HAW HAMBURG

Bachelorarbeit

Marcel Soika

Entwurf und Realisierung eines Teststands für Schrittmotorsteuerungen des Typs Phytron ZMX

Hochschule für Angewandte Wissenschaften Hamburg
Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Hamburg University of Applied Sciences
Faculty of Computer Science and Engineering
Department Information and Electrical Engineering

Marcel Soika

Entwurf und Realisierung eines Teststands für Schrittmotorsteuerungen des Typs Phytron ZMX

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Bachelorstudiengang Elektro- und Informationstechnik
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. rer. nat. Björn Lange
Zweitgutachter: Dipl. Ing. Julia Müller

Eingereicht am: 14.09.2021

Marcel Soika

Thema der Arbeit

Entwurf und Realisierung eines Teststands für Schrittmotorsteuerungen des Typs Phytron ZMX

Stichworte

ZMX-Überrahmen, Motorsteuerung, LabVIEW, Teststand, Schrittmotor, Arduino, Ansteuerung

Kurzzusammenfassung

Der ZMX-Überrahmen, mit dem sich bis zu 16 Schrittmotoren ansteuern lassen, läuft nicht immer fehlerfrei. Auf dieser Grundlage wird ein Teststand entwickelt, mit dem sich die Fehler leicht feststellen und klassifizieren lassen. Dies erleichtert die Fehlersuche und Reparatur. Zum Einsatz kommen mehrere Arduinos sowie LabVIEW für die Programmierung.

Marcel Soika

Title of Thesis

Design and implementation of a test stand for stepper motor controls of the Phytron ZMX type

Keywords

ZMX crate, motor control , LabVIEW, test stand, stepper motor, Arduino, control

Abstract

The ZMX crate, with which up to 16 stepper motors can be controlled, does not always run without errors. On this basis, a test stand is developed with which the errors can be easily identified and classified. This makes it easier to find and repair the errors. Several Arduinos and LabVIEW are used for programming.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Die ZMX-Überrahmen sind beim DESY entwickelte und eingesetzte 19-Zoll-Einschübe mit diversen integrierten Modulen. Eine genaue Beschreibung der einzelnen Module ist in dem Praktikumsbericht „Analyse und Ansteuerung einer ZMX-Motorsteuerung “ zu finden [6].
An einem ZMX-Überrahmen können bis zu 16 Schrittmotoren angeschlossen werden, welche beim DESY für die Justierung von Optiksystemen, z.B. Linsen und Spiegel, verwendet werden. Die Optiksysteme dienen der Fokussierung und Ablenkung des Elektronenstrahls in den Beschleunigern.

Bei einigen dieser Überrahmen treten jedoch hin und wieder Fehler auf, deren Ursachen sich nicht immer auf Anhieb feststellen lassen. Daher soll nun ein Teststand entwickelt werden, welcher die Möglichkeit bietet, diese Überrahmen in einem Langzeittest zu beobachten.

Hierbei sollen verschiedene Signale des Überrahmens kontinuierlich erfasst und mit den Soll-Werten verglichen werden. Im Falle eines auftretenden Fehlers soll das System selbständig abschalten und die Informationen der Signale protokollieren.

Für eine intuitive und optisch ansprechende Bedienung des zu entwickelnden Teststands wird das graphische Programmiersystem „LabVIEW“ von National Instruments verwendet [4].

In der folgenden Tabelle 1.1 sind alle für den Teststand verwendeten Soft- und Hardware Elemente aufgeführt.

Gerät	Bauteil/Baugruppe	Anzahl
ZMX-Überrahmen	Anschlussmodule	4
	Anschlussmodule	4
	Anzeige Endlagen	1
	ZMX-Endstufen	16
	Backplane Endstufen	1
	Backplane Anschlussmodule	1
	Schrittmotoren	16
	5 V Netzteil: DSP30-5	1
	15 V Netzteil: DSP60-15	1
	5 V Netzteil: DSP10-5	1
	60 V Netzteil: EXW 60.25/OP1	1
Entwickelte Testbox	Arduino Mega2560	2
	Arduino Nano	4
	Kondensator: 100 nF	8
	Widerstand: 390 Ω	16
	Widerstand: 3,3 kΩ	16
	Widerstand: 10 kΩ	1
	Optokoppler: HCPL2631	8
	ULN2803	1
	5 V Relais mit einem Schließerkontakt	8
	5 V Netzteil: VCE05US05	1
	9 V Netzteil: VCE10US09	1
	Kippschalter: Umschalter mit drei Kontakten	8
	Taster: Schließer mit zwei Kontakten	1
	Stufenschalter 4x2	1
Oszilloskop Tektronix TPS 2024	Tastköpfe	4
LabVIEW 2019	–	–

Tabelle 1.1: Verwendete Hard- und Software

Die folgenden Abbildungen 1.1 und 1.2 zeigen einen ZMX-Überrahmen sowie den zugehörigen Übersichtsplan. Eine genauere Beschreibung der integrierten Module ist dem vorangegangenen Praktikumsbericht des Praxissemesters zu entnehmen [6].

Abbildung 1.1: Abbildung eines ZMX-Überrahmens [3]

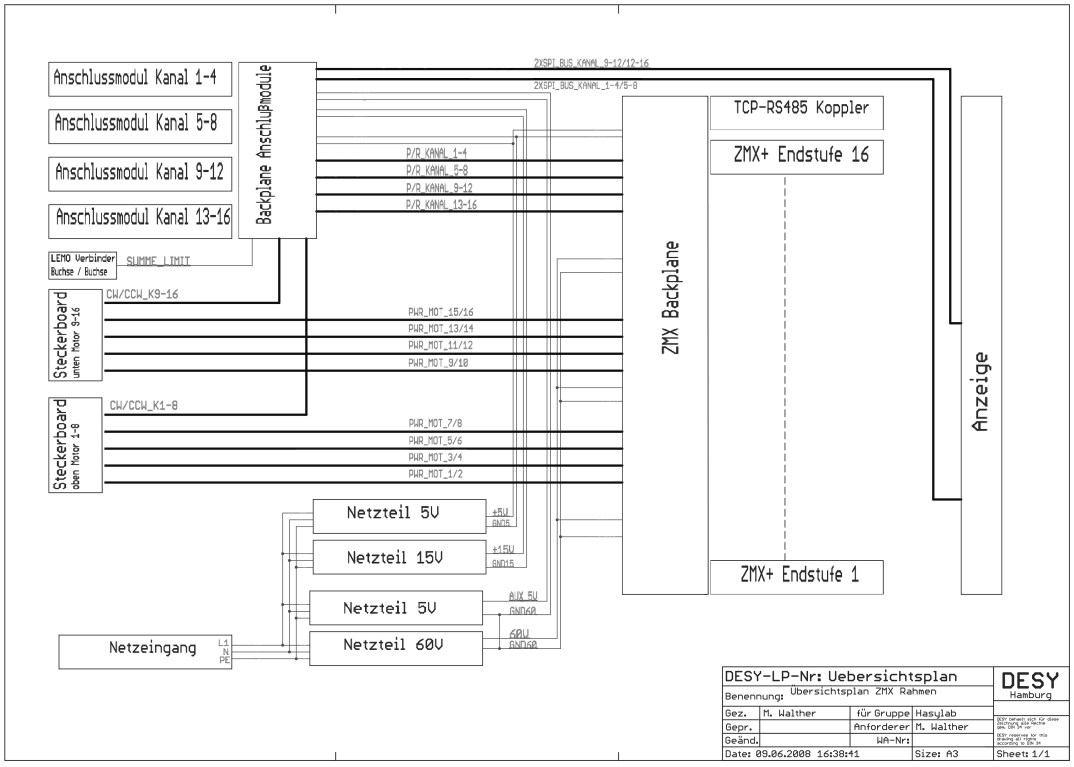


Abbildung 1.2: Übersichtsplan eines ZMX-Überrahmens [3]

2 Zielsetzung

Die Ansteuerung des ZMX-Überrahmens erfolgt über die vier integrierten Anschlussmodule mittels der Puls- und Richtungssignale. Hierzu werden die im Rahmen des Praxissemesters bereits entwickelten Programme verwendet [6].
Die Ansteuerung der am ZMX-Überrahmen angeschlossenen Schrittmotoren soll hierbei sowohl automatisch als auch manuell erfolgen können.
Die Bedienung des Teststands soll mit LabVIEW realisiert werden.
Da die Reihenfolge der Einschaltflanken zweier im ZMX-Überrahmen integrierter Netzteile in der Vergangenheit schon zu Fehlern geführt hat, soll es am Teststand die Möglichkeit geben, eben diese zu messen.
Die gemessenen Daten sollen anschließend tabellarisch protokolliert werden.
Um Spannungseinbrüche oder Schwankungen bei den vier im ZMX-Überrahmen verwendeten Netzteilen feststellen zu können, sollen diese ebenfalls während des Betriebs des Teststands gemessen werden. Im Fehlerfall sollen die Spannungswerte zu diesem Zeitpunkt tabellarisch gesichert werden.
Des Weiteren soll eine kontinuierliche Erfassung des Fehlersignals von allen 16 ZMX-Endstufen erfolgen, um auftretende Fehler innerhalb einer Endstufe erkennen zu können. Eine genauere Beschreibung des Fehlersignals folgt unter 3.1.
Zur Überprüfung der fehlerfreien Auswertung von den Signalen der Endlagenschalter der Schrittmotoren werden diese Schalter mittels Kippschaltern simuliert. Das jeweilige eingestellte Signal wird kontinuierlich mit dem zugehörigen Signal, ausgehend von den Anschlussmodulen, verglichen.
Des Weiteren sollen alle auftretenden Fehler sowie die zu diesem Zeitpunkt angesteuerten Motoren in einem Fehlerprotokoll gespeichert werden. Anschließend soll das System selbständig abschalten und signalisieren, dass beim Test Fehler aufgetreten sind.

3 Konzeption

3.1 Auswahl der zu überwachenden Signale

Für die Auswahl der zu überwachenden Signale sind verschiedene Aspekte ausschlaggebend. Zum Einen wird analysiert, welche Signale oder Werte sinnvoll für eine Fehlererkennung sind. Zum Anderen muss geprüft werden, welche Signale gut messbar sind und wo die Messung keine zusätzlichen Störungen verursacht.
Die Netzteilspannungen lassen sich ohne großen Aufwand beim Messaufbau erfassen und sind in jedem Fall sinnvoll zu überwachen, da Kurzschlüsse innerhalb des Überrahmens, die zu Spannungseinbrüchen führen können, erfasst werden. Außerdem ist es wichtig das Einschaltverhalten des 5 V Netzteils (DSP30-5) und des 15 V Netzteils (DSP60-15) zu überprüfen, da eine falsche Reihenfolge der Einschaltflanken in der Vergangenheit bereits zu fehlerhaftem Verhalten geführt hat.
Es ist ebenfalls wichtig, die Fehlersignale der ZMX-Endstufen zu überwachen. Ein Fehlersignal wird ausgelöst, wenn eine ZMX-Endstufe eine zu hohe Temperatur (höher als 90 °C), eine zu geringe Versorgungsspannung (weniger als 22 V) oder einen zu hohen Strom (größer als 30 A) erkannt hat [7]. Eine genaue Information welcher dieser Fehler erfasst wurde, liefert dieses Signal jedoch nicht. Diese Information lässt sich nur über den Service-Bus ermitteln. Die Fehlersignale sind mit geringem Aufwand in Form vom Adapter-Kabeln gut erreichbar.
Des Weiteren kann es zu Fehlern bei den Signalen der Endlagenschaltern der Schrittmotoren kommen. Die Anzeige dieser Signale wird von dem Modul „Anzeige Endlagen“ übernommen [6]. Diese Anzeige wird über einen SPI-Bus, ausgehend von den Anschlussmodulen, gesteuert. Da diese Busleitungen jedoch recht störanfällig sind, insbesondere wenn die Leitungen durch den Messaufbau deutlich verlängert werden müssen, ist eine direkte Überwachung des Bus-Protokolls nicht sinnvoll. Außerdem bedeutet eine Bus-Überwachung in diesem Fall auch einen erheblichen Entwicklungs- und Hardwareaufwand: Zum einen handelt es sich hierbei um vier separate SPI-Busse. Zum anderen arbeitet der SPI-Bus seriell und innerhalb verschiedener Module des Überrahmens werden jeweils weitere Signale, unter Verwendung serieller Bausteine wie z.B. 74HC165D, zu dem Bus hinzugefügt.
Somit ist eine einfache Signalerfassung direkt am Zugriffspunkt nicht möglich, es müsste ein direkter Abgriff am Mikrocontroller des Anschlussmoduls erfolgen, was technisch nur mit sehr hohem Aufwand zu realisieren ist.

Die Signale der Endlagenschalter sind jedoch auch über die SCSI-Buchsen der Anschlussmodule erreichbar, was einen erheblich geringen Aufwand bedeutet. Um die Richtigkeit dieser Signale kontinuierlich zu überprüfen und bestätigen, werden Endlagenschalter für vier Schrittmotoren in Form von Kippschaltern simuliert und das eingestellte Signal mit dem zugehörigen der Anschlussmodule verglichen.

Im Folgenden sind alle zu messenden Signale aufgelistet.

- Messung der Einschaltflanken der Netzteile DSP30-5 und DSP60-15
- kontinuierliche Spannungsmessung aller vier sich im ZMX-Überrahmen befindenden Netzteile
- Überwachung des Fehlersignals von allen 16 ZMX-Endstufen
- Überwachung der Endlagenschalter-Signale von vier wählbaren Schrittmotoren

3.2 Auswahl der Software für die grafische Bedienoberfläche des Teststands

Als Software für die grafische Bedienoberfläche des Teststands wird LabVIEW verwendet, welche zum DESY-internen Standard gehört. Die Programmierung in LabVIEW erfolgt in Blockdiagrammen und ist somit intuitiv zu verstehen und zu programmieren.

3.3 Auswahl der Hardware

Für die Überwachung der Netzteilspannungen des ZMX-Überrahmens bietet sich das Oszilloskop mit der Bezeichnung TPS 2024 des Herstellers Tektronix an [10]. Dieses verfügt über vier differentielle Messeingänge, was für diesen Anwendungsfall von Vorteil ist, da drei der vier Netzteile des ZMX-Überrahmens unterschiedliche Massebezüge haben. Somit lassen sich die Tastköpfe für die Messung vom Oszilloskop direkt an den Netzteilausgängen anschließen. Des Weiteren verfügt dieses Oszilloskop über eine RS232-Schnittstelle [8], worüber das Oszilloskop ferngesteuert und Werte ausgelesen werden können. Diese Funktion ermöglicht die Steuerung des Oszilloskops über LabVIEW.

Für die Überwachung und Auswertung der ausgewählten internen Signale des ZMX-Überrahmens wird wieder, wie auch für die Ansteuerung des Überrahmens, ein Entwicklungsboard des Typs Arduino Mega2560 verwendet. Die USB-Schnittstelle des Boards ermöglicht eine serielle Kommunikation mit LabVIEW. Die 54 I/O-Pins des sich darauf befindenden Mikrocontrollers vom Typ ATMEGA2560 sind ausreichend, um die 32 digitalen Eingangssignale, bestehend aus 16 Fehlersignalen sowie 16 Signale für die

Endlagenschalter (8 Signale von den Anschlussmodulen, 8 Signale von den Simulierten Endlagenschaltern), zu erfassen und auszuwerten [1].

3.4 Schematischer Testaufbau

Die nachfolgende Abbildung 3.1 zeigt das Blockschaltbild für den Teststand.
Bei der „Testbox“ handelt es sich um eine Eigenentwicklung, welche im Folgenden unter 4.3 genau beschrieben wird.

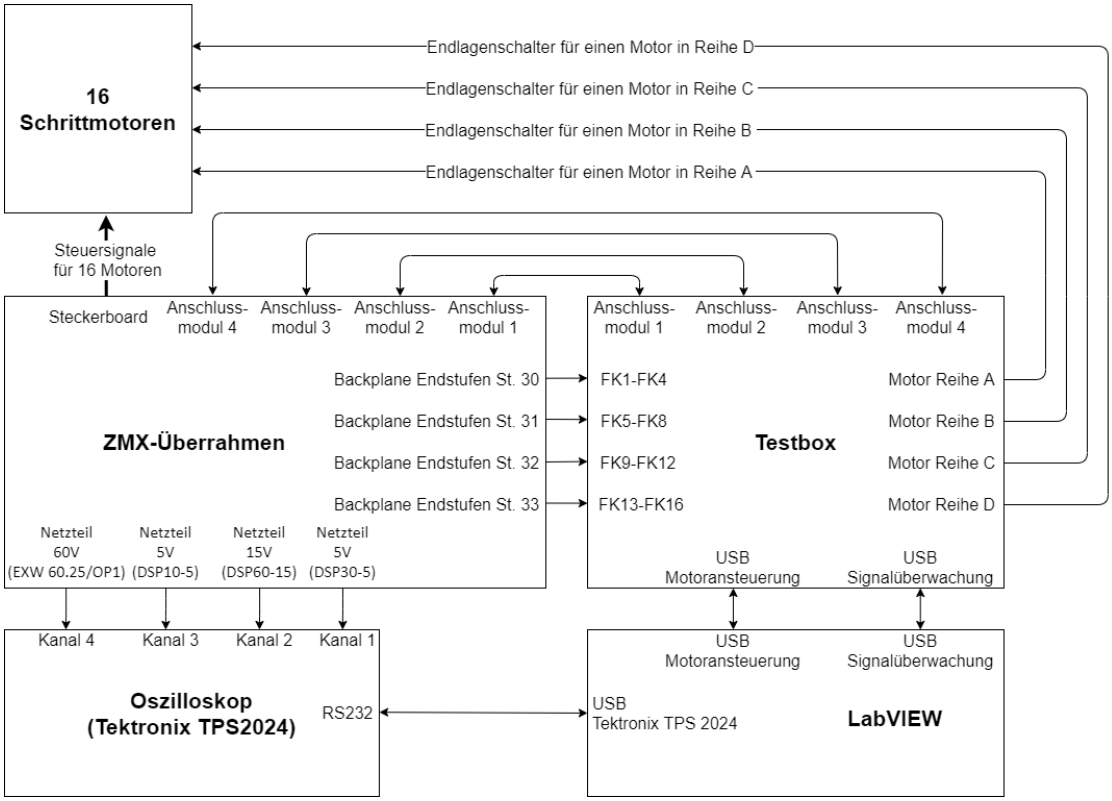


Abbildung 3.1: Blockschaltbild für den Aufbau des entwickelten Teststands

4 Technische Umsetzung

4.1 Ansteuerung der Schrittmotoren

Die folgenden Struktogramme 4.1, 4.2 und 4.3 zeigen die Funktionsweise der Ansteuerung der Schrittmotoren über die Anschlussmodule eines ZMX-Überrahmens. Diese Ansteuerung wurde bereits im Rahmen des vorangegangen Praxissemesters entwickelt [6].

In den Abbildung 4.1 und 4.2 ist die Funktion des I2C-Masters dargestellt. Dieser übernimmt zum einen die Aufgabe, die Informationen für die Motorgeschwindigkeiten an die vier am I2C-Bus angeschlossenen Slaves weiterzugeben, zum anderen die serielle Kommunikation über USB mit LabVIEW.

Der Betriebsmodus lässt sich zwischen „automatik_1“, „automatik_2“ und „manuell“ wählen. Da die Schrittmotoren im realen Betrieb „zufällig“ eingeschaltet werden, ist hierbei zwischen zwei Automatik-Modi wählbar, die dieses Verhalten simulieren.
In dem Modus „automatik_1“ erfolgt die Ansteuerung der Schrittmotoren nach dem Muster des binären Zahlensystems. Alle Motoren, bei denen eine 1 gesetzt ist, fahren zeitgleich an. Je nachdem, welchen Wert die Variable „max_motoren“ annimmt (1 bis 8), gibt es $2^{\text{max\_motoren}}$ mögliche Kombinationen der angesteuerten Schrittmotoren.

In dem Modus „automatik_2“ erfolgt die Ansteuerung der Schrittmotoren über mathematische Permutationsmuster. Hierbei werden, abhängig von der Variable „max_motoren“, genau so viele Motoren angesteuert. Die Motoren werden sequenziell zugeschaltet und in umgekehrter Reihenfolge wieder ausgeschaltet.

In dem manuellen Betriebsmodus lassen sich die Motoren in beliebiger Kombination nacheinander oder gemeinsam ansteuern.

Der zugehörige Quellcode ist im Anhang unter A.1.1 zu finden.

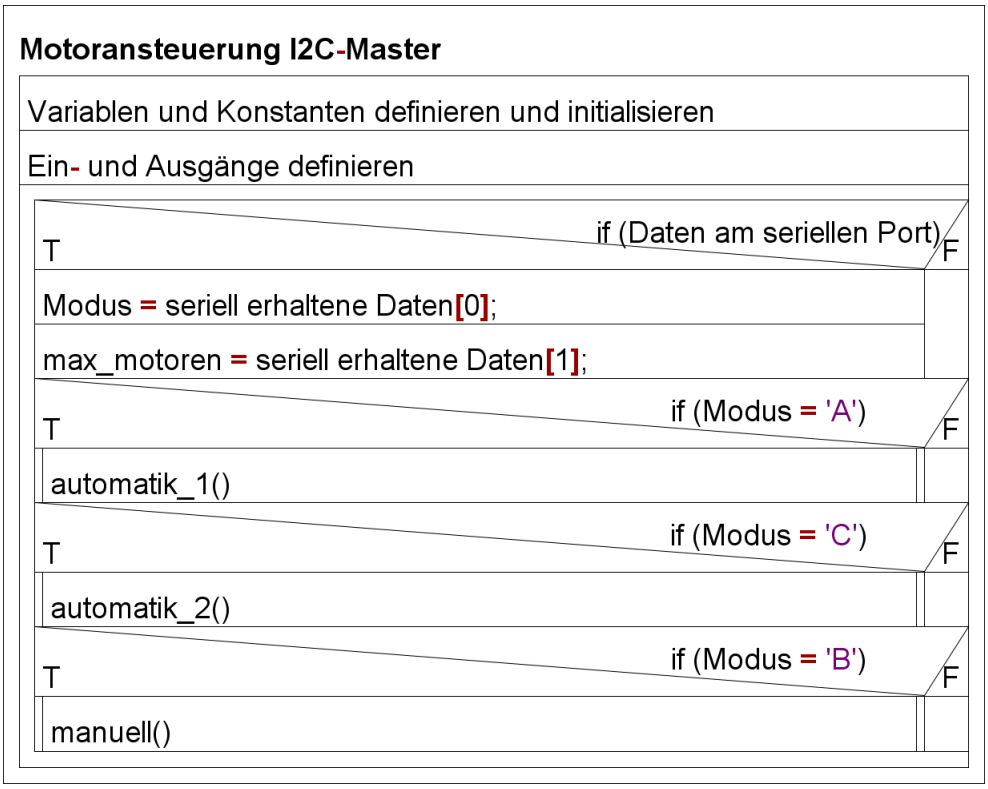


Abbildung 4.1: Struktogramm des Programms Motoransteuerung I2C-Master

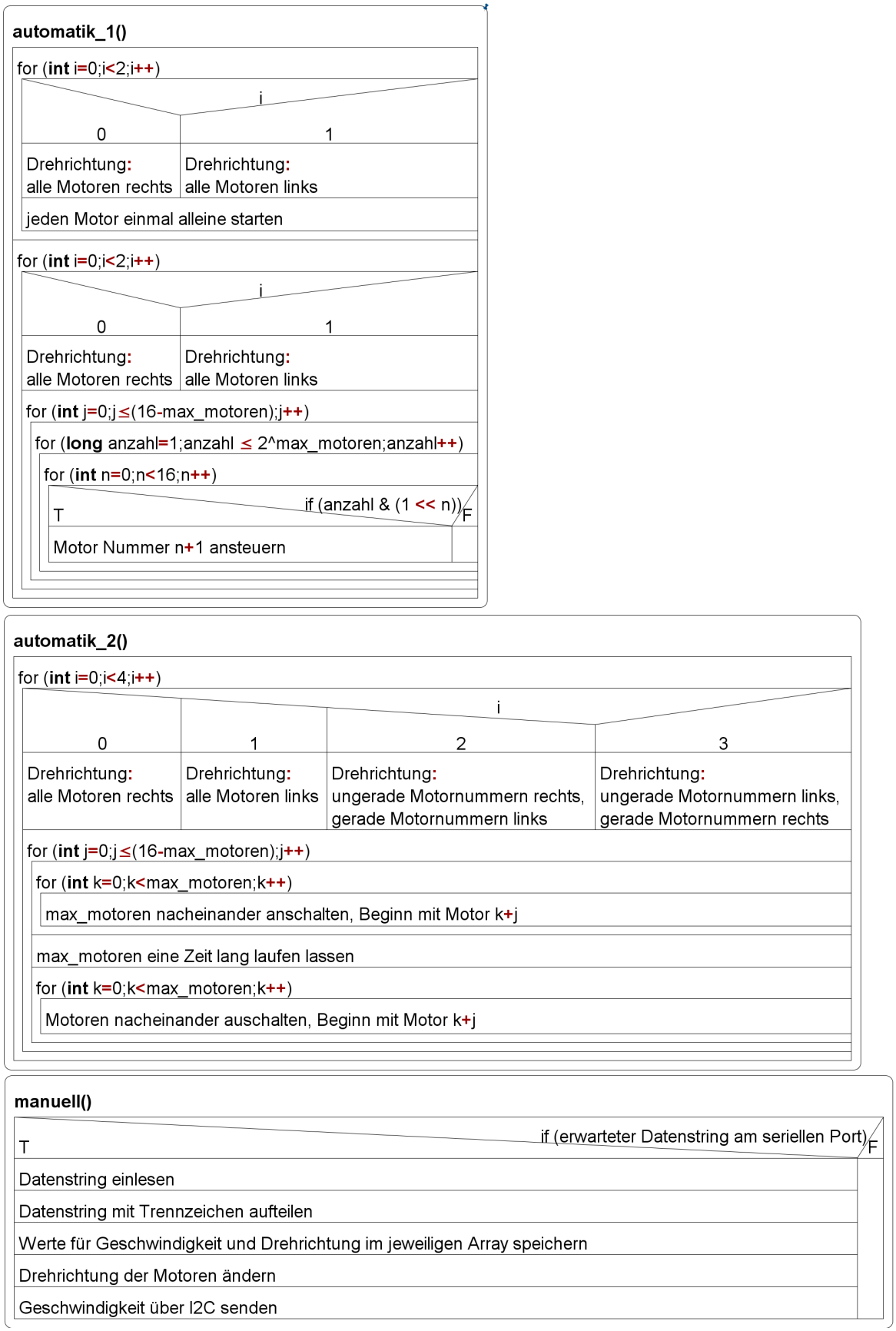


Abbildung 4.2: Struktogramme der Funktionen vom Programm Motoransteuerung I2C-Master

Die Abbildung 4.3 zeigt die Funktion des Programms für die Motoransteuerung von vier Schrittmotoren mittels Pulssignalen. Die Geschwindigkeit eines Motors wird durch die Frequenz des jeweiligen Pulssignals bestimmt. Je höher die Frequenz ist, desto schneller dreht sich der Motor.
Der zugehörige Quellcode ist im Anhang unter A.1.2 zu finden.

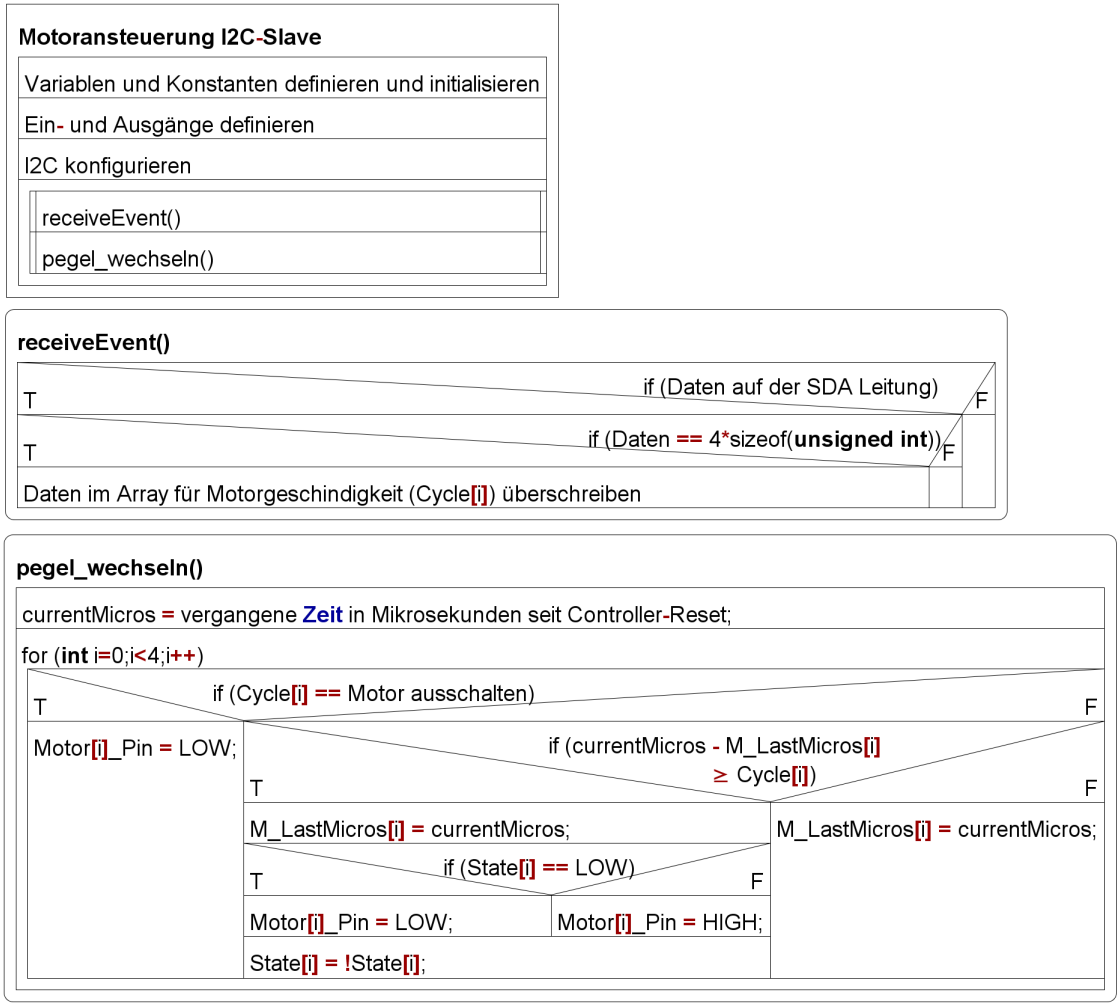


Abbildung 4.3: Struktogramm Motoransteuerung I2C-Slave

4.2 Signalüberwachung

4.2.1 Überwachung von Signalen im ZMX-Überrahmen

Die Überwachung der Fehlersignale der ZMX-Endstufen sowie der Signale der Endlagenschalter der Schrittmotoren erfolgt unter Verwendung des Mikrocontrollers des Arduino Mega2560 (Signalüberwachung).
Das folgende Struktogramm in der Abbildung 4.4 beschreibt den Programmablauf. Der zugehörige Programmcode befindet sich im Anhang unter A.3.
Das Entwicklungsboard Arduino Mega2560 kommuniziert über die integrierte USB Schnittstelle mit dem LabVIEW Programm „ZMX_Ueberrahmen_Pruefprogramm“. Hierbei wartet der Mikrocontroller zunächst auf das Startsignal von LabVIEW. Anschließend werden kontinuierlich die Fehlersignale der ZMX-Endstufen sowie die Signale der Endlagenschalter der Schrittmotoren überwacht. Sollte bei irgendwelchen Signalen ein Fehler aufgetreten sein, wird dieser in Textform in einem String gespeichert. Alle weiteren aufgetretenen Fehler werden mittels Konkatenation zu dem String hinzugefügt und anschließend über die USB-Schnittstelle an LabVIEW gesendet. Abschließend wird die Startbedingung zurückgesetzt und es wird erneut auf das Startsignal von LabVIEW gewartet.

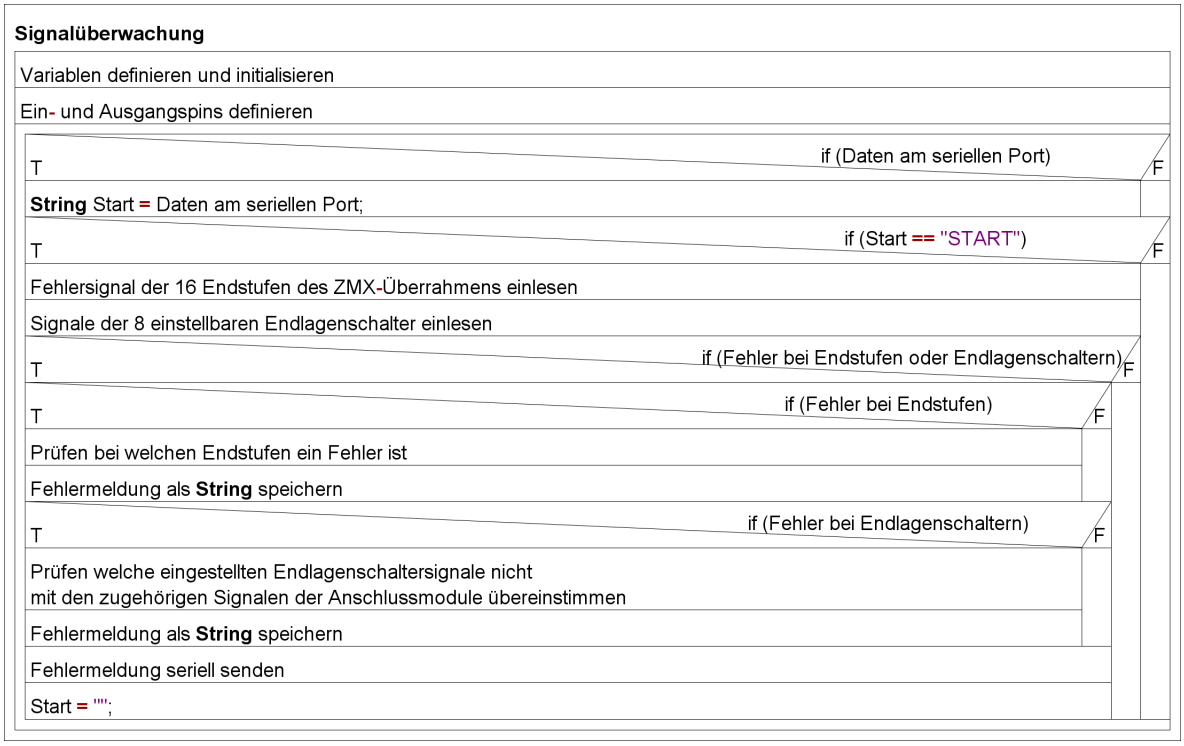


Abbildung 4.4: Struktogramm Signalüberwachung

4.2.2 Überwachung der Netzteilspannungen im ZMX-Überrahmen

Die Überwachung der Netzteilspannungen der vier Netzteile eines ZMX-Überrahmens ist mit dem Oszilloskop Tektronix TPS 2024 realisiert. Da die Auswertung der gemessenen Spannungswerte vom LabVIEW Programm übernommen wird, ist die Zuordnung vom Messkanal des Oszilloskop zum Netzteil vorgegeben. Der Kanal 1 wird am Netzteil DSP30-5, Kanal 2 am DSP60-15, Kanal 3 am DSP10-5 und Kanal 4 am Netzteil EXW 60.25/OP1 angeschlossen.
Da das Oszilloskop über eine RS232-Schnittstelle verfügt, wird zur Kommunikation mit LabVIEW ein Kabel verwendet, welches einen integrierten RS232 zu USB-Adapter enthält.
Die Konfiguration und Steuerung des Oszilloskops erfolgt ausschließlich vom LabVIEW Programm „ZMX_Ueberrahmen_Pruefprogramm“.
Die Spannungen dürfen die in 4.4.8 eingestellten Schwellwerte nicht über- bzw. unterschreiten.

4.3 Entwicklung der Testbox

Die folgende Abbildung 4.5 zeigt das Blockschaltbild der entwickelten Testbox.

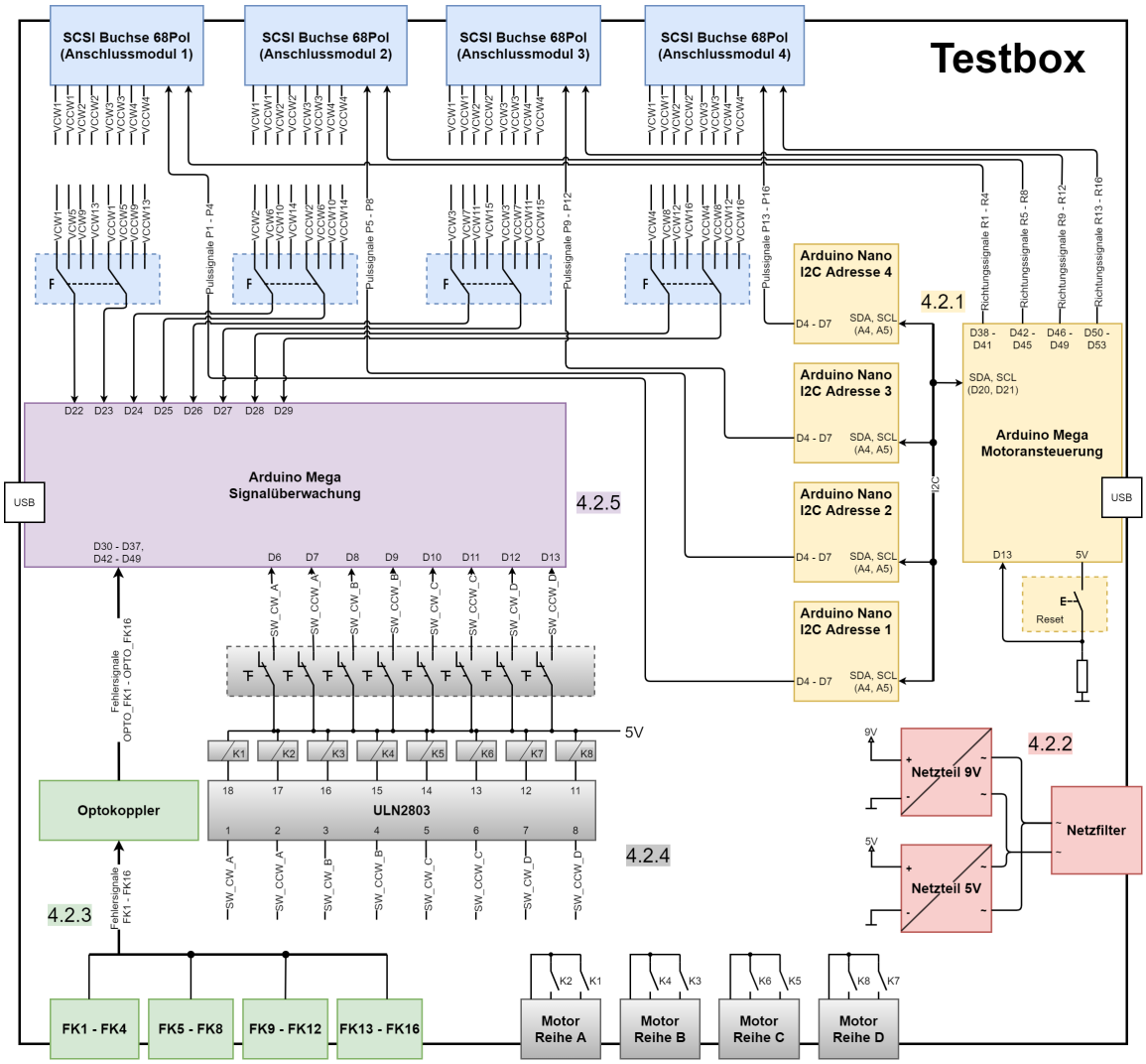


Abbildung 4.5: Blockschaltbild der Testbox

Alle dargestellten Taster und Schalter zur Bedienung befinden sich in der Frontplatte der Testbox.
Für die Auswahl der Bauteilwerte sowie der Schaltung von den Optokopplern und den Relais, wird sich an bereits bestehenden und bewährten Schaltungen vom DESY orientiert [3].
In den folgenden Abbildungen 4.6 - 4.9 ist die real aufgebaute Testbox von verschiedenen Außenseiten sowie eine Innenansicht zu erkennen.

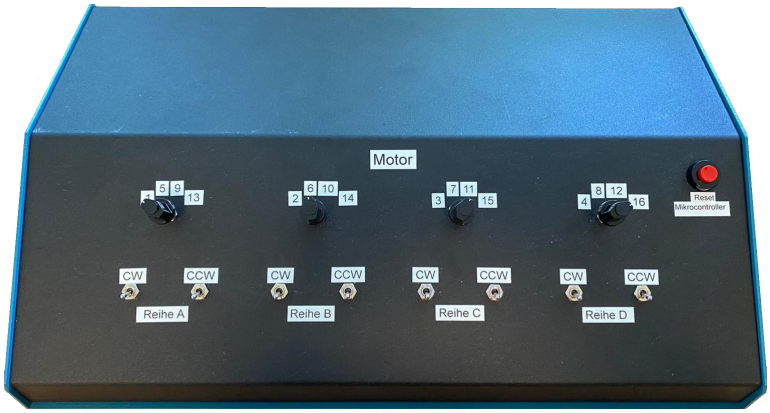


Abbildung 4.6: Frontplatte der Testbox

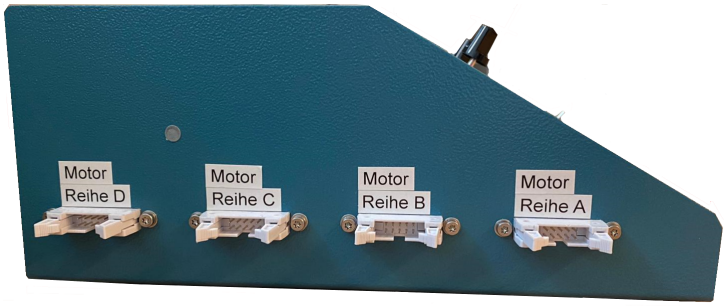


Abbildung 4.7: Seitenansicht der Testbox

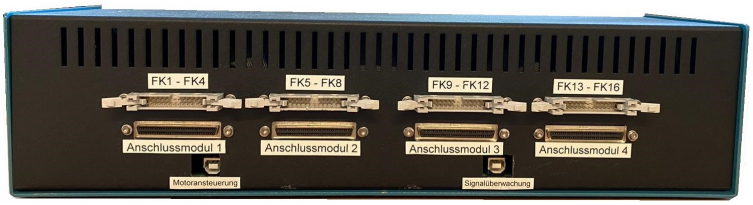


Abbildung 4.8: Rückseite der Testbox

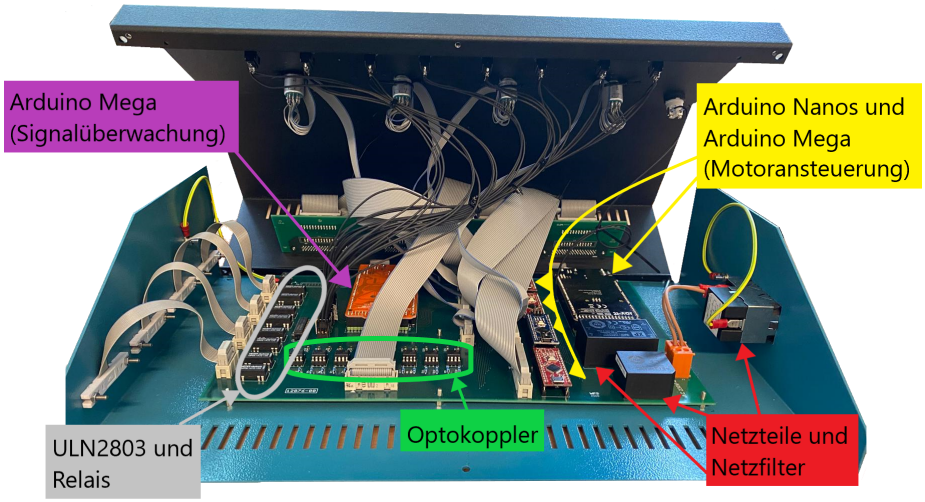


Abbildung 4.9: Innenansicht der Testbox

Im Folgenden werden die dargestellten Blöcke in nachstehender Reihenfolge beschrieben:

- Ansteuerung des ZMX-Überrahmens 4.3.1
- Spannungsversorgung der Testbox 4.3.2
- Fehlersignale der ZMX-Endstufen 4.3.3
- Endlagenschalter für vier Schrittmotoren 4.3.4
- Mikrocontroller für die Signalüberwachung und Auswertung 4.3.5

4.3.1 Ansteuerung des ZMX-Überrahmens

Auf der rechten Seite in den gelb markierten Bereichen sind in Abbildung 4.5 fünf Arduino Entwicklungsboards, welche für die Ansteuerung des ZMX-Überrahmens verwendet werden, zu finden. Die Kommunikation zwischen den vier Arduino Nanos mit dem gelb markierten Arduino Mega2560 erfolgt über das I2C-Protokoll. Ein Arduino Nano erzeugt die Pulssignale für vier Schrittmotoren und leitet diese an eine der im oberen Bereich des Blockschaltbilds zu findenden blau gekennzeichneten SCSI-Buchsen weiter. Über diese Buchsen erfolgt die Verbindung zum zugehörigen Anschlussmodul des ZMX-Überrahmens. Die Richtungssignale für alle 16 Schrittmotoren werden von dem gelb gekennzeichneten Arduino Mega2560 generiert. Hierbei werden immer vier Richtungssignale einer SCSI-Buchse zugeordnet.
In dem mit gelb gestrichelten Linien umrandeten Rechteck befindet sich ein Drucktaster, welcher für den Reset dieser fünf Mikrocontroller verwendet wird.

4.3.2 Spannungsversorgung der Testbox

Unten rechts in dem Blockschaltbild sind in roter Farbe ein Netzfilter sowie zwei Netzteile zu erkennen. Bei dem Netzfilter handelt es sich um ein Produkt vom Hersteller Schurter und ist aus der KMF-Serie. Das 9 V-Netzteil (VCE10US09) übernimmt die Spannungsversorgung der vier Arduino Nano Entwicklungsbords. Die empfohlene Betriebsspannung für diese Boards liegt zwischen 7 V und 12 V. Das 5 V-Netzteil (VCE05US05) versorgt die grau markierten Relais K1 bis K8, sowie die Ausgangsseite der grün markierten Optokoppler.

4.3.3 Fehlersignale der ZMX-Endstufen

Unten links befinden sich die Anschlussbuchsen für die Ausgabe der Fehlersignale FK1 bis FK16 der Endstufen des ZMX-Überrahmens. Da die Endstufen im ZMX-Überrahmen einen anderen Massebezug als die Anschlussmodule haben, werden die ankommenden

Signale über jeweils einen Optokoppler zum lila gekennzeichneten Arduino Mega2560 geleitet, um die galvanische Trennung der verschiedenen Massen beizubehalten.

4.3.4 Endlagenschalter für vier Schrittmotoren

In den grau markierten Bereichen befindet sich die Schaltungsrealisierung für die Endlagenschalter-Simulation für vier Schrittmotoren. In dem Rechteck mit den gestrichelten Linien sind acht Kippschalter zu erkennen. Ein Kippschalter simuliert die Endlage für eine Richtung eines Schrittmotors. Da die Schrittmotoren ebenfalls ein anderes Massepotential verwenden, müssen die Schalter galvanisch von der Masse innerhalb der Testbox getrennt werden. Dies geschieht durch Verwendung von Relais, welche jeweils über einen Schließerkontakt verfügen. Die Schaltung der Relais erfolgt über den Schaltbaustein ULN2803, welcher aus einem Darlington-Transistor-Array besteht.

Oben im Blockschaltbild, in den blauen Rechtecken mit den gestrichelten Linien, sind vier 4x2 Drehschalter zu erkennen. Die durchgeschalteten Signale dienen als Vergleichswerte für die Signale der simulierten Endlagenschalter.

4.3.5 Mikrocontroller für die Signalüberwachung und Auswertung

Der lila markierte Arduino Mega2560 hat die Funktion der Überwachung der dargestellten eingehenden Signale und deren Auswertung. Die Spannungsversorgung und der Datenaustausch mit LabVIEW erfolgt über die integrierte USB-Schnittstelle.

4.4 Programmiertechnische Umsetzung in LabVIEW

Der schematische Ablauf des LabVIEW Programms „ZMX_Ueberrahmen_Pruefprogramm“ ist dem Struktogramm in Abbildung 4.10 zu entnehmen.

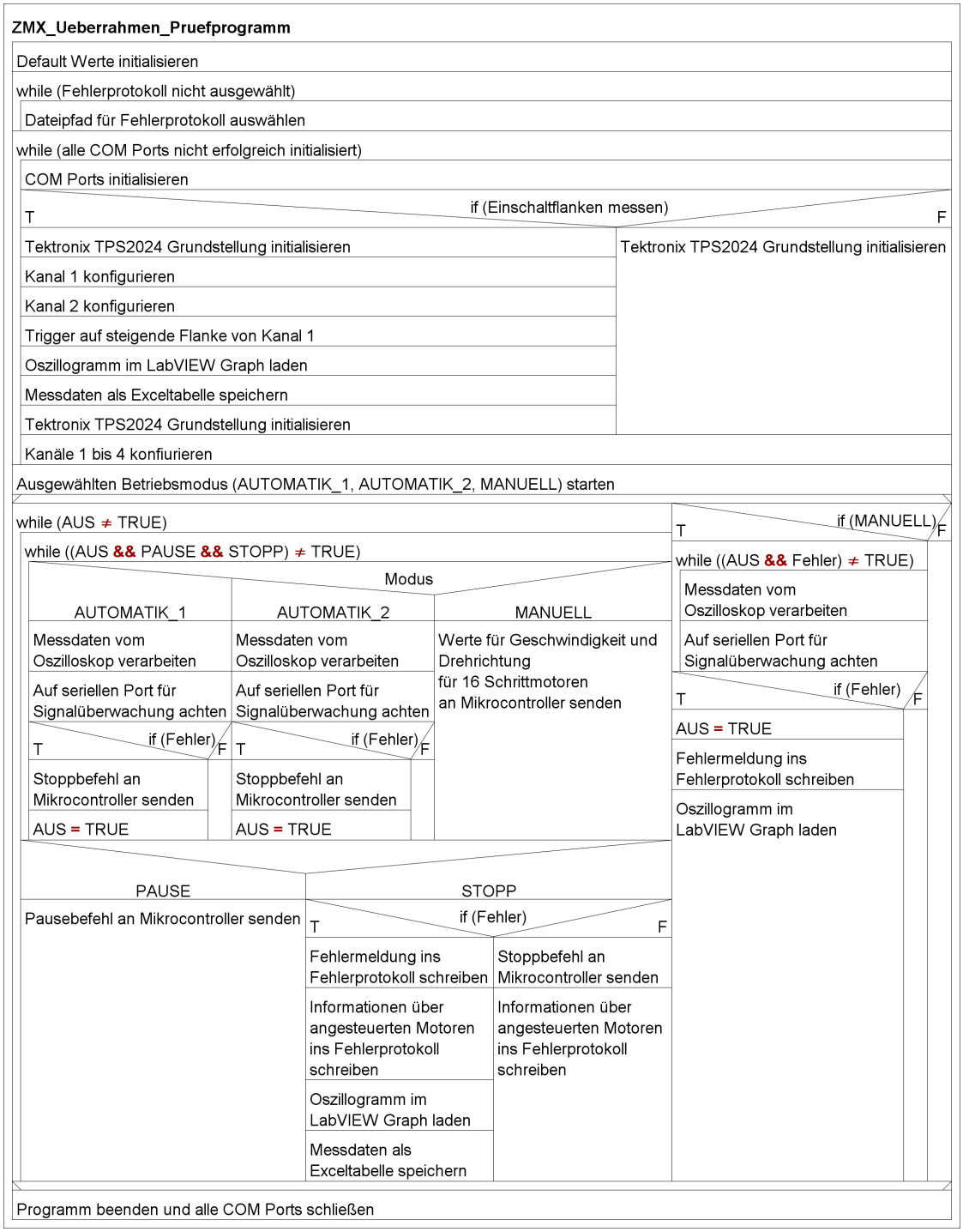


Abbildung 4.10: Struktogramm LabVIEW

Die folgende Abbildung 4.11 zeigt das in LabVIEW erstellte Frontpanel, welches die Bedienoberfläche des Teststands darstellt.

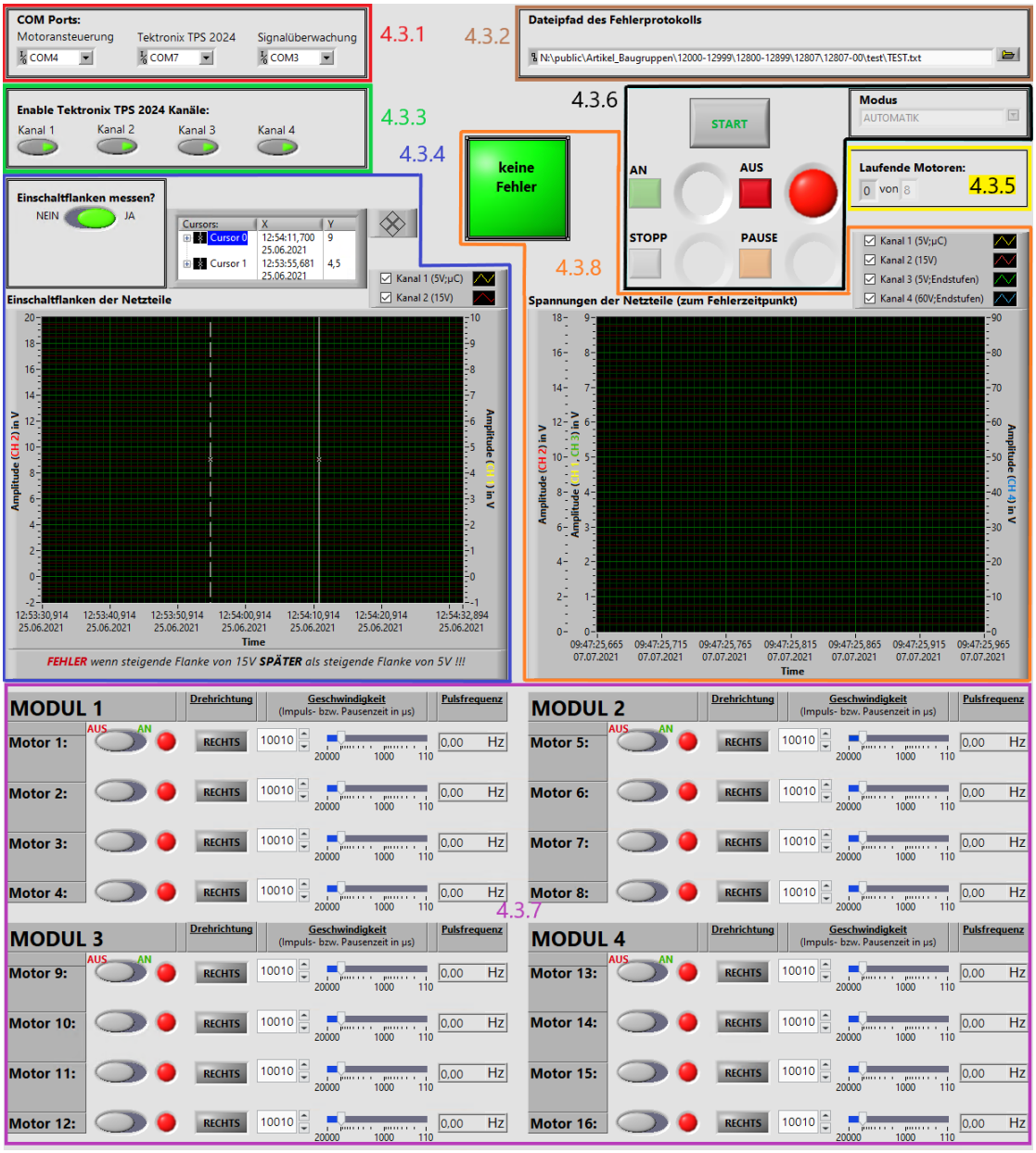


Abbildung 4.11: Frontpanel LabVIEW

Im Folgenden werden die einzelnen Funktionen der Bedienelemente in nachstehender Reihenfolge genauer beschrieben:

- Auswahl der USB Ports für die Testbox und das Oszilloskop 4.4.1
- Auswahl der Datei für das Fehlerprotokoll 4.4.2
- Einstellung der Oszilloskop-Kanäle 4.4.3
- Messung der Einschaltflanken von zwei Netzteilen 4.4.4
- Einstellung und Anzeige der maximalen Anzahl an gleichzeitig laufenden Schrittmotoren 4.4.5
- Bedienelemente für den Start und laufenden Betrieb des Teststands 4.4.6
- Bedienelemente für die Steuerung der Schrittmotoren im manuellen Modus 4.4.7
- Signalisierung von aufgetretenen Fehlern 4.4.8

4.4.1 Auswahl der USB Ports für die Testbox und das Oszilloskop

Die drei Combo-Boxen der Abbildung 4.12 dienen der Auswahl des jeweiligen USB Ports für die beiden Arduino Mega2560 in der Testbox sowie für das Oszilloskop Tektronix TPS 2024.

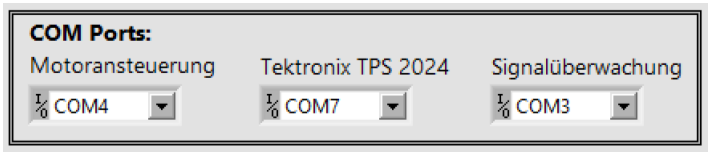


Abbildung 4.12: Combo-Boxen zur Auswahl der USB Ports für die Testbox und das Oszilloskop

4.4.2 Auswahl der Datei für das Fehlerprotokoll

Das in der Abbildung 4.13 dargestellte Auswahlfeld wird zur Angabe und Auswahl des Dateinamens und des Dateipfads vom Fehlerprotokoll verwendet.

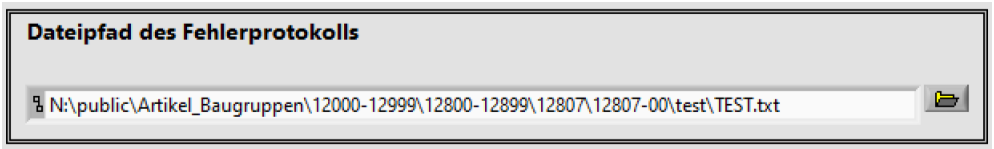


Abbildung 4.13: Auswahl der Datei für das Fehlerprotokoll

Die Auswahl erfolgt direkt zum Start des Programms und wird zusätzlich durch ein Dialogfeld hervorgerufen, in welchem der Benutzer aufgefordert, wird eine Datei auszuwählen. Dieses Dialog schließt sich erst nachdem eine Datei ausgewählt wurde und lässt

sich nicht abbrechen.
Das Fehlerprotokoll wird als Textdatei mit der Endung „txt“ erstellt. Falls der Benutzer einen anderen oder keinen Dateityp angibt, wird nur der angegebene Dateiname übernommen und die Endung „txt“ angehängt.

Der folgende Programmausschnitt in der Abbildung 4.14 zeigt die Realisierung in LabVIEW.

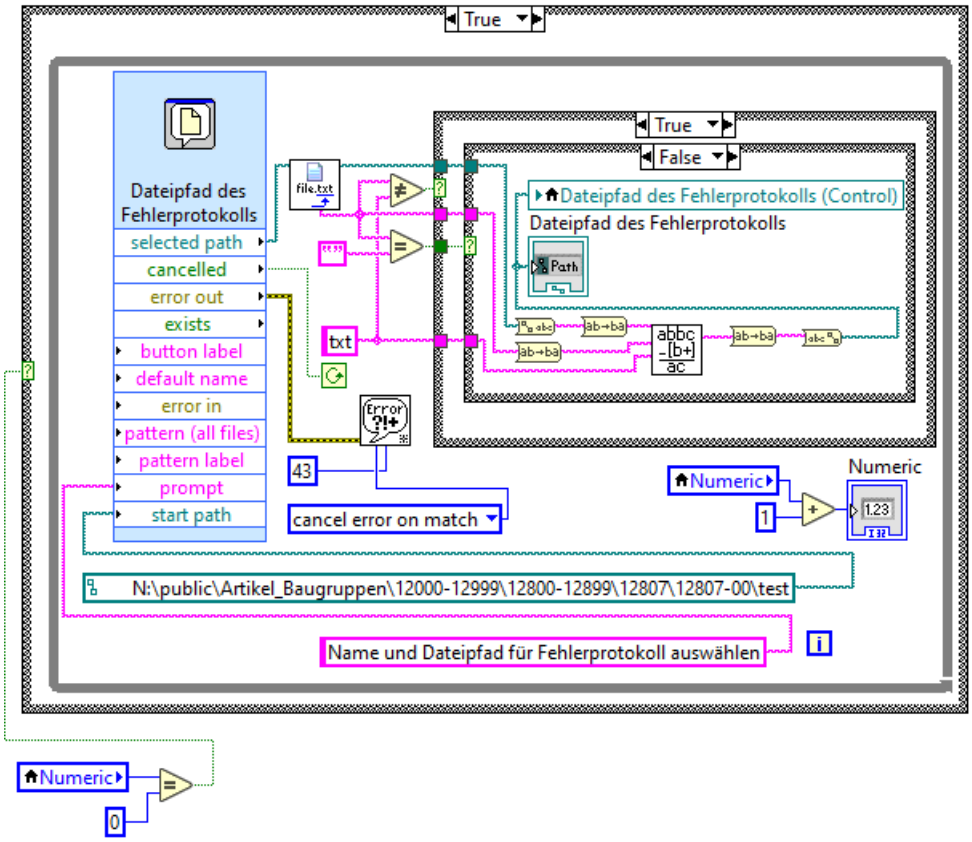


Abbildung 4.14: Auswahl des Dateinamens und des Dateipfads für das zu erstellende Fehlerprotokoll

4.4.3 Einstellung der Oszilloskop-Kanäle

Die vier Push-Buttons der folgenden Abbildung 4.15 dienen der Aktivierung der Oszilloskop-Kanäle.

Abbildung 4.15: Push-Buttons zur Aktivierung der Oszilloskop-Kanäle

Die folgende Abbildung 4.16 zeigt das zugehörige Blockdiagramm in LabVIEW. Für die vom Benutzer aktivierten Kanäle werden die X- und Y-Achsen-Einstellungen der Oszilloskop-Kanäle automatisch vorgenommen, um dem Nutzer die Bedienung des Teststands zu erleichtern.
Die zu erkennenden SubVIs mit der Aufschrift „TDS1K2X“ sind Inhalt des Gerätetreibers „Tektronix TDS 200 1000 2000 Series“ von National Instruments. Diese übernehmen die Kommunikation mit dem Oszilloskop über die serielle Schnittstelle.

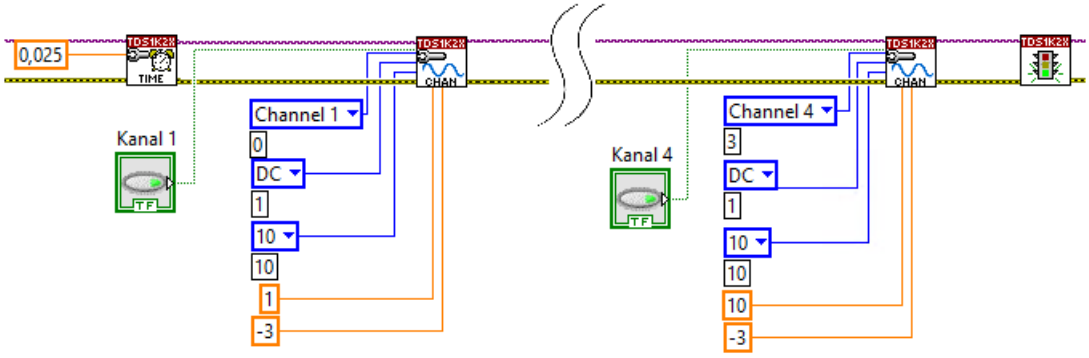


Abbildung 4.16: Aktivierung und Konfiguration der vier Oszilloskop-Kanäle

4.4.4 Messung der Einschaltflanken von zwei Netzteilen

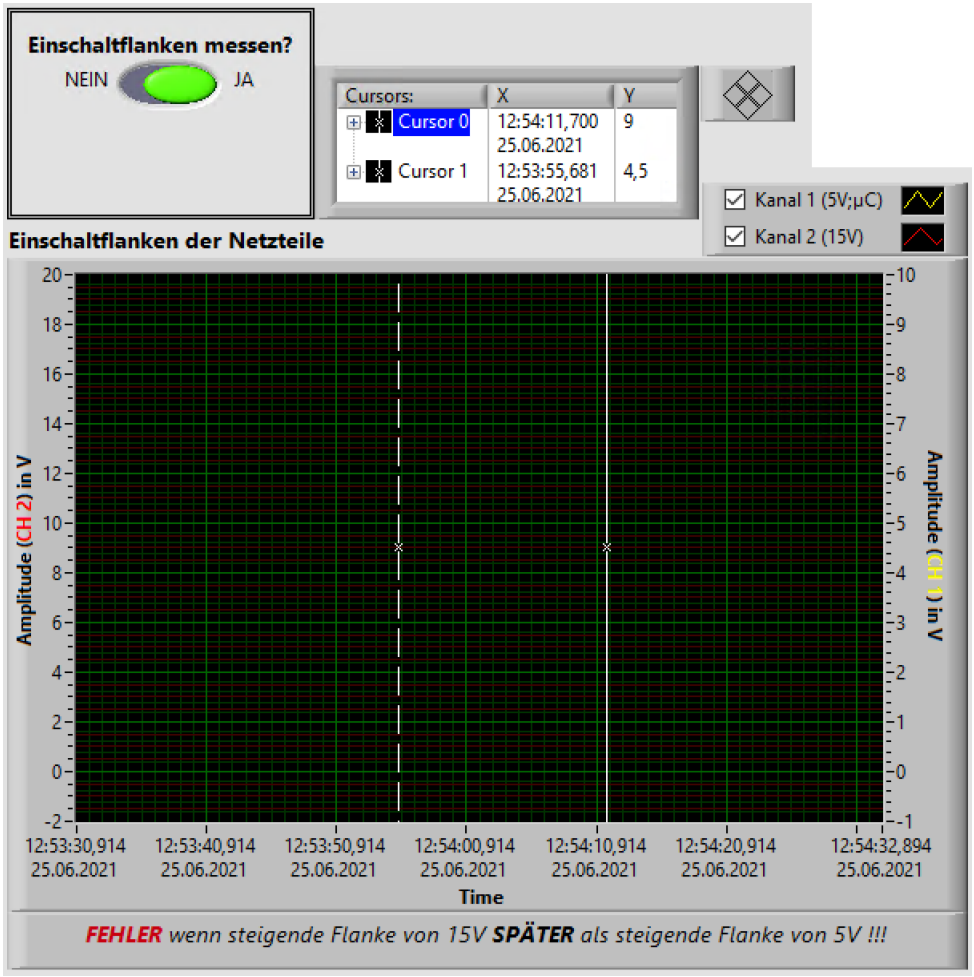


Abbildung 4.17: Oszillograph der gemessenen Einschaltflanken

Der Benutzer wählt über ein Dialogfeld aus, ob die Einschaltflanken eines der beiden 5 V-Netzteile (DSP30-5) und des 15 V-Netzteils (DSP60-15) im ZMX-Überrahmen gemessen werden sollen. Der in der Abbildung 4.17 zu erkennende Slide-Switch zeigt dies anschließend an. Das zugehörige Blockdiagramm ist zusammen mit dem Dialogfeld in der Abbildung 4.18 zu sehen.

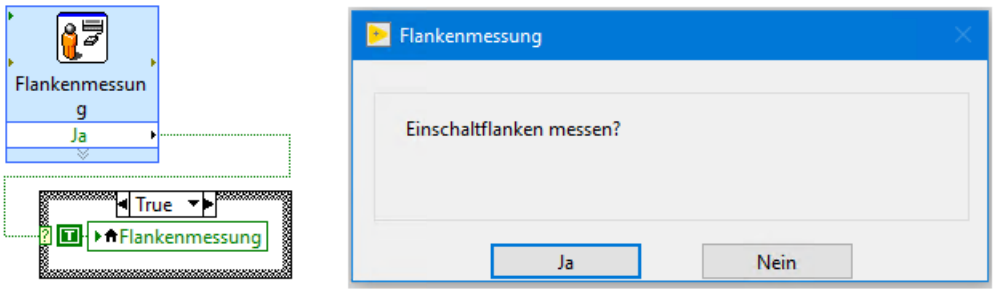


Abbildung 4.18: Dialogfeld für die Einstellung zur Aktivierung der Einschaltflankenmessung

Die nachfolgende Abbildung 4.19 zeigt das Blockdiagramm für die Konfiguration des Oszilloskops und das Speichern der gemessenen Werte.

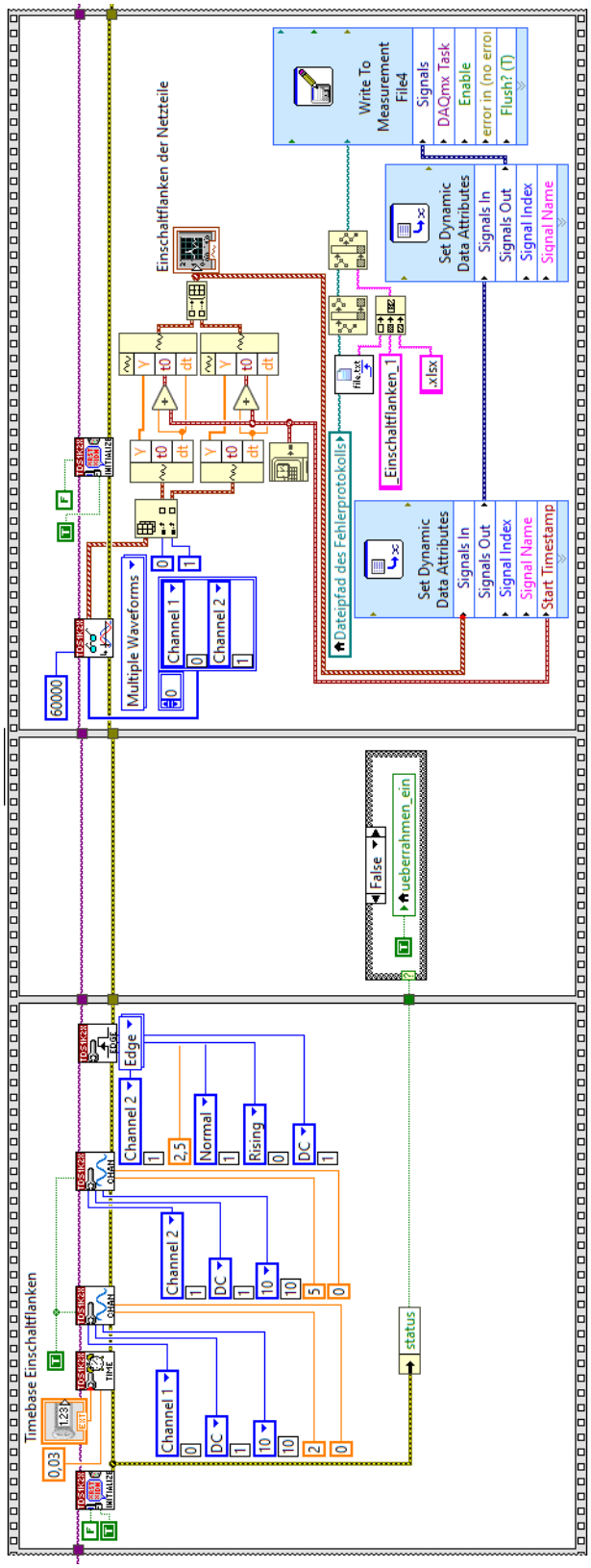


Abbildung 4.19: Konfiguration der Oszilloskop-Kanäle und Speichern der gemessenen Werte für die Einschaltflankenmessung

In dieser Abbildung 4.19 sind drei Sequenzen zu sehen. In der ersten erfolgt die Initialisierung und Einstellung der beiden zugehörigen Oszilloskop-Kanäle. Hierbei stellt Kanal 1 die Messung am 5 V-Netzteil und Kanal 2 die Messung am 15 V-Netzteil dar. Der Trigger wird auf die steigende Flanke von Kanal 2 gesetzt, da die Einschaltflanke des angeschlossenen 15 V-Netzteils zuerst kommen sollte. Die gesamte Initialisierung und Konfiguration erfolgt automatisch.
In der zweiten Sequenz wird der Benutzer durch eine Text-Variable dazu aufgefordert, den ZMX-Überrahmen einzuschalten. Die Aufforderung erfolgt wie in der nachfolgenden Abbildung 4.20 dargestellt.

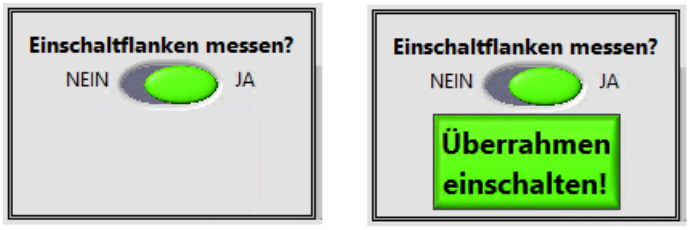


Abbildung 4.20: Aufforderung zum Einschalten des ZMX-Überrahmens, durch eine Text-Variable

Die dritte Sequenz zeigt das SubVI zum Auslesen der vom Oszilloskop gemessenen Werte. Diese werden dann zusätzlich im Oszillographen, aus der Abbildung 4.17, in LabVIEW dargestellt und in einer Excel-Tabelle gespeichert.
Abschließend bewertet der Nutzer die Qualität der gemessenen Einschaltflanken gemäß der Aussage unterhalb des Oszillographen aus der Abbildung 4.17.

4.4.5 Einstellung und Anzeige der maximalen Anzahl an gleichzeitig laufenden Schrittmotoren

Abbildung 4.21: Einstellung und Anzeige der maximalen Anzahl an gleichzeitig laufenden Schrittmotoren

Die linke Ziffer der Abbildung 4.21 zeigt an, wie viele Schrittmotoren im manuellen Betriebsmodus angesteuert werden. Im automatischen Betrieb hat diese Anzeige keine Funktion.
Die rechte Ziffer zeigt an, wie viele von den 16 angeschlossenen Schrittmotoren maximal gleichzeitig angesteuert werden sollen. Der Maximalwert ist hierbei historisch bedingt auf „8“ festgelegt. Minimal kann der Wert „1“ eingegeben werden. Bei Zahlenangaben

außerhalb dieses Wertebereichs oder Dezimalzahlen wird automatisch zum nächstgelegenen gültigen, ganzzahligem Wert gerundet.
Diese Einstellung gelten sowohl für den manuellen als auch für den automatischen Betrieb. Erst nach dem Beenden eines gestarteten Modus lassen sich diese Einstellungen wieder ändern.

4.4.6 Bedienelemente für den Start und laufenden Betrieb des Teststands

Die Einstellungen aus 4.4.1, 4.4.2 und 4.4.3 werden nach Betätigung des START-Buttons übernommen und sind ab dem Zeitpunkt nicht mehr veränderbar, bis das Programm beendet und erneut gestartet wird. Im Anschluss daran erfolgt der in 4.4.4 beschriebene Vorgang zur Messung der Einschaltflanken.
Nachdem die Initialisierung des Oszilloskops mit den gewählten Einstellungen abgeschlossen ist, lässt sich der Betriebsmodus aus der Combo-Box in der Abbildung 4.22 wählen.

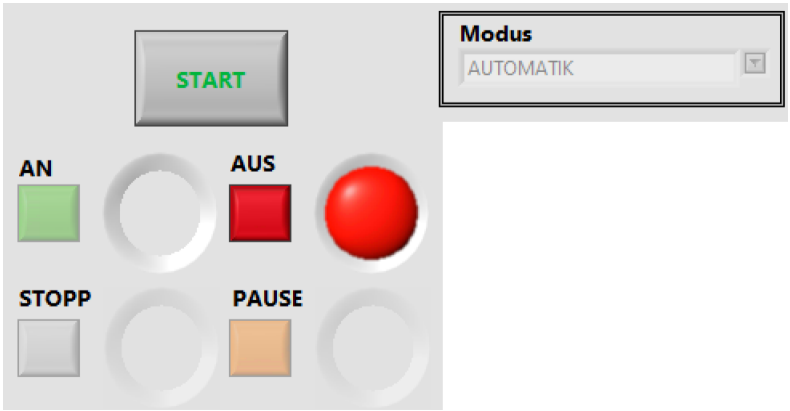


Abbildung 4.22: Bedienelemente für den Start und laufenden Betrieb des Teststands

Der STOPP-Button und der PAUSE-Button dienen im Automatikbetrieb dazu, das auf dem Arduino Mega2560 (Motoransteuerung) laufende Programm zu pausieren. Der STOPP-Button hat zusätzlich die Funktion, die Informationen für die zu dem Zeitpunkt angesteuerten Schrittmotoren in das Fehlerprotokoll zu schreiben. Im manuellen Betriebsmodus haben diese beiden Buttons keine Funktion und sind deaktiviert.
Der AN-Button und der AUS-Button dienen dem Start beziehungsweise dem Beenden des jeweiligen Betriebsmodus.

Die folgende Abbildung 4.23 zeigt das in LabVIEW erstellte Blockdiagramm für den Start des eingestellten Betriebsmodus und die dafür notwendige Kommunikation mit dem Arduino Mega2560 (Motoransteuerung).

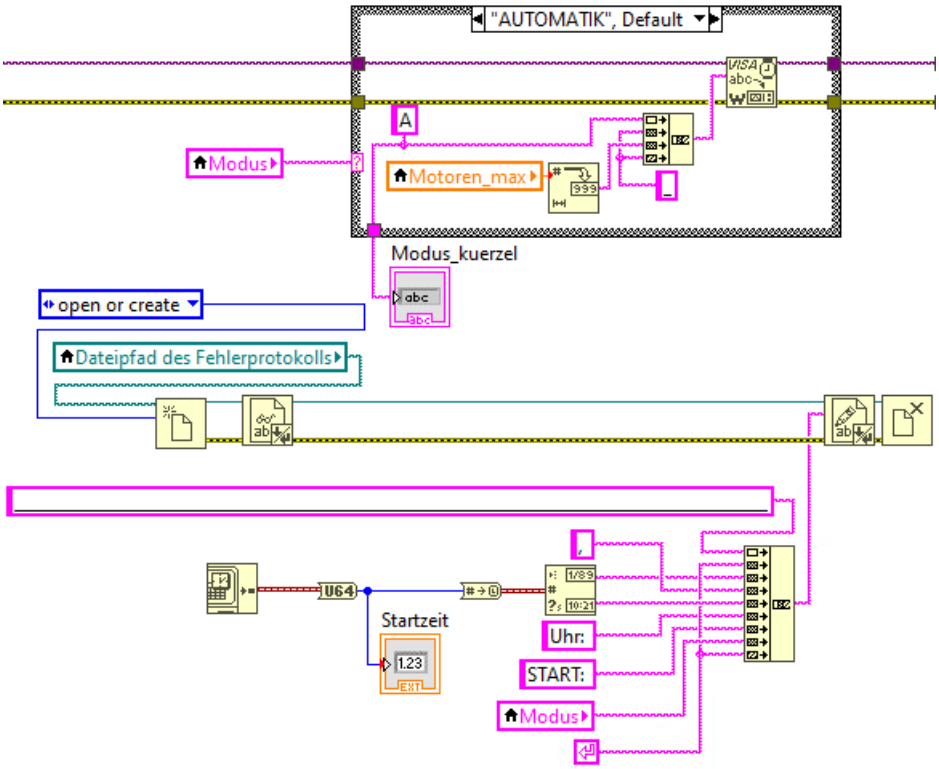


Abbildung 4.23: Starten des Betriebsmodus durch Senden des jeweiligen Kommandos an den Arduino Mega2560 (Motoransteuerung)

4.4.7 Bedienelemente für die Steuerung der Schrittmotoren im manuellen Modus

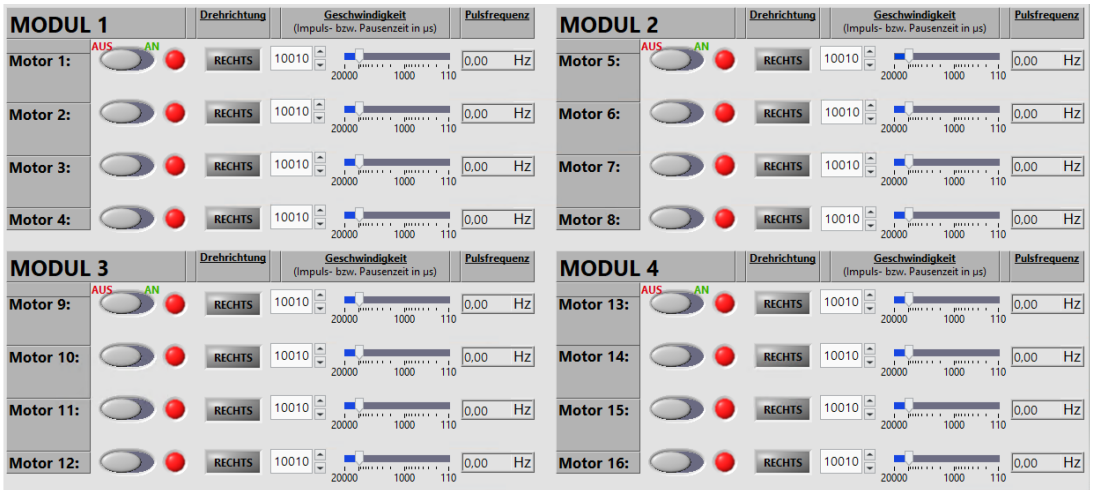


Abbildung 4.24: Bedienelemente für die Steuerung der Schrittmotoren im manuellen Modus

Die Bedienelemente der obigen Abbildung 4.24 dienen der Steuerung der 16 Schrittmotoren im manuellen Betriebsmodus. Im automatischen Betrieb haben diese Elemente keine Funktion und sind deaktiviert.

Die Slide-Switches schalten den jeweiligen Schrittmotor ein, sofern die zuvor festgelegte maximale Anzahl an gleichzeitig laufenden Motoren, wie in 4.4.5 beschrieben, noch nicht erreicht ist. Dass der jeweilige Motor läuft, wird durch eine grüne LED rechts neben dem Slide-Switches signalisiert. Die Geschwindigkeit beziehungsweise die Pulsfrequenz für den jeweiligen Motor ergibt sich aus dem eingestellten Wert beim zugehörigen Schieberegler. Durch Betätigung des Push-Buttons lässt sich die Drehrichtung des jeweiligen Schrittmotors einstellen.

In der folgenden Abbildung 4.25 ist das zugehörige Blockdiagramm in LabVIEW zu erkennen. Die eingestellten Werte der Push-Buttons (0 oder 1) und der Schieberegler (110 bis 20000) werden zu einem String konkateniert und zur Verarbeitung an den Arduino Mega2560 (Motoransteuerung) gesendet.

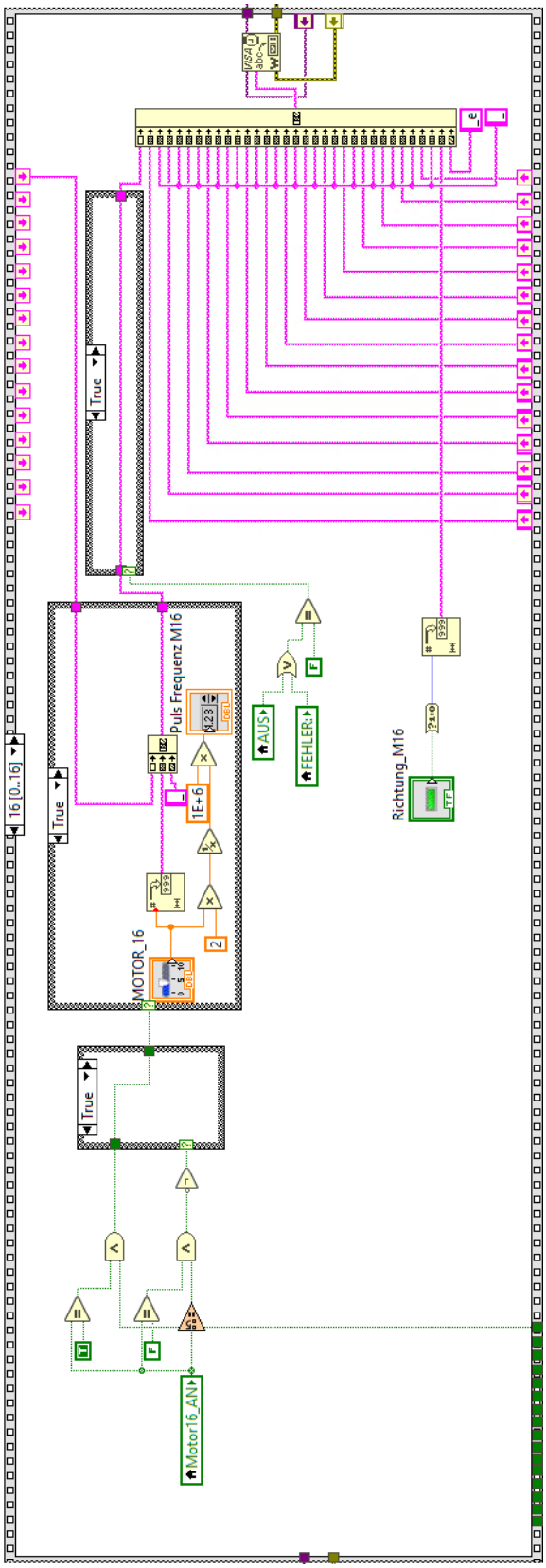


Abbildung 4.25: Senden des konkatenierten String mit Informationen über Drehrichtung und Geschwindigkeit an den Arduino Mega2560 (Motoransteuerung)

4.4.8 Signalisierung von aufgetretenen Fehlern

Die folgenden Abbildung 4.26 zeigt die digitale LED zur Fehlersignalisieung sowie den Oszillographen für Spannungsverläufe zum Fehlerzeitpunkt.

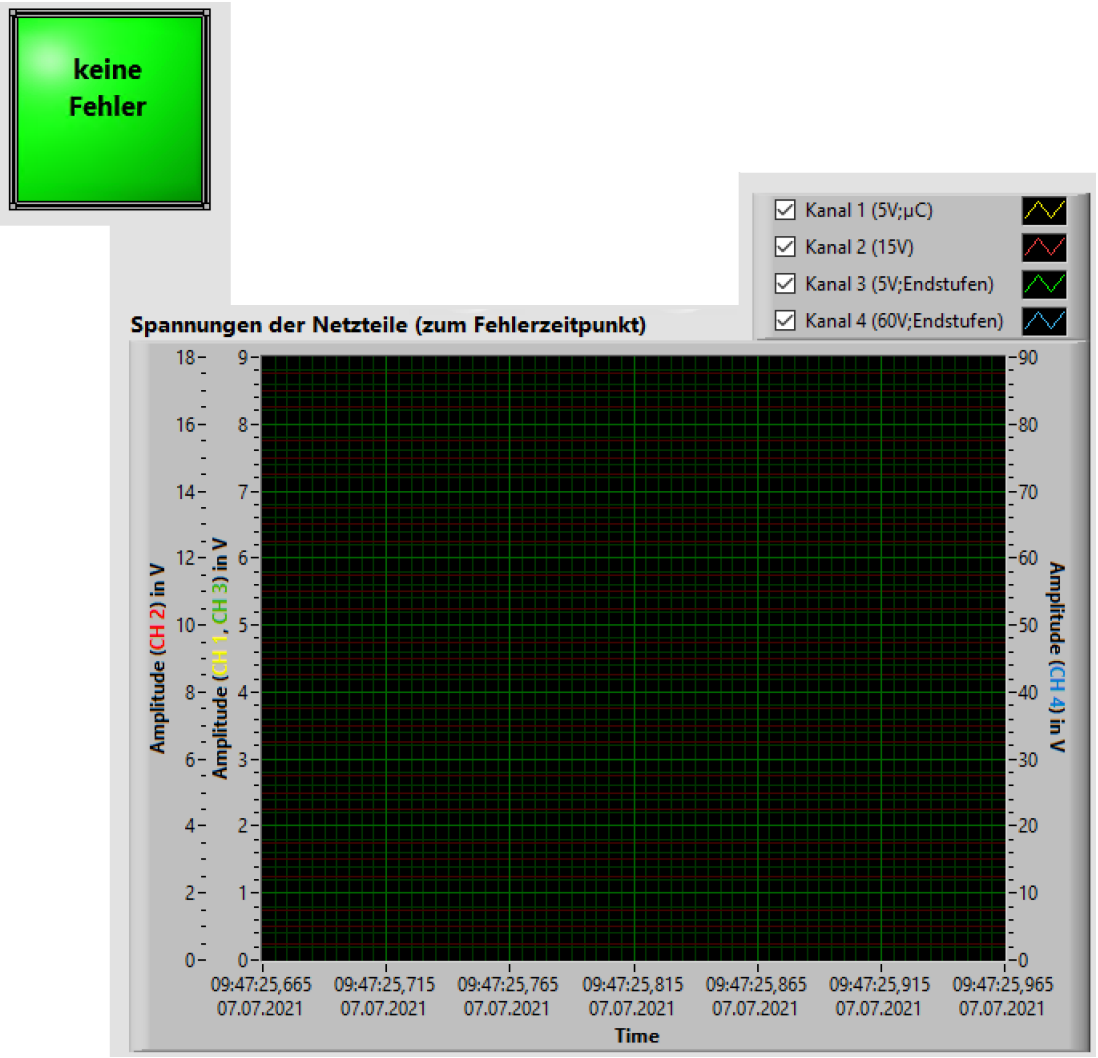


Abbildung 4.26: Fehleranzeige und Oszillograph für Spannungsverläufe zum Fehlerzeitpunkt

Die grüne, quadratische LED mit dem Schriftzug „keine Fehler“ dient zur Signalisierung, ob ein Fehler aufgetreten ist. Ist dies der Fall, wechselt die Farbe zu rot und der Schriftzug ändert sich zu „Fehler!“. Diese beiden Zustände sind in der folgenden Abbildung 4.27 dargestellt.
Zur Fehlererkennung werden die entwickelten SubVIs „ZMX_Spannungsmessung“ und „ZMX_Signalueberwachung“ verwendet, welche im Folgenden beschrieben werden.

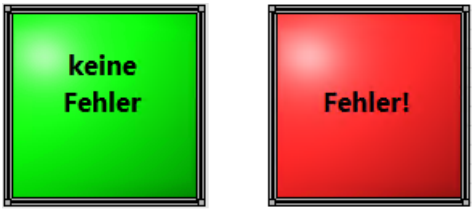


Abbildung 4.27: LED zur Fehlersignalisierung bei keinem aufgetretenen Fehler (links) und aufgetretenem Fehler (rechts)

SubVI „ZMX_Spannungsmessung“

Zur Überwachung der Spannungen der Netzteile des ZMX-Überrahmens wird sich der SubVIs des Gerätetreibers „Tektronix TDS 200 1000 2000 Series“ bedient. Es werden, abhängig vom Netzteil, die minimal und die maximal gemessenen Spannungswerte erfasst. Diese Erfassung wird vom Gerätetreiber bereitgestellt. Bei Über- bzw. Unterschreiten eines einstellbaren Schwellwerts wird die Messung gestoppt und das Oszilloskop behält den aktuellen Oszillographen bei.
Dieser Vorgang ist in der Abbildung 4.28 zu erkennen.

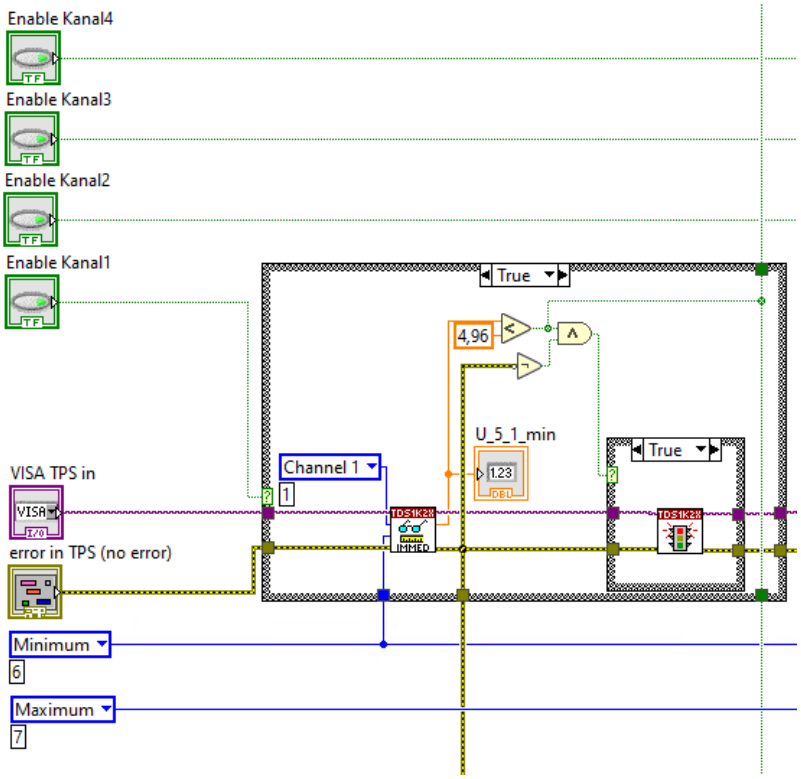


Abbildung 4.28: Ausschnitt des Blockdiagramms vom SubVI „ZMX_Spannungsmessung“

Die Informationen von allen fehlerhaften Spannungswerten werden zu dem String „Fehlermeldung“ konkateniert. Das Blockdiagramm für die Realisierung in LabVIEW ist der folgenden Abbildung 4.29 zu entnehmen.

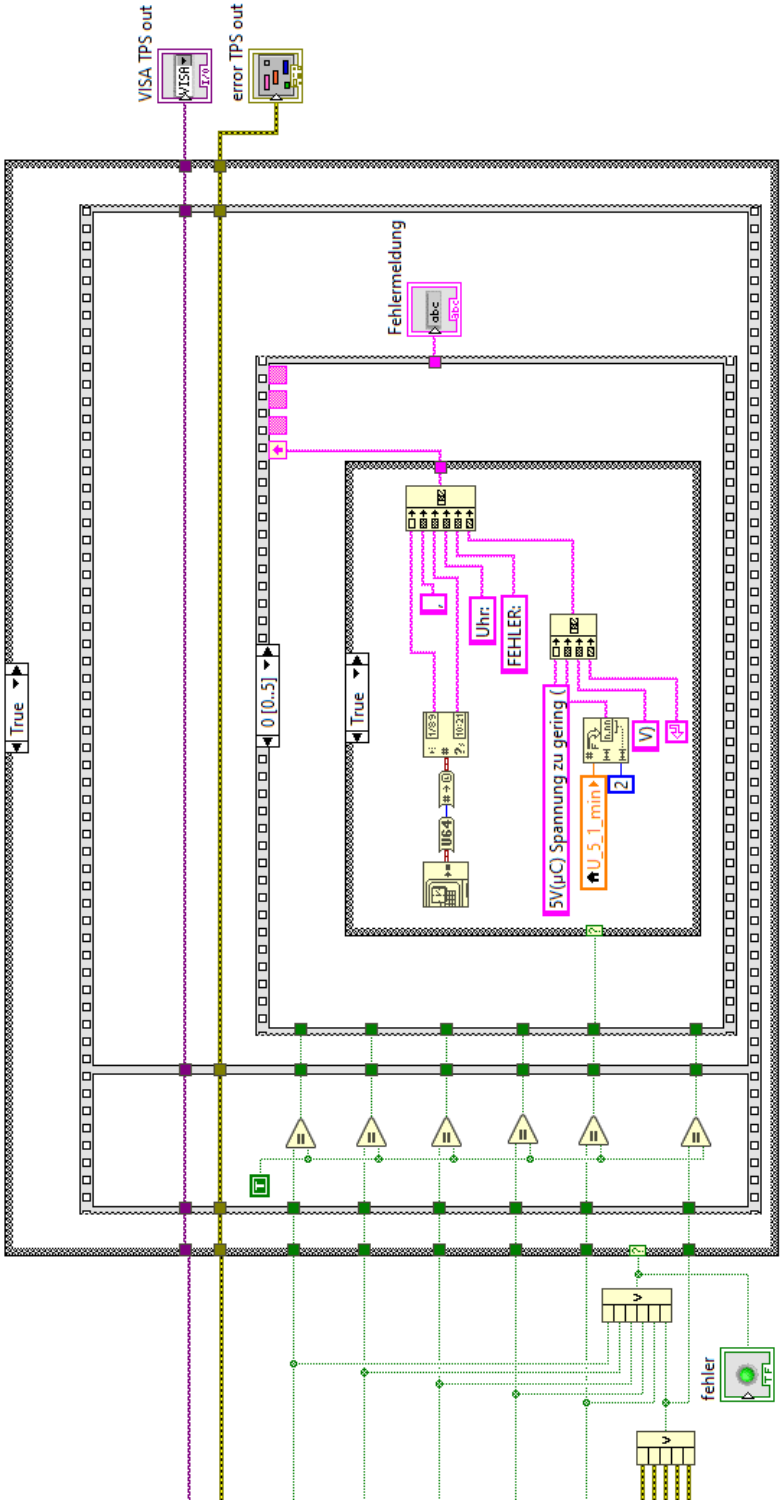


Abbildung 4.29: Ausschnitt des Blockdiagramms vom SubVI „ZMX_Spannungsmessung“

Die gemessenen Spannungsverläufe werden im zugehörigen Oszillographen im Frontpanel dargestellt. Zusätzlich werden die Spannungswerte in einer Excel-Tabelle abgespeichert. Dies geschieht gemäß dem Blockdiagramm der nachfolgenden Abbildung 4.30.

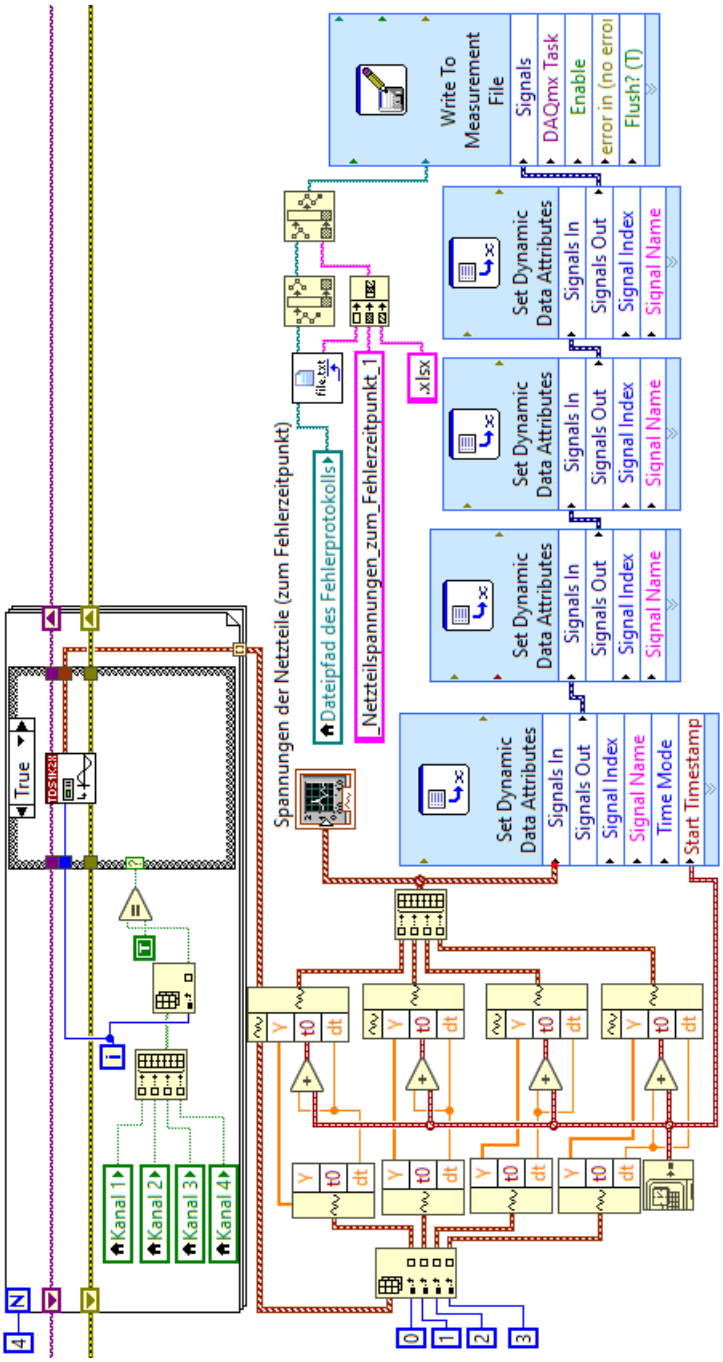


Abbildung 4.30: Gemessene Netzteilspannungen zum Fehlerzeitpunkt im Oszillographen in LabVIEW darstellen und als Excel-Tabelle abspeichern

SubVI „ZMX_Signalueberwachung“

Die folgende Abbildung 4.31 zeigt einen Ausschnitt des Blockdiagramms vom SubVI „ZMX_Signalüberwachung“.
Im Falle eines aufgetretenen Fehlers bei den ZMX-Endstufen und/oder den Signalen der Endlagenschalter wird vom Arduino Mega2560 (Signalüberwachung) die entsprechende Fehlermeldung seriell über USB an LabVIEW gesendet.
Eine genaue Beschreibung des Programms auf dem Mikrocontroller des Arduino folgt unter 4.2.1.
Wie in der Abbildung zu erkennen ist, wird in dem SubVI überwacht, ob Bytes an dem entsprechenden Port vorliegen. Ist dies der Fall, wird die empfangene Nachricht eingelesen und zum String „Fehlermeldung“ konkateniert.

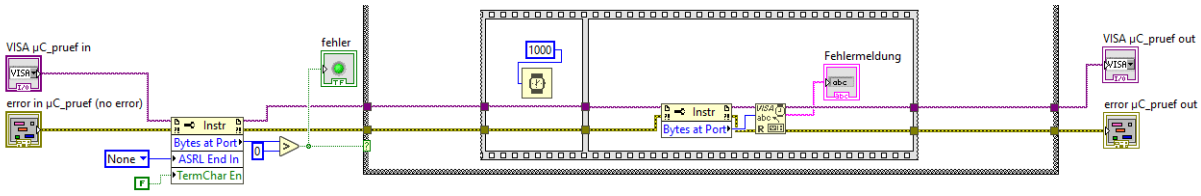


Abbildung 4.31: Ausschnitt des Blockdiagramms vom SubVI „ZMX_Signalüberwachung“

4.5 Fehlerprotokollierung

Die Fehlerprotokollierung erfolgt zum einen durch ein Fehlerprotokoll im „txt“-Format und zum anderen durch Erstellung von Excel-Tabellen. Das Fehlerprotokoll wird hierbei bei jeder Verwendung erweitert und alle enthaltenen Informationen mit Datum und Uhrzeit gekennzeichnet.
Exemplarische Ausschnitte eines Fehlerprotokolls folgen unter 5.3.
Die Excel-Tabellen dienen zur genauen Übersicht der gemessenen Spannungsverläufe. Somit wird zu jeder Messung von Einschaltflanken und bei jedem aufgetretenen Fehler eine neue Excel-Tabelle mit allen gemessenen Spannungswerten erzeugt. Zur graphischen Darstellung werden diese Kurvenverläufe während des Betriebs des Teststands im jeweiligen Oszillographen in LabVIEW angezeigt und können zusätzlich jederzeit anhand der Excel-Tabellen geplottet werden.

5 Verifizierung

5.1 Schrittmotorsteuerung

Zur Verifizierung der Funktionalität der Schrittmotorsteuerung wird die vorangegangene Arbeit des Praxissemesters als Grundlage genommen [6]. Diese hat sich bewährt und wird daher auch als Grundlage für die Ansteuerung eines ZMX-Überrahmens mit dem Teststand verwendet.
Die Ansteuerungsbefehle werden nun jedoch vom LabVIEW-Programm und nicht mehr über den „Serial Monitor“ der Arduino IDE gesendet.
Die Integration der bestehenden Ansteuerung zum LabVIEW-Programm funktioniert erwartungsgemäß.

5.2 Funktion des Programms zur Signalüberwachung

Bei der Verifizierung des Programms zur Signalüberwachung mit einem Arduino Mega2560 Entwicklungsboard gilt es zwei Aspekte zu überprüfen: Zum einen die Erkennung von Fehlern bei den 16 ZMX-Endstufen, zum anderen die Funktion der implementierten Endlagenschalter und der Vergleich des hierbei eingestellten Signals mit dem zugehörigen Signal der Anschlussmodule. Die gemeldeten Fehler werden seriell ausgeben, wodurch der „Serial Monitor“ der Arduino IDE hierbei zur Funktionsprüfung hilfreich ist.

Die Fehlersignale der ZMX-Endstufen sind high-aktiv, was bedeutet, dass ein aufgetretener Fehler bei einer Endstufe mit einer logischen 1 signalisiert wird [7]. Gemäß dem Schaltplan im Anhang unter A.2.1 werden zur galvanischen Trennung der unterschiedlichen Massepotentiale, wie bereits unter 4.3.3 beschrieben, Optokoppler verwendet. Diese sind vom Typ HCPL2631 und haben eine invertierende Funktion. Durch die realisierte Schaltung hat somit eine logische 1 beim Fehlersignal der Endstufe auch eine logische 1 am Ausgang des zugehörigen Optokopplers zur Folge.
Die Ausgangssignale sind mit den als Eingang konfigurierten Pins des Arduino Mega2560 verbunden. Zur Fehlererkennung kann daher im Programm abgefragt werden, ob alle Signale an den Pins eine logische 0 vorweisen. Ist dies nicht der Fall, liegt bei der jeweiligen Endstufe ein Fehler vor.

Um einen Fehler zu simulieren, kann der jeweilige Eingang des Optokopplers offen gelassen werden. Da die Fehlersignale mit den Kathoden der Optokoppler verbunden sind, führt ein offener Eingang ebenfalls zu einer logischen 1 am Ausgang des Optokopplers. Beim Testen dieser Funktion wird verifiziert, dass Fehler bei jeder Endstufe zuverlässig erkannt werden.

Zur Überprüfung der Fehlererkennung von Endlagenschaltern wird zunächst die Funktion von den simulierten Endlagenschaltern generell bestätigt. Der Aufbau erfolgt hierbei gemäß den Blockschaltbildern in den Abbildungen 3.1 und 4.5, sowie dem Schaltplan A.2.1. Zur Funktionsprüfung werden die die Schrittmotoren im manuellen Modus des Teststands angesteuert und die Kippschalter in der Testbox zur Simulation einer Endlage betätigt. Da die jeweiligen Schrittmotoren aufhören, sich in die Richtung zu drehen, wenn der Schalter betätigt wurde, ist diese Funktion wie gewünscht erfüllt.
Die Fehlerkennung bei Endlagenschaltern wird geprüft, indem der jeweilige DIP-Schalter im Modul „Anzeige Endlagen“ des ZMX-Überrahmens so eingestellt wird, dass die Endlagenschalter gebrückt sind. Dadurch reagiert das zugehörige Signal vom Anschlussmodul nicht auf Änderungen bei Signalen der Endlagenschalter und liefert kontinuierlich eine logische 0. Bei Betätigung eines Kippschalters für die Simulation einer Endlage entsteht eine logische 1 am Eingangspin des Mikrocontrollers des Arduino Mega2560 Entwicklungsboard. Da diese Ungleichheit der Signale vom Programm erkannt wird und auf dem „Serial Monitor“ angezeigt wird, ist die Fehlererkennung damit bestätigt.

Der zugehörige Programmcode befindet sich im Anhang unter A.3.

5.3 Fehlererkennung beim Betrieb des Teststands

Die folgende Abbildung 5.1 zeigt einen exemplarischen Ausschnitt aus einem Fehlerprotokoll, nachdem ein Fehler bei einem der Endlagenschalter-Signalen festgestellt wurde. Aus dieser Abbildung lassen sich zusätzlich zur Fehlermeldung Datum, Uhrzeit, sowie Betriebsmodus des Teststands und die zu diesem Zeitpunkt angesteuerten Schrittmotoren entnehmen.
Diese Fehlermeldung lässt sich zur Verifizierung der Funktionalität, wie unter 5.2 beschrieben, provozieren.

```
05.08.2021, 09:53Uhr: START: AUTOMATIK
-------------------------------------------------------
05.08.2021, 09:53Uhr: FEHLER: Fehlermeldung Limitsignale: CCW(Reihe B)
05.08.2021, 09:53Uhr: Motor: 1(RECHTS) Pulsfrequenz: 181.82Hz
-------------------------------------------------------
05.08.2021, 09:53Uhr: BEENDET: AUTOMATIK (Laufzeit: 0 Stunden, 0 Minuten, 4 Sekunden)
```

Abbildung 5.1: Exemplarisches Fehlerprotokoll bei einer Fehlermeldung von Endlagenschaltern

Die Abbildung 5.2 stellt eine exemplarische Ausgabe in einem Fehlerprotokoll dar, nachdem ein Fehler von einer ZMX-Endstufe gemeldet wurde.
Hierbei befand sich der Teststand im manuellen Betriebsmodus, daher ist in dieser Abbildung, anders als in Abbildung 5.1, keine Information zu den angesteuerten Schrittmotoren. Da die Schrittmotoren im manuellen Betriebsmodus von Benutzer direkt gesteuert werden, ist diese Information in diesem Fall nicht erforderlich.
Diese Fehlermeldung lässt sich zum Funktionstest provozieren, indem die Endstufe 1 nicht im ZMX-Überrahmen eingesetzt wird. Der Grund hierfür ist unter 5.2 beschrieben.

```
05.08.2021, 09:20Uhr: START: MANUELL
-----------------------------------------------------------
05.08.2021, 09:20Uhr: FEHLER: Fehlermeldung Endstufen: Endstufe 1
-----------------------------------------------------------
05.08.2021, 09:20Uhr: BEENDET: MANUELL (Laufzeit: 0 Stunden, 0 Minuten, 2 Sekunden)
```

Abbildung 5.2: Exemplarisches Fehlerprotokoll bei einer Fehlermeldung von ZMX-Endstufen

In der nachfolgenden Abbildung 5.3 ist eine exemplarische Ausgabe eines Fehlerprotokolls zu erkennen, nachdem ein Fehler bei einer der Netzteilspannungen festgestellt wurde.
Um diese Fehlermeldung zu provozieren kann der jeweilige, am Oszilloskop angeschlossene Tastkopf, entfernt werden. Dadurch wird eine Spannung von $0,00\,\mathrm{V}$ gemessen, welche unterhalb der festgelegten minimalen Schwelle liegt.

```
05.08.2021, 11:55Uhr: START: MANUELL
-----------------------------------------------------------
05.08.2021, 11:56Uhr: FEHLER: 15V Spannung zu gering (0,00V)
-----------------------------------------------------------
05.08.2021, 11:56Uhr: BEENDET: MANUELL (Laufzeit: 0 Stunden, 0 Minuten, 22 Sekunden)
```

Abbildung 5.3: Exemplarisches Fehlerprotokoll bei einer Fehlermeldung von Netzteilspannungen

6 Zusammenfassung und Ausblick

Der entwickelte Teststand bietet die Möglichkeit, ZMX-Überrahmen automatisch oder manuell anzusteuern und ist ohne spezielle Vorkenntnisse leicht zu bedienen. Bei dem der Bedienung handelt es sich um eine Kombination aus Hardware- und Softwarebedienelementen. Die Hardwarebedienelemente befinden sich in der entwickelten Testbox. Die softwareseitige Bedienung erfolgt über ein mit LabVIEW erstelltes Frontpanel.
Hierbei besteht die Möglichkeit, die Einschaltflanken des 5 V-Netzteils (DSP30-5) und des 15 V-Netzteils (DSP60-15) zu messen, um die korrekte Einschaltreihenfolge zu verifizieren. Eine kontinuierliche Überwachung erfolgt bei den Netzteilspannungen aller Netzteile eines ZMX-Überrahmens, bei den Endlagenschaltersignalen von vier wählbaren Schrittmotoren sowie bei den Fehlersignalen aller 16 ZMX-Endstufen.
Alle hierbei auftretenden Fehler werden in einem Fehlerprotokoll gespeichert und mit einem Zeitstempel versehen. Bei Fehlern der Netzteilspannungen wird zusätzlich eine Excel-Tabelle mit den gemessenen Werten erstellt.
Sofern ein Fehler aufgetreten ist, wird dies im Frontpanel der graphischen Bedienoberfläche signalisiert und das System wird abgeschaltet.
Die folgende Abbildung 6.1 zeigt den entwickelten Teststand im Betrieb.

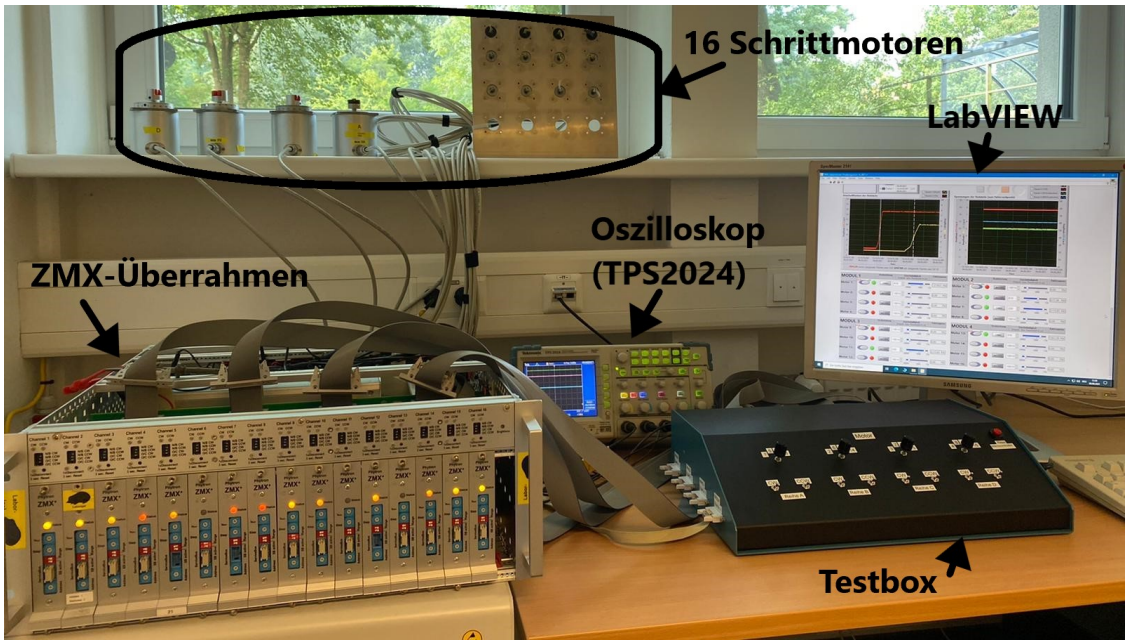


Abbildung 6.1: Aufgebauter Teststand im Betrieb mit allen Hardwarekomponenten

Weiterführend kann dieser Teststand dahingehend erweitert werden, dass der Service-Bus des ZMX-Überrahmens zur Signalüberwachung mit integriert wird. Über den Service-Bus lassen sich Werte von jeder ZMX-Endstufe auslesen. Darüber wäre es dann auch möglich, im Fehlerfall einer Endstufe die genaue Fehlermeldung zu erhalten.
Für die Kommunikation mit dem Service-Bus kann die Programmiersprache Python verwendet werden. Da LabVIEW die Möglichkeit bietet, verschiedene Skripte mit in das Programm einzubinden, sollte es auch generell möglich sein ein solches Python Skript, welches Werte von den ZMX-Endstufen ausliest, mit in das bestehende Programm „ZMX_Ueberrahmen_Pruefprogramm“ zu integrieren.
Für diese Bachelorarbeit hätte die Entwicklung eines Python Skripts und dessen Integration in LabVIEW jedoch den Rahmen gesprengt.

Literatur

[1] *Arduino Mega2560*. URL: https://store.arduino.cc/arduino-mega-2560-rev3 (besucht am 16. 08. 2021).

[2] *Arduino Nano*. URL: https://store.arduino.cc/arduino-nano (besucht am 16. 08. 2021).

[3] Deutsches Elektronen-Synchrotron DESY. *Interne Unterlagen*. Erhältlich über DESY, Gruppe ZE.

[4] *LabVIEW*. URL: https://de.wikipedia.org/wiki/LabVIEW (besucht am 16. 08. 2021).

[5] *LabVIEW*. URL: https://www.ni.com/getting-started/labview-basics/d/environment (besucht am 16. 08. 2021).

[6] Marcel Soika. *Praktikumsbericht: Analyse und Ansteuerung einer ZMX-Motorsteuerung*.

[7] Phytron. *ZMX+ Schrittmotorendstufe mit ServiceBus Hardware V5.0*. URL: https://www.phytron.de/fileadmin/user_upload/produkte/endstufen_controller/pdf/ma-zmxplus-de.pdf (besucht am 16. 08. 2021).

[8] *RS232*. URL: https://de.wikipedia.org/wiki/RS-232 (besucht am 16. 08. 2021).

[9] *split()-Funktion*. URL: http://wp.scalesoft.de/arduino-split/ (besucht am 16. 08. 2021).

[10] *Tektronix TPS 2024*. URL: https://de.tek.com/datasheet/digital-storage-oscilloscopes-1 (besucht am 16. 08. 2021).

A Anhang

A.1 Quellcode

A.1.1 Programm für die Schrittmotoransteuerung auf dem Arduino Mega2560 Entwicklungsboard

Listing A.1: Programm für die Schrittmotoransteuerung auf dem Arduino Mega2560 Entwicklungsboard

```
#include <SoftwareSerial.h>
#include <math.h>
#include <Wire.h>
#include <string.h>
/*-------------------------------VARIABLEN UND KONSTANTEN-------------------------------*/
#define CYCLE_MAX 5010
#define CYCLE_MIN 190
#define CYCLE_MOTOR_AUS 22
#define CYCLE_SCHRITTWEITE 20

enum Richtung {rechts, links};
bool Drehrichtung[16] = {};
String richtung[16] = {};

unsigned int cycle = CYCLE_MAX;
unsigned int cycle_motoren[16] = {};
unsigned int cycle_motoren_temp[16] = {};

// Modul 1 -------- Drehrichtung vom Arduino MEGA gesteuert
const int M1_dir = 38;
const int M2_dir = 39;
const int M3_dir = 40;
const int M4_dir = 41;
// Modul 2 -------- Drehrichtung vom Arduino MEGA gesteuert
const int M5_dir = 42;
const int M6_dir = 43;
const int M7_dir = 44;
const int M8_dir = 45;
// Modul 3 -------- Drehrichtung vom Arduino MEGA gesteuert
const int M9_dir = 46;
const int M10_dir = 47;
const int M11_dir = 48;
const int M12_dir = 49;
// Modul 4 -------- Drehrichtung vom Arduino MEGA gesteuert
const int M13_dir = 50;
const int M14_dir = 51;
const int M15_dir = 52;
const int M16_dir = 53;
int Motor_Dir[16] = {M1_dir, M2_dir, M3_dir, M4_dir, M5_dir, M6_dir, M7_dir, M8_dir, M9_dir,
    M10_dir, M11_dir, M12_dir, M13_dir, M14_dir, M15_dir, M16_dir};

String command_Motor[16] = {};
String command_Motoren;
volatile char command;
volatile char temp_command;

String command_modus = {};
int max_motoren = 8;
String temp_max_motoren = {};
```

```
53 /*-------------------------------------------PROTOTYPEN-------------------------------------------*/
54 void(* resetFunc) (void) = 0;                                  // Reset Funktion zur Adresse 0

56 // warten auf Serielle Pauseanforderung im automatischen Betrieb
57 void pause(unsigned int Geschwindigkeit_1, unsigned int Motoren, int Motornummer, unsigned
       int Geschwindigkeit_2);
58 void warten(unsigned long dauer, unsigned int Geschwindigkeit_1, unsigned int Motoren, int
       Motornummer, unsigned int Geschwindigkeit_2);

60 void drehrichtung (int *Motor_DIR, bool *DIR);          // DIR: 0 oder 1 (0=>rechts, 1=>links)
61 String split(String s, char parser, int index);        // String zerteilen
62 // Drehrichtung des jeweiligen Motors ueber Serial.print ausgeben
63 void print_richtung(int index);

65 // unsigned int Motoren => Dezimalwert eines Binaermusters, int Motornummer => Motornummer
       minus 1 (fuer Motor 1: int Motornummer=0;)
66 void Motoren_an(unsigned long dauer, unsigned int Geschwindigkeit_1, unsigned int Motoren,
       int Motornummer, unsigned int Geschwindigkeit_2);
67 void i2c_Uebertragung(unsigned int *cycle_motoren, unsigned int *cycle_motoren_temp, int
       i2c_addr);// Datenuebertragung an einen Arduino NANO

69 void manuell();                                            // manueller Betrieb
70 void automatik_1();                                       // Automatikbetrieb 1
71 void automatik_2();                                       // Automatikbetrieb 2
72 /*-------------------------------------------SETUP-------------------------------------------*/
73 void setup() {
74   Serial.begin(9600);                  // Baudrate: 9600bps
75   Wire.begin();                      // I2C

77   pinMode(13, OUTPUT);                 // Reset am Arduino NANO
78   pinMode(3, INPUT);                   // Interrupt PIN
79   attachInterrupt(digitalPinToInterrupt(3), stopp_alles, RISING);

81   // Resetimpuls an den Arduino NANOs ausloesen
82   digitalWrite(13, LOW);
83   delayMicroseconds(50);
84   // PIN 38 bis 53 als Ausgang definieren (fuer Aenderung der Drehrichtung)
85   for (int i = 38; i < 54; i++) {
86     pinMode(i, OUTPUT);
87   }
88   for (int n = 0; n < 16; n++) {  // Richtung zuweisen: Alle Motoren Drehrichtung rechts
89     Drehrichtung[n] = Richtung(rechts);
90     drehrichtung(&Motor_Dir[n], &Drehrichtung[n]);
91   }
92   digitalWrite(13, HIGH);
93 }
94 /*-------------------------------------------LOOP() SCHLEIFE-------------------------------------------*/
95 void loop() {
96   if (Serial.available() > 0) {                              // Auswahl des Modus
97     command_modus = Serial.readString();
98     temp_command = command_modus[0];
99     temp_max_motoren = (split(command_modus, '_', 1));
100    if ((temp_command == 'A') || (temp_command == 'B') || (temp_command == 'C')) {
101      command = temp_command;
102      max_motoren = temp_max_motoren.toInt();
103    }
104  }
105  if (command == 'A' ) {                                     // Automatikmodus 1
106    automatik_1();
107  }
108  if (command == 'C' ) {                                     // Automatikmodus 2
109    automatik_2();
110  }
111  if (command == 'B' ) {                                     // manueller Modus
112    manuell();

114    /*-------------------------------MANUELLEN MODUS TESTEN-------------------------------*/
115    /*String mit Informationen zu den Motoren, zum testen ueber den "Serial Monitor":-------*/
116    //1a200_22_22_22_200_22_22_22_200_22_22_22_200_22_22_22_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0_e
117    /*"a"-> Anfang des erwarteten Strings, "_" -> Trennzeichen, "e" -> Ende des Strings-----*/
118    /*vorderen 16 Zahlen->Impulszeit in Mikrosekunden (max:20000,min:110,AUS:22)-----------*/
119    /*----------------------resultierende Pulsfrequenz = (10^6)/(2*Impulszeit)-------------*/
120    /*hinteren 16 Zahlen->Richtung (0:rechts, 1:links)-----------------------------------*/
121    /*----------------------------------------------------------------------------------*/
122  }
123 }
124 /*-----------------FUNKTIONEN FUER NEUSTART DES PROGRAMMS DURCH INTERRUPT(PIN3)----------------*/
125 void stopp_alles() {
126   digitalWrite(13, LOW);
127   resetFunc();
128 }
```

```
130 /*-------------------------DATENUEBERTRAGUNG AN EINEN ARDUINO NANO-------------------------*/
131 void i2c_Uebertragung(unsigned int *cycle_motoren, unsigned int *cycle_motoren_temp, int
        i2c_addr) {
132   int array_idx = (i2c_addr - 1) * 4;
133   if ((cycle_motoren[array_idx] != cycle_motoren_temp[array_idx]) || (cycle_motoren[array_idx
        + 1] != cycle_motoren_temp[array_idx + 1])
134       || (cycle_motoren[array_idx + 2] != cycle_motoren_temp[array_idx + 2]) || (
        cycle_motoren[array_idx + 3] != cycle_motoren_temp[array_idx + 3])) {
135     cycle_motoren[array_idx] = cycle_motoren_temp[array_idx];
136     cycle_motoren[array_idx + 1] = cycle_motoren_temp[array_idx + 1];
137     cycle_motoren[array_idx + 2] = cycle_motoren_temp[array_idx + 2];
138     cycle_motoren[array_idx + 3] = cycle_motoren_temp[array_idx + 3];
139     Wire.beginTransmission(i2c_addr);
140     Wire.write((byte*)&cycle_motoren[array_idx], 4 * sizeof(unsigned int));
141     Wire.endTransmission();
142   }
143 }
144 /*-----PAUSE (warten auf Serielle Pause- oder Stoppanforderung im automatischen Betrieb)-----*/
145 void pause(unsigned int Geschwindigkeit_1, unsigned int Motoren, int Motornummer, unsigned
        int Geschwindigkeit_2) {
146   char current_command = command;
147   if (Serial.available() > 0) {
148     temp_command = (Serial.read());
149     if ((temp_command == 'P')) {                                  // Pause
150       command = temp_command;
151       Serial.print(command);
152       while ((command != current_command)) {
153         if (Serial.available() > 0) {
154           temp_command = Serial.read();
155           if ((temp_command == current_command)) {                // Programm weiter laufen lassen
156             command = temp_command;
157           }
158         }
159       }
160     }
161     if ((temp_command == 'S')) {                                  // Stopp
162       command = temp_command;
163       if ((Geschwindigkeit_1 == CYCLE_MOTOR_AUS) && (Motoren == 0)) {
164         if (Geschwindigkeit_2 != CYCLE_MOTOR_AUS) {
165           Serial.print("Motor: ");
166           Serial.print(Motornummer + 1);
167           print_richtung(Motornummer);
168           Serial.print(" Pulsfrequenz: ");
169           Serial.print((1 / ((float)Geschwindigkeit_2 * 2))*pow(10, 6));
170           Serial.println("Hz");
171         } else Serial.println("alle Motoren aus");
172       }
173       else if ((Geschwindigkeit_1 == Geschwindigkeit_2) && (current_command == 'A')) {
174         if (Geschwindigkeit_2 != CYCLE_MOTOR_AUS) {
175           int i = 0;
176           for (int n = 0; n < 16; n++) {
177             if (Motoren & (1 << n)) {
178               i++;
179             }
180           }
181           if (i == 1 ) {
182             Serial.print("Motor: ");
183           } else Serial.print("Motoren: ");
184           for (int n = 0, i = 0; n < 16; n++) {
185             if (Motoren & (1 << n)) {
186               i++;
187               if (i > 1) {
188                 Serial.print(", ");
189               }
190               Serial.print(n + 1);
191               print_richtung(n);
192             }
193           }
194           Serial.print(" Pulsfrequenz: ");
195           Serial.print((1 / ((float)Geschwindigkeit_2 * 2))*pow(10, 6));
196           Serial.println("Hz");
197         } else Serial.println("alle Motoren aus");
198       }
199       else if ((Motornummer >= 0) && ((current_command == 'C'))) {
200         if ((Geschwindigkeit_1 == CYCLE_MOTOR_AUS) && (Geschwindigkeit_2 == CYCLE_MOTOR_AUS))
        {
201           Serial.println("alle Motoren aus");
202         }
203         else {
204           if ((Motoren > 0)) {
205             int i = 0;
206             for (int n = 0; n < 16; n++) {
```

```
207                 if (Motoren & (1 << n)) {
208                   i++;
209                 }
210               }
211               if ((Motoren & (1 << Motornummer)) == 0)  {
212                 if (i == 1 ) {
213                   Serial.print("Motor: ");
214                 } else Serial.print("Motoren: ");

216                 for (int n = 0, i = 0; n < 16; n++) {
217                   if (Motoren & (1 << n)) {
218                     i++;
219                     if (i > 1) {
220                       Serial.print(", ");
221                     }
222                     Serial.print(n + 1);
223                     print_richtung(n);
224                   }
225                 }
226                 Serial.print(" Pulsfrequenz: ");
227                 Serial.print((1 / ((float)Geschwindigkeit_1 * 2))*pow(10, 6));
228                 Serial.print("Hz");
229                 if (Geschwindigkeit_2 != CYCLE_MOTOR_AUS) {
230                   Serial.print(" und ");
231                 } else Serial.println();
232               }
233             }
234             if (Geschwindigkeit_2 != CYCLE_MOTOR_AUS) {
235               Serial.print("Motor: ");
236               Serial.print(Motornummer + 1);
237               print_richtung(Motornummer);
238               Serial.print(" Pulsfrequenz: ");
239               Serial.print((1 / ((float)Geschwindigkeit_2 * 2))*pow(10, 6));
240               Serial.println("Hz");
241             }
242           }
243         }
244         while ((command != current_command)) {
245           if (Serial.available() > 0) {
246             temp_command = Serial.read();
247             if ((temp_command == 'X')) {                        // Reset durchfuehren
248               stopp_alles();
249             }
250             if ((temp_command == current_command)) {            // Programm weiter laufen lassen
251               command = temp_command;
252             }
253           }
254         }
255       }
256     }
257   }
258   /*-----------------------------------------WARTE FUNKTION-----------------------------------------*/
259   void warten(unsigned long dauer, unsigned int Geschwindigkeit_1, unsigned int Motoren, int
        Motornummer, unsigned int Geschwindigkeit_2) {
260     // aktuelle Geschwindigkeit fuer "dauer"-Schleifendurchlaeufe beibehalten
261     for (unsigned long warte = 0; warte <= dauer; warte++) {
262       // warten auf Pause Befehl
263       pause(Geschwindigkeit_1, Motoren, Motornummer, Geschwindigkeit_2);
264     }
265   }
266   /*-----------------------------FUNKTION FUER MOTORDREHRICHTUNG-----------------------------*/
267   void drehrichtung (int *Motor_DIR, bool * DIR) {
268     digitalWrite(*Motor_DIR, *DIR);             // DIR: 0 oder 1 (0=>rechts, 1=>links)
269   }
270   /*-----------------------------Funktion zum String zerteilen-----------------------------*/
271   String split(String s, char parser, int index) {
272     String rs = "";
273     //int parserIndex = index;
274     int parserCnt = 0;
275     int rFromIndex = 0, rToIndex = -1;
276     while (index >= parserCnt) {
277       rFromIndex = rToIndex + 1;
278       rToIndex = s.indexOf(parser, rFromIndex);
279       if (index == parserCnt) {
280         if (rToIndex == 0 || rToIndex == -1) return "";
281         return s.substring(rFromIndex, rToIndex);
282       } else parserCnt++;
283     }
284     return rs;
285   }
```

```
288 /*------------------------------DREHRICHTUNG SERIELL WEITERGEBEN------------------------------*/
289 void print_richtung(int index) {
290   switch ((int)Drehrichtung[index]) {
291     case 0: Serial.print("(RECHTS)");
292       break;
293     case 1: Serial.print("(LINKS)");
294       break;
295   }
296 }
297 /*----------------FUNKTIONEN FUER BELIEBIGE KOMBINATIONEN DER MOTOREN----------------*/
298 void Motoren_an(unsigned int Geschwindigkeit_1, unsigned int Motoren, int Motornummer,
      unsigned int Geschwindigkeit_2) {
299   // jedem Motor eine Geschwindigkeit zuweisen
300   for (int n = 0; n < 16; n++) {
301     if (Motoren & (1 << n)) {
302       cycle_motoren_temp[n] = Geschwindigkeit_1;
303     } else cycle_motoren_temp[n] = CYCLE_MOTOR_AUS;
304   }
305   if ((Motornummer >= 0) && (Motornummer < 16)) {
306     // einem Motor eine andere Geschwindigkeit zuweisen
307     cycle_motoren_temp[Motornummer] = Geschwindigkeit_2;
308   }
309   if ((Motornummer >= 4) || (Motoren >= 16)) {
310   // Datenuebertragung an einen Arduino NANO fuer Ansteuerung der Motoren 1 bis 4
311   i2c_Uebertragung(cycle_motoren, cycle_motoren_temp, 1);
312     // Datenuebertragung an einen Arduino NANO fuer Ansteuerung der Motoren 5 bis 8
313     i2c_Uebertragung(cycle_motoren, cycle_motoren_temp, 2);
314     if ((Motornummer >= 8) || (Motoren >= 256)) {
315         // Datenuebertragung an einen Arduino NANO fuer Ansteuerung der Motoren 9 bis 12
316       i2c_Uebertragung(cycle_motoren, cycle_motoren_temp, 3);
317       if ((Motornummer >= 12) || (Motoren >= 4096)) {
318         // Datenuebertragung an einen Arduino NANO fuer Ansteuerung der Motoren 13 bis 16
319         i2c_Uebertragung(cycle_motoren, cycle_motoren_temp, 4);
320       }
321     }
322   }
323   warten(1000, Geschwindigkeit_1, Motoren, Motornummer, Geschwindigkeit_2);
324 }
325 /*------------------------------------MANUELLER BETRIEB------------------------------------*/
326 void manuell() {
327   if ((Serial.readStringUntil('a').toInt() != 1)) {          // Warten auf String mit Daten
328   }
329   else {
330     // String mit Daten erhalten und bis zum Ende auslesen
331     command_Motoren = Serial.readStringUntil('e');
332     for (int i = 0; i < 32; i++) {
333       if (i < 16) {
334         // String zerteilen und Daten zuweisen
335         command_Motor[i] = split(command_Motoren, '_', i);
336         cycle_motoren_temp[i] = (unsigned int)command_Motor[i].toInt();
337       }
338       else {
339         richtung[i - 16] = (split(command_Motoren, '_', i));
340         switch (richtung[i - 16].toInt()) {
341           case 0:      Drehrichtung[i - 16] = 0;
342             break;
343           case 1:      Drehrichtung[i - 16] = 1;
344             break;
345         }
346         drehrichtung(&Motor_Dir[i - 16], &Drehrichtung[i - 16]);
347       }
348     }
349     // Datenuebertragung an einen Arduino NANO fuer Ansteuerung der Motoren 1 bis 4
350     i2c_Uebertragung(cycle_motoren, cycle_motoren_temp, 1);
351     // Datenuebertragung an einen Arduino NANO fuer Ansteuerung der Motoren 5 bis 8
352     i2c_Uebertragung(cycle_motoren, cycle_motoren_temp, 2);
353     // Datenuebertragung an einen Arduino NANO fuer Ansteuerung der Motoren 9 bis 12
354     i2c_Uebertragung(cycle_motoren, cycle_motoren_temp, 3);
355     // Datenuebertragung an einen Arduino NANO fuer Ansteuerung der Motoren 13 bis 16
356     i2c_Uebertragung(cycle_motoren, cycle_motoren_temp, 4);
357   }
358 }
359 /*------------------------------------AUTOMATIKBETRIEB 1------------------------------------*/
360 void automatik_1 () {
361   // Richtung wechseln fuer alle Motoren
362   for (int dir = 0; dir < 2; dir++) {
363   // Richtung zuweisen: dir=0(RECHTS),dir=1(LINKS)
364     for (int n = 0; n < 16; n++) {
365       Drehrichtung[n] = Richtung(dir);
366       drehrichtung(&Motor_Dir[n], &Drehrichtung[n]);
367     }
368     // jeden Motor einmal alleine starten
```

```
369     for (int motor = 0; motor < 16; motor++) {
370         // Geschwindigkeit erhoehen
371       for (cycle = CYCLE_MAX; cycle >= CYCLE_MIN; cycle = cycle - CYCLE_SCHRITTWEITE) {
372         Motoren_an(CYCLE_MOTOR_AUS, 0, motor, cycle);
373       }
374       Motoren_an(CYCLE_MOTOR_AUS, 0, motor, CYCLE_MIN);
375       warten(300000, CYCLE_MOTOR_AUS, 0, motor, CYCLE_MIN);
376       // Geschwindigkeit verringern
377       for (cycle = CYCLE_MIN; cycle <= CYCLE_MAX; cycle = cycle + CYCLE_SCHRITTWEITE) {
378         Motoren_an(CYCLE_MOTOR_AUS, 0, motor, cycle);
379       }
380       // Motor ausschalten
381       Motoren_an(CYCLE_MOTOR_AUS, 0, motor, CYCLE_MOTOR_AUS);
382       warten(300000, CYCLE_MOTOR_AUS, 0, motor, CYCLE_MOTOR_AUS);
383     }
384   }
385   // Richtung wechseln fuer alle Motoren
386   for (int dir = 0; dir < 2; dir++) {
387   // Richtung zuweisen: dir=0(RECHTS),dir=1(LINKS)
388     for (int n = 0; n < 16; n++) {
389       Drehrichtung[n] = Richtung(dir);
390       drehrichtung(&Motor_Dir[n], &Drehrichtung[n]);
391     }
392     for (int i = 0; i <= 16 - max_motoren; i++) {
393     // binaer aufwaerts zaehlen, Motoren in verschiedenen Kombinationen laufen lassen
394       for (long anzahl = 1; anzahl <= (long)pow(2, max_motoren) ; anzahl++) {
395         // jedem Motor eine Drehrichtung zuweisen
396         for (int n = 0; n < 16; n++) {
397           if ((anzahl << i) & (1 << n)) {
398             Drehrichtung[n] = !Drehrichtung[n];                    // Drehrichtung toggeln
399           } else Drehrichtung[n] = Drehrichtung[n];
400           drehrichtung(&Motor_Dir[n], &Drehrichtung[n]);
401         }
402         // Geschwindigkeit erhoehen
403         for (cycle = CYCLE_MAX; cycle >= CYCLE_MIN; cycle = cycle - CYCLE_SCHRITTWEITE) {
404           Motoren_an(cycle, anzahl << i, -1, cycle);
405         }
406         Motoren_an(CYCLE_MIN, anzahl << i, -1, CYCLE_MIN);
407         warten(300000, CYCLE_MIN, anzahl << i, -1, CYCLE_MIN); //ca. 8 Sekunden bei 800000
408         // laufende Motoren bei der Endgeschwindigkeit eine laengere Zeit laufen lassen
409         if (anzahl == (long)pow(2, max_motoren)) {
410           warten(1700000, CYCLE_MIN, anzahl << i, -1, CYCLE_MIN);
411         }
412         // Geschwindigkeit verringern
413         for (cycle = CYCLE_MIN; cycle <= CYCLE_MAX; cycle = cycle + CYCLE_SCHRITTWEITE) {
414           Motoren_an(cycle, anzahl << i, -1, cycle);
415         }
416          // Alle laufenden Motoren ausschalten
417         Motoren_an(CYCLE_MOTOR_AUS, anzahl << i, -1, CYCLE_MOTOR_AUS);
418         warten(300000, CYCLE_MOTOR_AUS, anzahl << i, -1, CYCLE_MOTOR_AUS);
419       }
420     }
421   }
422 }
423 /*-----------------------------------AUTOMATIKBETRIEB 2-----------------------------------*/
424 void automatik_2 () {
425   for (int dir = 0; dir < 4; dir++) {                                                    //
      Richtung wechseln fuer alle Motoren
426     if (dir < 2) {                                                                       //
      Richtung zuweisen: dir=0(RECHTS),dir=1(LINKS)
427       for (int i = 0; i < 16; i++) {
428         Drehrichtung[i] = Richtung(dir);
429         drehrichtung(&Motor_Dir[i], &Drehrichtung[i]);
430       }
431     }
432     else if (dir == 2) {                                                                 //
      Richtung zuweisen: ungerade Motornummern(RECHTS), gerade Motornummern(LINKS)
433       for (int i = 0; i < 16; i++) {
434         if ((i % 2 == 0) || (i == 0)) {
435           Drehrichtung[i] = Richtung(rechts);
436           drehrichtung(&Motor_Dir[i], &Drehrichtung[i]);
437         }
438         else {
439           Drehrichtung[i] = Richtung(links);
440           drehrichtung(&Motor_Dir[i], &Drehrichtung[i]);
441         }
442       }
443     }
444     else {                                                                               //
      Richtung zuweisen: ungerade Motornummern(LINKS), gerade Motornummern(RECHTS)
445       for (int i = 0; i < 16; i++) {
446         Drehrichtung[i] = !Drehrichtung[i];
```

```
447           drehrichtung(&Motor_Dir[i], &Drehrichtung[i]);
448         }
449       }
450       for (int j = 0; j <= (16 - max_motoren); j++) {
451         for (long i = 0, anzahl = 1; i < max_motoren; i++) {
      // Motoren nacheinander anschalten, Beginn mit Motor 1
452           if (i > 0) {
453             anzahl = (1 << i);
454           }
455           for (cycle = CYCLE_MAX; cycle >= CYCLE_MIN; cycle = cycle - CYCLE_SCHRITTWEITE) {
      // Geschwindigkeit erhoehen
456             Motoren_an(CYCLE_MIN, (anzahl - 1) << j, (i + j) , cycle);
457           }
458           Motoren_an(CYCLE_MIN, (anzahl - 1) << j, (i + j)  , CYCLE_MIN);
459           warten(600000, CYCLE_MIN, (anzahl - 1) << j, (i + j)  , CYCLE_MIN);
460         }
461         warten(1700000, CYCLE_MIN, ((long) pow(2, (int)max_motoren) - 1) << j, -1, CYCLE_MIN);
462         for (long i = 0, anzahl = (long) pow(2, (int)max_motoren) + 1; i < max_motoren; i++) {
      // Motoren nacheinander auschalten, Beginn mit Motor 1
463           anzahl = anzahl - (1 << i);
464           for (cycle = CYCLE_MIN; cycle <= CYCLE_MAX; cycle = cycle + CYCLE_SCHRITTWEITE) {
      // Geschwindigkeit verringern
465             Motoren_an(CYCLE_MIN, (anzahl - 1) << j, (i + j)  , cycle);
466           }
467           Motoren_an(CYCLE_MIN, (anzahl - 1) << j, (i + j)  , CYCLE_MOTOR_AUS);
      // Motor ausgeschalten
468           warten(300000, CYCLE_MIN, (anzahl - 1) << j, (i + j)  , CYCLE_MOTOR_AUS);
469         }
470       }
471     }
472 }
473 /*-------------------------------------------------------------------------------------------------------------*/
474 /*-------------------------------------------PROGRAMMENDE------------------------------------------------------*/
475 /*-------------------------------------------------------------------------------------------------------------*/
```

A.1.2 Programm für die Schrittmotoransteuerung auf den Arduino Nano Entwicklungsboards

Listing A.2: Programm für die Schrittmotoransteuerung auf den Arduino Nano Entwicklungsboards

```
#include <Wire.h>
/*-------------------------------VARIABLEN UND KONSTANTEN-------------------------------*/
volatile unsigned int Cycle_motoren[4] = {22, 22, 22, 22}; // Wert "22" wird als
    ausgeschalteter Zustand interpretiert

// Module 1 bis 4 -------- Motoren angesteuert ueber Arduino NANO
const int Motoren_1_5_9_13_Puls = 4;
const int Motoren_2_6_10_14_Puls = 5;
const int Motoren_3_7_11_15_Puls = 6;
const int Motoren_4_8_12_16_Puls = 7;
const int Motor_Puls[4] = {Motoren_1_5_9_13_Puls, Motoren_2_6_10_14_Puls,
    Motoren_3_7_11_15_Puls, Motoren_4_8_12_16_Puls};

bool Motoren_1_5_9_13_State = false;
bool Motoren_2_6_10_14_State = false;
bool Motoren_3_7_11_15_State = false;
bool Motoren_4_8_12_16_State = false;
bool Motor_State[4] = {Motoren_1_5_9_13_State, Motoren_2_6_10_14_State,
    Motoren_3_7_11_15_State, Motoren_4_8_12_16_State};

unsigned long MotorLastMicros[4] = {};
unsigned long currentMicros;
/*-------------------------------------------PROTOTYPEN-------------------------------------------*/
void pegel_wechseln (unsigned long *M_LastMicros, volatile unsigned int *Cycle , const int *
    Motor_Puls, bool *State); // Pegel fuer Ansteuerung der Motoren aendern
/*-------------------------------------------SETUP-------------------------------------------*/
void setup()
{
  DDRD |= (B11110000);              // Digitalpins D4 bis D7 als Ausgang definieren
  PORTD &= ~(B11110000);            // Digitalpins D4 bis D7 auf LOW-Pegel setzen
  Wire.begin(3);                    // I2C: Adresse 1 -> Modul 1, Adresse 2 -> Modul 2,
    Adresse 3 -> Modul 3, Adresse 4 -> Modul 4
  Wire.onReceive(receiveEvent);
}
/*-------------------------------------------LOOP() SCHLEIFE-------------------------------------------*/
void loop()
{
  pegel_wechseln(&MotorLastMicros[0], &Cycle_motoren[0], &Motor_Puls[0], &Motor_State[0]);
}
/*-------------------------------------------EVENT HANDLER-------------------------------------------*/
void receiveEvent(int anzahl)
{
  if (anzahl != (4 * sizeof(unsigned int))) {
    while (Wire.available()) {
      Wire.read();
    } return;
  }
  if (Wire.available()) {
    Wire.readBytes( (byte *) &Cycle_motoren[0], 4 * sizeof(unsigned int));
  }
}
/*-------------------------------FUNKTION FUER MOTORGESCHWINDIGKEIT-------------------------------*/
void pegel_wechseln (unsigned long *M_LastMicros, volatile unsigned int *Cycle , const int *
    Motor_Puls, bool *State) { // Pegel fuer Ansteuerung der Motoren aendern
  currentMicros = micros();
  for (int i = 0; i < 4; i++) {
    if ((Cycle[i] != 22)) {
      if (currentMicros - M_LastMicros[i] >= Cycle[i]) {
        M_LastMicros[i] = currentMicros;
        // Abfrage, ob aktueller Zustand = LOW
        if (State[i] == 0) {
          PORTD |= ((B11110000) & (1 << Motor_Puls[i]));       // PIN auf HIGH Pegel setzen
        } else PORTD &= ~((B11110000) & (1 << Motor_Puls[i]));// PIN auf LOW Pegel setzen
        State[i] = !State[i];                                  // State Variable toggeln
      }
    } else PORTD &= ~((B11110000) & (1 << Motor_Puls[i]));     // PIN auf LOW Pegel setzen
  }
  }
/*-------------------------------------------------------------------------------------------*/
/*-------------------------------------------PROGRAMMENDE-------------------------------------------*/
/*-------------------------------------------------------------------------------------------*/
```

A.1.3 Programm für die Signalüberwachung auf dem Arduino Mega2560 Entwicklungsboard

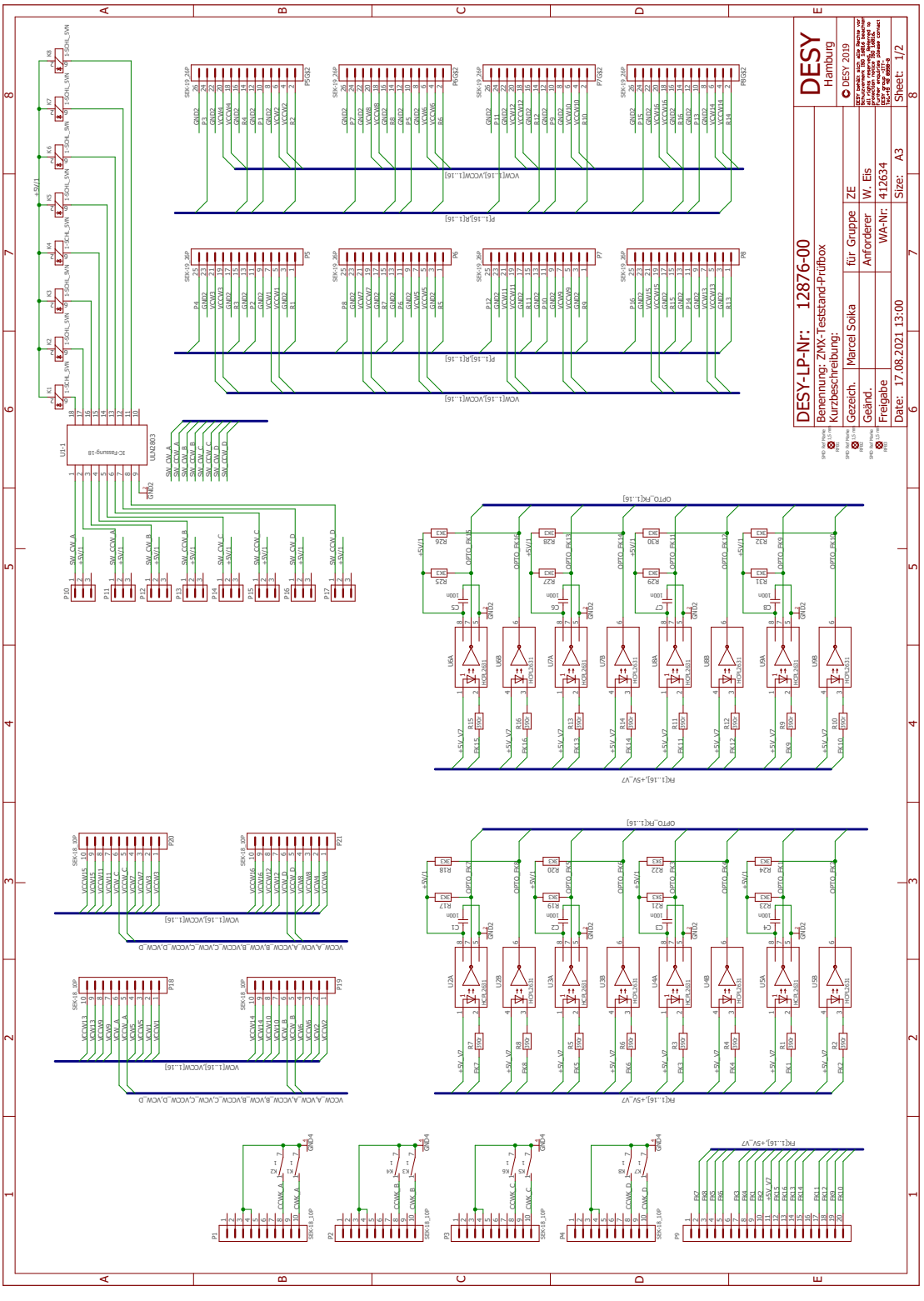
Listing A.3: Programm für die Signalüberwachung

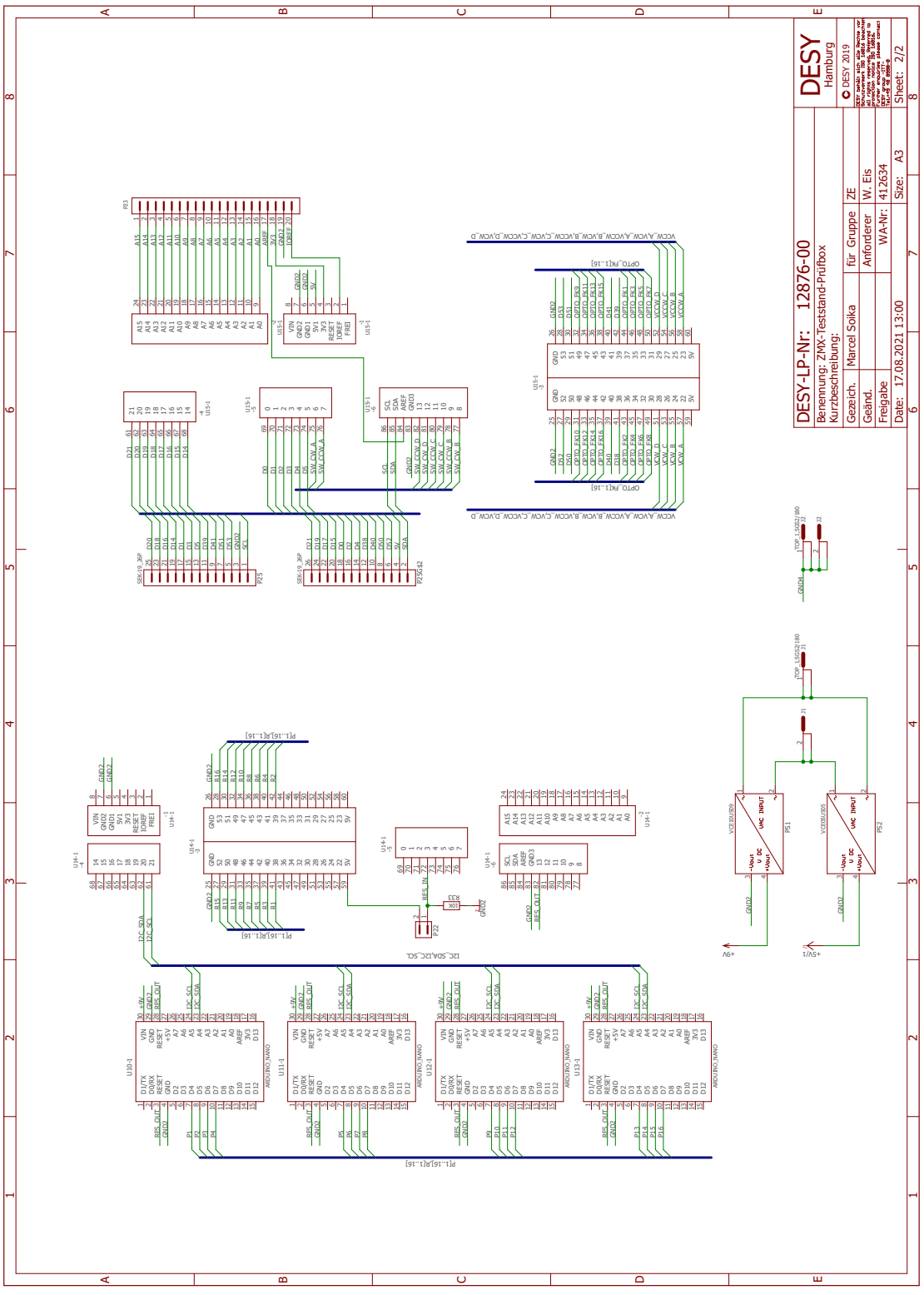
```
#include <string.h>
/*-------------------------------VARIABLEN UND KONSTANTEN-------------------------------*/
unsigned long Endstufen1bis8 = 0x00;
unsigned long Endstufen9bis16 = 0x00;
unsigned long Limitsignale = 0x00;
unsigned long Limit_Eingang_alt = 0x00;
unsigned long Limit_Eingang_neu = 0x00;
String Start = "";
char charToString[2];
/*-----------------------------------------PROTOTYPEN-----------------------------------------*/
void Fehler_Endstufe(char *Fehlermeldung);
void Fehler_Limitsignal(char *Fehlermeldung);
/*-------------------------------------------SETUP-------------------------------------------*/
void setup() {
  Serial.begin(9600);                          // Baudrate: 9600bps
  // Port A als Eingang
  DDRA &= ~(B11111111); // Limitsignale VCW + VCCW, Reihe A - D (D22(VCW_A) ,.., D29(VCCW_D))
  // Port C als Eingang
  DDRC &= ~(B11111111); // Fehlersignale der Endstufen: FK1 bis FK8 (D37 bis D30)
  // Port L als Eingang
  DDRL &= ~(B11111111); // Fehlersignale der Endstufen: FK9 bis FK16 (D49 bis D42)
  // PB7 - PB4 als Eingang
  DDRB &= ~(B11110000); // Limitsignale, Reihe C + D (PB4=D10(CW_C) ,.., PB7=D13(CCW_D))
  // PH6 - PH3 als Eingang
  DDRH &= ~(B01111000); // Limitsignale, Reihe A + B (PH3=D6 (CW_A) ,.., PH6=D9 (CCW_B))
  charToString[1] = '\0';
  Limit_Eingang_neu = (((PINB & B11110000) | ((PINH & B01111000) >> 3)) ^ (0xFF));
  Limit_Eingang_alt = Limit_Eingang_neu;
  Limitsignale = PINA;                         // Port A einlesen und in Variable abspeichern
}
/*-----------------------------------------LOOP() SCHLEIFE-----------------------------------------*/
void loop() {
  if (Serial.available() > 0) {                // warten auf Start-Befehl
    Start = Serial.readString();
  }
  if (Start == "START") {
    char Fehlermeldung[396] = {};
    // Limitschalter einlesen
    Limit_Eingang_neu = (((PINB & B11110000) | ((PINH & B01111000) >> 3)) ^ (0xFF));
    if (Limit_Eingang_alt != Limit_Eingang_neu) {
      // warten um Anschlussmodulen im ZMX-Ueberrahmen Zeit zum Verarbeiten neuer Werte zu
     geben
      for (unsigned long warte = 0; warte < 50000; warte++) {
        Limitsignale = PINA;                   // Port A einlesen und in Variable abspeichern
      }
      Limit_Eingang_alt = Limit_Eingang_neu;
    }
    else {
      Limitsignale = PINA;                     // Port A einlesen und in Variable abspeichern
    }
    Endstufen1bis8 = PINC;                     // Port C einlesen und in Variable abspeichern
    Endstufen9bis16 = PINL;                    // Port L einlesen und in Variable abspeichern
    if ((Endstufen1bis8 != 0x00) || (Endstufen9bis16 != 0x00) || (Limitsignale !=
     Limit_Eingang_neu)) {
      // Fehlermeldung von Endstufen
      if ((Endstufen1bis8 != 0x00) || (Endstufen9bis16 != 0x00)) {
        Fehler_Endstufe(Fehlermeldung);
      }
      // Fehlermeldung von Limitsignalen
      if (Limitsignale != Limit_Eingang_neu) {
        Fehler_Limitsignal(Fehlermeldung);
      }
      Serial.print(Fehlermeldung);             // Fehlermeldungen seriell senden
      Start = "";                              // Startbedingung zuruecksetzen
    }
  }
}
/*-------------------------------Fehler bei Endstufen-------------------------------*/
void Fehler_Endstufe(char *Fehlermeldung) {
  strcat(Fehlermeldung, "FEHLER: Fehlermeldung Endstufen: ");
  int fehler_anzahl = 0;
  char Endstufe_Nr = '1';
  int zehner = 0;
  // pruefen, bei welchen Endstufen ein Fehler gemeldet wurde
  for (int n = 0; n < 16; n++) {
```

```
74      charToString[0] = Endstufe_Nr;
75      if (n < 8) {
76        if (Endstufen1bis8 & (1 << n)) {
77          fehler_anzahl++;
78          if (fehler_anzahl > 1) {
79            strcat(Fehlermeldung, ", ");
80          }
81          strcat(Fehlermeldung, "Endstufe ");
82          strcat(Fehlermeldung, charToString);
83        }
84      }
85      else {
86        if (Endstufen9bis16 & (1 << (n - 8))) {
87          fehler_anzahl++;
88          if (fehler_anzahl > 1) {
89            strcat(Fehlermeldung, ", ");
90          }
91          strcat(Fehlermeldung, "Endstufe ");
92          if (zehner == 1) {
93            strcat(Fehlermeldung, "1");
94          }
95          strcat(Fehlermeldung, charToString);
96        }
97      }
98      if (Endstufe_Nr < '9') {
99        Endstufe_Nr++;
100     } else {
101       zehner = 1;
102       Endstufe_Nr = '0';
103     }
104   }
105   strcat(Fehlermeldung, "\n");
106 }
107 /*--------------------------------Fehler bei Limitsignalen-------------------------------------*/
108 void Fehler_Limitsignal(char *Fehlermeldung) {
109   if (strcmp(Fehlermeldung, "") != 0) {
110     strcat(Fehlermeldung, "----------------------");
111   }
112   strcat(Fehlermeldung, "FEHLER: Fehlermeldung Limitsignale: ");
113   int fehler_anzahl = 0;
114   char reihe = 'A';     // Variable fuer die Reihe auf der Rueckseite des ZMX-Ueberrahmens
115   for (int cw = 0, ccw = 1; cw < 8; ccw = ccw + 2, cw = cw + 2, reihe++) {
116     charToString[0] = reihe;
117     // pruefen, ob und in welcher Reihe (A bis D) Fehler bei Limitsignal-CW
118     if ((Limitsignale ^ Limit_Eingang_neu) & (1 << cw)) {
119       fehler_anzahl++;
120       if (fehler_anzahl > 1) {
121         strcat(Fehlermeldung, ", ");
122       }
123       strcat(Fehlermeldung, "CW");
124       strcat(Fehlermeldung, "(Reihe ");
125       strcat(Fehlermeldung, charToString);
126       strcat(Fehlermeldung, ")");
127     }
128     // pruefen, ob und in welcher Reihe (A bis D) Fehler bei Limitsignal-CCW
129     if ((Limitsignale ^ Limit_Eingang_neu) & (1 << ccw)) {
130       fehler_anzahl++;
131       if (fehler_anzahl > 1) {
132         strcat(Fehlermeldung, ", ");
133       }
134       strcat(Fehlermeldung, "CCW");
135       strcat(Fehlermeldung, "(Reihe ");
136       strcat(Fehlermeldung, charToString);
137       strcat(Fehlermeldung, ")");
138     }
139   }
140   strcat(Fehlermeldung, "\n");
141 }
142 /*-----------------------------------------------------------------------------------------------*/
143 /*-------------------------------------PROGRAMMENDE----------------------------------------------*/
144 /*-----------------------------------------------------------------------------------------------*/
```

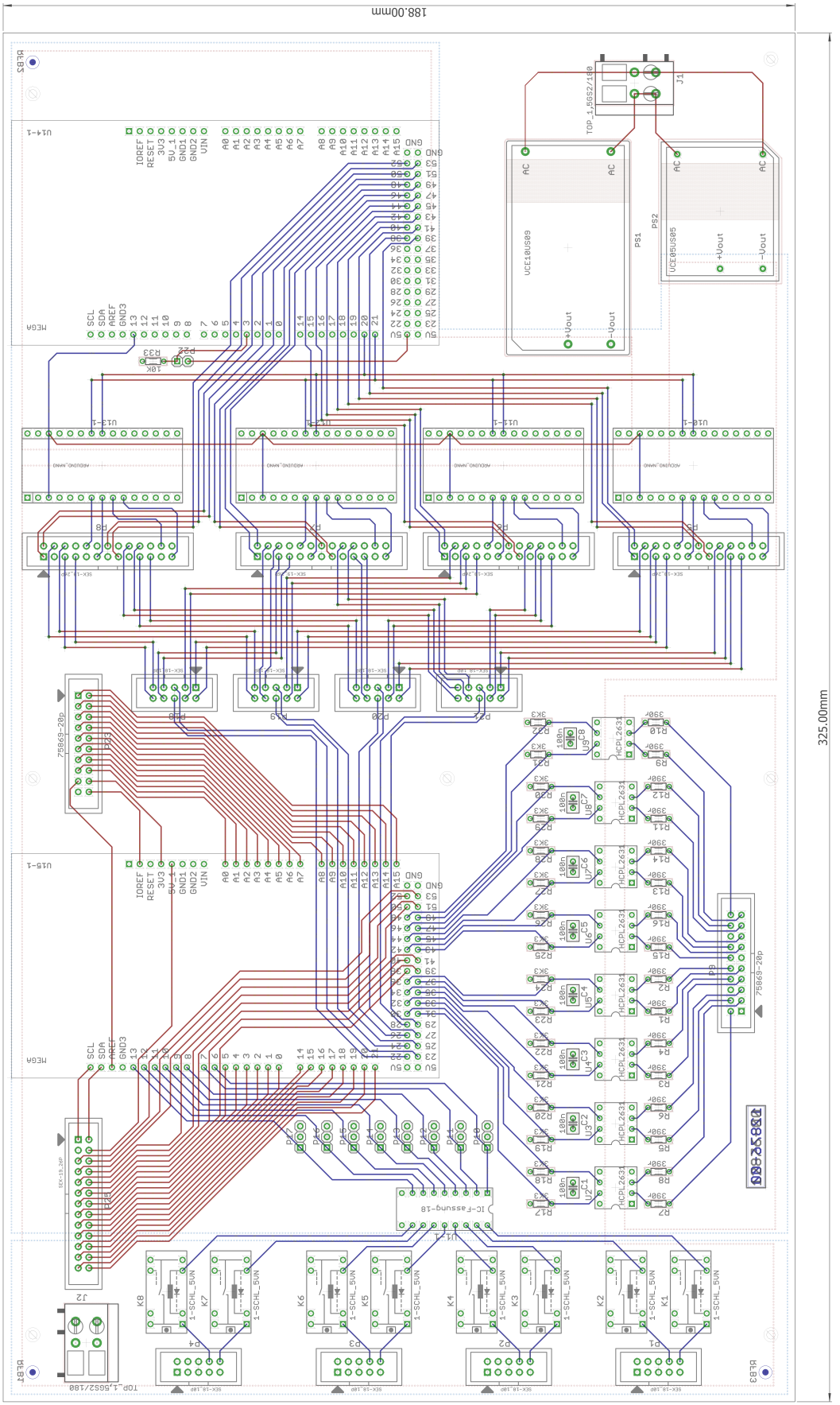
A.2 Leiterplatten für die Testbox

A.2.1 Schaltplan der Hauptleiterplatte der Testbox

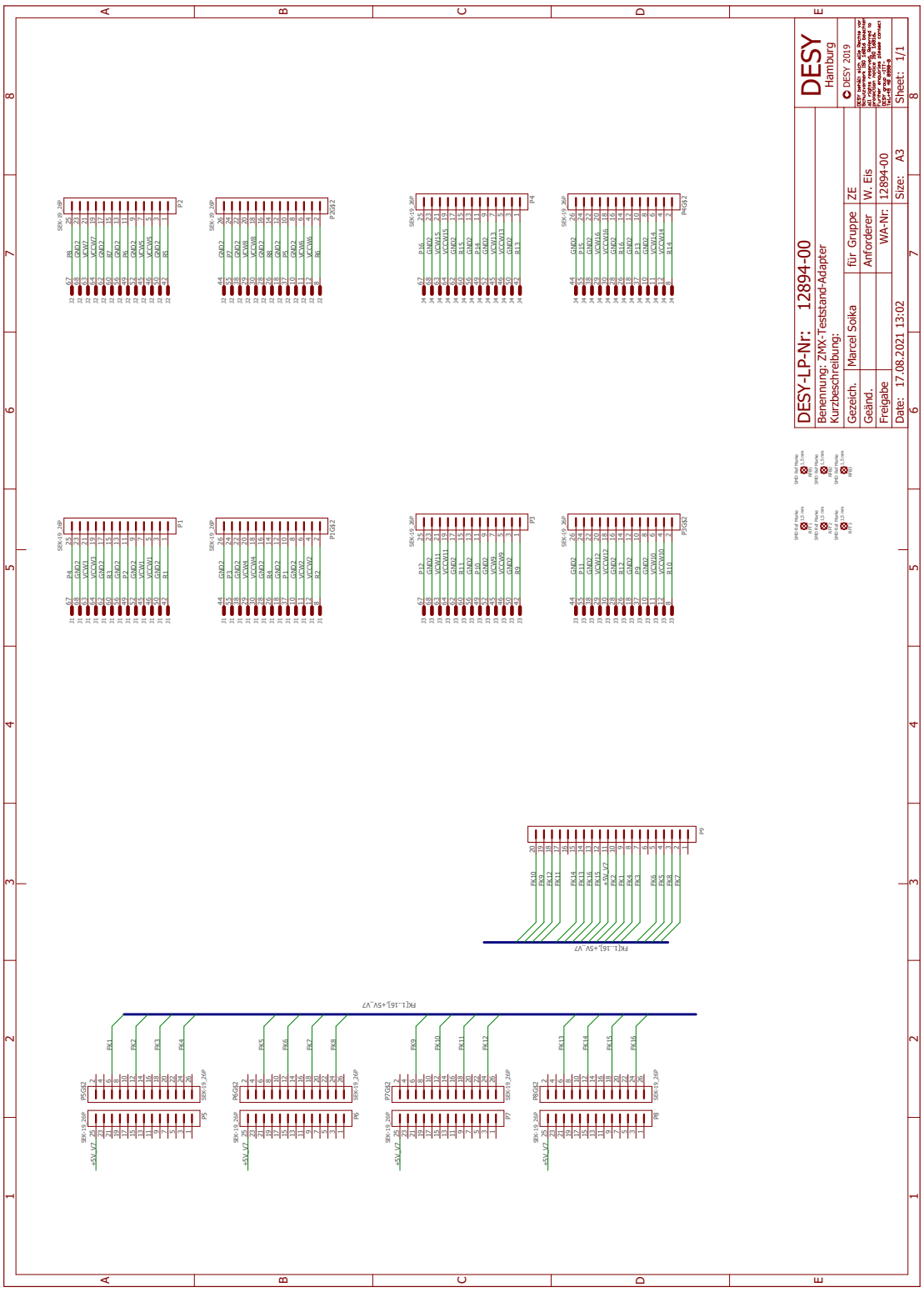



DESY
Hamburg
DESY-LP-Nr: 12876-00
Benennung: ZMX-Teststand-Prüfbox
Kurzbeschreibung:
Gezeich. Marcel Soika für Gruppe ZE
Geänd. Anforderer W. Eis
Freigabe WA-Nr: 412634
Date: 17.08.2021 13:00 Size: A3 Sheet: 2/2
© DESY 2019
17.08.2021 13:00 N:\public\Artikel_Baugruppen\12000-12999\12800-12899\12876\12876-00\Design_Daten\ZMX-Teststand-Prüfbox\12876-00.sch (Sheet: 2/2)

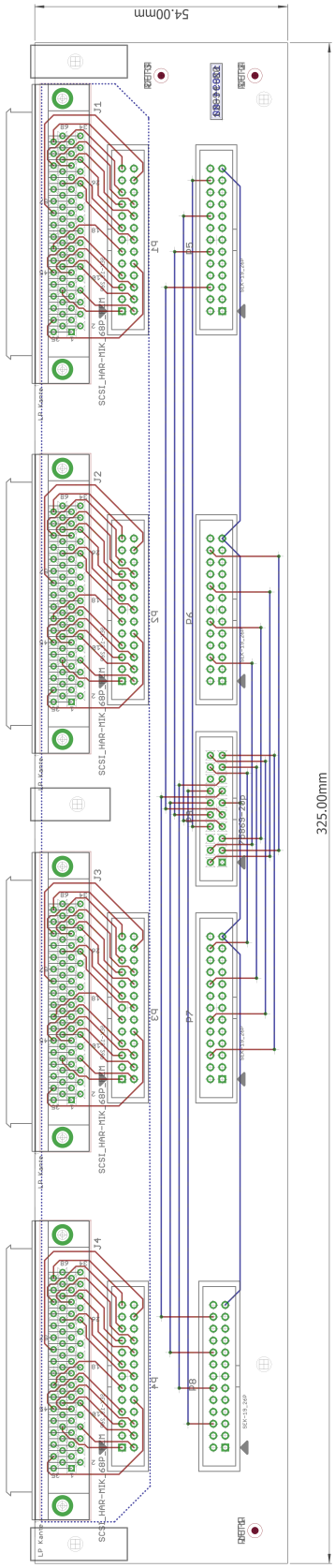
A.2.2 Layout der Hauptleiterplatte der Testbox



A.2.3 Schaltplan der Adapterleiterplatte der Testbox



A.2.4 Layout der Adapterleiterplatte der Testbox



Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Ort	Datum	Unterschrift im Original