Fachhochschule
Dortmund
University of Applied Sciences and Arts

Bachelorthesis

Entwurf eines VHDL-Designs und einer Applikationssoftware zur Konfiguration und Kalibrierung eines optischen Winkelgebers

im Studiengang Elektrotechnik

des Fachbereichs Elektrotechnik

Erstprüfer: Prof. Dr. Michael Karagounis

Zweitprüfer: Dipl.-Ing. Rolf Paulus

Abgabedatum: 11.11.2019

Vorgelegt von: Conrad Demske

Kurzzusammenfassung

Entwurf eines VHDL-Designs und einer Applikationssoftware zur Konfiguration und Kalibrierung eines optischen Winkelgebers

Diese Thesis handelt von der Konfiguration und Kalibrierung eines optischen Winkelgebers, welcher mit einer Entwicklungsplatine verbunden ist. Auf dieser Platine befinden sich Bauteile, die Signale des optischen Winkelgebers erhalten. Die digitalen Ausgangssignale der Bauteile auf der Entwicklungsplatine sind wiederum mit einem FPGA verbunden. Für die Konfiguration des FPGAs wird ein VHDL-Design zur Ansteuerung dieser Bauteile entworfen. Außerdem wird eine Software zur Nutzung des VHDL-Designs entworfen.

Abstract

VHDL design and application software for configuration and calibration of an optical angle encoder

This thesis is about the configuration and calibration of an optical angle encoder, which is connected to a development board. This board includes components that receive signals of the optical angle decoder. The digital output signals of the components on the development board are fed to an FPGA. For the configuration of the FPGA, a VHDL design will be designed to control these components. Furthermore, a software for the use of the VHDL design will be designed.

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass die von mir vorgelegte Prüfungsleistung selbstständig und ohne unzulässige fremde Hilfe erstellt worden ist. Alle verwendeten Quellen sind in der Arbeit so aufgeführt, dass Art und Umfang der Verwendung nachvollziehbar sind.

Ort, Datum	Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ADC	Analog Digital Converter
CentOS	Community Enterprise Operating System
DAC	Digital Analog Converter
FPGA	Field Programmable Gate Array
GUI	Graphical User Interface
LSB	Least Significant Bit
MSB	Most Significant Bit
POLDI	POLarisierende Photo-DIoden
USB	Universal Serial Bus
VHDL	Very High Speed Integrated Circuit Hardware Description Language

1 Einleitung

1.1 Einführung in POLDI

Im Projekt POLarisierende Photo-DIoden (POLDI) soll ein berührungsloser Winkelsensor entwickeln werden, welcher mithilfe von polarisierter Lichtstrahlung einen Winkel mit einer Genauigkeit von mindestens 0,1° bestimmt. Winkelsensoren finden in vielen verschiedenen Anwendungen Gebrauch wie in der Automobilindustrie, in der Medizin oder in der Robotik. Durch die Integration von Sensor und Elektronik auf einem gemeinsamen Chip in CMOS-Technologie wird erreicht, dass der POLDI Winkelsensor energieeffizient, miniaturisiert und kostengünstig ist. [5]

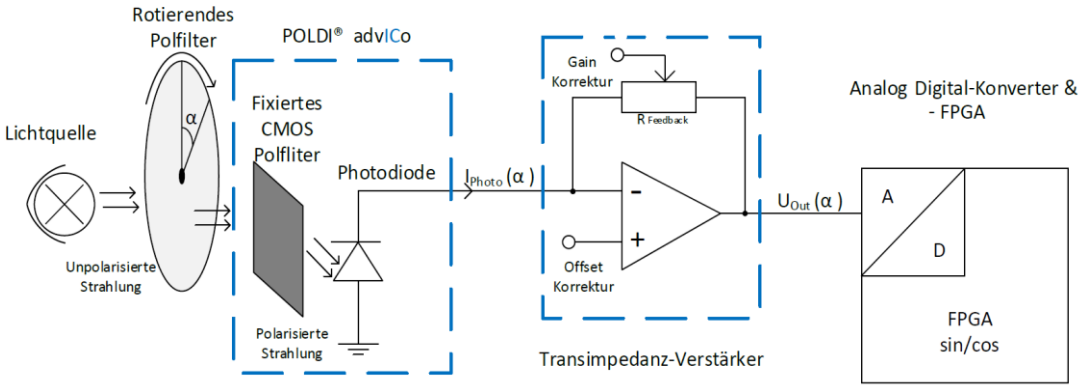

Abbildung 1: Konzept zur Winkelmessung des POLDI Sensors [5]

Abbildung 1 zeigt das Konzept zur Winkelmessung des POLDI Sensors. Durch parallele Leiterbahnen, welche sich auf der Oberfläche der Photodiode befinden, ist es möglich, integrierte Polarisationsfilter zu implementieren. Wird diese Kombination aus Photodiode und Polarisationsfilter mit linear polarisiertem Licht bestrahlt, so hängt die Intensität des auf die Dioden auftreffenden Lichtes vom Winkel der Lichtquelle ab. Der Photostrom der Dioden wird dann durch einen Transimpedanzverstärker in eine definierte Spannung gewandelt, welche an einem Analog Digital Converter (ADC) anliegt. [5]

Der POLDI Winkelsensor besteht aus vier solcher Photodioden, wobei die Polarisationsfilter der einzelnen Photodioden einen Winkel von jeweils 0°, 45°, 90° und 135° zueinander haben. Somit liefert jede Photodiode einen unterschiedlichen Photostrom. Jeder Photostrom fließt in einen separaten Transimpedanzverstärker. Die demnach unterschiedlich

definierten Spannungen liegen jeweils an separaten ADCs an und werden einem Field Programmable Gate Array (FPGA) zur Verfügung gestellt, welcher mit Algorithmen der digitalen Signalverarbeitung den Winkel des einfallenden Lichtes berechnen kann. [5]

1.2 Kommunikation zwischen Computer und POLDI

Die für die Auslesung nötigen Bauelemente wie ADCs und Digital Analog Converters (DACs) zur Einstellung der Referenzspannung der ADCs, die Gain- und Offsetkorrektur sowie weitere Elemente befinden sich auf einer Platine, welche Evalboard genannt wird. Auf dem sogenannten Mojoboard befinden sich wiederum der FPGA und ein Mikrocontroller, welcher den Datenaustausch zwischen dem FPGA und dem Computer verwaltet. Die Kommunikation des Mikrocontrollers mit dem Computer und mit dem FPGA erfolgt seriell. Für den Datenaustausch zwischen dem Computer und dem Mikrocontroller wird der Universal Serial Bus (USB)-Port des Computers mit der Micro-USB-Schnittstelle des Mikrocontrollers verbunden. Auf dem Computer wird unter der Linux-Distribution Community Enterprise Operating System (CentOS) die Entwicklungsumgebung QT-Creator verwendet, welche die Erstellung von C++-Programmen mit Graphical User Interfaces (GUIs) ermöglicht. Abbildung 2 zeigt schematisch den in diesem Kapitel beschriebenen Aufbau des Systems. [4]

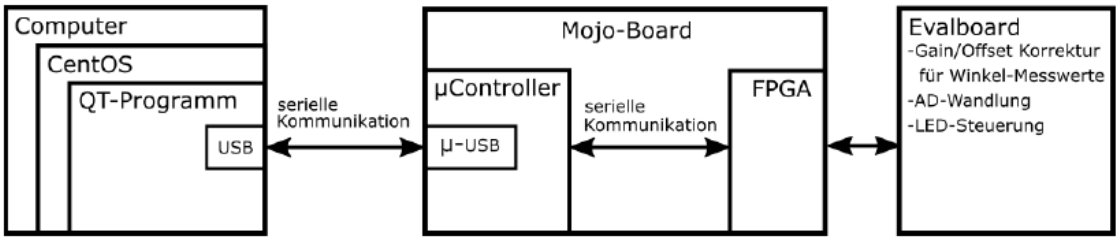

Abbildung 2: Kommunikation zwischen Computer und POLDI [4]

1.3 Ziel der Bachelorarbeit

Im Rahmen dieser Arbeit soll für das Projekt POLDI eine Ansteuerung aller sich auf dem Evalboard befindenden DACs und ADCs entworfen werden. Es befinden sich fünf DACs vom Typ „AD5544“, ein DAC vom Typ „AD5686“ und acht ADCs vom Typ „7980“ auf dem Evalboard. Die Ansteuerung besteht aus einer QT-Anwendung und einem Very High Speed Integrated Circuit Hardware Desription Language (VHDL)-Design, welches auf dem FPGA des Mojoboards implementiert wird.

Die zu erstellende QT-Anwendung soll dem Anwender ermöglichen, über ein GUI die verschiedenen Kanäle der DACs einzustellen. Außerdem sollen verschiedene ADCs ausgewählt werden können, deren anliegende Spannungen bzw. die entsprechenden digitalen Ausgangssignale im GUI angezeigt werden. Die QT-Anwendung muss dafür die in einem Kommunikationsprotokoll des VHDL-Designs festgelegten Daten zur Synchronisation sowie dort festgelegte Daten zur Kennung des ausgewählten DACs bzw. der ausgewählten ADCs an den FPGA senden. Soll ein DAC angesteuert werden, muss zusätzlich die vom Anwender ausgewählte Einstellung des DACs in einen Binärcode umgewandelt und gesendet werden. Außerdem muss die QT-Anwendung die vom FPGA ausgelesenen Daten der ADCs empfangen, diese in Spannungen, die an den jeweiligen ADCs anliegen, umrechnen und im GUI anzeigen.

Das VHDL-Design soll das o.g. Kommunikationsprotokoll enthalten, welches mit den empfangenen Daten für eine Synchronisation sorgt und anhand der Daten die unterschiedlichen DACs und ADCs auswählt. Es sollen alle DACs angesteuert und alle ADCs ausgelesen werden können. Zur Auslesung der ADCs soll es zwei verschiedene Modi geben. In einem Modus soll es möglich sein, unterschiedlich viele ADCs auszulesen, die ausgelesenen Daten abzuspeichern und diese an den PC zu senden. Da die Winkelberechnung letztendlich auf dem FPGA stattfinden wird, soll es einen Modus geben, in dem alle ADCs kontinuierlich ausgelesen werden, ohne dass die ausgelesenen Daten an den PC gesendet werden.

2 Software

2.1 HDL Designer

„HDL Designer“ ist eine Software zur Erstellung von HDL-Designs. Dabei können Module grafisch erzeugt und miteinander verbunden werden. Dadurch ist ein erstelltes Design deutlich übersichtlicher, als wenn es nur in Textform erstellt wird. Die Verschaltung von Modulen miteinander kann in dieser Software grafisch erfolgen. Die Software erstellt eine „vhd“-Datei, die die grafisch erzeugten Verbindungen der Module miteinander enthält.

Zudem lassen sich in dieser Software Zustandsmaschinen grafisch entwerfen. „HDL Designer“ erzeugt aus den grafischen Zustandsmaschinen HDL-Code. Dabei kann ausgewählt werden, in welchen Zustand die Zustandsmaschine bei einem Reset übergehen soll, ob Zustände mit steigenden oder fallenden Taktflanken erreicht werden und wie viel Prozesse die Zustandsmaschine haben soll. Außerdem gibt es hierarchische Zustände, die wiederum Zustände enthalten, was die Übersicht verbessert.

In dieser Bachelorarbeit wurde die Software genutzt, um Komponenten miteinander zu verschalten und um Zustandsmaschinen grafisch zu erstellen. Alle Zustandsmaschinen sind 3-Prozess-Zustandsmaschinen und erreichen andere Zustände mit steigenden Taktflanken. Die Zustandsmaschinen wurden als Moore-Automaten entworfen. Die Ausgabe der Automaten hängt also nur von den Zuständen ab.

2.2 Xilinx ISE 14.7 Design Suite

Die Entwicklungsumgebung „Xilinx ISE“ ermöglicht mithilfe von Hardwarebeschreibungssprachen wie VHDL oder Verilog, digitale Schaltungen zu entwerfen und auf Xilinx FPGAs zu implementieren. Außerdem ist es möglich, entworfene Designs zu simulieren und die Schaltbilder der Designs anzeigen zu lassen. Im Rahmen der Bachelorarbeit wurde „Xilinx ISE“ genutzt, um Simulationen durchzuführen und eine „bin“-Datei des VHDL-Designs für die FPGA Konfiguration zu generieren.

Nachdem ein Projekt geöffnet wurde, können unten links die Schaltflächen „Start“, „Design“, „Files“ und „Libraries“ angeklickt werden. Unter „Files“ lassen sich die Quelltexte

der einzelnen Dateien öffnen und bearbeiten. Unter Designs lässt sich oben zwischen „Implementation“ und „Simulation“ wechseln. Wenn „Implementation“ ausgewählt wurde, kann das ausgewählte Modul mit einem Doppelklick auf „Synthesize - XST“ synthetisiert und mit einem Doppellklick auf „Implement Design“ implementiert werden. Um das Schaltbild eines Moduls anzeigen zu lassen, muss erst eine Synthese stattfinden. Das Schaltbild kann eingesehen werden, indem auf das kleine „+“-Zeichen neben „Synthesize - XST“ geklickt wird und „View RTL Schematic“ mit einem Doppelklick ausgewählt wird. Wenn zu „Simulation“ gewechselt wird, kann dort oben links „Behavioral“ oder „Post-Route“ ausgewählt werden und eine Behavioral oder Timing Simulation gestartet werden.

2.3 QT-Creator

Die Entwicklungsumgebung QT-Creator ermöglicht die Erstellung von C++-Programmen mit GUIs. In dieser Arbeit wird die Entwicklungsumgebung genutzt, um eine Anwendung zu entwerfen, welche die grafische Eingabe von Daten und die Versendung dieser Daten an den FPGA erlaubt.

Zur Erstellung eines neuen Projektes wird nach dem Start des QT-Creators die Option „neues Projekt“ angeklickt. Es öffnet sich ein Fenster, in welchem ausgewählt werden muss, was für ein Projekt erstellt werden soll. Da in dieser Bachelorarbeit eine Anwendung mit einem GUI erzeugt werden soll, wird unter „Projekte“ -> „Anwendung“ -> „Qt-Widgets-Anwendung“ ausgewählt. Durch einen Klick auf „Auswählen“ erscheint eine neue Maske, in welcher ein Projektname und Speicherort festgelegt wird. Nachdem auf „Weiter“ geklickt wird, erscheint die Kit-Auswahl, in welcher schon das richtige Kit ausgewählt ist. Es wird wieder „Weiter“ angeklickt. Die nächsten Masken „Details“ und „Zusammenfassung“ können ohne Änderungen mit einem Klick auf „Weiter“ bzw. „abschließen“ übersprungen werden.

Oben links ist nun ein Ordnersymbol mit dem entsprechenden Namen des Projektes zu sehen. Dieser Ordner kann mit einem Klick auf den kleinen Pfeil ausgeklappt werden, sodass die „pro“-Datei sowie die Unterordner „Header-Dateien“, „Quelldateien“ und „Formulardateien“ erscheinen, welche weiter ausgeklappt werden können. Unter den einzelnen Ordnern erscheinen die zugehörigen Dateien, welche sich mit einem Doppelklick öffnen und bearbeiten lassen. Das Hinzufügen einer neuen Header-, Quell- oder

Formulardatei geschieht mit einem Rechtsklick auf ein Ordnersymbol oben links und das Anklicken von „Hinzufügen“. Es öffnet sich ein neues Fenster, in welchem sich verschiedene Dateien und Klassen auswählen lassen.

Mit einem Doppelklick auf die Datei unter dem Ordner „Formulardateien“ erscheint das GUI, welches sich mit vorgegebenen Objekten gestalten lässt. Um zurück zu den Ordnersymbolen zu gelangen, muss am linken grauen Rand „Editieren“ angeklickt werden.

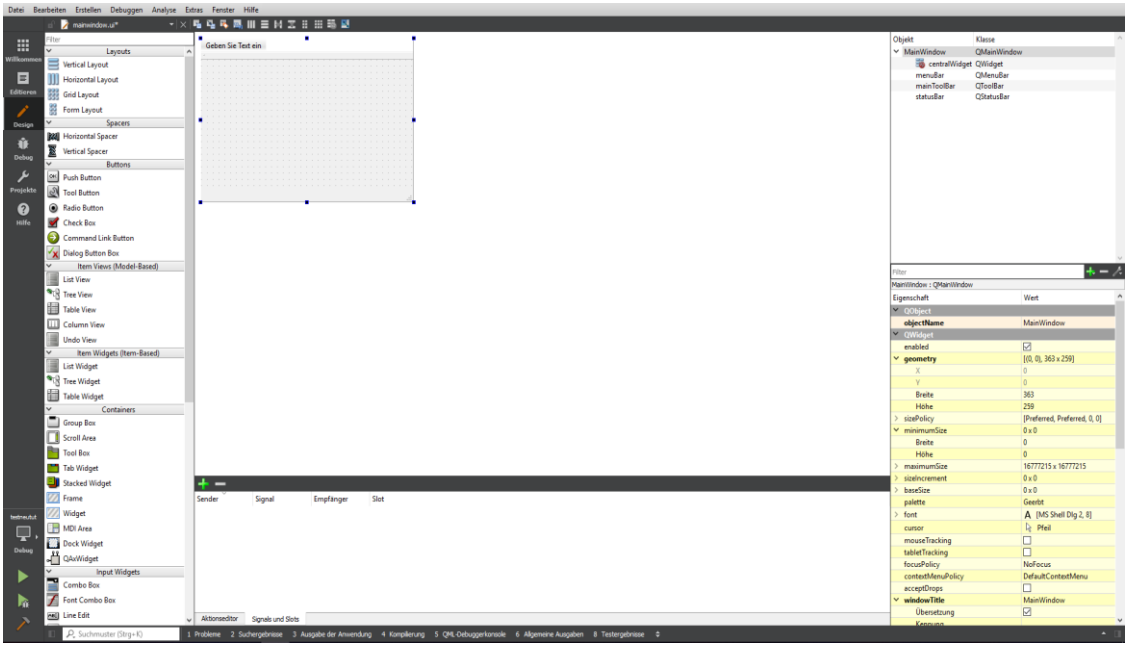

Abbildung 3: leeres GUI im QT-Creator

Auf der linken Seite im weißen Feld in Abbildung 3 gibt es verschiedene Objekte, die per Drag & Drop auf dem GUI, welches sich mittig befindet, platziert werden können. Beispielsweise sind verschiedene Buttons, Eingabefelder und Anzeigeobjekte verfügbar. Oben rechts befindet sich eine Übersicht mit allen Objekten. Das GUI und alle hinzugefügten Elemente lassen sich in ihren Eigenschaften verändern. Möglich ist dies auf der rechten Seite unter der Übersicht der Objekte unter „Eigenschaften“. Um die Eigenschaften eines Objektes zu verändern, muss dieses entweder im GUI oder in der Übersicht der Objekte angeklickt werden. Im GUI direkt lassen sich z. B. auch die Größe, der Name oder die Position verändern.

Per Rechtsklick auf ein Objekt -> „Slots anzeigen“ lassen sich den Objekten verschiedene Aktionen hinzufügen, welche Signale auslösen. Nach dem Hinzufügen einer Aktion öffnet sich die Quelldatei mit dem Projektnamen und der Endung „cpp“, in welcher automatisch ein Quelltext mit einer der Aktion entsprechenden Funktion generiert wurde. In

diesen Rahmen kann dann der Programmcode eingefügt werden, der ausgeführt wird, wenn die Aktion im GUI ausgelöst wird. Die folgenden zwei Abbildungen zeigen dies exemplarisch für einen „Radio Button“ und die Aktion „clicked()“.

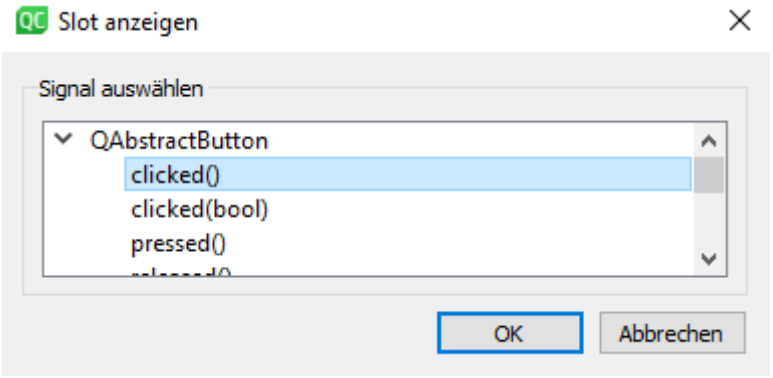

Abbildung 4: Slots anzeigen im QT Creator

```
void MainWindow::on_radioButton_clicked()
{

}
```

Abbildung 5: Funktion einer hinzugefügten Aktion im QT Creator

Um ein aktuelles Projekt auszuführen, muss am linken grauen Rand der Benutzeroberfläche unten auf den oberen grünen Pfeil geklickt werden. Um eine Anwendung zu erstellen, muss auf das Hammersymbol zwei Symbole unter dem grünen Pfeil geklickt werden.

2.4 Mojo Loader

Um mit dem implementierten Design den FPGA zu konfigurieren, wird der Mojo Loader benötigt. Die mit ISE 14.7 erzeugte „bin“-Datei des VHDL-Designs wird mit dem Mojo Loader auf das Mojoboard übertragen. Nachdem das Programm gestartet wurde, muss eine serielle Schnittstelle ausgewählt werden. Dafür muss auf das Feld neben „Serial Port:“ geklickt werden. Durch einmalige Selektion per Mausklick wird die richtige serielle Schnittstelle gefunden, wenn das Mojoboard mit dem Computer verbunden ist. Soll die „bin“-Datei nicht im Flash-Speicher des Boards gespeichert werden, muss der Haken bei „Store to Flash“ entfernt werden. Mit einem Klick auf „Open Bin File“ kann die gewünschte „bin“-Datei ausgewählt werden. Die Übertragung beginnt mit einem Klick auf „Load“. Abbildung 6 zeigt die Benutzeroberfläche des Mojo Loaders mit einer ausgewählten seriellen Schnittstelle und ausgewählten „bin“-Datei.

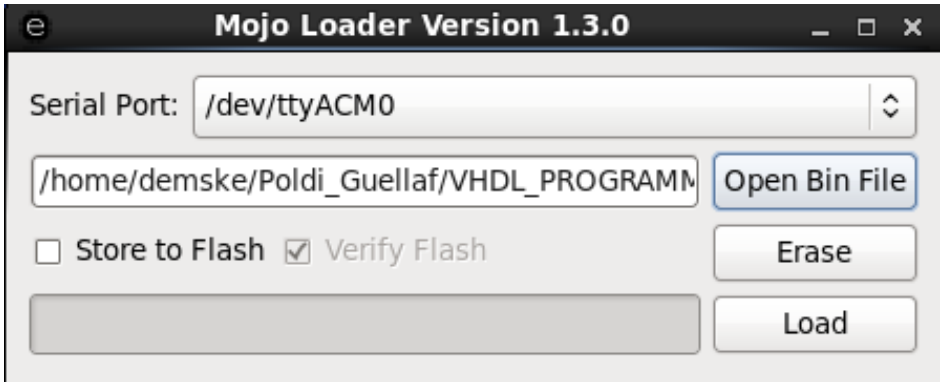

Abbildung 6: Benutzeroberfläche des Mojo Loaders

3 DACs und ADCs

3.1 DAC „AD5686“

3.1.1 Funktionsweise

Der DAC „AD5686“ besitzt vier verschiedene Kanäle und funktioniert mit einer Versorgungsspannung von 2,7 V bis 5,5 V. Er gibt eine gepufferte Spannung mit einer 16-Bit Auflösung aus. [2]

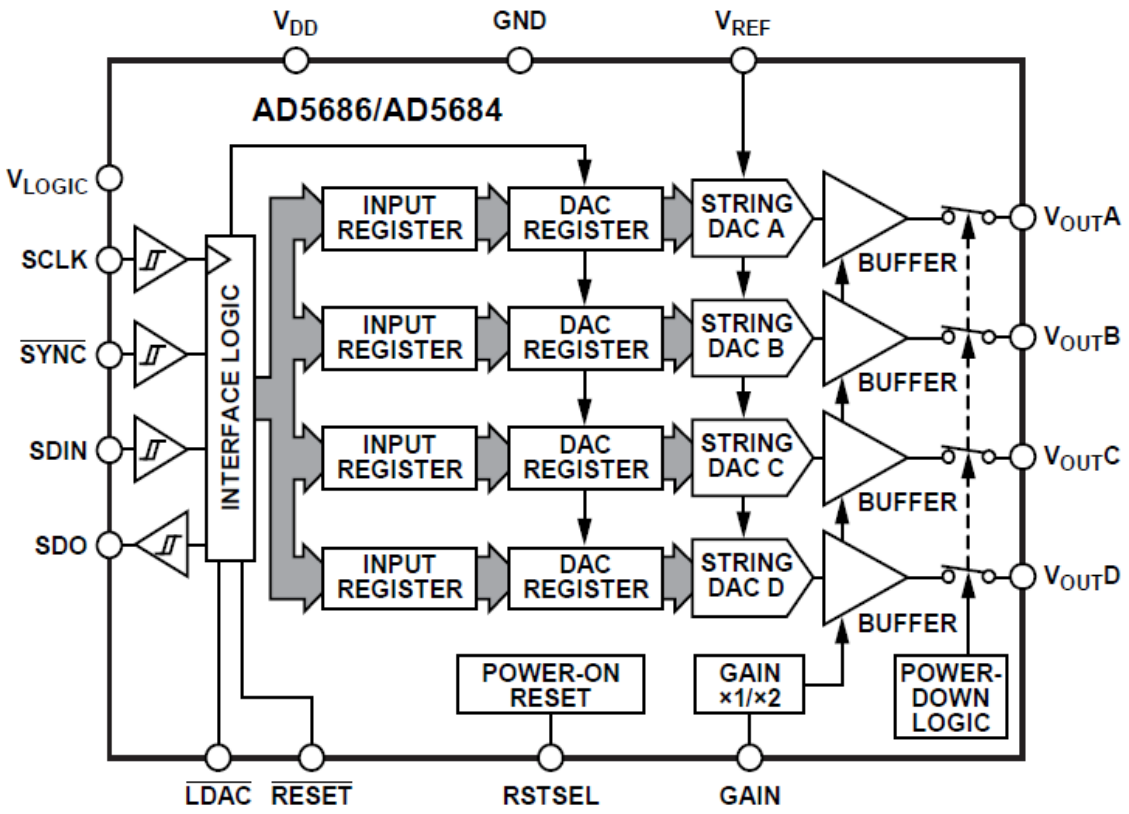

Abbildung 7: funktionelles Blockdiagramm des DACs „AD5686“ [2]

In Abbildung 7 sind die vier verschiedenen Kanäle des DACs zu erkennen. Ist der „GAIN“-Pin am „GND“-Pin angeschlossen, entspricht die maximale Ausgangsspannung aller DAC-Kanäle der Versorgungsspannung. Wenn der „GAIN“-Pin mit dem „V_{LOGIC}“-Pin verbunden ist, entspricht die maximale Ausgangsspannung der doppelten Versorgungsspannung. Der „RSTSEL“-Pin ermöglicht, die Ausgangsspannung der DAC-Kanäle direkt nach dem Einschalten des DACs auf 0 V oder auf die halbe Versorgungsspannung einzustellen. Soll die Ausgangsspannung 0 V betragen, muss der „RSTSEL“-Pin mit dem „GND“-Pin verbunden werden, für die halbe Versorgungsspannung

muss der „RSTSEL“-Pin mit dem „V_{LOGIC}“-Pin verbunden werden. In diesem Projekt ist der „GAIN“-Pin mit dem „GND“-Pin und der „RSTSEL“-Pin mit dem „V_{LOGIC}“-Pin verbunden. [2]

Daten werden über eine serielle 3-Draht-Schnittstelle, bestehend aus „SDIN“, „SCLK“ und „$\overline{\text{SYNC}}$“, in einem 24-Bit-Wortformat in den DAC geschrieben. Die serielle Schnittstelle arbeitet mit Taktraten von bis zu 50 MHz. Die Daten werden in die für die einzelnen Kanäle zuständigen Eingangsschieberegister geschrieben, von welchen sie in DAC-Register weitergeleitet werden. Von dort aus werden sie in Strings (Widerstandsketten) geschrieben. Die geschriebenen Daten bestimmen darüber, von welchem Knoten des Strings die Spannung abgegriffen wird, sodass unterschiedliche Spannungen generiert werden können. Die abgegriffene Spannung wird einem Verstärker zugeführt. Das Ausgangssignal des Verstärkers liegt dann am Ausgang des DACs an. [2]

3.1.2 Funktionsbeschreibung der für die FPGA-Ansteuerung relevanten Pins

SCLK ist der serielle Eingang für den Takt. Daten werden mit fallenden Flanken des Taktes übernommen. Daten können mit einer Frequenz von bis zu 50 MHz übertragen werden. [2]

SDIN ist der Eingang des 24-Bit-Eingangsschieberegisters und bietet die Möglichkeit, seriell Daten mit fallenden Flanken von „SLCK“ in das Empfangsschieberegister zu schreiben. [2]

$\overline{\text{SYNC}}$ ist ein Synchronisationssignal für Eingangsdaten. Bei einem Low-Pegel des Signals „$\overline{\text{SYNC}}$“ werden Daten auf dem „SDIN“ Bus mit den nächsten 24 fallenden Flanken von „SLCK“ empfangen. [2]

SDO kann genutzt werden, um mehrere AD5686- oder AD5684-DACs in Reihe zu schalten oder um die eingelesenen Daten seriell zurück zu lesen, wobei sie mit der steigenden Flanke von „SCLK“ übertragen werden und bei der fallenden Flanke des Taktes gültig sind. [2]

$\overline{\text{LDAC}}$ kann synchron oder asynchron verwendet werden. Bei einem anliegenden Low-Pegel aktualisieren sich die DAC-Register, deren Eingangsregister neue Daten erhalten haben. „$\overline{\text{LDAC}}$“ kann auch dauerhaft low gehalten werden. [2]

$\overline{\text{RESET}}$ reagiert auf Pegel des am „$\overline{\text{RESET}}$“-Pin anliegenden Signals und ist asynchron zum Takt. Wenn ein Low-Pegel erkannt wird, wird das Eingangs- und das DAC-Register auf null oder auf den halben Wert gehalten, je nach Verbindung des „GAIN“-Pins (siehe Kapitel 3.1.1). Wenn „$\overline{\text{RESET}}$“ low gehalten wird, werden alle „$\overline{\text{LDAC}}$“ -Impulse ignoriert. [2]

3.1.3 Inhalt des 24-Bit-Eingangsschieberegisters

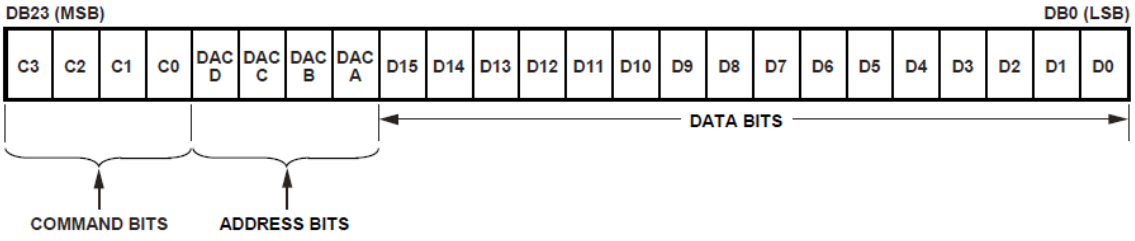

Abbildung 8: Eingangsschieberegister des DACs „AD5686“ [2]

Das Eingangsschieberegister, in welches seriell Daten geschrieben werden, füllt seine 24 Bits (DB23 bis D0) beginnend vom Most Significant Bit (MSB) in absteigender Reihenfolge bis hin zum Least Significant Bit (LSB). Wenn der Pin $\overline{\text{SYNC}}$ low wird, werden mit den nächsten 24 fallenden Flanken des Takteingangs Daten in das Eingangsschieberegister geschrieben und mit der ansteigenden Flanke von $\overline{\text{SYNC}}$ aktualisiert. [2]

Die ersten vier Bits sind Befehlsbits (C3, C2, C1, C0) zur Ausführung unterschiedlicher Befehle, die nachfolgenden vier (DAC D, DAC C, DAC B, DAC A) bilden eine Adresse und ermöglichen die Ansteuerung eines oder mehrerer DAC-Kanäle. Die letzten 16 Bits sind Daten (D15 bis D0), welche die Ausgangsspannung bestimmen. [2]

Command Bits				
C3	C2	C1	C0	Description
0	0	0	0	No operation
0	0	0	1	Write to Input Register n (dependent on $\overline{\text{LDAC}}$)
0	0	1	0	Update DAC Register n with contents of Input Register n
0	0	1	1	Write to and update DAC Channel n
0	1	0	0	Power down/power up DAC
0	1	0	1	Hardware $\overline{\text{LDAC}}$ mask register
0	1	1	0	Software reset (power-on reset)
0	1	1	1	Reserved
1	0	0	0	Set up DCEN register (daisy-chain enable)
1	0	0	1	Set up readback register (readback enable)
1	0	1	0	Reserved
…	…	…	…	Reserved
1	1	1	1	No operation, daisy-chain mode

Abbildung 9: Definitonen der Befehlsbits des DACs „AD5686“ [2]

Abbildung 9 zeigt verschiedene Kombinationen der Befehlsbits und die dazugehörige Beschreibung. Im Rahmen der Bachelorarbeit sollen die Daten für einen oder mehrere DAC-Kanäle übertragen und direkt aktualisiert werden, weshalb die Kombination „0011“ genutzt wird.

Zur Ansteuerung eines einzelnen DAC-Kanals muss das dazugehörige Adressbit den Wert eins und die anderen Adressbits den Wert null annehmen. Es ist auch möglich, mehrere Kanäle anzusteuern. Dafür müssen die Adressbits der gewünschten Kanäle logisch eins und die anderen Adressbits bzw. das andere Adressbit logisch null sein. Wenn alle Kanäle angesteuert werden sollen, müssen alle Adressbits den Wert eins haben. [2]

Die folgende Übertragungsfunktion des DACs zeigt, wie die 16 Datenbits die Ausgangsspannung V_{OUT} beeinflussen. [2]

$$V_{OUT} = V_{REF} * Gain[\frac{D}{2^N}] \qquad [2]$$

V_{REF} steht dabei für die Versorgungs- bzw. Referenzspannung, Gain für den Verstärkungsfaktor, welcher in diesem Projekt den Wert eins besitzt. N ist die Auflösung des DACs, also in diesem Fall 16, und D das dezimale Äquivalent der 16 Datenbits. Eine Ausgangspannung von 0 V ergibt sich, wenn alle Datenbits den Wert null haben. Das dezimale Äquivalent nimmt dann auch den Wert 0 an und somit ergibt die ganze

Gleichung null. Haben alle Datenbits den Wert eins, ist das dezimale Äquivalent und somit auch die Ausgangsspannung am größten. Laut der Formel entspricht dieser Wert nahezu der Referenzspannung. [2]

3.2 DAC „AD5544“

3.2.1 Funktionsweise

Der DAC „AD5544“ besitzt vier verschiedene Kanäle und funktioniert mit einer Versorgungsspannung von 2,7 V bis 5,5 V. Er gibt Ströme mit einer 16-Bit Auflösung aus, wobei die einzelnen Ausgangsströme maximal 2,5 mA groß sein können. [1]

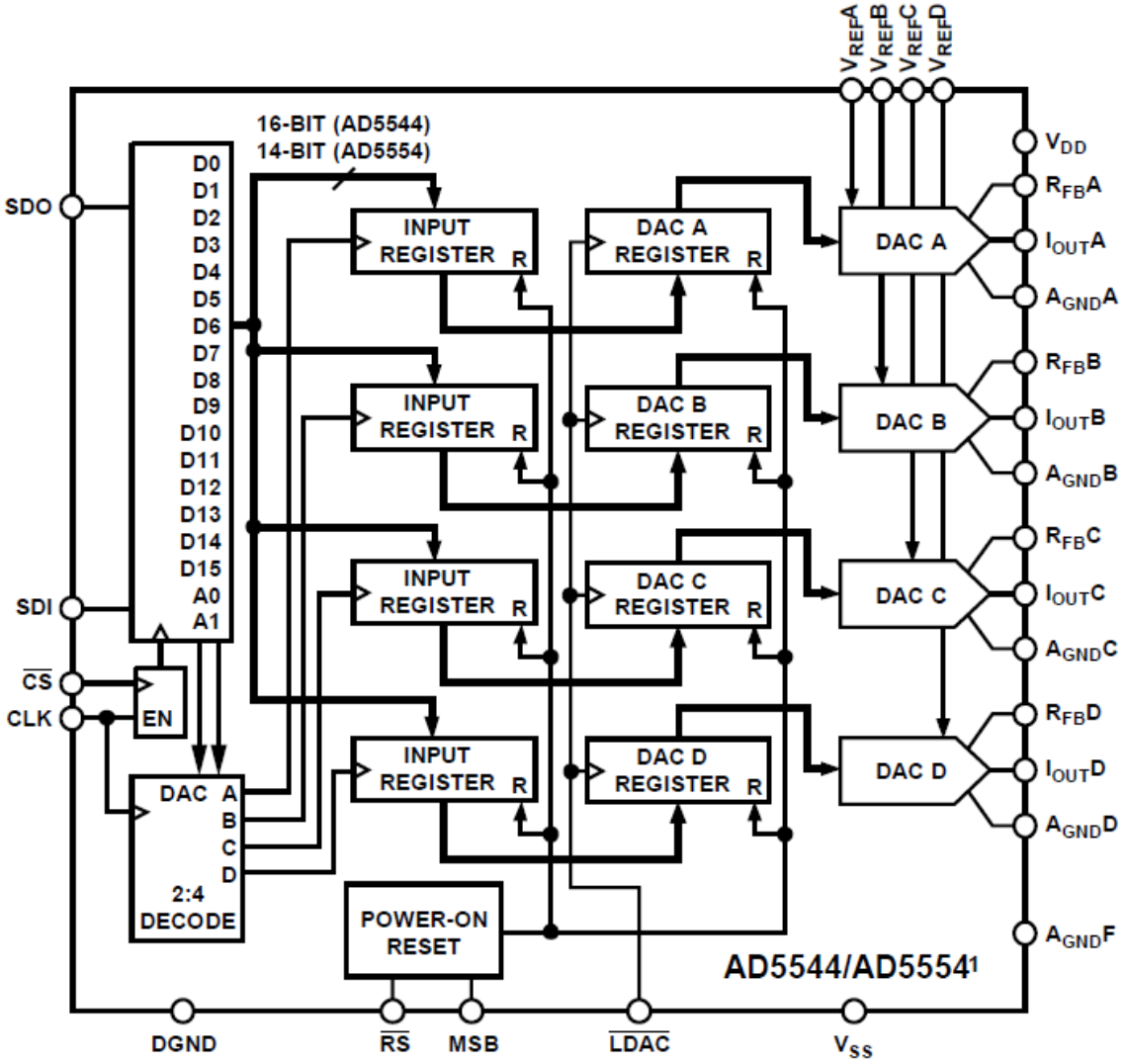

Abbildung 10: funktionelles Blockdiagramm des DACs „AD5544“ [1]

Abbildung 10 zeigt das funktionelle Blockdiagramm des DACs. Wie oben rechts zu erkennen ist, hat jeder Kanal einen eigenen Pin für die Referenzspannung. Die

Referenzspannung eines Kanals bestimmt den Bereich, den der Ausgangsstrom des Kanals annehmen kann. Der unten mittig zu sehende Block „Power-On Reset“ sorgt dafür, dass die Ausgangsströme nach dem Einschalten entweder 0 A oder der Hälfte des maximalen Ausgangsstromes entsprechen. Wenn am „MSB“-Pin ein Signal mit einem High-Pegel anliegt, nimmt der Wert den halben maximalen Ausgangsstrom an. Wenn dort ein Low-Signal anliegt, beträgt der Wert 0 A. In diesem Projekt liegt am „MSB“-Pin ein Low-Signal an. [1]

Daten werden über eine doppelt gepufferte serielle 3-Draht-Schnittstelle, bestehend aus „SDI“, „CLK“ und „$\overline{\text{CS}}$“, in das Eingangsschieberegister geschrieben. Vom Eingangsschieberegister werden die Daten in das Eingangsregister eines Kanals geschrieben, wobei ein 2-4-Dekodierer mithilfe der Adressbits erkennt, welches Eingangsregister der vier Kanäle ausgewählt wurde. Von den Eingangsregistern werden die Daten in DAC-Register weitergeleitet. Die empfangenen Daten bestimmen über die Verschaltung eines Widerstandsnetzwerkes und somit über die Größe des Ausgangsstromes. [1]

3.2.2 Funktionsbeschreibung der für die FPGA-Ansteuerung relevanten Pins

$\overline{\text{RS}}$ sorgt bei anliegendem Low-Signal dafür, dass alle Eingangs- und DAC-Register je nach anliegender Spannung am **MSB**-Pin auf null oder auf die Hälfte des Maximalwertes gesetzt werden. Die Register werden auf null gesetzt, wenn an „MSB“ ein Low-Signal anliegt und auf die Hälfte des Maximalwertes, wenn dort ein High-Signal anliegt. [1]

$\overline{\text{CS}}$ deaktiviert die Datenübertragung in das Schieberegister, wenn an diesem Pin ein Signal mit einem High-Pegel anliegt. Bei einer steigenden Flanke von „$\overline{\text{CS}}$/$\overline{\text{LDAC}}$“ werden die letzten 18 Bits, die in das Schieberegister geschrieben worden sind, in das Eingangsregister übertragen. [1]

CLK ist der Takteingang. Die Taktfrequenz kann bis zu 20 MHz betragen. [1]

SDI ist der Eingang des Schieberegisters. Mit den steigenden Flanken von „CLK“ werden die am „SDI“-Pin anliegenden Daten seriell in das Schieberegister geschrieben. [1]

SDO ermöglicht, die Daten des Schieberegisters auszulesen. Die Daten erscheinen 19 Takte nachdem sie am „SDI“-Pin eingelesen worden sind. [1]

$\overline{\text{LDAC}}$ überträgt die Daten der Eingangsregister in die DAC-Register, wenn an diesem Pin ein Signal mit einem Low-Pegel anliegt. [1]

3.2.3 Serielles Eingangsregister

Tabelle 1: Eingangsschieberegister des DACs „AD5544“ [1]

MSB																	LSB
B17	B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
A1	A0	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

In Tabelle 1 ist der Inhalt des 18-Bit-Schieberegisters des DACs zu sehen. Daten werden seriell in das Schieberegister geschrieben, wobei die 18 Bits vom MSB bis hin zum LSB in absteigender Reihenfolge gefüllt werden. Die ersten zwei Bits sind Adressbits (A1, A0) und ermöglichen, einen der vier Kanäle anzusteuern. Tabelle 2 zeigt, mit welchen Bitkombinationen die unterschiedlichen Kanäle angesteuert werden können. [1]

Tabelle 2: Definitionen der Adressbits des DACs „AD5544“ [1]

A1	A0	Kanal
0	0	A
0	1	B
1	0	C
1	1	D

Die letzten 16 Bits sind Datenbits und legen die Größe des Ausgangsstromes fest. Dabei ist „D15“ das MSB und „D0“ das LSB. Umso größer der Wert der Datenbits ist, desto größer ist der Betrag des Ausgangsstromes. [1]

3.3 ADC „AD7980“

Der ADC vom Typ „AD7980“ hat eine 16 Bit Auflösung. Er wandelt die Differenz der an den „**V+**“ und „**V-**“-Pins anliegenden Spannungen in Bezug auf die am „**REF**“-Pin anliegende Referenzspannung in einen Digitalwert um. [3]

Mit der folgenden Funktion lässt sich die Differenz der an den „V+“ und „V-“-Pins anliegenden Spannungen berechnen. [3]

$$V_{IN} = V_{REF} * \frac{D}{2^N} \qquad [3]$$

V_{IN} steht dabei für die Differenz der an den „V+“ und „V-“-Pins anliegenden Spannungen. V_{REF} steht für die Referenzspannung, welche in diesem Projekt 3 V beträgt. N ist die Auflösung des ADCs, also in diesem Fall 16, und D das dezimale Äquivalent der 16 Bits, die mit einem Auslesevorgang erhalten werden. [3]

Die folgenden Pins sind relevant für die FPGA-Ansteuerung:

Mithilfe des **SDI**-Pins können verschiedene Modi ausgewählt werden. In diesem Projekt ist das an diesem Pin anliegende Signal immer high, sodass der $\overline{CS}$-Modus ausgewählt ist. [3]

CNV sorgt dafür, dass die anliegende Spannung in einen Digitalwert umgewandelt wird, wenn das an diesem Pin anliegende Signal high wird. Zudem wird erkannt, dass der $\overline{CS}$-Modus ausgewählt ist. Dadurch wird der SDO-Pin freigegeben. [3]

SDO ist der Pin, an dem das Umwandlungsergebnis seriell erscheint. Dabei werden die Bits mit den fallenden Taktflanken gesetzt und sollten somit mit den steigenden Taktflanken abgespeichert werden. [3]

SCK ist der Takteingang. [3]

4 VHDL-Design

4.1 Allgemeines

Das VHDL-Design besteht aus acht Hauptkomponenten, die miteinander verschaltet sind. Jede Komponente des VHDL-Designs enthält einen asynchronen Reset. Die Zustandsmaschinen gehen bei einem Reset in ihre Anfangszustände über.

Abbildung 11 zeigt die Verschaltung der einzelnen Hauptkomponenten miteinander.

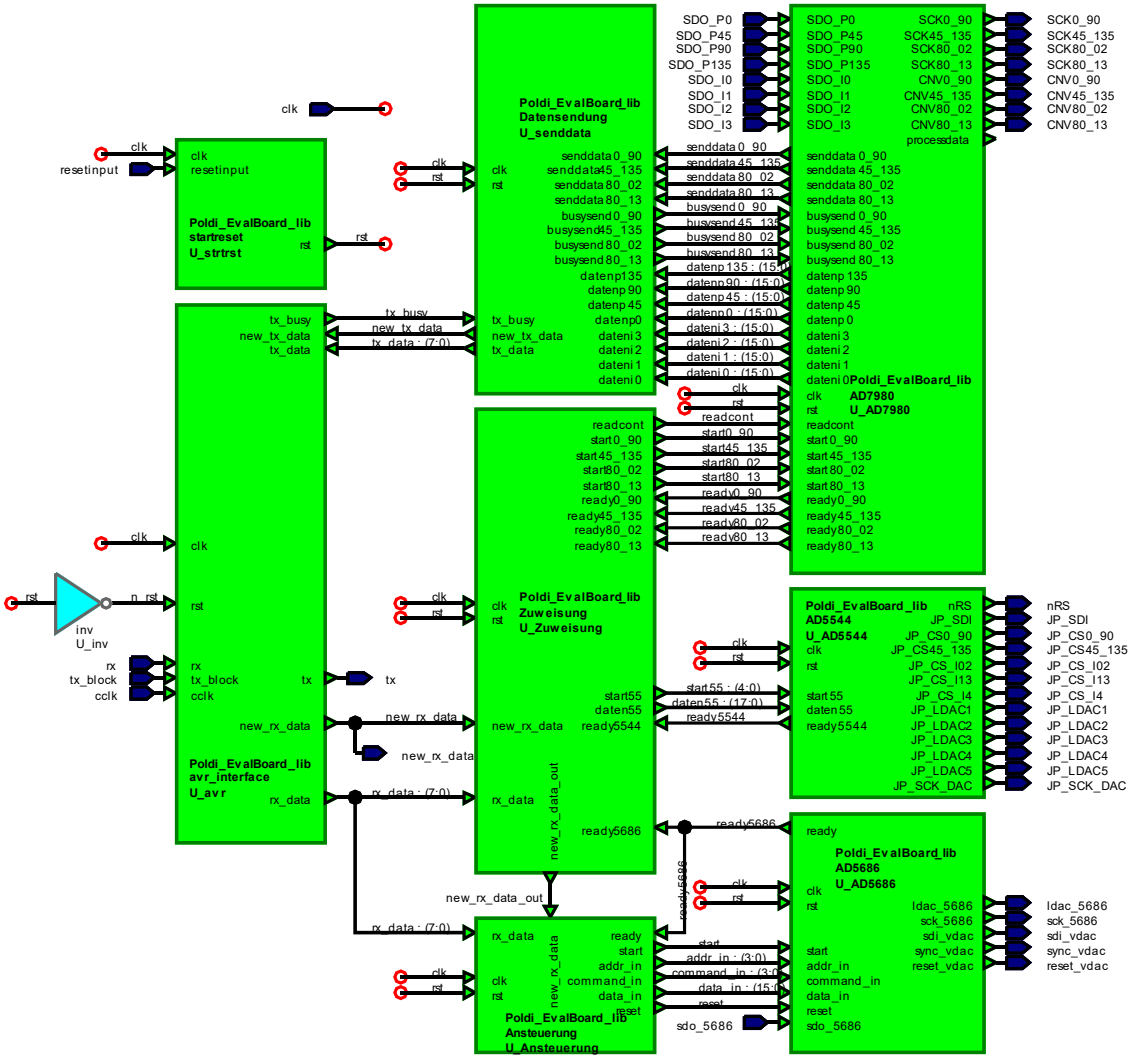

Abbildung 11: Struktur des VHDL-Designs

Die in Abbildung 11 zu sehenden Eingangs- und Ausgangsports wurden in der Datei „mojo.ucf“ Pins des Mojoboards zugewiesen, wobei der Port „clk“ dem 50 MHz-Systemtakt des Mojoboards zugewiesen wurde. Die Ports der rechten drei Komponenten

wurden Pins, die mit den DACs und ADCs verbunden sind, zugewiesen. Dabei heißen diese Ports genauso wie die Pins der DACs und ADCs in den Dateien des Projektes „POLDI“. Auf den Port „resetinput“ sowie auf die Ports, die mit der Komponente „avr_interface“ verbunden sind, wird in Kapitel 4.3 und in Kapitel 4.2 eingegangen.

Im Folgenden wird kurz erklärt, welche Hauptfunktionen die einzelnen Hauptkomponenten haben. Die Funktionsweise dieser Komponenten werden in den nachfolgenden Kapiteln der jeweiligen Komponenten detailliert erklärt.

- „**startreset**“ sorgt dafür, dass die restlichen Komponenten nach einer Konfiguration des FPGAs zurückgesetzt werden.
- „**avr_interface**“ wird vom Hersteller des Mojoboards zur Verfügung gestellt und verwaltet die Kommunikation zwischen dem Mikrocontroller und dem FPGA.
- „**Zuweisung**“ enthält ein Kommunikationsprotokoll und empfängt die Daten von „avr_interface“. Es sorgt für eine Synchronisation zwischen der Applikationssoftware und der FPGA Logik und übergibt den für die Ansteuerung der DACs und ADCs zuständigen Komponenten Steuersignale und Daten.
- „**Ansteuerung**“ und „**AD5686**“ sind für die Ansteuerung des DACs vom Typ „AD5686“ zuständig.
- „**AD5544**“ steuert die fünf verschiedenen DACs vom Typ „AD5544“ an.
- „**AD7980**“ liest die acht verschiedenen ADCs aus und speichert die ausgelesenen Daten ab.
- Mit „**Datensendung**“ können die von „AD7980“ gespeicherten Daten an die Komponente „avr_interface“ übergeben werden, welche die Daten mithilfe des Mikrocontrollers an den PC sendet.

Erklärungen der einzelnen Ein- und Ausgangssignale der Komponenten sind in den Kapiteln der jeweiligen Komponenten zu finden. Da fast alle Komponenten die Eingangssignale „clk“ und „rst“ haben, werden diese zur Vereinfachung einmalig erklärt:

- „**clk**“ ist der Systemtakt und beträgt 50MHz.
- „**rst**“ wird von der Komponente „startreset“ erzeugt. Es leitet das Signal der Resettaste des Mojoboards an die anderen Komponenten weiter und sorgt nach Implementierung des VHDL-Designs auf dem FPGA dafür, dass die anderen Komponenten zurückgesetzt werden.

4.2 Die Komponente „startreset“

Das Eingangssignal „resetinput“ dieser Komponente ist mit dem Port „resetinput“ verbunden (siehe Abbildung 11). Dieser Port wurde der Resettaste des Mojoboards zugewiesen. Diese Komponente hat die Aufgabe, nach einer Konfiguration des FPGAs alle Komponenten des FPGAs zurückzusetzen. Zusätzlich leitet sie das Signal der Resettaste des Mojoboards weiter an die anderen Komponenten. Dazu nutzen alle anderen Komponenten das Ausgangssignal „rst“ dieser Komponente als Reseteingang. Die anderen Komponenten werden zurückgesetzt, wenn „rst“ gleich null ist. Das Ausgangssignal „rst“ wird von dieser Komponente nach 50.000 Takten nach einer Konfiguration des FPGAs für 100 Takte gleich null gesetzt, sodass alle anderen Komponenten zurückgesetzt werden.

4.3 Die Komponente „avr_interface“

Die Komponente „avr_interface“ wird vom Hersteller des Mojoboards zur Verfügung gestellt und verwaltet die serielle Kommunikation zwischen dem Mikrocontroller und dem FPGA. Die Ports in Abbildung 11, die mit den Ein- und Ausgängen der Komponente „avr_interface“ verbunden sind, entsprechen den Pins des Mojoboards, die mit dem Mikrocontroller verbunden sind. Dies ist notwendig, damit die Kommunikation zwischen dem Mikrocontroller und dem FPGA stattfinden kann. [4]

Die Komponente ist ein Top-Modul und fügt die Module „cclk_detector“, „spi_slave“, „serial_rx“ und „serial_tx“ zusammen und ergänzt sie um einige Eigenschaften, wobei „cclk_detector“ dafür sorgt, dass der FPGA erst nach seiner Konfiguration die serielle Schnittstelle zur Kommunikation nutzen kann. Daten können über die serielle Schnittstelle mithilfe von „serial_rx“ empfangen und mithilfe von „serial_tx“ versendet werden. Bei dem Modul „spi_slave“ handelt es sich um eine SPI-Schnittstelle [4], welche in diesem Projekt jedoch nicht verwendet und deswegen entfernt wurde.
Zudem wurde das Top-Modul „avr_interface“ so bearbeitet, dass nichts mehr enthalten ist, was für die Steuerung des Moduls „spi_slave“ zuständig ist.

Die für die serielle Kommunikation relevanten Ein- und Ausgangssignale, die nicht mit Ports verbunden sind, haben folgende Bedeutungen:

- **„new_rx_data**" ist gleich eins, wenn neue Daten empfangen werden, die der 8-Bit-Vektor „**rx_data**" enthält. [4]
- „**tx_busy**" zeigt an, ob Daten an den Mikrocontroller gesendet werden können.[4]
- „**new_tx_data**" muss gleich eins gesetzt werden, wenn „tx_busy" gleich null ist und Daten gesendet werden sollen. Die zu sendenden Daten werden dem Modul über den 8-Bit-Vektor „**tx_data**" übergeben. Wenn „tx_busy" gleich eins ist, sollte das Signal „new_tx_data" wieder gleich null gesetzt werden. [4]

4.4 Die Komponente „Zuweisung"

4.4.1 Kommunikationsprotokoll

Tabelle 3: Format des Kommunikationsprotokolls

Synch (8 Bits)	Synch (8 Bits)	Synch (8 Bits)	Synch (8 Bits)	Befehlscode (8 Bits)	Daten (8 Bits)	Daten (8 Bits)	Daten (8 Bits)

Tabelle 3 zeigt das Format des in dieser Komponente enthaltenen Kommunikationsprotokolls. Die Daten der Komponente „avr_interface" werden byteweise empfangen. Zur Synchronisation müssen nacheinander vier Bytes empfangen werden, welche nur Einsen enthalten. Falls dies der Fall ist, entspricht das nächste empfangene Byte einem Befehlscode, der entscheidet, welcher DAC konfiguriert wird bzw. welche ADCs ausgelesen werden und was mit den ausgelesenen Daten passiert. Tabelle 4 zeigt, welcher DAC mit welchem Befehlscode ausgewählt wird, wobei die Bezeichnungen in der Tabelle den Bezeichnungen der Schaltpläne des Projektes POLDI entsprechen.

Tabelle 4: Kommunikationsprotokoll: Befehlscodes zur Auswahl der DACs

Befehlscode	ausgewählter DAC	einstellbare Signale
00000000	U$6 (AD5544)	SIN0, SIN90
00000001	U$12 (AD5544)	SIN45, SIN135
00000010	U$26 (AD5544)	VOH0, VOH2
00000011	U$28 (AD5544)	VOH1, VOH3
00000100	U$21 (AD5544)	V_FREE, REF_INTENSITY
00000101	U$7 (AD5686)	SHIELD, VT_INTEN, VT_POLA

Wird nach einer erfolgreichen Synchronisation mit dem nächsten Byte ein DAC ausgewählt, sind die nächsten drei Bytes für das Eingangsschieberegister des ausgewählten DACs bestimmt. Bei einer Auswahl eines DACs vom Typ „AD5544“ werden vom ersten Byte allerdings nur die zwei LSBs berücksichtigt, da diese DACs kein 24-Bit großes, sondern ein 18-Bit großes Eingangsschieberegister enthalten (siehe Kapitel 3.2.3).

Da die auf dem Evalboard vorhandenen ADCs immer paarweise mit einem SPI Bus verbunden sind, erlaubt die entworfene Logik die zeitgleiche Auslesung von entweder zwei, vier, sechs oder acht ADCs, wobei die ausgelesenen Daten anschließend an den PC gesendet werden. Alternativ kann die kontinuierliche Auslesung aller ADCs gestartet bzw. anschließend wieder gestoppt werden. Die kontinuierliche Auslesung kann nach einer Synchronisation mit dem Byte „10000111“ gestartet und mit „00000111“ wieder gestoppt werden.
Falls ADCs ausgelesen und die ausgelesenen Daten anschließend an den PC gesendet werden sollen, müssen nach einer Synchronisation die vier LSBs des nächsten Bytes gleich „0110“ sein. Mit den vier MSBs dieses Bytes wird eine Auswahl der ADCs getroffen. Mit Variation der vier MSBs des Bytes können unterschiedliche ADCs ausgelesen werden, deren Daten anschließend an den PC gesendet werden. Mit einem Bit der vier MSBs werden zwei ADCs ausgewählt. Tabelle 5 zeigt Variationen des Befehlscodes, aus denen hervorgeht, welches der vier MSBs des Befehlscodes welche ADCs auswählt. Dabei folgen die Bezeichnungen in der Tabelle den Bezeichnungen der Schaltpläne des Projektes POLDI. Die vier MSBs können dabei beliebig variiert werden.

Tabelle 5: Kommunikationsprotokoll: Befehlscodes zur Auswahl der ADCs

Befehlscode	ausgewählte ADCs	Signale, die mit den „V+“-Pins der ADCs verbunden sind
00010110	U$19, U$18	VOH1, VOH3
00100110	U$17, U$16	VOH0, VOH2
01000110	U$15, U$14	SIN45, SIN135
10000110	U$13, U$5	SIN0, SIN90
11110110	U$13, U$5, U$15, U$14, U$17, U$16, U$19, U$18	SIN0, SIN90, SIN45, SIN135, VOH0, VOH2, VOH1, VOH3

In Tabelle 7 im Anhang sind alle implementierten Befehlscodes des Kommunikationsprotokolls und deren Bedeutungen zu finden.

4.4.2 Funktionsweise

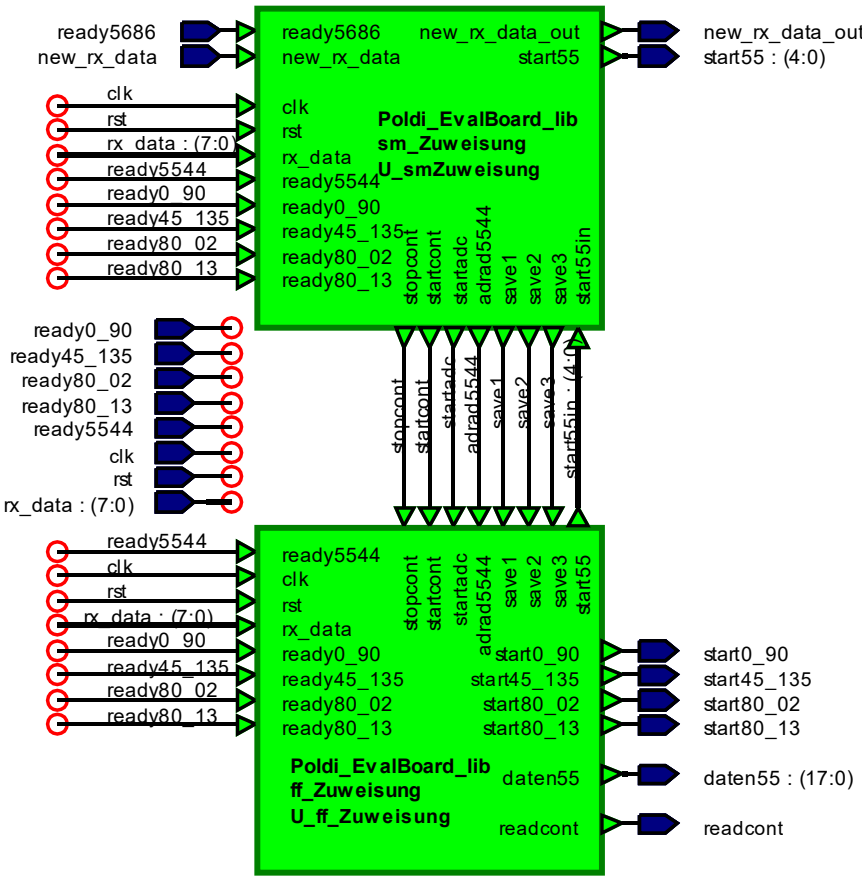

Abbildung 12: Struktur der Komponente „Zuweisung“

Abbildung 12 stellt die Struktur der Komponente „Zuweisung“ dar. Sie besteht aus der Komponente „sm_Zuweisung“, welche als Zustandsmaschine entworfen worden ist, und aus der Komponente „ff_Zuweisung“, welche nur synchronisierte Speicherelemente bzw. Flipflops beinhaltet.

Eingangssignale der Komponente „Zuweisung“:

- Wenn „**new_rx_data**“ gleich eins ist, bedeutet dies, dass neue Daten empfangen werden.
- „**rx_data**“ ist ein 8-Bit-Vektor und beinhaltet die empfangenen Daten.
- „**ready5686**“ zeigt an, ob die Komponente „AD5686“ bereit ist, neue Daten zu empfangen.
- „**ready5544**“ zeigt an, ob die Komponente „AD5544“ neue Daten erhalten kann.

- „**ready0_90**“, „**ready45_135**“, „**ready80_02**“ und „**ready80_13**“ geben an, ob die Komponente „AD7980“ bereit ist, Signale zu verarbeiten, die bestimmen, welche ADCs ausgelesen und deren Daten anschließend an den PC gesendet werden.

Ausgangssignale der Komponente „Zuweisung“:

- „**new_rx_data_out**“ teilt der Komponente „Ansteuerung“ mit, ob sie die Daten, die der Vektor „rx_data“ der Komponente „avr_interface“ enthält, verarbeiten soll.
- „**start55**“ ist ein 5-Bit-Vektor, welcher der Komponente „AD5544“ mitteilt, für welchen DAC vom Typ „AD5544“ die Daten, die der 18-Bit-Vektor „**daten55**“ enthält, bestimmt sind.
- „**start0_90**“, „**start45_135**“, „**start80_02**“ und „**start80_13**“ sind Startsignale für die Komponente „AD7980“. Mit dem Setzen eines dieser Signale wird die Auslesung von zwei ADCs und die anschließende Versendung der ausgelesenen Daten an den PC gestartet.
- Mit „**readcont**” wird die kontinuierliche Auslesung aller ADCs gestartet und gestoppt.

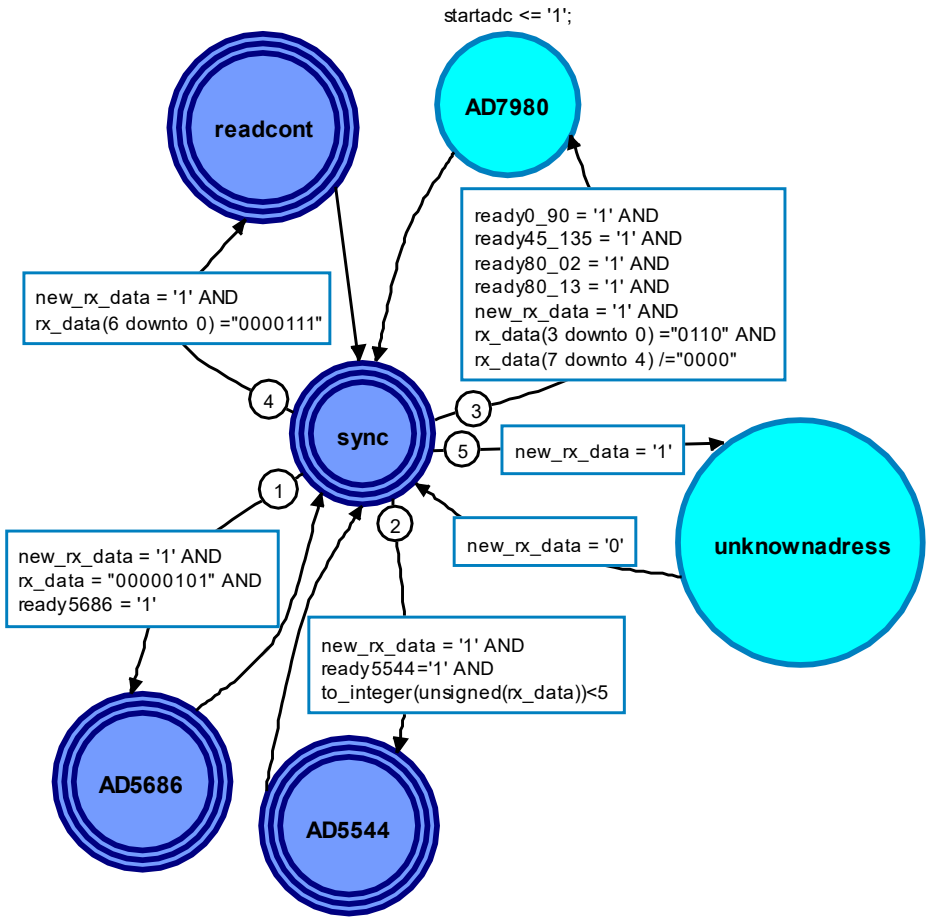

Abbildung 13: Zustandsmaschine „sm_Zuweisung“

Abbildung 13 zeigt das Zustandsdiagramm der Komponente „sm_Zuweisung“. Es besteht aus den hierarchischen Zuständen „sync“, „AD5686“. „AD5544“, „readcont“ und aus den Zuständen „AD7980“ und „unknownaddress“.

Nachdem die in Kapitel 4.4.1 erklärte Synchronisation im hierarchischen Zustand „sync“ abgeschlossen ist, entscheidet u.a. das nächste empfangene Byte über den nächsten Zustand. Die Übergangsbedingungen werden jeweils in den Abschnitten, die den hierarchischen Zuständen zugeordnet sind, erläutert. Die Bedingung für den Übergang der Zustandsmaschine in den Zustand „AD7980“ sowie der Zustand an sich werden im Abschnitt beschrieben, der den hierarchischen Zustand „readcont“ erklärt, da die beiden Zustände logisch zusammenhängen.

In Abbildung 13 sind die Bedingungen zu sehen, mit denen die Zustandsmaschine vom letzten Zustand des hierarchischen Zustands „sync“ mit der nächsten steigenden Taktflanke in einen anderen Zustand übergeht. Falls die Bedingungen mit den vier höchsten Prioritäten nicht erfüllt sind, aber dennoch neue Daten anliegen, wechselt die

Zustandsmaschine in den Zustand „unknownadress“. Sie geht erst wieder in „sync“ über, wenn keine neuen Daten mehr anliegen.

Die Zustandsmaschine nimmt bis auf einige begründete Ausnahmen jedes Mal bevor die Bedingung abgefragt wird, ob ein neues Byte anliegt, einen Zustand an mit der Bedingung, dass keine neuen Daten anliegen. Dies ist notwendig, da das Signal „new_rx_data“ länger als einen Takt gleich eins ist und somit ansonsten mehrere Zustände mit der Bedingung, dass „new_rx_data“ gleich eins ist, nacheinander mit demselben Byte erreicht werden könnten, obwohl nur ein einziges neues Byte empfangen wurde.

Der Anfangszustand der Zustandsmaschine befindet sich im hierarchischen Zustand „sync“.

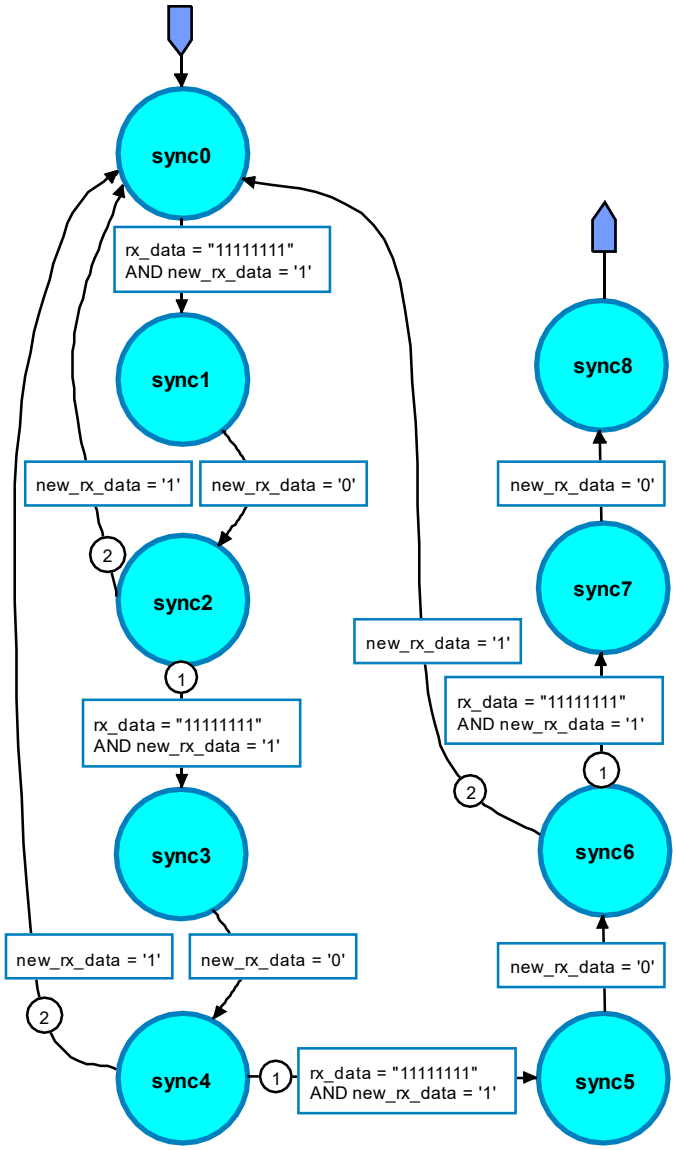

Abbildung 14: hierarchischer Zustand „sync“ der Komponente „sm_Zuweisung“

In Abbildung 14 ist das Zustandsdiagramm des hierarchischen Zustands „sync“ zu sehen. Dieser Zustand dient zur Synchronisation. Der Anfangszustand der Komponente „sm_Zuweisung“ ist „sync0“. Der Zustand „sync1“ wird mit einer steigenden Taktflanke erreicht, wenn ein neues Byte empfangen wird, welches nur mit Einsen gefüllt ist. Von „sync1“ wird „sync2“ mit einer steigenden Taktflanke erreicht, wenn keine neuen Daten anliegen. Die erläuterten Bedingungen, um von „sync0“ zu „sync1“ und von „sync1“ zu „sync2“ zu gelangen, müssen vier Mal nacheinander erfüllt werden, um den Zustand „sync8“ zu erreichen. Falls in den Zuständen „sync2“, „sync4“ oder „sync6“ ein neues Byte empfangen wird, welches nicht nur Einsen enthält, wird mit der nächsten steigenden Taktflanke in den Zustand „sync0“ gewechselt, um zu gewährleisten, dass eine vollständige Synchronisation mit vier Bytes, welche vollständig mit Einsen gefüllt sind, stattgefunden hat, bevor der hierarchische Zustand „sync“ verlassen wird.

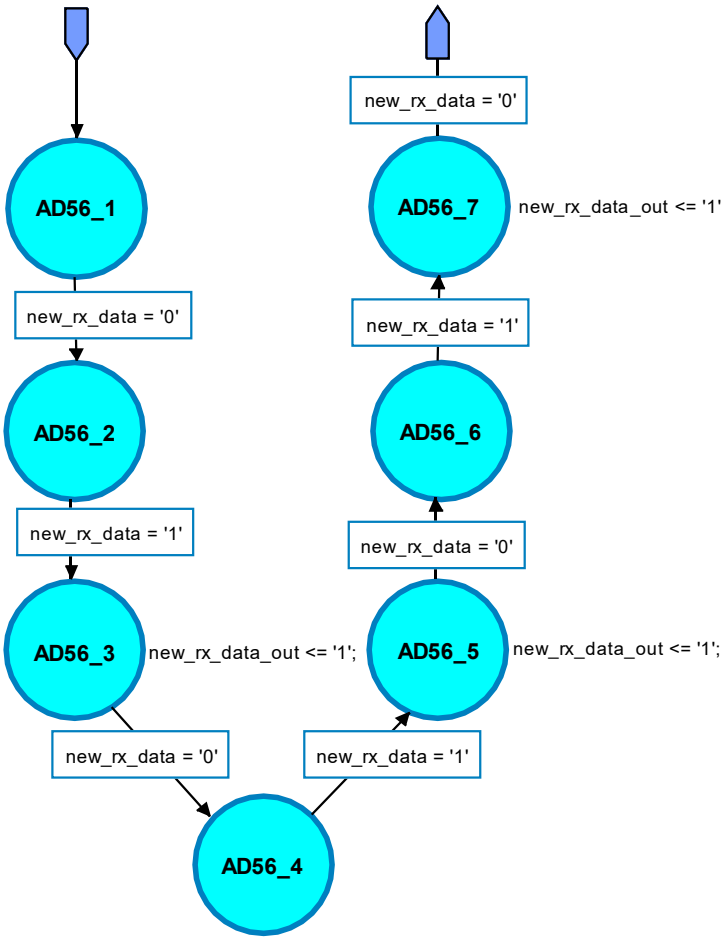

Abbildung 15: hierarchischer Zustand „AD5686“ der Komponente „sm_Zuweisung“

Der hierarchische Zustand „AD5686“, welcher in Abbildung 15 dargestellt ist, wird mit einer steigenden Taktflanke nach einer Synchronisation erreicht, wenn ein neues Byte mit

dem Inhalt „00000101“ empfangen wird und die Komponente „AD5686“ bereit ist, neue Daten zu empfangen. Vom Zustand „AD56_1“ wird mit einer steigenden Taktflanke in den Zustand „AD56_2“ übergegangen, wenn keine neuen Daten anliegen. Danach müssen dreimal hintereinander abwechselnd neue und dann keine neuen Daten anliegen, damit die Zustandsmaschine wieder in den hierarchischen Zustand „sync“ gelangt. In den Zuständen, die erreicht werden, wenn neue Daten anliegen, wird das Signal „new_rx_data_out“ gleich eins gesetzt. Dadurch weiß die Komponente „Ansteuerung“, dass sie die Daten, die der Vektor „rx_data“ der Komponente „avr_interface“ enthält, verarbeiten und mithilfe der Komponente „AD5686“ in das Eingangsschieberegister des DACs vom Typ „AD5686“ schreiben soll.

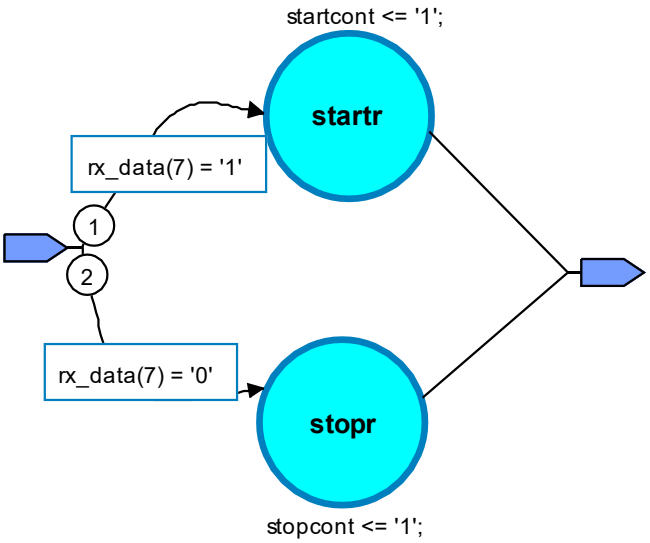

Abbildung 16: hierarchischer Zustand „readcont“ der Komponente „sm_Zuweisung“

In Abbildung 16 ist der hierarchische Zustand „readcont“ zu sehen. Dieser wird mit einer steigenden Taktflanke erreicht, wenn die sieben LSBs des nach einer Synchronisation empfangenen Bytes gleich „0000111“ sind (siehe Abbildung 13). Das MSB des Bytes bestimmt über den Nachfolgezustand. Ist dieses gleich eins, wechselt die Zustandsmaschine in den Zustand „startr“ und das Signal „startcont“ wird gleich eins gesetzt und die Komponente „ff_Zuweisung“ setzt mit der nächsten steigenden Taktflanke das Signal „readcont“ high, sodass die Komponente „AD7980“ weiß, dass eine kontinuierliche Auslesung aller ADCs gestartet werden soll. Ist das MSB des Bytes allerdings gleich null, so wird das Signal „stopcont“ im Zustand „stopr“ gleich eins gesetzt, was zur Folge hat, dass die Komponente „ff_Zuweisung“ das Ausgangssignal „readcont“ mit der nächsten

steigenden Taktflanke gleich null setzt. Dadurch wird der Komponente „AD7980“ mitgeteilt, dass die kontinuierliche Auslesung gestoppt werden soll.

Sowohl vom Zustand „stopr“ als auch vom Zustand „startr“ wird ohne Bedingung mit der nächsten steigenden Taktflanke in den hierarchischen Zustand „sync“ übergegangen. Es muss also nicht abgewartet werden, bis keine neuen Daten mehr anliegen, da diese Zustände nur erreicht werden können, wenn das empfangene Byte ungleich „11111111“ ist. Somit ist sichergestellt, dass das Byte nicht dazu führt, dass es als ein Byte zur Synchronisation genutzt wird. Dasselbe gilt auch für den in Abbildung 13 dargestellten Zustand „AD7980“, da dieser Zustand nur erreicht werden kann, wenn das nach einer Synchronisation empfangene Byte nicht voller Einsen ist.

Damit der Zustand „AD7980“ nach einer Synchronisation mit einer steigenden Taktflanke erreicht wird, muss ein neues Byte empfangen werden, bei welchem die vier LSBs gleich „0110“ und die vier MSBs ungleich „0000“ sind. Zudem muss die Komponente „AD7980“ bereit sein, die Signale, die die Auslesung der ADCs und die Versendung der ausgelesenen Daten an den PC einleiten, zu verarbeiten. Im Zustand „AD7980“ wird das Signal „startadc“ high gesetzt. Dadurch verarbeitet die Komponente „ff_Zuweisung“ mit der nächsten steigenden Taktflanke die vier MSBs des empfangenen Bytes. Sie trifft mithilfe dieser vier MSBs eine Auswahl der Ausgangssignale „start0_90“, „start45_135“, „start80_02“ und „start80_13“, die sie solange gleich eins setzt, bis die Signale von der Komponente „AD7980“ verarbeitet worden sind. Es werden die Signale gleich eins gesetzt, welche die Auslesung der ausgewählten ADCs und die anschließende Versendung der ausgelesenen Daten an den PC einleiten (siehe Tabelle 5).

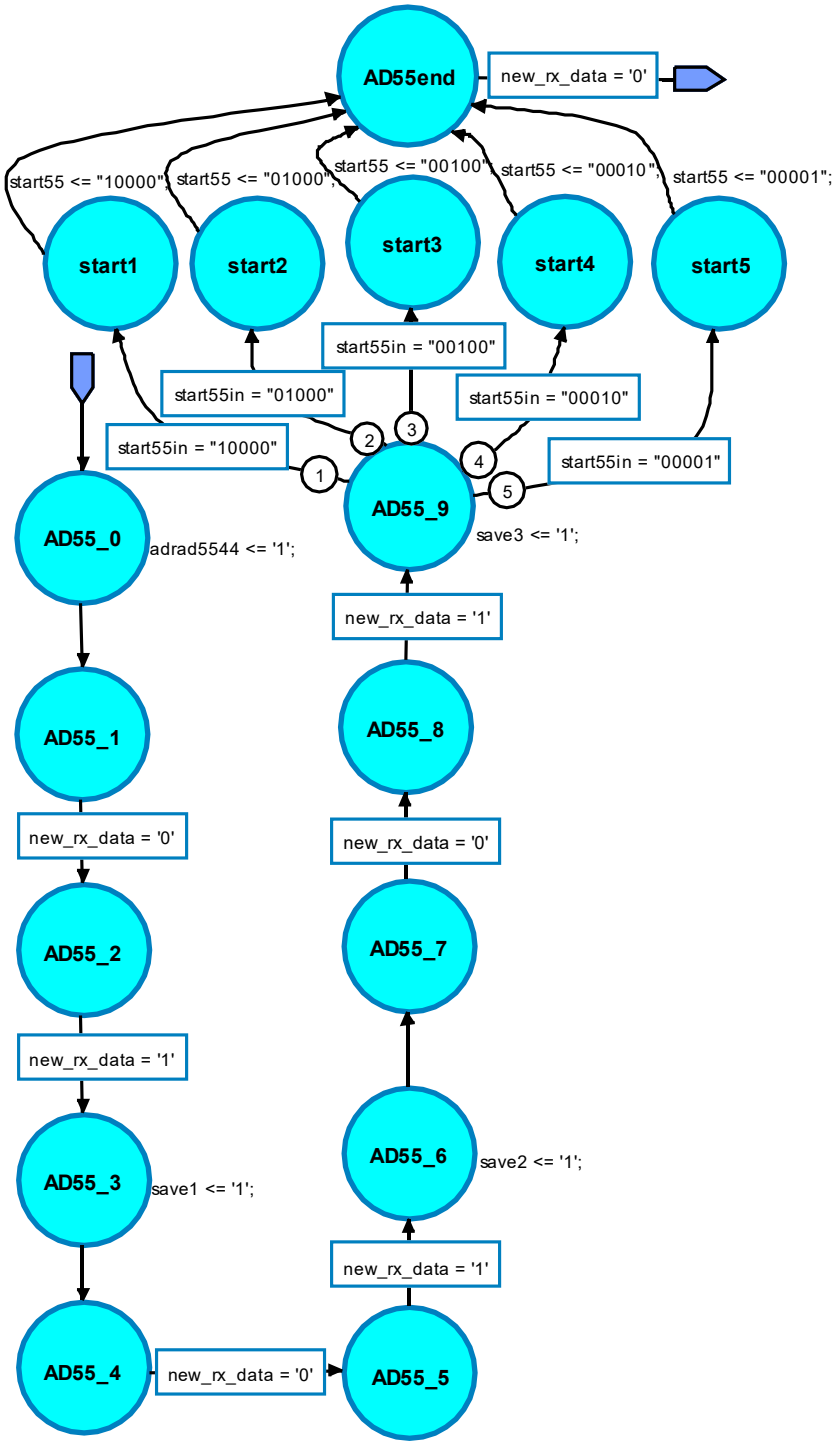

Abbildung 17: hierarchischer Zustand „AD5544“ der Komponente „sm_Zuweisung“

Abbildung 17 zeigt den hierarchischen Zustand „AD5544“ der Zustandsmaschine, welcher mit einer steigenden Taktflanke erreicht wird, wenn das dezimale Äquivalent des nach einer Synchronisation empfangenen Bytes kleiner als fünf ist. Im ersten Zustand „AD55_0“ wird das Signal „adrad5544“ gleich eins gesetzt. Dadurch wird der Komponente „ff_Zuweisung“ mitgeteilt, dass mit dem empfangenen Befehlscode ein DAC vom Typ „AD5544“ ausgewählt worden ist. Die Komponente „ff_Zuweisung“ setzt

dann mit der nächsten steigenden Taktflanke die Bits ihres Ausgangsvektors „start55“, um den adressierten DAC auszuwählen. Der Ausgangsvektor „start55“ ist mit dem Eingangsvektor „start55in“ der Komponente „sm_Zuweisung“ verbunden.

Mit den nächsten drei empfangenen Bytes setzt die Zustandsmaschine jeweils unterschiedliche Signale gleich eins. Dadurch speichert die Komponente „ff_Zuweisung“ die drei Bytes an der korrekten Stelle im Vektor „daten55“ ab, wobei vom ersten Byte nur die zwei LSBs berücksichtigt werden (siehe Kapitel 4.4.1). Im Zustand „AD55_9“ wird durch das Setzen des entsprechenden Signals das letzte für das Eingangsschieberegister des ausgewählten DACs bestimmte Byte abgespeichert. Von diesem Zustand aus wird entsprechend des Inhalts des Eingangsvektors „start55in“ in den passenden Zustand gewechselt, der die Bits des Ausgangsvektor „start55“ gemäß des Eingangsvektors „start55in“ setzt. Dadurch wird der Komponente „AD5544“ mitgeteilt, dass sie die im Vektor „daten55“ abgespeicherten Daten in den mit dem Befehlscode adressierten DAC vom Typ „AD5544“ schreiben soll. Ausgehend vom Zustand, der den Ausgangsvektor „start55“ setzt, wird mit der nächsten steigenden Taktflanke in den Zustand „AD55end“ gewechselt. Mit einer steigenden Taktflanke wird dieser letzte Zustand und damit auch insgesamt der hierarchische Zustand „AD5544“ verlassen, sobald der Empfang neuer Daten nicht mehr angezeigt wird.

4.4.3 Simulation

In den folgenden Simulationen werden nur die für die Funktionsweise des jeweils gezeigten Ablaufs relevanten Signale der Komponente „Zuweisung“ dargestellt. Die Bits des Vektors „rx_data“ sind in jeder Abbildung einzeln dargestellt, um nachvollziehen zu können, welche Daten zu welchem Zeitpunkt empfangen werden.

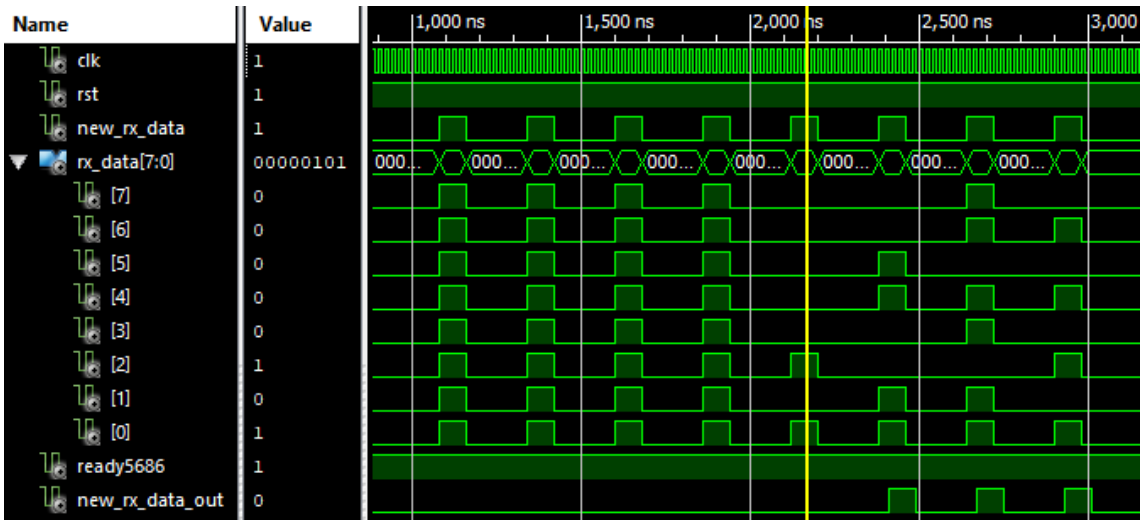

Abbildung 18: Simulation 1 der Komponente „Zuweisung"

In Abbildung 18 ist zu erkennen, dass vier neue Bytes voller Einsen nacheinander empfangen werden. Damit ist die Synchronisation abgeschlossen. Da das Signal „ready5686“ gleich eins und das nächste empfangene Byte gleich „00000101“ ist, wird damit (siehe Kapitels 4.4.1) der DAC vom Typ „AD5686“ ausgewählt. Da das Signal „ready5686“ gleich eins ist, wird mit den nächsten drei empfangenen Bytes das Ausgangssignal „new_rx_data_out“ gleich eins gesetzt, wodurch der Komponente „Ansteuerung“ mitgeteilt wird, dass sie die empfangenen Bytes verarbeiten soll und mithilfe der Komponente „AD5686“ in das Eingangsschieberegister des DACs vom Typ „AD5686“ schreiben soll.

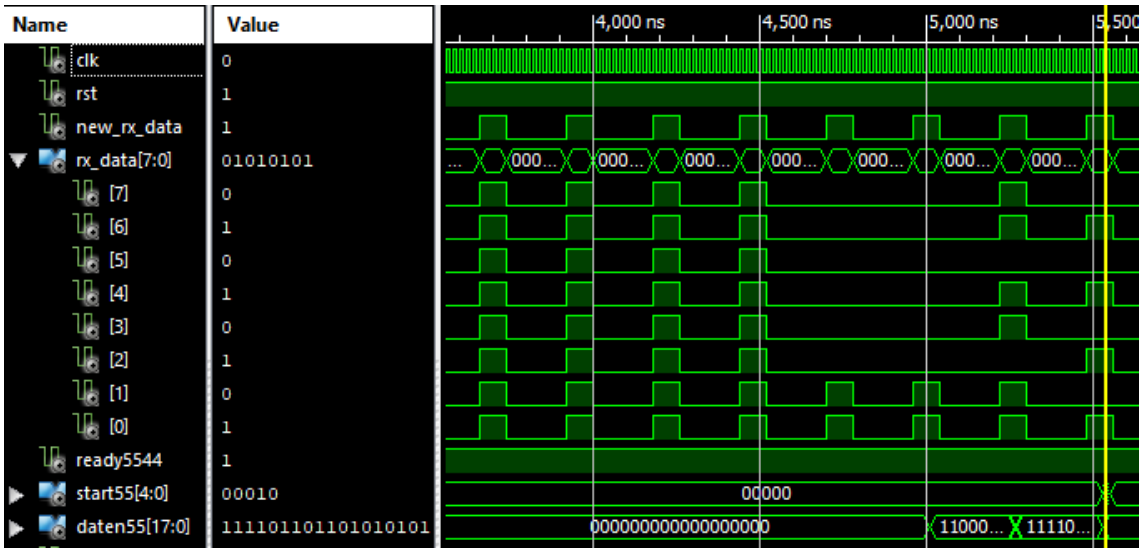

Abbildung 19: Simulation 2 der Komponente „Zuweisung"

Wie in Abbildung 19 zu sehen ist, werden vier Bytes voller Einsen zur Synchronisation empfangen. Das nächste Byte ist gleich „00000011“. Da das Signal „ready5544“ gleich eins ist, wird damit ein DAC vom Typ „AD5544“ ausgewählt (siehe Kapitel 4.4.1). Die

zwei LSBs des nächsten empfangenen Bytes und die zwei darauf empfangenen Bytes werden im Vektor „daten55“ abgespeichert. Die Daten unter „Value“ oben links beziehen sich auf den Zeitpunkt, an dem sich der gelbe Marker befindet. Es lässt sich erkennen, dass die Daten richtig im Vektor „daten55“ abgespeichert wurden. Zudem ist der Vektor „start55“ zu diesem Zeitpunkt für einen Takt gleich „00010“, um der Komponente „AD5544“ mitzuteilen, dass die im Vektor „daten55“ abgespeicherten Daten in den adressierten DAC vom Typ „AD5544“ geschrieben werden sollen.

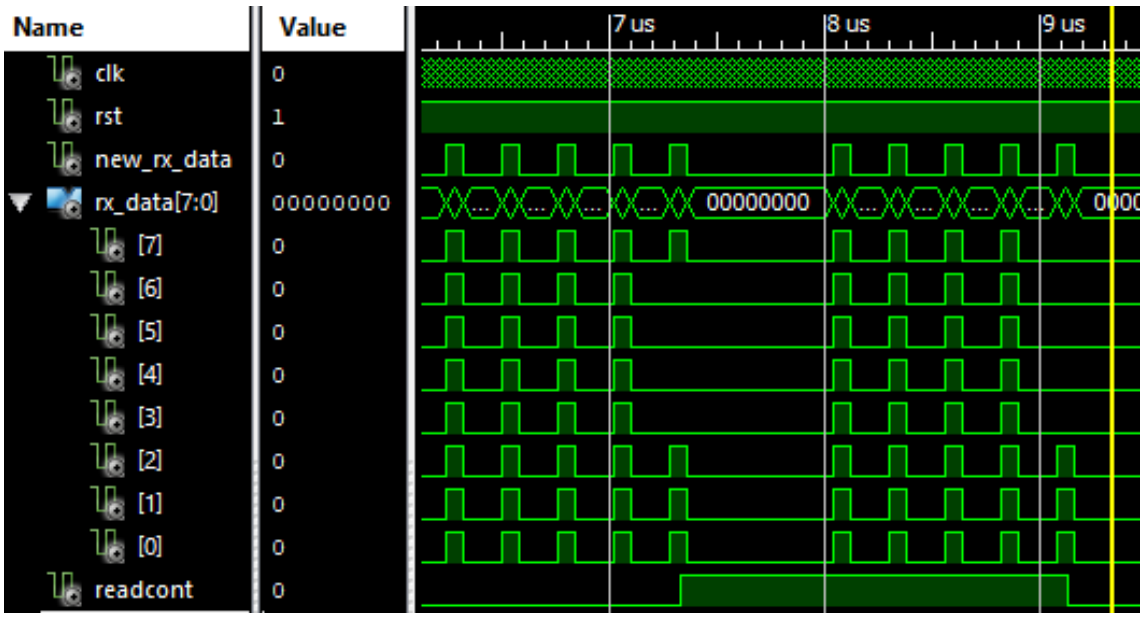

Abbildung 20: Simulation 3 der Komponente „Zuweisung"

Abbildung 20 zeigt, dass das Signal „readcont“, welches der Komponente „AD7980“ mitteilt, dass eine kontinuierliches Auslesung aller ADCs stattfinden soll, nach erfolgter Synchronisation mit dem Befehlscode „10000111“ gesetzt und mit dem Befehlscode „00000111“ zurückgesetzt wird.

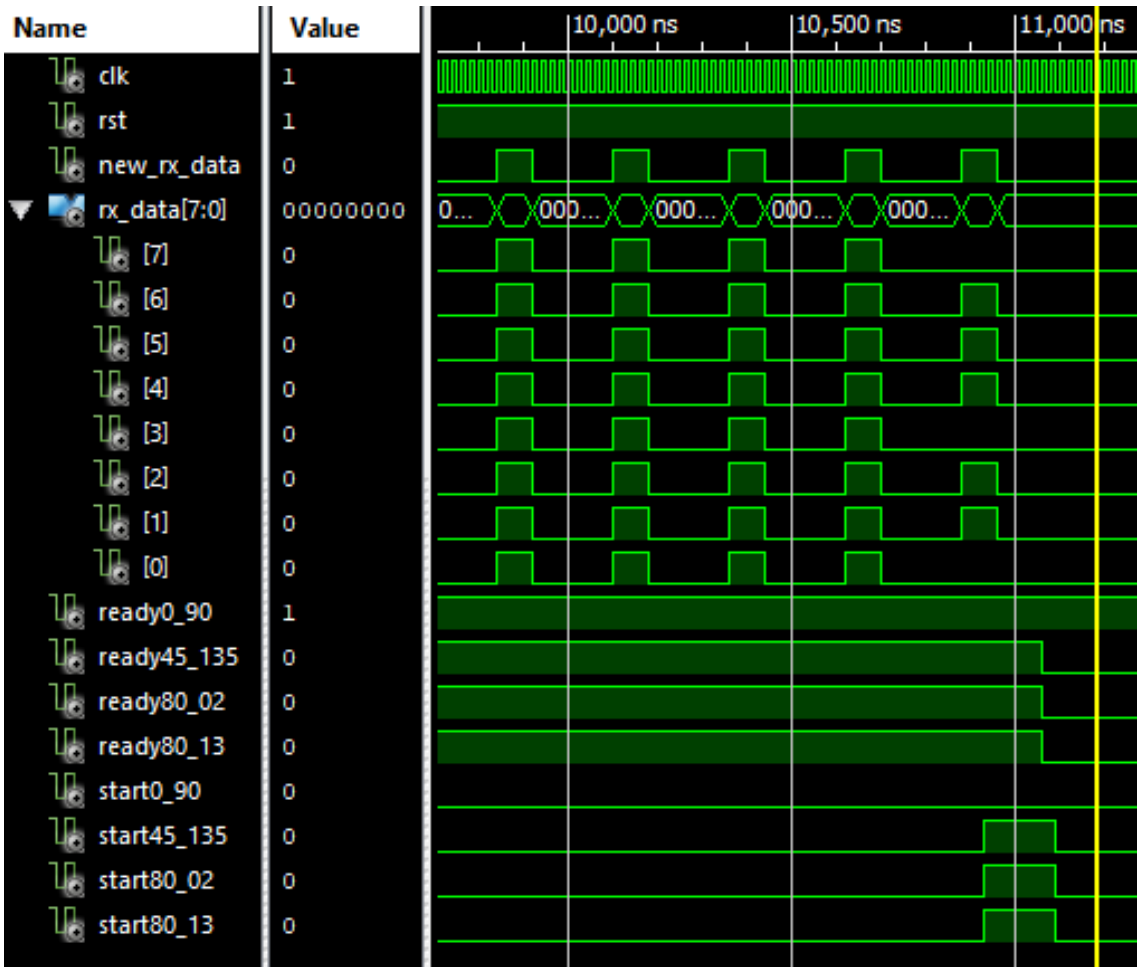

Abbildung 21: Simulation 4 der Komponente „Zuweisung"

In Abbildung 21 ist zu sehen, dass vier Bytes voller Einsen zur Synchronisation empfangen werden. Danach wird ein Befehlscode mit dem Inhalt „01110110“ empfangen, mit dem sechs ADCs gleichzeitig für die Auslesung ausgewählt werden. Da die zu sehenden „ready“-Signale alle gleich eins sind, nehmen drei „start“-Signale einen High-Pegel an und leiten damit die Auslesung der adressierten ADCs und die Versendung der ausgelesenen Daten an den PC ein. Nachdem die dazugehörigen „ready“-Signale nicht mehr gleich eins sind, nehmen die drei „start“-Signale wieder den Wert null an.

4.5 Die Komponenten „Ansteuerung“ & „AD5686“

Die Komponenten „Ansteuerung“ und „AD5686“ wurden von der Bachelorarbeit [6] übernommen und in dieses VHDL-Design integriert. Sie dienen der Ansteuerung des DACs vom Typ „AD5686“. Die Funktionsweise kann dieser Bachelorarbeit entnommen werden. Das Eingangssignal „new_rx_data“ der Komponente „Ansteuerung“ war in dieser Arbeit mit dem Ausgangssignal „new_rx_data“ der Komponente „avr_interface“ verbunden. Zudem war der Eingangsvektor „rx_data“ der Komponente „Ansteuerung“ mit dem Ausgangsvektor „rx_data“ der Komponente „avr_interface“ verbunden. Die Komponente „Ansteuerung“ verarbeitet die Daten des Vektors „rx_data“ nur, wenn das Eingangssignal „new_rx_data“ gleich eins ist [6]. Da in diesem VHDL-Design aber nicht alle Daten für den DAC vom Typ „AD5686“ bestimmt sind, wird das Eingangssignal „new_rx_data“ der Komponente „Ansteuerung“ mit dem Ausgangssignal „new_rx_data_out“ der Komponente „Zuweisung“ verbunden. Dieses Signal wird nur dann high gesetzt, wenn Daten empfangen worden sind, die für den DAC vom Typ „AD5686“ bestimmt sind.

Außerdem verfügten die Komponenten „Ansteuerung“ und „AD5686“ über keine asynchronen Resets, weshalb welche hinzugefügt worden sind.

4.6 Die Komponente „AD5544“

4.6.1 Funktionsweise

Die Komponente „AD5544“ dient der Ansteuerung der fünf DACs vom Typ „AD5544“. Die Takt- und Dateneingänge der fünf DACs sind jeweils mit denselben Pins des Mojoboards verbunden. Die Auswahl des zu beschreibenden DACs erfolgt über individuelle Chip-Select (CS) Signale. Folglich können während eines Schreibvorganges keine unterschiedlichen Daten in verschiedene DACs geschrieben werden und die Konfiguration der DACs muss dementsprechend einzeln und sequentiell erfolgen.

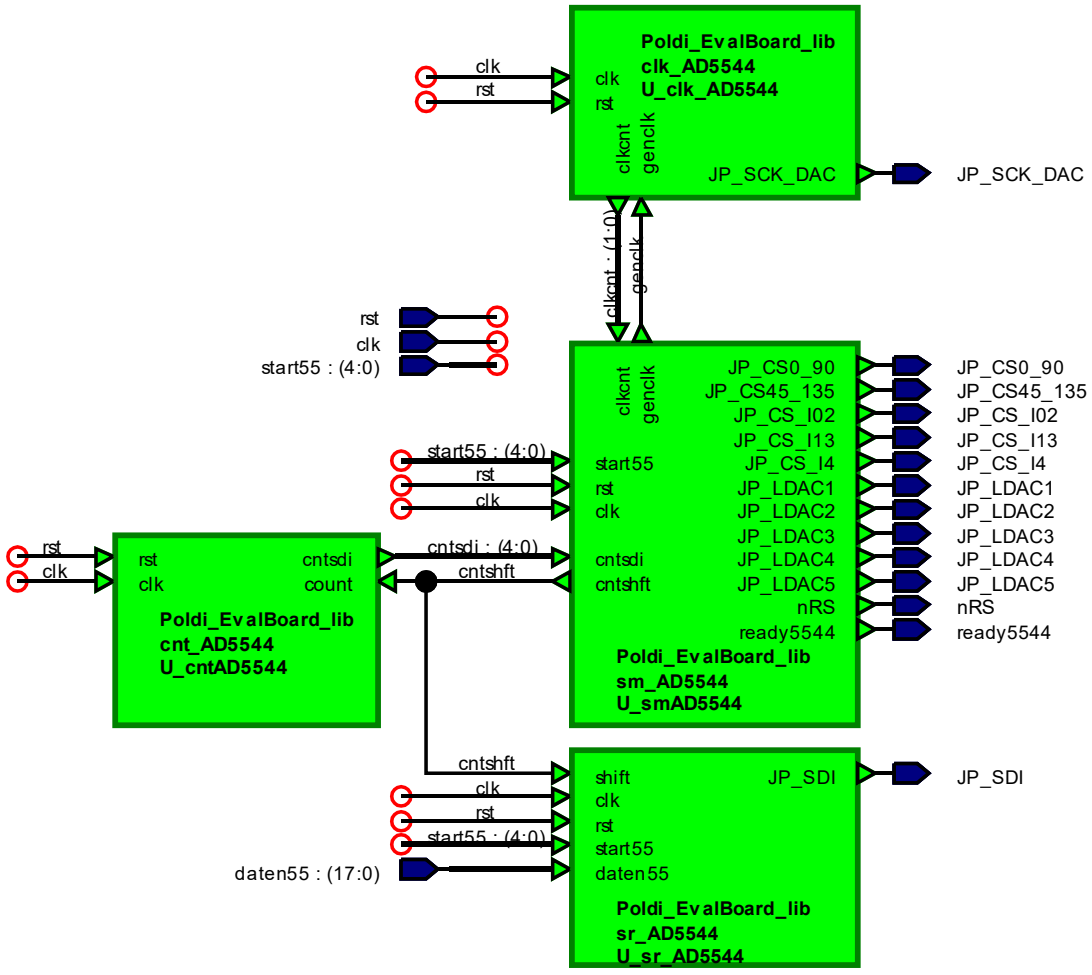

Abbildung 22: Struktur der Komponente „AD5544"

Abbildung 22 visualisiert die Struktur der Komponente „AD5544“. Die Komponente besteht aus vier Modulen, wobei „cnt_AD5544“ einem Zähler, sr_AD5544“ einem Schieberegister, „clk_AD5544“ einem Taktgenerator und „sm_AD5544“ einer Zustandsmaschine entspricht.

Eingangssignale der Komponente „AD5544“:

- „**start55**“ gibt an, welcher der fünf DACs vom Typ „AD5544“ angesteuert werden soll. Dieser Vektor sollte nur für einen Takt ungleich „00000“ gesetzt werden.
- „**daten55**“ ist ein 18 Bit großer Vektor und enthält die Daten, die in das Eingangsschieberegister des ausgewählten DACs geschrieben werden sollen.

Ausgangssignale der Komponente „AD5544“:

- „**ready5544**“ signalisiert, ob die Komponente bereit ist, einen DAC vom Typ „AD5544“ anzusteuern.
- „**nRS**“ ist ein Signal für die „$\overline{\text{RS}}$“-Pins der DACs vom Typ „AD5544“ und bleibt konstant high, sodass die Register dieser DACs nie durch dieses Signal auf null gesetzt werden.
- „**JP_CS_0_90**“, „**JP_CS_45_135**“, „**JP_CS_I02**“, „**JP_CS_I13**“ und „**JP_CS_I4**“ sind Signale für die „$\overline{\text{CS}}$“-Pins der DACs vom Typ „AD5544“.
- „**JP_LDAC1**“, „**JP_LDAC2**“, „**JP_LDAC3**“, „**JP_LDAC4**“ und „**JP_LDAC5**“ sind Signale für die „$\overline{\text{LDAC}}$“-Pins der DACs vom Typ „AD5544“.
- „**JP_SDI**“ ist das Signal für die „SDI“-Pins der DACs vom Typ „AD5544“.
- „**JP_SCK_DAC**“ ist das Signal für die „CLK“-Pins der DACs vom Typ „AD5544“.

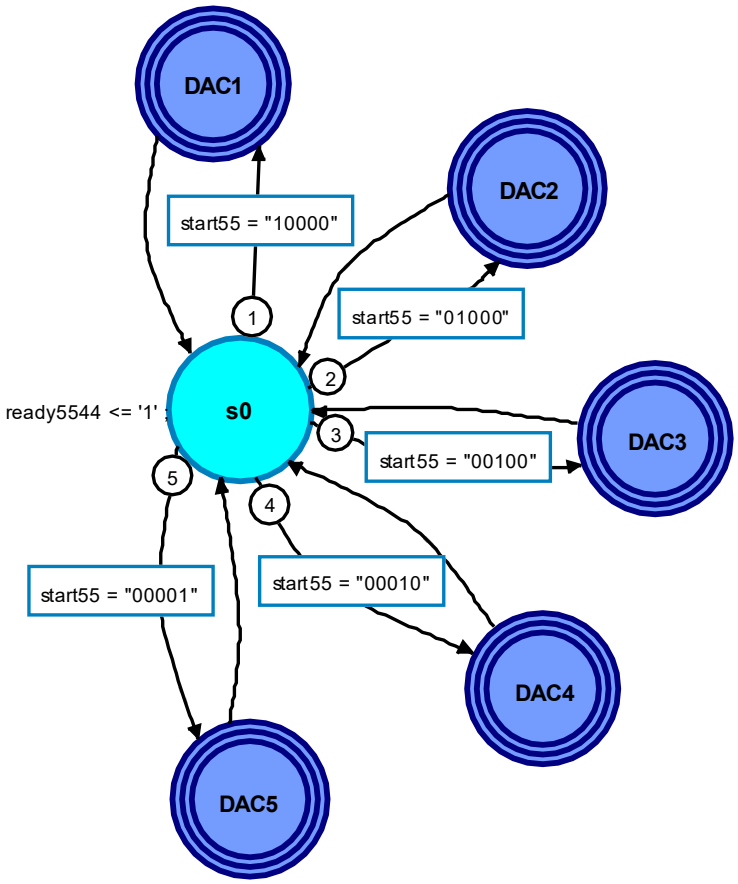

Abbildung 23: Zustandsmaschine „sm_AD5544"

Abbildung 23 zeigt das Zustandsdiagramm der Komponente „sm_AD5544“. Wenn der Vektor „start55“ mit einer der in der Abbildung zu sehenden Übergangsbedingungen übereinstimmt, wechselt die Zustandsmaschine mit der nächsten steigenden Taktflanke vom Anfangszustand „s0“ in einen der fünf hierarchischen Zustände. Dabei bestimmt der Vektor „start55“, in welchen der fünf hierarchischen Zustände „DAC1“ bis „DAC5“ gewechselt und somit, welcher der fünf DACs vom Typ „AD5544“ angesteuert wird. Wenn der Vektor „start55“ ungleich „00000“ ist, schreibt die Komponente „sr_AD5544“ mit der nächsten steigenden Taktflanke das MSB des Vektors „daten55“ auf das Signal „JP_SDI“. Das Signal „ready5544“ wird nur im Anfangszustand „s0“ gleich eins gesetzt.

Die hierarchischen Zustände sind jeweils für die Aktivierung der Signale, die alleinig an einem der fünf DACs vom Typ „AD5544“ anliegen, zuständig. Deshalb haben diese Zustände auch ein sehr ähnliches Aussehen. Die Signale „JP_SDI“ für die Daten und „JP_SCK_DAC“ für den Takt, die an allen DACs vom Typ „AD5544“ gleichzeitig anliegen, werden durch die Komponenten „sr_AD5544“ und „clk_AD5544“ generiert.

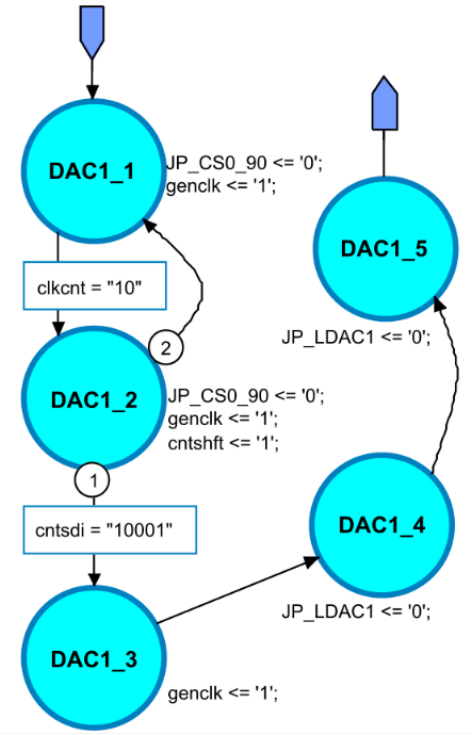

Abbildung 24: hierarchischer Zustand „DAC1“ der Komponente „sm_AD5544“

Da die fünf hierarchischen Zustände gleich aufgebaut sind, wird nur der hierarchische Zustand „DAC1“ erläutert, welcher in Abbildung 24 zu sehen ist. Wenn das Signal „genclk“ high ist, inkrementiert „clk_AD5544“ den Vektor „clkcnt“ mit jeder steigenden Taktflanken um eins. Der Vektor wird mit dem Anfangswert „00“ initialisiert, wird dann bis „11“ hochgezählt und fällt dann wieder auf den Wert „00“. Dementsprechend wird mithilfe dieses Vektors ein 12,5 MHz Takt für die Takteingänge der DACs vom Typ „AD5544“ generiert. Wenn das Signal „genclk“ nicht gleich eins ist, setzt die Komponente „clk_AD5544“ den Vektor „clkcnt“ gleich „00“. Der generierte 12,5 MHz Takt hat einen Low-Pegel, wenn der Vektor „clkcnt“ gleich „00“ oder „01“ ist und einen High-Pegel, wenn der Vektor „clkcnt“ gleich „10“ oder „11“ ist. Das Signal „genclk“ wird in den ersten drei Zuständen in Abbildung 24 high gesetzt. Außerdem wird in den ersten zwei Zuständen das zur Ansteuerung des DACs notwendige Signal „JP_CS0_90“ gleich „0“ gesetzt.

Wenn der Vektor „clkcnt“ gleich „10“ ist, wird mit der nächsten steigenden Taktflanke in den Zustand „DAC1_2“ gewechselt. Dort wird das Signal „cntshft“ high gesetzt. Immer, wenn das Signal „cntshft“ gleich eins ist, inkrementiert mit der nächsten steigenden Taktflake das Modul „cnt_AD5544“ den Vektor „cntsdi“, ausgehend vom Anfangswert „00000“ um eins bis der Wert „10001“ erreicht ist. Gleichzeitig wird durch das Modul

„sr_AD5544“ das Signal „JP_SDI“ mit dem jeweiligen Bit des Vektors „daten55“ aktualisiert. Dadurch entspricht das Signal „JP_SDI“ immer, wenn der Vektor „clkcnt“ den Wert „00“ annimmt, d.h. mit jeder fallenden Flanke des erzeugten 12,5 MHz Taktes, dem nächsten Bit des Vektors „daten55“.

Vom Zustand „DAC1_2“ wird, sofern der Vektor „cntsdi“ nicht gleich „10001“ ist, mit der nächsten steigenden Taktflanke in den Zustand „DAC1_1“ gewechselt. Ist der Vektor „cntsdi“ gleich „10001“, wechselt die Zustandsmaschine mit der nächsten steigenden Taktflanke vom Zustand „DAC1_2“ in den Zustand „DAC1_3“. Wenn das Signal „cntshft“ gleich eins ist, obwohl der Vektor „cntsdi“ gleich „10001“ ist, setzt die Komponente „cnt_AD5544“ den Vektor „cntshf“ mit der nächsten steigenden Taktflanke gleich null, sodass er wieder gleich seinem Anfangswert ist. Dies passiert also gleichzeitig mit dem Wechsel vom Zustand „DAC1_2“ in den Zustand „DAC1_3“. Wenn sich die Zustandsmaschine im Zustand „DAC1_3“ befindet, sind 18 steigende Taktflanken des 12,5 MHz Taktes generiert worden. Somit hat das Signal „JP_SDI“ jedem Bit des Vektors „daten55“ entsprochen.

Vom Zustand „DAC1_3“ wird der Zustand „DAC1_4“ mit der nächsten und von diesem der Zustand „DAC1_5“ mit der übernächsten steigenden Taktflanken erreicht. In den beiden zuletzt genannten Zuständen wird das für die Ansteuerung erforderliche Signal „JP_LDAC1“ high gesetzt. Der Zustand „DAC1_3“ ist notwendig, um eine gewisse benötigte Zeit zwischen der letzten fallenden Flanke des 12,5 MHz Taktes und dem Setzen des Signals „JP_LDAC1“ auf einen Low-Pegel verstreichen zu lassen. Aus dem Zustand „DAC1_5“ wird mit der nächsten steigenden Taktflanke der hierarchische Zustand „DAC1“ verlassen und in den Anfangszustand „s0“ gewechselt.

4.6.2 Simulation

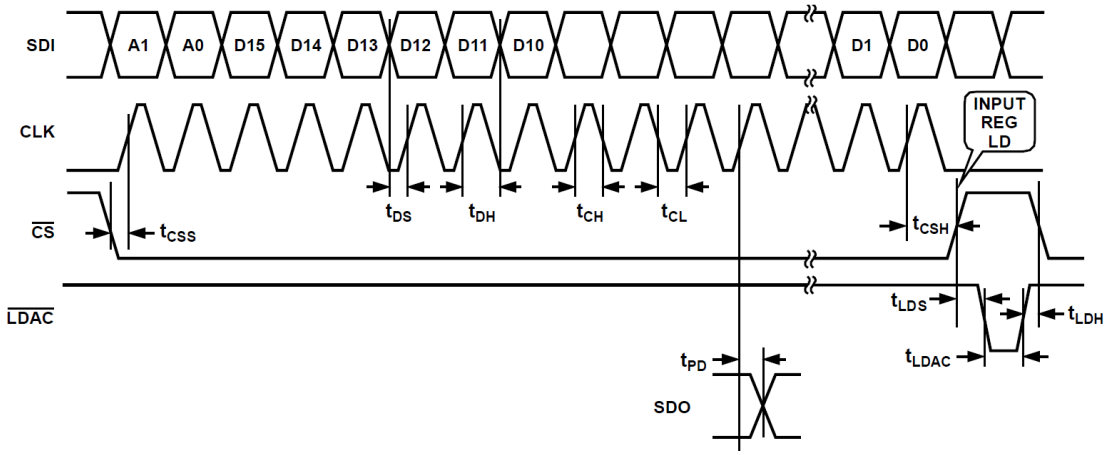

Abbildung 25: Zeitablaufdiagramm der Ansteuerung eines DACs vom Typ „AD5544“ [1]

In Abbildung 25 ist zu sehen, wie die Signale, die an den Pins eines DACs vom Typ „AD5544“ anliegen, für eine erfolgreiche Ansteuerung verlaufen müssen. Zusammen mit zusätzlichen Informationen des DAC Datenblatts wird der Verlauf erklärt. Zuerst muss das am „$\overline{\text{CS}}$“-Pin anliegende Signal low gesetzt werden. Danach sollte am „CLK“-Pin ein maximal 20 MHz schnelles Taktsignal anliegen. Das erste Bit, was am „SDI“-Pin anliegt, muss mindestens 20 ns vor der ersten steigenden Taktflanke anliegen. Das am „SDI“-Pin anliegende Signal wird mit der steigenden Flanke des Taktsignals übernommen, weswegen es mit der fallenden Flanke des Signals gesetzt werden muss. Nach 18 steigenden und fallenden Flanken, muss das Taktsignal wieder konstant gleich null sein und das Signal, welches am „$\overline{\text{CS}}$“-Pin anliegt, muss wieder high werden. Dies darf frühestens 25 ns nach der letzten steigenden Taktflanke geschehen. Nachdem dieses Signal wieder high ist, müssen mindestens 5 ns vergehen, bis das am „$\overline{\text{LDAC}}$“-Pin anliegende Signal für mindestens 25 ns low und anschließend wieder high ist. Diese Sequenz führt zu einer erfolgreichen Ansteuerung des DACs. [1]

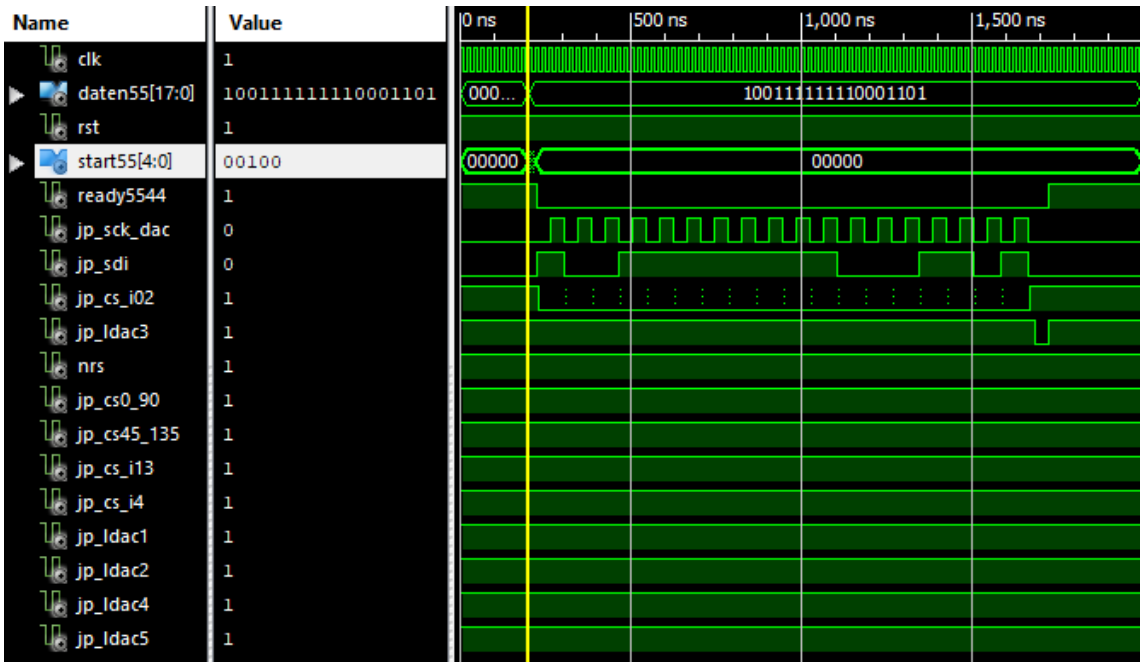

Abbildung 26: Simulation der Komponente „AD5544"

In Abbildung 26 ist eine Simulation der Komponente „AD5544“ zu sehen. Nachdem der Vektor „start55“ für einen Takt gleich „00100“ ist, beginnt die Ansteuerung des mit diesem Signal ausgewählten DACs vom Typ „AD5544“. Als erstes nimmt das Signal für den „$\overline{\text{CS}}$“-Pin des ausgewählten DACs einen Low-Pegel und das Signal „JP_SDI“ den Pegel des MSBs des Vektors „daten55“ an. Gleichzeitig wird das Signal „ready5544“ gleich null gesetzt. Nach 40 ns liegt am Signal „JP_SCK_DAC“ für den „CLK“-Pin die erste steigende Taktflanke des folgenden 12,5 MHz Taktes an. Es lässt sich erkennen, dass mit jeder weiteren fallenden Flanke des „JP_SCK_DAC“-Signals das Signal „JP_SDI“ den Pegel des nächst minderwertigeren Bits des Vektors „daten55“ annimmt. 40 ns nach der letzten steigenden Taktflanke nimmt das Signal für den „$\overline{\text{CS}}$“-Pin des ausgewählten DACs wieder einen High-Pegel an. Nach weiteren 20 ns nimmt das Signal für den „$\overline{\text{LDAC}}$“-Pin des ausgewählten DACs für 40 ns einen Low-Pegel an. Danach ist das Signal „ready5544“ wieder high.

Der Simulationsverlauf entspricht somit dem für eine erfolgreiche Ansteuerung erforderlichen Verlauf.

4.7 Die Komponente „AD7980“

4.7.1 Funktionsweise

Die Komponente „AD7980“ dient der Auslesung der ADCs und beinhaltet hierfür zwei Modi. Bei einem Modus erfolgt eine einmalige Auslesung, während beim alternativen Modus eine kontinuierliche Auslesung der ADCs erfolgt. Die Pins von je zwei ADCs, die die Umwandlung einer an einem ADC anliegenden Spannung in einen Digitalwert steuern, sind immer mit demselben Pin des Mojoboards verbunden. Dasselbe gilt auch für die Pins des Taktes zweier ADCs. Die Ansteuerung der ADCs erfolgt demnach immer paarweise. Aus diesem Grund ist es sinnvoll, dass immer entweder zwei, vier, sechs oder acht ADCs ausgelesen werden. Im Modus der einmaligen Auslesung werden die ausgelesenen Daten anschließend mithilfe der Komponente „Datensendung“ an den PC gesendet.

Im kontinuierlichen Modus beginnt nach Vollendung eines Auslesevorganges direkt der nächste. In diesem Modus werden die ausgelesenen Daten nicht an den PC gesendet, da für die Applikation eine Vorverarbeitung der Daten im FPGA vorgesehen ist. Außerdem werden in diesem Modus alle ADCs ausgelesen.

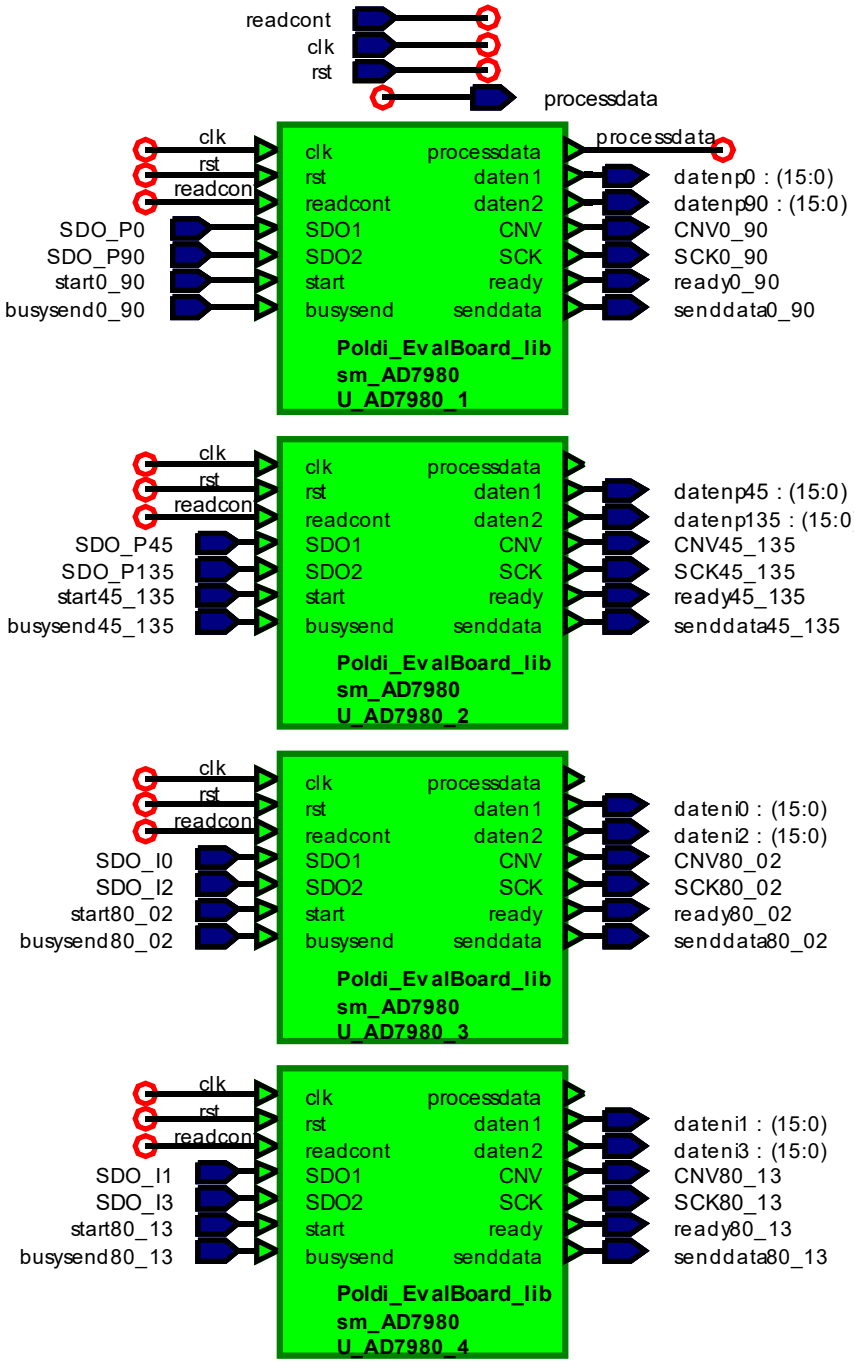

Abbildung 27: Struktur der Komponente „AD7980"

Abbildung 27 zeigt die Struktur der Komponente „AD7980“. Zu sehen ist, dass diese Komponente vier Instanzen des Moduls „sm_AD7980“ nutzt. Jede der vier Instanzen ist für die Auslesung von zwei ADCs zuständig.

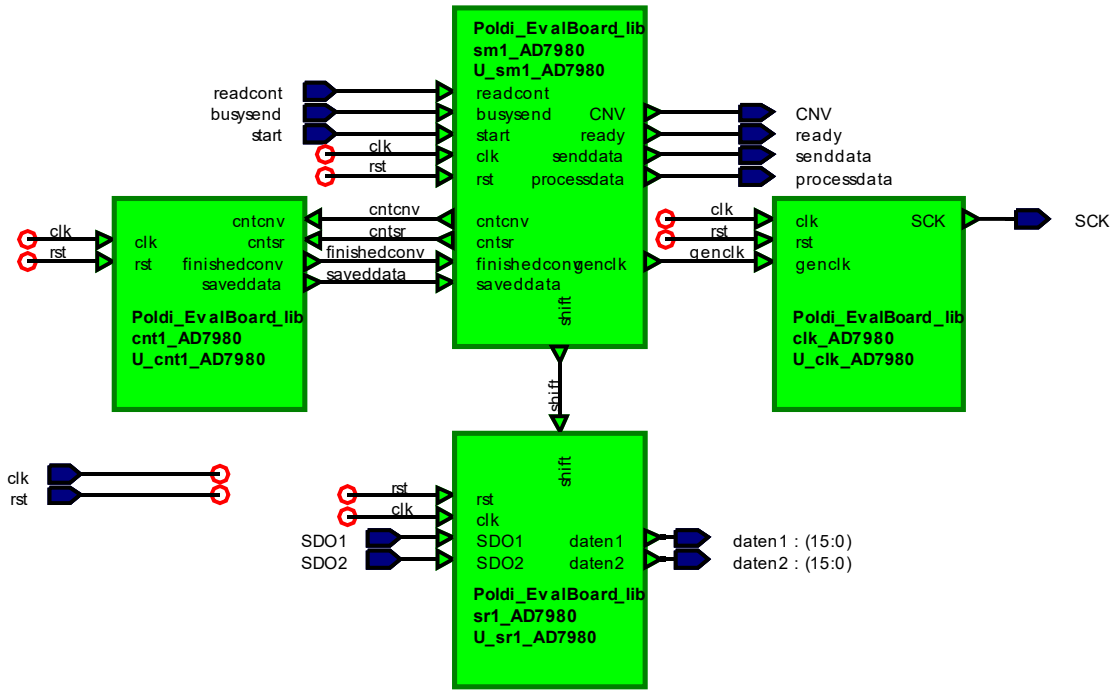

Abbildung 28: Struktur der Komponente „sm_AD7980"

In Abbildung 28 ist die Struktur der Komponente „sm_AD7980“ zu sehen. Diese Komponente steuert die zwei am Anfang dieses Kapitels erwähnten Auslesungsmodi für zwei ADCs und besteht aus vier Komponenten, wobei „cnt1_AD7980“ einem Zähler, „sm1_AD7980“ einer Zustandsmaschine, „clk_AD7980“ einem Taktgenerator und „sr1_AD7980“ einem Schieberegister entspricht.

Eingangssignale der Komponente „sm_AD7980“:

- Mit „**readcont**“ kann die kontinuierliche Auslesung der ADCs gestartet und gestoppt werden. Da bei der kontinuierlichen Auslesung alle ADCs zeitgleich ausgelesen werden sollen, wird, wie in Abbildung 27 zu sehen ist, „readcont“ allen vier Instanzen der Komponente „sm_AD7980“ zugeführt. Die kontinuierliche Auslesung bleibt so lange, wie dieses Signal high ist, aktiviert.
- „**busysend**“ zeigt an, ob die Komponente „Datensendung“ damit beschäftigt ist, die ausgelesenen Daten der Komponente „sm_AD7980“ an den PC zu senden.
- Wenn „**start**“ high ist, sollen die zwei ADCs ausgelesen und die ausgelesenen Daten anschließend mithilfe der Komponente „Datensendung“ an den PC gesendet werden. Dieses Signal sollte wieder low werden, sobald das Signal „ready“ dieser Komponente gleich null ist.

- „**SDO1**“ und „**SDO2**“ werden von den „SDO“-Pins zweier ADCs getrieben, an denen jeweils die Daten eines ADCs anliegen.

Ausgangssignale der Komponente „sm_AD7980“:

- „**CNV**“ treibt die „CNV“-Pins von jeweils zwei ADCs.
- „**SCK**“ treibt die Takteingänge „SCK“ von jeweils zwei ADCs.
- „**ready**“ zeigt an, ob die Komponente bereit ist, einen neuen Auslesevorgang mit anschließender Sendung der ausgelesenen Daten an den PC durch die Komponente „Datensendung“ zu starten.
- „**senddata**“ signalisiert der Komponente „Datensendung“, dass sie die Daten von zwei ausgelesenen ADCs an den PC senden soll. Dabei beinhaltet der 16-Bit Vektor „**daten1**“ die Daten des ADCs, der mit dem Eingangssignal „SDO1“ verbunden ist und der 16-Bit-Vektor „**daten2**“ die Daten des ADCs, der mit dem Eingangssignal „SDO2“ verbunden ist.
- „**processdata**“ zeigt an, dass ein Auslesevorgang im Modus der kontinuierlichen Auslesung abgeschlossen worden ist. Da die vier genutzten Instanzen des Moduls „sm_AD7980“ in diesem Modus gleichzeitig arbeiten, wird nur das Ausgangssignal einer der Instanzen mit dem Signal „processdata“ in Abbildung 27 verknüpft.

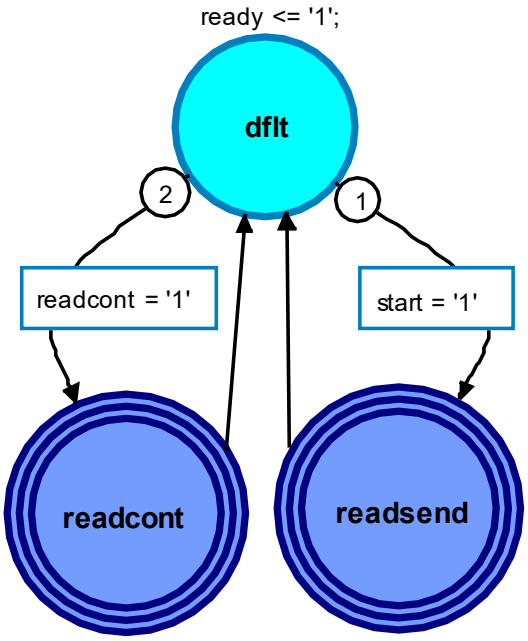

Abbildung 29: Zustandsmaschine „sm1_AD7980“

Abbildung 29 zeigt das Zustandsdiagramm der Komponente „sm1_AD7980“. Nur im Anfangszustand „dflt“ wird das Signal „ready“ high gesetzt. Wenn das Signal „start“ in diesem Zustand gleich eins ist, wechselt die Zustandsmaschine mit der nächsten steigenden Taktflanke in den hierarchischen Zustand „readsend“. Wenn das Signal „readcont“ gleich eins ist, wechselt sie vom Anfangszustand „dflt“ mit der nächsten steigenden Taktflanke in den hierarchischen Zustand „readcont“.

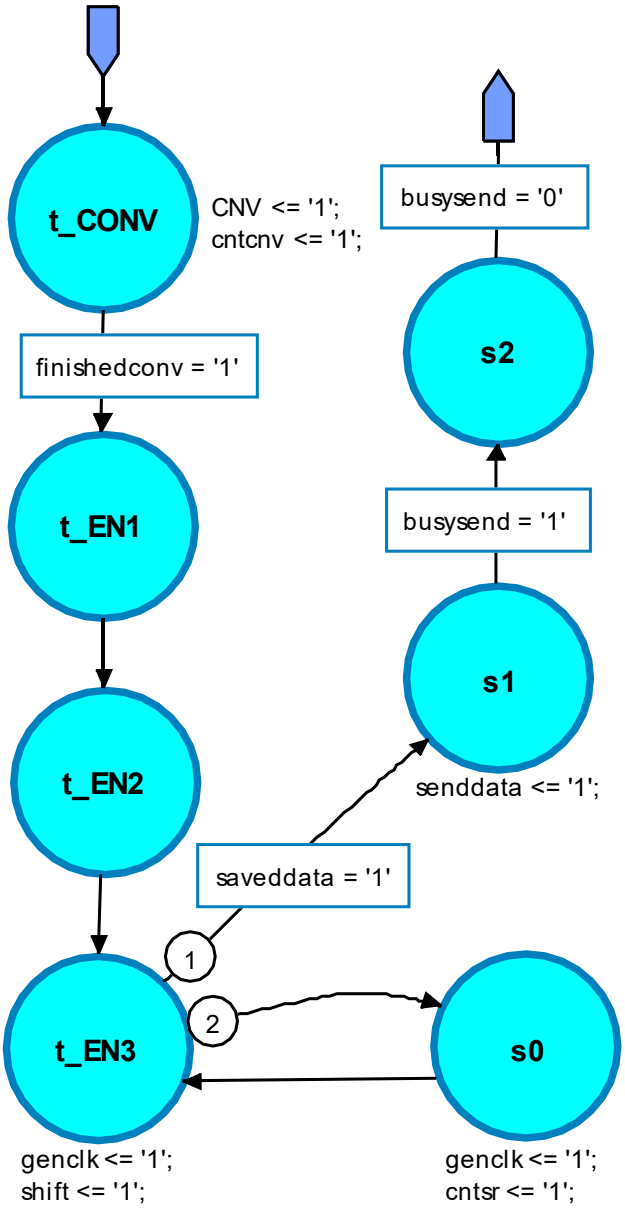

Abbildung 30: hierarchischer Zustand „readsend“ der Komponente „sm1_AD7980"

Abbildung 30 zeigt den hierarchischen Zustand „readsend“ der Komponente „sm1_AD7890“, welche im Zusammenspiel mit den anderen Modulen der Komponente „sm_AD7980“ zwei ADCs ausliest und die ausgelesenen Daten mithilfe der Komponente „Datensendung“ an den PC sendet. Im Zustand „t_CONV“ wird das für die Ansteuerung

der ADCs notwendige Signal „CNV“ high gesetzt. Außerdem wird das Signal „cntcnv“ gleich eins gesetzt. Dadurch zählt die Komponente „cnt1_AD7980“ ab diesem Zeitpunkt die steigenden Taktflanken mit und setzt mit der vierzigsten steigenden Taktflanke das Signal „finishedconv“ gleich eins. Wenn dies der Fall ist, wechselt die Zustandsmaschine mit der nächsten steigenden Taktflanke in den Zustand „t_EN1“, in welchem das Signal „CNV“ nicht mehr gleich eins ist.

Im Zustand „t_EN1“ sowie im nächsten Zustand „t_EN2“, der mit der nächsten steigenden Taktflanke erreicht wird, wird kein Signal gesetzt. Dies ist für die erfolgreiche Auslesung der ADCs notwendig. Von „t_EN2“ wird in den Zustand „t_EN3“ mit der nächsten steigenden Taktflanke übergegangen. Dort werden die Signale „genclk“ und „shift“ gleich eins gesetzt. Solange das Eingangssignal „saveddata“ nicht gleich eins ist, wird von „t_EN3“ immer mit einer steigenden Taktflanke in den Zustand „s0“ übergegangen. In diesem Zustand werden die Signale „genclk“ und „cntsr“ high gesetzt und es wird mit der nächsten steigenden Taktflanke in den Zustand „t_EN3“ gewechselt.

Wenn das Signal „genclk“ high ist, erzeugt die Komponente „clk_AD7980“ mit dem Ausgangssignal „SCK“ einen 25 MHz Takt für den Takteingang der ADCs. Wenn das Signal „shift“ gleich eins ist, speichert die Komponente „sr1_AD7980“ mit der nächsten steigenden Taktflanke den Wert, den das Signal „SDO1“ zu diesem Zeitpunkt hat, im Vektor „daten1“ und den Wert, den das Signal „SDO2“ zu diesem Zeitpunkt hat, im Vektor „daten2“ ab. Die Werte werden in die LSBs der Vektoren „daten1“ und „daten2“ geschoben, wobei zeitgleich die alten 15 LSBs der Vektoren jeweils ein Bit an Wertigkeit gewinnen, sodass nach Abspeichern aller 16 Bits die zuerst abgespeicherten Werte den MSBs und die zuletzt abgespeicherten Werte den LSBs der Vektoren „daten1“ und „daten2“ entsprechen. Die Abspeicherung geschieht zeitgleich mit den steigenden Taktflanken des 25 MHz Taktes.
Immer, wenn „cntsr“ gleich eins ist, inkrementiert die Komponente „cnt1_AD7980“ mit der nächsten steigenden Taktflanke einen internen Vektor um eins. Ist das dezimale Äquivalent dieses Vektors gleich 15, so wird das Signal „saveddata“ gleich eins gesetzt und die Zustandsmaschine wechselt mit der nächsten steigenden Taktflanke vom Zustand „t_EN3“ in den Zustand „s1“. Wird dieser Zustand erreicht, wurden alle 16 Bits der beiden ADCs empfangen und das Signal „senddata“ wird in Folge high gesetzt. Dadurch sendet die Komponente „Datensendung“ die in den Vektoren „daten1“ und „daten2“

gespeicherten Daten an den PC. Ist die Komponente „Datensendung“ mit der Versendung beschäftigt, wird mit der nächsten steigenden Taktflanke in den Zustand „s2“ übergegangen. Wenn die Daten an den PC gesendet worden sind, wird mit der nächsten steigenden Taktflanke der hierarchische Zustand „sm1_ad7980“ verlassen und in den Anfangszustand „dflt“ der Komponente „sm1_AD7980“ gewechselt.

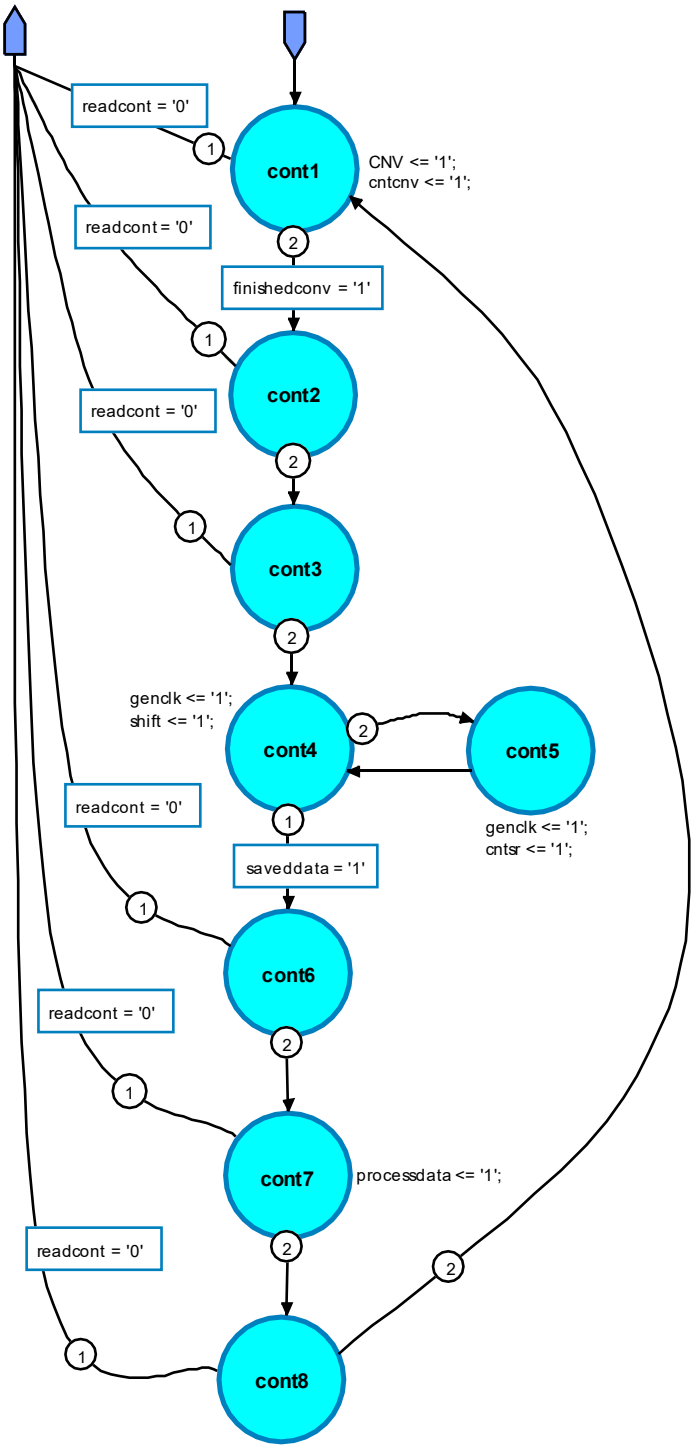

Abbildung 31: hierarchischer Zustand „readcont“ der Komponente „sm1_AD7980"

In Abbildung 31 ist der hierarchische Zustand „readcont“ der Komponente „sm1_AD7980“ zu sehen. Dieser sorgt mit den anderen Modulen der Komponente „sm_AD7980“ für die kontinuierliche Auslesung der ADCs. Grundsätzlich funktioniert dieser hierarchische Zustand genauso wie der zuvor erklärte hierarchische Zustand „readsend“, weshalb in diesem Abschnitt nur die Unterschiede der beiden hierarchischen Zustände aufgeführt werden.

Die letzten zwei Zustände des hierarchischen Zustands „readsend“ der Komponente „sm1_AD7980“ werden in diesem hierarchischen Zustand durch die drei Zustände „cont6“, „cont7“ und „cont8“ ersetzt. Durch diese drei Zustände wird u.a. erreicht, dass zwischen zwei Auslesevorgängen eine gewisse Zeit verstreicht. Dies ist für eine erfolgreiche Ansteuerung der ADCs notwendig. Im Zustand „cont7“ wird zusätzlich das Signal „processdata“ gleich eins gesetzt, um zu signalisieren, dass ein Auslesevorgang abgeschlossen ist und die in den Vektoren „daten1“ und „daten2“ enthaltenen Daten zur weiteren Signalverarbeitung verwendet werden können. Von allen Zuständen außer den Zuständen „cont4“ und „cont5“ wird mit der nächsten steigenden Taktflanke der hierachische Zustand „readcont“ verlassen und in den Anfangszustand „dflt“ der Komponente „sm1_AD7980“ gewechselt, sobald die kontinuierliche Auslesung gestoppt wird. Ansonsten wird der hierarchische Zustand nicht verlassen und es findet zyklisch die kontinuierliche Auslesung der ADCs statt. Die einzige Ausnahme sind die Zustände „cont4“ und „cont5“, bei denen der hierarchische Zustand „readcont“ nicht verlassen wird, wenn die kontinuierliche Auslesung gestoppt wird. Das ist darin begründet, dass die Vektoren „daten1“ und „daten2“ überschrieben werden, wenn sich die Zustandsmaschine in diesen Zuständen befindet und die Vektoren „daten1“ und „daten2“ andernfalls keine gültigen Daten enthalten würden.

4.7.2 Simulation

In den Abbildungen der Simulationen werden nur die für die Funktionsweise des jeweils gezeigten Ablaufs relevanten Signale der Komponente „AD7980“ dargestellt.

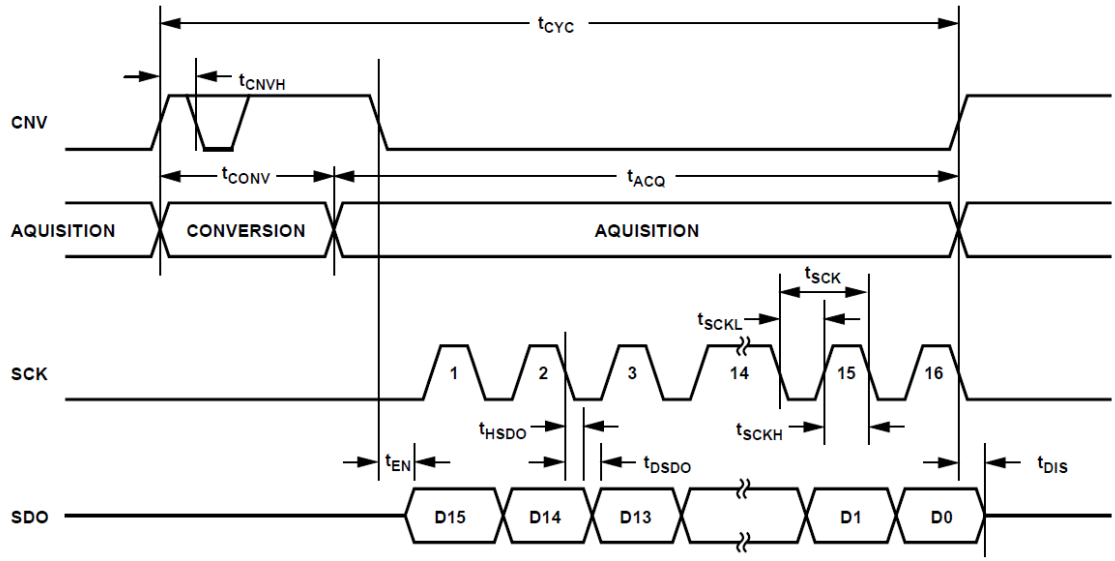

Abbildung 32: Ablauf eines Auslesevorganges eines ADCs vom Typ „AD7980“ [3]

Abbildung 32 zeigt, wie der Ablauf eines Auslesevorgang eines ADCs vom Typ „AD7980“ des in diesem Projekt genutzten Modus auszusehen hat. Der dargestellte Ablauf wird anhand zusätzlicher Informationen aus dem Datenblatt des ADCs erklärt. Als erstes muss das am „CNV“-Pin anliegende Signal für mindestens 800 ns high gesetzt werden. Danach muss es wieder low gesetzt werden und nach maximal weiteren 40 ns ist die Umwandlung der anliegenden Spannung in einen Digitalwert abgeschlossen. Anschließend muss am „SCK“-Pin ein Takt anliegen, mit dessen steigenden Flanken die am „SDO“-Pin anliegenden Daten abgespeichert werden sollten. Das ist darin begründet, dass die Bits, die am „SDO“-Pin anliegen, mit der Ausnahme des MSBs, mit den fallenden Taktflanken gesetzt werden. Das MSB hingegen liegt schon vor der ersten steigenden Taktflanke am „SDO“-Pin an. Eine Periode des „SCK“-Taktes darf maximal 22 ms lang sein. Nach der 16. fallenden Taktflanke sollte das am „SCK“-Pin anliegende Signal wieder auf null gehalten werden und es müssen mindestens 20 ns vergehen, bis der nächste Auslesevorgang durch die Aktivierung des „CNV“-Signals auf einen High-Pegel beginnen kann. [3]

In den Abbildungen der Simulationen werden nur die für die Funktionsweise des jeweils gezeigten Ablaufs relevanten Signale der Komponente „AD7980“ dargestellt.

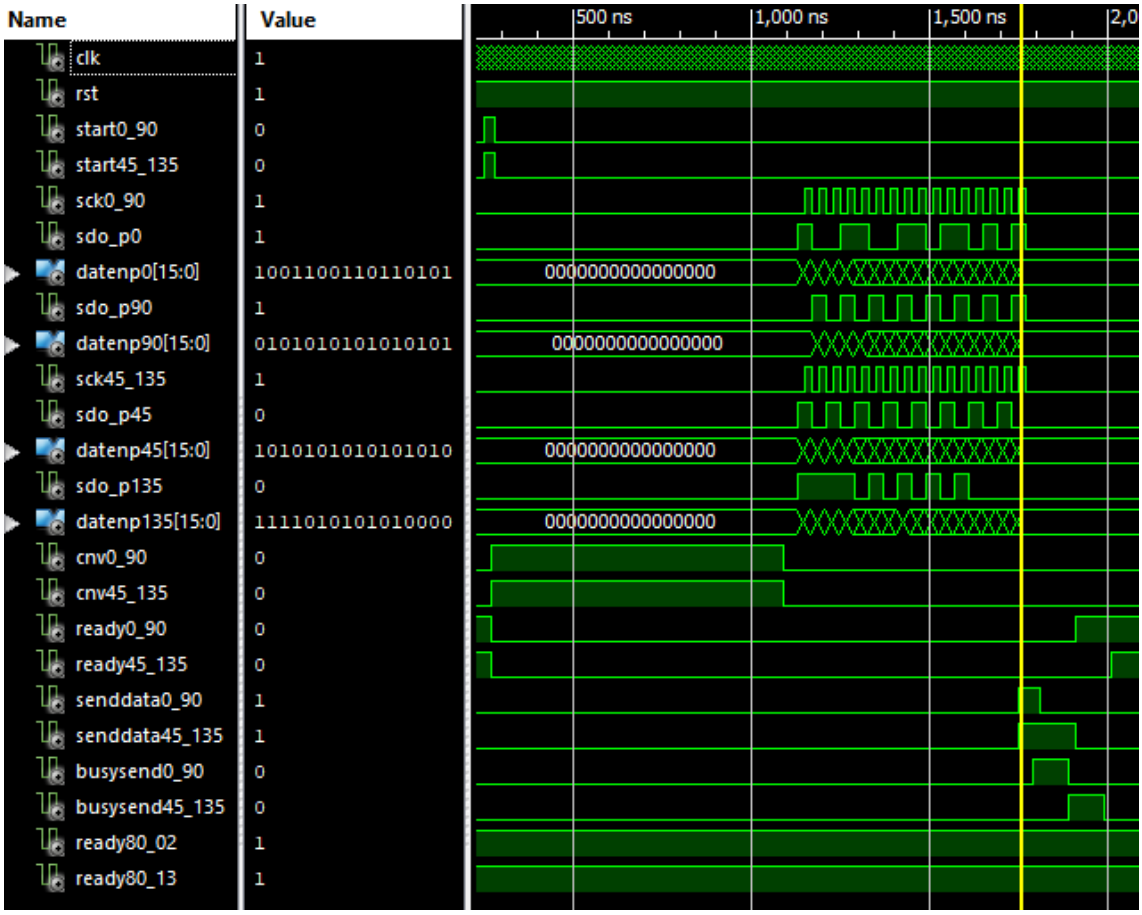

Abbildung 33: Simulation 1 der Komponente „AD7980"

Abbildung 33 stellt eine Simulation der Komponente „AD7980“ bei einmaliger Auslesung dar. Zu sehen ist, dass als erstes die Signale „start0_90“ und „start45_135“ den Wert eins annehmen. Es sind also vier ADCs ausgewählt, die ausgelesen und deren ausgelesenen Daten anschließend an den PC gesendet werden sollen. Nachdem die beiden Signale einen High-Pegel annehmen, erhalten die Signale „ready0_90“ und „ready45_135“ den Wert null und die für die Ansteuerung notwendigen Signale „cnv_0_90“ und „cnv_45_135“ sind für 820ms gleich eins. Danach verstreichen 60 ms, bis die Signale „sck0_90“ und „sck45_135“ zum ersten Mal high sind und fortan einen 25 MHz Takt ausgeben. Mit den steigenden Taktflanken dieser Takte werden die Werte der „sdo“-Signale in den Vektoren „daten“ abgespeichert. Kapitel 4.7.1 kann entnommen werden, welche Signalwerte in welchem Vektor abgespeichert werden.

Zum Zeitpunkt des gelben Markers sind alle Bits abgespeichert worden. Die Werte unter „Value“ oben links beziehen sich auf den Zeitpunkt dieses Markers. Es ist zu erkennen,

dass die Vektoren die Daten korrekt abgespeichert haben. Nach der Abspeicherung nehmen die Signale „senddata0_90“ und „senddata45_135“ einen High-Pegel an. Danach wird das Signal „busysend0_90“ high und das Signal „senddata0_90“ wieder low. Nachdem das Signal „busysend0_90“ wieder low ist, nimmt das Signal „ready0_90“ den Wert eins an. Danach wird das Signal „busysend45_135“ high und das Signal „senddata45_135“ low. Nachdem das Signal „busysend45_135“ wieder low ist, nimmt das Signal „ready45_135“ einen High-Pegel an.

Abbildung 34: Simulation 2 der Komponente „AD7980"

Abbildung 34 zeigt eine Simulation der Komponente „AD7980“ bei kontinuierlicher Auslesung aller ADCs. Wie zu sehen ist, nimmt das Signal „readcont“ einen High-Pegel an. Danach sind die „ready“-Signale gleich eins und die Auslesung aller ADCs beginnt. Die Signale verhalten sich genauso wie in Abbildung 33 dargestellt, bis die Daten der ADCs abgespeichert sind. Da alle ADCs ausgelesen werden, sind jedoch alle Signale, die zur Ansteuerung und zur Abspeicherung aller ADCs notwendig sind, beteiligt. Nach der Abspeicherung ist in Abbildung 34 das Signal „processdata“ für einen Takt gleich eins.

Zwischen der letzten fallenden Taktflanke der „SCK“-Taktsignale und der Initialisierung eines neuen Auslesevorganges durch das Setzen der „CNV“-Signale vergehen 40 ns. Nachdem das Signal „readcont“ den Wert null annimmt, nehmen die „CNV“-Signale ebenfalls den Wert null und die „ready“-Signale den Wert eins an. Somit findet keine Auslesung mehr statt.

Die meisten Zeiten, die auf der Seite 50 genannt sind, sind in diesem VHDL-Design jeweils bewusst um 20 ns vergrößert worden, um einen Puffer zu den Mindestzeiten einzuräumen. So wird sichergestellt, dass die einzuhaltenden Zeiten definitiv befolgt werden. Die Periode der „SCK“-Takte beträgt hierbei 40 ns anstatt 22ns. Diese Wahl ist durch die Periode des Systemtaktes des Mojoboards begründet, die 20 ns beträgt und somit ungeeignet ist. Der nächste generierbare Takt ist ein 25 MHz Takt mit einer Taktperiode von 40 ns.

Die Zeit für eine vollständige Auslesung ergibt sich aus der Aktivierung der „CNV“-Signale für 820 ns, der anschließenden Wartezeit von 60 ns bis zur ersten steigenden Taktflanke der 25 MHz Takte, den 15,5 Perioden des 25 MHz Taktes und dem anschließenden Warten von 40 ns bis zum Start eines neuen Auslesevorganges zu 1540 ns. Dadurch lässt sich die folgende Abtastrate berechnen:

$$f_{read} = \frac{1}{t_{read}} = \frac{1}{1540\ ns} = 649{,}351\ KSpS$$

Mit diesem VHDL-Design wird im Modus der kontinuierlichen Auslesung jeder ADC also ca. 649351 Mal pro Sekunde ausgelesen.

4.8 Die Komponente „Datensendung“

4.8.1 Funktionsweise

Mit dieser Komponente können die ausgelesenen Daten der ADCs mithilfe der Komponente „avr_interface“ an den PC gesendet werden.

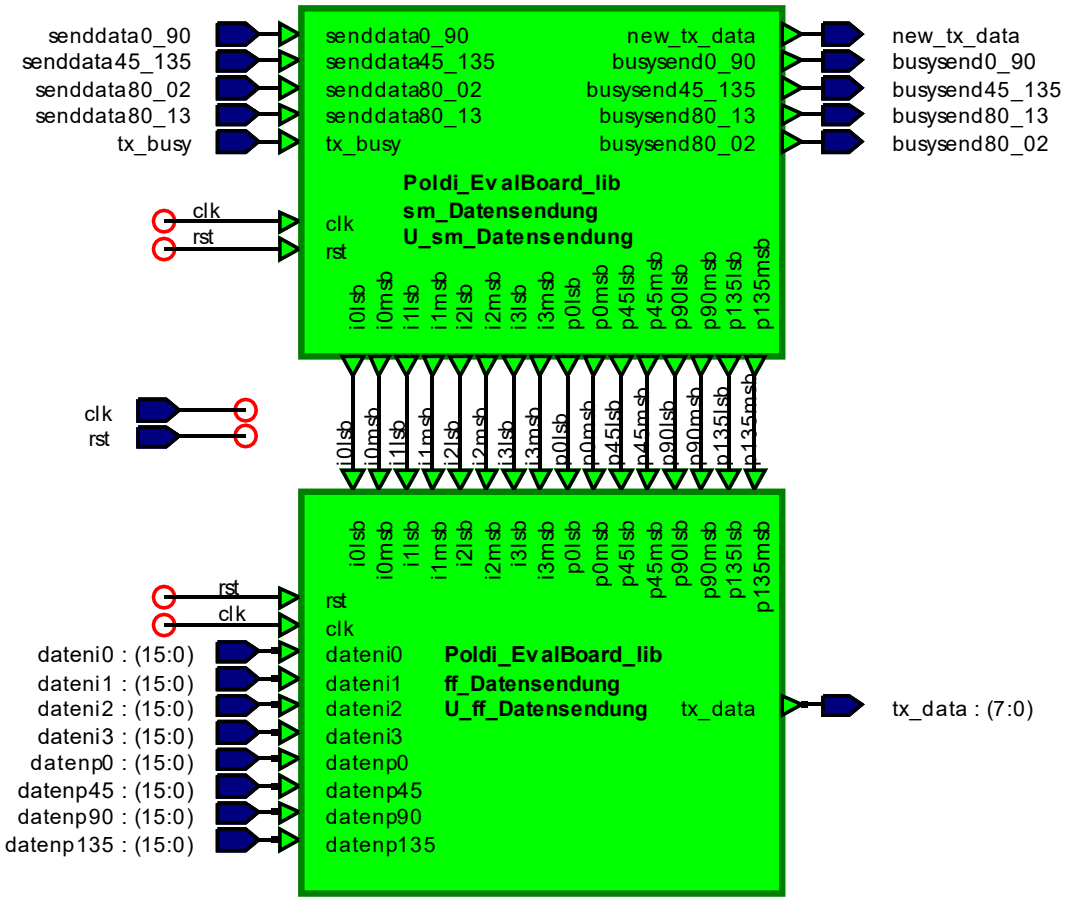

Abbildung 35: Struktur der Komponente „Datensendung"

In Abbildung 35 ist die Struktur der Komponente „Datensendung“ zu sehen. Diese Komponente enthält das Modul „sm_Datensendung“, welches einer Zustandsmaschine entspricht, und das Modul „ff_Datensendung“.

Eingangssignale der Komponente „Datensendung“:

- „**tx_busy**“ zeigt an, ob die Komponente „avr_interface“ damit beschäftigt ist, Daten an den Mikrocontroller zu senden.
- Die „**senddata**“-Signale, geben jeweils an, ob Daten von zwei ausgelesenen ADCs an den PC gesendet werden sollen. Dabei sind jeweils zwei

„**daten**“-Vektoren einem „senddata“-Signal zuzuordnen. Diese Vektoren beinhalten die Daten von zwei ausgelesenen ADCs. Abbildung 27 des Kapitels 4.7.1 kann dabei entnommen werden, welche „daten“-Vektoren zu welchem „senddata“-Signal gehören. Ein „senddata“-Signal sollte, wenn das dazugehörige Signal „busysend“ high ist, wieder auf null gesetzt werden.

Ausgangssignale der Komponente „Datensendung“:

- Wenn „**new_tx_data**“ gleich eins gesetzt wird, während „tx_busy“ gleich null ist, werden die in dem Vektor „**tx_data**“ enthaltenen Daten an den PC gesendet.
- Die „**busysend**“ Signale geben jeweils an, ob die Komponente mit der Versendung der Daten von zwei ADCs beschäftigt ist.

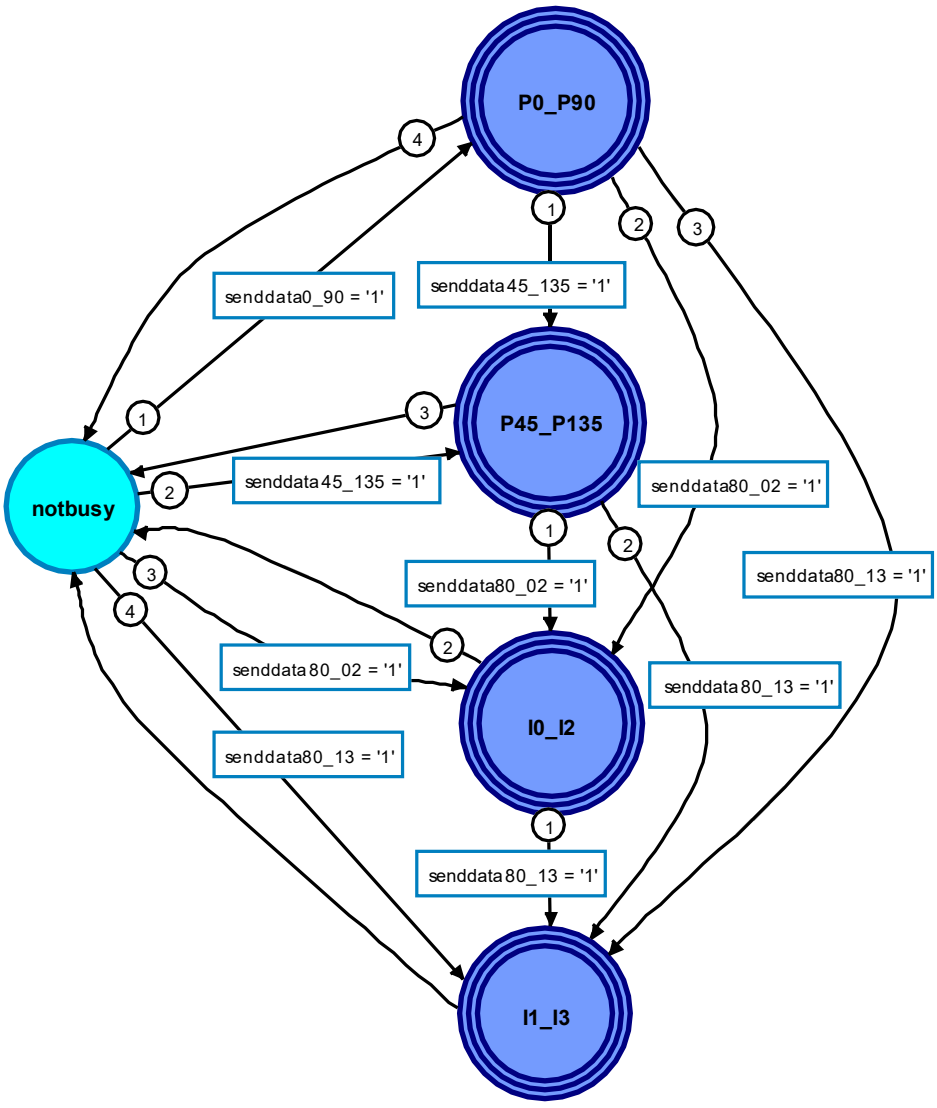

Abbildung 36: Zustandsmaschine „sm_Datensendung“

Abbildung 36 zeigt das Zustandsdiagramm der Komponente „sm_Datensendung“. Einer der vier hierarchischen Zustände der Abbildung 36 sorgt mithilfe der Komponente „ff_Datensendung“ dafür, dass die Daten von zwei ausgelesenen ADCs an den PC gesendet werden.

Wenn die Komponente „AD7980“ Daten von mehr als zwei ADCs an den PC senden möchte, setzt sie die „senddata“-Signale, die den zu sendenden Daten zuzuordnen sind, gleichzeitig gleich eins.

Vom Anfangszustand „notbusy“ wird mit der nächsten steigenden Taktflanke in den hierarchischen Zustand „P0_P90“ gewechselt, wenn das Signal „senddata0_90“ gleich eins ist. Wenn das Signal nicht gleich eins ist, dafür aber das Signal „senddata45_135“, wird mit der nächsten steigenden Taktflanke in den hierarchischen Zustand „P45_P135“ gewechselt. Wenn weder das Signal „senddata0_90“, noch das Signal „senddata45_135“ gleich eins ist, aber das Signal „senddata80_02“, geht die Zustandsmaschine mit der nächsten steigenden Taktflanke in den hierarchischen Zustand „I0_I2“ über. Wenn keines der Signale „senddata0_90“, „senddata45_135“ oder „senddata80_02“ gleich eins ist, dafür aber das Signal „senddata80_13“, wechselt die Zustandsmaschine mit der nächsten steigenden Taktflanke in den hierachischen Zustand „I1_I3“.

Vom letzten Zustand des hierarchischen Zustands „P0_P90“ wird, wenn „tx_busy“ gleich null ist, mit der nächsten steigenden Taktflanke entweder in einen anderen hierarchischen Zustand oder in den Anfangszustand „notbusy“ gewechselt. Ist das Signal „senddata45_135“ high, geht die Zustandsmaschine in den hierarchischen Zustand „P45_P135“ über. Ist das Signal nicht gleich eins, aber das Signal „senddata80_02“, geht die Zustandsmaschine in den hierarchischen Zustand „I0_I2“ über. Ist weder das Signal „P45_P135“, noch das Signal „senddata80_02“ high, aber das Signal „senddata80_13“, geht die Zustandsmaschine in den hierarchischen Zustand „I1_I3“ über. Wenn die Signale „senddata45_135“, „senddata80_02“ und „senddata80_13“ alle gleich null sind, geht die Zustandsmaschine in den Anfangszustand „notbusy“ über.

Vom letzten Zustand des hierarchischen Zustands „P45_P135“ wird, wenn „tx_busy“ gleich null ist, mit der nächsten steigenden Taktflanke entweder in den Anfangszustand „notbusy“ oder in einen der beiden hierarchischen Zustände „I0_I2“ oder „I1_I3“ gewechselt. Wenn das Signal „senddata80_02“ high ist, geht die Zustandsmaschine in den

hierarchischen Zustand „I0_I2“ über. Ist dieses Signal low, dafür aber das Signal „senddata80_13“ high, geht die Zustandsmaschine in den hierarchischen Zustand „I1_I3“ über. Wenn die Signale „senddata80_02“ und „senddata80_13“ beide gleich null sind, ist der Nachfolgezustand der Anfangszustand „notbusy“.

Vom letzten Zustand des hierarchischen Zustands „I0_I2“ wird, wenn „tx_busy“ gleich null ist, mit der nächsten steigenden Taktflanke in den hierarchischen Zustand „I1_I3“ gewechselt, wenn das Signal „senddata80_13“ high ist. Ist dieses Signal low, wird mit der nächsten steigenden Taktflanke in den Anfangszustand „notbusy“ gewechselt.

Vom letzten Zustand des hierarchischen Zustands „I1_I3“ wird, wenn „tx_busy“ gleich null ist, mit der nächsten steigenden Taktflanke in den Anfangszustand „notbusy“ gewechselt.

Durch die beschriebenen Bedingungen haben die hierarchischen Zustände also verschiedene Prioritäten. Wenn alle „senddata“-Signale gleich eins sind, werden zuerst die Daten, für die der hierarchische Zustand „P0_P90“ zuständig ist, danach die Daten, für die der hierarchische Zustand „P45_P135“ zuständig ist, danach die Daten, für die der hierarchische Zustand „I0_I2“ zuständig ist, und als letztes die Daten, für die der hierarchische Zustand „I1_I3“ zuständig ist, an den PC gesendet. Wenn „senddata“-Signale nicht gleich eins sind, werden die hierarchischen Zustände, die mit diesen Signalen erreicht worden wären, übersprungen und nur die Daten der hierarchischen Zustände, die erreicht werden, entsprechend der genannten Reihenfolge an den PC gesendet.

In Tabelle 7 im Anhang ist zu erkennen, in welcher Reihenfolge die ausgelesenen Daten der ADCs an den PC gesendet werden.

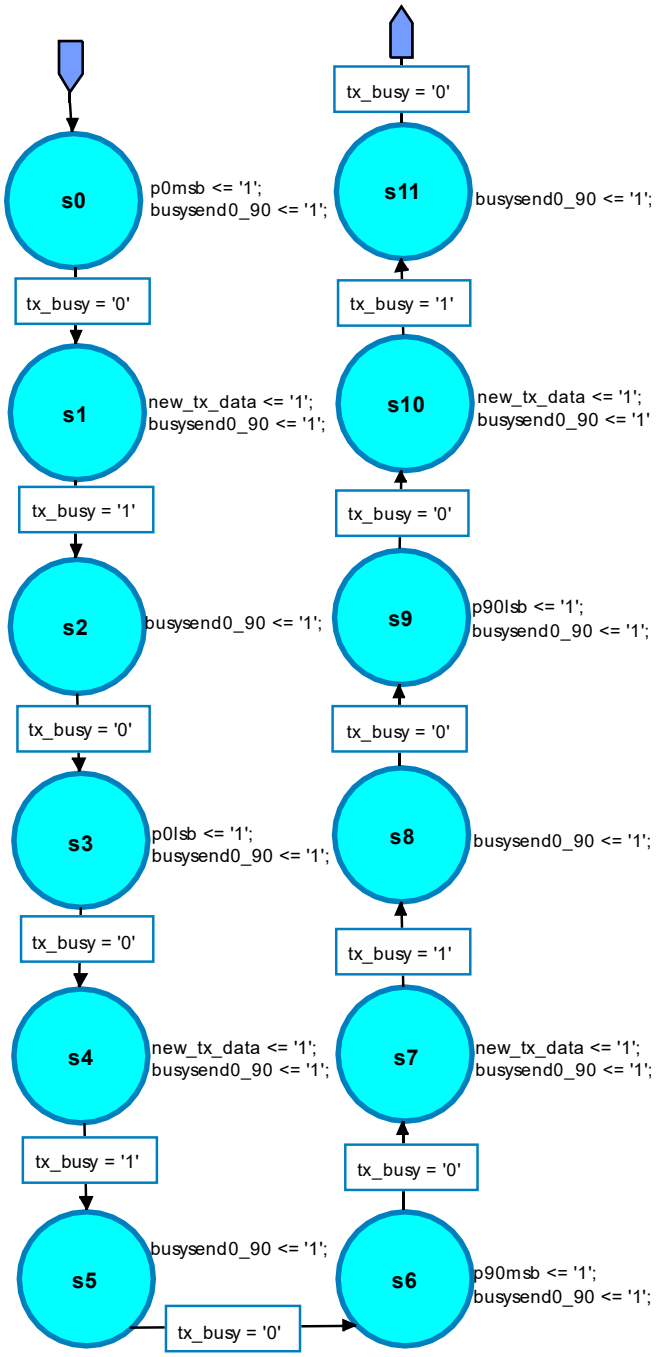

Abbildung 37: hierarchischer Zustand „P0_P90“ der Komponente „sm_Datensendung"

Abbildung 37 zeigt den hierarchischen Zustand „P0_P90“ der Komponente „sm_Datensendung“. Da die vier hierachischen Zustände gleich funktionieren, wird nur einer erklärt. In jedem Zustand dieses hierarchischen Zustands wird das Signal „busysend0_90“ gleich eins gesetzt. Im ersten Zustand „s0“ wird das Signal „p0msb“ gleich eins gesetzt. Dadurch setzt die Komponente „ff_Datensendung“ mit der nächsten steigenden Taktflanke die Daten des Vektors „tx_data“ gleich den acht MSBs des Vektors „datenp0“. Wenn die Komponente „avr_interface“ bereit ist, Daten an den Mikrocontroller zu senden, wechselt die

Zustandsmaschine mit der nächsten steigenden Taktflanke vom Zustand „s0“ in den Zustand „s1“, in welchem das Signal „new_tx_data“ high gesetzt wird. Dadurch beinhaltet der Vektor „tx_data“ die acht MSBs des Vektors „datenp0“, wenn das Signal „new_tx_data“ high gesetzt wird. Während die Komponente „avr_interface“ mit der Versendung der Daten beschäftigt ist, wechselt die Zustandsmaschine mit der nächsten steigenden Taktflanke in den Zustand „s2“. Von diesem wird, wenn die Komponente „avr_interface“ bereit ist, Daten an den Mikrocontroller zu senden, mit der nächsten steigenden Taktflanke in den Zustand „s3“ übergegangen.

Die erläuterten drei Übergangsbedingungen sowie die Funktionsweise der erläuterten Zustände wiederholen sich in den folgenden Zuständen in der Reihenfolge drei Mal, wobei in den Zuständen „s3“, „s6“ und „s9“ jedoch genau die Signale gesetzt werden, die dafür sorgen, dass der Vektor „tx_data“ die jeweils passenden weiteren zu sendenden Daten erhält. Der Nachfolgezustand des Zustands „s11“ wurde bereits im Rahmen der Beschreibung von Abbildung 36 erläutert. Durch diesen hierarchischen Zustand wird dafür gesorgt, dass zuerst die acht MSBs, danach die acht LSBs des Vektors „datenp0“, danach die acht MSBs und als letztes die acht LSBs des Vektors „datenp90“ an den PC gesendet werden.

Wie in Abbildung 27 dargestellt, ist der Vektor „datenp0“ mit dem Ausgangsvektor „daten1“ und der Vektor „datenp90“ mit dem Ausgangsvektor „daten2“ der ersten Komponente „sm_AD7980“ verbunden. Die Ausgangsvektoren der Komponente „AD7980“ sind mit den gleichnamigen Eingangsvektoren der Komponente „Datensendung“ verbunden. Mithilfe eines hierarchischen Zustands der Komponente „Datensendung“ werden also zuerst die Daten „daten1“ und dann die Daten „daten2“ einer Komponente „sm_AD7980“ an den PC gesendet.

4.8.2 Simulation

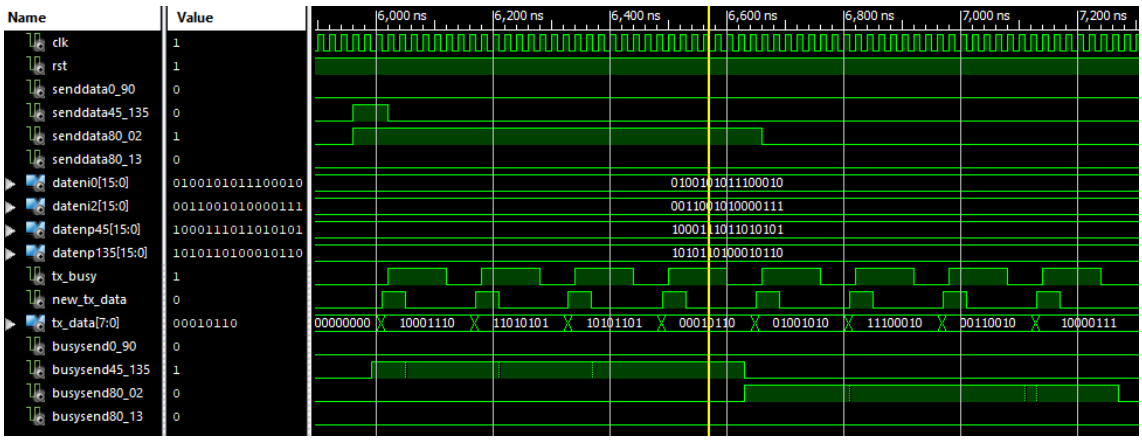

Abbildung 38: Simulation der Komponente „Datensendung"

Abbildung 38 zeigt eine Simulation der Komponente „Datensendung“. Diese Abbildung beinhaltet nur die für die Funktionsweise des gezeigten Ablaufs relevanten Signale. Wie zu sehen ist, werden die Eingangssignale „senddata45_135“ und „senddata80_02“ anfangs gleich eins gesetzt, was bedeutet, dass die Daten der Vektoren „datenp45“, „datenp135“, „dateni0“ und „dateni2“ an den PC gesendet werden sollen. Danach ist das Signal „busysend45_135“ gleich eins, bis die Daten der Vektoren „datenp45“ und „datenp135“ übertragen worden sind. Danach ist das Signal „busysend80_02“ gleich eins, bis die Daten der Vektoren „dateni0“ und „dateni2“ an den PC gesendet worden sind.

Es lässt sich erkennen, dass das Signal „tx_busy“ low wird, wenn das Signal „new_tx_data“ einen High-Pegel annimmt. Kurz nachdem das Signal „tx_busy“ high wird, nimmt das Signal „new_tx_data“ einen Low-Pegel an. Zudem wechselt der Vektor „tx_data“ immer seinen Inhalt, wenn das Signal „new_tx_data“ von einem Low- auf einen High-Pegel übergeht. Mit der ersten steigenden Flanke des Signals „new_tx_data“ entspricht der Vektor „tx_data“ den acht MSBs des Vektors „datenp45“. Mit der zweiten steigenden Flanke des Signals „new_tx_data“ entspricht der Vektor „tx_data“ den acht LSBs des Vektors „datenp45“. Mit der zweiten fallenden Flanke des Signals „tx_busy“ sind die Daten des Vektors „datenp45“ an den PC gesendet worden.

Wie zu sehen ist, werden als nächstes die Daten des Vektors „datenp135“, dann die des Vektors „dateni0“ und als letztes die des Vektors „dateni2“ übertragen, wobei immer zuerst die MSBs und dann die LSBs eines Vektors übertragen werden.

5 QT-Anwendung

5.1 Erstellung des GUIs

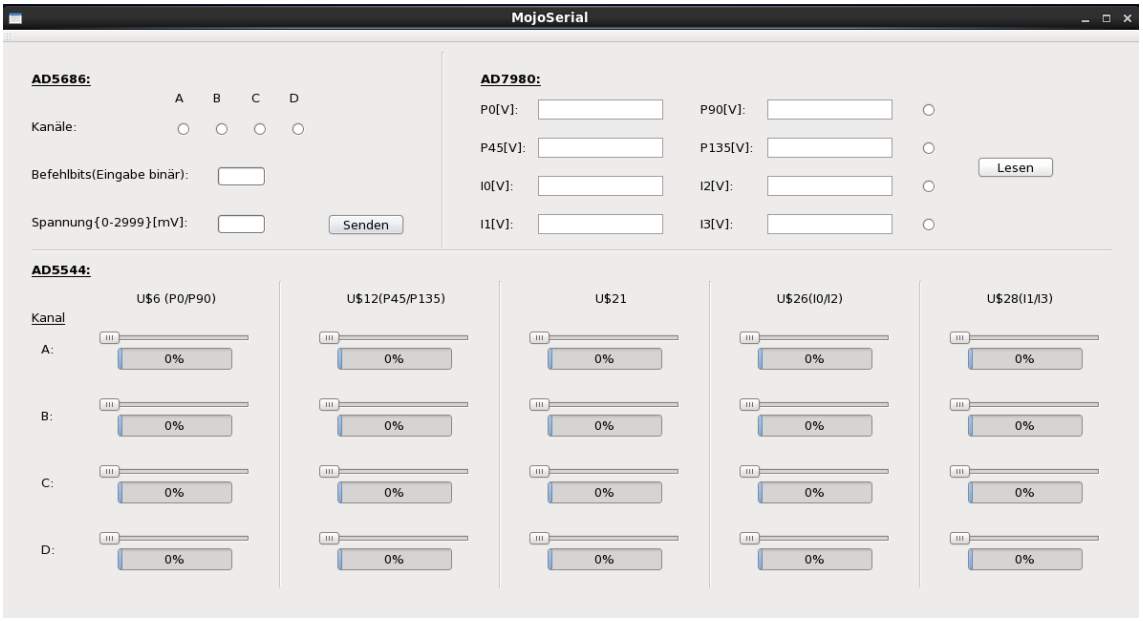

Abbildung 39: Hauptfenster des GUIs

Bei der Erstellung des GUIs der QT-Anwendung wurde mit dem in Abbildung 39 sichtbaren Hauptfenster begonnen. Durch Trennlinien wurden drei Felder erzeugt.

Das Feld oben links ist für die Ansteuerung des DACs vom Typ „AD5686“ vorgesehen worden. Dort wurde deshalb ein Label mit dem Text „AD5686“ platziert. Für die Auswahl der Kanäle des DACs „AD5686“ wurden vier „Radio Buttons“ verwendet. Über den einzelnen Buttons wurden Labels mit den Namen der DAC-Kanäle platziert. Ein Label mit dem Text „Kanäle“ wurde links neben den Buttons erstellt, um zu zeigen, dass die Buttons für die Auswahl der Kanäle genutzt werden sollen. Da auch mehrere Kanäle angesteuert werden können, müssen auch mehrere Buttons gleichzeitig aktiviert werden können. Die Häkchen in den Kästchen „auto-exclusive“ in den Eigenschaften der vier verwendeten Buttons wurden entfernt, da keine weiteren Buttons „Radio Button“ aktiviert werden können, wenn ein Button „radioButton“ mit der Eigenschaft „auto-exclusive“ aktiviert ist.

Zur Eingabe der Befehlsbits und der Spannung wurden zwei Eingabefelder „lineEdit“ eingefügt. Bei diesen Feldern wurde die maximal eingebbare Zeichenkettenlänge unter den Eigenschaften bei „maxLength“ auf vier begrenzt, da es vier Befehlsbits gibt und die

einzugebende Spannung in Millivolt maximal vier Zeichen lang sein soll. Neben dem Eingabefeld, welches für die Befehlsbits zuständig ist, wurde ein Label mit dem Text „Befehlsbits(Eingabe binär):“ platziert, damit dem Anwender klar ist, wofür das nebenstehenden Eingabefeld gedacht ist. Neben dem zweiten Eingabefeld wurde ein Label mit dem Text „Spannung{0-2999}[mV]“ positioniert, welches signalisiert, dass die Spannungseingabe mit der Einheit mV erfolgt und Spannungen in einem Bereich zwischen 0 und 2999 mV möglich sind (Erklärung in Kapitel 5.3.2). Außerdem wurde noch ein „Push Button“ mit dem Text „Senden“ erzeugt, der zur Versendung der eingegebenen Daten dient.

Das Feld oben rechts im Hauptfenster des GUIs dient dazu, ADCs auszuwählen, die ausgelesen werden sollen, und die Spannung, die an den ausgewählten ADCs anliegt, anzuzeigen. In das Feld wurde ein Label mit dem Text „AD7980“ platziert. Da zwei, vier, sechs oder acht ADCs gleichzeitig ausgelesen werden können, wurden in einer Reihe jeweils zwei „Text Browser“-Felder für die Spannungen, die an zwei ADCs anliegen, und ein „Radio Button“ zur Auswahl platziert. Neben den für die Spannungen der ADCs vorgesehenen Feldern sind Labels mit Bezeichnungen der ADCs und „[V]“ platziert worden, sodass ersichtlich ist, mit welchem Button welche ADCs ausgewählt werden und dass die Spannung in Volt angezeigt wird. Für die Bezeichnungen der ADCs wurde jeweils der Name des Pins eines ADCs, an dem die Daten ausgelesen werden, verwendet. Da es möglich sein soll, mehrere Buttons gleichzeitig zu aktivieren, wurden auch hier die Häkchen in den Kästchen „auto-exclusive“ in den Eigenschaften der vier verwendeten Buttons entfernt. Rechts in dem Feld wurde ein Button „Push Button“ mit dem Text „Lesen“ platziert, um die ausgewählten ADCs auslesen zu lassen, deren anliegende Spannungen anschließend in den jeweiligen Feldern angezeigt werden.

Das untere Feld im Hauptfenster des GUIs dient dazu, die fünf DACs vom Typ „AD5544“ einzustellen. Daher wurde dort ein Label mit dem Text „AD5544“ platziert. Durch Trennlinien wurden fünf Spalten erzeugt. In jeder der Spalten wurden vier Slider angeordnet. Unter jedem Slider wurde eine Progress Bar platziert. Diese dient zur Fortschrittsanzeige der Slider. Links in dem Feld wurde ein Label mit dem Text „Kanal:“ und darunter Labels mit den Namen der DAC-Kanäle platziert. Oben in jeder Spalte wurde ein Label mit der Bezeichnung eines DACs vom Typ „AD5544“ platziert. Hinter den Namen der DACs, welche die Spannung an ADCs direkt beeinflussen, wurden die Bezeichnungen dieser

ADCs, die auch im Feld der ADCs verwendet werden, in Klammern hinter den Bezeichnungen der DACs hinzugefügt. Somit ist ersichtlich, welcher Slider für welchen Kanal von welchem DAC zuständig ist.

Falls der Button „Senden“ im Feld des DACs vom Typ „AD5686“ oben links betätigt wird und dort kein DAC-Kanal ausgewählt ist oder die Eingabe in einem der beiden Eingabefelder fehlerhaft ist, öffnet sich ein neues Fenster mit einer Fehlermeldung. Dasselbe Fenster öffnet sich, wenn im Feld der ADCs der Button „Lesen“ betätigt wird und kein ADC ausgewählt ist Dafür wurde dem Projekt eine neue Qt-Designer-Formularklasse mit der Formularvorlage „Dialog without Buttons“ hinzugefügt. In der neuen Formulardatei wird ein neues Textfeld mit dem Inhalt „Bitte überprüfen Sie Ihre Eingabe!“ erstellt.

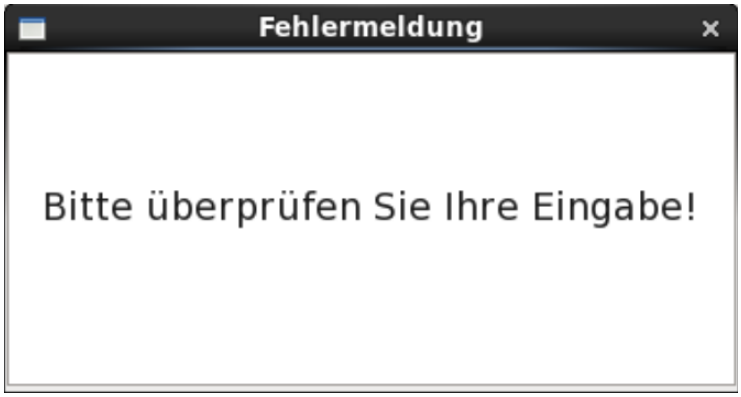

Abbildung 40: Dialog „Fehlermeldung“ des GUIs

5.2 Die Klasse „communication_EvalBoard“

Die Klasse „communication_EvalBoard“ ist für die Versendung von Daten an den FPGA und für den Empfang der vom FPGA gesendeten Daten zuständig. In der Headerdatei dieser Klasse wurden folgende Variablen definiert:

```
QString USB_Port;          //Bezeichnung des Ports
QSerialPort serial;
QString Byte1;             //String für die Synchronisation
QByteArray DatenDAC;       //QByteArray DatenDAC für die Übertragung
QByteArray DatenADC;       //QByteArray DatenDAC für die Übertragung
QByteArray readarray;      //QByteArray readyarray für die empfangenen Daten
bool ok2;                  //boolscher Wert zur Umwandlung von Werten
```

Der Konstruktor und der Destruktor dieser Klasse sind größtenteils von der Projektarbeit [4] übernommen worden. Der Konstruktor sorgt für die automatische Ermittlung des Ports, für die Konfiguration der Schnittstelle und dafür, dass die Funktion „read_serial()“ aufgerufen wird, sobald Daten empfangen worden sind [4]. Im Konstruktor wurde der Synchronisationsalgorithmus berücksichtigt, indem die Füllung des Strings „Byte1“ mit acht Einsen hinzugefügt wurde. Der Destruktor sorgt dafür, dass die Verbindung zum Port beendet wird [4]. Die folgenden Zeilen zeigen den Quelltext des Konstruktors und den des Destruktors:

```
//Konstruktor
communication_EvalBoard::communication_EvalBoard()
{
  //automatische Ermittlung des Ports
  foreach(const QSerialPortInfo &serialPortInfo, QSerialPortInfo::availablePorts()){
    if((serialPortInfo.vendorIdentifier()==10717) &&
      (serialPortInfo.productIdenti fier()==32769)){
      USB_Port = serialPortInfo.portName();
      qDebug() << USB_Port << "\n";
    }
  }
  //Konfiguration der Schnittstelle:
```

```
    serial.setPortName(USB_Port);                    //Arduino COM9 // Mojo COM3
        //  Linux: "/dev/ttyACM0"
    serial.setBaudRate(QSerialPort::Baud115200);
    serial.setDataBits(QSerialPort::Data8);
    serial.setParity(QSerialPort::NoParity);
    serial.setStopBits(QSerialPort::OneStop);
    serial.setFlowControl(QSerialPort::NoFlowControl);
    serial.open(QIODevice::ReadWrite);  //starte Port
    //readyRead-Signal einbinden:
    QObject::connect(&serial, SIGNAL(readyRead()), this, SLOT(read_serial()));
    Byte1 ="11111111";          //QString Byte 1 mit acht Einsen füllen
}

//Destruktor
communication_EvalBoard::~communication_EvalBoard()
{
    //Port schließen
    serial.close();
}
```

Für die Versendung von DAC Daten an den FPGA wurde die Funktion „sendDACdata(QString address, QString commandandchannel, QString data)“ definiert. Wenn diese Funktion aufgerufen wird, sendet sie erst vier Bytes voller Einsen zur Synchronisation und dann die Daten, die diese Funktion mit ihrem Aufruf erhält. Der String „address“ sollte hierbei dem Befehlscode des Kommunikationsprotokolls, mit dem der gewünschte DAC ausgewählt wird, entsprechen. Der String „commandandchannel“ sollte acht Zeichen lang sein und, falls die Daten für DAC vom Typ „AD5686“ bestimmt sind, die acht MSBs für diesen DAC enthalten. Falls sie für einen DAC vom Typ „AD5544“ bestimmt sind, sollten die ersten sechs Zeichen gleich null sein und die anderen beiden die ersten zwei Bits für diesen DAC enthalten. Der String „data“ sollte vier Zeichen lang sein und das hexadezimale Äquivalent der Datenbits, die in den ausgewählten DAC geschrieben werden sollen, beinhalten.

Diese Funktion nutzt ein QByteArray, welches an den FPGA gesendet wird. Ein Feld dieses Arrays ist dabei ein Byte groß. Die ersten vier Felder werden mit Bytes voller Einsen gefüllt. Das fünfte Feld wird mit dem String „address“, das sechste mit dem String „commandandchannel“ und die letzten beiden mit dem String „data“ gefüllt. Anschließend wird das QByteArray an den FPGA gesendet. Die folgenden Zeilen zeigen den Quelltext der beschriebenen Funktion:

```
//Funktion für die Ansteuerung eines DACs
void communication_EvalBoard::sendDACdata(QString address, QString commandandchannel, QString data) {
   //Füllen des QByteArrays
DatenDAC[0]=Byte1.toInt(&ok2,2); //Element 1 = Integerwert des Bytes mit acht Einsen
DatenDAC[1]=Byte1.toInt(&ok2,2); //Element 2 = Integerwert des Bytes mit acht Einsen
DatenDAC[2]=Byte1.toInt(&ok2,2); //Element 3 = Integerwert des Bytes mit acht Einsen
DatenDAC[3]=Byte1.toInt(&ok2,2); //Element 4 = Integerwert des Bytes mit acht Einsen
DatenDAC[4]=address.toInt(&ok2,2);                //Element 5 = Integerwert von Adresse
DatenDAC[5]=commandandchannel.toInt(&ok2,2);          //Element 6 = Integerwert von commandandchannel
DatenDAC[6]=data.left(2).toInt(&ok2,16);    //Element 7 = Integerwert der zwei linken-Zeichen von Data
DatenDAC[7]=data.right(2).toInt(&ok2,16);   //Element 8 = Integerwert der zwei rechten Zeichen von Data
serial.write(QByteArray::fromRawData(DatenDAC,8));    //QByteArray DatenDAC senden
}
```

Da zur Auslesung der ADCs neben den Bytes, die zur Synchronisation dienen, nur ein weiteres Byte gesendet werden muss, wurde eine weitere Funktion „sendADCdata(QString Adresse)“ für diese Aufgabe erstellt. Die Funktion sendet vier Bytes voller Einsen zur Synchronisation und die Daten des Strings „Adresse“, die ihr mit Aufruf der Funktion übergeben werden, an den FPGA. Der String „Adresse“ sollte also einem Befehlscode des Kommunikationsprotokolls entsprechen, welches die Auslesung von ADCs auslöst. Auch diese Funktion nutzt ein QByteArray, welches an den FPGA gesendet wird. Die ersten vier Felder werden auch hierbei mit Bytes voller Einsen gefüllt.

Das fünfte Feld wird mit dem String „Adresse“ gefüllt. Anschließend wird das QByte-Array an den FPGA gesendet. Die folgenden Zeilen zeigen den Quelltext der beschriebenen Funktion:

```
//Funktion für die Auslesung von ADCs
void communication_EvalBoard::sendADCdata(QString Adresse) {
//Füllen des QByteArrays
DatenADC[0]=Byte1.toInt(&ok2,2); //Element 1 = Integerwert des Bytes mit acht Einsen
DatenADC[1]=Byte1.toInt(&ok2,2); //Element 2 = Integerwert des Bytes mit acht Einsen
DatenADC[2]=Byte1.toInt(&ok2,2); //Element 3 = Integerwert des Bytes mit acht Einsen
DatenADC[3]=Byte1.toInt(&ok2,2); //Element 4 = Integerwert des Bytes mit acht Einsen
DatenADC[4]=Adresse.toInt(&ok2,2);      //Element 5 = Integerwert der zwei rechte
Zeichen von datenhex
serial.write(QByteArray::fromRawData(DatenADC,5));    //QByteArray DatenADC
senden
}
```

Für den Start und den Stopp der kontinuierlichen Auslesung aller ADCs wurden Funktionen erstellt, welche die Funktion „sendADCdata(QString Adresse)“ nutzen und dabei den Befehlscode übergeben, der nach der Synchronisation den Start bzw. den Stopp der kontinuierlichen Auslesung initiiert. Der Quelltext dieser Funktionen sieht wie folgt aus:

```
//Funktion zum Starten der kontinuierlichen Auslesung aller ADCs
void communication_EvalBoard::startreadcont()    {
   sendADCdata("10000111");
}
//Funktion zum Stoppen der kontinuierlichen Auslesung aller ADCs
void communication_EvalBoard::stopreadcont()     {
   sendADCdata("00000111");
}
```

Die Funktion „read_serial()“ wird aufgerufen, wenn Daten empfangen werden. Die empfangenen Daten werden im QByteArray „readarray“ gespeichert. Sobald die Daten vollständig empfangen worden sind, wird das Signal „connectsignal()“ gesetzt, welches in der Headerdatei dieser Klasse definiert worden ist. Der Grund für die Aktivierung dieses

Signals wird in Kapitel 5.3.1 erläutert. Der Quelltext der Funktion „read_serial()“ steht in den folgenden Zeilen:

```
//Funktion zum Datenempfang über USB
void communication_EvalBoard::read_serial() {
  readarray = serial.readAll(); //QByteArray readarray mit den empfangenen Daten füllen
  connectsignal();              //Das Signal connectsignal setzen
}
```

Die Funktion „getdata()“ gibt das QByteArray „readarray“, welches die zuletzt empfangenen Daten enthält, zurück. Der Quelltext dieser Funktion sieht wie folgt aus:

```
//Funktion zur Rückgabe der empfangenen Daten
QByteArray communication_EvalBoard::getdata() {
   return readarray;
}
```

5.3 Die Klasse „MojoSerial“

5.3.1 Allgemeines

Die Klasse „MojoSerial“ dient dazu, die vom Benutzer im GUI getätigten Eingaben der DACs für die Nutzung der verschiedenen Funktionen der Klasse „communication_EvalBoard“ zu präparieren und anschließend mithilfe dieser Funktionen Daten an den FPGA zu senden, sodass sich in Folge die im GUI getätigten Eingaben an den ausgewählten DACs einstellen. Außerdem dient die Klasse dazu, mithilfe der Klasse „communication_EvalBoard“ die Daten, die für die Auslesung der vom Benutzer ausgewählten ADCs sorgen, an den FPGA zu senden. Zudem soll sie die vom FPGA an den PC gesendeten Daten von der Klasse „communication_EvalBoard“ entgegennehmen und anhand dieser Daten ausrechnen, welche Spannung an welchem ADC anliegt und dies im GUI anzeigen.

Der Konstruktor und Destruktor dieser Klasse wurde vom QT Creator selbst erzeugt, da eine „Qt-Widgets-Anwendung“ erstellt wurde. Im Konstruktor dieser Klasse wird das Signal „connectsignal()“, welches in der Klasse „communication_EvalBoard“ gesetzt wird, nachdem Daten empfangen worden sind, mit der Funktion „ad7980()“ dieser Klasse verknüpft. Die Funktion wird folglich immer aufgerufen, nachdem Daten empfangen worden sind. Zudem wird im Konstruktor dafür gesorgt, dass mit „Kommunikation“ auf Funktionen der Klasse „communication_EvalBoard“ zugegriffen werden kann. Der Quelltext des Konstruktors und der des Destruktors sieht wie folgt aus:

```
MojoSerial::MojoSerial(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MojoSerial)
{
    ui->setupUi(this);
    Kommunikation = new communication_EvalBoard();
    connect(Kommunikation, SIGNAL(connectsignal()), this, SLOT(ad7980()));
}

MojoSerial::~MojoSerial()
{
```

```
    delete ui;
}
```

In der Headerdatei dieser Klasse wurden folgende Variablen definiert:

```
QString P0P90, P45P135, I0I2, I1I3; //Strings für die Auswahl der Browserfelder
QString Adresse;                    //String für die Adresse
QString Kanal;                      //String für die Kanalauswahl
bool ok;                            //boolscher Wert zur Umwandlung von Werten
```

5.3.2 Funktionen für das Feld „AD5686“ des GUIs

In der Formulardatei des Hauptfensters wurde dem Button „Senden“ die Aktion „clicked()“ hinzugefügt. In der Quelldatei mit dem Projektnamen wurde folgende Funktion generiert:

```
void MojoSerial::on_pushButton_clicked()
{

}
```

Diese Funktion legt fest, was passiert, wenn der Button angeklickt wird. Zuerst werden die folgenden Variablen definiert:

```
QString commandstr;       //String für die eingegebenen Command Bits
QString D, C, B, A;       //Strings für die Kanäle D, C, B, A
QString DCBA;             //Strings zum Zusammenfassen von D, C, B, A
QString spannungsstr;     //String für die eingegebene Spannung
QString Adresse="00000101"; //QString Adresse für die Übertragung
```

Danach wird der Inhalt des Textfeldes „lineEdit_3“, in welches die vier Befehlsbits für den DAC vom Typ „AD5686“ eingetragen werden sollen, dem String „commandstr“ übergeben. Da die Befehlsbits binär eingegeben werden und das dezimale Äquivalent dieser Eingabe für eine spätere Abfrage benötigt wird, wird dem Integerwert „commandbits“ der String „commandstr“ zugewiesen, nachdem dieser in einen ganzzahligen Wert umgewandelt wurde. Der Quelltext dafür sieht wie folgt aus:

```
//Abspeichern der eingegebenen Befehlsbits und Umwandlung in einen Integerwert
commandstr = ui->lineEdit_3->text();
int commandbits=commandstr.toInt(&ok,2);
```

Die Buttons für die DAC-Kanäle werden jeweils mit einer if-Bedingung abgefragt. Falls ein Button aktiviert ist, wird ein String, der genauso heißt wie der ausgewählte DAC-Kanal, mit „1“ gefüllt, ansonsten mit „0“. Nach den if-Abfragen werden alle Strings zu einem String „DCBA“ zusammengefügt. Dieser String wird dann an den String mit den Befehlsbits gehängt und dem String „commchanstring“ übergeben. Somit stehen die ersten acht MSBs für das Eingangsschieberegister (siehe Kapitel 3.1.3) in einem String. Der Quelltext dazu steht in den folgenden Zeilen:

```
//Überprüfung, ob die jeweiligen Buttons für die Kanäle true oder false sind und das Füllen der dazugehörigen Strings
if (ui->radioButton->isChecked()) {      //wenn radioButton true ist,
   D="1";                                //String D mit "1" füllen,
} else {                                 //ansonsten
   D="0";                                //mit "0"
}
if (ui->radioButton_2->isChecked()) {    //wenn radioButton_2 true ist,
   C="1";                                //String D mit "1" füllen,
} else {                                 //ansonsten
   C="0";                                //mit "0"
}
if (ui->radioButton_3->isChecked()) {    //wenn radioButton_3 true ist,
   B="1";                                //String D mit "1" füllen,
} else {                                 //ansonsten
   B="0";                                //mit "0"
}
if (ui->radioButton_4->isChecked()) {    //wenn radioButton_4 true ist,
   A="1";                                //String D mit "1" füllen,
} else {                                 //ansonsten
   A="0";                                //mit "0"
}
```

```
//Verbinden der Befehlsbits und der ausgewählten DAC-Kanäle zu einem String (binär)
DCBA=D+C+B+A;
QString commchanstring=commandstr+DCBA;
```

Die maximal mögliche Ausgangsspannung bei einer in diesem Projekt bestehenden Referenzspannung von 3 V beträgt laut Berechnung mit der Übertragungsfunktion aus Kapitel 3.1.3:

$$V_{OUT} = 3\,V * \frac{65535}{65536} = 2{,}99995\,V.$$

Die kleinstmögliche einstellbare Ausgangsspannung nach 0 V (siehe Kapitel 3.1.3) beträgt:

$$V_{OUT} = 3\,V * \frac{1}{65536} = 0{,}00005\,V.$$

Aufgrund der 16-Bit Auflösung ergibt sich also eine Spannungsdifferenz von mindestens 0,00005 V zwischen zwei unterschiedlich eingestellten Ausgangsspannungen. Für diese QT-Anwendung soll es ausreichend sein, ganzzahlige mV Werte in die Anwendung eingeben zu können. Die maximal einstellbare Ausgangsspannung beträgt also 2999 mV. Für die spätere Übertragung der 16 Datenbits muss also erstmal das dezimale Äquivalent dieser Eingabe mit der Übertragungsfunktion aus Kapitel 3.1.3 bestimmt werden. Dafür muss die Formel nach „D“ umgestellt werden. Eine Umstellung unter Berücksichtigung, dass der Verstärkungsfaktor Gain in diesem Projekt gleich eins ist, ergibt:

$$D = \frac{V_{OUT}}{V_{REF}} * 2^N$$

Nachdem der Inhalt des Textfeldes der Spannung im String „spannungsstr“ gespeichert wurde, wird er in einen ganzzahligen Wert umgewandelt und dem Integerwert „spannungseingabe“ zugewiesen. Mit der umgestellten Formel wird das dezimale Äquivalent der 16 Datenbits errechnet und dem Integerwert „Di“ zugewiesen. Dieser wird in einen hexadezimalen Wert und dann in einen String umgewandelt, welcher im String „Distr“ gespeichert wird. Es findet eine if-Abfrage, mit der festgestellt wird, ob „Distr“ vier Zeichen beinhaltet, statt. Falls der String weniger Zeichen beinhaltet, werden vor den String so viele Nullen gehängt, dass er genau vier Zeichen lang ist. Das ist wichtig, da der hexadezimale String die 16 Datenbits beinhalten soll und dafür vier Zeichen lang sein muss.

Dabei ist zu berücksichtigen, dass eine Hexadezimalzahl, die aus vier Zeichen besteht, einer Binärzahl entspricht, die aus 16 Zeichen besteht. Der Quelltext für den in diesem Abschnitt beschriebenen Ablauf sieht wie folgt aus:

```
//Abspeichern der eingegebenen Spannung, Umwandlung in einen Integerwert und Berechnung des Wertes für die Übertragung
spannungsstr = ui->lineEdit_4->text()
int spannungseingabe = spannungsstr.toInt(&ok, 10);
int Di=((spannungseingabe*65536)/3000);
//Umwandlung des dezimalen Übertragungwertes in einen hexadezimalen Wert und einen String
QString Distr = QString::number(Di,16);
//falls Distr kleiner als vier Zeichen groß ist, wird mit so vielen Nullen aufgefüllt, dass Distr genau vier Zeichen beinhaltet
if (Distr.size()<2) {
   Distr= "000" + Distr;
}  else if (Distr.size()<3) {
   Distr= "00" + Distr;
}  else if (Distr.size()<4) {
   Distr= "0" + Distr;
}
```

Der anfangs definierte String „Adresse“ enthält den Befehlscode mit dem der DAC vom Typ „AD5686“ ausgewählt wird. Der String „commchanstring“ enthält die acht MSBs in binärer Form und der String „Distr“ die 16 Datenbits für das Schieberegister des DACs vom Typ „AD5686“ in hexadezimaler Form. Die genannten Strings werden bei Bedarf der Funktion „sendDACdata(QString address, QString commandandchannel, QString data)“ übergeben, sodass die Daten an den FPGA gesendet werden. Dies soll aber nur geschehen, wenn die Eingaben korrekt erfolgt sind. Ansonsten soll der Dialog „Fehlermeldung“ (siehe Abbildung 40) erscheinen.

Mithilfe einer if-Bedingung werden Fehler bei der Eingabe überprüft. Der String „DCBA“ darf nicht nur Nullen enthalten, da in diesem Fall kein Kanal ausgewählt wurde. Die Größe des Strings „commandstr“ darf nicht kleiner als vier sein, da es vier Befehlsbits gibt. Falls andere Zeichen als Einsen und Nullen als Befehlsbits eingegeben werden, ist

das ganzzahlige Äquivalent gleich null. Das ganzzahlige Äquivalent darf also nur gleich null sein, wenn auch vier Nullen eingegeben wurden. Falls andere Zeichen als ganze Zahlen in die Spannungseingabe eingegeben werden, ist das ganzzahlige Äquivalent der Spannungseingabe gleich null. Es darf also nur null sein, wenn mindestens eine Null und kein anderes Zeichen eingegeben wurde. Außerdem darf das ganzzahlige Äquivalent nicht größer als 2999 sein, da keine größere Ausgangsspannung eingestellt werden kann. Falls mindestens eine dieser Bedingungen nicht erfüllt ist, erscheint der Dialog mit der Fehlermeldung, welcher das Hauptfenster deaktiviert. Es kann erst wieder eine Änderung in die Eingabe des Hauptfensters erfolgen, wenn der Dialog geschlossen wurde. Wenn die Eingaben korrekt erfolgt sind, wird die Funktion „sendDACdata(QString address, QString commandandchannel, QString data)“ aufgerufen, sodass die Daten an den FPGA gesendet werden. Wenn dies der Fall ist, wird das Eingabefeld der Spannung geleert, damit dies für eine neue Datenübertragung nicht durch den Anwender geschehen muss. Der Quelltext dafür sieht wie folgt aus:

```
/*Überprüfung, ob bei der Eingabe mindestens ein Kanal ausgewählt wurde, ob vier Befehlsbits und keine anderen Zeichen eingegeben wurden und ob kein größerer Wert als 2999 bei der Spannung eingegeben wurde und keine anderen Zeichen. Falls die Eingabe falsch erfolgt ist, öffnet sich ein neues Fenster mit einer Fehlermeldung, ansonsten werden die Daten der Funktion "sendDACdata" übergeben und das Textfeld für die Spannungseingabe wird geleert*/
if (DCBA == "0000" or commandstr.size()<4 or (commandbits==0 and commandstr !="0000") or spannungseingabe>2999 or ((spannungsstr!="0" and spannungsstr!="00" and spannungsstr!="000" and spannungsstr!="0000") and spannungseingabe==0)) {
   FensterZwei fensterzwei;
   fensterzwei.setModal(true);
   fensterzwei.exec();
} else {
Kommunikation->sendDACdata(Adresse, commchanstring, Distr);
//Inhalte des Textfeldes der Spannung löschen
ui->lineEdit_4->clear();
}
```

5.3.3 Funktionen für das Feld „AD5544“ des GUIs

Für jeden Kanal jedes DACs vom Typ „AD5544“ gibt es einen Slider und eine Progress Bar. Da jede Progress Bar den Wert des dazugehörigen Sliders annehmen und somit anzeigen soll, wurde jedem Slider in der Formulardatei des Hauptfensters die Aktion „valueChanged(int)“ hinzugefügt. In jede dieser Funktionen wurde programmiert, dass die dazugehörige Progress Bar denselben Wert annimmt. Der Quelltext für eine dieser Funktionen sieht wie folgt aus:

```
void MojoSerial::on_SliderU28A_valueChanged(int value)
{
   ui->BarU28A->setValue(value);
}
```

Wenn ein Slider per Maus gezogen und wieder losgelassen wird, soll der ausgewählte Kanal des ausgewählten DACs auf die Prozentzahl der dazugehörigen Progress Bar gestellt werden. Dafür wurde jedem Slider in der Formulardatei des Hauptfensters die Aktion „sliderReleased()“ hinzugefügt. In jeder dieser Funktionen wird die Funktion „AD5544(int progress, QString Adresse, QString Kanal)“ aufgerufen und ihr der Wert der jeweiligen Progress Bar, der Befehlscode, welcher dem ausgewählten DACs im Kommunikationsprotokoll entspricht, und ein Byte, bei welchem die zwei LSBs dem ausgewählten Kanal entsprechen (siehe Tabelle 2), übergeben. Die folgenden Zeilen zeigen ein Beispiel für die Funktion „sliderReleased()“ und den Aufruf der erwähnten Funktion:

```
void MojoSerial::on_SliderU12C_sliderReleased()
{
   AD5544(ui->BarU12C->value(), "00000001", "00000010");
}
```

Die erwähnte Funktion „AD5544(int progress, QString Adresse, QString Kanal)“ rechnet aus dem erhaltenen Wert einer Progress Bar einen hexadezimalen Wert aus. Die DACs vom Typ „AD5544“ erhalten 16 Datenbits. Wenn alle 16 Bits gleich eins sind, beträgt das dezimale Äquivalent 65535. Eine Progress Bar nimmt ganzzahlige Werte zwischen null und 100 an. Deshalb wird der erhaltene Wert einer Progress Bar durch 100 geteilt und mit 65535 multipliziert. Dadurch erhält man das dezimale Äquivalent, das den 16 Datenbits entsprechen muss, damit sich der gewünschte Wert an dem ausgewählten DAC

einstellt. Dieser Wert wird noch in eine hexadezimale Zahl umgewandelt. Wenn der hexadezimale Wert kleiner als vier Zeichen groß ist, werden vor diesem noch so viele Nullen gehängt, bis er genau vier Zeichen groß ist, damit dieser Wert alle 16 Datenbits enthält. Anschließend werden der Befehlscode, welcher dem ausgewählten DACs im Kommunikationsprotokoll entspricht, das Byte, bei welchem die zwei LSBs dem ausgewählten Kanal entsprechen und der errechnete hexadezimale Wert der Funktion „sendDACdata(QString address, QString commandandchannel, QString data)“ übergeben, sodass diese Daten an den FPGA gesendet werden. Die folgenden Zeilen zeigen den Quelltext dafür:

```
//Funktion, welche nach Loslassen eines Sliders aufgerufen wird, um Daten für die DACs vom Typ "AD5544" zu berechnen
void MojoSerial::AD5544(int progress, QString Adresse, QString Kanal) {
   int datenint = (progress * 65535)/100;
   //Umwandlung des dezimalen Wertes in einen hexadezimalen Wert und einen String, um das QByteArray Daten füllen zu können
   QString datenhex = QString::number(datenint,16);
   //falls datenhex kleiner als vier Zeichen groß ist, wird mit so vielen Nullen aufgefüllt, dass datenhex genau vier Zeichen beinhaltet
   if (datenhex.size()<2) {
      datenhex= "000" + datenhex;
   }  else if (datenhex.size()<3) {
      datenhex= "00" + datenhex;
   }  else if (datenhex.size()<4) {
      datenhex= "0" + datenhex;
   }
   Kommunikation->sendDACdata(Adresse, Kanal, datenhex);
}
```

5.3.4 Funktionen für das Feld „AD7980“ des GUIs

Ein Klick auf den Button „Lesen“ im Feld der ADCs ruft die Funktion „on_buttonlesen_clicked()“ auf. Diese Funktion überprüft, welche ADCs ausgewählt worden sind. Wenn ein Button aktiviert ist, wird ein String, der genauso heißt wie die Bezeichnungen der ausgewählten ADCs im Feld „AD7980“, mit „1“ gefüllt, ansonsten mit „0“. Die vier Strings der Buttons werden aneinandergekettet und mit dem String „0110“ zusammengeführt und dem String „ADCAdresse“ zugewiesen. Dieser String entspricht, wenn mindestens zwei ADCs ausgewählt worden sind, dem Befehlscode des Kommunikationsprotokolls, welcher dafür sorgt, dass die ausgewählten ADCs ausgelesen werden und die ausgelesenen Daten anschließend an den PC gesendet werden. Daher wird, wenn mindestens zwei ADCs ausgewählt worden sind, die Funktion „sendADCdata(ADCAdresse)“ aufgerufen und der String „ADCAdresse“ dieser Funktion übergeben, sodass die Daten an den FPGA gesendet werden. Falls kein Button aktiviert ist, öffnet sich der Dialog mit der Fehlermeldung, welcher das Hauptfenster deaktiviert. Das Hauptfenster wird erst wieder aktiviert, wenn der Dialog geschlossen wurde. Die folgenden Zeilen zeigen die Funktion „on_buttonlesen_clicked()“:

```
//Button "Lesen" im Feld der ADCs vom Typ "AD7980" wird betätigt:
void MojoSerial::on_Buttonlesen_clicked()
{
   QString ADCAdresse;       //String für die Adresse der ADCs
   //Überprüfung, ob die jeweiligen Buttons für die Kanäle true oder false sind und das
Füllen der dazugehörigen Strings
   if (ui->ButtonP0P90->isChecked()) {      //wenn ButtonP0P90 true ist,
      P0P90="1";                  //String P0P90 mit "1" füllen,
   } else {                       //ansonsten
      P0P90="0";                  //mit "0"
}
   if (ui->ButtonP45P135->isChecked()) {    //wenn ButtonP45P135 true ist,
      P45P135="1";                //String P45P135 mit "1" füllen,
   } else {                       //ansonsten
      P45P135="0";                //mit "0"
}
```

```
if (ui->ButtonI0I2->isChecked()) {    //wenn ButtonI0I2 true ist,
      I0I2="1";                       //String I0I2 mit "1" füllen
   } else {                          //ansonsten
      I0I2="0";                        //mit "0"
   }
if (ui->ButtonI1I3->isChecked()) {    //wenn ButtonI1I3 true ist,
   } else {                          //ansonsten
      I1I3="0";                        //mit "0
   }
   ADCAdresse= P0P90 + P45P135 + I0I2 + I1I3 +"0110";
//wenn kein ADC ausgewählt wurde, erscheint die Fehlermeldung, ansonsten werden die
Daten der Funktion "sendADCdata" übergeben
    if (ADCAdresse=="00000110") {
       FensterZwei fensterzwei;
       fensterzwei.setModal(true);
       fensterzwei.exec();
    } else {
     Kommunikation->sendADCdata(ADCAdresse);
    }
}
```

Wie in Kapitel 5.3.1 erklärt wurde, wird die Funktion „ad7980()“ aufgerufen, nachdem Daten, die der FPGA an den PC gesendet hat, empfangen worden sind. In dieser Funktion wird die Funktion „getdata()“ der Klasse „communication_EvalBoard“ aufgerufen. Dadurch werden die empfangenen Daten zurückgegeben, welche im QByteArray „readarray“ abgespeichert werden. Da der FPGA die Daten der ausgelesenen ADCs in einer festen Reihenfolge sendet (siehe Kapitel 4.8), können sie mithilfe von if-Abfragen den einzelnen ADCs zugewiesen werden. Die Reihenfolge, in der der FPGA die Daten an den PC sendet, ist im Anhang in Tabelle 7 erkennbar.

Die Zuweisung der empfangenen Daten wird exemplarisch für zwei dieser ADCs erklärt. Es wird abgefragt, ob der Button „P45P135“ gleich eins ist. Wenn dies der Fall ist, wurde dem FPGA beim Klicken des Buttons „Lesen“ mitgeteilt, dass diese ADCs ausgelesen und die ausgelesenen Daten an den PC zurückgeschickt werden sollen. Somit sind Daten

dieser beiden ADCs empfangen worden. Es wird abgefragt, ob der Button „P0P90“ aktiviert ist. Ist dies der Fall, sind die fünften und sechsten Bytes des QByteArrays „readarray“ dem ADC „P45“ und die siebten und achten Bytes von „readarray“ dem ADC „P135“ zuzordnen, da die ersten vier Bytes die Daten der ADCs „P0“ und „P90“ enthalten. Ist der Button „P0P90“ nicht aktiviert, enthalten die ersten zwei Bytes die Daten des ADCs „P45“ und das dritte und vierte Byte die Daten des ADCs „P135“. Diese Daten werden in hexadezimale Werte umgewandelt und Strings übergeben. Danach werden der Funktion „writeinBrowser(QString daten1str, QString daten3str, QString textfelder)“ die Strings sowie ein String, aus dem hervorgeht, dass die Daten von den ADCs „P45“ und „P135“ stammen, übergeben. Nach dem erläuterten Verfahren werden die empfangenen Daten sortiert und der genannten Funktion übergeben. Die folgenden Zeilen zeigen den Quelltext der Funktion „ad7980()“:

```
//Funktion, die die per USB empfangenen Daten auswertet
void MojoSerial::ad7980() {
   QByteArray readarray = Kommunikation->getdata();
if (P0P90=="1") {
   QString p0str = readarray.left(2).toHex();
   QString p90str = readarray.mid(2,2).toHex();
   writeinBrowser(p0str, p90str, "textP0P90");
}
if (P45P135=="1") {
   QString p45str;
   QString p135str;
   if (P0P90=="1") {
     p45str = readarray.mid(4,2).toHex();
     p135str = readarray.mid(6,2).toHex();
   } else {
     p45str = readarray.left(2).toHex();
     p135str = readarray.mid(2,2).toHex();
   }
   writeinBrowser(p45str,p135str, "textP45P135");
}
if (I0I2=="1") {
```

```
    QString i0str;
    QString i2str;
    if (P0P90=="1" and P45P135=="1"){
      i0str = readarray.mid(8,2).toHex();
      i2str = readarray.mid(10,2).toHex();
    } else if (P0P90=="1" or P45P135=="1") {
      i0str = readarray.mid(4,2).toHex();
      i2str = readarray.mid(6,2).toHex();
    } else {
      i0str = readarray.left(2).toHex();
      i2str = readarray.mid(2,2).toHex();
    }
    writeinBrowser(i0str, i2str, "textI0I2");
}
if (I1I3=="1") {
   QByteArray i1i3array;
   i1i3array = readarray.right(4);
   QString i1str = i1i3array.left(2).toHex();
   QString i3str = i1i3array.right(2).toHex();
   writeinBrowser(i1str, i3str, "textI1I3");
   }
}
```

Die Funktion „writeinBrowser(QString daten1str, QString daten3str, QString textfelder)“ erhält als Parameter die Daten von zwei ADCs in hexadezimaler Form und die Information, von welchen ADCs diese Daten sind. Sie dient dazu, die Daten in die an den ADCs anliegenden Spannungen umzurechnen und diese in die dafür vorgesehenen Felder des GUIs zu schreiben. Die Daten der zwei ADCs in hexadezimaler Form werden in Integerwerte umgewandelt. Anschließend werden mit der Formel aus Kapitel 3.3 die an den beiden ADCs anliegenden Spannungen berechnet und Strings übergeben. Diese Strings werden in Textfelder der jeweiligen ADCs im GUI platziert, sodass die an den ausgewählten ADCs anliegenden Spannungen im GUI angezeigt werden. Die folgenden Zeilen zeigen die in diesem Abschnitt beschriebene Funktion:

```
//Funktion, um die Daten zweier ADCs in Spannungen umzurechen und in die dafür vor-
gesehenen Felder zu schreiben
void MojoSerial::writeinBrowser(QString daten1str, QString daten2str, QString textfel-
der) {
    int daten1 = daten1str.toInt(&ok, 16);
    float spannung1 = ((daten1 * 3.0000) / 65536.0000);
    QString spannung1str = QString::number(spannung1, 'f', 6);

    int daten2 = daten2str.toInt(&ok, 16);
    float spannung2 = ((daten2 * 3.0000) / 65536.0000);
    QString spannung2str = QString::number(spannung2, 'f', 6);

    if (textfelder == "textP0P90") {
        ui -> textP0 -> setText(spannung1str);
        ui -> textP90 -> setText(spannung2str);
    }   else if (textfelder == "textP45P135") {
        ui -> textP45 -> setText(spannung1str);
        ui -> textP135 -> setText(spannung2str);
    }   else if (textfelder == "textI0I2") {
        ui -> textI0 -> setText(spannung1str);
        ui -> textI2 -> setText(spannung2str);
    }   else if (textfelder == "textI1I3") {
        ui -> textI1 -> setText(spannung1str);
        ui -> textI3 -> setText(spannung2str);
    }
}
```

6 Praktische Durchführung

Die Auslesung der ADCs und die Versendung der ausgelesenen Daten an den PC erfolgte ohne Probleme. Die Felder der ausgewählten ADCs im GUI zeigen, nachdem auf den Button „Lesen“ geklickt wird, die Differenz der an den „V+“ und „V-“-Pins gemessenen Spannungen der verschiedenen ADCs bis auf wenige mV genau an. Es wurden alle ADCs gemessen und mit den im GUI angezeigten Werten verglichen.

Auch lässt sich der DAC vom Typ „AD5686“ einwandfrei ansteuern. Dafür wurden für die Befehlsbits „0011“ in die QT-Anwendung eingegeben, da sich die eingegebene Spannung direkt nach der Versendung an dem angesteuerten DAC-Kanal einstellen sollte. Abbildung 41 zeigt exemplarisch die Messung des DAC-Kanals „B“.

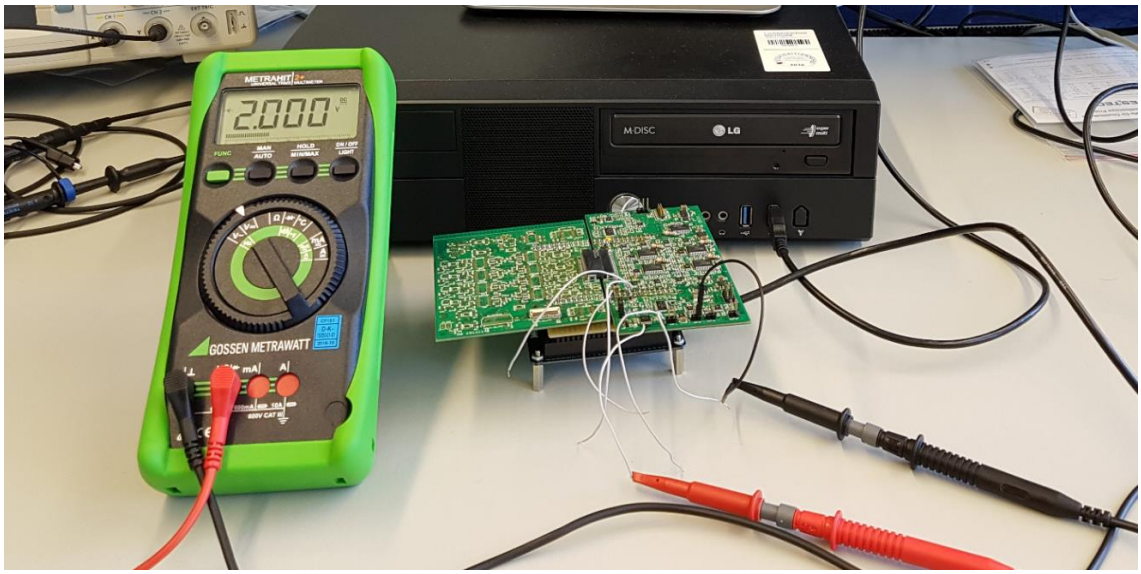

Abbildung 41: Messung der Ausgangsspannung des DAC-Kanals „B“ des DACs vom Typ „AD5686“

Die Referenzspannung beträgt in diesem Projekt 3000 mV. Nachdem das VHDL-Design auf dem FPGA implementiert wurde, sollte sich an den Ausgängen, wie in Kapitel 3.1.1 erklärt, die halbe Referenzspannung einstellen. Mit einem Multimeter wurden 1490 mV gemessen.

In der folgenden Tabelle wird die Differenz in mV zwischen den mit der QT-Anwendung gesendeten Spannungswerten und den Messwerten der Ausgangsspannung dargestellt. Dabei wurde der DAC-Kanal „A“ angesteuert und gemessen.

Tabelle 6: Differenz der gesendeten und gemessenen Werte des DACs vom Typ „AD5686“

gesendeter Wert [mV]	gemessener Wert [mV]	Differenz [mV]
0	9,9	-9,9
375	367	8
750	741	9
1125	1115	10
1500	1490	10
1875	1864	11
2250	2239	11
2625	2614	11
2999	2987	12

Anhand der Tabelle lässt sich erkennen, dass die Differenz im unteren Spannungsbereich, bis auf den gesendeten Wert von 0 mV, 8 mV und mit zunehmender Spannung maximal 12 mV beträgt. Da vor der ersten Sendung 1490 mV gemessen wurden und die halbe Referenzspannung, also 1500 mV anliegen müssten, ist die Differenz zwischen den gesendeten und gemessenen Werten noch im Rahmen und u. a. auf die nicht optimale Messung zurückzuführen.
Außerdem ließen sich alle DAC-Kanäle des DACs vom Typ „AD5686“ einwandfrei einzeln und in Kombination ansteuern, wobei die Differenz zwischen den gesendeten und den gemessenen Werten immer zwischen 8 mV und 12 mV lag.

Die Kanäle der ADCs vom Typ „AD5544“ konnten nicht direkt gemessen werden, da sie Ströme und keine Spannungen ausgeben und eine Strommessung auf der Platine nicht ohne Weiteres möglich ist. Allerdings sind diese DACs mit Verstärkern, die eine Spannung ausgeben, verschaltet. Diese Spannungen liegen an den ADCs an. Dadurch konnte mit Messungen bzw. mit der Auslesung der ADCs überprüft werden, ob sich diese Spannungen mit unterschiedlichen Einstellungen der einzelnen Kanäle dieser DACs so veränderten, wie es zu erwarten war.
Die Spannungen, die mit den DACs vom Typ „AD5544“ beeinflusst werden, haben sich je nachdem, wie die verschiedenen Jumper auf der Platine, welche u.a. die Referenzspannungen und die analogen Grounds der DACs vom Typ „AD5544“ beeinflussen, gesteckt waren, anders verhalten.

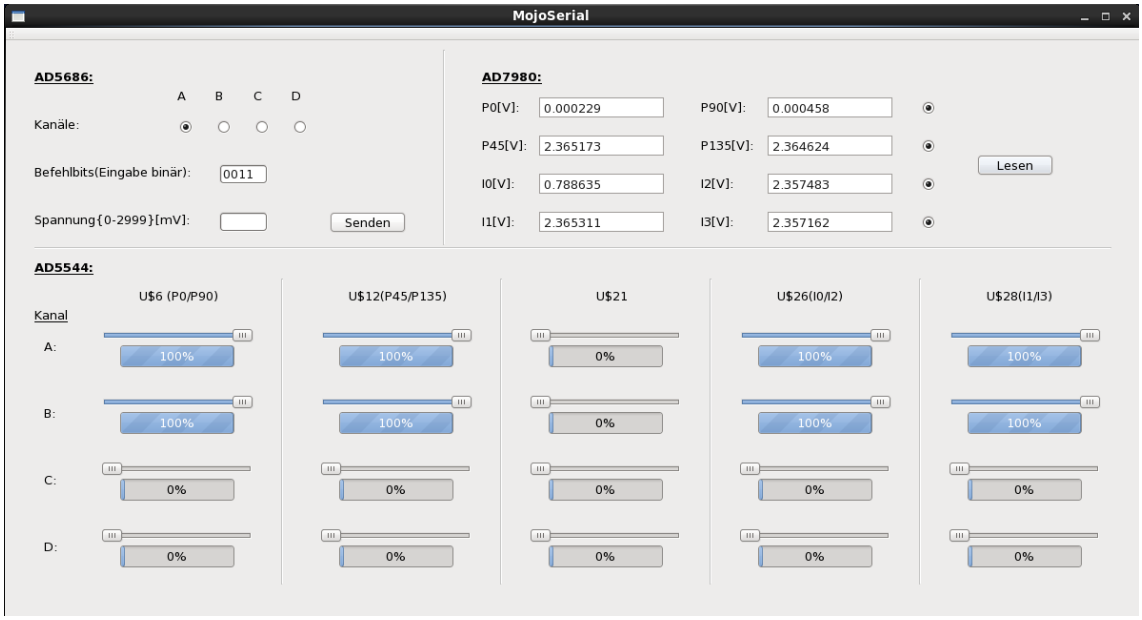

Abbildung 42: das Hauptfenster des GUIs bei einem Test

Abbildung 42 zeigt, wie die DACs vom Typ „AD5544 bei einem Test eingestellt waren und welche Spannungen an den ADCs mit diesen Einstellungen anlagen. Bei diesem Test wurden die Jumper folgendermaßen gesteckt:

- „JP_3V“: Pin 1 mit Pin 2
- „JP_P1V“: Pin 1 mit Pin 2
- „JP_I1V“: Pin 1 mit Pin 2
- „JP_1VREF“: Pin 4 mit Pin 6

Die nicht erwähnten Jumper wurden nicht gesetzt. Eigentlich müssten mit diesen Einstellungen bei diesem Test an allen ADCs ca. die gleichen Spannungen anliegen. Allerdings ist zu sehen, dass bei dem ADC „I0“ eine viel kleinere Spannung und bei den ADCs „P0“ und „P90“ keine nennenswerten Spannungen anlagen. Zudem ließen sich bei diesen Jumperpositionen die Spannungen einiger ADCs, nachdem sie auf die zu sehenden 2,36 V gesetzt worden sind, nicht mehr regulieren.

Bei einem anderen Test, bei dem nur der Jumper „JP_3V“ (Pin 1 mit Pin 2) gesteckt war und die DACs vom Typ „AD5544“ genauso eingestellt waren wie in Abbildung 42, stimmte die Spannung am ADC „P45“ mit der am ADC „P135“ überein. Zudem war die Spannung am ADC „I0“ genauso groß wie die am ADCs „I3“ und die am ADC „I1“ genauso groß wie die am ADC „I2“. An den ADCs „P0“ und „P90“ lagen erneut keine

nennenswerten Spannungen an. Bei diesem Test ließen sich die Spannungen an den anderen ADCs, nachdem sie hochgesetzt worden sind, weiterhin regulieren.

Die Jumper wurden auf verschiedenste Art und Weise gesteckt. Was auffiel war, dass sich die Spannungen der ADCs mit verschiedenen Jumperpositionen anders verhalten. Lediglich an den ADCs „P0“ und „P90“ ließen sich die Spannungen nie einstellen. Dafür ist der DAC „U$6“ zuständig. Es wird vermutet, dass dieser oder der Verstärker, der mit diesem DAC verschaltet ist, defekt ist.

Es wurden Oszilloskopmessungen an den vom FPGA angesteuerten Pins des DACs „U$6“ durchgeführt. Wenn im GUI eine Einstellung des DACs „U$6“ vorgenommen wird, werden die Signale, die an den Pins anliegen, korrekt gesetzt. Daher und dadurch, dass sich die anderen ADCs vom Typ „AD5544“ regulieren lassen, kann davon ausgegangen werden, dass in der QT-Anwendung und im VHDL-Design keine Fehler vorliegen.

7 Zusammenfassung & Ausblick

In dieser Bachelorarbeit sollte für das Projekt POLDI eine Ansteuerung mehrerer DACs und ADCs, welche sich auf dem Evalboard befinden, entworfen werden.

Das erstellte VHDL-Design ermöglicht, alle DACs anzusteuern. Außerdem lassen sich durch das Design alle ADCs auslesen. Dabei kann entweder ein Auslesevorgang von ausgewählten ADCs gestartet werden, bei dem die ausgelesenen Daten an den PC gesendet werden oder eine kontinuierliche Auslesung aller ADCs stattfinden.

Über das GUI der erstellten QT-Anwendung kann der Benutzer die verschiedenen DACs ansteuern. Zudem lassen sich darüber verschiedene ADCs auswählen, deren anliegende Spannungen im GUI angezeigt wird.

Zusammenfassend lässt sich sagen, dass das Ziel der Bachelorarbeit erfolgreich erfüllt worden ist. Jedoch verhalten sich nicht alle Bauteile auf der Platine so wie gewünscht. Zukünftig sollten der DAC „U$6“ und der mit diesem DAC verbundene Verstärker überprüft werden, da dort wahrscheinlich ein Defekt vorliegt. Die Ansteuerung kann genutzt werden, um das Verhalten auf der Platine zu analysieren.

Da sich mit dem VHDL-Design alle DACs ansteuern und alle ADCs auslesen lassen, kann es für zukünftigen Projekte verwendet werden. Beispielsweise kann es durch eine Komponente, die den Winkel des auf den POLDI einfallenden Lichtes berechnet, erweitert werden. Dafür wurde ein Modus vorgesehen, in dem alle ADCs kontinuierlich ausgelesen werden.

Literaturverzeichnis

[1] *Analog Devices: AD5544 Data Sheet.* (2015). Abgerufen am 19. September 2019 von https://www.analog.com/media/en/technical-documentation/data-sheets/AD5544_5554.pdf

[2] *Analog Devices: AD5686 Data Sheet.* (2017). Abgerufen am 19. September 2019 von https://www.analog.com/media/en/technical-documentation/data-sheets/AD5686_5684.pdf

[3] *Analog Devices: AD7980 Data Sheet.* (2017). Abgerufen am 19. September 2019 von https://www.analog.com/media/en/technical-documentation/data-sheets/AD7980.pdf

[4] Düperthal, J. (2017). *Inbetriebnahme der seriellen Kommunikation zwischen Mojo-Board und CentOS zur späteren Implementierung in POLDI.* Fachhochschule Dortmund.

[5] Gummer, S. (2015). *Entwurf von Signalverarbeitungs-Elektronik und Algorithmen zur hochgenauen Verarbeitung von WInkel-Sensor-Daten.* Universität Paderborn.

[6] Othmane, G. (2018). *Entwicklung einer Ansteuerung eines an einem FPGA angeschlossenen DA-Wandlers unter Nutzung einer QT-Anwendung.* Fachhochschule Dortmund.

Anhang

CD mit folgendem Inhalt:

- Bachelorthesis als PDF
- genutzte Quellen
- VHDL-Design
- QT-Anwendung

Tabelle mit allen Befehlscodes des Kommunikationsprotokolls:

Tabelle 7: alle Befehlscodes des Kommunikationsprotokolls

Befehlscode	ausgewählter DAC	einstellbare Signale
00000000	U$6 (AD5544)	SIN0, SIN90
00000001	U$12 (AD5544)	SIN45, SIN135
00000010	U$26 (AD5544)	VOH0, VOH2
00000011	U$28 (AD5544)	VOH1, VOH3
00000100	U$21 (AD5544)	V_FREE, REF_INTENSITY
00000101	U$7 (AD5544)	SHIELD, VT_INTEN, VT_POLA
Befehlscodes für die kontinuierliche Auslesung aller ADCs ohne die Versendung der ausgelesenen Daten an den PC.		
Befehlscode	Befehl	
10000111	Start der kontinuierlichen Auslesung	
00000111	Stopp der kontinuierlichen Auslesung	
Befehlscodes zur einmaligen Auslesung verschiedener ADCs und anschließenden Versendung der ausgelesenen Daten an den PC. Die Daten werden in der Reihenfolge an den PC gesendet, in der die ADCs und die Signale in der Tabelle angeordnet sind (von links nach rechts).		
Befehlscode	ausgewählte ADCs	Signale, die mit den „V+“-Pins der ADCs verbunden sind
10000110	U$13, U$5	SIN0, SIN90
01000110	U$15, U$14	SIN45, SIN135
11000110	U$13, U$5, U$15, U$14	SIN0, SIN90, SIN45, SIN135
00100110	U$17, U$16	VOH0, VOH2
10100110	U$13, U$5, U$17, U$16	SIN0, SIN90, VOH0, VOH2
01100110	U$15, U$14, U$17, U$16	SIN45, SIN135, VOH0, VOH2
11100110	U$13, U$5, U$15, U$14, U$17, U$16	SIN0, SIN90, SIN45, SIN135, VOH0, VOH2
00010110	U$19, U$18	VOH1, VOH3
10010110	U$13, U$5, U$19, U$18	SIN0, SIN90, VOH1, VOH3
01010110	U$15, U$14, U$19, U$18	SIN45, SIN135, VOH1, VOH3
11010110	U$13, U$5, U$15, U$14, U$19, U$18	SIN0, SIN90, SIN45, SIN135, VOH1, VOH3
00110110	U$17, U$16, U$19, U$18	VOH0, VOH2, VOH1, VOH3
10110110	U$13, U$5, U$17, U$16, U$19, U$18	SIN0, SIN90, VOH0, VOH2, VOH1, VOH3
01110110	U$15, U$14, U$17, U$16, U$19, U$18	SIN45, SIN135, VOH0, VOH2, VOH1, VOH3
11110110	U$13, U$5, U$15, U$14, U$17, U$16, U$19, U$18	SIN0, SIN90, SIN45, SIN135, VOH0, VOH2, VOH1, VOH3