
    startup_ctr=0;
    while(startup_ctr!=4);//warte 2 Sekunden bis Endstufe bereit
    /*Schreibe Parameternamen in Display Buffer*/
    display_puts(1,1, "IRun");
    display_puts(2,1, "IStop");
    display_puts(3,1, "Res");
    display_puts(1,12, "Encoder");
    display_puts(3,12, "Speed");

    Timer5_Init();//Init. Timer 5: Auswertung Eingaben und
       Schreiben in den Display Buffer
    Timer4_Init();//Init. Timer 4: FPGA Auswertung Endlagen, Step/
       Encoder Register, Sende neuen Zustand

/************************************************************
 * Hauptschleife: Auswertung der Taster, Senden von Parameter an
 *      die ZMX-Endstufe, Ansteuerung der LEDs
************************************************************/
while(1)
{
/************************************************************
*  Auswertung der Taster, Interrupt setzt alle 20ms Flag zum
*       auswerten der Taster
************************************************************/
    if(button_get_state==1){//Wenn Flag aktiv
        button_get_state=0;//setze Flag zurück.
        if(!button_flag){//kein Taster gedrückt
            /*Taster Drehencoder Menü*/
            if(!BUT_ENC1){//Wenn der Taster betätigt wird,
```

```
            button_flag=1;
            if(menu_state==0){//und das Menü nicht aktiv ist
                if(fpga_steps_enc==0){//und Encoder angezeigt
                   werden
                    fpga_steps_enc=1;//Wechselt die Anzeige
                       auf Steps.
                }else if(fpga_steps_enc==1){//Wenn Steps
                   andgezeigt werden
                    fpga_steps_enc=0;//wechselt die Anzeige
                       auf Encoder
                }
            }else if(menu_state==1){//Wenn das Menü aktiv ist
                if(menu_select==0){//und kein Parameter
                   ausgewählt ist
                    menu_select=1;//wird der Parameter
                       selektiert.
                    menu_print_select=1;//Flag für Cursor
                       Auswahl anzeigen
                }else if(menu_select==1){//Wenn ein Parameter
                   ausgewählt war
                    menu_select=0;//wird er deselktiert
                    TIM8->CNT=0;//Der Encoder auf 0 gesetzt
                    Blink_ctr=3;//und der Blink counter reset
                }
            }
        }
        /*Taster Drehencoder Speed*/
        if(!BUT_ENC2){//noch keine Funktion
            button_flag=1;
            enc2_button=1;
        }
        /*Taster Menü*/
        if(!BUT_MENU){//Wenn der Taster betätigt wird
            button_flag=1;
            if(menu_state==0){//und das Menü nicht aktiv ist
                menu_state=1;//wird das Menü aktiviert.
                Blink_ctr=3;//Blink counter reset
                TIM8->CNT=0;//Encoder auf 0
            }else if(menu_state==1){//Wenn das Menü aktiv ist
                menu_state=0;//wird das Menü deaktiviert.
                menu_select=0;//Die Parameterauswahl gelöscht
                eeprom_save=1;//Alle Werte im EEPROM
                   gespeichert
```



```
                    update_speed=1;//und die aktuelle
                       Impulsfrequenz angezeigt.
                }
            }
            /*Taster Pos. Reset*/
            if(!BUT_P_RESET){//Wenn der Taster betätigt wird,
                button_flag=1;
                fpga_pos_reset=1;//wird die aktuelle Position
                   zurück gesetzt.
            }
            /*Taster Entregen*/
            if(!BUT_ENTREGEN){//Wenn der Taster betätigt wird,
                button_flag=1;
                if(zmx_entregen==0){//und die ZMX-Endstufe aktiv
                   ist
                    zmx_entregen=1;//wird die ZMX-Endstufe
                       deaktiviert.
                }else if(zmx_entregen==1){//Ist die ZMX-Endstufe
                   deaktivert
                    zmx_entregen=0;//wird sie aktiviert.
                }
            }
            /*Taster Stop*/
            if(!BUT_STOP){//Wenn der Taster betätigt wird,
                button_flag=1;
                fpga_stop=1;//Wird die Impulserzeugung
                   unterbrochen
            }
        }
/***********************************************************
 * Auswertung der Taster CW und CCW wird nur bei aktiver Endstufe
 * ausgeführt
 ***********************************************************/
        if(zmx_entregen_old == 0){//Wenn die ZMX-Endstufe aktiv
           ist:
            /*Taster CW*/
            if(!BUT_CW){//Wenn der Taster betätigt wird
                button_cw_flag=1;//wird die cw Flag gesetzt.
            }
            /*Taster CCW*/
            if(!BUT_CCW){//Wenn der Taster betätigt wird
                button_ccw_flag=1;//wird die ccw Flag gesetzt.
            }
```

```
            /*Auswertung abhängig davon wie die Taster betätigt
               sind*/
            /*CW wurde Betätigt und wieder losgelassen. CCW wurde
               nicht betätigt*/
            if(button_cw_flag == 1 && button_ccw_flag == 0 &&
               BUT_CW){
                fpga_cw_drive=1;//Fahre in Richtung CW
            }
            /*CCW wurde Betätigt und wieder losgelassen. CW wurde
               nicht betätigt*/
            if(button_cw_flag == 0 && button_ccw_flag == 1 &&
               BUT_CCW){
                fpga_ccw_drive=1;//Fahre in Richtung CCW
            }
            /*Wurden beide Taster gleichzeitig betätigt*/
            if(button_cw_flag == 1 && button_ccw_flag == 1 &&
               fpga_last_pos_drive_flag == 0){
                fpga_last_pos_drive_flag = 1;//Setzte flag
                fpga_last_pos_drive=1;//Fahre die letzte position
                   an
            }
        }
        /*Wenn kein Taster betätig sind, werden die flags zurück
           gesetzt.*/
        if(BUT_MENU && BUT_P_RESET && BUT_ENTREGEN && BUT_CW &&
           BUT_CCW && BUT_STOP && BUT_RES1 && BUT_ENC1 && BUT_ENC2
            && (button_flag||button_cw_flag==1||button_ccw_flag==1
            )){//Reset flag if no Button is pressed
            button_flag = 0;//Taster betätigt Flag
            button_cw_flag = 0;//Taster cw betätigt Flag
            button_ccw_flag = 0;//Taster ccw betätigt Flag
            fpga_last_pos_drive_flag = 0;//Fahre letzte Position
               an flag

            //Reset ruecksetzen, damit nicht dauerhaft ausgeführt
               wird.
            fpga_pos_reset = 0;
            fpga_stop = 0;
            }
    }
/************************************************************
 * Auswertung der Eingaben durch die Taster, Variablen können ggf.
 * auch durch andere Mechanismen gesetzt werden (z.B. eine
    Mögliche
```



```
 * zukünftige EtherCAT Anbindung.
*************************************************************/
    /*Reset Encoder/Steps*/
    if(fpga_pos_reset!=fpga_pos_reset_old){//Ändert sich der
       Zustand
        fpga_pos_reset_old=fpga_pos_reset;//wird der neue Zustand
           gespeichert
        if(fpga_pos_reset_old==1){//Wird der Reset gesetzt
            if(fpga_steps_enc_old==0){//und die Darstellung
               Encoder ist gesetzt
                send_FPGA(W, FPGA_ENCODER_REG, 0);//wird das
                   Encoder Register im FPGA auf 0 gesetzt.
            } else{//Ist die Darstellunug Steps aktiv
                send_FPGA(W, FPGA_STEP_CTR_REG, 0);//wird das
                   Schrittregister im FPGA auf 0 gesetzt.
            }
        }
    }
    /*Stoppen der Impulserzeugung*/
    if(fpga_stop!=fpga_stop_old){//Ändert sich der Zustand
        fpga_stop_old=fpga_stop;//wird der neue Zustand übernommen
        if(fpga_stop_old==1){//Wird Stop gesetzt,
            fpga_ctrl_mode = FPGA_MODE_RES;//wird das
               Kontrollregister im FPGA resetet
            fpga_cw_drive=0;//CW drive beendet
            fpga_ccw_drive=0;//CCW drive beendet
            fpga_last_pos_drive=0;//das Anfahren der letzten
               Position beendet
            fpga_update=1;//und das Kontrollregister refresht
        }
    }
    /*Änderung des ZMX Zustands aktiviert/deaktiviert und LED
       anzeige*/
    if(zmx_entregen!=zmx_entregen_old){//Ändert sich der Zustand
        zmx_entregen_old=zmx_entregen;//wird der neue Zustand
           übernommen
        fpga_update=1;//und an den FPGA übertragen.
        if(zmx_entregen_old==1){//Ist der Zustand deaktiviert
            fpga_ctrl_mode = FPGA_MODE_RES;//erhält das
               Kontrollregister einen Reset Befehl
            LED_ZMX_G_ON();//Die Led für den Status der Endstufe
            LED_ZMX_R_OFF();//wird grün
            fpga_update=1;//der Reset befehl wird mit der nächsten
                Übertragung zum FPGA ausgeführt.
```

```
            fpga_cw_drive=0;//CW drive
            fpga_ccw_drive=0;// und CCW drive werden unterbrochen.
        }else{//ist die ZMX-Endstufe aktiv
            LED_ZMX_G_OFF();//wird die Led für den Status der
               Endstufe
            LED_ZMX_R_ON();//rot
        }
    }
    /*Schritte im Step Register im FPGA zum Rücksetzen von
       Autodrive und Last Pos. drive*/
    if(fpga_steps_left!= fpga_steps_left_old){//Ändert sich der
       Zustand
        fpga_steps_left_old=fpga_steps_left;//wird der neue
           Zustand übernommen
        if((fpga_cw_drive==1 ||fpga_ccw_drive ==1 ||
           fpga_last_pos_drive==1)&& fpga_steps_left_old==0){//
           Autodrive oder Last pos drive aktiv und keine Schritte
           mehr übrig
            fpga_cw_drive=0;//wird Autodrive cw,
            fpga_ccw_drive=0;//Autodrive cc
            fpga_last_pos_drive=0;//und Last Pos. Drive zurück
               gesetzt.
        }
    }
    /*Einstellung Laufmodi. Autodrive oder kontinuierlich bis
       Endlage CW*/
    if(fpga_cw_drive_old!=fpga_cw_drive && zmx_entregen_old == 0){
       //wird der Befehl CW fahren gesetzt oder rückgesetzt und
       die Endstufe ist aktiv,
        fpga_cw_drive_old=fpga_cw_drive;//wird der neue Zustand
           übernommen.
        if(fpga_cw_drive_old==1){//Soll in Richtung CW gefahren
           werden,
            fpga_cw_drive=1;//wird die CW drive Flag gesetzt
            fpga_ccw_drive=0;//und die CCW Flag zurück gesetzt
            if(fpga_autodrive_steps==0){//Wenn Autodrive steps 0
               ist
                send_FPGA(W, FPGA_DIR_STEP_REG, FPGA_RUN_CW);//
                   wird die Drehrichtung CW gesetzt
                fpga_ctrl_mode = FPGA_MODE_CONT;// und der
                   kontinuierliche Modus aktiviert
            }else{//Ist eine Anzahl an Schritte vorgegeben
                fpga_cw_autodrive=1;//wird Autodrive Flag Richtung
                    CW gesetzt
```



```
                fpga_ctrl_mode=FPGA_MODE_STEP;//und das Step Modus
                    im FPGA aktivert.
            }
            fpga_update=1;//Refresh FPGA
            update_speed=1;//Im Display werden die aktuelle
               Impulsfrequenz
            update_enc_speed=1;// und der Parametername angezeigt
        }else if(fpga_cw_drive_old==0){//Soll nicht mehr in die
           Richtung CW gefahren werden
            if(menu_state == 1){//und das Menü ist aktiv
                menu_state_old = 0;//wird das Menü overlay im
                   Display angezeigt.
            }
        }
    }
    /*Einstellung Laufmodi. Autodrive oder kontinuierlich bis
       Endlage CCW*/
    if(fpga_ccw_drive_old!=fpga_ccw_drive && zmx_entregen_old ==
       0){//wird der Befehl CCW fahren gesetzt oder rückgesetzt
       und die Endstufe ist aktiv,
        fpga_ccw_drive_old=fpga_ccw_drive;//wird der neue Zustand
           übernommen.
        if(fpga_ccw_drive_old==1){//Soll in Richtung CCW gefahren
           werden,
            fpga_ccw_drive=1;//wird die CCW drive Flag gesetzt
            fpga_cw_drive=0;//und die CW Flag zurück gesetzt
            if(fpga_autodrive_steps==0){//Wenn Autodrive steps 0
               ist
                send_FPGA(W, FPGA_DIR_STEP_REG, FPGA_RUN_CCW);//
                   wird die Drehrichtung CCW gesetzt
                fpga_ctrl_mode = FPGA_MODE_CONT;// und der
                   kontinuierliche Modus aktiviert
            }else{//Ist eine Anzahl an Schritte vorgegeben
                fpga_ccw_autodrive=1;//wird Autodrive Flag
                   Richtung CCW gesetzt
                fpga_ctrl_mode=FPGA_MODE_STEP;//und das Step Modus
                    im FPGA aktivert.
            }
            fpga_update=1;//Refresh FPGA
            update_speed=1;//Im Display werden die aktuelle
               Impulsfrequenz
            update_enc_speed=1;
        }else if(fpga_ccw_drive_old==0){//Soll nicht mehr in die
           Richtung CCW gefahren werden
```

```
            if(menu_state == 1){//und das Menü ist aktiv
                menu_state_old = 0;//wird das Menü overlay im
                   Display angezeigt.
            }
        }
    }
    /*Hin und her fahren zwischen den letzten beiden Positionen*/
    if(fpga_last_pos_drive!=fpga_last_pos_drive_old){//Wird Last
       pos. Drive aktiviert oder deaktiviert
        fpga_last_pos_drive_old=fpga_last_pos_drive;//wird der
           neue Zustand übernommen.
        if(fpga_last_pos_drive_old==1){//Wenn die letzte Position
           angefahren werden soll
            if(fpga_pos_1<fpga_pos_2){//und die neue Position
               kleiner als die alte Position ist,
                fpga_cw_last_pos_drive=1;//soll in Richtung CW
                   gefahren werden.
                fpga_ctrl_mode=FPGA_MODE_STEP;//Step modus wird
                   gesetzt
            }
            if(fpga_pos_1>fpga_pos_2){//Wenn die neue Position
               größer ist als die alte Position
                fpga_ccw_last_pos_drive=1;//soll in Richtung CCW
                   gefahren werden.
                fpga_ctrl_mode=FPGA_MODE_STEP;//Step modus wird
                   gesetzt
            }
            fpga_update=1;//update FPGA
            update_speed=1;//Update Frequenzanzeige
            update_enc_speed=1;//Update Parameteranzeige
        }
    }
/************************************************************
 * Reaktion auf Änderung von Endstufenparametern: Auswertung und
 * Anzeige
***********************************************************/
    /*Änderung Laufstrom*/
    if(Irun!=Irun_old){//Wird der Laufstom geändert
        Irun_old=Irun;//wird der neue Wert übernommen.
        send_ZMX("R", Irun_old*10);//Sende den neuen Wert an die
           ZMX-Endstufe
        if(Irun_old<Istop_old){//Ist der Laufstrom kleiner als der
            Eingstellte Stoppstrom
```



```
            Istop=Irun_old;//wird der Stoppstrom = Laufstrom
               gesetzt.
        }
        if(Istop_old==-1){//Ist der Stoppstrom auf Auto
           eingestellt,
            send_ZMX("S", (Irun_old>>1)*10);//berechne den Wert
               und Sende den neuen Wert an die ZMX-Endstufe
        }
        update_Irun=1;//update Display Laufstrom
    }
    /*Änderung Stoppstrom*/
    if(Istop!=Istop_old){//Wird der Stoppstrom geändert
        Istop_old=Istop;//wird der neue Wert übernommen.
        update_Istop=1;//update Display Stoppstrom
        send_ZMX("S", Istop_old*10);//Sende den neuen Wert an die
           ZMX-Endstufe
    }
    /*Änderung Schrittauflösung*/
    if(res!=res_old){//Wird die Schrittauflösung geändert
        res_old=res;//wird der neue Wert übernommen.
        send_ZMX("M", res_old);//Sende den neuen Wert an die ZMX-
           Endstufe
        //Für den Displaycursor muss die Position angepasst werden
           , da der Cursor immer vor dem Wert stehen soll.
        if(res_old ==2 || res_old > 10){//Notwendig, da die Werte
           unterschiedlich lange Strings sind
            old_x=5;
        }else if(res_old > 5){
            old_x=6;
        }else{
            old_x=7;
        }
        update_res=1;//update Display Schrittauflösung
    }
    /*CW Endlage Einstellung*/
    if(fpga_limit_cw!=fpga_limit_cw_old){//Änderung der
       Endlageneinstellung CW
        fpga_limit_cw_old=fpga_limit_cw;//wird der neue Zustand
           übernommen.
        update_cw=1;//update Display CW
        fpga_update=1;//update FPGA
    }
    /*CCW Endlage Einstellung*/
```

```
    if(fpga_limit_ccw!=fpga_limit_ccw_old){//Änderung der
       Endlageneinstellung CCW
        fpga_limit_ccw_old=fpga_limit_ccw;//wird der neue Zustand
           übernommen.
        update_ccw=1;;//update Display CCW
        fpga_update=1;//update FPGA
    }
    /*Zustand Endlage CW*/
    if(CW_Limit!=cw_limit_old){//Änderung Zustand der Endlage CW
        cw_limit_old=CW_Limit;//wird der neue Zustand übernommen.
        update_cw=1;//update Display CW
    }
    /*Zustand Endlage CW*/
    if(CCW_Limit!=ccw_limit_old){//Änderung Zustand der Endlage
       CCW
        ccw_limit_old=CCW_Limit;//wird der neue Zustand übernommen
           .
        update_ccw=1;//update Display CCW
    }
    /*Single Step Faktor*/
    if(fpga_single_step_factor!=fpga_single_step_factor_old){//
       Änderung am Single Step Faktor?
        fpga_single_step_factor_old=fpga_single_step_factor;//wird
            der neue Zustand übernommen.
        update_single_step=1;//update Display single Step
    }
    /*Autodrive Faktor*/
    if(fpga_autodrive_steps!=fpga_autodrive_steps_old){//Änderung
       an den Autodrive Schritten
        fpga_autodrive_steps_old=fpga_autodrive_steps;//wird der
           neue Zustand übernommen.
        update_autodrive=1;//Update Display Autodrive Schritte
    }
    //Wenn die Endstufe nicht aktiv ist, darf der Menüencoder
       keine Änderung herovorrufen.
    if(zmx_entregen==1 && menu_state==0){
        TIM8->CNT=0;
    }
/************************************************************
 * Auswertung des Zustands der Endlangen CW und CCW (FPGA) mit LED
     Anzeige
************************************************************/
  /*CW Endlage*/
```



```
  switch(fpga_limit_cw){//Abhängig von der Einstellung der Endlage
     , wird auf den Zustand reagiert
        /*Endlage CW active high*/
        case FPGA_LIMIT_HIGH:   if(CW_Limit){//Hat die Endlage
           ausgelöst
                                    fpga_cw_drive = 0;//wird die
                                       Impulserzeugung gestoppt.
                                    /*LED rot: 0,5s aus 1,5s an*/
                                    if(LED_ctr_high < 2){
                                        LED_CW_OFF();
                                    }else if(LED_ctr_high > 0 &&
                                       LED_ctr_high < 8){
                                        LED_CW_ON();
                                    }else{
                                        LED_ctr_high = 0;
                                    }
                                }else{//Im Normalzustand
                                    /*/*LED rot: 1,5s aus 0,5s an
                                       */
                                    if(LED_ctr_high < 2){
                                        LED_CW_ON();
                                    }else if(LED_ctr_high > 0 &&
                                       LED_ctr_high < 8){
                                        LED_CW_OFF();
                                    }else{
                                        LED_ctr_high = 0;
                                    }
                                }break;
        /*Endlage CW active low*/
        case FPGA_LIMIT_LOW:    if(CW_Limit){//Hat die Endlage
           ausgelöst
                                    fpga_cw_drive = 0;//wird die
                                       Impulserzeugung gestoppt.
                                    LED_CW_ON();//LED leuchtet
                                       dauerhaft rot
                                }else{//Im normalzustand
                                    LED_CW_OFF();//Dauerhaft aus
                                }break;
        /*Endlage CW deaktivert: LED 1s rot 1s aus*/
        case FPGA_LIMIT_DIS:    if(LED_ctr_dis < 6){
                                    LED_CW_ON();
                                }else if(LED_ctr_dis > 4 &&
                                   LED_ctr_dis < 12){
                                    LED_CW_OFF();
```

```
                                }else{
                                    LED_ctr_dis = 0;
                                }break;
    }
    /*CCW Endlage*/
    switch(fpga_limit_ccw){//Abhängig von der Einstellung der
       Endlage, wird auf den Zustand reagiert
        /*Endlage CCW active high*/
        case FPGA_LIMIT_HIGH:   if(CCW_Limit){//Hat die Endlage
           ausgelöst
                                    fpga_ccw_drive= 0;//wird die
                                       Impulserzeugung gestoppt.
                                     /*LED rot: 0,5s aus 1,5s an*/
                                    if(LED_ctr_high < 2){
                                        LED_CCW_OFF();
                                    }else if(LED_ctr_high > 0 &&
                                       LED_ctr_high < 8){
                                        LED_CCW_ON();
                                    }else{
                                        LED_ctr_high = 0;
                                    }
                                }else{//Im Normalzustand
                                    /*/*LED rot: 1,5s aus 0,5s an
                                       */
                                    if(LED_ctr_high < 2){
                                        LED_CCW_ON();
                                    }else if(LED_ctr_high > 0 &&
                                       LED_ctr_high < 8){
                                        LED_CCW_OFF();
                                    }else{
                                        LED_ctr_high = 0;
                                    }
                                } break;
        /*Endlage CCW active low*/
        case FPGA_LIMIT_LOW:    if(CCW_Limit){//Hat die Endlage
           ausgelöst
                                    fpga_ccw_drive = 0;//wird die
                                       Impulserzeugung gestoppt.
                                    LED_CCW_ON();//LED leuchtet
                                       dauerhaft rot
                                }else{//Im normalzustand
                                    LED_CCW_OFF();//Dauerhaft aus
                                }break;
        /*Endlage CCW deaktivert: LED 1s rot 1s aus*/
```



```
        case FPGA_LIMIT_DIS:    if(LED_ctr_dis < 6){
                                    LED_CCW_ON();
                                }else if(LED_ctr_dis > 4 &&
                                   LED_ctr_dis < 12){
                                    LED_CCW_OFF();
                                }else{
                                    LED_ctr_dis = 0;
                                }break;
    }
}
}
/***************************************************************
 * TIMER 5 Interrupt-Handler 20ms Schreibe Daten in den Display
    Buffer
**************************************************************/
void TIM5_IRQHandler(){
    if(TIM_GetITStatus(TIM5, TIM_IT_Update)){
        TIM_ClearITPendingBit(TIM5, TIM_IT_Update);//clear
           interrupt flag
        button_get_state = 1;//Taster auswerte Flag
        /*
           ***********************************************************

         * Anzeige der Endstufenparameter im Display
        ***********************************************************
           */
        /*Encoder oder Schritte*/
        if(menu_state==0 || fpga_cw_drive || fpga_ccw_drive){//
           Wenn das Menü nicht aktiv ist, oder der Motor sich
           bewegt
            if(fpga_steps_enc==0){//werden entweder die Encoder
                display_puts_frame_s(2,12,fpga_receive_enc, 0, 9);
            }else{//oder die Schritte angezeigt
                display_puts_frame_s(2,12,fpga_receive_steps, 0,
                   9);
            }

        }
        /*Laufstrom*/
        if(update_Irun==1){//Wenn der Laufstrom geändert wurde
            update_Irun=0;
            display_put_ASCII_Current(1,7, Irun_old);//wird der
               neue Wert im Display angezeigt.
        }
```

```
/*Stoppstrom*/
if(update_Istop==1){//Wenn der Stoppstrom geändert wurde
    update_Istop=0;
    if(Istop_old>=0){//und nicht der Auto modus aktiviert
       ist
        display_put_ASCII_Current(2,7, Istop_old);//wird
           der aktuelle Wert im Display dargestellt.
    }else if(Istop_old==-1){//Ansonsten
        display_puts(2, 7, "Auto");//wird "Auto"
           dargestellt.
    }
}
/*Schrittauflösung*/
if(update_res==1){//Wenn die Schrittauflösung geändert
   wurde
    update_res=0;
    display_putc(old_y,old_x, 0x10);//Pfeil rechts vor den
        Wert
    switch(res_old){//Abhängig vom Wert wird der passende
       Faktor dargestellt:
        case 0:display_puts(3,5, "␣␣␣1/1");break;
        case 1:display_puts(3,5, "␣␣␣1/2");break;
        case 2:display_puts(3,5, "␣1/2,5");break;
        case 3:display_puts(3,5, "␣␣␣1/4");break;
        case 4:display_puts(3,5, "␣␣␣1/5");break;
        case 5:display_puts(3,5, "␣␣␣1/8");break;
        case 6:display_puts(3,5, "␣␣1/10");break;
        case 7:display_puts(3,5, "␣␣1/16");break;
        case 8:display_puts(3,5, "␣␣1/20");break;
        case 9:display_puts(3,5, "␣␣1/32");break;
        case 10:display_puts(3,5, "␣␣1/64");break;
        case 11:display_puts(3,5, "␣1/128");break;
        case 12:display_puts(3,5, "␣1/256");break;
        case 13:display_puts(3,5, "␣1/512");break;
    }
}
/*Endlage CW*/
if(update_cw==1){//Wenn sich der Zustand oder die
   Einstellung der CW Endlage geändert hat
    update_cw=0;
        if(CW_Limit==0){//und die Endlage nicht ausgelöst
           hat
            switch(fpga_limit_cw){//wird entsprechend der
               Einstellung das Symbol dargestellt
```



```
                case FPGA_LIMIT_LOW:display_puts(4,6, "cw"
                   ) ;break;//cw normal
                case FPGA_LIMIT_HIGH: display_putc(4,6, 0
                   x04);display_putc(4,7, 0x05);break;//cw
                    mit Strich
                case FPGA_LIMIT_DIS: display_putc(4,6, 0
                   x06);display_putc(4,7, 0x07);break;//CW
                    durchgestrichen
            }
        }else{//Hat die Endlage ausgelöst
            switch(fpga_limit_cw){//wird entsprechend der
               Einstellung das Symbol dargestellt
                case FPGA_LIMIT_LOW:    display_putc(4,6,
                   0x02);display_putc(4,7, 0x03);break;//
                   cw invertiert
                case FPGA_LIMIT_HIGH: display_putc(4,6, 0
                   x02);display_putc(4,7, 0x03);break;//cw
                    invertiert
                case FPGA_LIMIT_DIS: display_putc(4,6, 0
                   x06);display_putc(4,7, 0x07);break;//CW
                    durchgestrichen
            }
        }

    }
/*Endlage CCW*/
if(update_ccw==1){//Wenn sich der Zustand oder die
   Einstellung der CW Endlage geändert hat
    update_ccw=0;
    if(CCW_Limit==0){//und die Endlage nicht ausgelöst hat
        switch(fpga_limit_ccw){//wird entsprechend der
           Einstellung das Symbol dargestellt
            case FPGA_LIMIT_LOW:display_puts(4,1, "ccw") ;
               break;//ccw normal
            case FPGA_LIMIT_HIGH: display_putc(4,1, 0x04);
               display_putc(4,2, 0x04);display_putc(4,3, 0
               x05);break;//ccw mit Strich
            case FPGA_LIMIT_DIS: display_putc(4,1, 0x06);
               display_putc(4,2, 0x06);display_putc(4,3, 0
               x07);break;//CCW durchgestrichen
        }
    }else{//Hat die Endlage ausgelöst
        switch(fpga_limit_ccw){//wird entsprechend der
           Einstellung das Symbol dargestellt
```

```
            case FPGA_LIMIT_LOW:display_putc(4,1, 0x02);
               display_putc(4,2, 0x02);display_putc(4,3, 0
               x03);break;//ccw invertiert
            case FPGA_LIMIT_HIGH: display_putc(4,1, 0x02);
               display_putc(4,2, 0x02);display_putc(4,3, 0
               x03);break;//ccw invertiert
            case FPGA_LIMIT_DIS: display_putc(4,1, 0x06);
               display_putc(4,2, 0x06);display_putc(4,3, 0
               x07);break;//CCW durchgestrichen
        }
    }
}
/*Darstellung Parametername Steps oder Encoder*/
if(fpga_steps_enc!= fpga_steps_enc_old){//Wenn die
   Darstellung geändert wurde
    fpga_steps_enc_old= fpga_steps_enc;//wird der neue
       Wert übernommen.
    if(fpga_steps_enc_old==0){//und entsprechend
        display_puts(1,12,"Encoder␣␣");//der Parameter "
           Encoder"
    }else if(fpga_steps_enc_old==1){
        display_puts(1,12,"Steps␣␣␣␣");//oder der Paramter
            "Steps" dargestellt.
    }
}
/*Darstellung aktuelle Impulsfrequenz*/
if(update_speed==1 && menu_state==0 || update_speed==1 &&
   (fpga_cw_drive== 1 || fpga_ccw_drive ==1)){
    //Ist das Menü nicht aktiv und eine die
       Geschwindigkeit hat sich geändert, oder ein
       Laufmodus ist aktiv und die Geschw. hat sich
       geändert
    update_speed=0;
    display_puts_frame(4,12,encoder_speed_value_old, "Hz"
       ,9);//wird die aktuelle Frequenz dargestellt
}
/*Darstellung single Step Faktor*/
if(update_single_step==1){//Wenn der Single Step Faktor
   geändert wurde
    update_single_step=0;
    if(menu_state==1 && encoder_menu_value_old == 6){//das
        Menü aktiv ist und der Cursor an der Position 6
       steht
```



```
        display_puts_frame(4,13, fpga_single_step_factor,
           "x",8);//Wird der Wert um eine Stelle nach
           rechts verschoben dargestellt
    }else{//ansonsten
        display_puts_frame(4,12, fpga_single_step_factor,
           "x",9);//wird der Wert eine Stelle nach link
           verschoben dargestellt.
    }
}
/*Darstellung Autodrive Faktor*/
if(update_autodrive==1){//Wenn der Autodrive Faktor
   geändert wurde
    update_autodrive=0;
    if(menu_state==1 && encoder_menu_value_old == 7){//das
        Mneü aktiv ist und der Cursor an der Position 7
       steht
        //Wird der Wert um eine Stelle nach rechts
           verschoben dargestellt
        display_putc(2,13, 0x8C);// mit Plus-Minus Zeichen
        display_puts_frame(2,14, fpga_autodrive_steps,
           0,7);
    }else{//Ansonsten
        //Wird der Wert um eine Stelle nach links
           verschoben dargestellt
        display_putc(2,12, 0x8C);//mit Plus-Minus Zeichen
        display_puts_frame(2,13, fpga_autodrive_steps,
           0,8);
    }
}
/*
   ***********************************************************

 * Menüführung
***********************************************************
   */
/*Blinkender Menücursor*/
if(menu_state==1 && menu_select == 0){//Wenn das Menü
   aktiv ist und keine Parameter selektiert wurde
    switch (encoder_menu_value_old){//wird die Position
       des Cursors druch den Menüencoder festgelegt.
        case 1: blink_cursor(">",1,6);break;//Entsprechend
            des Werts wird der Cursor an der passenden
        case 2: blink_cursor(">",2,6);break;//Position
           dargstellt.
```

```
        //Die Position des Cursors vor der
           Schrittauflösung hängt von dem eingestellten
           Wert ab.
        case 3: if(res_old ==2 || res_old > 10){
                    blink_cursor(">",3,5);
                }else if(res_old > 5){
                    blink_cursor(">",3,6);
                }else{
                    blink_cursor(">",3,7);
                }break;
        case 4: blink_cursor("<",4,4);break;
        case 5: blink_cursor("<",4,8);break;
        case 6: blink_cursor(">",4,12);update_single_step
           =1;break;//Erreicht der Cursor die Pos. vor
           single step Faktor
        case 7: blink_cursor(">",2,12);update_autodrive=1;
           break;//oder Autodrive Faktor, muss der Wert um
            eine Stelle nach rechts geschoben werden.
    }
}
/*Darstellung Parametername Speed, Encoder und Steps*/
if(menu_state == 0 && menu_state_old == 1 ||
   update_enc_speed==1){//Wenn das Menü deaktiviert wird,
   oder die Dartstellung von Steps/Enc
    update_enc_speed=0;//geändert wurde
    menu_state_old = menu_state;
    display_puts(old_y, old_x, "␣");//wird der Menücursor
       gelöscht
    display_puts(3,12, "Speed␣␣␣␣");//Der Parametername
       Speed
    if(fpga_steps_enc_old==0){//sowie je nach Einstellung
        display_puts(1,12,"Encoder␣␣");//der Parametername
            Encoder
    }else{//oder
        display_puts(1,12,"Steps␣␣␣␣");//Steps dargestellt
           .
    }
    update_speed=1;//Zuletzt wird die aktuelle
       Impulsfrequenz angezeigt.
}
/*Dastellung Parameter Autodrive und Single Step Faktor im
    Menü*/
if(menu_state == 1 && menu_state_old == 0){//Wenn das Menü
    aktiviert wird
```



```
            menu_state_old = menu_state;
            display_puts(1,12, "Autodrive");//Werden die
               Parameternamen Autodrive
            display_putc(2,12, 0x8C);//plusminus
            display_puts_frame(2,13, fpga_autodrive_steps, 0,8);//
               mit dem dazugehörigen Wert
            display_puts(3,12, "Factor␣␣␣");//und der Single Step
               Faktor
            display_puts_frame(4,12, fpga_single_step_factor, "x"
               ,8);//mit zugehörigem Wert dargestellt.
        }
        /*Darstellung ausgewählter Parameter*/
        if(menu_state==1 && menu_select==1 && menu_print_select
           ==1){//Wird ein Parameter zur Einstellung ausgewählt
            menu_print_select=0;
            if(encoder_menu_value_old==4 || encoder_menu_value_old
               ==5){//wird je nach Position ein ausgefüllter Pfeil
                nach links
                display_putc(old_y, old_x, 0x11);
            }else{//oder
                display_putc(old_y, old_x, 0x10);//nach rechts
                   dargestellt.
            }
        }

    /*Nachdem alle aktualisierten Werte in den Display Buffer
       geschrieben wurden, wird der Buffer zum übertragen
       freigegeben*/
    display_tx_data= 1;
    }
}
/**********************************************************
 * TIMER 4 Interrupt-Handler 20ms lese/schreibe Daten FPGA
**********************************************************/
void TIM4_IRQHandler(){
    if(TIM_GetITStatus(TIM4, TIM_IT_Update)){
        TIM_ClearITPendingBit(TIM4, TIM_IT_Update);//clear
           interrupt flag
        static uint32_t fpga_ctrl_reg = 0;//Buffer
         /* Endlagen aus FPGA auslesen */
        fpga_receive_data = send_FPGA(R, FPGA_STEP_MODE_REG, 0);//
           Lese entlagen aus FPGA
        CW_Limit = (fpga_receive_data >> 16) & 0x1;//und Speicher
           sie in den passenden
```

```
CCW_Limit = (fpga_receive_data >> 17) & 0x1;//Variablen ab
   .
fpga_steps_left =(fpga_receive_data >> 19) & 0x1;//
   Speicher für noch Schirtte übrig Flag
/*Kontroll und Richtungsregister update*/
if(fpga_update==1){//Wenn neue Daten für den FPGA bereit
   liegen
    fpga_update=0;
    send_FPGA(W, FPGA_STEP_MODE_REG, FPGA_MODE_RES);//wird
        zuerst das Kontrollregister zurückgesetzt bevor
       neue Daten gesendet werden.
    fpga_ctrl_reg = zmx_reset;//Schreibe Reset Zustand im
       Buffer
    fpga_ctrl_reg = (fpga_ctrl_reg << 1) |
       zmx_entregen_old;//Schiebe 1 links und schreibe
       Endstufenzustand in Buffer
    fpga_ctrl_reg = (fpga_ctrl_reg << 12) | fpga_limit_cw;
       //Schiebe 12 links und schreibe Endlageneinstellung
        CW in Buffer
    fpga_ctrl_reg = (fpga_ctrl_reg << 4) | fpga_limit_ccw;
       //Schiebe 4 links und schreibe Endlageneinstellung
       CCW in Buffer
    fpga_ctrl_reg = (fpga_ctrl_reg << 8) | fpga_ctrl_mode;
       //Scheibe 8 links und schreibe Laufmodus in Buffer
    send_FPGA(W, FPGA_STEP_MODE_REG, fpga_ctrl_reg);//
       Aktualisiere Kontrollregister in FPGA
    /*Beschreiben des Step/Direction-Register im FPGA*/
    if(fpga_cw_autodrive==1){//Wenn Autodrive in Richtung
       CW
        fpga_cw_autodrive=0;//Rücksetzten der update Flag
        send_FPGA(W, FPGA_DIR_STEP_REG, FPGA_RUN_CW+
           fpga_autodrive_steps);//schreibe Laufrichtung
           CW und Anzahl der Schritte
    }else if(fpga_ccw_autodrive==1){//Wenn Autodrive in
       Richtung CCW
        fpga_ccw_autodrive=0;//Rücksetzten der update Flag
        send_FPGA(W, FPGA_DIR_STEP_REG, FPGA_RUN_CCW+
           fpga_autodrive_steps-1);//schreibe Laufrichtung
            CW und Anzahl der Schritte
    }else if(fpga_ccw_last_pos_drive==1){//Wenn die letzte
        Position in Richtung CCW angefahren werden soll
        fpga_ccw_last_pos_drive=0;//Rücksetzten der update
            Flag
```



```
        send_FPGA(W, FPGA_DIR_STEP_REG, FPGA_RUN_CCW+(
           fpga_pos_1-fpga_pos_2)-1);//schreibe
           Laufrichtung CW und Anzahl der Schritte
    }else if(fpga_cw_last_pos_drive==1){//Wenn die letzte
       Position in Richtung CW angefahren werden soll
        fpga_cw_last_pos_drive=0;//Rücksetzten der update
           Flag
        send_FPGA(W, FPGA_DIR_STEP_REG, FPGA_RUN_CW+(
           fpga_pos_2-fpga_pos_1));//schreibe Laufrichtung
            CW und Anzahl der Schritte
    }else if(fpga_single_step_cw==1){//Wenn Einzelschritt
       in Richtung CW gefahren werden soll
        fpga_single_step_cw=0;//Rücksetzten der update
           Flag
        send_FPGA(W, FPGA_DIR_STEP_REG, FPGA_RUN_CW+(
           fpga_single_step_ctr*fpga_single_step_factor));
           //schreibe Laufrichtung CW und Anzahl der
           Schritte*Faktor
        fpga_single_step_ctr=0;//Rücksetzten des
           Signelstep counter (bei Mehrfachdrehung des
           Drehencoders)
    }else if(fpga_single_step_ccw==1){//Wenn Einzelschritt
        in Richtung CCW gefahren werden soll
        fpga_single_step_ccw=0;//Rücksetzten der update
           Flag
        send_FPGA(W, FPGA_DIR_STEP_REG, FPGA_RUN_CCW+(
           fpga_single_step_ctr*fpga_single_step_factor)
           -1);//schreibe Laufrichtung CW und Anzahl der
           Schritte*Faktor
        fpga_single_step_ctr=0;//Rücksetzten des
           Signelstep counter (bei Mehrfachdrehung des
           Drehencoders)
    }
}
/*Auslesen des Encoder und Schrittzählers*/
fpga_receive_enc= send_FPGA(R, FPGA_ENCODER_REG, 0);//Lese
    gezählte Encoder und Speicher in Variable
fpga_receive_steps= send_FPGA(R, FPGA_STEP_CTR_REG, 0);//
   Lese gefahrene Schritte und Speicher in Variable
/*Speichern der Positionen für das Anfahren der letzten
   Position*/
if(fpga_pos_1!=fpga_receive_steps && fpga_ccw_drive==0 &&
   fpga_cw_drive==0 && fpga_steps_left==0){//Ist kein
```

```
           Laufmodus aktiv und die aktuelle Position ist ungleich
           der letzten gespeicherten
            fpga_pos_2=fpga_pos_1;//wird die startposition
               gespeichert
            fpga_pos_1=fpga_receive_steps;//und die aktuelle
               Position übernommen.
        }

    }
}
/***********************************************************
 * TIMER 3 Interrupt-Handler 1us, Sende Daten über SPI an das
    Display
***********************************************************/
void TIM3_IRQHandler(){
    if(TIM_GetITStatus(TIM3, TIM_IT_Update)){
        TIM_ClearITPendingBit(TIM3, TIM_IT_Update);//clear
           interrupt flag
        /*Wartezeit bevor eine neue Übertragung starten darf*/
        if(display_tx_wait>0){//Wenn noch gewartet werden muss
            display_tx_wait--;//wird der Zähler dekrementiert.
        }
    /*Senden an das Display*/
    if(display_tx_data==1 && display_tx_busy ==0 &&
       display_tx_wait == 0 ){//Befinden sich Daten im Buffer, es
       werden gerade keine Daten gesendet und die Wartezeit ist
       abgelaufen
        static int i = 0;
        display_tx_busy = 1;//wird die Busy Flag gesetzt.
        if(display_tx_data_ctr>0){//befinden sich noch Daten im
           Buffer
            CS_DISP_LOW;//wird CS für das Display low gesetzt
            if(databuffer[display_tx_data_ctr-1][i]!= '\0'){//
               solange nicht das Ende des String erreicht wurde
               werden Daten gesendet.
                if(i>1){//wurden schon Adresse und Startbyte
                   gesendet
                    Write_Display(databuffer[display_tx_data_ctr
                       -1][i], 0);//wird der jeweilige Char ohne
                       Startbyte übertragen
                }else if(i==1){//Wenn der erste Chhar im Array
                   übertragen wird
```



```
                    Write_Display(databuffer[display_tx_data_ctr
                       -1][i], 2);//muss zuerst ein Startbyte
                       gesendet werden.
                }else if(i==0){//Wird die Adresse übertragen
                   Write_Display(databuffer[display_tx_data_ctr
                      -1][i], 1);//muss hierfür das passende
                      Startbyte gesendet werden.
                }
                i++;//nach jedem Zeichen wird die Stelle im Array
                   erhöht.
            }else{//Wird der 0 Terminator erreicht
                i = 0;//wird die Array Stelle auf 0 gesetzt
                display_tx_data_ctr--;//und der Datencounter für
                   den Buffer um 1 verringert.
            }
        }
        if(display_tx_data_ctr==0){//Wenn der gesamte Buffer
           übertragen wurde
           display_tx_data=0;//Wird das Übertragungsflag zurück
              gesetzt
            CS_DISP_HIGH;//und CS wieder high.
        }
        display_tx_busy = 0;//ist eine Übertragung abgeschlossen
           wird auch Busy wieder auf 0 gesetzt.
        }
    }
}
/***********************************************************
 * TIMER 2 Interrupt-Handler 250ms (LED CW, LED CCW)
***********************************************************/
void TIM2_IRQHandler(){
    if(TIM_GetITStatus(TIM2, TIM_IT_Update)){
        TIM_ClearITPendingBit(TIM2, TIM_IT_Update);//clear
           interrupt flag
        startup_ctr++;//Wartezeit beim Einschalten der
           Handsteuerung
        LED_ctr_high++;//counter für CW und CCW LED
        LED_ctr_dis++;//counter für CW und CCW LED wenn deaktivert
        Blink_ctr++;//counter für blinkenden Cursor im Menü
        if(Blink_ctr>= 8){//Nach 2 Sekunden
            Blink_ctr = 0;//wird der Counter zurück gesetzt
        }
        /*Speichern von Parametern im EEPROM*/
```

```
        if(eeprom_save_ctr==120 || eeprom_save==1){//Wird die
           Impulsfrequenz für 30s nicht verändert, oder das Menü
           verlassen
            eeprom_save=0;//sollen die Werte im EEPROM gespeichert
                werden.
            EE_WriteVariable(eeprom_addr[0], (uint16_t)Irun);//
               Speicher Laufstrom
            EE_WriteVariable(eeprom_addr[1], (uint16_t)Istop);//
               Speicher Stoppstrom
            EE_WriteVariable(eeprom_addr[2], (uint16_t)res);//
               Speicher Schrittauflösung
            EE_WriteVariable(eeprom_addr[3], (uint16_t)
               fpga_limit_ccw);//Speicher Endlageneinstellung CCW
            EE_WriteVariable(eeprom_addr[4], (uint16_t)
               fpga_limit_cw);//Speicher Endlageneinstellung CW
            EE_WriteVariable(eeprom_addr[5], (uint16_t)
               encoder_speed_value);//Speicher Impulsfrequenz
            eeprom_save_ctr++;//Speichercounter um 1 erhöhen
        }else if(eeprom_save_ctr<120){//Solange nicht 30 Sekunden
           vorbei sind
            eeprom_save_ctr++;//wird der Speichercounter um 1
               erhöht
        }
    }
}
/************************************************************
 * TIMER 6 Interrupt-Handler 50ms, Auswertung der Drehencoder
************************************************************/
void TIM6_DAC_IRQHandler(){
    if(TIM_GetITStatus(TIM6, TIM_IT_Update)){
        TIM_ClearITPendingBit(TIM6, TIM_IT_Update);//clear
           interrupt flag
        /*
           ***********************************************************

         * Auswertung Menü-Drehencoder
        ***********************************************************
           */
        static int8_t buffer = 0;//Speicher für den aktuellen
           Zählerstand
        /*Positionsbestimmung im Menü*/
        if(menu_state==1 && menu_select == 0){//Wenn das Menü
           aktiv ist und keine Parameter ausgewählt
```



```
            buffer=(int8_t)TIM_GetCounter(MENU);//Speicher den
               aktuellen Zählerstand im Buffer
            encoder_menu_value = encoder_menu_value + buffer;//und
                addiere zur Menüposition
            if(fpga_cw_drive || fpga_ccw_drive){//Wenn der Motor
               in Bewegung
                 encoder_menu_value= Grenzwert(encoder_menu_value
                    ,1,5);//sind nur Parameter 1 bis 5 einstellbar
                    .
            }else{//Ist Motor nicht in Bewegung
                 encoder_menu_value= Grenzwert(encoder_menu_value
                    ,1,7);//sind alle Parameter einstellbar
            }
            TIM8->CNT=0;//Damit nicht dauerhaft der Zählerstand
               addiert wird, wird er auf 0 gesetzt
        /*Einstellung des ausgewählten Parameters*/
        }else if(menu_state==1 && menu_select == 1){//Wenn das
           Menü aktiv ist und ein Parameter ausgewählt ist
             buffer=(int8_t)TIM_GetCounter(MENU);//wird der
                aktuelle Zählerstand gelesen
             if(buffer!=0){//Hat sich der Wert verändert,
                switch(encoder_menu_value_old){//wird je nach
                   Auswahl des Parameters dessen Wert verändert.
                    /*Laufstrom*/
                    case 1:     Irun = Irun+ buffer;//Addiere
                       aktuellen Zählerstand zum Wert des
                       Parameters
                                Irun = Grenzwert(Irun,1,50);//Der
                                   Laufstrom ist einstellbar von
                                   0,1A bis 5,0A
                                TIM8->CNT=0;//Damit nicht
                                   dauerhaft der Zählerstand
                                   addiert wird, wird er auf 0
                                   gesetzt
                                break;
                    /*Stoppstrom*/
                    case 2:     Istop = Istop+ buffer;//Addiere
                       aktuellen Zählerstand zum Wert des
                       Parameters
                                Istop = Grenzwert(Istop,-1,Irun);
                                   //-1 = Auto, ansonsten 0A bis
                                   5,0A
                                TIM8->CNT=0;//Damit nicht
                                   dauerhaft der Zählerstand
```

```
                addiert wird, wird er auf 0
                gesetzt
            break;
/*Schrittauflösung*/
case 3:     res = res+ buffer;//Addiere
    aktuellen Zählerstand zum Wert des
    Parameters
            res = Grenzwert(res,0,13);//14
                unterschiedliche
                Schrittauflösungen (siehe
                Manual)
            TIM8->CNT=0;//Damit nicht
                dauerhaft der Zählerstand
                addiert wird, wird er auf 0
                gesetzt
            break;
/*Endlage CCW*/
case 4:     fpga_limit_ccw = fpga_limit_ccw +
    buffer;//Addiere aktuellen Zählerstand zum
    Wert des Parameters
            fpga_limit_ccw = Grenzwert(
                fpga_limit_ccw, 1, 3);//
                Endlageneinstellung: high, low,
                 disabled
            TIM8->CNT=0;//Damit nicht
                dauerhaft der Zählerstand
                addiert wird, wird er auf 0
                gesetzt
            break;
/*Endlage CW*/
case 5:     fpga_limit_cw = fpga_limit_cw +
    buffer;//Addiere aktuellen Zählerstand zum
    Wert des Parameters
            fpga_limit_cw = Grenzwert(
                fpga_limit_cw, 1, 3);//
                Endlageneinstellung: high, low,
                 disabled
            TIM8->CNT=0;//Damit nicht
                dauerhaft der Zählerstand
                addiert wird, wird er auf 0
                gesetzt
            break;
case 6:     fpga_single_step_factor =
    fpga_single_step_factor +buffer;//Addiere
```



```
                    aktuellen Zählerstand zum Wert des
                    Parameters
                            fpga_single_step_factor= Grenzwert
                                (fpga_single_step_factor,
                                SINGLESTEP_MIN,SINGLESTEP_MAX);
                                //Singlestep Faktor 1x bis 100x
                            TIM8->CNT=0;//Damit nicht
                                dauerhaft der Zählerstand
                                addiert wird, wird er auf 0
                                gesetzt
                            break;
                case 7:     fpga_autodrive_steps=
                    fpga_autodrive_steps+buffer*
                    AUTODRIVE_FAKTOR;//Addiere aktuellen
                    Zählerstand zum Wert des Parameters*Faktor
                            fpga_autodrive_steps=Grenzwert(
                                fpga_autodrive_steps,
                                AUTODRIVE_MIN,AUTODRIVE_MAX);//
                                Autodrive Steps
                            TIM8->CNT=0;//Damit nicht
                                dauerhaft der Zählerstand
                                addiert wird, wird er auf 0
                                gesetzt
            }
        }
}
/*Encoderwert wird gespeichert für weitere Auswertung*/
if(encoder_menu_value!=encoder_menu_value_old){//Nur bei
    Änderung des Wertes
     encoder_menu_value_old=encoder_menu_value;//wird er
        gespeichert.
}
/*Single Step Modus, Auswertung des Drehencoders*/
if(zmx_entregen_old==0 && menu_state==0 && fpga_cw_drive
    ==0 && fpga_ccw_drive==0){//Ist die Endstufe aktiv und
    der Motor steht ist der Single Step Modus aktiv
     buffer=(int8_t)TIM_GetCounter(MENU);//Auswertung
        Drehencoder
     fpga_single_step_ctr = fpga_single_step_ctr + buffer;
        //und Speicherung der Anzahl der Schritte
     TIM8->CNT=0;//
     /*Single Step CW*/
     if(fpga_single_step_ctr > 0){//Wenn die Anzahl der
        Schritte positiv ist
```

```
        fpga_ctrl_mode=FPGA_MODE_STEP;//setze den
           Laufmodus auf Single Step
        fpga_single_step_cw=1;//und die Flag für CW run.
        fpga_update=1;//Update Kontrollregister
    /*Single Step CCW*/
    }else if(fpga_single_step_ctr < 0){//Wenn die Anzahl
       der Schritte negativ ist
        fpga_single_step_ctr=~fpga_single_step_ctr+1;//
           Invertiere den Counter+1 um eine positiv Zahl
           zu erhalten
        fpga_ctrl_mode=FPGA_MODE_STEP;//setze den
           Laufmodus auf Single Step
        fpga_single_step_ccw=1;//und die Flag für CCW run.
        fpga_update=1;//Update Kontrollregister
    }
}
/*
   **********************************************************

 * Auswertung Speed-Drehencoder
**********************************************************
   */
buffer=(int8_t)TIM_GetCounter(SPEED);//Auswertung
   Drehencoder Speed
if(buffer!=0){//Ändert sich der Wert
    if(encoder_speed_value <100){//und ist kleiner als 100
        encoder_speed_value = encoder_speed_value + buffer
           ;//wird die Geschwindigkeit um 1Hz erhöht.
    }else if(99< encoder_speed_value &&
       encoder_speed_value <1000){//Ist der Wert zwischen
       100 und 999
        encoder_speed_value = encoder_speed_value + 10*
           buffer;//wird der Wert um 10Hz erhöht.
    }else if(999< encoder_speed_value &&
       encoder_speed_value <10000 ){//Ist der Wert
       zwischen 100 und 999
        encoder_speed_value = encoder_speed_value + 100*
           buffer;//wird der Wert um 100Hz erhöht.
    }else if(9999< encoder_speed_value &&
       encoder_speed_value  <100000){//Ist der Wert
       zwischen 100 und 999
        encoder_speed_value = encoder_speed_value + 1000*
           buffer;//wird der Wert um 1000Hz erhöht.
    }
```



```
            encoder_speed_value= Grenzwert(encoder_speed_value
               ,1,100000);//Min und Max wert sind auf 1Hz bzw 100
               kHz begrenzt
            TIM1->CNT=0;//Rücksetzen des Timers um dauerthafte
               Addidition zu verhindern.
        }
        /*Senden an FPGA und speichern im EEPROM*/
        if(encoder_speed_value != encoder_speed_value_old){//Wird
           die Geschwindigkeit verändert
            encoder_speed_value_old= encoder_speed_value;//wird
               der neue Wert gespeichert.
            update_speed=1;//Die Anzeige  im Display wird
               aktualisert
            eeprom_save_ctr=0;//und der EEPROM Speicher Counter
               auf 0 gesetzt.
            send_FPGA(W, FPGA_DDS_REG, calc_DDS_count(
               encoder_speed_value_old));//Der DDS Zählerwert wird
                berechnet und an den FPGA übertragen.
        }
    }
}
```

D.2. encoder.c

```
/**********************************************************
Firma: DESY Hamburg
Autor: Joshua Supra
Projekt: ZMX-Handsteuerung Serie 3
Datei: encoder.c
Letzte Aenderung: 30.08.2018
Kurzbeschreibung: Initialiserung fuer Drehencoder Menue und
   Frqueunzeinstellung
(c) Copyright 2018 DESY
**********************************************************/

#include "stm32f4xx_conf.h"
#include "encoder.h"
#include "various.h"
#include "main.h"

/**********************************************************
Initialisierung der GPIOs für Encoder Eingänge
**********************************************************/
```

```
void init_GPIO_encoder(void){
    /*Drehencoder 1 = Menue*/
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);//
       aktiviere Clock an PortC (84Mhz)
    /* Time base configuration */
    GPIO_InitTypeDef GPIO_InitStructure;
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6|GPIO_Pin_7;//Pin 6 =
       Encoder Menü A, Pin 7 = Encoder Menue B
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//Konfiguriere
       Pins Alternate Function
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//Push Pull Mode
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_25MHz;//Clock an
       Pins 25MHz
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;//kein Pull up
       /down
    GPIO_Init(GPIOC, &GPIO_InitStructure);

    GPIO_PinAFConfig(GPIOC,GPIO_PinSource6,GPIO_AF_TIM8);//Timer 8
        an Pin C6
    GPIO_PinAFConfig(GPIOC,GPIO_PinSource7,GPIO_AF_TIM8);//Timer 8
        an Pin C7

    /*Drehencoder 2 = Speed*/
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOE, ENABLE);//
       aktiviere Clock an PortE
    GPIO_InitTypeDef GPIO_InitStruct;
    GPIO_InitStruct.GPIO_Pin = GPIO_Pin_9|GPIO_Pin_11;//Pin 9 =
       Encoder Frequenz A, Pin 11 = Encoder Frequenz B
    GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AF;//Konfiguriere Pins
       Alternate Function
    GPIO_InitStruct.GPIO_OType = GPIO_OType_PP;//Push Pull Mode
    GPIO_InitStruct.GPIO_Speed = GPIO_Speed_25MHz;//Clock an Pins
       25MHz
    GPIO_InitStruct.GPIO_PuPd = GPIO_PuPd_NOPULL;//kein Pull up/
       down
    GPIO_Init(GPIOE, &GPIO_InitStruct);

    GPIO_PinAFConfig(GPIOE,GPIO_PinSource9,GPIO_AF_TIM1);//Timer 1
        an Pin E9
    GPIO_PinAFConfig(GPIOE,GPIO_PinSource11,GPIO_AF_TIM1);//Timer
       1 an Pin E11
}
/**********************************************************
Initialisierung Timer für den Encoder für die Menüsteuerung
```



```
**********************************************************/
void init_menu_encoder(void)
{
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM8, ENABLE);//
       aktiviere Clock fuer Timer 8 (168Mhz)
    /* Time base configuration */
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
    TIM_TimeBaseStructure.TIM_Period = ENCODER_PERIOD;//lege Timer
        Periode fest
    TIM_TimeBaseStructure.TIM_Prescaler = 0x1;//Inkrementiere um 1
    TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV4;//Timer
        clock = 41Mhz
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;//
       Aufwärtszähler

    TIM_TimeBaseInit(TIM8, &TIM_TimeBaseStructure);
    TIM_EncoderInterfaceConfig(TIM8, TIM_EncoderMode_TI1,
       TIM_ICPolarity_Rising, TIM_ICPolarity_Rising);//Nur Channel
        1 wird ausgewertet

    TIM8->CNT=0;//Lade Wert in Timer

    /*Filter konfiguration*/
    TIM_ICInitTypeDef TIM_ICInitStruct;
    TIM_ICInitStruct.TIM_Channel = TIM_Channel_1;//Filter an
       Channel 1
    TIM_ICInitStruct.TIM_ICSelection= TIM_ICSelection_DirectTI;//
       Direct Capture
    TIM_ICInitStruct.TIM_ICPrescaler= TIM_ICPSC_DIV8;
    TIM_ICInitStruct.TIM_ICFilter=0x0F;
    TIM_ICInit(TIM8, &TIM_ICInitStruct);

    TIM_Cmd(TIM8, ENABLE);//Timer 8 EN
}
/**********************************************************
Initialisierung Timer für den Encoder für die Adjustierung
**********************************************************/
void init_adjust_encoder(void)
{
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM1, ENABLE);//
       aktiviere Clock fuer Timer 1 (168Mhz)

    /* Time base configuration */
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStruct;
```

```
    TIM_TimeBaseStruct.TIM_Period = ENCODER_PERIOD;//lege Timer
       Periode fest
    TIM_TimeBaseStruct.TIM_Prescaler = 0x1;//Inkrementiere um 1
    TIM_TimeBaseStruct.TIM_ClockDivision = TIM_CKD_DIV4 ;//Timer
       clock = 42Mhz
    TIM_TimeBaseStruct.TIM_CounterMode = TIM_CounterMode_Up;//
       Aufwärtszähler

    TIM_TimeBaseInit(TIM1, &TIM_TimeBaseStruct);
    TIM_EncoderInterfaceConfig(TIM1, TIM_EncoderMode_TI1,
       TIM_ICPolarity_Rising, TIM_ICPolarity_Rising);//Nur Channel
        1 wird ausgewertet

    TIM1->CNT=0;//Lade Wert in Timer

    /*Filter konfiguration*/
    TIM_ICInitTypeDef TIM_ICInitStruct;
    TIM_ICInitStruct.TIM_Channel = TIM_Channel_1;//Filter an
       Channel 1
    TIM_ICInitStruct.TIM_ICSelection= TIM_ICSelection_DirectTI;//
       Direct Capture
    TIM_ICInitStruct.TIM_ICPrescaler= TIM_ICPSC_DIV8;
    TIM_ICInitStruct.TIM_ICFilter=0x0F;
    TIM_ICInit(TIM1, &TIM_ICInitStruct);
    TIM_Cmd(TIM1, ENABLE);
}
/****************************************************************
 * TIMER 6 Initialisierung (84MHz clock), 20Hz = 84MHz/[(499+1)
    *(8399+1)]
 * Update alle 50ms, Interrupt
****************************************************************/
void Timer6_Init(void){
    TIM_TimeBaseInitTypeDef TIM_TimeBaseInitStruct;
    NVIC_InitTypeDef NVIC_InitStruct;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM6, ENABLE);

    TIM_TimeBaseInitStruct.TIM_Prescaler = 8399;
    TIM_TimeBaseInitStruct.TIM_Period = 499;
    TIM_TimeBaseInitStruct.TIM_ClockDivision = TIM_CKD_DIV4;
    TIM_TimeBaseInitStruct.TIM_CounterMode = TIM_CounterMode_Up;

    TIM_TimeBaseInit(TIM6, &TIM_TimeBaseInitStruct);
```



```
    TIM_ITConfig(TIM6, TIM_IT_Update, ENABLE);

    TIM_Cmd(TIM6, ENABLE);

    NVIC_InitStruct.NVIC_IRQChannel = TIM6_DAC_IRQn;
    NVIC_InitStruct.NVIC_IRQChannelPreemptionPriority = 3;
    NVIC_InitStruct.NVIC_IRQChannelSubPriority = 0;
    NVIC_InitStruct.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStruct);
}
```

D.3. ea_dip203B.c

```
/************************************************************
Firma: DESY Hamburg
Autor: Joshua Supra
Projekt: ZMX-Handsteuerung Serie 3
Datei: ea_dip203B.c
Letzte Aenderung: 30.08.2018
Kurzbeschreibung: Initialiserung fuer das Display, Funktionen für
   das Senden
und Aufbereiten von Daten an das Display
(c) Copyright 2018 DESY
************************************************************/
#include "stm32f4xx_conf.h"
#include "ea_dip203B.h"
#include "various.h"
#include "main.h"

/************************************************************
 * Initialisierung: SPI1, Frquenz: 42MHz
 *      PIN A4: CS
 *      PIN A5: SCLK
 *      PIN A6: MISO
 *      PIN A7: MOSI
************************************************************/
void init_display_SPI1(void){
    //GPIO und SPI Typedef
    GPIO_InitTypeDef GPIO_InitStruct;
    SPI_InitTypeDef SPI_InitStruct;

    // Enable Clock an GPIOA
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);
```

```
    //Konfigueriere GPIO Pins
    GPIO_InitStruct.GPIO_Pin = GPIO_Pin_7 | GPIO_Pin_6 |
       GPIO_Pin_5;
    GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStruct);

    // Verbinde SPI1 mit Pins
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource5, GPIO_AF_SPI1);
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource6, GPIO_AF_SPI1);
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource7, GPIO_AF_SPI1);

    //Konfigueriere NSS Pin
    GPIO_InitStruct.GPIO_Pin = GPIO_Pin_4;
    GPIO_InitStruct.GPIO_Mode = GPIO_Mode_OUT;
    GPIO_InitStruct.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStruct.GPIO_PuPd = GPIO_PuPd_UP;
    GPIO_Init(GPIOA, &GPIO_InitStruct);

    // enable peripheral clock
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1, ENABLE);
    SPI_InitStruct.SPI_Direction = SPI_Direction_2Lines_FullDuplex
       ; // FUll Duplex, MISO, MOSI
    SPI_InitStruct.SPI_Mode = SPI_Mode_Master;     //
       Microkontroller ist Master
    SPI_InitStruct.SPI_DataSize = SPI_DataSize_8b; // Paketgröße =
        8 Bit
    SPI_InitStruct.SPI_CPOL = SPI_CPOL_High;        // Clock high
       wenn idle
    SPI_InitStruct.SPI_CPHA = SPI_CPHA_2Edge;      // Daten werden
        mit 2 Flanke übernommen
    SPI_InitStruct.SPI_NSS = SPI_NSS_Soft |
       SPI_NSSInternalSoft_Set; // Soft NSS
    SPI_InitStruct.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_2
       ; // SPI Frequenz ist 42MHz
    SPI_InitStruct.SPI_FirstBit = SPI_FirstBit_LSB;// LSB wird
       zuerst übertragen
    SPI_Init(SPI1, &SPI_InitStruct);

    SPI_Cmd(SPI1, ENABLE); // enable SPI1
}
/**************************************************************
```



```
 * TIMER 5 Initialisierung (84MHz clock), 50Hz = 84MHz/[(499+1)
    *(3359+1)]
 * Update alle 20ms, Interrupt
**************************************************************/
void Timer5_Init(void){
    TIM_TimeBaseInitTypeDef TIM_TimeBaseInitStruct;
    NVIC_InitTypeDef NVIC_InitStruct;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM5, ENABLE);

    TIM_TimeBaseInitStruct.TIM_Prescaler = 3359;
    TIM_TimeBaseInitStruct.TIM_Period = 499;
    TIM_TimeBaseInitStruct.TIM_ClockDivision = TIM_CKD_DIV4;
    TIM_TimeBaseInitStruct.TIM_CounterMode = TIM_CounterMode_Up;

    TIM_TimeBaseInit(TIM5, &TIM_TimeBaseInitStruct);

    TIM_ITConfig(TIM5, TIM_IT_Update, ENABLE);

    TIM_Cmd(TIM5, ENABLE);

    NVIC_InitStruct.NVIC_IRQChannel = TIM5_IRQn;
    NVIC_InitStruct.NVIC_IRQChannelPreemptionPriority = 1;
    NVIC_InitStruct.NVIC_IRQChannelSubPriority = 0;
    NVIC_InitStruct.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStruct);
}
/**************************************************************
 * TIMER 3 Initialisierung (84MHz clock), 1MHz := 84MHz/[(1+1)
    *(41+1)]
 * Update alle 1us, Interrupt
**************************************************************/
void Timer3_Init(void){
    TIM_TimeBaseInitTypeDef TIM_TimeBaseInitStruct;
    NVIC_InitTypeDef NVIC_InitStruct;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);//
       aktiviere Clock fuer Timer 3

    TIM_TimeBaseInitStruct.TIM_Prescaler = 1;
    TIM_TimeBaseInitStruct.TIM_Period = 41;
    TIM_TimeBaseInitStruct.TIM_ClockDivision = TIM_CKD_DIV1;
    TIM_TimeBaseInitStruct.TIM_CounterMode = TIM_CounterMode_Up;
```

```
    TIM_TimeBaseInit(TIM3, &TIM_TimeBaseInitStruct);

    TIM_ITConfig(TIM3, TIM_IT_Update, ENABLE);

    TIM_Cmd(TIM3, ENABLE);

    NVIC_InitStruct.NVIC_IRQChannel = TIM3_IRQn;
    NVIC_InitStruct.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_InitStruct.NVIC_IRQChannelSubPriority = 1;
    NVIC_InitStruct.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStruct);

}
/****************************************************************
 * Initialisierungsvorgang fuer das Display (Siehe Manual
    ea_dip203B), der Kontroller
 * im Display wird fuer die Nutzung konfiguriert
****************************************************************/
void init_display(void){
    CS_DISP_LOW;
    /* Array mit selbstdefienierten Zeichen*/
    char Zeichen[64] ={
    0x00, 0x1F, 0x11, 0x0F, 0x0F, 0x0E, 0x11, 0x1F, // c mit
       strich negiert
    0x00, 0x1F, 0x0E, 0x0E, 0x0A, 0x0A, 0x15, 0x1F, // w mit
       strich negiert
    0x1F, 0x1F, 0x11, 0x0F, 0x0F, 0x0E, 0x11, 0x1F, // c negiert
    0x1F, 0x1F, 0x0E, 0x0E, 0x0A, 0x0A, 0x15, 0x1F, // w negiert
    0x1F, 0x00, 0x0E, 0x10, 0x10, 0x11, 0x0E, 0x00, // c mit
       strich
    0x1F, 0x00, 0x11, 0x11, 0x15, 0x15, 0x0A, 0x00, // w mit
       strich
    0x0E, 0x11, 0x10, 0x1F, 0x10, 0x11, 0x0E, 0x00, // C
       durchgestrichen
    0x11, 0x11, 0x15, 0x1F, 0x15, 0x15, 0x0A, 0x00, // W
       durchgestrichen
    };
    Write_Display(EXTENSION_REGISTER_OFF, 1);
    while(display_tx_wait);//warte Ausfuerungszeit
    Write_Display(INIT_ENTRY_MD_SET, 1);
    while(display_tx_wait);//warte Ausfuerungszeit
    Write_Display(EXTENSION_REGISTER_ON, 1);
    while(display_tx_wait);//warte Ausfuerungszeit
    Write_Display(INIT_4_LINES_SET,1);
```



```
    while(display_tx_wait);//warte Ausfuerungszeit
    Write_Display(SEGRAM_ADR_0,1);
    while(display_tx_wait);//warte Ausfuerungszeit
    for(int i=0;i!=16;i++){
        Write_Display(EMPTY_CHARACTER, 2);
    while(display_tx_wait);//warte Ausfuerungszeit
    }
    Write_Display(EXTENSION_REGISTER_OFF, 1);
    while(display_tx_wait);//warte Ausfuerungszeit
    Write_Display(DISPLAY_ON, 1);
    while(display_tx_wait);//warte Ausfuerungszeit
    Write_Display(CLEAR_DISPLAY, 1);
    while(display_tx_wait);//warte Ausfuerungszeit
    User_symbol(Zeichen);//Schreibe selbst definierte Zeichen in
       das Display
    CS_DISP_HIGH;
}
/**************************************************************
 * Sende Daten an das Display, mit passenden Start Byte.
    Festlegung der
 * der Warte Zeit fuer SPI
 *      data: Zeichen oder Befehl der gesendet wird
 *      mode: Am Anfang einer Uebertraung muss das passend Start
    Byte
 *            uebertragen werden.
**************************************************************/
void Write_Display(unsigned char data, uint16_t mode){
    switch(mode){
        case 1: SPI_I2S_SendData(SPI1, START_INSTRUC); break;//
           wenn mode Anweisung sende START_INSTUC
        case 2: SPI_I2S_SendData(SPI1, START_WRITE); break;//wenn
           mode sende Zeichen -> START_WRITE
    }
    while( !(SPI1->SR & SPI_I2S_FLAG_TXE) );//warte bis
       Uebertragung abgeschlossen
    SPI1_send(data);//sende Character ueber SPI
    if(data == CLEAR_DISPLAY || data ==  DISPLAY_ON){//Wenn die
       Anweisung loesche Display oder Display an
        display_tx_wait = 1530;// muss 1,53ms gewartet werden
           bevor die neachste Anweisung gesendet werden darf
    }
    else{//wird ein Zeichen in das Display geschrieben
        display_tx_wait = 43;//muss 43 us gewartet werden.
    }
```

```
}
/**********************************************************
 * Sende Char an das Display ueber SPI, Aufteilung in lower und
   higher Byte
 * (vergl. Manual)
 *      data: Daten die in das SPI Senderegister geschrieben
   werden
**********************************************************/
void SPI1_send(unsigned char data){
    SPI_I2S_SendData(SPI1, (data & 0x0F));// sende lower Byte
    while( !(SPI1->SR & SPI_I2S_FLAG_TXE) ); // Warte bis transmit
        complete
    SPI_I2S_SendData(SPI1, ((data>>4) & 0x0F));//sende higher Byte
    while( !(SPI1->SR & SPI_I2S_FLAG_TXE) ); // Warte bis transmit
        complete
}
/**********************************************************
 * Display Fenster unsigned, erstellt ein vorzeichenloses
   Schreibfenster
 * im Display
 *      y: Y-Koordinate im Display
 *      x: X-Koordinate im Display
 *      value: Wert der im Fenster angezeigt werden soll
 *      data: Zusätzliche Zeichen können angefügt werden
 *      max_digits: maximale Fensterbreite
**********************************************************/
void display_puts_frame(uint8_t y, uint8_t x, uint32_t value, char
    *data, uint8_t max_digits){
    uint8_t cnt = 0;
    uint8_t n = 0;
    char Display_Array[9]= "0";
    n= ItoA_unsigned(value, Display_Array);// wandel den Int to
       ASCII signed und speicher Anzahl der Elemente im Array
    while(*data){// wenn zusaetzliche Chars angefuegt werden
       sollen
        Display_Array[n+cnt]= *data;
        cnt++;// Anzahl der Chars +1
        data++;// erhoehe Array pointer
    }
    for(uint8_t i= 0; i<(max_digits-cnt-n); i++){//noch freie
       Stellen im "Fenster"
        Display_Array[n+cnt+i]= '␣';// werden mit Leerzeichen
           aufgefüllt
    }
```



```
    Display_Array[9]= '\0';
    display_puts(y,x,Display_Array);//schreibe Daten in das den
       Buffer
}
/**********************************************************
 * Display Fenster signed, erstellt ein vorzeichenbehaftetes
   Schreibfenster
 * im Display
 *      y: Y-Koordinate im Display
 *      x: X-Koordinate im Display
 *      value: Wert der im Fenster angezeigt werden soll
 *      data: Zusätzliche Zeichen können angefügt werden
 *      max_digits: maximale Fensterbreite
**********************************************************/
void display_puts_frame_s(uint8_t y, uint8_t x, uint32_t value,
   char *data, uint8_t max_digits){
    uint8_t cnt = 0;
    uint8_t n = 0;
    char Display_Array[9]= "0";
    n= ItoA_signed(value, Display_Array);// wandel den Int to
       ASCII signed und speicher Anzahl der Elemente im Array
    while(*data){// wenn zusätzliche Chars angefügt werden sollen
        Display_Array[n+cnt]= *data;
        cnt++;// Anzahl der Chars +1
        data++;// erhöhe Array pointer
    }
    for(uint8_t i= 0; i<(max_digits-cnt-n); i++){//noch freie
       Stellen im "Fenster"
        Display_Array[n+cnt+i]= '␣';// werden mit Leerzeichen
           aufgefüllt
    }
    Display_Array[9]= '\0';
    display_puts(y,x,Display_Array);//schreibe Daten in das den
       Buffer
}
/**********************************************************
 * schreibe String in den Display Buffer
 *      y: Y-Koordinate im Display
 *      x: X-Koordinate im Display
 *      data: String der dargestellt werden soll
**********************************************************/
void display_puts(int y, int x,  char *data){
    databuffer[display_tx_data_ctr][0] = convert_addr(y, x);//
       Schreibe Koordinaten in den Buffer
```

```
    int i= 1;
    while(*data){//Solange der String nicht zu Ende ist
        databuffer[display_tx_data_ctr][i]= *data;//wird das
           jeweiligen Zeichen in den Buffer geschrieben
        data++;
        i++;
    }
    databuffer[display_tx_data_ctr][i]= '\0';//Füge Terminator am
       Ende ein
    display_tx_data_ctr++;//Erhöhe die Anzahl der zu übertragenen
       Daten
}
/************************************************************
 * schreibe ein Zeichen in den Display Buffer (für Sonderzeichen)
 *      y: Y-Koordinate im Display
 *      x: X-Koordinate im Display
 *      data: Zeichen die dargestellt werden sollen
 ************************************************************/
void display_putc(int y, int x,  uint8_t data){
    databuffer[display_tx_data_ctr][0] = convert_addr(y, x);//
       Schreibe Koordinaten in den Buffer
    databuffer[display_tx_data_ctr][1]= data;//Schreibe Zeichen in
        den Buffer
    databuffer[display_tx_data_ctr][2]= '\0';//Füge Terminator am
       Ende ein
    display_tx_data_ctr++;//Erhöhe die Anzahl der zu übertragenen
       Daten
}
/************************************************************
 * Umwandlung der Koordinaten für das Display
 *      y: Y-Koordinate im Display
 *      x: X-Koordinate im Display
 ************************************************************/
int convert_addr(int y, int x){
    int line;
    switch(y){//Setze Offset für die Zeile
    case 1: line=0x80;break; //Zeile 1
    case 2: line=0xA0;break; //Zeile 2
    case 3: line=0xC0;break; //Zeile 3
    case 4: line=0xE0;break; //Zeile 4
    }
    return line+x-1;//gebe Adresse zurück. Zeile+Spalte-1
}
/*
```



```
   ********************************************************************

* Schreibe Selbsdefiniertes Zeichen in das Display GC RAM
********************************************************************
   */
void User_symbol(char *zeichen){
unsigned char lcd_i; //Anzahl der Zeichen
  for(lcd_i = 0; lcd_i < 8; lcd_i++) {
    Write_Display_CG(lcd_i, &zeichen[lcd_i * 8]);
  }
}
/*
   ********************************************************************

 * Schreibe in das Display GC RAM (Pixel für Pixel)
 *      addr: Adresse im GC RAM
 *      zeichen: Adresse im GC RAM
********************************************************************
   */
void Write_Display_CG(unsigned char addr, char *zeichen) {
unsigned char lcd_i;
  for(lcd_i = 0; lcd_i < 8; lcd_i++) {
    Write_Display(0x40 + addr * 8 + lcd_i,1);//Setze Adresse im GC
        RAM
    while(display_tx_wait);//warte bis Display bereit
    Write_Display(*zeichen,2);//Schreibe Pixel
    while(display_tx_wait);//warte bis Display bereit
    zeichen++;
  }
}
/*
   ********************************************************************

 * Schreibe Strom Wert in das Display. Format: X,XA
 *      y: Y-Koordinate im Display
 *      x: X-Koordinate im Display
 *      Wert: Stromwert der Anzeigt werden soll
********************************************************************
   */
void display_put_ASCII_Current(uint8_t y, uint8_t x, unsigned int
   Wert){
    char Wert_ACSII[4];//Buffer
    ItoA_point(Wert,Wert_ACSII);// Wandel Wert in ASCII mit Komma
       darstellung
```

```
    Wert_ACSII[3]='A';//füge A an
    Wert_ACSII[4]='\0';//Terminator am Ende
    display_puts(y,x,Wert_ACSII);//Schreibe Buffer in Display
       Buffer
    }
/*
   ********************************************************************
 * Int Umwandlung in ASCII für die Ströme, Spannung
 *      u: Int Wert der gewandelt werden soll
 *      Buffer: Speicher für den ASCII Wert
**********************************************************************
   */
void ItoA_point(uint16_t u, char* Buffer){
   int i = 0;
    do {
      Buffer[i++] = '0'+ u % 10;// Den Restwert aus '0'+u/10
         brechnen
        if (u<10){// Wenn u kleiner als 10 ist wird im Buffer eine
            0 vor u erstellt
            if(i==1){// An der 2 stellen im Sting wird ein
                Buffer[i++]=',';//Komma eingefügt
                }
            if(i==2){// An der 3 Stelle im String wird
                Buffer[i++]='0';// Terminator eingefügt
                }
            }
        if (u>=10){//Wenn u größer als 10 kein Terminator
            if(i==1){// An der 2 stellen im Sting wird ein
                Buffer[i++]=',';// komma eingefügt
                }
            }
      u /= 10;// U durch 10 teilen.
    } while( u > 0 );// Solange U ist größer 0 wird die funktion
       wiederhollt.

    Buffer[i]=ASCII_Spiegeln(Buffer,i); //ASCII wert spiegeln
    }
/************************************************************
 * Blink Cursor: Erzeugt ein einen blinkendes Symbol
 *      y: Y-Koordinate im Display
 *      x: X-Koordinate im Display
 *      data: Symbol das angezeigt wird
************************************************************/
```



```
void blink_cursor(char *data, uint8_t y, uint8_t x){
    static unsigned char flag = 0;
    if(old_x != x || old_y != y){//Wenn neue Zeigerposition
        Blink_ctr = 0;//Rücksetze Blink Counter
        flag = 0;
        display_puts(old_y,old_x,"␣");//lösche vorherige
           Zeigerposition
        old_x = x;//Speicher neue x Position
        old_y = y;//Speicher neue y Position
    }
    if(Blink_ctr==0 && flag==0){//Init Status
        flag = 1;
        display_puts(y,x,data);//Zeichen wird angezeigt
    }else if(Blink_ctr == 3 && flag == 1){//Nach 1,5s
        flag = 0;
        display_puts(y,x,"␣");//wird das Zeichen wieder gelöscht
    }else if(Blink_ctr > 3){//Nach 4s
        Blink_ctr = 0;//wird der Blink Counter zurück gesetzt
    }
}
```

D.4. various.c

```
/************************************************************
Firma: DESY Hamburg
Autor: Joshua Supra
Projekt: ZMX-Handsteuerung Serie 3
Datei: various.c
Letzte Aenderung: 30.08.2018
Kurzbeschreibung: Diverse Funktionen
(c) Copyright 2018 DESY
************************************************************/
#include "stm32f4xx_conf.h"
#include "various.h"

/************************************************************
 * TIMER 2 Initialisierung (84MHz clock), 4Hz = 84MHz/[(4999+1)
    *(4199+1)]
 * Update alle 250ms, Interrupt
************************************************************/
void Timer2_Init(void){
    TIM_TimeBaseInitTypeDef TIM_TimeBaseInitStruct;
    NVIC_InitTypeDef NVIC_InitStruct;
```

```
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);

    TIM_TimeBaseInitStruct.TIM_Prescaler = 4999;
    TIM_TimeBaseInitStruct.TIM_Period = 4199;
    TIM_TimeBaseInitStruct.TIM_ClockDivision = TIM_CKD_DIV1;
    TIM_TimeBaseInitStruct.TIM_CounterMode = TIM_CounterMode_Up;

    TIM_TimeBaseInit(TIM2, &TIM_TimeBaseInitStruct);

    TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE);

    TIM_Cmd(TIM2, ENABLE);

    NVIC_InitStruct.NVIC_IRQChannel = TIM2_IRQn;
    NVIC_InitStruct.NVIC_IRQChannelPreemptionPriority = 4;
    NVIC_InitStruct.NVIC_IRQChannelSubPriority = 0;
    NVIC_InitStruct.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStruct);
}
/***********************************************************
 * UART5 putc
***********************************************************/
void uart5_putc(unsigned char c){
    while(USART_GetFlagStatus(UART5, USART_FLAG_TXE)== RESET);
    USART_SendData(UART5, c);
}
/***********************************************************
 * UART5 puts
***********************************************************/
void uart5_puts(const char *s){
    while(*s){
        uart5_putc(*s);
        s++;
    }
}
/***********************************************************
 * UART 5 Initialization
***********************************************************/
void UART5_Init(void){
    USART_DeInit(UART5);
    USART_InitTypeDef UART5_init;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_UART5, ENABLE);
```



```
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);

    GPIO_InitTypeDef GPIO;

    GPIO.GPIO_Mode=GPIO_Mode_AF;
    GPIO.GPIO_OType=GPIO_OType_PP;
    GPIO.GPIO_Pin=GPIO_Pin_12;
    GPIO.GPIO_PuPd=GPIO_PuPd_UP;
    GPIO.GPIO_Speed=GPIO_Speed_50MHz;
    GPIO_Init(GPIOC, &GPIO);

    GPIO_PinAFConfig(GPIOC, GPIO_PinSource12, GPIO_AF_UART5);

    UART5_init.USART_BaudRate =256000;
    UART5_init.USART_WordLength=USART_WordLength_8b;
    UART5_init.USART_StopBits=USART_StopBits_1;
    UART5_init.USART_Parity=USART_Parity_No;
    UART5_init.USART_Mode=USART_Mode_Tx;
    UART5_init.USART_HardwareFlowControl=
       USART_HardwareFlowControl_None;
    USART_Init(UART5, &UART5_init);

    USART_Cmd(UART5, ENABLE);
}
/***********************************************************
 * UART 5 Initialization
***********************************************************/
void gpio_UART5_Init(void){
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);

    GPIO_InitTypeDef GPIO;

    GPIO.GPIO_Mode=GPIO_Mode_OUT;
    GPIO.GPIO_Pin=GPIO_OType_PP;
    GPIO.GPIO_Pin=GPIO_Pin_2;
    GPIO.GPIO_PuPd=GPIO_PuPd_UP;
    GPIO.GPIO_Speed=GPIO_Speed_100MHz;
    GPIO_Init(GPIOD, &GPIO);
}

/*
   ***********************************************************************

 * Unsigned Integer to ASCII Wandlung
```

```
 *      u: Wert der gewandelt werden soll
 *      Buffer: Speicher für die ASCII Zeichen
 *      return: Anzahl der Zeichen
*************************************************************************
   */
char ItoA_unsigned(uint32_t u, char* Buffer ){
    int i = 0;
     do {
       Buffer[i++] = '0' + u % 10;        // Den Restwert aus '0'+
          u/10 brechnen
       u /= 10;                        // U druch 10 teilen.
     } while( u > 0 );                  // Solange U ist größer
        0 wird die funktion wiederhollt.
     Buffer[i]=ASCII_Spiegeln(Buffer,i);// ASCII wert spiegeln

     return i;
     }
/*
   *************************************************************************

 * Signed Integer to ASCII Wandlung
 *      u: Wert der gewandelt werden soll
 *      Buffer: Speicher für die ASCII Zeichen
 *      return: Anzahl der Zeichen
*************************************************************************
   */
char ItoA_signed(int32_t u, char* Buffer){
    int i = 0;
    uint8_t sign = 0;
    if (u>0){
        Buffer[0] = '+';                // Wenn u größer als 0 ist
            wird im Buffer ein + vor u erstellt
        Buffer++;
        sign = 1;
        }
    if (u<0){
        Buffer[0] = '-';                // Wenn u kleiner als 0
           ist wird im Buffer ein - vor u erstellt
        Buffer++;
        u = (u*-1);
        sign= 1;
        }
    do  {
```



```
        Buffer[i++] = '0' + u % 10; // Den Restwert aus '0'+u/10
           brechnen
        u /= 10;                         // U druch 10 teilen.
        } while( u > 0 );                // Solange U ist größer 0
           wird die funktion wiederhollt.

    Buffer[i]=ASCII_Spiegeln(Buffer,i);// ASCII wert spiegeln
    if(sign==1){
        return i+1;
    }else{
        return i;
    }
}
/*
   *************************************************************************

 * ASCII String spiegeln
 *      Buffer: Wert der gespiegelt wird
 *      i: Anzahl der Zeichen
 *      return: Rückgabe der Zeichen
*************************************************************************
   */
char ASCII_Spiegeln(char* Buffer,int i){

   int j=0;
   char tmp=0;

     for( j = 0; j < i / 2; ++j ) {        // den String in sich
        spiegeln
       tmp = Buffer[j];
       Buffer[j] = Buffer[i-j-1];
       Buffer[i-j-1] = tmp;
     }
     Buffer[i] = '\0';                    //einfügen String ende
     return Buffer[i];
     }
/*
   *************************************************************************

 * Genzwert für Variablen
 *      Wert: Wert der begrenzt wird
 *      min: Wert Minimum
 *      max: Wert Maximum
 *      return: Wert begrenzt
```

```
/**********************************************************************
   */
int32_t Grenzwert(int32_t Wert,int32_t min,int32_t max){
    if (Wert>=max){//Wenn der Wert größer als das Maximum
        Wert=max;//setze Wert = Maxium.
        }
    if (Wert<=min){//Wenn der Wert Kleiner als das Minimum
        Wert=min;//setze Wert=  Minimum
        }
    return Wert;
    }
/***********************************************************
 * Hex to ASCII Wandlung
 *      HEX_Wert: Wert der gewandelt werden soll
 *      return: ASCII Wert
***********************************************************/
char HtoA(char HEX_Wert){
    char ASCII_Wert;
    if(HEX_Wert <= 9){// Von Hex in ASCII Zeichen umwandlung
        ASCII_Wert = HEX_Wert +0x30;// Wenn H wert kleine/gleich 9
            ist mit 0x30 addieren
        }
    else{
        ASCII_Wert = HEX_Wert +0x37;// Wenn H wert größer als 9
           ist mit 0x37addieren
        }
    return ASCII_Wert;
    }
/***********************************************************
Debug UART 5 (putty)
***********************************************************/
void debug(char* strn, int value){
    char buffer[10] = "0";
    ItoA_unsigned(value, buffer);
    uart5_puts(strn);
    uart5_puts(buffer);
    uart5_putc('\r');uart5_putc('\n');
}
```

D.5. spi_fpga.c

```
/***********************************************************
Firma: DESY Hamburg
```



```
Autor: Joshua Supra
Projekt: ZMX-Handsteuerung Serie 3
Datei: spi_fpga.c
Letzte Aenderung: 30.08.2018
Kurzbeschreibung: Initialiserung der Kommunikation mit dem FPGA
   über SPI,
Funktionen zum Senden und Empfangen von Daten
(c) Copyright 2018 DESY
***********************************************************/
#include "stm32f4xx.h"
#include "spi_fpga.h"
#include "various.h"
/***********************************************************
 * Initialisierung: SPI2, Frequenz: 5,25MHz
 *      PIN B10: SCLK
 *      PIN A12: CS
 *      PIN C2: MISO
 *      PIN C3: MOSI
***********************************************************/
void init_fpga_SPI2(void){
    GPIO_InitTypeDef GPIO_InitStruct;
    SPI_InitTypeDef SPI_InitStruct;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI2, ENABLE);// Enable
       Clock für SPI2

    SPI_InitStruct.SPI_Direction = SPI_Direction_2Lines_FullDuplex
       ; // FUll Duplex, MISO, MOSI
    SPI_InitStruct.SPI_Mode = SPI_Mode_Master;     //
       Microkontroller ist Master
    SPI_InitStruct.SPI_DataSize = SPI_DataSize_8b; // Paketgrösse
       = 8 Bit
    SPI_InitStruct.SPI_CPOL = SPI_CPOL_Low;        // Clock high
       wenn idle
    SPI_InitStruct.SPI_CPHA = SPI_CPHA_2Edge;      // Daten werden
        mit 2 Flanke übernommen
    SPI_InitStruct.SPI_NSS = SPI_NSS_Soft |
       SPI_NSSInternalSoft_Set; // Soft NSS
    SPI_InitStruct.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_8
       ; // SPI Frequenz ist 5.25MHZ
    SPI_InitStruct.SPI_FirstBit = SPI_FirstBit_MSB;// LSB wird
       zuerst übertragen
    SPI_Init(SPI2, &SPI_InitStruct);
    // Enable Clock für GPIOs
```

```
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);

    GPIO_InitStruct.GPIO_Pin = GPIO_Pin_10;
    GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOB, &GPIO_InitStruct);

    GPIO_InitStruct.GPIO_Pin = GPIO_Pin_2 | GPIO_Pin_3;
    GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOC, &GPIO_InitStruct);

    // Konfiguriere NSS Pin, wird über Software realisiert
    GPIO_InitStruct.GPIO_Pin = GPIO_Pin_12;
    GPIO_InitStruct.GPIO_Mode = GPIO_Mode_OUT;
    GPIO_InitStruct.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStruct.GPIO_PuPd = GPIO_PuPd_UP;
    GPIO_Init(GPIOB, &GPIO_InitStruct);
    // Aktiviere AF an GPIOB Pin 10, GPIOC Pin 2 und 3
    GPIO_PinAFConfig(GPIOB, GPIO_PinSource10, GPIO_AF_SPI2);
    GPIO_PinAFConfig(GPIOC, GPIO_PinSource2, GPIO_AF_SPI2);
    GPIO_PinAFConfig(GPIOC, GPIO_PinSource3, GPIO_AF_SPI2);

    SPI_Cmd(SPI2, ENABLE);
}
/*************************************************************
 * TIMER 4 Initialisierung (84MHz clock), 50Hz := 84MHz/[(3359+1)
   *(499+1)]
 * Update alle 20ms, Interrupt
*************************************************************/
void Timer4_Init(void){
    TIM_TimeBaseInitTypeDef TIM_TimeBaseInitStruct;
    NVIC_InitTypeDef NVIC_InitStruct;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4, ENABLE);

    TIM_TimeBaseInitStruct.TIM_Prescaler = 3359;
    TIM_TimeBaseInitStruct.TIM_Period = 499;
    TIM_TimeBaseInitStruct.TIM_ClockDivision = TIM_CKD_DIV1;
    TIM_TimeBaseInitStruct.TIM_CounterMode = TIM_CounterMode_Up;
    TIM_TimeBaseInit(TIM4, &TIM_TimeBaseInitStruct);
```



```
    TIM_ITConfig(TIM4, TIM_IT_Update, ENABLE);

    TIM_Cmd(TIM4, ENABLE);

    NVIC_InitStruct.NVIC_IRQChannel = TIM4_IRQn;
    NVIC_InitStruct.NVIC_IRQChannelPreemptionPriority = 2;
    NVIC_InitStruct.NVIC_IRQChannelSubPriority = 1;
    NVIC_InitStruct.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStruct);
}
/*************************************************************
 * Senden von Daten an FPGA insgesamt 48 Takte, R/W (1 Bit, MSB),
   Adresse
 * (7 Bit, MSB +1 bis MSB+7) und Daten (Letzten 32 Bit)
 *      rw: Lese/Schreib-Anweisung: R=0, W=1
 *      addr: Registeradresse im FPGA
 *      data: Daten die gesendet werden sollen
*************************************************************/
signed long int send_FPGA(uint8_t rw, uint8_t addr, uint32_t data)
   {
    CS_FPGA_LOW;//Setze CS low
    signed long int buffer = 0; // init Buffer
    //Zerlege 32 Bit Daten in 4 Byte
    uint8_t a = data & 0xFF; // 7 downto 0
    uint8_t b = (data >> 8) & 0xFF;// 15 downto 8
    uint8_t c = (data >> 16) & 0xFF;// 23 downto 16
    uint8_t d = (data >> 24) & 0xFF;// 31 downto 24
    if(rw==1){
        SPI2_send(addr + 0x80);// wenn W setze MSB high und sende
           mit Adresse
    }
    else{
        SPI2_send(addr);// ansonsten MSB low und Adresse
    }
    SPI2_send(0);// Dummy Byte um FPGA zu ermöglichen Daten auf
       MISO zu legen
    buffer = SPI2_send(d); // sende 31 downto 24 und schreibe
       empfangene Daten in Buffer
    buffer = (buffer << 8 ) | SPI2_send(c);// sende 23 downto 16
       und schreibe empfangene Daten in Buffer
    buffer = (buffer << 8 ) | SPI2_send(b);// sende 15 downto 8
       und schreibe empfangene Daten in Buffer
    buffer = (buffer << 8 ) | SPI2_send(a);// sende 7 downto 0 und
        schreibe empfangene Daten in Buffer
```

```
    CS_FPGA_HIGH;// Setze CS high
    return buffer;// gebe empfangene Daten zurÃ¼ck
}
/**********************************************************
 * Senden von Char an FPGA, mit gleichzeitiger aufnahme der Daten
    vom MISO,
 * return empfangenen Char
 *      data: Daten die gesendet werden sollen
**********************************************************/
uint8_t SPI2_send(unsigned char data){
    SPI_I2S_SendData(SPI2, data);
    while( !(SPI2->SR & SPI_I2S_FLAG_TXE) ); // warte bis Senden
       vollständig
    while( !(SPI2->SR & SPI_I2S_FLAG_RXNE) ); // warte bis
       Empfangen vollständig
    return SPI_I2S_ReceiveData(SPI2);// gebe empfangene Daten
       zurück
}
/************************************************************
 * Berechnung des DDS Zähler Wert
 *      f: Frequenz die erzeugt werden soll
 *      return: Rückgabe Zählerwert fÃ¼r DDS
************************************************************/
uint32_t calc_DDS_count(uint32_t f){
    float count_temp = 0;
    count_temp =(f*DDS_K)+0.5;//Berechne Wert, +0.5 für korrekte
       Rundung
    return count_temp;
}
```

D.6. I2C.c

```
/**********************************************************
Firma: DESY Hamburg
Autor: Joshua Supra
Projekt: ZMX-Handsteuerung Serie 3
Datei: I2C.c
Letzte Aenderung: 30.08.2018
Kurzbeschreibung: Initialiserung fuer das digitale Potentiometer
   zum Einstellen
des Kontrasts vom Display
(c) Copyright 2018 DESY
**********************************************************/
```



```
#include "stm32f4xx_conf.h"
#include "I2C.h"
#include "various.h"
/**********************************************************
 * Setze den Wert des Potentiometers
 *      wert: Poti Wert 0-255
**********************************************************/
void Set_Display_contrast(uint8_t wert){
    uint8_t addr = 0b00101111; //Adresse des Poti
    while(I2C_GetFlagStatus(I2C1, I2C_FLAG_BUSY));
    I2C_GenerateSTART(I2C1, ENABLE);//erzeuge Startbedingung
    while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT));//
       warte auf Ack Start vom Poti
    I2C_Send7bitAddress(I2C1, addr << 1, I2C_Direction_Transmitter
       );// Sende Adresse
    while(!I2C_CheckEvent(I2C1,
       I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED));// warte auf
       Ack Addresse vom Poti
    I2C_Write(0b00111111);
    I2C_Write(wert);// schreibe Poti Wert
    I2C_GenerateSTOP(I2C1, ENABLE);//erzeuge Stoppbedingung
}
/**********************************************************
 * Sende Byte Ã¼ber I2C
 *      data: Byte das Ã¼ber I2C gesendet werden soll
**********************************************************/
void I2C_Write(uint8_t data){
    I2C_SendData(I2C1, data);
    while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_BYTE_TRANSMITTED)
       );//Warte bis Byte Ã¼bertragen
}
/**********************************************************
 * Initialisierung I2C fÃ¼r Diplay Kontrast Potentiometer
 * Parameter:
 *      Clock : 50kHz
 *      Adresse: 7 Bit
 *      Ack:    Adresse

 *      PIN B6     SCL
 *      PIN B7     SDA
**********************************************************/
void init_I2C(void){
    //GPIO und I2C Typedefs
    GPIO_InitTypeDef GPIO_InitStruct;
```

```
    I2C_InitTypeDef I2C_InitStruct;

    //Enable CLock fÃ¼r GPIOB und I2C1
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_I2C1, ENABLE);
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);

    //Konfiguriere PINs fÃ¼r I2C
    GPIO_InitStruct.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7;
    GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStruct.GPIO_Speed = GPIO_Speed_2MHz;
    GPIO_InitStruct.GPIO_OType = GPIO_OType_OD;
    GPIO_InitStruct.GPIO_PuPd = GPIO_PuPd_UP;
    GPIO_Init(GPIOB, &GPIO_InitStruct);

    // Enable alternate Function
    GPIO_PinAFConfig(GPIOB, GPIO_PinSource6, GPIO_AF_I2C2);
    GPIO_PinAFConfig(GPIOB, GPIO_PinSource7, GPIO_AF_I2C2);

    //Konfiguriere I2C
    I2C_InitStruct.I2C_ClockSpeed = 50000;//I2C clock 50kHz
    I2C_InitStruct.I2C_Mode = I2C_Mode_I2C;//I2C Modus
    I2C_InitStruct.I2C_DutyCycle = I2C_DutyCycle_2;
    I2C_InitStruct.I2C_OwnAddress1 = 0x00;//Eigene Adresse 0x00
    I2C_InitStruct.I2C_Ack = I2C_Ack_Disable;//Kein Data Ack
    I2C_InitStruct.I2C_AcknowledgedAddress =
       I2C_AcknowledgedAddress_7bit;

    I2C_Init(I2C1, &I2C_InitStruct);

    I2C_Cmd(I2C1, ENABLE);
}
```

D.7. gpio.c

```
/************************************************************
Firma: DESY Hamburg
Autor: Joshua Supra
Projekt: ZMX-Handsteuerung Serie 3
Datei: gpio.c
Letzte Aenderung: 30.08.2018
Kurzbeschreibung: Initialiserung der GPIOs fÃ¼r Taster und LEDs
(c) Copyright 2018 DESY
************************************************************/
```



```
#include "stm32f4xx.h"
#include "gpio.h"

/************************************************************
 * Initialisierung der GPIOs fÃ¼r Taster und Leds
 *      PIN A8      LEDCW_1             C8      LED_ZMX_1
 *      PIN A9      LEDCW_2             C9      LED_ZMX_2
 *      PIN A10     LEDCCW_1
 *      PIN A11     LEDCCW_2
 *
 *      PIN D8      Taster Menu         D12     Taster CCW
 *      PIN D9      Taster P-Reset      D13     Taster Stop
 *      PIN D10     Taster Entregen     D14     Taster Reserve
 *      PIN D11     Taster CW
 *
 *      PIN C5      Taster Encoder Menu
 *      PIN E12     Taster Encoder Frequenz
************************************************************/
void init_GPIOS(void){
    GPIO_InitTypeDef GPIO;

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);//Enable
       Clock GPIO bank A
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);//Enable
       Clock GPIO bank C
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);//Enable
       Clock GPIO bank D
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOE, ENABLE);//Enable
       Clock GPIO bank D
/*---------------  LED Limits CW & CCW ---------------*/
    GPIO.GPIO_Mode=GPIO_Mode_OUT;
    GPIO.GPIO_Pin=GPIO_OType_PP;
    GPIO.GPIO_Pin=GPIO_Pin_8 | GPIO_Pin_9 | GPIO_Pin_10 |
       GPIO_Pin_11;
    GPIO.GPIO_PuPd=GPIO_PuPd_UP;
    GPIO.GPIO_Speed=GPIO_Speed_2MHz;
    GPIO_Init(GPIOA, &GPIO);

    GPIO_ResetBits(GPIOA, GPIO_Pin_9);
    GPIO_ResetBits(GPIOA, GPIO_Pin_11);
/*---------------  LED ZMX ON ---------------*/
    GPIO.GPIO_Mode=GPIO_Mode_OUT;
    GPIO.GPIO_Pin=GPIO_OType_PP;
    GPIO.GPIO_Pin=GPIO_Pin_8 | GPIO_Pin_9;
```

```
    GPIO.GPIO_PuPd=GPIO_PuPd_UP;
    GPIO.GPIO_Speed=GPIO_Speed_2MHz;
    GPIO_Init(GPIOC, &GPIO);
/*---------------  Input Buttons ---------------*/
    GPIO.GPIO_Mode=GPIO_Mode_IN;
    GPIO.GPIO_Pin=GPIO_OType_PP;
    GPIO.GPIO_Pin=GPIO_Pin_8 | GPIO_Pin_9 | GPIO_Pin_10 |
       GPIO_Pin_11 | GPIO_Pin_12 | GPIO_Pin_13 | GPIO_Pin_14;
    GPIO.GPIO_PuPd=GPIO_PuPd_UP;
    GPIO.GPIO_Speed=GPIO_Speed_2MHz;
    GPIO_Init(GPIOD, &GPIO);

    GPIO.GPIO_Mode=GPIO_Mode_IN;
    GPIO.GPIO_Pin=GPIO_OType_PP;
    GPIO.GPIO_Pin=GPIO_Pin_5;
    GPIO.GPIO_PuPd=GPIO_PuPd_NOPULL;
    GPIO.GPIO_Speed=GPIO_Speed_2MHz;
    GPIO_Init(GPIOC, &GPIO);

    GPIO.GPIO_Mode=GPIO_Mode_IN;
    GPIO.GPIO_Pin=GPIO_OType_PP;
    GPIO.GPIO_Pin=GPIO_Pin_12;
    GPIO.GPIO_PuPd=GPIO_PuPd_NOPULL;
    GPIO.GPIO_Speed=GPIO_Speed_2MHz;
    GPIO_Init(GPIOE, &GPIO);
}
```

D.8. ZMX.c

```
/*************************************************************
Firma: DESY Hamburg
Autor: Joshua Supra
Projekt: ZMX-Handsteuerung Serie 3
Datei: ZMX.c
Letzte Aenderung: 30.08.2018
Kurzbeschreibung: Initialiserung der Kommunikation mit der ZMX-
   Endstufe über
UART, Funktionen zum Senden und Empfangen von Daten
(c) Copyright 2018 DESY
*************************************************************/
#include "stm32f4xx.h"
#include "ZMX.h"
#include "various.h"
```



```
#include "main.h"


/*************************************************************
 * UART 4 Initialisierung (ZMX Kommunikation)
 * Parameter:
 *      Baudrate : 57000
 *      8 Bits/Byte
 *      1 Stoppbit
 *      1 Paritybit
 *  PIN A0      TX
 *  PIN A1      RX
*************************************************************/
void init_uart_ZMX(void){
    USART_InitTypeDef UART4_init;
    NVIC_InitTypeDef NVIC_InitStructure;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_UART4, ENABLE);//
       aktiviere Clock an Port A
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);//
       aktiviere Clock an UART4

    GPIO_InitTypeDef GPIO;

    GPIO.GPIO_Pin=GPIO_Pin_0 | GPIO_Pin_1;//wähle Pin A0=TX und A1
       =RX aus,
    GPIO.GPIO_Mode=GPIO_Mode_AF;//aktiviere alternative Funktion
    GPIO.GPIO_OType=GPIO_OType_PP;//Pins im Push-Pull Modus
    GPIO.GPIO_PuPd=GPIO_PuPd_UP;//aktiviere internen Pull up
    GPIO.GPIO_Speed=GPIO_Speed_50MHz;// Clock an den Pins 50MHz

    /*initialisiere Pins*/
    GPIO_Init(GPIOA, &GPIO);

    GPIO_PinAFConfig(GPIOA, GPIO_PinSource0, GPIO_AF_UART4);//
       Konfiguriere A0 als UART
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource1, GPIO_AF_UART4);//
       Konfiguriere A1 als UART
    UART4_init.USART_BaudRate =57600;
    UART4_init.USART_WordLength=USART_WordLength_8b;
    UART4_init.USART_StopBits=USART_StopBits_1;
    UART4_init.USART_Parity=USART_Parity_No;
    UART4_init.USART_Mode=USART_Mode_Tx | USART_Mode_Rx;//Duplex
       betrieb
```

```
    UART4_init.USART_HardwareFlowControl=
       USART_HardwareFlowControl_None;
    USART_Init(UART4, &UART4_init);// initialisiere UART4

    USART_ITConfig(UART4, USART_IT_RXNE, ENABLE);// aktiviere
       UART4 RX Interrupt

    /*Konfiguriere Interrupt für UART4  RX*/
    NVIC_InitStructure.NVIC_IRQChannel = UART4_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 2;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);

    /*aktiviere UART4*/
    USART_Cmd(UART4, ENABLE);
}
/************************************************************
 * sende Char über UART4 an ZMX
 *      data: Character der übertragen wird
************************************************************/
void uart4_putc(unsigned char data){
while(USART_GetFlagStatus(UART4, USART_FLAG_TXE)== RESET);//Warte
   bis Senderegister leer
    USART_SendData(UART4, data);//sende Daten
}
/************************************************************
 * sende String über UART4 an ZMX
 *      data: String der übertragen wird
************************************************************/
void uart4_puts(char *data){
    while(*data){//Solange das Array nicht leer ist
        uart4_putc(*data);//sende das jeweilige Array Elemente
        data++;//erhöhe Adresse
    }
}
/************************************************************
 * Befehl mit Wert an die ZMX-Endstufe senden
 *      Befehl: Befehl der gesendet werden soll
 *      wert_binaer: Wert des Befehls (z.B. Laufstrom in A)
************************************************************/
void send_ZMX(char *Befehl, uint16_t wert_binaer){
    char ZMX_wert_ASCII[4];// ZMX_wert_ASCII(0|0|0|String Ende)
    uint8_t Pruefsumme=0;// variable für die Prüfsumme
```



```
    uint8_t csh=0;// variable für High Nibble der Prüfsumme
    uint8_t csl=0;// variable für low Nibble der Prüfsumme
    uint8_t k=0;// Lauf variable für die Prüfsummmen berechnung
    uint8_t i = 0;// Lauf variable für das umwandel von Binär in
       ASCHII
    uint8_t l=0;
/*Den zu senden Wert von Binär zu ASCII umwandel*/
    do {
        ZMX_wert_ASCII[i++] = '0' + wert_binaer % 10;
        wert_binaer /= 10;// U durch 10 teilen.
        } while( wert_binaer > 0 );// Solange U ist größer 0 wird
           die funktion wiederhollt
    ZMX_wert_ASCII[i]=ASCII_Spiegeln(ZMX_wert_ASCII,i);// ASCII
       wert spiegeln
/*Prüfsummenberechnung*/
    while (ZMX_wert_ASCII[k]){// Solange druchlaufen bis der Sting
        zuende ist
        Pruefsumme=(Pruefsumme^ZMX_wert_ASCII[k++]);// Jeder
           Stelle vom Sting (ver)Xor und auf Prüfsumme schreiben
        }
    Pruefsumme=(Pruefsumme^':'^ZMX_Adresse_H^ZMX_Adresse_L);//
       Prüfsumme mit ':' (Begrenzer) und dem Befele (ver)Xor
    while(*Befehl){
        Pruefsumme=Pruefsumme^*Befehl;
        Befehl++;
        l++;
        }
    Befehl=Befehl-l;
/*Prüfsummenberechnung von Binär in ASCII umwandeln*/
    csh=(Pruefsumme>>4);// Prüfsumme 4 Bits nach Rechts schieben
       um High Nibbel zu erzeugen
    csl=(Pruefsumme&0x0F);// Prüfsumme mit 0F (ver)AND um den Low
       Nibble zu erzeugen

    csh=HtoA(csh);// ASCII wert erstellen
    csl=HtoA(csl);// ASCII wert erstellen
/*Prüfsummenberechnung von Binär in ASCII umwandeln*/
    uart4_putc(0x02);// Start_Sring Senden
    uart4_putc(ZMX_Adresse_H);// ZMX_Adresse_H senden
    uart4_putc(ZMX_Adresse_L);// ZMX_Adresse_L senden
    while(*Befehl){// Befehl(e) Senden
        uart4_putc(*Befehl);
        Befehl++;
        }
```

```
    uart4_puts(ZMX_wert_ASCII);// Wert Senden
    uart4_putc(':');// Begrenzer Senden
    uart4_putc(csh);// Prüfsumme High Nibble Senden
    uart4_putc(csl);// Prüfsumme LOW Nibble Senden
    uart4_putc(0x03);// Stop_Sring Senden
    ZMX_Daten_gesendet=1;// ZMX_Daten_gesendet auf 1 setzen
}
/*************************************************************
 * ZMX Adresse herausfinden
*************************************************************/
void ZMX_get_address(void){
    static uint8_t adresse=0x30;//Startadresse
    static uint8_t adresse_ok=0;//Flag für Adresse gefunden
    while(adresse_ok== 0){//Solange die Adresse nicht gefunden
       wurde
        send_ZMX("U", 0);//wird versucht die ZMX-Endstufe zu
           aktiveren
        send_ZMX("B", 0);//und die Version auszulesen.
        if(ZMX_Daten_Empfangen[3]!= 'b'){//Wenn der Befehl nicht
           zurück gegeben wurde
            adresse++;//wird die Adresse Schrittweise erhöht.
            if(adresse>0xF){//Wenn die Adresse größer 15 ist
                adresse=0;//wid sie zurück gesetzt
            }
            ZMX_Adresse_L=HtoA(adresse);//Wandeln der Adresse in
               ASCII
            ZMX_Datensatz_Empfangen=0;
        }else if(ZMX_Datensatz_Empfangen==1 && ZMX_Daten_Empfangen
           [3]== 'b'){//Wenn Befehl erfolgreich empfangen wurde
            adresse_ok=1;//Adresse gefunden
            ZMX_Adresse_L=ZMX_Daten_Empfangen[2];//Die Adresse
               wird gespeichert
        }
    }
}
/*************************************************************
 * Interrupt-Handler für Empfang von Daten über UART4, ZMX-
    Endstufe
*************************************************************/
void UART4_IRQHandler(void){
    static uint8_t Start = 0;//Startbit Flag
    static uint8_t cnt = 0;//Datencounter
    static uint8_t data= 0;//Buffer für empfangene Daten
    /*Wenn Empfangsregister nicht leer ist*/
```



```
    if( USART_GetITStatus(UART4, USART_IT_RXNE) != RESET ){
        data = UART4->DR;//kopiere Daten aus Empfangsregister
        /*Wenn Startbit empfangen und keine Übertragung*/
        if(data == 0x02 && Start == 0){
            Start = 1;// Setze Startbit Flag
            }
        /*Wenn Beginn einer Übertragung*/
        if(Start == 1){
            ZMX_Daten_Empfangen[cnt++]= data;//schreibe empfangene
                Daten in Empfangsregister, inkrementiere
               Datencounter
            /*Wenn Ende der Übetragung und Datencounter nicht
               überlaufen*/
            if(data == 0x03 && cnt < 20){
                ZMX_Daten_Empfangen[cnt++]= '\0';//hänge \0 an das
                    Arrayende
                ZMX_Datensatz_Empfangen =1;//Setze Flag für
                   Datensatz empfangen
                Start =0;
                cnt =0;
            }
            /*Wenn Überlauf beende Übertragung*/
            else if(cnt == 20){
                cnt= 0;
                Start = 0;
            }
        }
    }
    USART_ClearITPendingBit (UART4,USART_IT_RXNE);//Lösche RX
       Interruptflag
}
```

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.

Ort, Datum	Unterschrift