Fachhochschule Dortmund
University of Applied Sciences and Arts

elmos

AUTOMATISIERTE MESSTECHNISCHE CHARAKTERISIERUNG EINES TIME-TO-DIGITAL CONVERTERS FÜR EINE TIME-OF-FLIGHT ANWENDUNG

Bachelorarbeit

Remzi Karaarslan

Oktober 2021

Zur Erlangung des akademischen Grades

Bachelor of Engineering

Erstprüfer: Prof. Dr. -Ing. Michael Athanassios Karagounis
Zweitprüfer: Dr. Wolfram Budde

Fachhochschule Dortmund
Fachbereich Elektrotechnik
In Kooperation mit:
Elmos Semiconductor SE

Kurzzusammenfassung

In dieser Bachelorarbeit wird ein Time-to-Digital Converter für eine Time-of-Flight Anwendung untersucht und durch Messungen charakterisiert. Der Time-to-Digital Converter, der von zwei ehemaligen Studenten der Fachhochschule Dortmund entwickelt wurde, ist dafür zuständig, die temperaturbedingte Änderung der Einschaltverzögerung der Lichtquelle**,** die während den Signalaufnahmen einer Time-of-Flight Kamera entsteht, zu erfassen und die entstandenen Verzögerungen zu korrigieren. Die Aufgabe dieser Arbeit ist es, den entwickelten Testchip mit einer vorentwickelten Platine von Elmos Semiconductor SE zu analysieren und zu überprüfen, ob der Testchip ordnungsgemäß funktioniert. Um den Testchip so aussagekräftig wie möglich zu testen, wird unter Einbindung der Programmiersoftware Qt der Test des TDC automatisiert.

Abstract

In this bachelor thesis a time-to-digital converter for a time-of-flight application is investigated and characterized by measurement. The Time-to-Digital-Converter, which was developed by two former students of the University of Applied Sciences Dortmund, is responsible to detect the temperature induced change of the turn-on delay of the light source, which occurs during the signal recording of a Time-of-Flight camera, and to correct the resulting delays. The task of this work is to analyze the developed test chip with a pre-developed circuit board from Elmos Semiconductor SE, and to verify the proper function of the test chip. In order to test the test chip as meaningful as possible, the testing of the TDC is automated with the integration of the programming software Qt.

Selbständigkeitserklärung

Hiermit versichere ich, 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.

Herne, 11.10.2021 

Ort, Datum Unterschrift

Inhalt

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

CSV	Comma-Separated Values
DL	Delay-Line
DLL	Delay-Locked-Loop
FIFO	First in, First out
GUI	Graphical User Interface
LSB	Least Significant Bit
MPSSE	Multi Protocol Synchronous Serial Engine
SCPI	Standard Commands for Programmable Instruments
TDC	Time-to-Digital Converter
TOF	Time-of-Flight
VCP	Virtual Com Port

1 Einleitung

Die Halbleiterindustrie erlebt seit 50 Jahren einen großen Aufschwung. Die Fortschritte in dieser Branche tragen dazu bei, die Lebensqualität der Menschen zu steigern. Diese Innovationen sind im alltäglichen Leben im digitalen Zeitalter nicht wegzudenken. In dieser Bachelorarbeit geht es um das Thema „Automatisierte messtechnische Charakterisierung eines Time-to-Digital Converters für eine Time-of-Flight Anwendung“. Der Time-to-Digital Converter (TDC) erfasst Zeitintervalle und gibt sie als digitale Repräsentation wieder. Er ist dafür zuständig, die Verzögerungen, die während der Signalverarbeitung in einer Time-of-Flight Anwendung (TOF) entstehen, zu korrigieren.

In dieser Arbeit wird der Chip zunächst manuell auf seine Eigenschaften hin untersucht und getestet. Anschließend soll eine Softwareumgebung zum automatisierten Test des TDC entwickelt werden. Dabei wird eine Verbindung zwischen einem Linux Rechner und dem FT2232H-56Q Mini Modul über USB mit der Programmierumgebung Qt-Creator erreicht. Zwei Waveform-Generatoren der Firma *Keysight Technologies* und ein Multimeter der Firma *Keithley Instruments* werden in diesem Projekt über USB mit dem Linux Rechner verbunden und über SCPI Befehle angesteuert.

Die Testumgebung wird in C++ programmiert und mit Hilfe einer grafischen Benutzeroberfläche (GUI, engl.: Graphical User Interface) visualisiert. Die Visualisierung in einer GUI erleichtert es dem Anwender, die Testumgebung zu bedienen.

Das Ziel dieses Projektes ist es, den entwickelten TDC-Chip für eine Time-of-Flight-Anwendung hinsichtlich ihrer Eigenschaften zu untersuchen.

Diese Bachelorarbeit wird in Kooperation mit der Firma *Elmos Semiconductor SE* im Labor für integrierten Schaltungsentwurf an der Fachhochschule Dortmund durchgeführt und soll in der Zukunft auch ermöglichen, weitere entwickelte TDC mit dieser Testumgebung messtechnisch zu untersuchen.

2 Time-of-Flight Kamera

Die Time-of-Flight-Technologie (TOF) wird für die 3D-Bildgebung verwendet. Sie wird z.B. im Fahrzeug eingesetzt, um mit Gesten die Entertainment-Anwendung und andere Komfortfunktionen zu steuern oder außerhalb des Fahrzeugs Hindernisse zu erkennen. Durch Verwendung einer Lichtquelle und eines Sensors wird Mithilfe der Lichtgeschwindigkeit die Distanz zu detektierten Objekten bestimmt. Der Abstand L wird durch die Formel berechnet:

$$L = \frac{1}{2} * c_0 * \Delta T \tag{1}$$

Die Konstante c_0 steht für die Lichtgeschwindigkeit und ΔT für die gemessene Laufzeit des Lichtpulses. Da der Lichtstrahl die doppelte Strecke L absolviert, wird die Laufzeit durch den Faktor 2 geteilt.

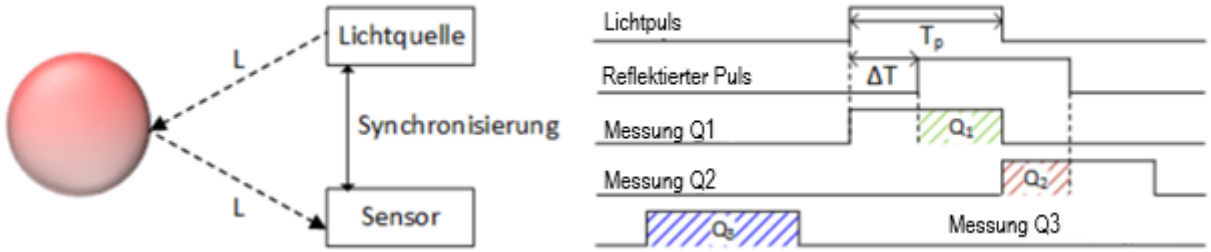

Abbildung 1: Time-of-Flight Prinzip[1] [2]

Der Sensor besteht aus einem Array von lichtempfindlichen Pixelelementen, die das empfangene Licht in elektrische Signale umwandeln. Jedes einzelne dieser Pixelelemente wird ausgelesen und daraus ergibt sich ein dreidimensionales Abstandsbild. Die Abbildung 1 zeigt, dass zeitgleich zum Laserpuls Tp die Messung Q1 gestartet wird. Wenn während dieser Messung das reflektierte Licht auf den Sensor trifft, wird die Ladungsmenge Q1 generiert. Im Anschluss wird in einer zweiten Messung die Ladungsmenge Q2 aufgenommen.

Da man während den Messungen auch den Einfluss des Umgebungslichtes berücksichtigen muss, wird eine zusätzliche Messung Q3 durchgeführt. Diese

[1] [1]
[2] [2]

Messung wird bei ausgeschalteter Lichtquelle durchgeführt. Die Ladungsmenge Q3 wird von jeder Teilladung abgezogen. Daraus folgt die Gleichung:

$$\frac{\Delta T}{T_p} = \frac{Q2 - Q3}{Q1 + Q2 - 2 * Q3} \tag{2}$$

Wenn die Gleichung nach ΔT umgeformt wird und in die Gleichung 2 eingesetzt wird ergibt sich:

$$L = \frac{1}{2} * c_0 * T_p * \frac{Q2 - Q3}{Q1 + Q2 - 2 * Q3} \tag{3}$$

Die Zuverlässigkeit des oben beschriebenen Messverfahrens ist von der Synchronität des Lichtimpulses und des Signals zur Aktivierung der Sensoren, d.h. des Shutters, abhängig. Wenn einer der beiden Signale verzögert auftritt, führt dies zu Messfehlern. Zudem hat die Lichtquelle auch eine Einschaltverzögerung, welche mit der Temperatur variiert. Aufgrund der hohen Lichtgeschwindigkeit können bereits Messfehler im Nanosekundenbereich zu Messfehlern im Zentimeterbereich führen.

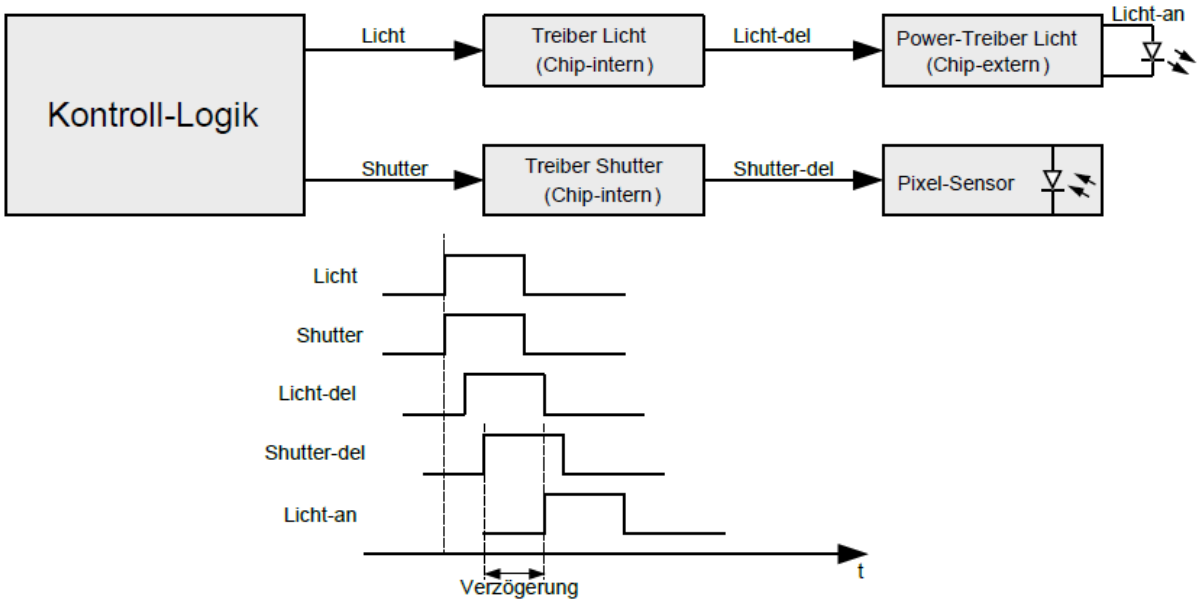

Abbildung 2: Laser- und Shuttersignal[3]

Abbildung 2 zeigt die Kontrolllogik des TOF 3D Kamerasystems, die gleichzeitig das Licht- und das Shuttersignal startet. Diese Signale werden durch interne Chiptreiber

[3] [3]

verstärkt. Für die Lichtquelle werden weitere externe Power-Treiber benutzt. Diese Treiber führen zu weiteren Verzögerungen. Das führt dazu, dass die Lichtsteuerung und Shuttersignale nicht synchron liegen. Der TDC soll die Verzögerungen zwischen den zwei Signalen messen und daraus eine digitale Repräsentation erzeugen, damit die Verzögerung durch eine nachgelagerte Reglereinheit ausgeglichen werden kann.

3 Time-to-Digital Converter - Theoretische Grundlagen

Oft muss man in der Sensorik physikalische Größen in einen digitalen Wert umwandeln. Die zu messenden Sensorsignale werden zunächst als analoge Größen z.B. als Spannung oder Zeit erfasst. Analog-Digital-Wandler nutzen die analogen Größen und ändern sie in eine digitale Darstellung. TDC werden oft verwendet, um Sensorsignale als zeitäquivalente Signale wiederzugeben. Primär werden TDC jedoch benutzt, um erfasste Zeitintervalle in einer digitalen Repräsentation wiederzugeben.

Zur Quantisierung der Zeitintervalle wird ein Referenztakt in die Schaltung eingeführt. Das Zeitintervall wird, wie in der Abbildung 3 zu sehen ist, durch ein Start- und ein Stoppsignal begrenzt.

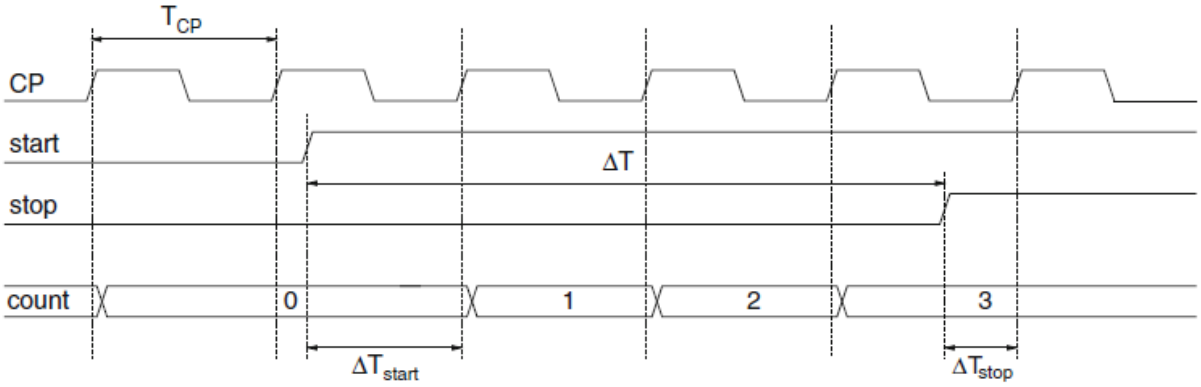

Abbildung 3: Prinzip der zählerbasierten TDC[4]

$$\Delta T = N * T_{CP} + \left(T_{CP} - \Delta T_{stop}\right) - \left(T_{CP} - \Delta T_{start}\right) \quad (4)$$

[4] [4] S.12

Das Messintervall ΔT wird durch die Anzahl N der Taktperioden, die in das Zeitintervall fallen, beschrieben. T_{CP} gibt die Referenztaktperiode an. ΔT_{start} und ΔT_{stop} sind die Zeitintervalle zwischen den Start- und Stoppsignalen und der nächsten ansteigenden Flanke des Taktsignals. Die Genauigkeit der Messungen kann durch Erhöhung der Clock-Frequenz gewährleistet werden. Der Entwurfs- und Kostenaufwand skaliert jedoch mit der Frequenz des Referenztaktes und ist ab einer bestimmten Höhe nicht mehr vertretbar. Aus diesem Grund werden oft Schaltungskonzepte verwendet, die eine gute Auflösung auch bei moderaten Referenztaktfrequenzen ermöglichen.

3.1 Delay-Line basierte TDC

Um die Messauflösung über die maximal realisierbare Taktfrequenz hinaus zu erhöhen, wird eine Kette aus identischen Verzögerungselementen (engl. Delay-Line) verwendet, welche phasenverschobene Duplikate des Referenztaktes erzeugen. Abbildung 4 zeigt den Aufbau eines Delay-Line basierten TDC.

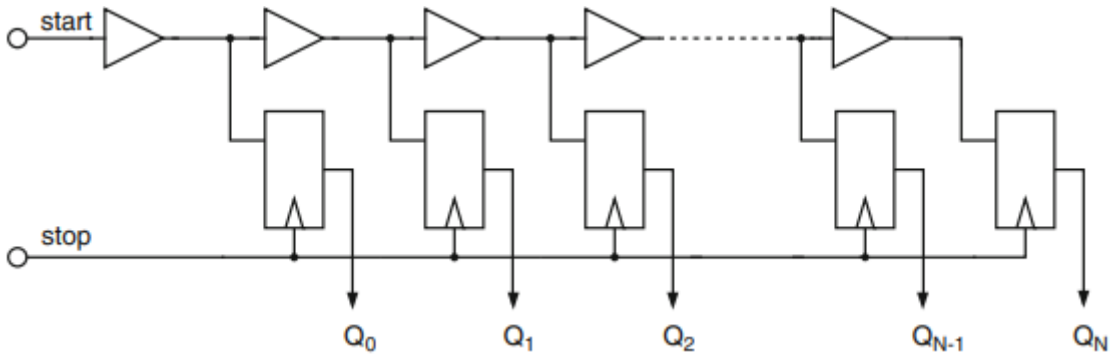

Abbildung 4: Aufbau eines Delay-Line basierten TDC[5]

Das Startsignal wird durch die Delay-Elemente verzögert und nach jedem Element als Eingangssignal der Flipflops benutzt. Das Stoppsignal wird als Taktsignal der flankengesteuerten Speicherelemente verwendet. Trifft das Stoppsignal ein, wird bei allen Flipflops, bei denen bereits ein Startsignal anliegt, automatisch ein High-Pegel gespeichert. Die Ausgänge der übrigen Elemente bleiben auf LOW. Hierdurch entsteht ein Thermometer-Code, welcher die Anzahl der durchlaufenen Delay-Elemente angibt. Das erfasste Zeitintervall kann mit der folgenden Gleichung dargestellt werden:

$$\Delta T = N * t_{Stufe} + \varepsilon \tag{5}$$

[5] [4] S.10

N steht dabei für die Anzahl der Flipflops mit logischem High-Signal und t_{Stufe} ist die Verzögerungszeit eines einzelnen Delay-Elementes. Dazu muss noch ein Quantisierungsfehler ε betrachtet werden, weil die Speicherelemente sich nur im High- oder Low-Zustand befinden und Zwischenstufen nicht erkannt werden können.

In der in diesem Projekt eingesetzten TDC entsteht ein Bitmuster von 650Bits. Da in diesem Projekt die Verzögerung der Delay-Line auf 25ns eingeregelt ist, kann die Verzögerung pro durchlaufene Stufe mit der Gleichung 5 ermittelt werden.

$$t_{Stufe} = \frac{25ns}{650\ Bits} \tag{6}$$

Als t_{Stufe} erhält man das Ergebnis 38,5ps. Die Anzahl der durchlaufenen Delay-Elemente, welche der Anzahl der gesetzten Flip-Flops entspricht, wird mit der Zeit t_{Stufe} multipliziert und entspricht der Zeitverzögerung zwischen dem Start- und dem Stoppsignal.

Die Delay-Line besteht aus Invertern oder aus Bufferstufen. Die Bufferstufen wiederum bestehen aus zwei CMOS-Invertern, wobei inverter-basierte Delay-Lines meist eine höhere Auflösung als Buffer basierte Delay-Lines erreichen.

3.2 Funktionsprinzip des TDC

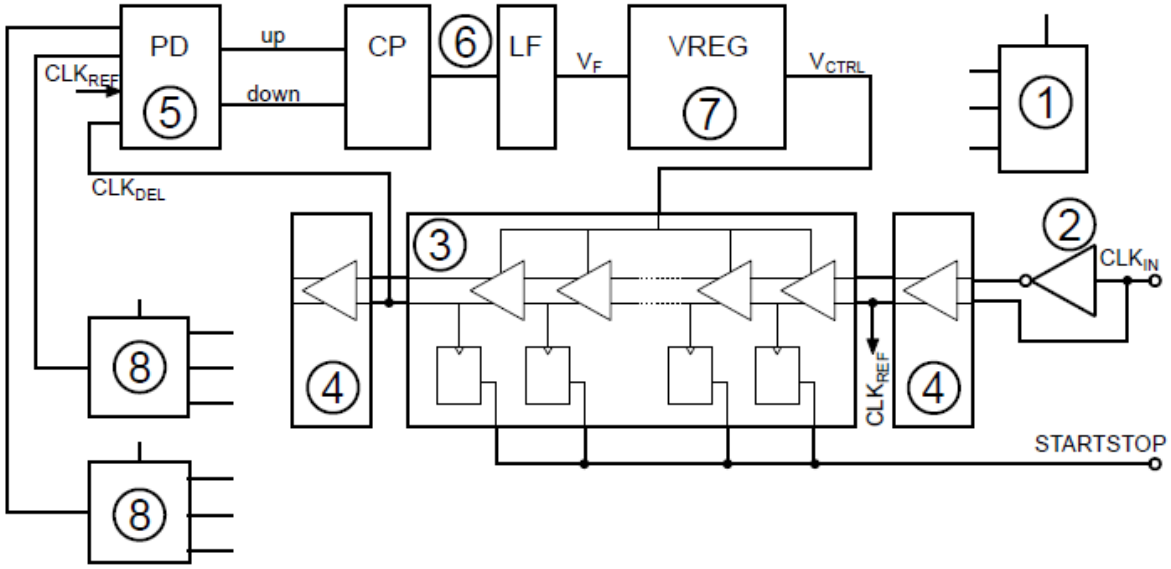

Abbildung 5: Blockdiagramm des TDC[6]

Für den TDC wird ein Delay-Locked-Loop (DLL) verwendet. Die DLL ermöglicht die Korrektur der Phasenverschiebung zwischen dem angelegten Referenzsignal und dem verzögerten Ausgangssignal. Temperatur- und Spannungs-schwankungen werden ebenfalls mit diesem System ausgeregelt.

Die Abbildung 5 zeigt ein Blockdiagramm des Time-to-Digital Converters mit den einzelnen Komponenten. Der entwickelte TDC ist ein Local-Passive-Interpolation TDC. Hier wird eine Delay-Line mit Inverter-Verzögerungselementen benutzt. Zwischen den Ein- und Ausgängen werden Interpolationselemente eingefügt.

Zur Erzeugung der 3,3V Versorgungsspannung der digitalen und analogen Schaltungsteile wird eine Schaltung von Elmos Semiconductor SE zur Verfügung gestellt (1).

Abweichend von dem konventionellen Ansatz aus Abschnitt 3.1 wird in diesem Entwurf das Taktsignal mit 40MHz als Referenzsignal in die Delay-Line eingeführt (2). Das verzögerte Taktsignal, wird an den Dateneingang des jeweiligen Flip-Flops geführt. Die Start-Stop-Signale werden auf einer einzelnen Leitung zum Signal STARTSTOP zusammengefasst und an die Dateneingänge der Flip-Flops geführt. Während der Zeitmessung nimmt das STARTSTOP Signal den Wert 1 an. Alle Flip-

[6] [3]

Flops, die das verzögerte Taktsignal aus der Delay-Line während dieser Zeit erreichen, speichern dann die anliegende logische 1. Nach Ende der Zeitmessung geht das STARTSTOP auf den Wert Null zurück. Alle Flip-Flops, die nach Ende der Messzeit aus der Delay-Line getaktet werden, behalten den logischen Wert 0. Die Delay-Line besteht in diesem Fall aus einer Aneinanderreihung differentieller Inverter, die das eingehende Taktsignal um eine bestimmte Zeit, die eingeregelt wird, verzögert. Das eingehende Taktsignal durchläuft die Verzögerungselemente und wird anschließend zum Phasendetektor geführt. Die gleichmäßige Belastung der Delay-Zellen wird durch Dummy-Delay-Elemente (4) am Anfang und am Ende der Delay-Line gewährleistet.

Der Phasendetektor vergleicht die Phasenlage der anliegenden Signale mithilfe zweier D-Flipflops und einem AND-Gatter. Dieser Aufbau wird auch Positive-Edge-Triggered (PET) genannt, weil er auf die positiven Flanken der Eingangssignale reagiert. Die beiden Zeitdiagramme in der Abbildung 6 veranschaulichen den Ablauf der Signale. Das obere Diagramm zeigt, dass der Referenztakt dem verzögerten Signal nacheilt, weil die positive Flanke vor dem Referenztakt erscheint. Der DOWN-Puls wird so lange auf HIGH gesetzt bis der Referenztakt erscheint. Sobald die steigende Flanke des Referenztaktsignals im zweiten Flipflop erscheint, wird UP auf HIGH geschaltet und dabei wird ein RESET ausgelöst, das die Ausgänge der Flipflops zurücksetzt. Beim unteren Diagramm ist der Referenztakt voreilend, wodurch ein UP-Puls generiert wird, bis die steigende Flanke des verzögerten Signals eintrifft und ein DOWN Puls generiert wird.

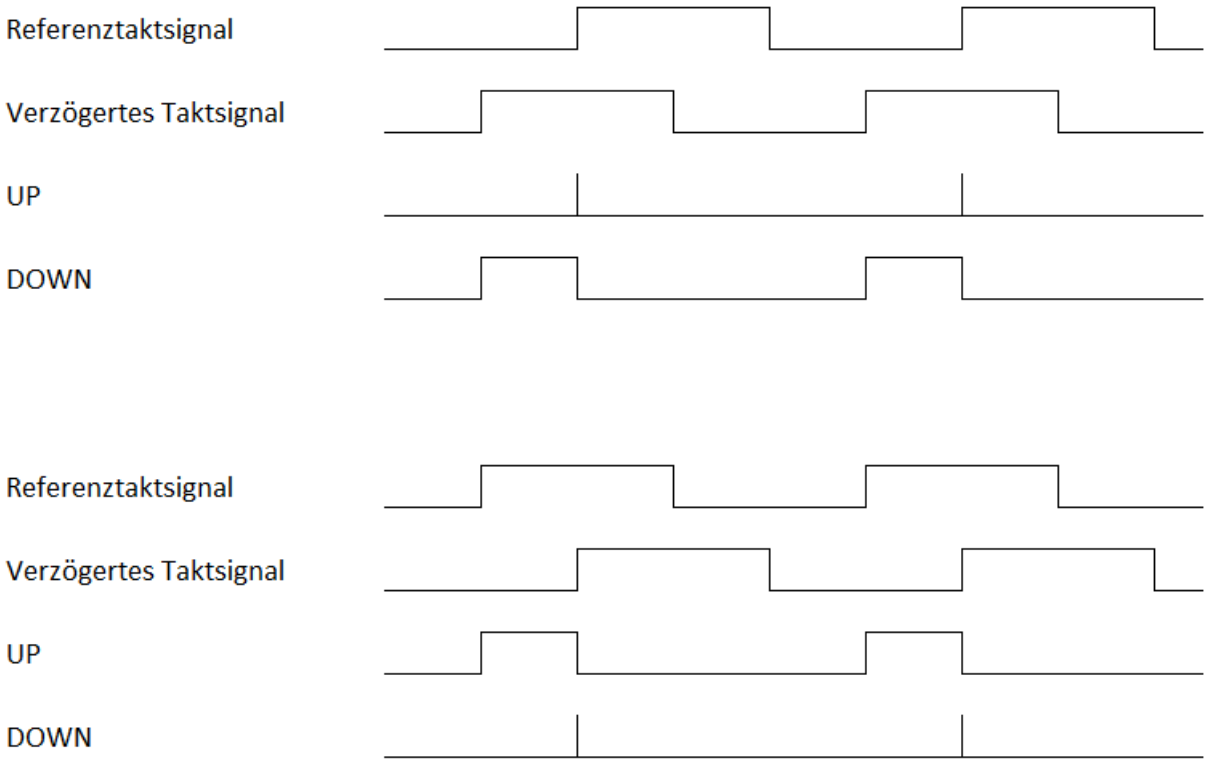

Abbildung 6: Signalverlauf des Phasendetektor mit unterschiedlichen Phasenlängen

Der Phasendetektor vergleicht das eingehende Taktsignal und das verzögerte Taktsignal. Es erzeugt dementsprechend Up- oder Down-Pulse. Diese Pulse werden einer Charge-Pump (6) zugeführt, welche die Filterspannung beeinflussen. Je nachdem, ob UP- oder Down-Signale vom Phasendetektor generiert werden, hat die Charge Pump die Aufgabe, Strom zuzuführen oder zu entziehen. Die Filterspannung Vf wird als Referenzspannung des Linearreglers (7) verwendet, welcher die Kontrollspannung VCTRL erzeugt, mit der die Delay-Line versorgt wird.

Dies wiederum führt dazu, dass die Kontrollspannung bzw. die Versorgungsspannung der Delay-Line erhöht oder reduziert wird und dadurch die Verzögerung entweder verkürzt oder vergrößert wird bzw. umgekehrt. Schließlich wird die Delay-Line auf eine Verzögerung von 25ns geregelt.

False- und Harmonic-Lock Detektoren (8) sollen das fehlerhafte Einschwingen des TDC verhindern.

Das Zeitintervall, das zwischen dem Start- und dem Stoppsignal liegt und welches der Verzögerung zwischen der Lichtsteuerung und den Shuttersignalen entspricht, soll gemessen werden. In dieser Arbeit werden die Start- und Stoppsignale durch einen Waveform-Generator generiert.

3.3 Aufbau der Verzögerungselemente des Testchips

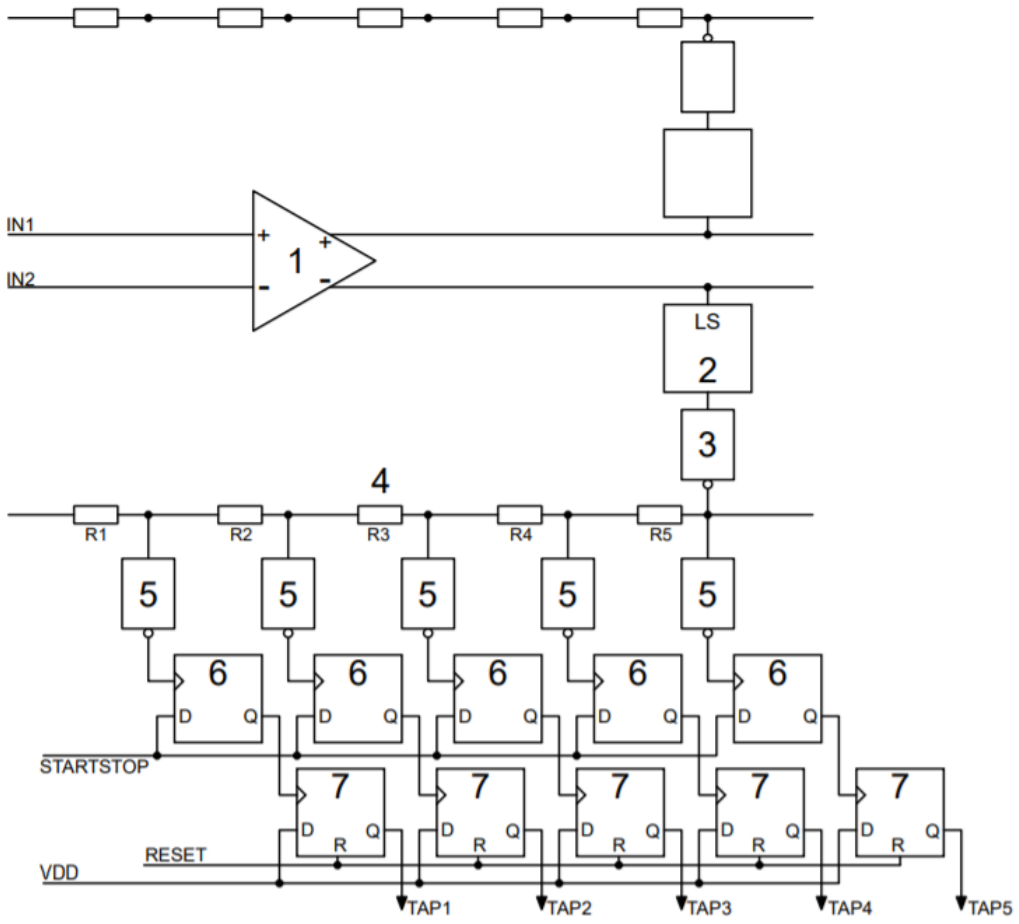

Abbildung 7: Aufbau eines Verzögerungselements[7]

Die Abbildung 7 zeigt den Aufbau eines Verzögerungselements. Die Kontrollspannung dient als Versorgungspannung für die Inverterkette und somit sind die Ausgangspannungen der Inverter (1) auf den Wert der Kontrollspannung begrenzt. Für die weitere Nutzung wird der Signalpegel durch den Inverter-Level-Shifter (2) auf 3.3V erhöht. Eine starke Bufferstufe (3) treibt die Interpolationswiderstände (4) und die damit verbundenen Ausgangs-Bufferstufen (5).

Die Takteingänge der D-Flipflops (6) sind über die Bufferstufen mit den Ausgängen der Interpolationswiderstände verbunden. Alle Flip-Flops, bei denen die steigende Flanke des verzögerten Taktsignals aus der Delay-Line am Takteingang erscheint, während das STARTSTOP Signal an den Flip-Flop Dateneingängen gesetzt ist, übernehmen den High-Zustand und setzen ihn auf ihren Datenausgang. Flip-Flops,

[7] [3] S.29

die vom verzögerten Taktsignal aus der Delay-Line erreicht werden, während das STARTSTOP Signal nicht gesetzt ist, behalten den Wert Low am Datenausgang. Dadurch wird ein Thermometer-Code erzeugt, der die Länge des gemessenen Zeitintervalls repräsentiert. Die Dateneingänge der Flipflops in der zweiten Reihe (7) sind mit der Versorgungsspannung verbunden und die Ausgänge der Flipflops in der ersten Reihe sind mit den Takteingängen der Flipflops (7) verbunden. Um die Flipflops zurückzusetzen, wird ein Resetsignal verwendet.

3.4 Verlauf der Signale des TDC-Chips

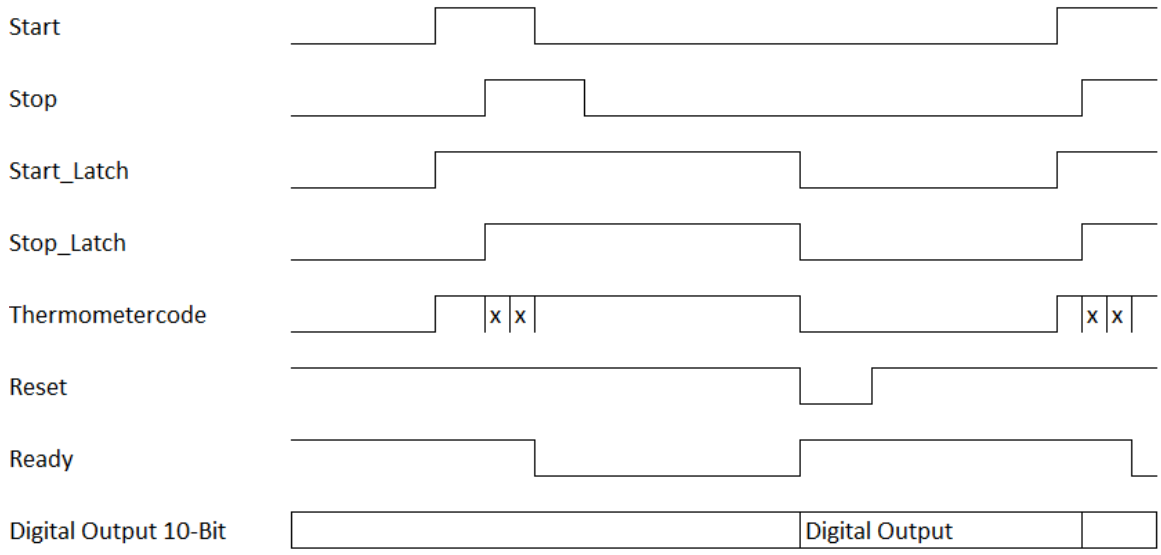

Abbildung 8: Timing-Diagramm der Signale

Das Timing-Diagramm in der Abbildung 8 zeigt den genauen zeitlichen Ablauf eines Messvorgangs im TDC-Testchip. Der Systemtakt der Delay-Line beträgt 40MHz. Die Umsetzung des Thermometer-Codes in eine binäre Darstellung durch Addition erfolgt mit einer 10MHz Clock. Die Messung startet asynchron mit den Signalen START und STOP und wird in zwei Latches als Signale START_LATCH und STOP_LATCH gespeichert. Hierbei ist es wichtig, dass STOP nach START aktiv wird, da sich sonst kein gültiger Messablauf ergibt. Die Ermittlung des Thermometercodes startet mit der steigenden Flanke des Startsignals, während die Auswertung des Codes mit der steigenden Flanke des Stoppsignals beginnt und insgesamt 650 Takte des 10MHz Takts, also 65µs in Anspruch nimmt. Das READY-Signal zeigt die Vollendung der Auswertung des Thermometercodes an und wird

zwei Takte nach der steigenden Flanke des Stoppsignals auf LOW bzw. 0 gesetzt und bleibt für 65µs auf Null. Nach Ablauf der Auswertung wird das Readysignal auf HIGH bzw. 1 gesetzt und der Binär-Code ausgegeben. Synchron dazu wird das Signal RESET_FF auf Null gesetzt, was dazu führt, dass alle Flipflops in der Delay-Line und die START_LATCH und STOP_LATCH Signale zurückgesetzt werden. Das Zurücksetzen der Signale dauert 100ns und erst nach Ablauf dieser 100ns kann die nächste Messung gestartet werden.

4 Messung des Time-to-Digital-Converters

Folgende Messgeräte bzw. Waveform-Generatoren wurden zur manuellen Untersuchung des Time-to-Digital-Converters verwendet:

Anzahl	**Bezeichnung**	**Hersteller**	**Typ**
1x	Waveform Generator	Keysight	33622A
1x	Waveform Generator	Siglent	SDG 2042X
1x	Oszilloskop	Rohde & Schwarz	RTB 2004
2x	Logikanalysator	Rohde & Schwarz	RT-ZL03
1x	Labornetzteil	Rohde & Schwarz	HMC 8043
1x	Multimeter	Metrahit	2+ Multimeter
1x	Multimeter	Keithley	DMM6500 6 1/2

Tabelle 1: Verwendete Laborgeräte für die Messungen

Die beiden Waveform-Generatoren werden dazu verwendet, um Rechtecksignale zu erzeugen. In diesem Projekt wird ein Waveform-Generator für das Clocksignal verwendet und ein anderer Waveform-Generator für die Start- und Stoppsignale. Mit einem Oszilloskop und zwei Logikanalysatoren werden die Signalverläufe und die digitalen Ausgänge des TDC betrachtet. Eine Spannungsquelle versorgt den TDC mit den benötigten Spannungen. Durch die beiden Multimeter kann die ausgegebene Spannung des TDC ausgewertet werden.

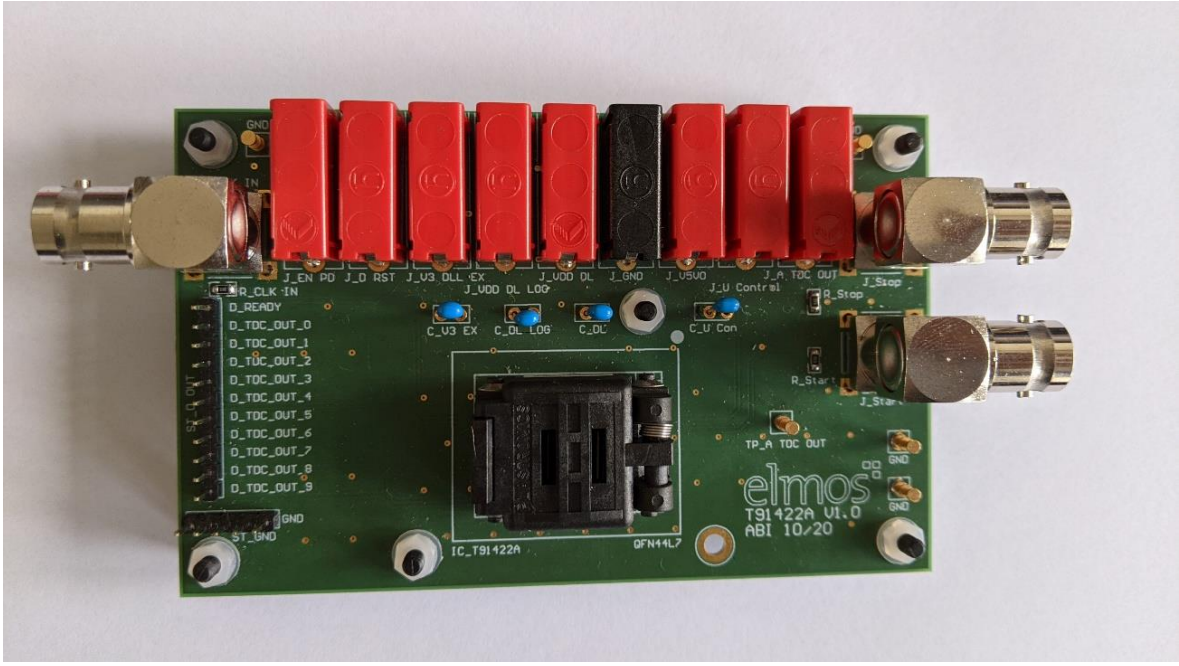

Abbildung 9: Leiterplatte der Firme ELMOS SE für den Testchip

Abbildung 9 zeigt die von Elmos entwickelte Platine für den Testchip. Der Chip wird mit 5V versorgt. Zudem werden ein Enable- und ein Reset-Signal mit 3.3V Pegel benötigt. Das Enablesignal ist für die Ein- und Ausschaltung des Phasendetektors zuständig. Mit dem Reseteingang werden die Operationen, die in der Delay-Line stattfinden, zurückgesetzt, wobei dieser Eingang Active-Low ist. Als Clocksignal wird ein Rechtecksignal mit 40MHz und 3.3 Volt Pegel generiert und die Start- und Stoppsignale werden mit 10kHz und 3.3V Pegeln betrieben. Diese werden mit einer manuell eingestellten Verzögerung zwischen 0-25ns generiert.

Abbildung 10: Start- und Stopp-Puls mit 10ns Verzögerung

In Abbildung 10 werden die beiden Start- und Stopp-Pulse mit 10ns Verzögerung dargestellt.

Am Anschluss J_U Control kann die Versorgungs- bzw. die Kontrollspannung der Delay-Line abgegriffen werden. Diese sollte im Bereich zwischen 2V-3.6V liegen. Die analoge Repräsentation der Messergebnisse kann durch TDC_OUT ermittelt werden. Der TDC_OUT Ausgang liefert einen Spannungswert, welcher je nach Verzögerung unterschiedlich groß ist. Auf der linken Seite der Leiterplatte befinden sich digitale 10Bit-Ausgänge. Diese zeigen die digitale Repräsentation der Messergebnisse. Wenn das D_READY Signal aktiv ist, wird signalisiert, dass die nächste Messung durchgeführt werden kann. Die analoge und digitale Repräsentation der Ergebnisse werden im weiteren Verlauf dieser Arbeit näher erläutert.

Delay [ns]	TDC_OUT [V] Start 150cm & Stop 100cm	TDC_OUT[V]Start 160cm & Stop 100cm	TDC_OUT [V]Start &Stop 100cm
0	0,0135	0,0137	0,0122
1	0,0135	0,0137	0,0122
2	0,0135	0,0137	0,062
3	0,0135	0,0137	0,309
4	0,0183	0,0137	0,517
5	0,095	0,031	0,688
6	0,438	0,283	0,824
7	0,6	0,5	0,94
8	0,74	0,67	1,07
9	0,88	0,8	1,17
10	0,99	0,927	1,3
11	1,11	1,04	1,44
12	1,25	1,169	1,56
13	1,38	1,29	1,67
14	1,51	1,42	1,8
15	1,64	1,56	1,92
16	1,751	1,67	2,024
17	1,86	1,79	2,156
18	2	1,92	2,26
19	2,1	2,04	2,394
20	2,22	2,15	2,54
21	2,36	2,3	2,64
22	2,49	2,4	2,77
23	2,6	2,519	2,9
24	2,72	2,654	3,01
25	2,83	2,755	3,09

Tabelle 2: Einfluss der Kabellänge auf die Messungen

Tabelle 2 zeigt die aufgenommenen Messergebnisse bei Verwendung des analogen Ausgangssignals [TDC_OUT]. Dabei werden Start/Stopp-Verzögerungen von 0ns -25ns an den TDC angelegt. Verzögerung bedeutet, dass das Stoppsignal nach einer bestimmten Zeit nach der steigenden Flanke des Startsignals gesetzt wird. Die Pulsbreite beträgt hierbei 100ns und die steigende Flanke 8.4ns. Außerdem ist es wichtig, die gleiche BNC-Kabellänge für die Start- und Stoppsignale zu verwenden, da sonst wegen der unterschiedlichen Kabellängen durch die variierende Laufzeit der Signale auf den Leitungen, die Zeit zwischen dem Start- und dem Stoppsignal verfälscht wird.

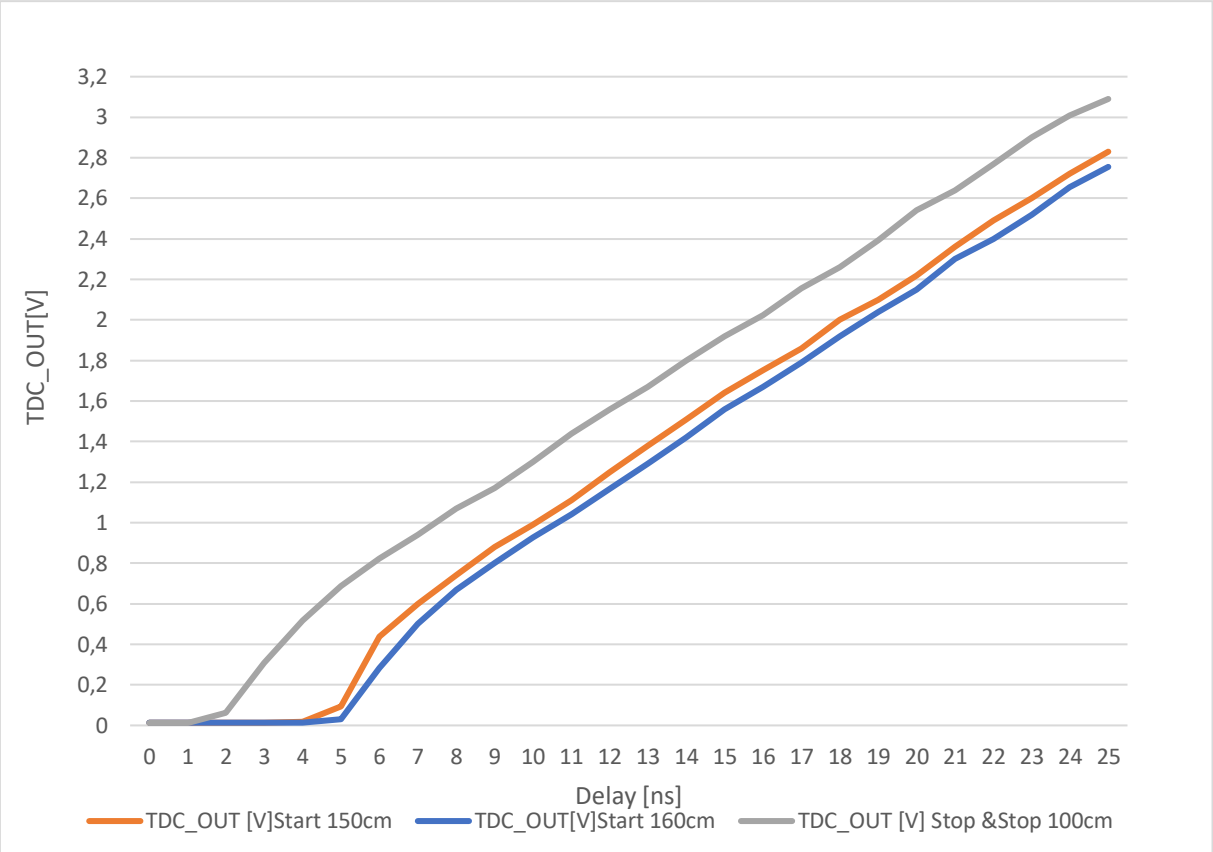

Abbildung 11: Diagramm der Messungen mit dem Einfluss der Kabellänge

Diagramm 11 zeigt, dass die Spannung, welche der analogen Repräsentation des TDC Wandlungsergebnisses entspricht, mit zunehmender Start/Stopp-Verzögerung größer wird. Allerdings ist die Kennlinie nicht linear. Erst ab einer Verzögerung von 2ns reagiert der TDC und das Wandlungsergebnis steigt monoton an. Die graue Linie ist am aussagekräftigsten, da bei dieser Messung die Kabellänge für die Start- und Stoppsignale gleich lang gewählt worden ist (100cm). Die orangene und die blaue Linie machen die Verfälschung der Ergebnisse deutlich, die sich durch ungleich lange Leitungen ergibt. Die Länge der Start-Kabellänge ist für die orangene Linie 150cm und für die blaue Linie 160cm. Die Stopp-Kabellänge ist bei diesen Messungen bei 100cm geblieben. Die Verzögerungswerte für Start/Stopp werden bei diesen beiden Messungen erst ab einer Verzögerung von ca. 4ns wahrgenommen. Auffällig ist auch, dass bei einer Verzögerung von 25ns die endgültige Spannung von ca. 3,1V bei beiden Messungen nicht erreicht wird. Die Startsignale kommen aufgrund der längeren Kabellänge verzögert am TDC an.

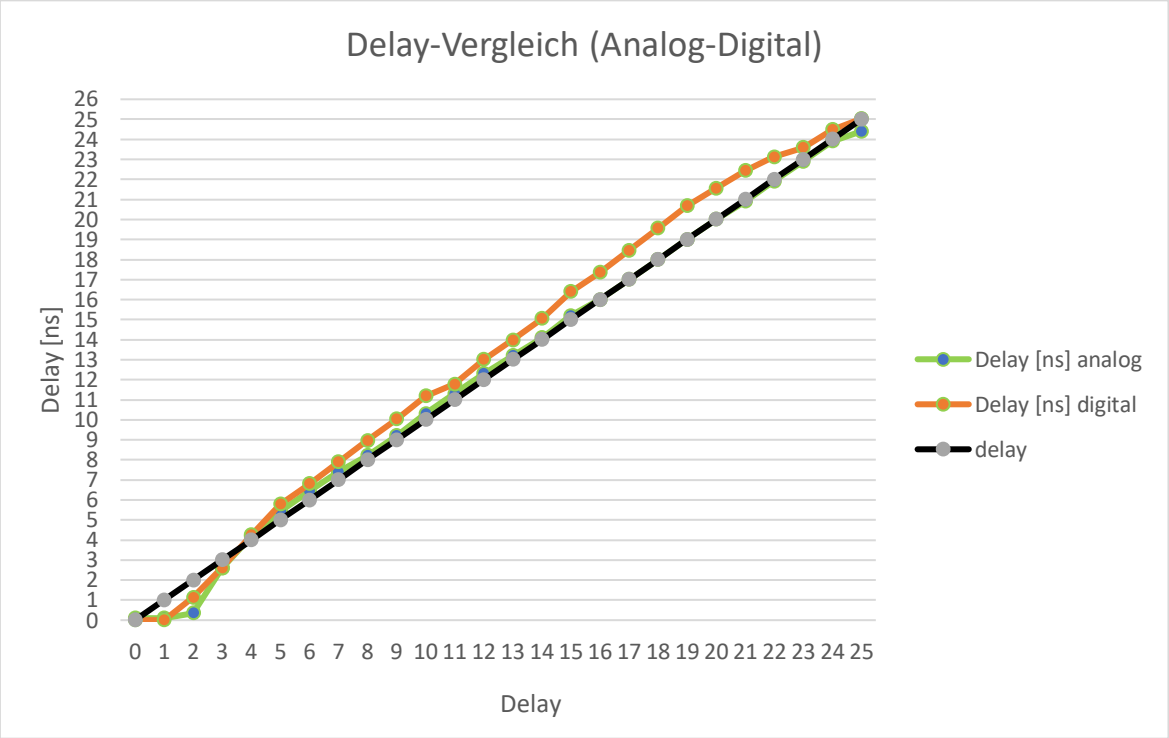

Abbildung 12: Vergleich der digitalen und der analogen Verzögerung mit der manuell eingestellten Verzögerung

Diagramm 12 zeigt die analoge und digitale Repräsentation der Verzögerung. Die schwarze Linie ist die eingestellte Verzögerung, die grüne Linie ist die analoge Verzögerung und die orangene Linie ist die digitale Verzögerung. Die analoge Verzögerung wird mit Hilfe der Gleichung 6 ermittelt:

$$T_{an} = \frac{V_{TDC}}{V_{LSB}} * t_{stufe} \tag{9}$$

V_LSB wird durch Gleichung 7 berechnet.

$$V_{LSB} = \frac{5V}{2^{10}} \tag{10}$$

Bei der Auswertung der digitalen Messwerte muss mit Formel 8 gerechnet werden.

$$T_{dig} = t_{stufe} * Dezimalwert \tag{11}$$

Der TDC liefert einen Code mit 650Bits an den Digitalteil. Jede 1 in diesem Code steht für die Verzögerung einer Stufe im TDC. Das TDC-Wandlungsergebnis wird dann im Digitalteil in einen Binärcode umgewandelt. Die dezimale Repräsentation dieses Wertes wird mit 38,5ps multipliziert, um die Verzögerungszeit zu ermitteln. Eine Verzögerung beträgt 38,5ps, da der TDC 650 Stufen und eine Gesamtverzögerung von 25ns besitzt (siehe Gleichung 5).

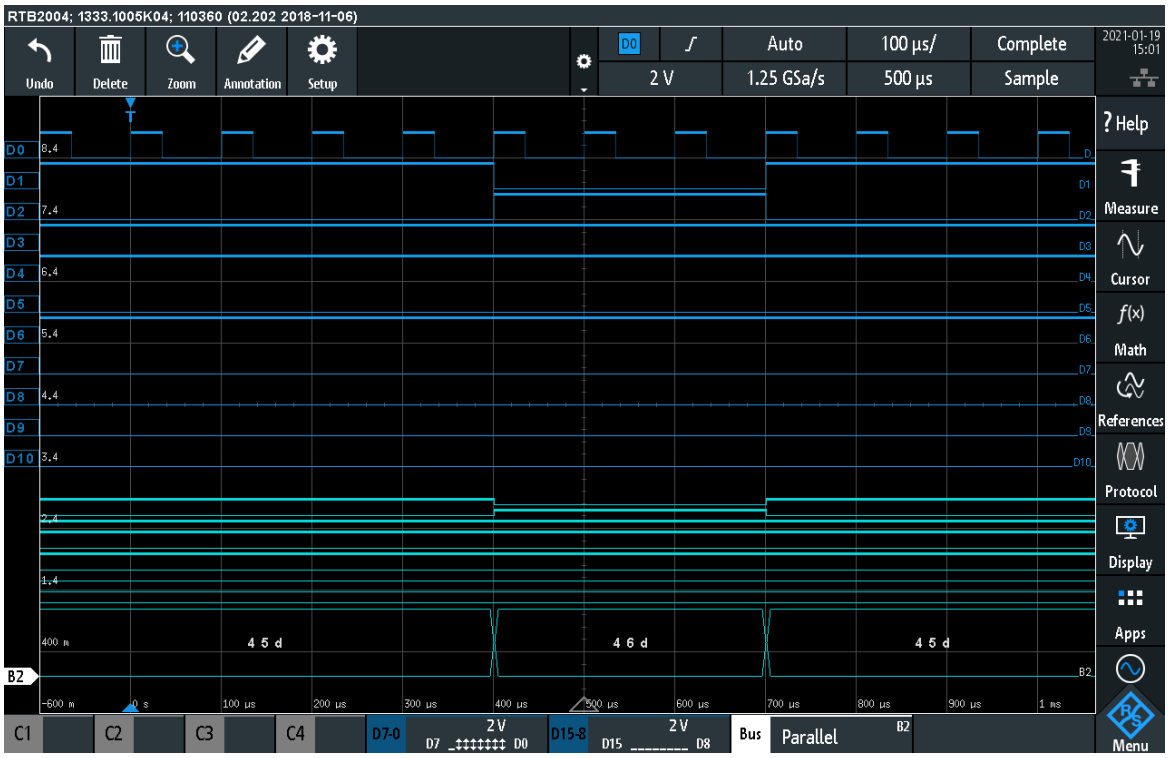

Abbildung 13: Screenshot des Oszilloskops für eine Verzögerung von 5ns

Die digitale Verzögerungsrepräsentation des TDC Wandlungsergebnisses wurde in Abbildung 13 mit einem Logikanalysator aufgenommen, der die aufgenommenen Binärzahlen automatisch in einen Dezimalwert wandelt und am unteren Bildschirmrand anzeigt. Die auf diese Weise aufgenommenen Werte weisen aber Schwankungen von +/- 40 LSB bei konstanter Start/Stopp-Zeit auf. Es wurde vermutet, dass diese Messfehler durch den Jitter der Signalgeneratoren ausgelöst wurden. Ein Jitter entspricht einem Phasenrauschen der generierten Signale, was zu einer zufälligen Verlängerung oder Verkürzung der Steuersignale führt. Die Abbildung 13 zeigt einen Screenshot des Oszilloskops, während eine Messzeit von 2ns am digitalen Ausgang mit dem Logikanalysatoren ausgewertet wird. Als Dezimalwert wird die Zahl 46 angezeigt. Setzt man diesen Wert in die Gleichung 8 ein, resultiert das Ergebnis der Verzögerung zu $Tdig = 1{,}77\text{ns}$. Durch den Jitter

könnte es sein, dass nicht der gewünschte Wert von 2ns erreicht wird. Die orangene Linie in der Abbildung 27 steht für die Messung der Verzögerungszeit bei Verwendung der digitalen Schnittstelle. Diese Linie weicht aufgrund der Messfehler stark von der manuell eingestellten Verzögerung ab.

Delay [ns]	TDC_OUT[V]	Delay [ns] analog	Delay [ns] digital
0	0,012825	0,1	0
1	0,012825	0,1	0
2	0,04625	0,36	1,14
3	0,328625	2,59	2,61
4	0,531625	4,19	4,25
5	0,6875	5,42	5,78
6	0,81625	6,44	6,8
7	0,93875	7,4	7,89
8	1,0425	8,22	8,96
9	1,16725	9,2	10,04
10	1,3	10,3	11,18
11	1,43	11,3	11,77
12	1,555	12,3	13
13	1,67125	13,2	13,99
14	1,78875	14,1	15,05
15	1,9275	15,2	16,39
16	2,033125	16	17,36
17	2,16375	17	18,45
18	2,28625	18	19,56
19	2,415	19	20,68
20	2,53375	20	21,55
21	2,65125	20,9	22,44
22	2,7825	21,9	23,12
23	2,905	22,9	23,59
24	3,035	23,9	24,48
25	3,1	24,4	25,03

Tabelle 3: Messungen der digitalen und analogen Verzögerungen

Tabelle 3 zeigt eine wiederholte Messung als Funktion der eingestellten Verzögerungszeit, wobei man hier nur die digitalen und die analogen Ergebnisse

betrachtet. Die Messungen wurden 10-mal wiederholt und anschließend wurde der Durchschnitt ermittelt. Der TDC reagiert erst auf Verzögerungen größer als 2ns. Bei einer Verzögerungszeit ab 25ns pendelt sich die Spannung TDC_OUT auf 3,1 V ein.

4.1 Untersuchung der Kontrollspannung bei variabler Taktfrequenz

Taktsignal [MHz]	Kontrollspannung[V]
10	1,352
15	1,578
20	1,804
25	2,041
30	2,523
35	2,584
40	2,865
45	3,143
50	3,086
55	2,999
60	2,947
65	2,921
70	2,898
75	2,871
80	2,826

Tabelle 4: Einfluss des Taktsignals auf die Kontrollspannung

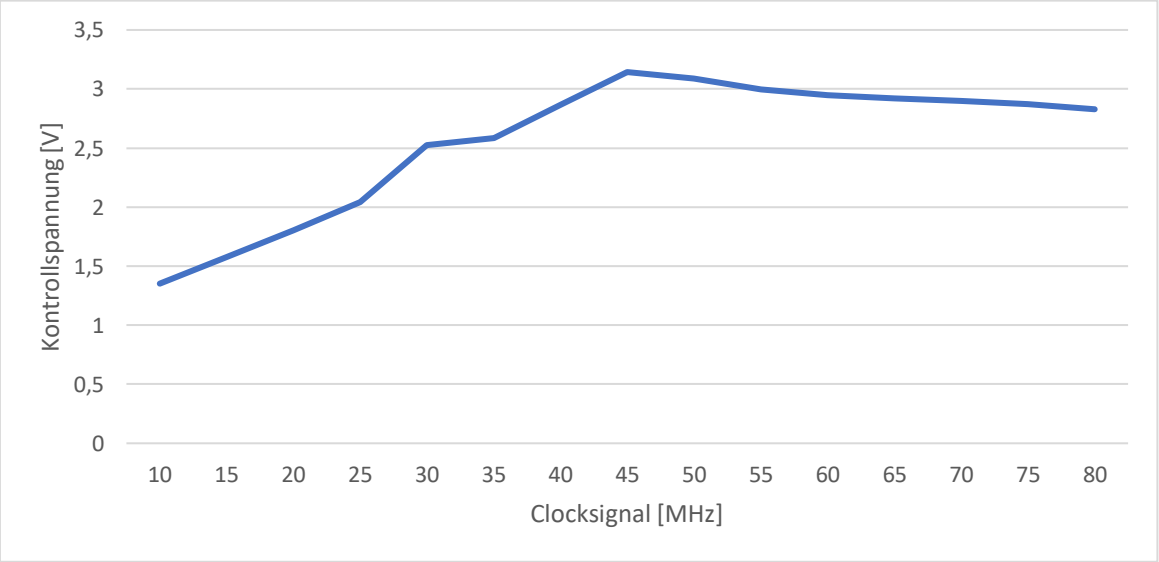

Abbildung 14: Diagramm zum Einfluss der Frequenz des Clocksignals auf die Kontrollspannung

Tabelle 4 und die Abbildung 14 machen deutlich, dass die Kontrollspannung sehr stark von der Frequenz des Taktsignals abhängt. Bei den Messungen mit 40MHz

beträgt die Kontrollspannung 2,865V. Die Kontrollspannung soll im Bereich von 2V-3,6V liegen. Mit dieser Spannung wird die Verzögerungszeit der Delay-Line geregelt. Die Abbildung 27 zeigt den Verlauf der Spannung. Man hat erwartet, dass die Kontrollspannung bis 3,6V steigen kann. Jedoch hat die Messung gezeigt, dass die Spannung ab einer Frequenz von ca. 47MHz einen Spannungswert von 3,1V erreicht hat und mit weiter steigender Frequenz gefallen ist.

5 Einführung in den Messplatz zum automatisierten Test des TDC

Der für das Projekt benutzte Messplatz setzt sich aus einem Rechner mit CentOS Betriebssystem, zwei Keysight 33622A Waveform-Generatoren, ein Keithley-Multimeter DM6500, ein Agilent-Multimeter, ein FTDI Modul und einer Platine zum Test des TDC zusammen. Die grafische Bedienoberfläche des Messstandes wird in Qt Creator entwickelt. Im Folgenden werden die benutzten Geräte und die Software vorgestellt.

5.1 Keithley Multimeter DMM6500

Das Multimeter (Abbildung 15) verfügt über mehrere Funktionen wie z.B. die DC-Strom- und Spannungsmessung oder die AC-Messung. In diesem Projekt wird das Multimeter zur Messung der analogen Repräsentation des TDC Wandlungsergebnisses benutzt. Ein wichtiger Vorteil des Geräts ist, dass es auf SCPI-Befehle (Standard Commands for Programmable Instruments) reagieren kann. Das heißt, dass man die Möglichkeit hat, das Gerät über die Programmierumgebung zu steuern.[8]

Abbildung 15: Keithley Multimeter DMM6500[9]

[8] [5]
[9] [6]

5.2 Keysight Waveform-Generator 33622A

Die beiden Waveform-Generatoren der Firma *Keysight* bieten eine umfangreiche Funktionsausstattung. Sie verfügen über zwei Kanäle, die z.B. Sinus-, Rechteck-, Dreieck- und Pulssignale erzeugen können. Besonders geeignet ist der Waveform-Generator für die Generierung von langen Wellenformen mit vielen Punkten und Synchronisation von zwei Signalen. Die Abtastrate des Geräts beträgt 1 GSa/s und die Bandbreite liegt bei 120MHz. Wie das *Keithley*-Multimeter können die Geräte über SCPI-Befehle gesteuert werden.[10]

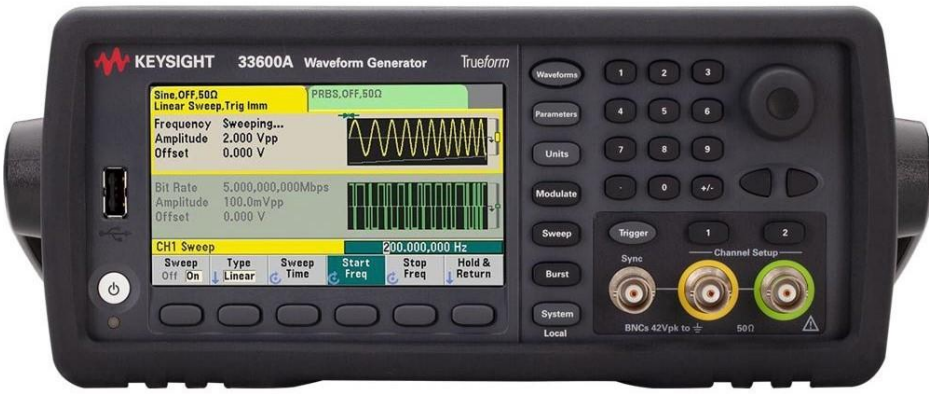

Abbildung 16: Waveform-Generator Keysight 33622A[11]

5.3 SCPI

SCPI (Standard Commands for Programmable Instruments) ermöglicht unter Anwendung des ASCII-Codes die Steuerung von Test- und Messgeräten. Diese Befehle sind international normiert und können in kurzer oder langer Form angegeben werden, wie z.B. „VOLT“ oder „VOLTage“. Es ist dabei zu beachten, dass die in dem Handbuch beschriebenen Befehle einzuhalten sind. Der in dem Projekt verwendete SCPI Befehl “SOURce1:APPLy:SQUare: 10000,3.3,1.65” konfiguriert beispielsweise den Kanal 1 des Waveform-Generators, sodass ein Rechtecksignal mit 10kHz, 3.3V Amplitude und 1.65 Offset generiert wird.

5.4 QT Creator

Mit der Entwicklungsumgebung QT-Creator hat man die Möglichkeit C++-Programme zu erstellen, die plattformunabhängig ausführbar sind. Diese Programme können durch GUIs visualisiert werden. In dieser Arbeit wird die

[10] [7]
[11] [7]

Visualisierung benutzt, um Eingaben durchzuführen und anschließend die Messergebnisse in Diagrammen darstellen zu können. Das Programm Qt-Creator kann auf der Seite des Herstellers, je nach passendem Betriebssystem runtergeladen und anschließend installiert werden.[12]

Nach dem Start des Programmes Qt-Creator hat man oben links in der Taskleiste die Option ein neues Projekt zu erstellen, indem man auf „Datei“ und dann auf „Neu“ klickt. Auch hat man die Möglichkeit, mit dem Schnellbefehl „Strg + Neu“ ein neues Projekt zu starten.

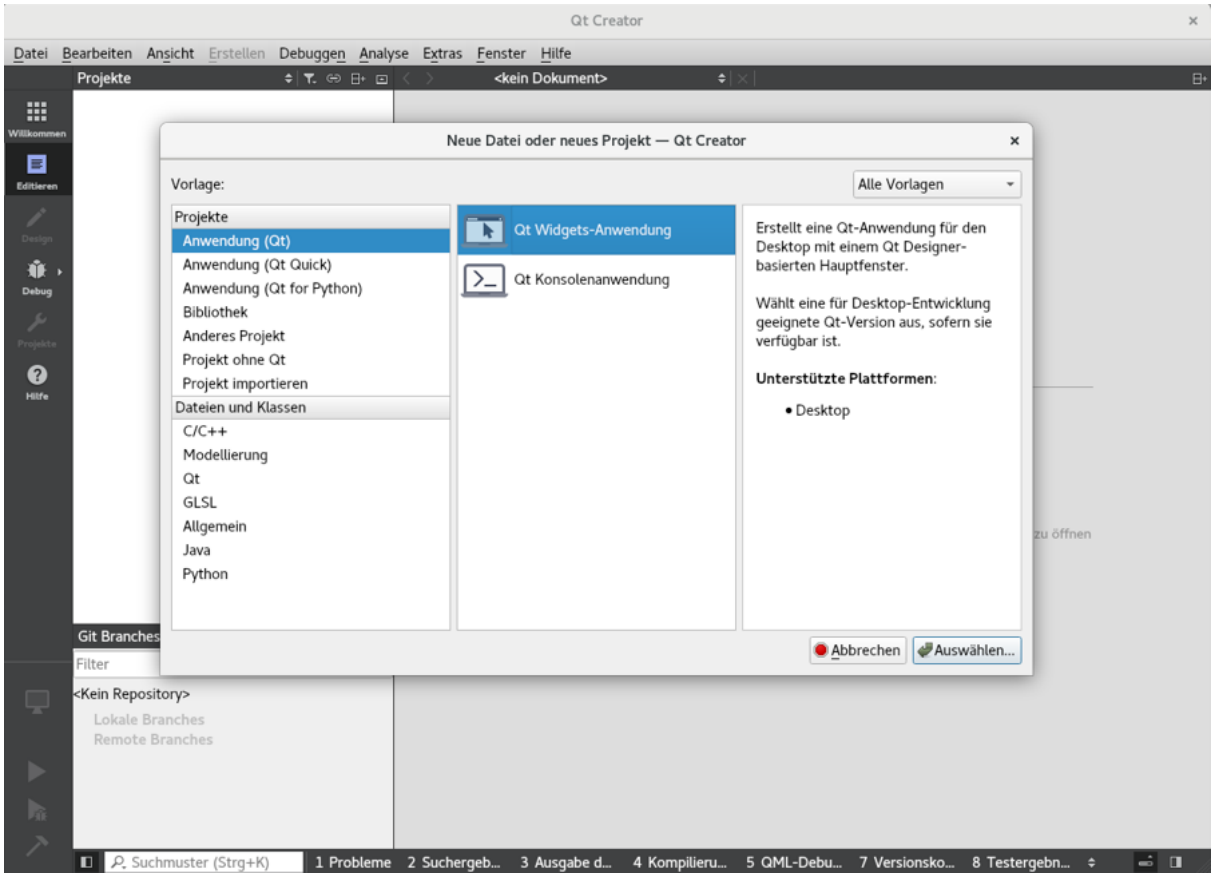

Abbildung 17: Qt Projekt "Neue Datei"

Abbildung 17 zeigt, dass sich danach ein neues Fenster öffnet, in dem ausgewählt werden muss, welche Art von Projekt erstellt werden soll. In dieser Bachelorarbeit wird eine Programmierumgebung mit einer GUI bevorzugt und deswegen wird unter „Projekte“ der Unterpunkt „Anwendung“ (Qt) und Qt Widgets-Anwendung selektiert. Durch das Klicken auf „Auswählen“ gelangt man zum nächsten Fenster.

[12] [9]

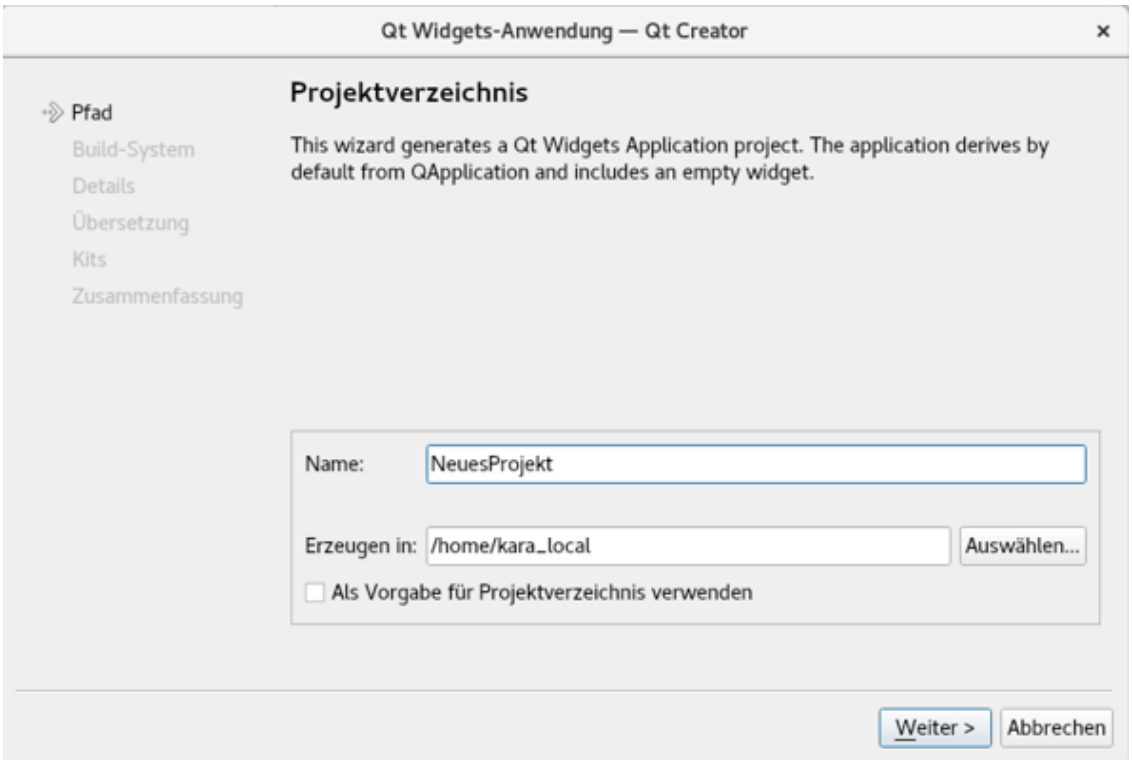

Abbildung 18: Qt Projekt "Projektverzeichnis"

Abbildung 18 zeigt das nächste Fenster, worin man den Projektnamen und den Speicherort festlegen kann. Durch das Klicken auf „Weiter“ wählt man als nächstes das Build-System aus. In diesem Projekt wurde das System *qmake* verwendet. In dem Unterpunkt *Details* richtet man die Klasseninformationen für die GUI ein. Dort kann man zum Beispiel die Namen der Dateien ändern (Abbildung 19).

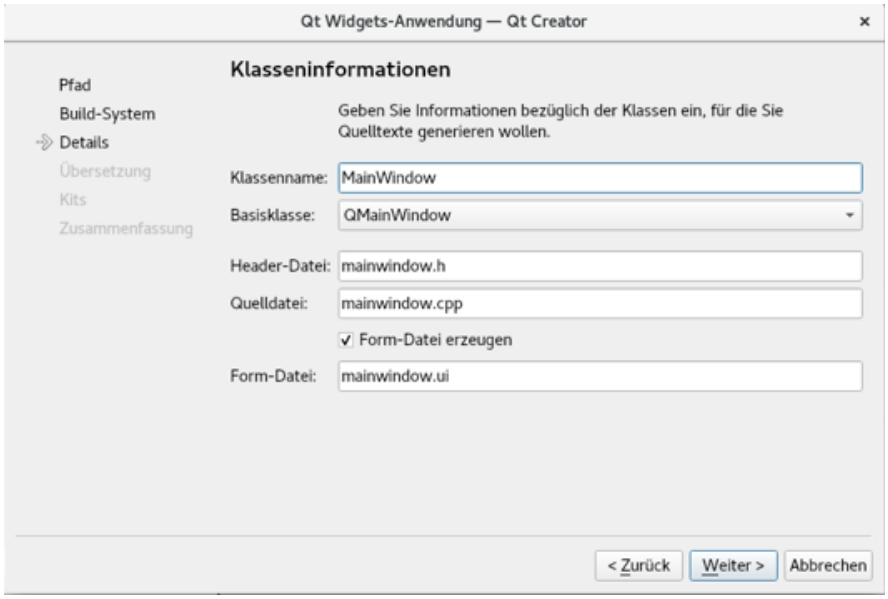

Abbildung 19: Qt Projekt "Details"

Falls eine Übersetzung des Projektes notwendig ist, kann dies in dem Unterpunkt „Übersetzung“ mit der nötigen Sprache ausgewählt werden. Als vorletzter Schritt erscheint die Auswahl des Kits. Hierbei wird das richtige Kit schon vorgegeben. Zum Schluss - bei der Erzeugung des Projektes - erscheint eine Zusammenfassung, in der man die gewählten Konfigurationen zusammengefasst sehen kann (Abbildung 20).

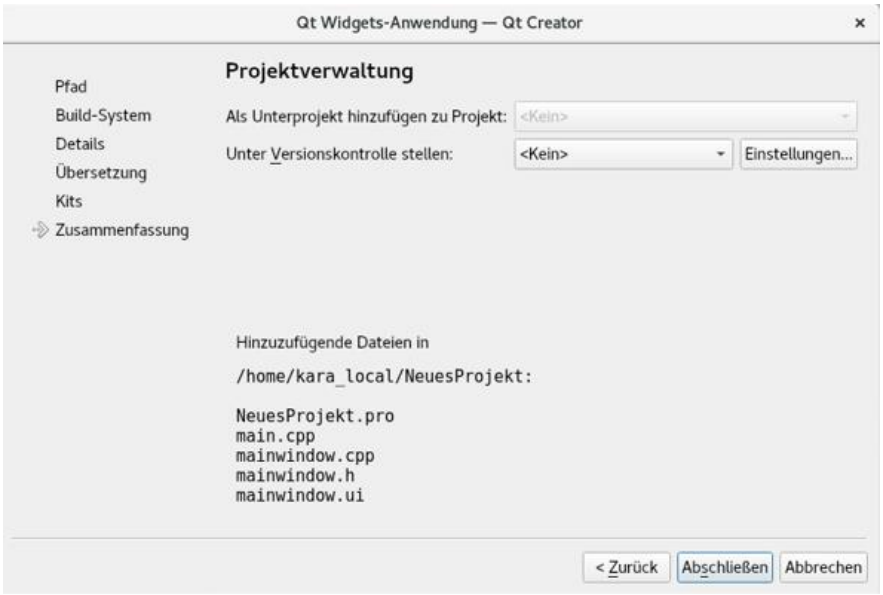

Abbildung 20: Qt Projekt "Zusammenfassung"

Mit „Abschließen“ öffnet das Projekt in der Progammierumgebung Qt.

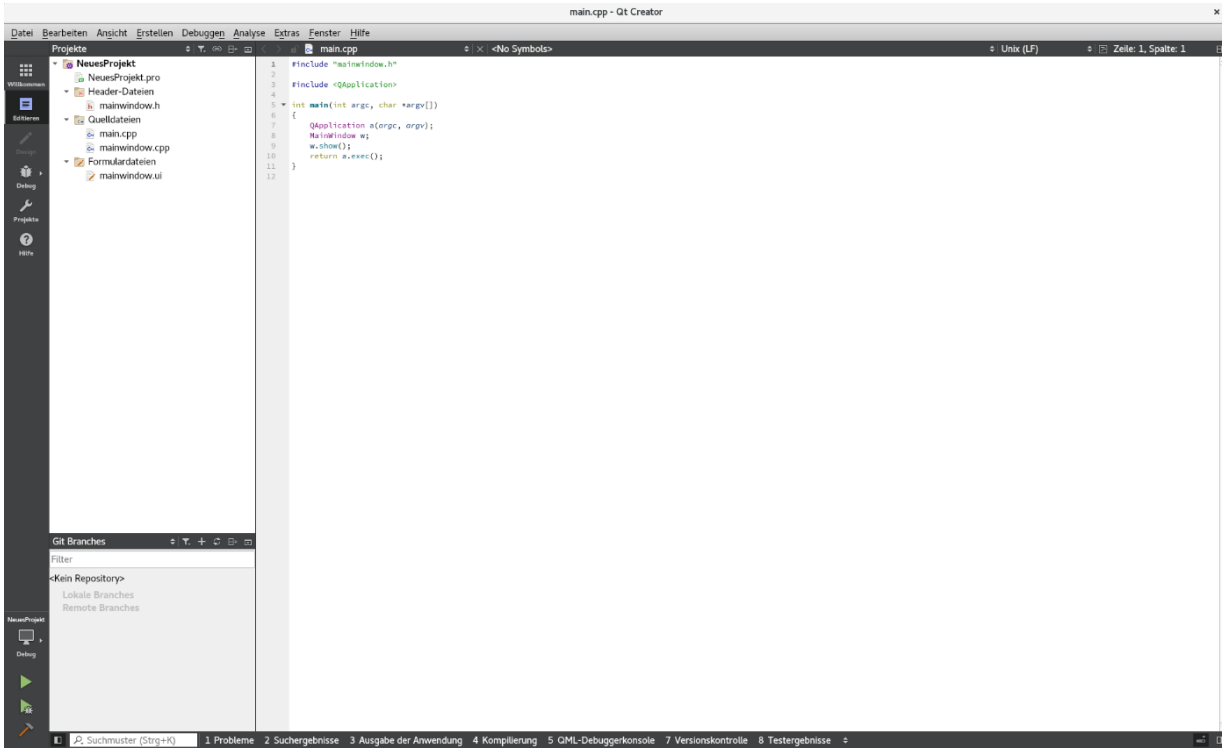

Abbildung 21: Qt Projekt "Projektübersicht"

In der Abbildung 21 sieht man das geöffnete Projekt mit den entsprechenden Dateien. Durch das Klicken auf die Pfeile können die Ordner ausgeklappt werden. Das Projekt besteht aus einer „pro-Datei“, „Header-Dateien“, „Quelldateien“ und „Formulardateien“. In diese Dateien kann der Code, der für die Erstellung der Messumgebung notwendig ist, programmiert werden. Die Abbildung 21 zeigt die „main.cpp“-Datei. Man hat die Möglichkeit, neue Header-, Quell- oder Formulardateien hinzuzufügen. Dies geschieht mit einem Rechtsklick auf das Ordnersymbol und Klicken auf „Hinzufügen“. Dadurch öffnet sich ein neues Fenster, worin man die passenden Dateien und Klassen auswählen kann.

Mit der Auswahl des „mainwindow.ui“ unter Formulardateien öffnet sich die GUI. Dort kann man mit vorgegebenen Objekten die GUI gestalten. Durch Klicken auf „Editieren“ gelangt man wieder in die Übersicht mit den einzelnen Dateien.

In der Abbildung 22 sieht man, dass auf der linken Seite verschiedene Objekte zur Verfügung stehen, die mit Drag & Drop auf der GUI platziert werden können. Hierbei stehen beispielsweise Eingabefelder, Buttons und Anzeigeobjekte zur Verfügung. Die hinzugefügten Elemente lassen sich in ihren Eigenschaften unter dem Unterpunkt „Eigenschaften“, wie in der Abbildung unten rechts zu sehen ist,

verändern. Auch besteht die Möglichkeit, die Eigenschaften eines Objektes in der GUI oder in der Übersicht der Objekte zu verändern.

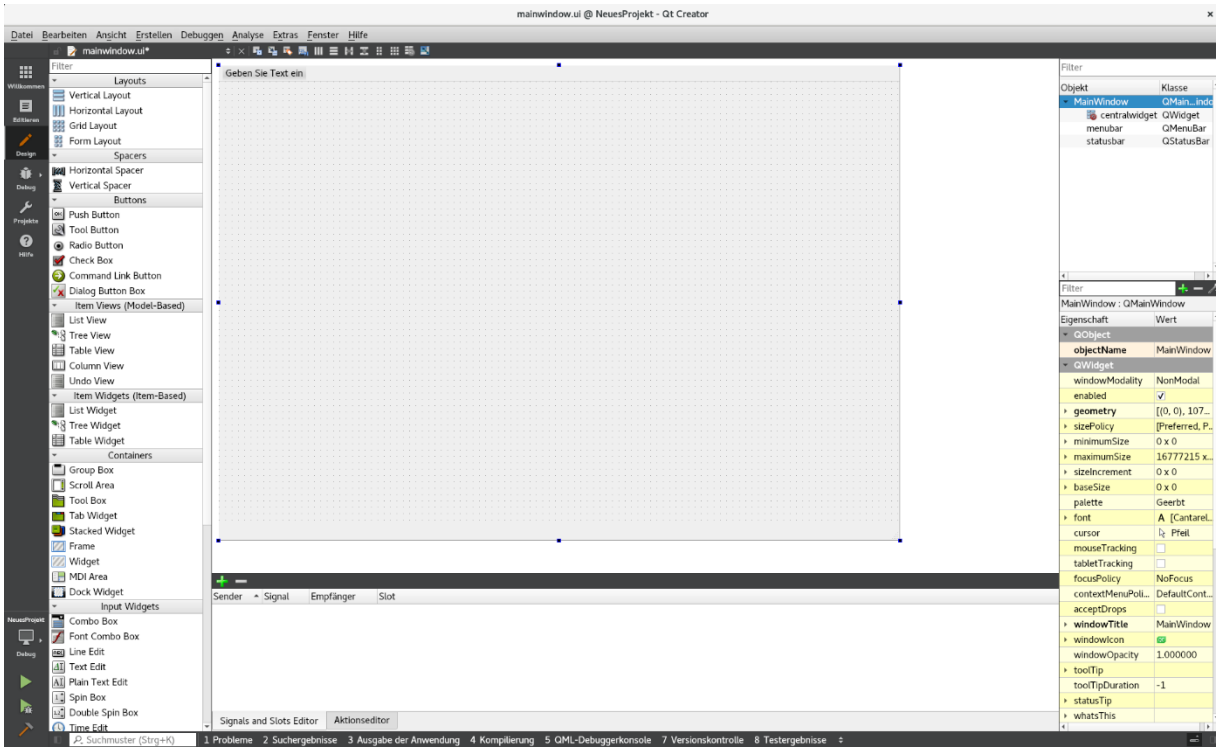

Abbildung 22: Qt Projekt "GUI Oberfläche"

In der Abbildung 23 wird ein „Push Button“ in die GUI-Umgebung mit Drag & Drop platziert. Anschließend kann der Name des Buttons verändert werden, wie zum Beispiel zu „Hallo“. Mit Rechtsklick auf den Button und „Objektnamen ändern“, hat man die Option, dem Objekt einen Namen zu geben. In diesem Beispiel wird der Name als „ObjektHallo“ bezeichnet.

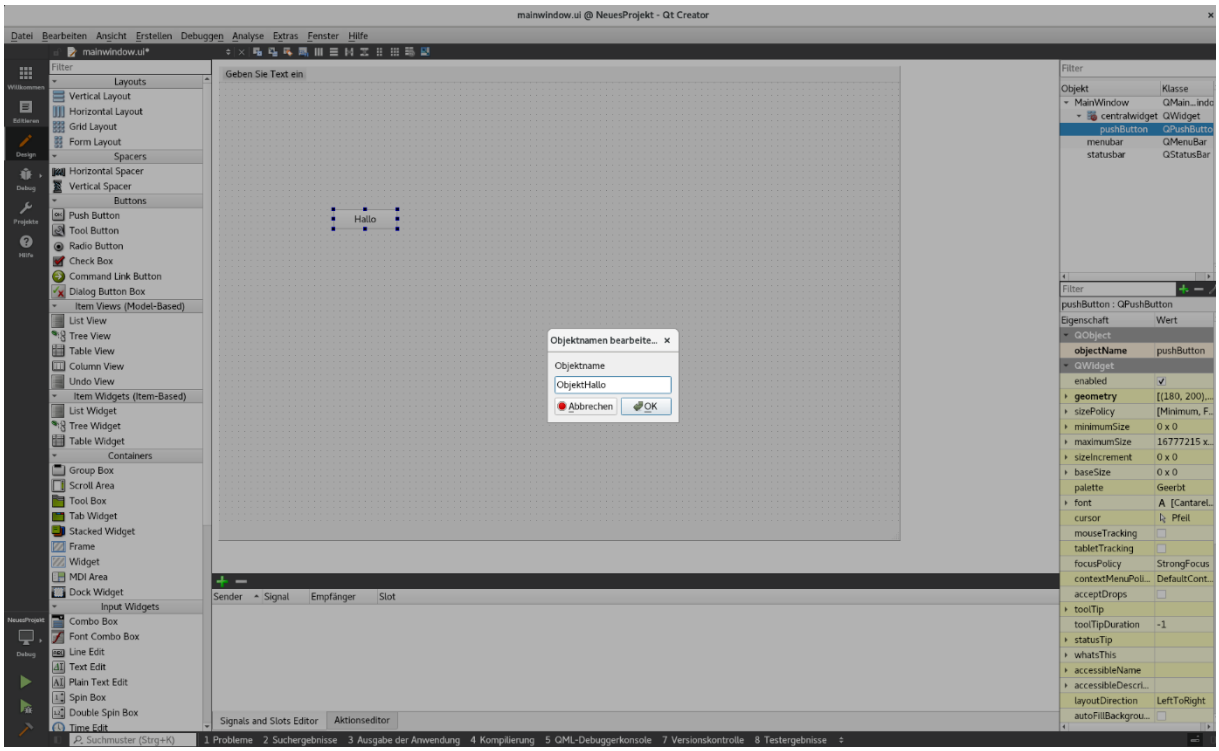

Abbildung 23: Qt Projekt "Push Button“

Anschließend wird, wie in der Abbildung 24 zu sehen ist, mit „Rechtsklick → Slot anzeigen“ die Funktion „clicked()“ ausgewählt. Jetzt hat man die Möglichkeit, die Funktion zu programmieren.

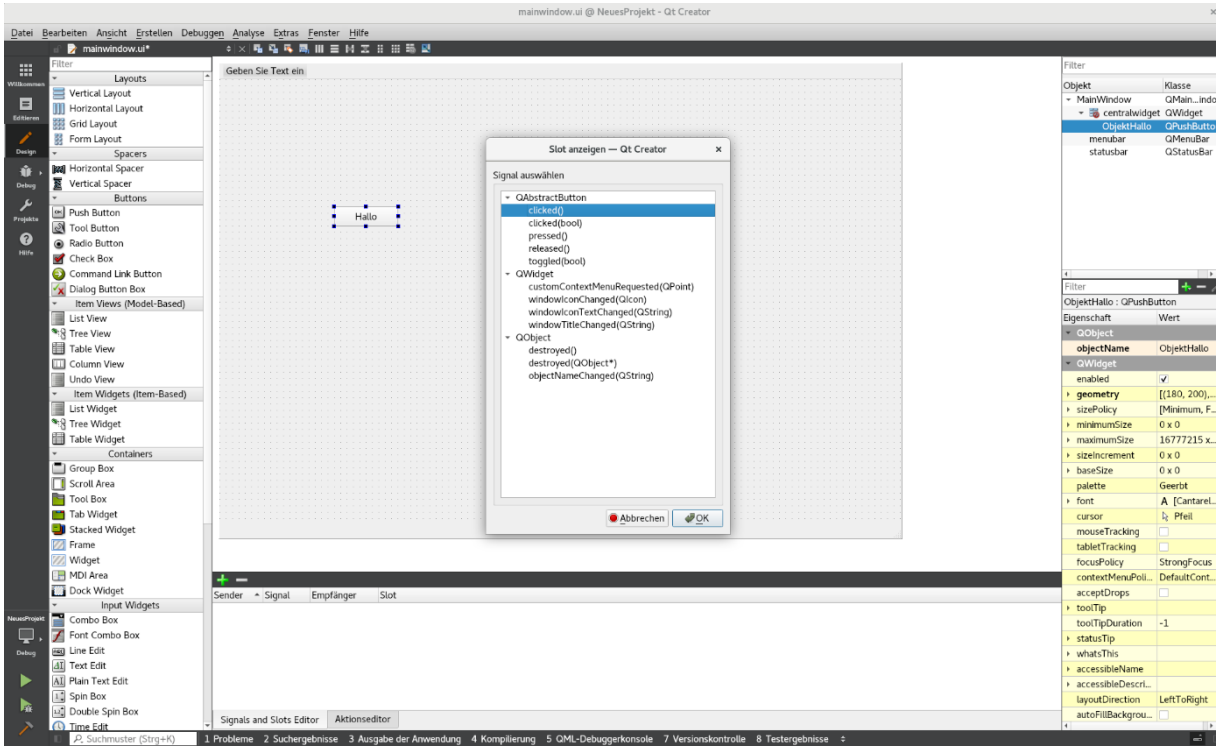

Abbildung 24: Qt Projekt "clicked()"

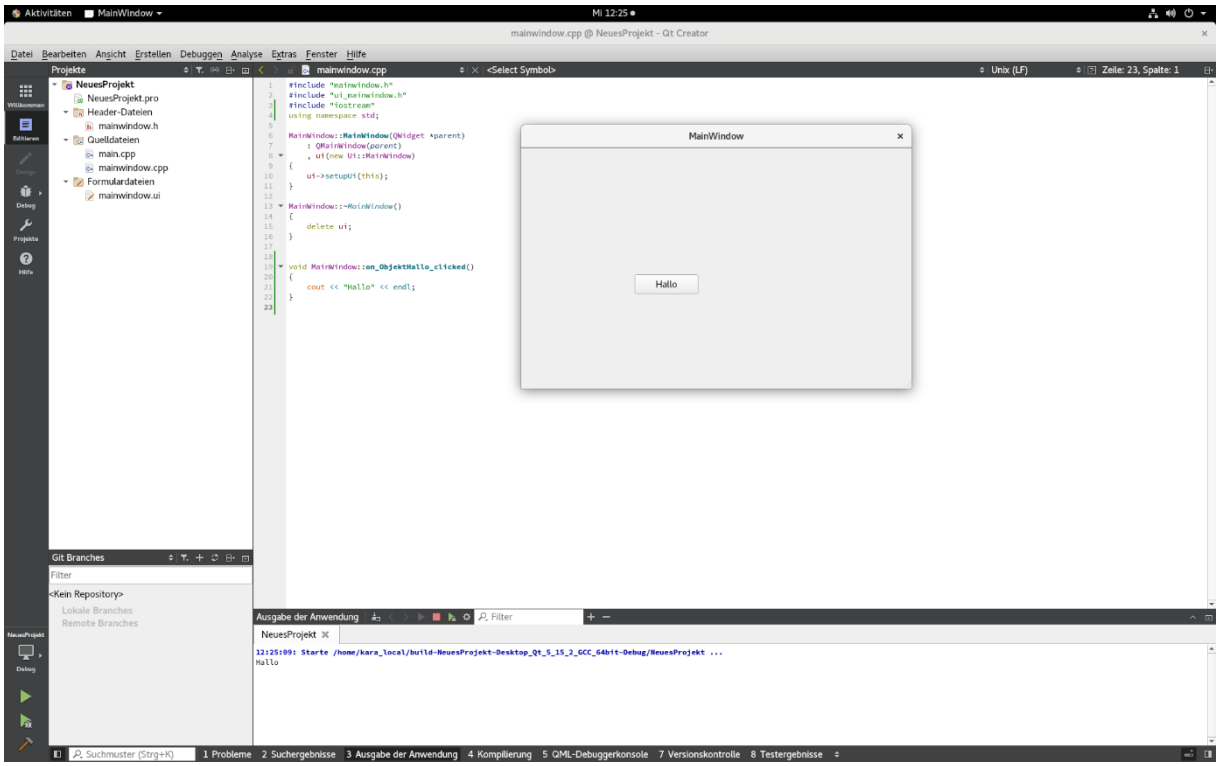

Abbildung 25: Qt Projekt "Funktion des Buttons"

Abbildung 25 zeigt die Funktion on_ObjektHallo_clicked. Die Funktion soll eine Ausgabe mit dem Wort „Hallo“ erzeugen, wenn man auf den Knopf „Hallo“ drückt. Unten im Terminal wird bei Ausführung des Programms, nach dem Klick auf die entsprechende Schaltfläche, das Wort „Hallo“ angezeigt.

5.5 FTDI

Das FT2232-56Q Mini-Modul (Abbildung 26) ist ein USB-seriell / FIFO-Entwicklungsmodul aus der FTDI-Produktpalette der Firma *Future Technology Device International Ltd*. FIFO ist ein Akronym für „First in, First out“ und ermöglicht eine Kommunikation mit höheren Geschwindigkeiten. Das Modul verwendet den FT2232H Chip, der alle USB-Protokolle, I^2C, SPI und JTAG beherrscht. Es verfügt über einen USB 2.0 Adapter, über den die Daten mit einer Geschwindigkeit von bis zu 480 Mb/s übertragen werden können.

Es ist ideal für die Verwendung in USB basierten Systemen und verfügt über zwei Modulkanäle, die synchron, asynchron oder als parallele FIFO-Schnittstelle programmiert werden können. Die beiden Kanäle können auch unabhängig voneinander für die Verwendung einer MPSSE-Engine konfiguriert werden. Der FT2232H ist die 5. Generation der FTDI Geräte und bietet erweiterte Funktionen, wie z.B. das MPSSE (Dual Multi-Protocol Synchronous Serial Engine), welches

auch in dieser Arbeit verwendet wird. Das FT2232H-56Q Mini-Modul verbindet die Signale des FT2232H-56Q ICs mit zwei 26-poligen zweireihigen Stiftleisten, die einen einfachen Anschluss von Leiterplattensteckern und Flachbandkabeln ermöglichen.[13]

Abbildung 26: FT2232-56Q Mini Modul[14]

Das Datenblatt und die USB-Gerätetreiber des FT2232H können auf der Internetseite des Herstellers kostenlos runtergeladen werden. Für die Verbindung zwischen dem FTDI und dem PC wird ein USB-Treiber benötigt. Die Verbindung kann mit einem Virtual COM Port (VCP) (Abbildung 27) aufgebaut werden, welches einen standartmäßigen PC-Anschluss über die USB-Schnittstelle emuliert. Eine weitere Möglichkeit ist, den D2XX-Treiber zu benutzen. Diese Methode wird in dieser Arbeit angewendet und ermöglicht, aus einer Anwendungssoftware mit Hilfe einer DLL (Dynamic Link Library) den Treiber direkt anzusprechen (Abbildung 28).[15]

[13] [10]
[14] [10]
[15] [10]

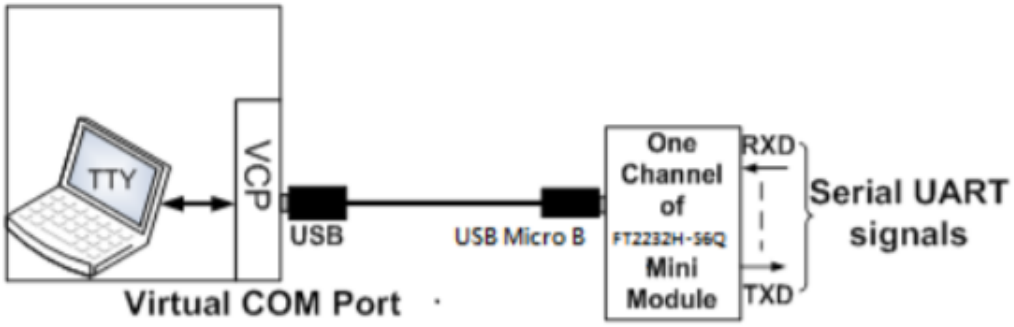

Abbildung 27: Virtual COM Port[16]

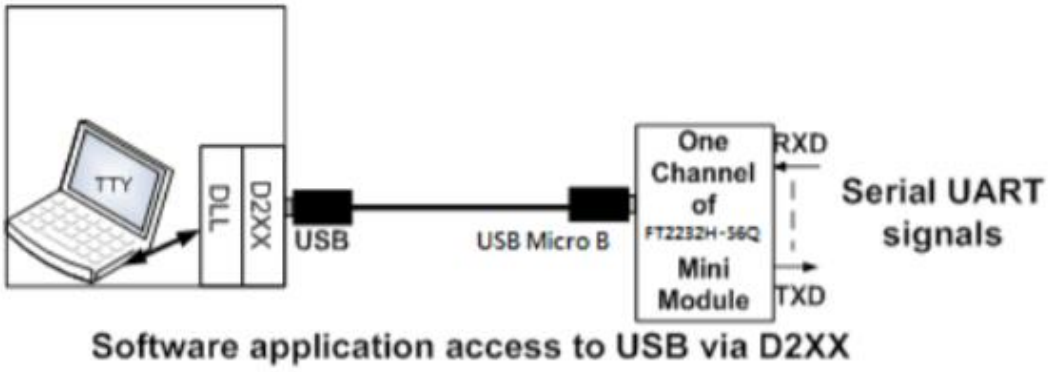

Abbildung 28: Software application access[17]

5.5.1 Pin Belegung

Die elektrischen Anschlüsse des Mini-Moduls FT2232H-56Q sind in Abbildung 29, Tabelle 5 und Tabelle 6 dargestellt.

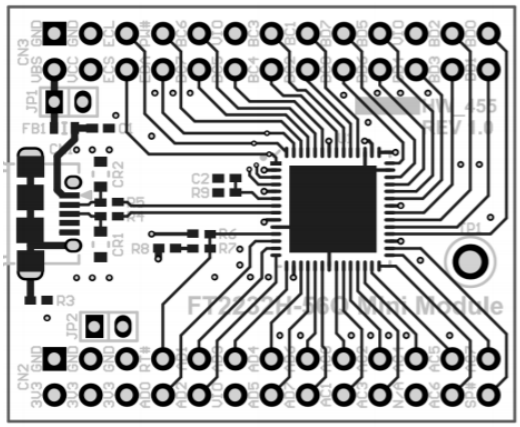

Abbildung 29: Elektrische Anschlüsse des Mini-Moduls FT2232H-56Q[18]

[16] [10]
[17] [10]
[18] [10]

Connector Pin	Name	Description
CN2-1	GND	0V Power pin
CN2-2	V3V3	3.3V DC generated from VCC (output)
CN2-3	GND	0V Power pin
CN2-4	V3V3	3.3V DC generated from VCC (output)
CN2-5	GND	0V Power pin
CN2-6	V3V3	3.3V DC generated from VCC (output)
CN2-7	RESET#	FT2232H-56Q RESET# pin
CN2-8	AD0	FT2232H-56Q AD0 pin
CN2-9	AD1	FT2232H-56Q AD1 pin
CN2-10	AD2	FT2232H-56Q AD2 pin
CN2-11	AD3	FT2232H-56Q AD3 pin
CN2-12	VIO	Connected to all FT2232H-56Q VCCIO pins (input)
CN2-13	AD4	FT2232H-56Q AD4 pin
CN2-14	AD5	FT2232H-56Q AD5 pin
CN2-15	AD6	FT2232H-56Q AD6 pin
CN2-16	AD7	FT2232H-56Q AD7 pin
CN2-17	AC0	FT2232H-56Q AC0 pin
CN2-18	AC1	FT2232H-56Q AC1 pin
CN2-19	AC2	FT2232H-56Q AC2 pin
CN2-20	AC3	FT2232H-56Q AC3pin
CN2-21	AC4	FT2232H-56Q AC4 pin
CN2-22	N/A	0
CN2-23	AC5	FT2232H-56Q AC5 pin
CN2-24	AC6	FT2232H-56Q AC6 pin
CN2-25	AC7	FT2232H-56Q AC7 pin
CN2-26	SUSPEND#	FT2232H-56Q SUSPEND# pin

Tabelle 5: Connector Pins CN2[19]

Connector Pin	Name	Description
CN3-1	GND	0V Power pin
CN3-2	VBUS	USB VBUS power pin (output)
CN3-3	GND	0V Power pin
CN3-4	VCC	+5V Power pin (input) used to generate V3V3, VPLL and VUSB
CN3-5	CLK	FT2232H-56Q EECLK pin
CN3-6	CS	FT2232H-56Q EECS pin
CN3-7	PWREN#	FT2232H-56Q PWREN#
CN3-8	DATA	FT2232H-56Q EEDATA pin
CN3-9	BC6	FT2232H-56Q BC6 pin
CN3-10	BC7	FT2232H-56Q BC7 pin
CN3-11	VIO	Connected to all FT2232H-56Q VCCIO pins (input)
CN3-12	BC5	FT2232H-56Q BC5 pin
CN3-13	BC3	FT2232H-56Q BC3 pin
CN3-14	BC4	FT2232H-56Q BC4 pin
CN3-15	BC1	FT2232H-56Q BC1 pin
CN3-16	BC2	FT2232H-56Q BC2 pin
CN3-17	BD7	FT2232H-56Q BD7 pin
CN3-18	BC0	FT2232H-56Q BC0 pin
CN3-19	BD5	FT2232H-56Q BD5 pin
CN3-20	BD6	FT2232H-56Q BD6 pin
CN3-21	VIO	Connected to all FT2232H-56Q VCCIO pins (input)
CN3-22	BD4	FT2232H-56Q BD4 pin
CN3-23	BD2	FT2232H-56Q BD2 pin
CN3-24	BD3	FT2232H-56Q BD3 pin
CN3-25	BD0	FT2232H-56Q BD0 pin
CN3-26	BD1	FT2232H-56Q BD1 pin

Tabelle 6: Connector Pins CN3[20]

Zumeist wird der FT2232H-56Q Pin direkt mit dem entsprechenden Pin an CN2 oder CN3 verbunden. Das FT2232H-56Q Minimodul erlaubt die Versorgung über eine externe Spannungsquelle. Alternativ kann das Modul direkt über den USB-Bus versorgt werden.

[19] [10]
[20] [10]

Je nachdem, welche Versorgungsquelle man auswählt, müssen bestimmte Pins durch Jumper verbunden werden. Bei der Versorgung über den USB muss VBUS über den Jumper1 mit VCC verbunden werden. Diese Verbindung sorgt für die Stromaufnahme von VBUS und verbindet ihn mit dem Spannungsreglereingang des Moduls. Der Spannungsregler versorgt den Chip mit den Spannungseingängen V3V3, VPLL und VUSB. Jumper 2 ist mit V3V3 mit VIO verbunden und liefert die korrekte Spannung von 3,3V für VCCIO auf dem Chip.[21]

5.5.2 D2XX

Die D2XX Schnittstelle ist eine proprietäre Schnittstelle, die speziell für FTDI-Geräte konzipiert ist. Die Funktionen werden den Anwendungsentwicklern über die FTD2XX-Bibliothek zur Verfügung gestellt, welche den Aufwand für die Softwareentwicklung verringert, da die Methoden von FTDI bereits vorentwickelt worden sind und in die Programmierumgebung eingebunden werden können. Um eine Verbindung über den USB-Port aufbauen zu können, muss zuerst der Treiber für das jeweilige Betriebssystem installiert werden. Den geeigneten Treiber findet man auf der Internetseite von FTDI.[22] [23]

5.5.3 MPSSE

Der Modus bietet flexible Möglichkeiten zur Anbindung von synchronen seriellen Geräten an einen USB-Port. Da MPSSE „Multi-Protocol“, ermöglicht die Kommunikation mit vielen verschiedenen Arten von synchronen Geräten, wie z.B. SPI, I^2C und JTAG. Datenformatierung und Taktsynchronisation können auf verschiedener Weise, je nach Anforderung konfiguriert werden. Zusätzlich zu den seriellen Datenpins sind weitere GPIO (General Purpose Input/Output) Signale verfügbar. Die im GPIO Modus verwendeten Pins samt ihren Beschreibungen werden in Tabelle 7 dargestellt.[24] [25]

[21] [10]
[22] [10]
[23] [11]
[24] [10]
[25] [12]

Channel A Pin No.	Channel B Pin No.	Name	Type	MPSSE Configuration Description
12	32	TCK/SK	OUTPUT	Clock Signal Output. For example: JTAG – TCK, Test interface clock SPI – SK, Serial Clock
13	33	TDI/DO	OUTPUT	Serial Data Output. For example: JTAG – TDI, Test Data Input SPI – DO
14	34	TDO/DI	INPUT	Serial Data Input. For example: JTAG – TDO, Test Data output SPI – DI, Serial Data Input
15	35	TMS/CS	OUTPUT	Output Signal Select. For example: JTAG – TMS, Test Mode Select SPI – CS, Serial Chip Select
17	37	GPIOL0	I/O	General Purpose input/output
18	38	GPIOL1	I/O	General Purpose input/output
19	39	GPIOL2	I/O	General Purpose input/output
20	40	GPIOL3	I/O	General Purpose input/output
22	42	GPIOH0	I/O	General Purpose input/output
23	46	GPIOH1	I/O	General Purpose input/output
24	47	GPIOH2	I/O	General Purpose input/output
25	48	GPIOH3	I/O	General Purpose input/output
26	49	GPIOH4	I/O	General Purpose input/output
27	51	GPIOH5	I/O	General Purpose input/output
28	52	GPIOH6	I/O	General Purpose input/output
29	53	GPIOH7	I/O	General Purpose input/output

Tabelle 7: MPSSE Pins[26]

In dieser Arbeit werden die Pins AC0, AC1, AC7 und AD0-AD6 verwendet, die in den nächsten Kapiteln näher erläutert werden.

6 Testprogramm

Um den TDC so aussagekräftig wie möglich zu testen, wird unter der Anwendung von Qt ein automatisiertes Testprogramm entwickelt. Das entwickelte Testprogramm wird in den folgenden Kapiteln umfassender beschrieben. Bei der Entwicklung wurde die objektorientierte Programmiersprache C++ verwendet. Das folgende Klassendiagramm in der Abbildung 30 zeigt einen Überblick auf die einzelnen Klassen. Der gesamten Programmcode mit Header- und Quelldateien kann im Anhang nachgelesen werden.

[26] [13]

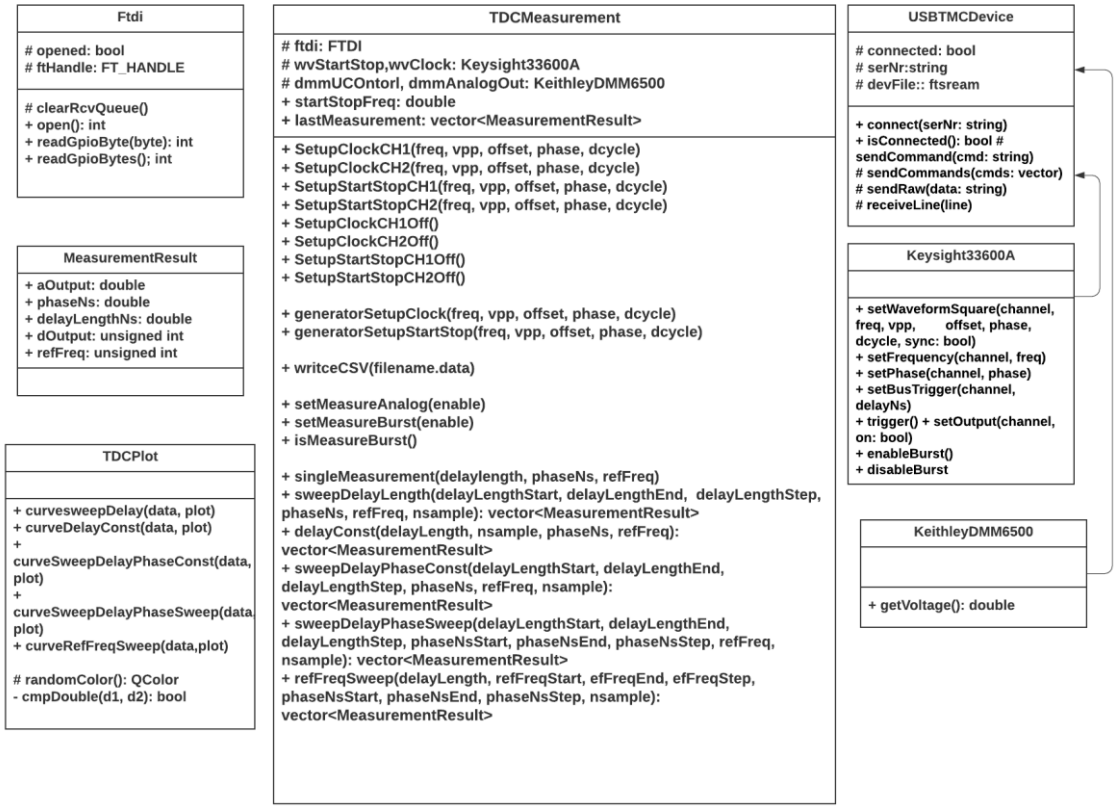

Abbildung 30:Klassendiagramm

6.1 „USBTMCDevice

Die „usbtmcdevice“-Klasse wird verwendet, um eine Verbindung mit den benötigten Geräten, die für das automatisierte Testen benötigt werden, herzustellen. Hier wird unter Benutzung des SCPI Befehls nach der ID-Nummer der Geräte gefragt und auf die Antwort der Geräte gewartet. Die Geräte werden auf die richtige ID-Nummer überprüft. Diese Klasse wird auch dazu benötigt, um einzelne oder zusammengesetzte Befehle an die Geräte zu senden.

6.2 „Keysight33600A“

In dieser Klasse wird das Messgerät Keysight Typ 33622A in das Projekt integriert. Hauptsächlich wird diese Klasse benutzt, um den Waveform-Generator unter Verwendung der „USBTMCDevice“ Klasse mit SCPI Befehlen zu versorgen. Die folgende Funktion „setWaveformSquare“ erzeugt ein Rechtecksignal, welches durch die grafische Oberfläche bedient werden kann. ´

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void Keysight33600A::setWaveformSquare(Channel ch, double freq, double
vpp, double offset, double phase, double dcycle, bool sync)
{
    std::vector<std::string> cmds;
    std::string channel = (ch == Keysight33600A::Channel::CH1) ? "1" :
"2";

    std::stringstream cmd;


    cmd << "SOURce" << channel << ":APPLy:SQUare " << freq << // SCPI
commands
           "," << vpp <<
           "," << offset;

    cmds.push_back(cmd.str());
    cmd.str("");

    cmd << "SOURce" << channel << ":FUNCtion:SQUare:DCYCle " << dcycle;

    cmds.push_back(cmd.str());
    cmd.str("");

    cmd << "SOURce" << channel << ":PHASe " << phase;
    cmds.push_back(cmd.str());

    if (sync)
    {
        cmds.push_back("SOURce" + channel + ":PHASe:SYNChronize");
    }

    sendCommands(cmds);

}
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Hierbei hat man die Möglichkeit, die Frequenz, Amplitude und den Offset der Spannung anzugeben. Für das Projekt ist es wichtig, Phasenverschiebungen bzw. Delays zu generieren. Eine Verzögerung zwischen Start- und Stoppsignalen wird mit einem Waveform-Generator erzeugt. Zusätzlich hat man die Möglichkeit, eine Phasenverschiebung zwischen der steigenden Flanke des Clock- und Startsignals einzustellen. Folgender Programmcode stellt diese Funktion dar.

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void Keysight33600A::setPhase(Channel ch, double phaseDeg, bool sync)
{
    std::vector<std::string> cmds;
    std::string channel = (ch == Keysight33600A::Channel::CH1) ? "1" :
"2";
    std::stringstream cmd;

        if (sync)
    {
```

```
        cmds.push_back("SOURce" + channel + ":PHASe:SYNChronize");
    }
    cmd << "SOURce" << channel << ":PHASe " << phaseDeg;
    cmds.push_back(cmd.str());
    sendCommands(cmds);
}
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Der Waveform-Generator kann im normalen Modus oder im Burst-Modus betrieben werden.

Im Burst-Modus wartet der Generator auf einen Triggerbefehl und erzeugt anschließend eine konfigurierbare Anzahl an Start/Stopp-Pulsen. Die Länge der Start- und Stopppulse ist abhängig von der Verzögerungszeit zwischen dem Trigger und dem Beginn der Start- und Stoppzyklen. Wenn nur ein Zyklus erzeugt wird, bleibt das Messergebnis bis zum nächsten Trigger für unbestimmte Zeit erhalten.

Die Abbildung 31 veranschaulicht den Ablauf der Messungen im Burst-Modus.

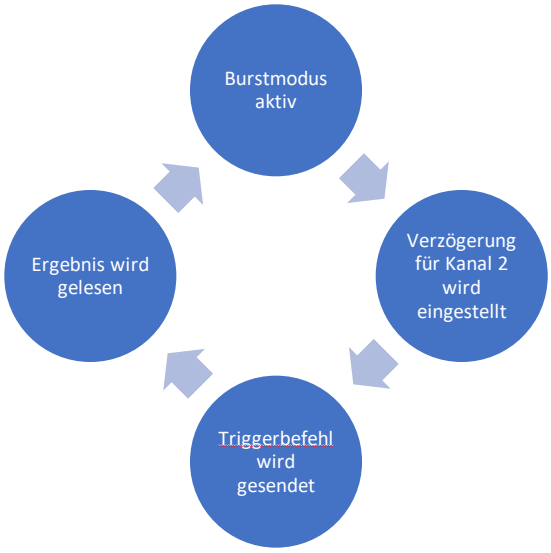

Abbildung 31: Burst-Modus

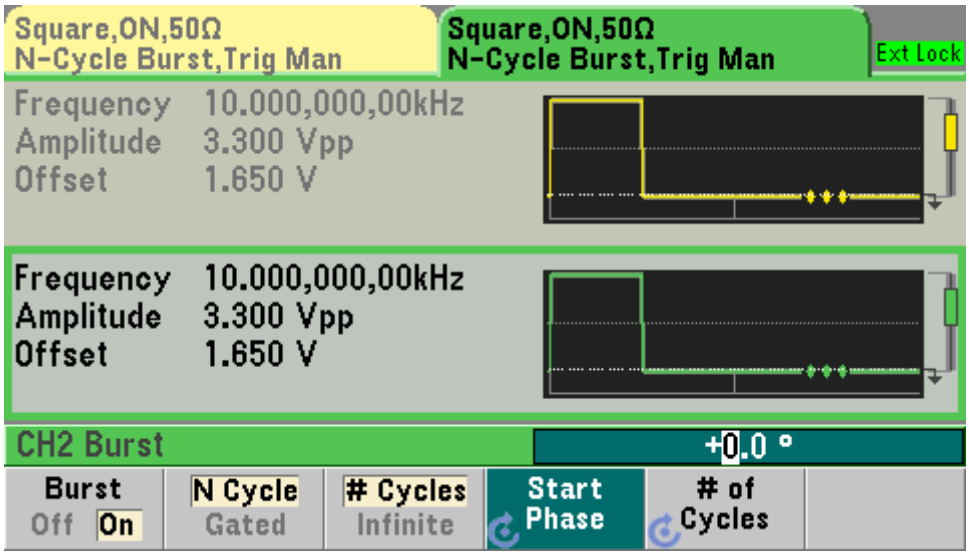

Abbildung 32: Burst Modus, Oberfläche

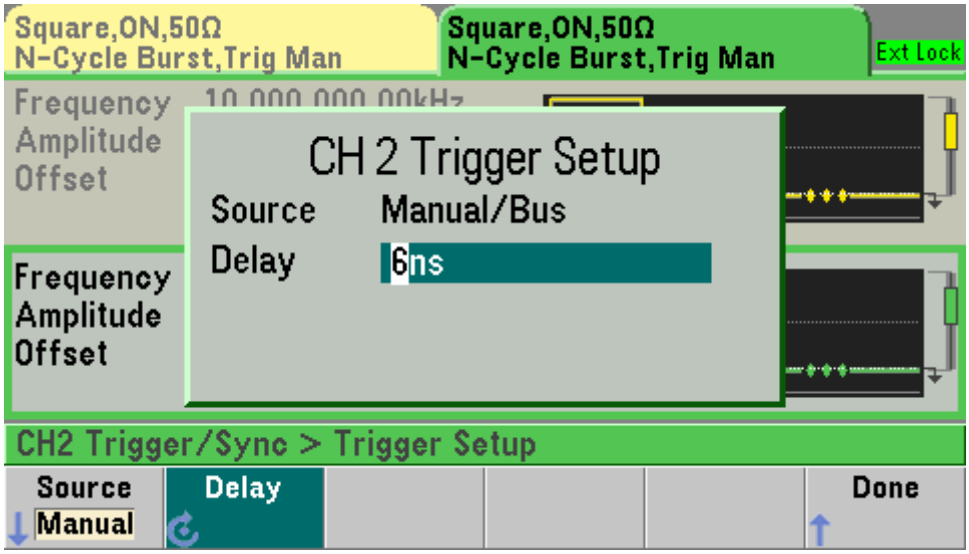

Abbildung 33: Burst Modus Oberfläche, Verzögerung

Die Abbildungen 32 und 33 zeigen die Oberfläche des Waveform-Generators im Burst-Modus. In der Abbildung 32 sieht man die beiden Kanäle – Kanal 1 für Start und Kanal 2 für Stopp - untereinander und in der Abbildung 33 wird im zweiten Kanal eine Verzögerung eingestellt, welcher mit einem Triggersignal ausgelöst wird.

Im normalen Modus wird kontinuierlich gemessen, das heißt Start und Stopp werden als kontinuierliche Rechteckwellen erzeugt. Die Länge des Starts und Stoppimpulses ist abhängig von der Frequenz und der Phase der Rechteckwellen. Ein möglicher Nachteil des normalen Modus ist, dass eine neue Messung ausgelöst wird, bevor das Ergebnis der aktuellen Messung ausgelesen worden ist, da bei jeder

steigenden Flanke von Start das letzte Messergebnis gelöscht wird. Die Abbildung 34 zeigt die Oberfläche des Waveform-Generators im normalen Modus.

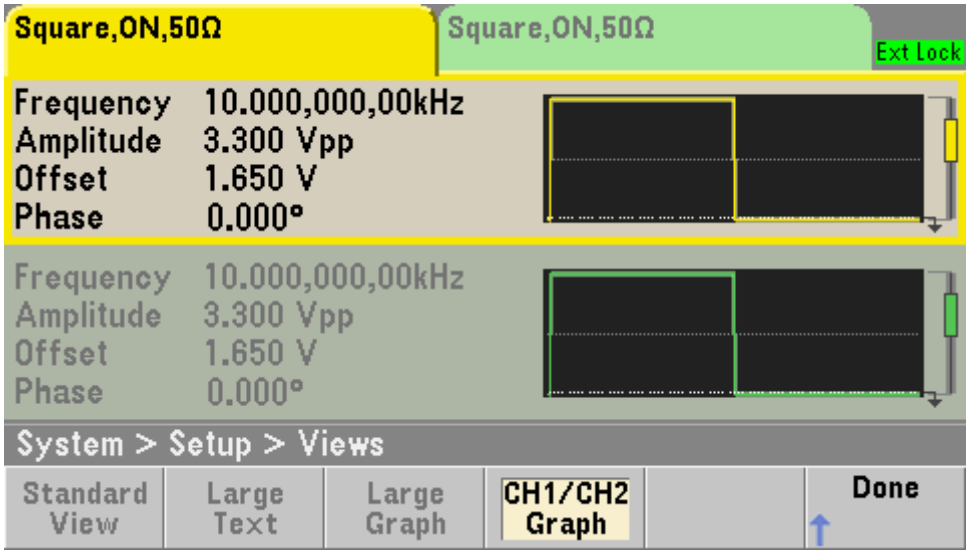

Abbildung 34: Normaler Modus, Oberfläche

6.3 „KeithleyDMM6500“

Das *Keithley* Multimeter DMM6500 wird verwendet, um ergänzend zu dem digitalen Ausgang, die analoge Spannung auszuwerten. Mit der unten dargestellten Funktion „getVoltage()“ werden die ermittelten Spannungswerte aufgenommen und können später ausgewertet werden.

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
double KeithleyDMM6500::getVoltage()
{
    sendCommand(":SENSE:FUNC 'VOLT:DC'");
    sendCommand("READ?");

    std::string response;

    if (receiveLine(response) != USBTMCStatus::OK)
    {
        std::cout << "Error measuring voltage" << std::endl;
        return 0;
    }
    size_t decimal_point = response.find(".");

    if (decimal_point != std::string::npos)
    {
        response = response.replace(decimal_point, 1, ",");
    }

    return std::stod(response);
}
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

6.4 „Ftdi“

Durch die „Ftdi“ Klasse wird die Einlesung des digitalen TDC Ausgangssignals ermöglicht. Es ist hierbei zu beachten, die vorentwickelten Bibliotheken „ftd2xx“ und „wintypes.h“ des Herstellers in das Projekt einzubinden. Durch Aktivierung des MPSSE Modus können die Pins gelesen und die digitalen Ausgänge ausgewertet werden. Die Abbildung 35 zeigt den schematischen Aufbau der Verbindung der Platine des TDC (auf der linken Seite des Bildes) mit dem FTDI (auf der rechten Seite des Bildes). Zur Realisierung des Anschlusses werden die Pins AC0-AC1, AC7, AD0-AD6 und ein GND Pin verwendet.

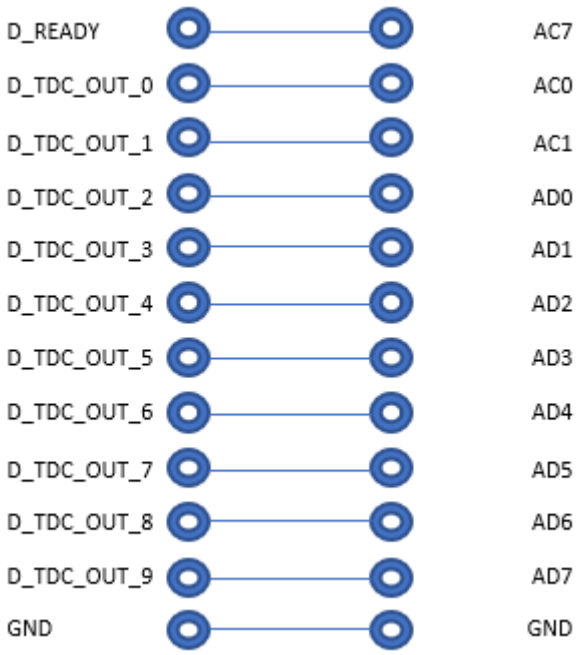

Abbildung 35: Pin-Verbindungen

Die Abbildung 36 zeigt ein Beispiel, wie die Ablesung der digitalen Ausgänge abläuft. Zuerst wird das DReady Signal gelesen und bei einem HIGH Signal wird die Auswertung gestartet.

Da die Datenports des FTDI Byte orientiert sind, während der TDC-Datenausgang 10 Bits besitzt, muss die Einlesung des TDC-Resultats in zwei Schritten durchgeführt werden. Von den Pins AC7-AC0 werden nur die niederwertigsten zwei Bits benötigt. Alle weiteren Bits werden durch eine UND-Operation mit dem Wert 3 maskiert. Anschließend werden die Daten des AD Ports um zwei Bitpositionen nach links geschoben. Schließlich werden die an den AD und AC Ports anliegenden

Daten durch eine Veroderung zusammengeführt. Hierbei entsteht ein Binärcode mit insgesamt 10 Bits.

Schritt 1:

	AC Pins							
AND	1	1	1	1	1	1	1	1
							1	1
	0	0	0	0	0	0	1	1

Schritt 2:

	AD Pins									
<<2	1	1	1	1	1	1	1	1		
	1	1	1	1	1	1	1	1	0	0

Schritt 3:

OR	1	1	1	1	1	1	1	1	0	0
									1	1
	1	1	1	1	1	1	1	1	1	1

Abbildung 36: Auswertung der Pins

6.5 „Mainwindow“

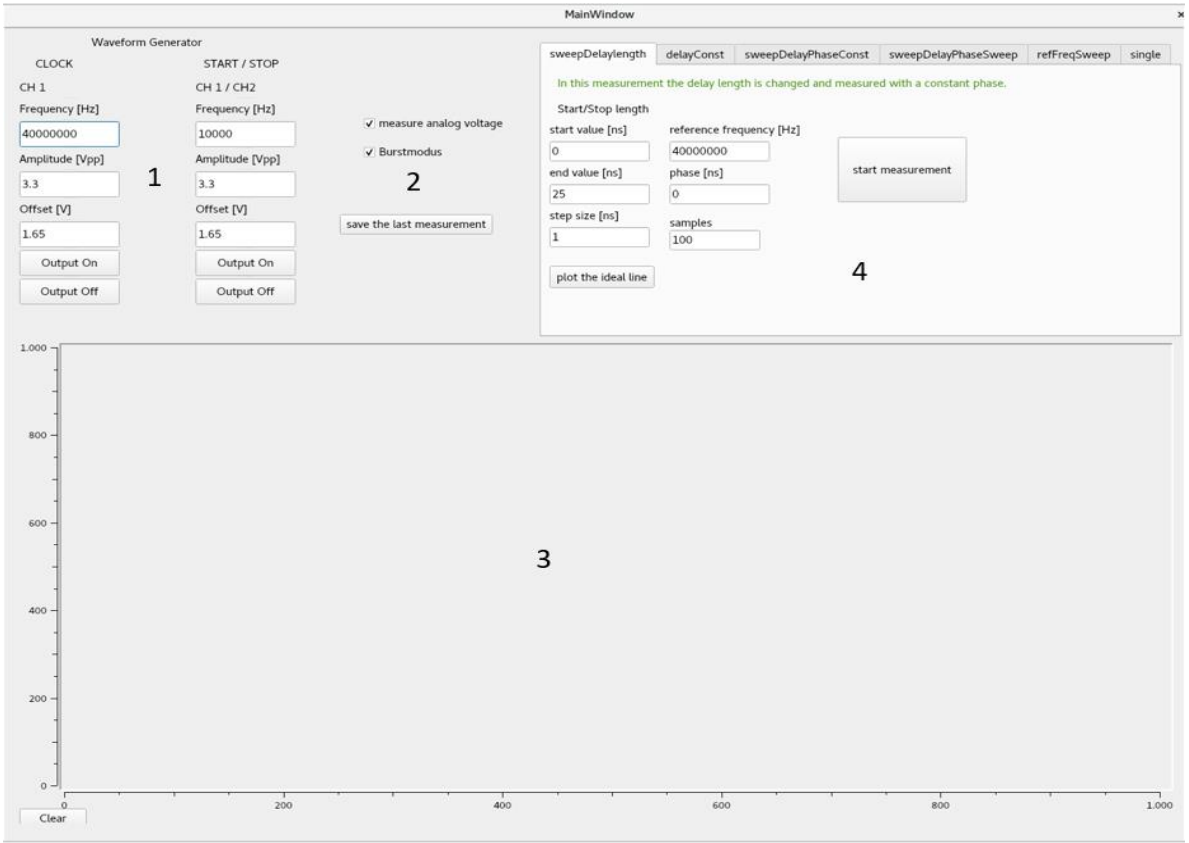

Abbildung 37: Mainwindow

Diese Klasse entspricht der GUI des Projektes. Die Abbildung 37 zeigt den Aufbau der Benutzeroberfläche. Im Bereich 1 hat man die Möglichkeit, die Werte für die beiden Waveform-Generatoren einzustellen. Diese sind bereits mit vorgegebenen Default-Werten gefüllt, jedoch hat man die Möglichkeit, die Werte zu ändern. Der Bereich 2 erlaubt die Auswahl zwischen dem Burst und dem normalen Modus. Man hat auch die Möglichkeit, die Ergebnisse nur von der digitalen oder zusätzlich auch über die analoge Schnittstelle des TDC aufzunehmen. Die letzten aufgenommenen Ergebnisse können nach Wunsch mit dem Button „save the last measurement“ abgespeichert werden.

An der dritten Position befindet sich der Diagrammbereich. Hier werden die aufgenommenen Ergebnisse in Diagrammen abgebildet. Der 4. Bereich erlaubt die Auswahl verschiedener Messungen. Durch die Auswahl der Tabs erscheinen die Konfigurationsmöglichkeiten der jeweiligen Messung. Diese Messungen werden im folgenden Kapitel TDCMeasurement näher erläutert. Die Darstellung der Diagramme der Messungen erfolgt in der Klasse TDCPlot und wird im nächsten Abschnitt beschrieben.

6.6 „Tdcplot“

In dieser Klasse werden die aufgenommenen Ergebnisse in der GUI als Diagramme visualisiert. Dies hat den Vorteil, dass der Benutzer nach Beendigung der Messergebnisse auf einem Blick die Messergebnisse beurteilen kann. Die Darstellung der Diagramme wird in Qt selbst nicht angeboten und muss durch die zusätzliche Installation des Qwt Widgets in das Projekt eingebunden werden.[27]

Da man die Möglichkeit hat, in den jeweiligen Messungen viele Auswertungen durchzuführen, wird in den einzelnen Messfunktionen der Durchschnitt der Anzahl der Messungen durchgeführt und als ein Wert für die Diagramme ausgegeben.

Folgende Abbildungen zeigen Beispiele, wie die Diagramme in der GUI aussehen.

[27] [14]

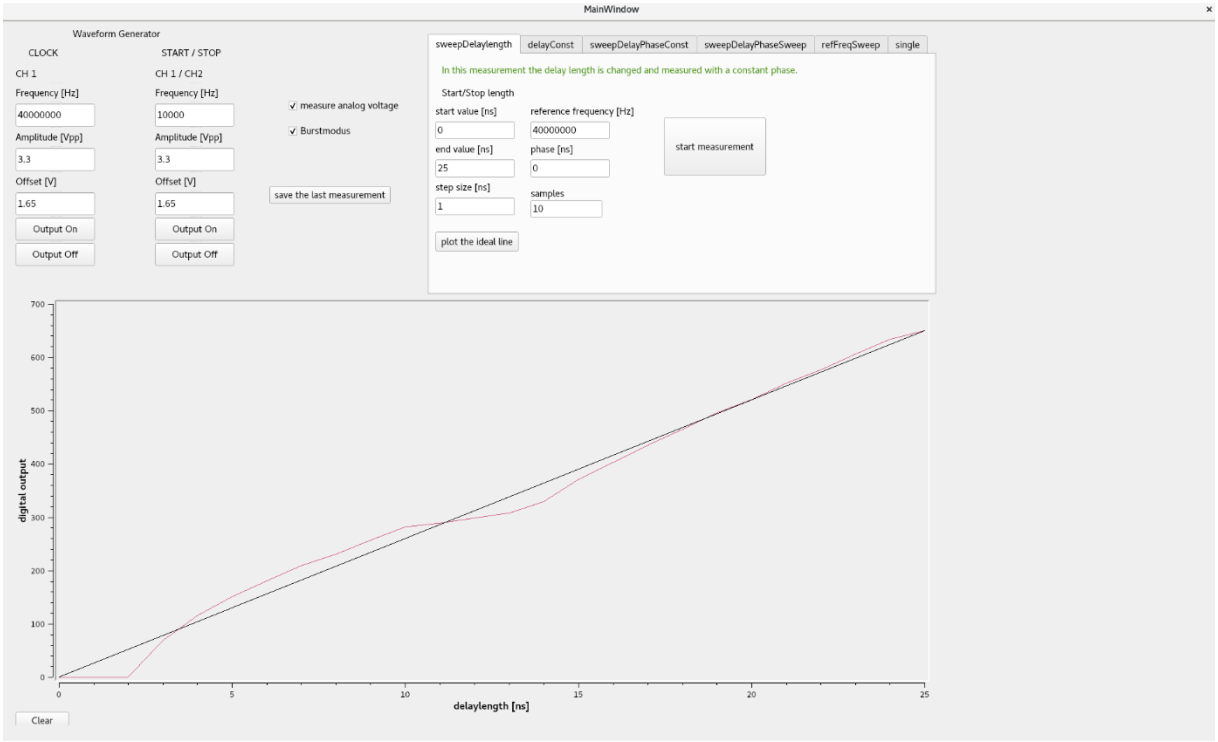

Abbildung 38: Diagramm "sweepDelaylength"

Das Diagramm in der Abbildung 38 zeigt ein Ergebnis der Messung „sweepDelaylength“ und zusätzlich in Schwarz die Ideallinie des TDC. Die Ideallinie hilft dabei auf den ersten Blick den Verlauf der Ergebnisse einzuschätzen. Auf der x-Achse wird die Delaylänge und auf der y-Achse wird das digitale Ergebnis dargestellt. Unten links befindet sich ein „clear“-Knopf, mit dem man das ausgegebene Diagramm löschen kann, um z.B. für die nächste Messung ein leeres Diagramm zu erhalten.

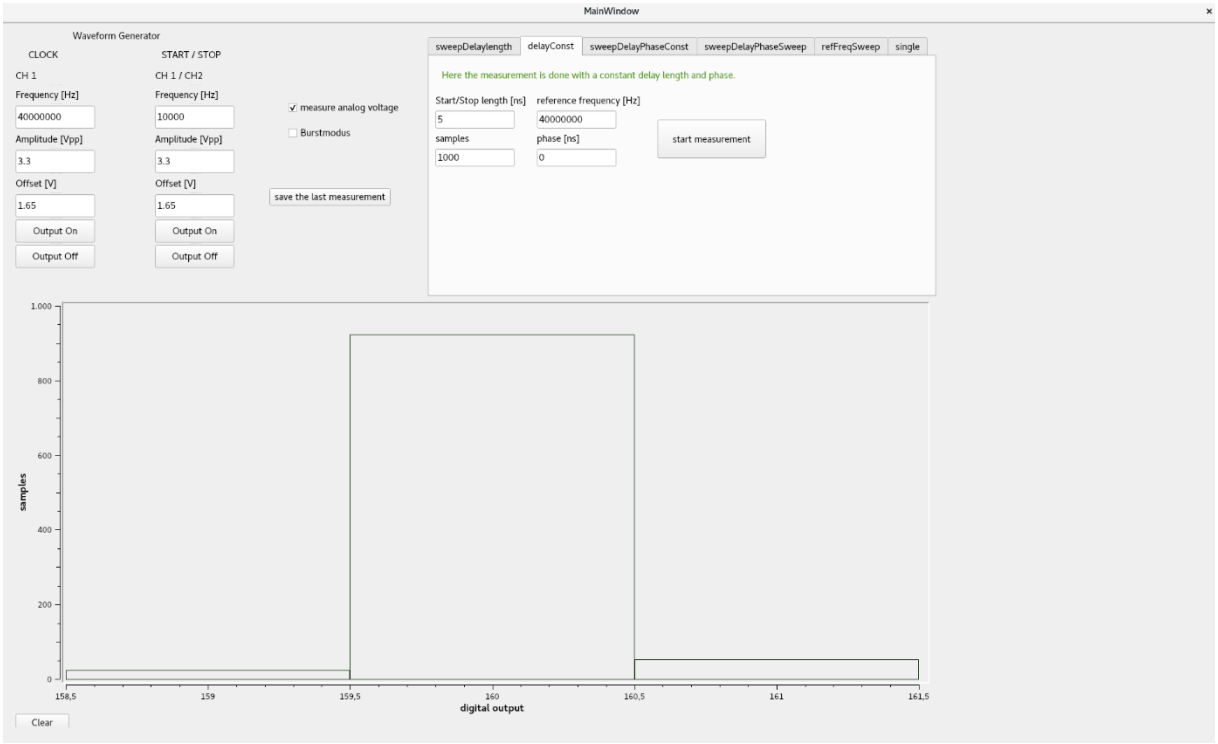

Abbildung 39: Diagramm "delayConst"

Die Abbildung 39 zeigt ein Histogramm für die Messung „delayConst“. Hier kann unmittelbar nach der abgeschlossenen Messung die Verteilung der digitalen Messergebnisse ausgewertet werden.

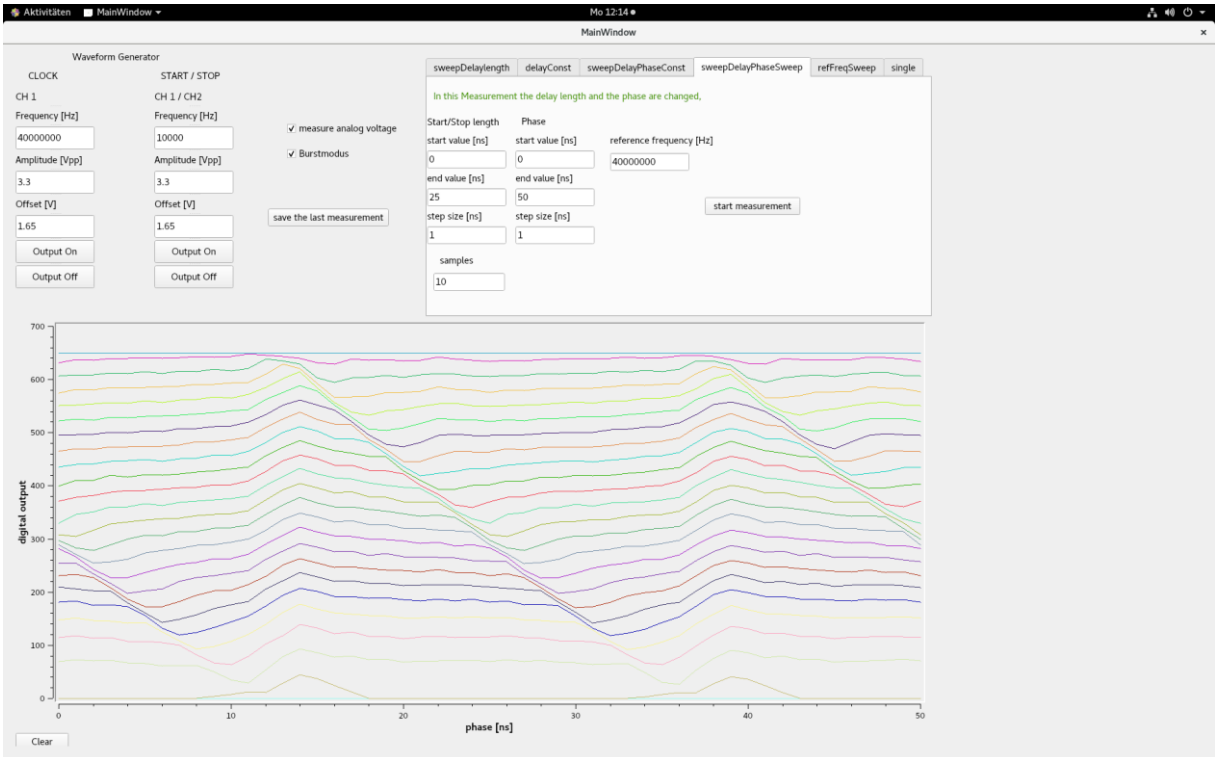

Abbildung 40: Diagramm "sweepDelayPhaseSweep“

In der Abbildung 40 ist ein Ergebnis der Messung „sweepDelayPhaseSweep“ dargestellt und man erkennt mehrere Kurvenschare, wobei eine Kurvenschar für eine Delaylänge steht. Da ein Phasen-Sweep durchgeführt wird, befindet sich auf der x-Achse die Phase. Zu der Phase werden die digitalen Ergebnisse ausgewertet.

Die einzelnen Kurven der Diagramme werden durch zufällige Farben visualisiert, damit der Benutzer einen besseren Überblick über die Messergebnisse haben kann.

6.7 „MeasurementResult“

In dieser Klasse werden die ausgewerteten Ergebnisse gespeichert und können in der GUI als Diagramm visualisiert werden. Außerdem können die Ergebnisse in csv-Dateien gespeichert und ausgewertet werden. CSV steht für „Comma-Separated Values“ und ermöglicht die Speicherung und Auswertung der Textdateien.

6.8 „TDCMeasurement“

In dieser Klasse wird die Steuerung für die einzelnen Messungen des TDC programmiert. In dieser Arbeit wird der TDC mit sechs verschiedenen Messfunktionen getestet. Die Ergebnisse werden als Vektor vom Typ MeasurementResult gespeichert. Dies ist notwendig, um die Ergebnisse als Diagramm in der GUI zu visualisieren. Außerdem hat man mit Vektoren die Möglichkeit, beliebig viele Ergebnisse zu speichern. Diese Ergebnisse werden in einer csv-Datei abgelegt und im Anschluss mit anderen Software-Paketen ausgewertet.

Bei allen Messfunktionen ist der Ablauf ähnlich. Zunächst wird abgefragt, in welchem Modus die Waveform-Generatoren betrieben werden sollen. Anschließend wird die eingegebene Referenzfrequenz eingestellt. Im Burst-Modus hat man die Möglichkeit, in 1ns Schrittweiten zu messen, während man im normalen Modus auch die Gelegenheit hat, mit kleineren Schrittweiten zu messen.
Im Burst-Modus werden Trigger-Befehle nach jeder eingestellten Start- und Stoppverzögerung gesendet, wobei im anderen Modus die Verzögerung in Grad umgewandelt und an den Waveform-Generator übergeben wird.
Die Werte für die digitalen und analogen Ausgänge, der Referenzfrequenz, der Delaylänge und der Phase werden als Objekt von der erstellten Klasse MeasurementResult gespeichert.

Die Phase beschreibt die zeitliche Verschiebung zwischen der steigenden Flanke des Referenztaktes und des Startsignals. Die Delaylänge ist die Zeit zwischen der steigenden Flanke des Start- und des Stoppsignals, und entspricht damit der eigentlichen Messzeit.
Diese werden darauffolgend in einem Vektor abgelegt. Außerdem hat man die Option, die Messungen nur mit den digitalen oder auch mit den analogen Werten aufzunehmen. Für jede Messung besteht die Möglichkeit, je nach Auswahl beliebig viele Auswertungen durchzuführen. Diese Auswahl kann in der GUI getroffen werden. Der Programmcode für diese Funktionen läuft ähnlich ab.

++

```
std::vector<MeasurementResult> TDCMeasurement::sweepDelayLength(double
delayLengthStart, double delayLengthEnd, double delayLengthStep, double
phaseNs, int refFreq, unsigned int nsample)
{
    std::vector<MeasurementResult> result_vec;

    if (measureBurst)
    {
        wvStartStop.enableBurst();
        wvStartStop.setBusTrigger(Keysight33600A::CH1, phaseNs);
        usleep(10000);
    }
    else
    {
        wvStartStop.disableBurst();
        wvStartStop.setPhase(Keysight33600A::CH1, ((phaseNs) *
startStopFreq * 360) / 1000000000.0, true);
        usleep(10000);
    }

    wvClock.setFrequency(Keysight33600A::CH1, refFreq);

    for (double sweep = delayLengthStart; sweep <= delayLengthEnd; sweep
+= delayLengthStep)
    {
        if (measureBurst)
        {
            wvStartStop.setBusTrigger(Keysight33600A::CH2, phaseNs +
sweep);
        }
        else
        {
            double roundedPhase = round((phaseNs - sweep) * startStopFreq
* 360 / 1000000000.0 * 1000) / 1000.0;

            std::cout << "sweep " << sweep << " Phase auf 3 NKS: " <<
roundedPhase << std::endl;
            wvStartStop.setPhase(Keysight33600A::CH2, roundedPhase,
true);
        }

        usleep(100000);

        for (int i = 0; i < nsample; i++)
```

```
        {
            MeasurementResult s;

            if (measureBurst)
            {
                wvStartStop.trigger();
                usleep(1000);
            }
            else
            {
                //std::cout << "Sleep for " << 2 * 1000000.0 /
startStopFreq << " µs" << std::endl;
                usleep(2 * 1000000.0 / startStopFreq);
            }

            s.dOutput = digitalOutput();
            if (measureAnalog)
            {
                s.aOutput = dmmAnalogOut.getVoltage();
            }

            s.refFreq = refFreq;
            s.delayLengthNs = sweep;
            s.phaseNs = phaseNs;

            result_vec.push_back(s);
        }
    }

    lastMeasurement = result_vec;
    return lastMeasurement;
}
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

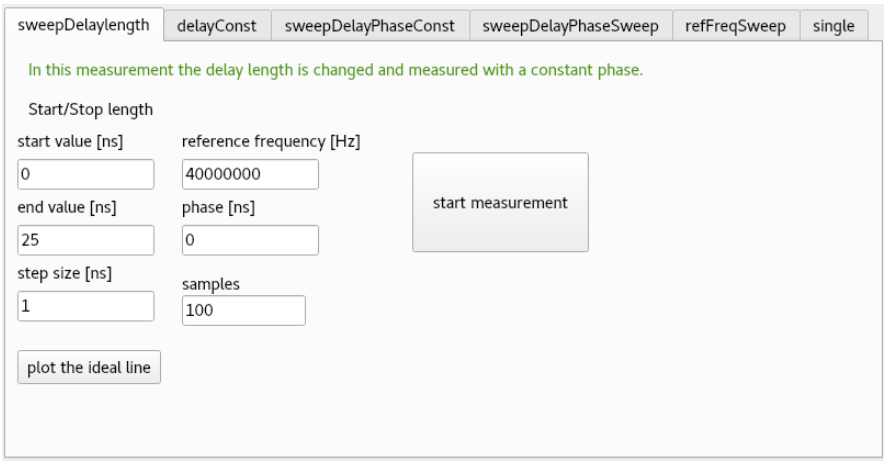

Abbildung 41: GUI "sweepDelaylength"

Bei dieser Messung wird die Verzögerung zwischen Start und Stopp von 0ns bis 25ns gemessen. Die For-Schleife ermöglicht einen Sweep von 0ns bis 25ns.

Die Abbildung 41 zeigt die Bedienfelder für die Messung „sweepDelaylength“. Der Benutzer kann einen beliebigen Start-, End- und Schrittwert für die Start/Stopp

Länge eintippen, wobei die Felder schon im Voraus mit Werten gefüllt sind. Wird für die Schrittweite der Wert 0 eingegeben, wird die kleinstmögliche Schrittweite für die Messungen angenommen.

Ergänzend dazu gibt es die Möglichkeit, die Referenzfrequenz, die Phase und Anzahl der Messungen einzugeben. Zusätzlich kann man bei der Messung durch Klicken auf den „plot the ideal line“-Button die Ideallinie des TDC in das Diagramm zeichnen. Mit „start measurement“ startet die Messung.

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
std::vector<MeasurementResult> TDCMeasurement::delayConst(double
delayLength, unsigned int nsample,double phaseNs, int refFreq)
{

    std::vector<MeasurementResult> result_vec;

    if (measureBurst)
    {
        wvStartStop.enableBurst();
        wvStartStop.setBusTrigger(Keysight33600A::CH1, phaseNs);
        wvStartStop.setBusTrigger(Keysight33600A::CH2, phaseNs +
delayLength);

    }
    else
    {
        wvStartStop.disableBurst();
        wvStartStop.setPhase(Keysight33600A::CH1, ((phaseNs) *
startStopFreq * 360) / 1000000000.0, true);
        double roundedPhase = round((phaseNs - delayLength) *
startStopFreq * 360 / 1000000000.0 * 1000) / 1000.0;

        wvStartStop.setPhase(Keysight33600A::CH2, roundedPhase, true);


    }


    wvClock.setFrequency(Keysight33600A::CH1, refFreq);
    usleep(50000);


    for (int i = 0; i < nsample; i++)
    {
        if (measureBurst)
        {
            wvStartStop.trigger();
            usleep(10000);
        }
        else
        {
            usleep(2 * 1000000.0 / startStopFreq);
        }

        MeasurementResult s;
        s.dOutput = digitalOutput();
        if (measureAnalog)
```

```
        {
            s.aOutput = dmmAnalogOut.getVoltage();
        }

        s.delayLengthNs = delayLength;
        s.phaseNs = phaseNs;
        s.refFreq = refFreq;

        result_vec.push_back(s);
    }

    lastMeasurement = result_vec;
    return lastMeasurement;
}
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

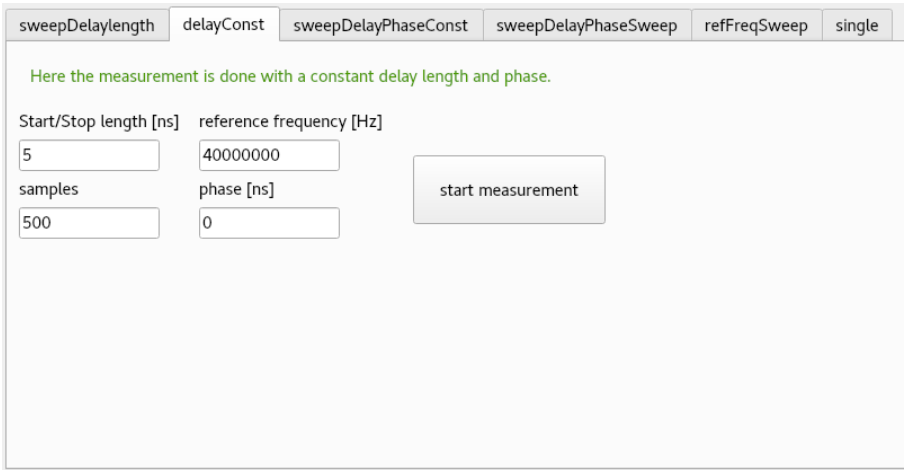

Abbildung 42: GUI "delayConst"

Die nächste Abbildung 42 zeigt, wie das Bedienfeld in der GUI für die Messung „delayConst“ dargestellt wird. Da die Messung hier mit einer konstanten Delaylänge und Phase abläuft, braucht man nur einen Wert einzugeben.

Zudem gibt es auch hier die Möglichkeit, Referenzfrequenz und die Anzahl der Messungen einzugeben. Am Ende dieser Messung soll ein Histogramm der Häufigkeit der Ergebnisse angezeigt werden.

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

std::vector<MeasurementResult>
TDCMeasurement::sweepDelayPhaseConst(double delayLengthStart, double
delayLengthEnd, double delayLengthStep, double phaseNs, int refFreq,
unsigned int nsample)
{
    std::vector<MeasurementResult> result_vec;

    if (measureBurst)
    {
        wvStartStop.enableBurst();
        wvStartStop.setBusTrigger(Keysight33600A::CH1, phaseNs);
    }
```

```
    else
    {
        wvStartStop.disableBurst();
        wvStartStop.setPhase(Keysight33600A::CH1, ((phaseNs) *
startStopFreq * 360) / 1000000000.0, true);
    }

    wvClock.setFrequency(Keysight33600A::CH1, refFreq);

    for (double sweep = delayLengthStart; sweep <= delayLengthEnd; sweep
+= delayLengthStep)
    {
        if (measureBurst)
        {
            wvStartStop.setBusTrigger(Keysight33600A::CH2, phaseNs +
sweep);
        }
        else
        {
            double roundedPhase = round((phaseNs - sweep) * startStopFreq
* 360 / 1000000000.0 * 1000) / 1000.0;

            wvStartStop.setPhase(Keysight33600A::CH2, roundedPhase,
true);

        }

        usleep(100000);

        for (int i = 0; i < nsample; i++)
        {
            MeasurementResult s;

            if (measureBurst)
            {
                wvStartStop.trigger();
                usleep(1000);
            }
            else
            {
                usleep(2 * 1000000.0 / startStopFreq);
            }

            s.dOutput = digitalOutput();
            if (measureAnalog)
            {
                s.aOutput = dmmAnalogOut.getVoltage();
            }

            s.refFreq = refFreq;
            s.delayLengthNs = sweep;
            s.phaseNs = phaseNs;

            result_vec.push_back(s);
        }
    }

    lastMeasurement = result_vec;
    return lastMeasurement;
}

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Der Programmcode für diese Messung ist ähnlich wie in der ersten beschriebenen Messung „sweepDelaylength“. Nur ist der Unterschied, dass die Messungen unsynchronisiert stattfinden. Die beiden Waveform-Generatoren wurden miteinander durch spezielle Ausgänge verbunden, sodass die Phase des Referenzsignals mit der Phase des Starts und Stopps synchronisiert ist. Die in der Arbeit manuell gemessenen Ergebnisse sind unsynchronisiert. Im weiteren Verlauf des Projektes wurde aber festgestellt, dass die Synchronisation der Geräte zu plausibleren Ergebnissen führt. Bei dieser Messung wird deswegen unsynchronisiert gemessen, um einen Vergleich mit den synchronisierten Messungen zu haben.

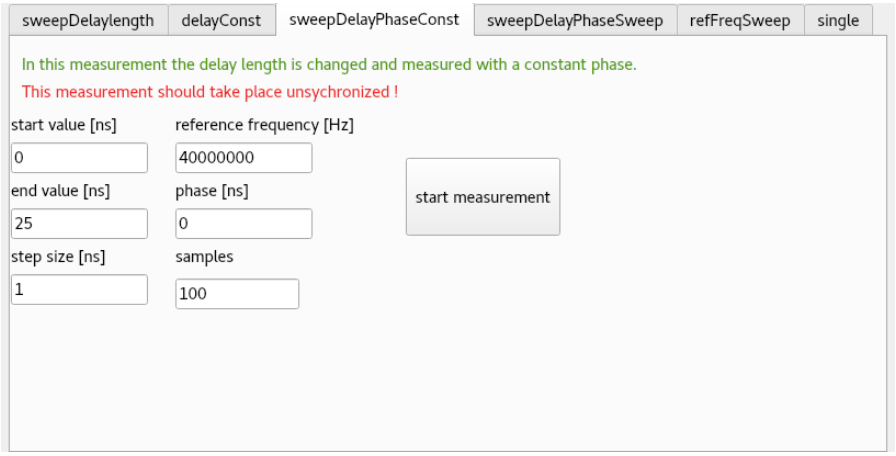

Abbildung 43: GUI "sweepDelayPhaseConst"

Die Abbildung 43 zeigt das Bedienfeld für die Messung „sweepDelayPhaseConst“. Der Nutzer bekommt nach dem Drücken der Starttaste die folgende Meldung in der Abbildung 44.

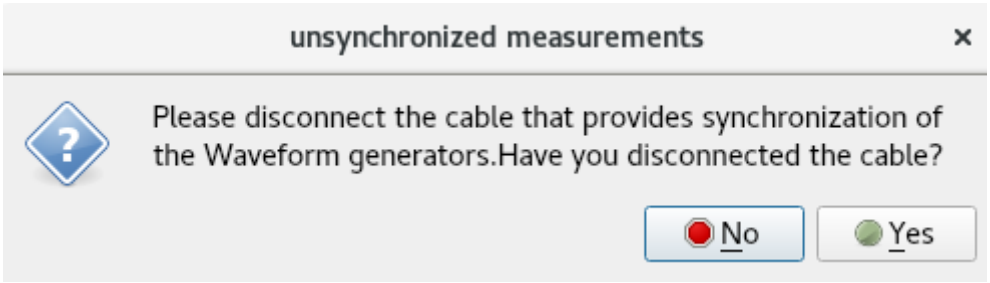

Abbildung 44: GUI "unsynchronized measurements"

Diese Meldung soll den Anwender zur Überprüfung auffordern, ob die Messung wirklich unsynchronisiert stattfindet. Dies bedeutet, dass überprüft werden soll, ob das Kabel, das für die Synchronisation der Geräte zuständig ist, getrennt worden ist.

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
std::vector<MeasurementResult>
TDCMeasurement::sweepDelayPhaseSweep(double delayLengthStart, double
delayLengthEnd, double delayLengthStep, double phaseNsStart, double
phaseNsEnd, double phaseNsStep, int refFreq, unsigned int nsample)
{
    std::vector<MeasurementResult> result_vec;

    if (measureBurst)
    {
        wvStartStop.enableBurst();
    }
    else
    {
        wvStartStop.disableBurst();
    }

    wvClock.setFrequency(Keysight33600A::CH1, refFreq);

    for (double pulsewidth = delayLengthStart; pulsewidth <=
delayLengthEnd; pulsewidth += delayLengthStep)
    {
        for (double phase = phaseNsStart; phase <= phaseNsEnd; phase +=
phaseNsStep)
        {

            if (measureBurst)
            {
                wvStartStop.setBusTrigger(Keysight33600A::CH1, 0 +
phase);
                wvStartStop.setBusTrigger(Keysight33600A::CH2, pulsewidth
+ phase);
                usleep(50000);
            }
            else
            {
                double roundedPhase = round((phase -pulsewidth) *
startStopFreq * 360 / 1000000000.0 * 1000) / 1000.0;
                wvStartStop.setPhase(Keysight33600A::CH1, ((phase) *
startStopFreq * 360) / 1000000000.0, true);
                wvStartStop.setPhase(Keysight33600A::CH2, roundedPhase,
true);

                 std::cout << "sweep " << phase << " Phase auf 3 NKS: "
<< roundedPhase << std::endl;
                //wvStartStop.setPhase(Keysight33600A::CH2, ((phase-
pulsewidth) * startStopFreq * 360) / 1000000000.0, true);
            }

            usleep(50000);


            for (int i = 0; i < nsample; i++)
            {
```

```
                if (measureBurst)
                {
                    wvStartStop.trigger();
                    usleep(1000);
                }
                else
                {
                    usleep(2 * 1000000.0 / startStopFreq);
                }

                MeasurementResult s;
                s.dOutput = digitalOutput();
                s.delayLengthNs = pulsewidth;
                s.phaseNs = phase;

                if (measureAnalog)
                {
                    s.aOutput = dmmAnalogOut.getVoltage();
                }

                s.refFreq = refFreq;
                result_vec.push_back(s);
            }
        }
    }

    lastMeasurement = result_vec;
    return lastMeasurement;
}
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Es werden zwei For-Schleifen für diese Messung benötigt, da wir über zwei Parameter sweepen. In der ersten Schleife findet die Schleife für die Pulsbreite des Start- und Stoppsignals statt und die zweite Schleife führt einen Sweep über die Phase durch.

Die Abbildung 42 zeigt ein Beispiel, wie der Benutzer über die GUI, diese Messung steuern kann. Anschließend werden die Ergebnisse in einem Vektor gespeichert.

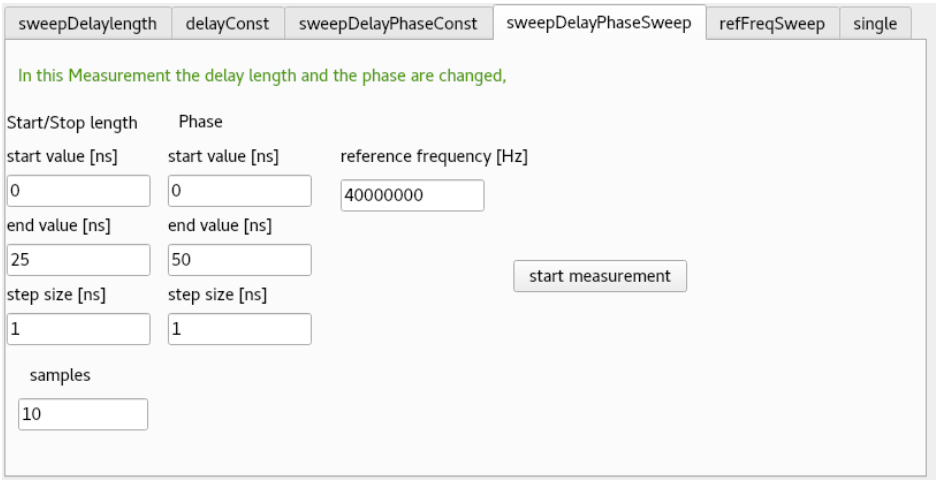

Abbildung 45: GUI "sweepDelayPhaseSweep“

In der Abbildung 45 erkennt man, dass bei der Messung „sweepDelayPhaseSweep“ mehrere Sweeps durchgeführt werden sollen. Für die Start/Stopp-Länge und für die Phasenlänge werden die Start-, End- und Schrittwerte eingetippt. Wie bei den anderen Messungen auch, können die Referenzfrequenz und die Anzahl der Samples eingegeben werden.

++

```
std::vector<MeasurementResult> TDCMeasurement::refFreqSweep(double
delayLength, double refFreqStart, double refFreqEnd, double refFreqStep,
double phaseNsStart,double phaseNsEnd, double phaseNsStep, unsigned int
nsample)
{
    std::vector<MeasurementResult> result_vec;

    if (measureBurst)
    {
        wvStartStop.enableBurst();
    }
    else
    {
        wvStartStop.disableBurst();
    }

    for (double f = refFreqStart; f <= refFreqEnd; f += refFreqStep)
    {
        wvClock.setFrequency(Keysight33600A::CH1, f);

        for (double phase = phaseNsStart; phase <= phaseNsEnd; phase +=
phaseNsStep)
        {

            if (measureBurst)
            {
                wvStartStop.setBusTrigger(Keysight33600A::CH1, phase);
                wvStartStop.setBusTrigger(Keysight33600A::CH2, phase +
delayLength);
            }
            else
            {
                double roundedPhase = round((phase - delayLength) *
startStopFreq * 360 / 1000000000.0 * 1000) / 1000.0;
                wvStartStop.setPhase(Keysight33600A::CH1, ((phase) *
startStopFreq * 360) / 1000000000.0, true);
                wvStartStop.setPhase(Keysight33600A::CH2, roundedPhase,
true);

                //wvStartStop.setPhase(Keysight33600A::CH2, ((phase -
delayLength) * startStopFreq * 360) / 1000000000.0, true);
            }


            for (int i = 0; i < nsample; i++)
            {
                MeasurementResult s;

                if (measureBurst)
                {
```

```
                    wvStartStop.trigger();
                    usleep(1000);
                }
                else
                {
                    usleep(2 * 1000000.0 / startStopFreq);
                }

                s.dOutput = digitalOutput();
                if (measureAnalog)
                {
                    s.aOutput = dmmAnalogOut.getVoltage();
                }


                s.refFreq = f;
                s.delayLengthNs = delayLength;
                s.phaseNs = phase;

                result_vec.push_back(s);
            }
        }
    }

    lastMeasurement = result_vec;
    return lastMeasurement;
}
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Bei dieser Messung findet ein Sweep über die Frequenz und der Phase statt.

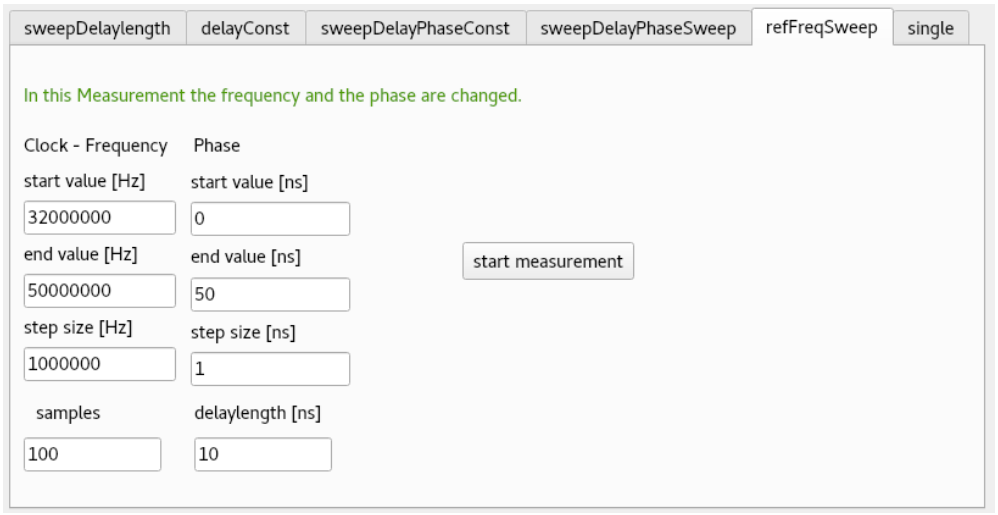

Abbildung 46: GUI "refFreqSweep"

Die Eingabefelder für die Messung „refFreqSweep“ wird in der Abbildung 46 dargestellt. Man hat auch hier die Möglichkeit, für die Phase Start-, End- und Schrittwerte einzutragen. Da hier ein Frequenzsweep stattfindet, muss der Benutzer Frequenzen in die entsprechenden Felder eintragen. Die Messung findet hier mit einer konstanten Delaylänge statt und die Anzahl der Messungen kann ebenfalls eingegeben werden.

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
MeasurementResult TDCMeasurement::singleMeasurement(double
delayLength,double phaseNs, int refFreq)
{
    cout << "Delaylaenge " << delayLength<< "phase " << phaseNs << "Freq
" << refFreq << "Startstopfreq "<< startStopFreq << endl;
    if (measureBurst)
    {
        wvStartStop.enableBurst();
        wvStartStop.setBusTrigger(Keysight33600A::CH1, phaseNs);
        wvStartStop.setBusTrigger(Keysight33600A::CH2, phaseNs +
delayLength);
    }
    else
    {
        wvStartStop.disableBurst();

        wvStartStop.setPhase(Keysight33600A::CH1, ((phaseNs) *
startStopFreq * 360) / 1000000000.0, true);
        double roundedPhase = round((phaseNs - delayLength) *
startStopFreq * 360 / 1000000000.0 * 1000) / 1000.0;

        wvStartStop.setPhase(Keysight33600A::CH2, roundedPhase, true);

    }


    wvClock.setFrequency(Keysight33600A::CH1, refFreq);
    usleep(50000);

    if (measureBurst)
    {
        wvStartStop.trigger();
        usleep(10000);
    }
    else
    {
        usleep(2 * 1000000.0 / startStopFreq);
    }

    MeasurementResult s;
    s.dOutput = digitalOutput();

    if (measureAnalog)
    {
        s.aOutput = dmmAnalogOut.getVoltage();
    }

    s.delayLengthNs = delayLength;
    s.phaseNs = phaseNs;
    s.refFreq = refFreq;

    return s;
}
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Diese Messung ermöglicht es, einzelne Messwerte für die analogen und digitalen Werte aufzunehmen.

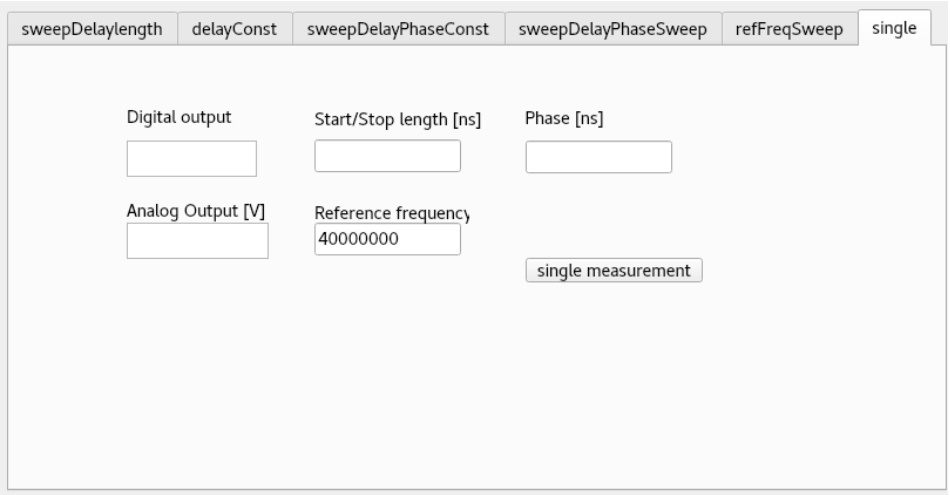

Abbildung 47: GUI "single"

Die Abbildung 47 zeigt das Bedienfeld für die Einzelmessung. Es besteht die Möglichkeit, die Phase, die Delaylänge und die Referenzfrequenz einzugeben. Als Ergebnis werden in den vorgesehenen Feldern die digitalen und analogen Ergebnisse ausgegeben.

Außerdem werden in der TDCMeasurement-Klasse die Ergebnisse in csv-Dateien gespeichert, sodass man die Möglichkeit hat, die Ergebnisse in externen Programmen wie z.B. Excel und Matlab auszuwerten. Folgender Programmcode stellt die Speicherung der Ergebnisse in einer csv-Datei dar. Die Zahlen werden durch Kommata getrennt.

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void TDCMeasurement::writeCSV(std::string filename,
std::vector<MeasurementResult> &data)
{

    ofstream myfile;
    myfile.open(filename);


    myfile << "Delaylaenge,";
    myfile << "DOutput,";
    myfile << "phase,";
    myfile << "aOutput,";
    myfile << "refFreq\n";

    for (auto& value: data)
    {

        myfile << value.delayLengthNs <<",";
        myfile << value.dOutput <<",";
        myfile << value.phaseNs<< ",";
        myfile << value.aOutput << ",";
        myfile << value.refFreq << endl;
```

```
    }

    myfile.close();
}
```

++

Durch Klicken auf den Button „save the last measurement“ erscheint folgende Meldung wie in der Abbildung 48.

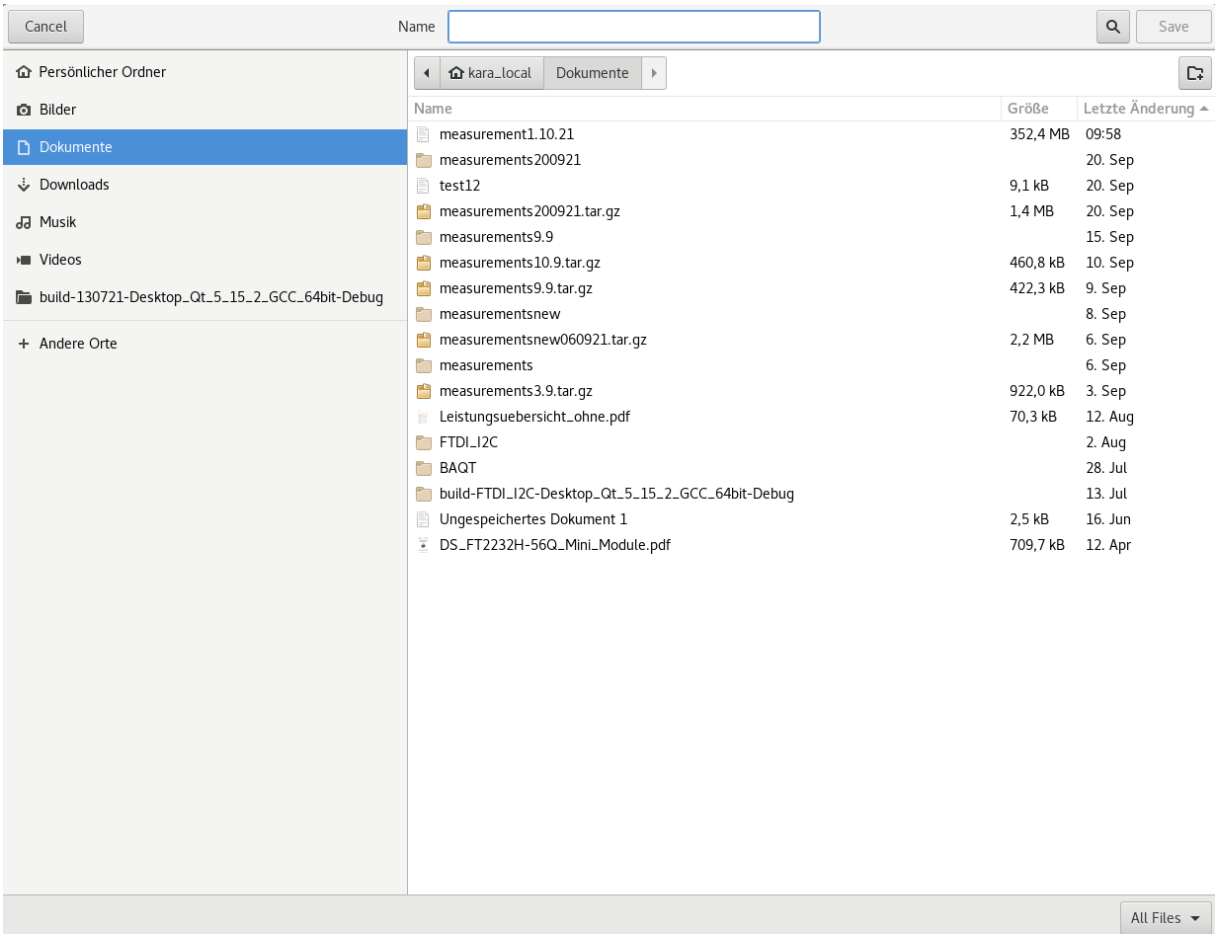

Abbildung 48: GUI "save the last measurement"

Man kann die Auswahl treffen, wo und unter welchem Namen die Datei gespeichert werden soll. Diese wird dann in einem csv-Format gespeichert.

7 Messergebnisse

Das Kapitel zeigt beispielhafte Auswertungen der Ergebnisse der einzelnen Messungen. Die Ergebnisse wurden aus einer csv-Datei in Excel importiert und dort ausgewertet. Während der Bearbeitung des Projektes konnte festgestellt werden, dass die gleichzeitige Aufnahme der Ergebnisse keine Auswirkung auf die digitalen Ergebnisse hat. Es hat nur den Nachteil, dass die Messung länger dauert.

7.1 Messung 1 – „sweepDelayLength“

Der Schwerpunkt dieser Messung liegt auf dem Quantisierungsverhalten des TDC. Erwartet wird ein weitestgehend linearer Verlauf der Quantisierungskennlinie des TDC.

In der Messung werden die analogen und digitalen Repräsentationen der TDC Wandlungsergebnisse in Diagrammen visualisiert. Für die analogen Werte wird die analoge Ausgangsspannung und für die digitalen Werte die digitale Repräsentation des TDC ausgewertet.

Die Ergebnisse im Burst und normalen Modus des Waveform-Generators werden in den folgenden Abschnitten gezeigt. Im Burst-Modus wird als Clocksignal 40 MHz, für die Start/Stopp-Pulswiederholungsfrequenz 10kHz und für die Phase 0ns eingestellt.

Ergänzend zu der Kennlinie des TDC werden auch die DNL (Differenzielle Nichtlinearität) und INL (Integrale Nichtlinearität) des TDC für diese Messung ausgewertet.

Das Diagramm in der Abbildung 49 zeigt den idealen, analogen und digitalen Kurvenverlauf des TDC. Es fällt auf, dass die Kurve der digitalen und analogen Darstellung der TDC Messung fast parallel verlaufen. Bis 2ns gibt es einen Offset und zwischen 10-15ns gibt es eine stärkere Abweichung von der idealen Linie des TDC.

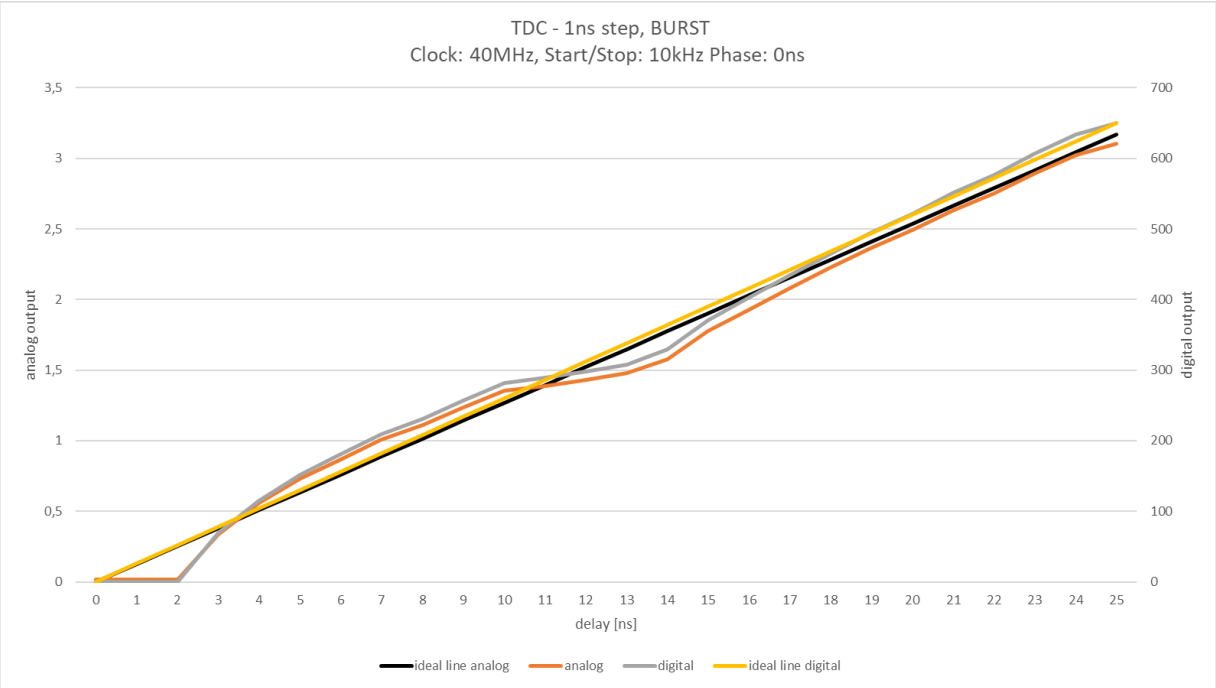

Abbildung 49: Diagramm TDC,Burst

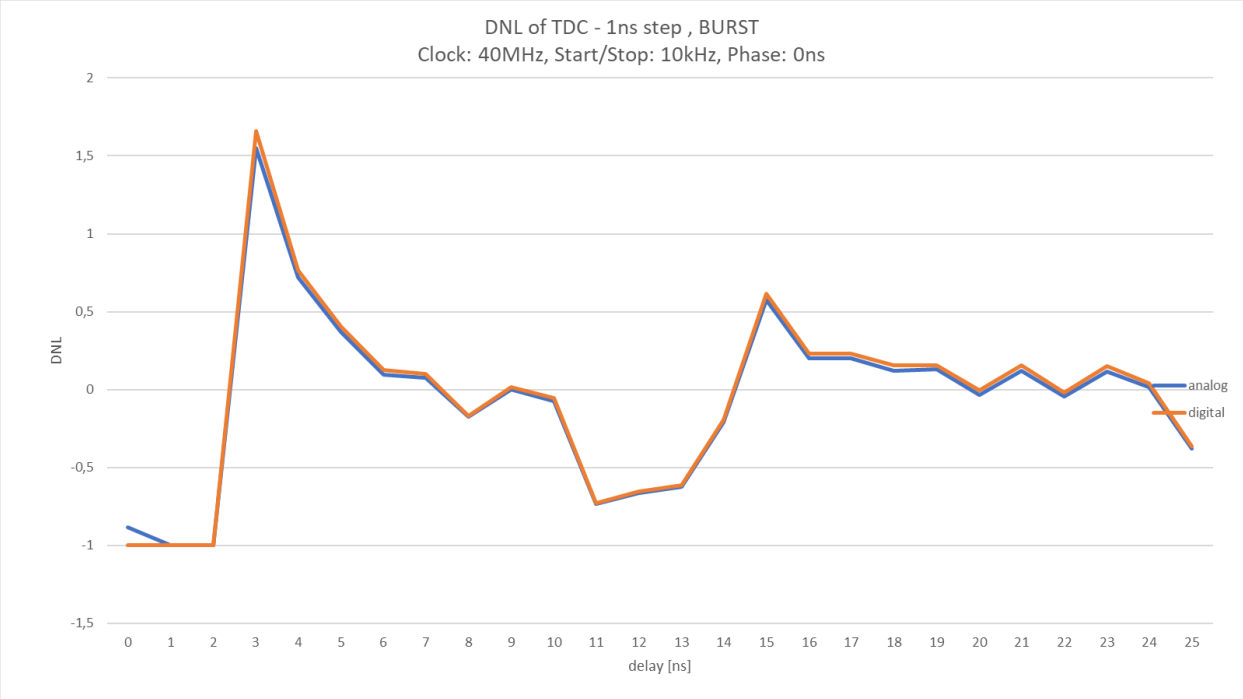

Abbildung 50: DNL, Burst

Das Diagramm in der Abbildung 50 stellt die DNL des TDC in 1ns Schritten im Burst-Modus dar und wird mit folgender Formel 12 berechnet.

$$DNL = \left(\frac{TDC(n) - TDC(n-1)}{\frac{t_{step}}{\frac{25ns}{650}}} \right) - 1 \tag{12}$$

Die DNL beschreibt, wie hoch die Fehler in der Breite der Quantisierungsschritte des TDC ist. Die lokale Steigung der Übertragungsfunktion wird mit der idealen Steigung verglichen.

Ein DNL von 0 sagt aus, dass der TDC ordnungsgemäß funktioniert. Die y-Achse beschreibt die prozentualen Steigungsfehler der Schrittweite des TDC. Zu Beginn der Kurve sind es positive Fehler und zum Ende sind es negative Fehler.

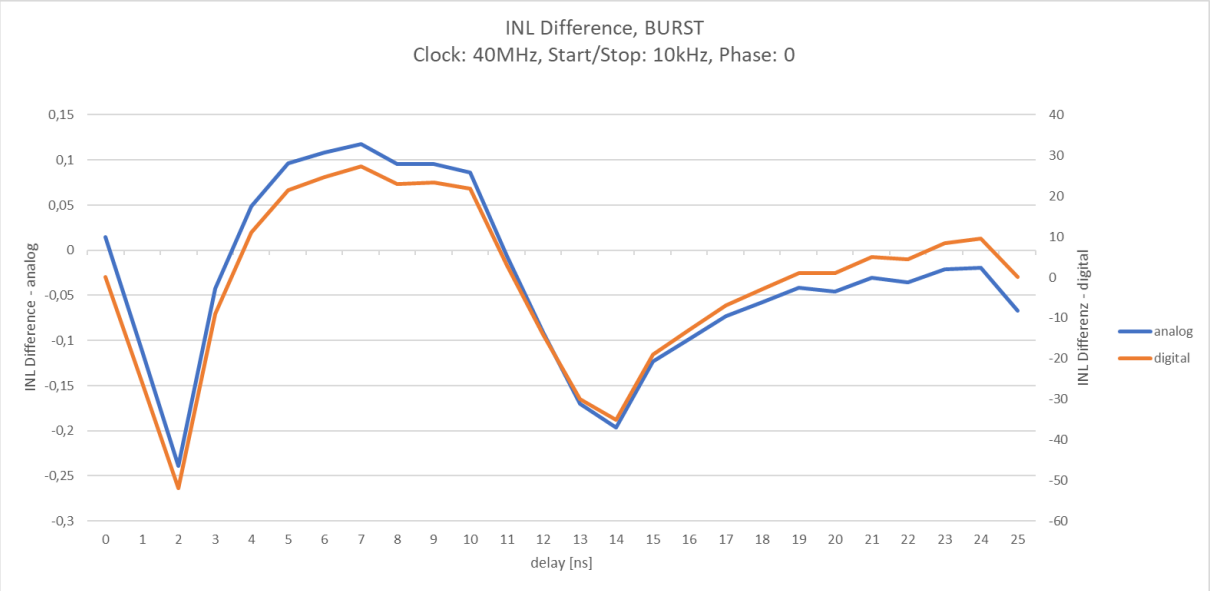

Abbildung 51: INL Differenz, Burst

Bei dem Diagramm in Abbildung 51 für die INL Differenz erkennt man, dass zu Beginn ein negativer INL Wert vorhanden ist, der in der Mitte des Messbereiches eher positive Werte annimmt. Für diese Berechnung wird die Formel 14, mit der man die absolute Messabweichung ermitteln kann, verwendet.

$$INL_{dif} = TDC(mean) - TDC(ideal) \tag{14}$$

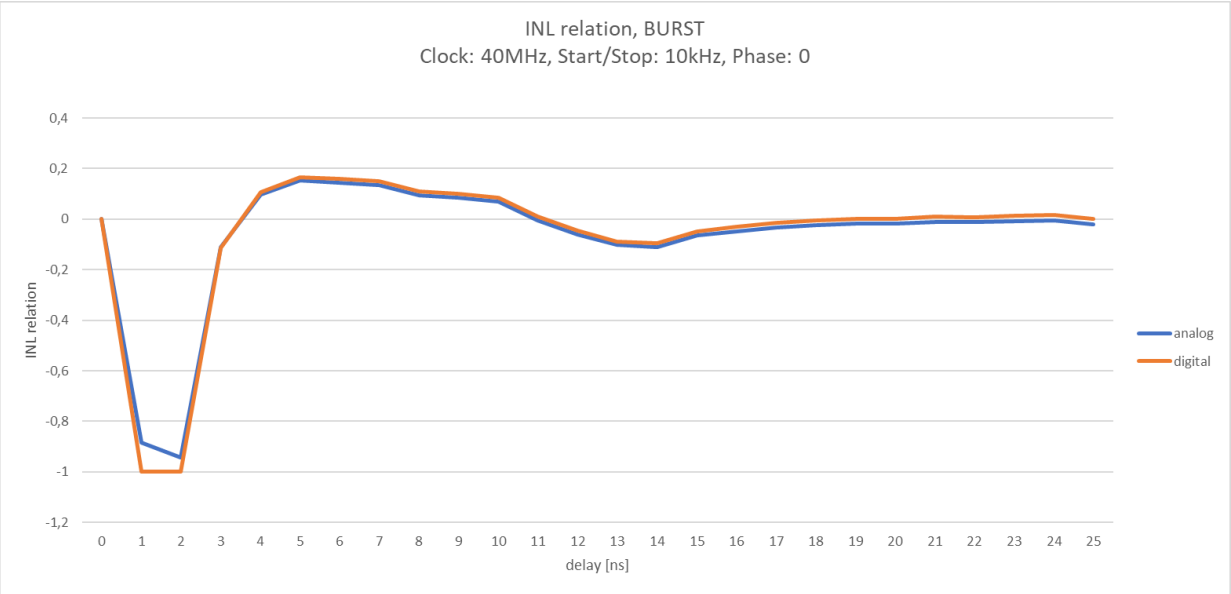

Abbildung 52: INL Verhalten, Burst

$$INL_{rel} = \left(\frac{TDC(mean)}{TDC(ideal)}\right) - 1 \tag{13}$$

Das in der Abbildung 52 dargestellte Diagramm beschreibt mit der Formel 13 die Verhältnisse der ausgewählten Werte und der idealen Werte zueinander. Hierbei wird die relative Messabweichung untersucht. Am Anfang ist das Verhältnis bis ca. 2ns negativ und ab diesem Zeitpunkt steigt es bis 0 an und bleibt ungefähr bis 25ns bei diesem Verhältnis.

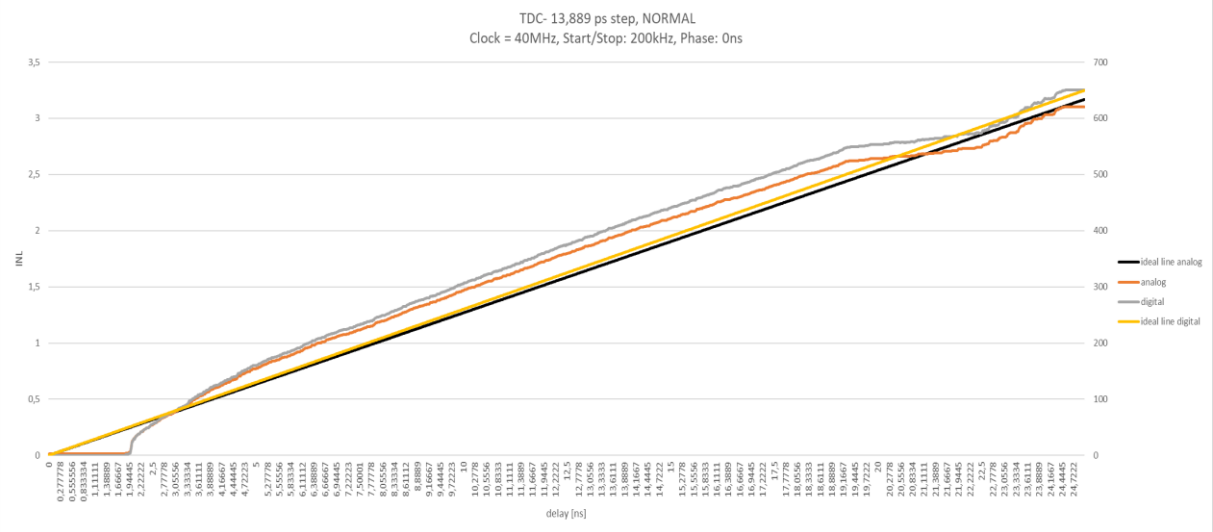

Abbildung 53: TDC, Normal

Das Diagramm in Abbildung 53 zeigt die TDC-Ergebnisse im normalen Modus. In diesem Modus wird ein Sweep über die Delayzeit mit einer Schrittweite von 13.889ps durchgeführt. Auch hier erkennt man den Offset von ca. 2ns. Anhand des Diagramms wird deutlich, dass sich die Linien für die digitalen und analogen Ergebnisse zum größten Teil über der jeweiligen idealen Linie befinden. Die Steigungen sind allerdings größer als erwartet.

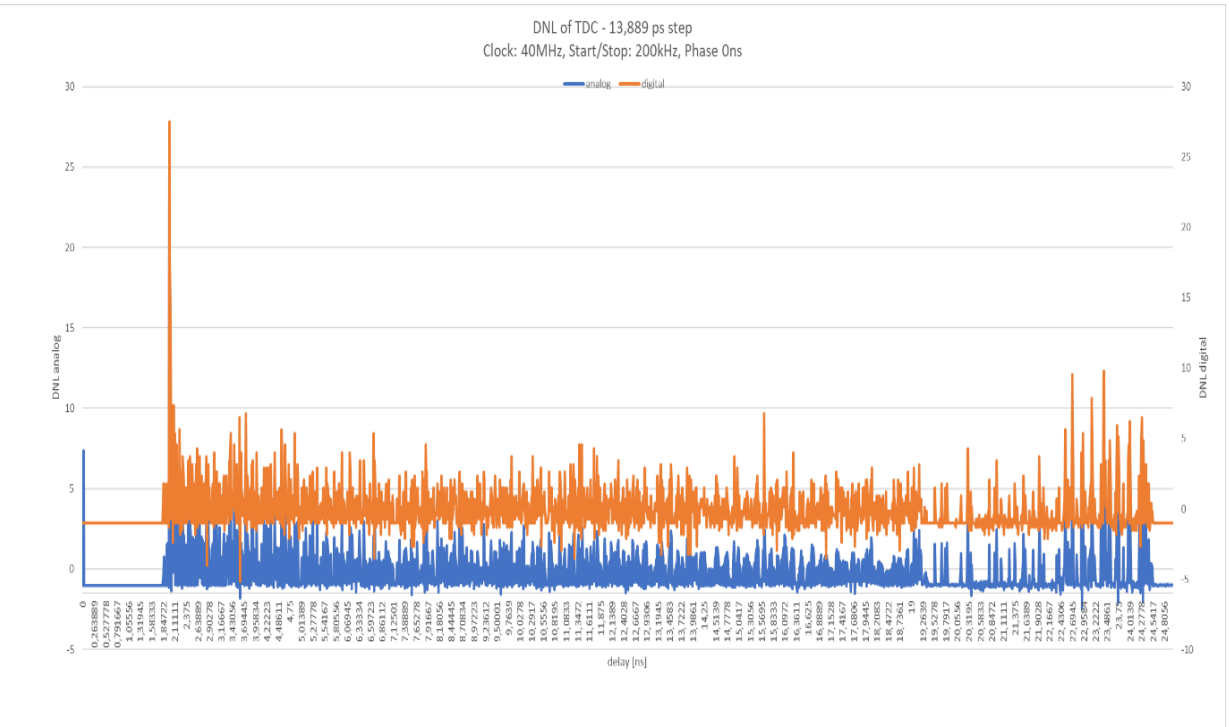

Abbildung 54: DNL, Normal

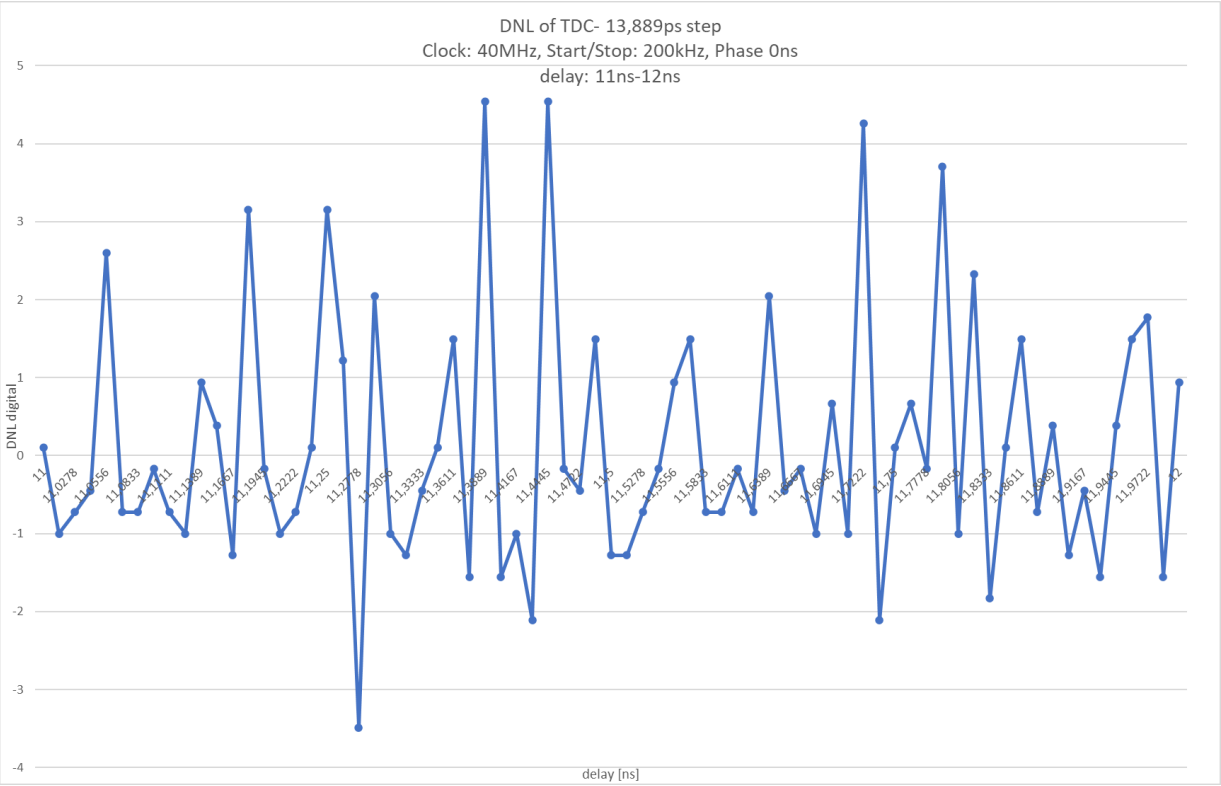

Abbildung 55: DNL 11ns-12ns, Normal

Die Diagramme in den Abbildungen 54 und 55 zeigen die DNL-Auswertungen im normalen Modus, die mit der Formel 12 ausgewertet worden sind. Da das Diagramm 54 sehr unübersichtlich ist, wird ein Ausschnitt des Diagramms in dem Verzögerungsbereich von 11ns-12ns dargestellt. Die Punkte in den Diagrammen zeigen, wie hoch prozentual die lokale Steigung an der Stelle von der idealen Steigung abweicht.

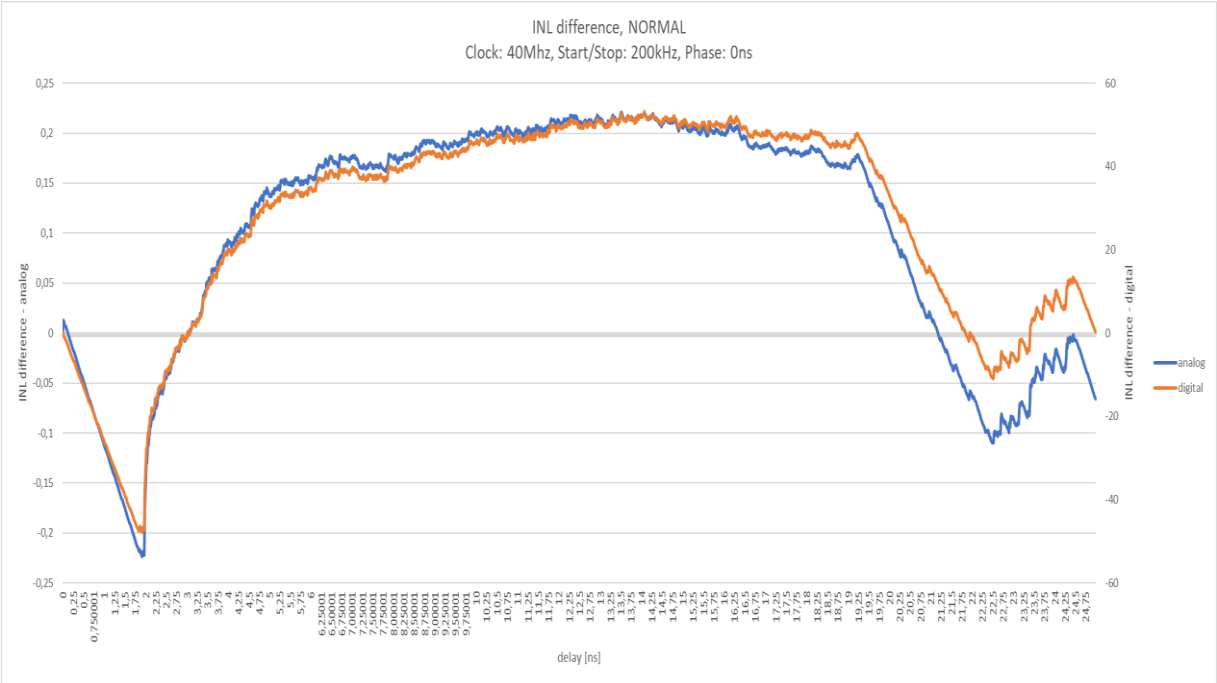

Abbildung 56: INL Differenz, Normal

Für die Auswertung des Diagramms in der Abbildung 56 wird die Formel 13 verwendet. Zu Beginn erkennt man eine absolute Messabweichung von -50 LSB. Dieses Verhalten hängt mit dem Offset von 2ns des TDC zusammen. Nach diesem Verhalten steigt die Linie bis +50 LSB, weil die Wandlungszeit immer ca. 2ns zu hoch liegt.

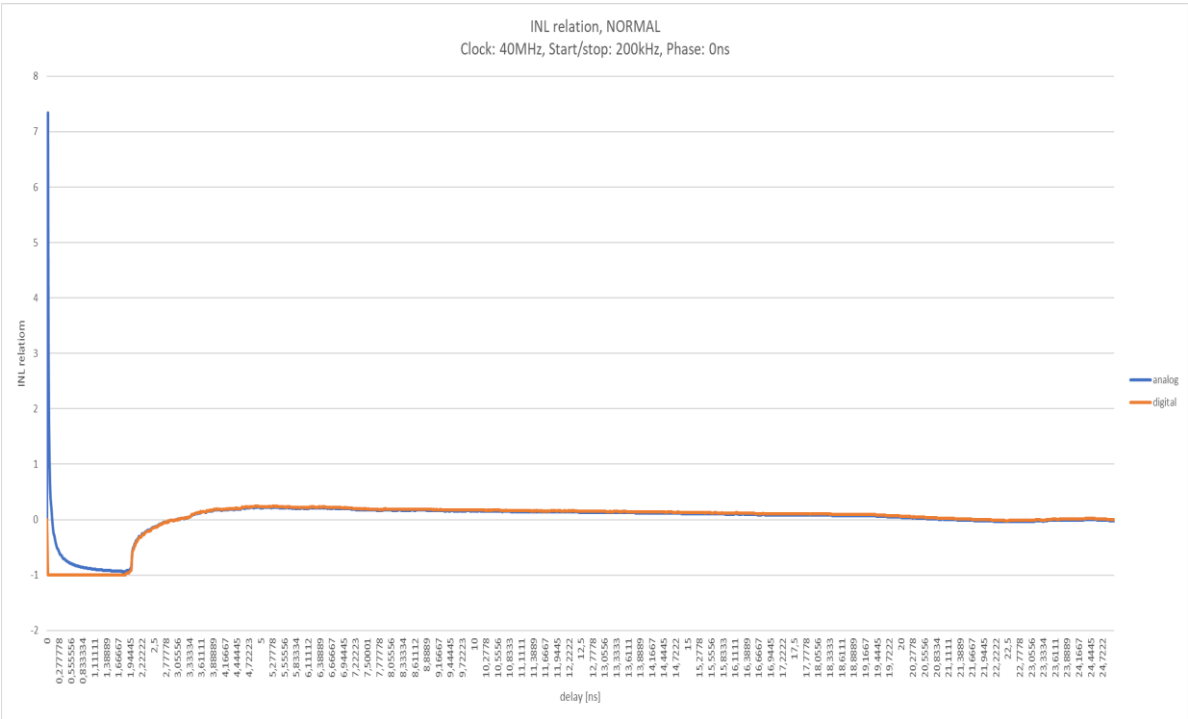

Abbildung 57: INL Verhältnis, Normal

Das Diagramm in der Abbildung 57 wird mit der Formel 14 dargestellt. Das Ergebnis für den digitalen Teil beschreibt die relative Messabweichung. Der Verlauf der digitalen Ergebnisse nimmt am Anfang einen Fehler von 100% an und nach dem Offset liegt der Fehler bei ca. 10 %. Zum Schluss der Linie ist der Fehler bei 0.

Ein möglicher Grund für dieses Fehlverhalten könnte ein Spannungsabfall über der Delayline sein. Dies führt dazu, dass am Ende der Delayline eine kleinere Spannung ankommt als am Anfang. Manche Delayelemente werden kürzer eingeriegelt als die anderen Delayelemente.

7.2 Messung 2 – „delayConst“

Mit dieser Messung wird mit konstanter Delaylänge und Phase gemessen. Diese Messung soll verdeutlichen, wie stark die Variation der gemessenen Ergebnisse ist. Zuerst werden die Ergebnisse im Burst-Modus aufgenommen und dargestellt.

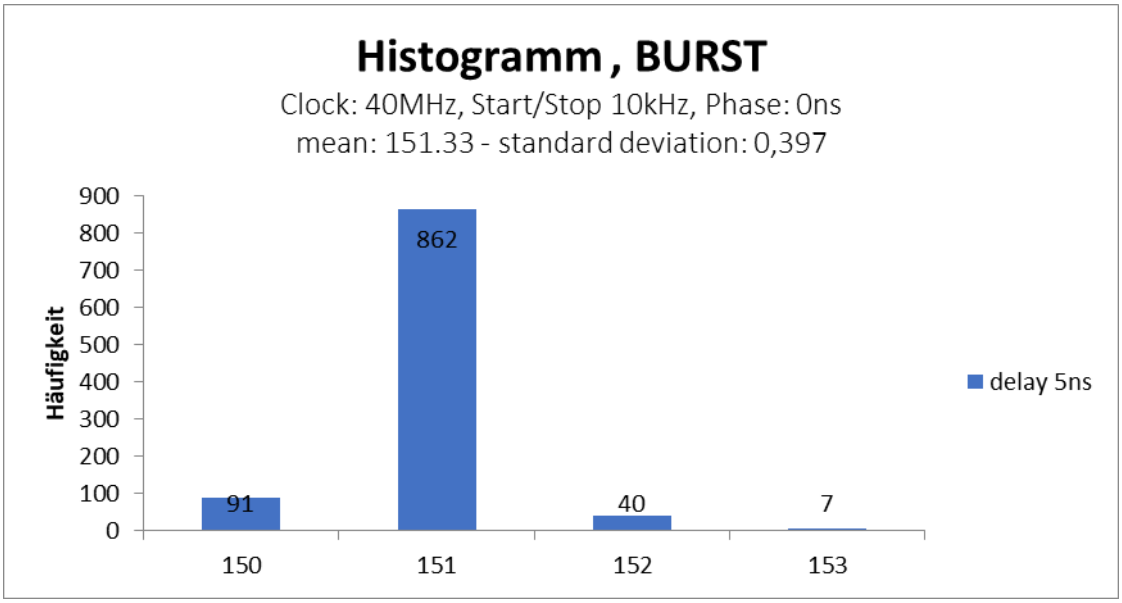

Abbildung 58: Histogramm 5ns, Burst

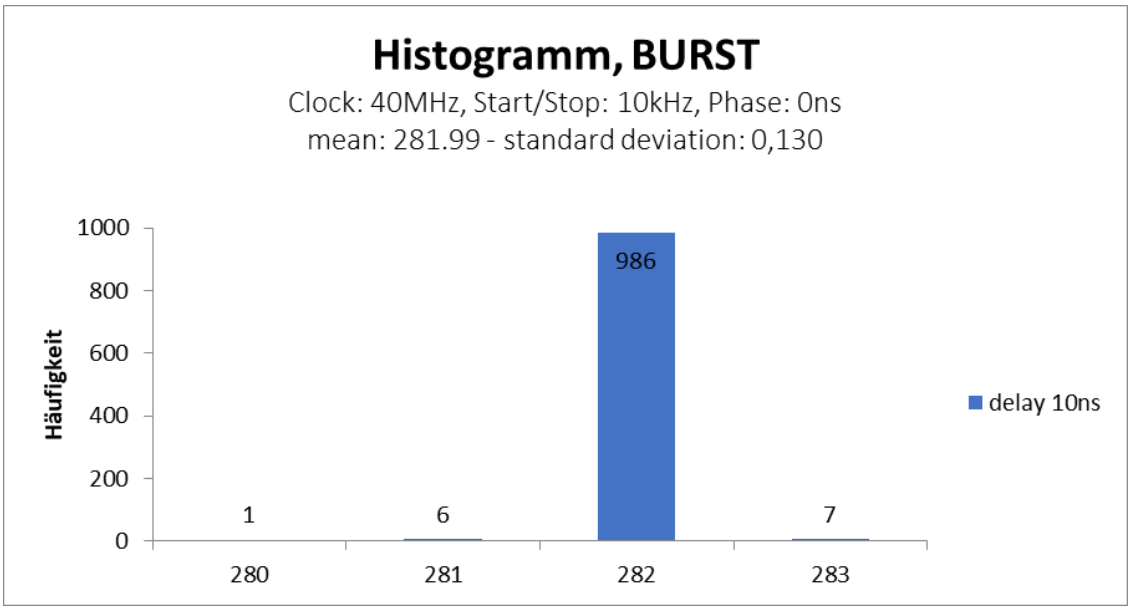

Abbildung 59: Histogramm 10ns, Burst

Die Histogramme in den Abbildungen 58 und 59 zeigen, dass die Ergebnisse bei 1000 Wiederholungen der Ergebnisse sehr geringe Variationen haben. Bei 5ns Verzögerung kommt der digitale Wert 151 862 mal vor und bei 10ns resultiert ein Ergebnis von 986 bei 1000 Wiederholungen. Die Standardabweichungen für die Messungen sind sehr gering.

Im normalen Modus werden die gleichen Einstellungen wie im Burst-Modus übernommen und ausgewertet.

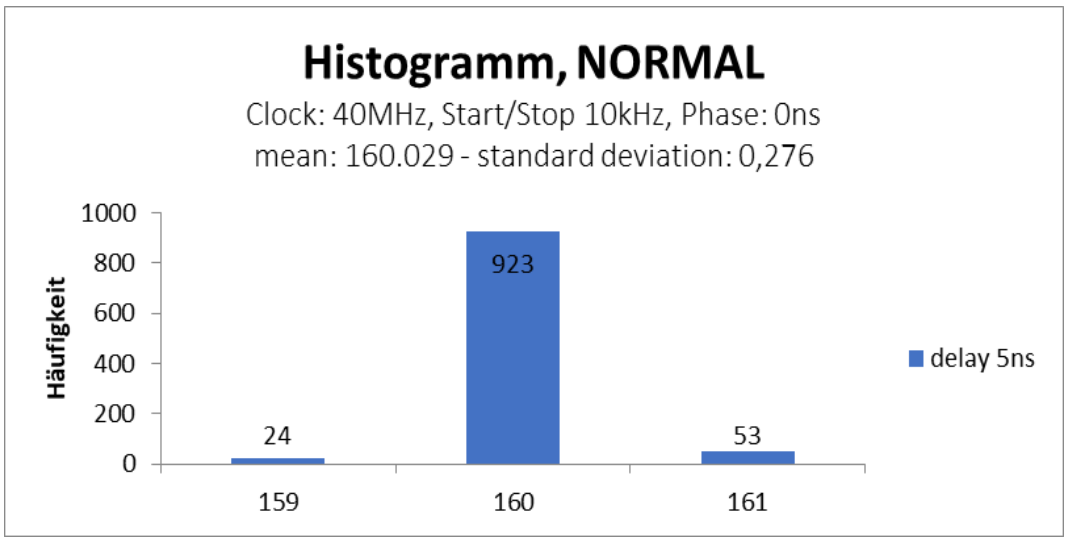

Abbildung 60: Histogramm 5ns, Normal

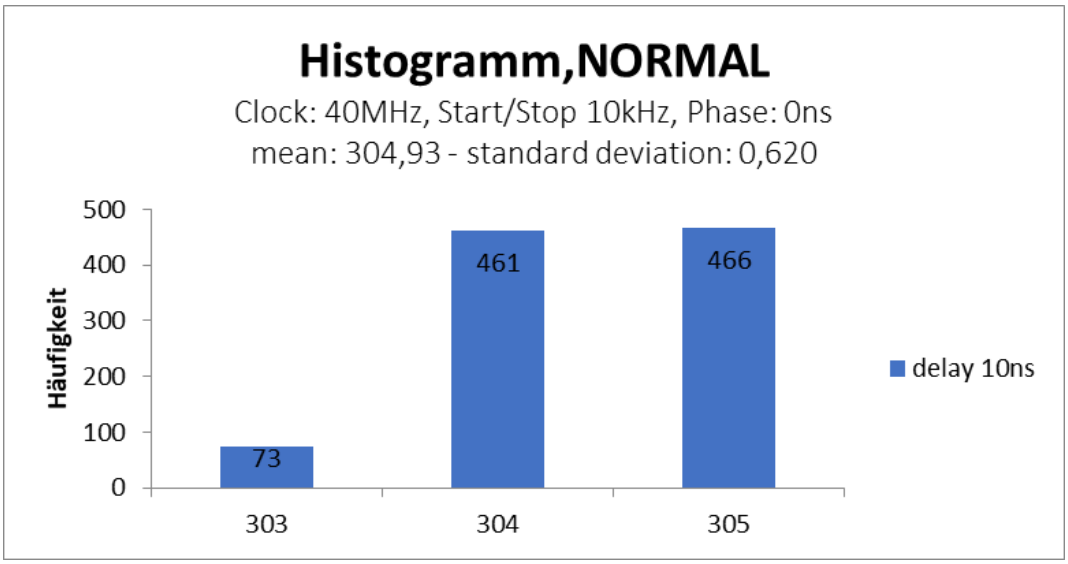

Abbildung 61: Histogramm 10ns, Normal

Die digitalen Ergebnisse der Histogramme in den Abbildungen 60 und 61 zeigen nicht die gleichen Resultate wie im Burst-Modus. Die Phase ist bei den Modi jedoch unterschiedlich. Die Ergebnisse unterscheiden sich und liegen für 5ns Delay-Zeit bei ca. 10 LSB und für 10ns Verzögerung bei 20 LSB.

Die Messung „sweepDelayPhaseSweep“, die im weiteren Verlauf dieser Arbeit vorgestellt wird, bestätigt dieses Fehlverhalten, da in der Messung ein Phasensweep durchgeführt wird.

7.3 Messung 3 – „sweepDelayPhaseConst“

Diese Messung ist unsynchronisiert durchgeführt worden und verdeutlicht den Unterschied zu den vorgestellten synchronen Messungen. Als erstes werden die Histogramme im Burst-Modus vorgestellt.

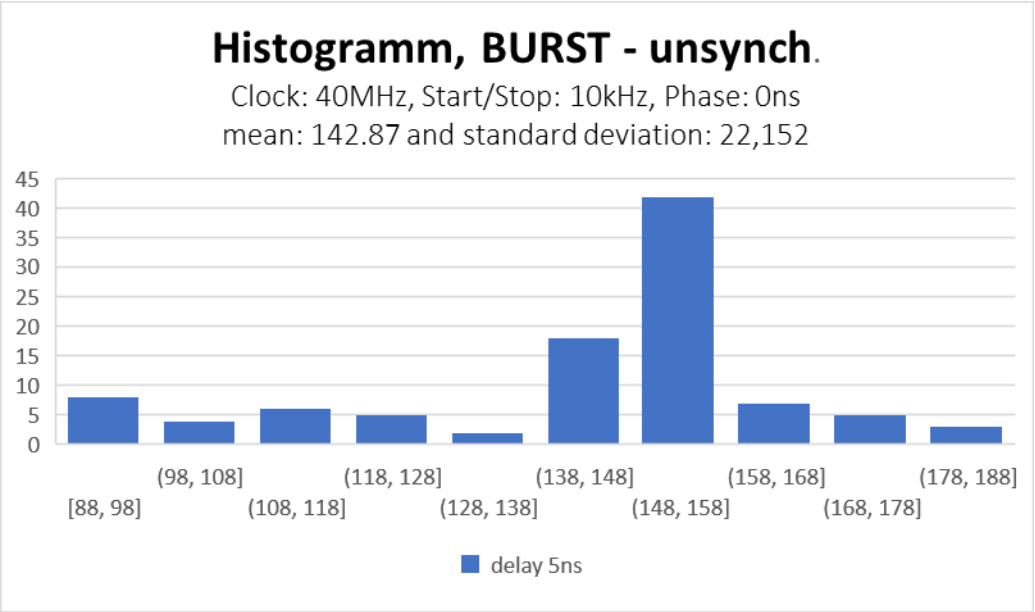

Abbildung 62: Histogramm 5ns unsynch., Burst

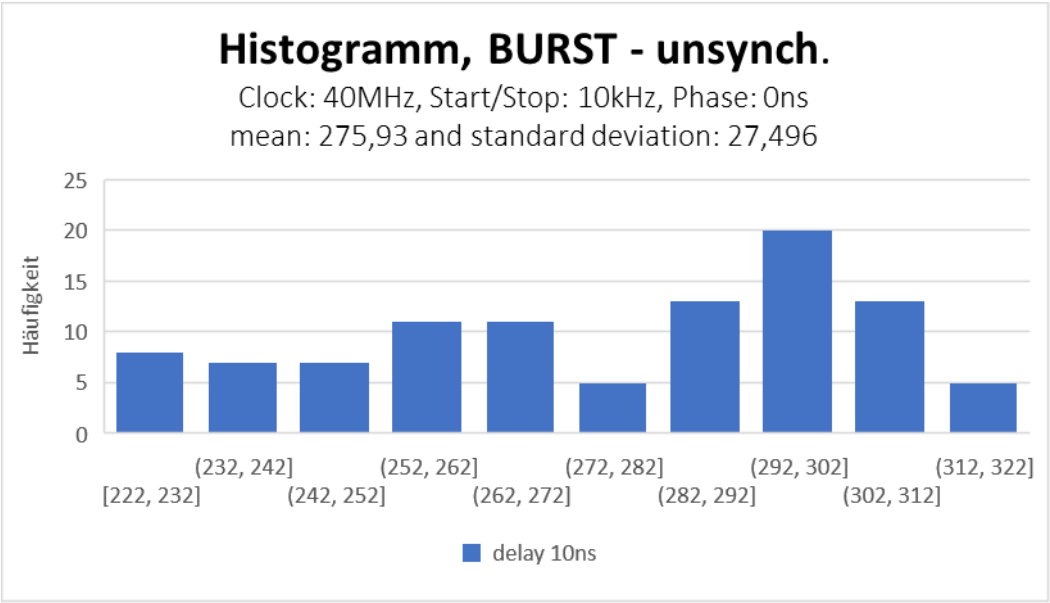

Abbildung 63: Histogramm 10ns unsynch., Burst

Bei Betrachtung der Histogramme in den Abbildungen 62 und 63 fällt sofort auf, dass die Streuung der Ergebnisse stark angestiegen ist. Außerdem ist die Standardabweichung für die beiden ausgewerteten Streuungen sehr hoch. Eine asynchrone Messung ist sehr stark von der Abhängigkeit des Wandlungsergebnisses von der Phasenverschiebung betroffen und durch diesen Effekt variiert die Auswertung der Ergebnisse übermäßig.

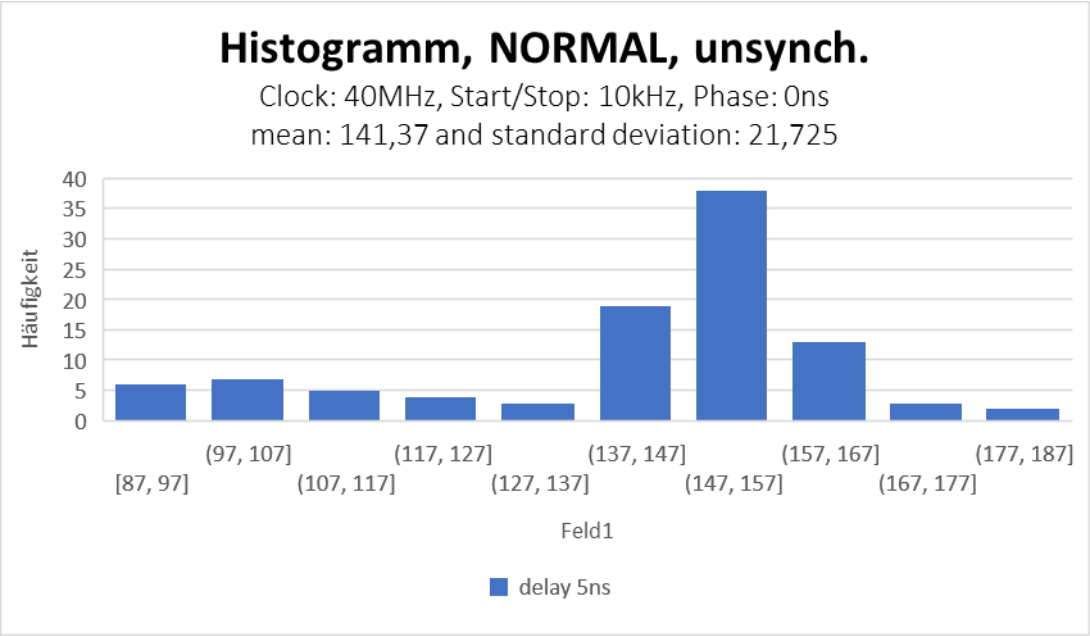

Abbildung 64: Histogramm 5ns unsynch., Normal

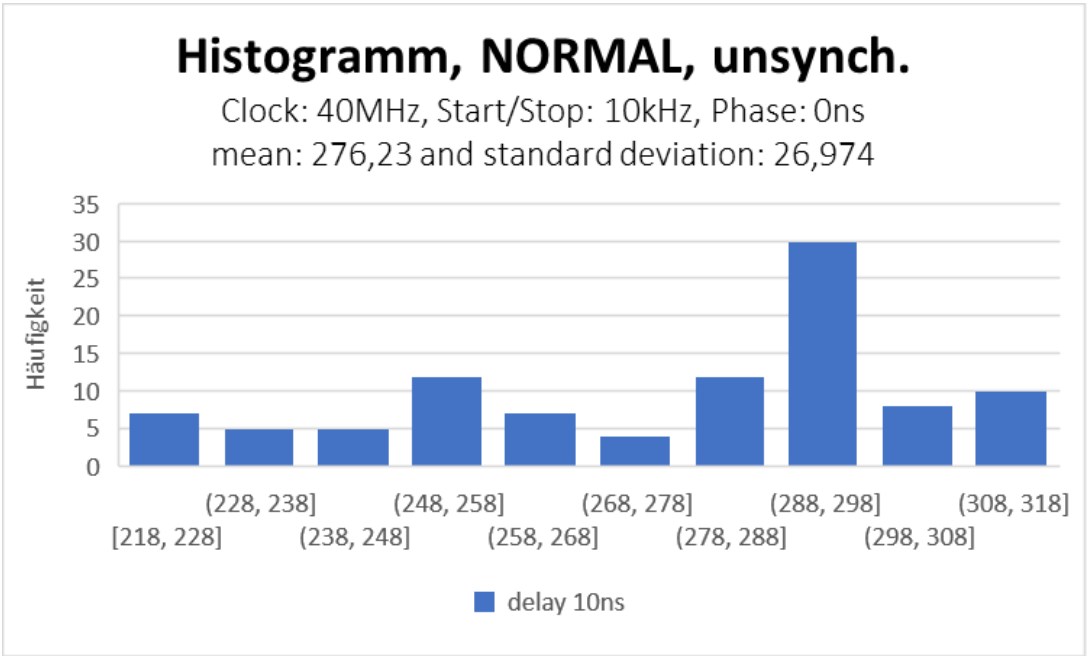

Abbildung 65: Histogramm 10ns unsynch., Normal

Auch im normalen Modus verdeutlichen die Histogramme in den Abbildungen 64 und 65, dass die Streuung der Werte und die Standardabweichung, wegen der zufälligen Phasenverschiebung während den Messungen sehr hoch sind.

7.4 Messung 4 – „sweepDelayPhaseSweep“

Bei dieser Messreihe soll gezeigt werden, welchen Einfluss die Phasenvariation auf die Messungen hat. Die ersten Messungen der Messreihe wurden im Burst-Modus aufgenommen.

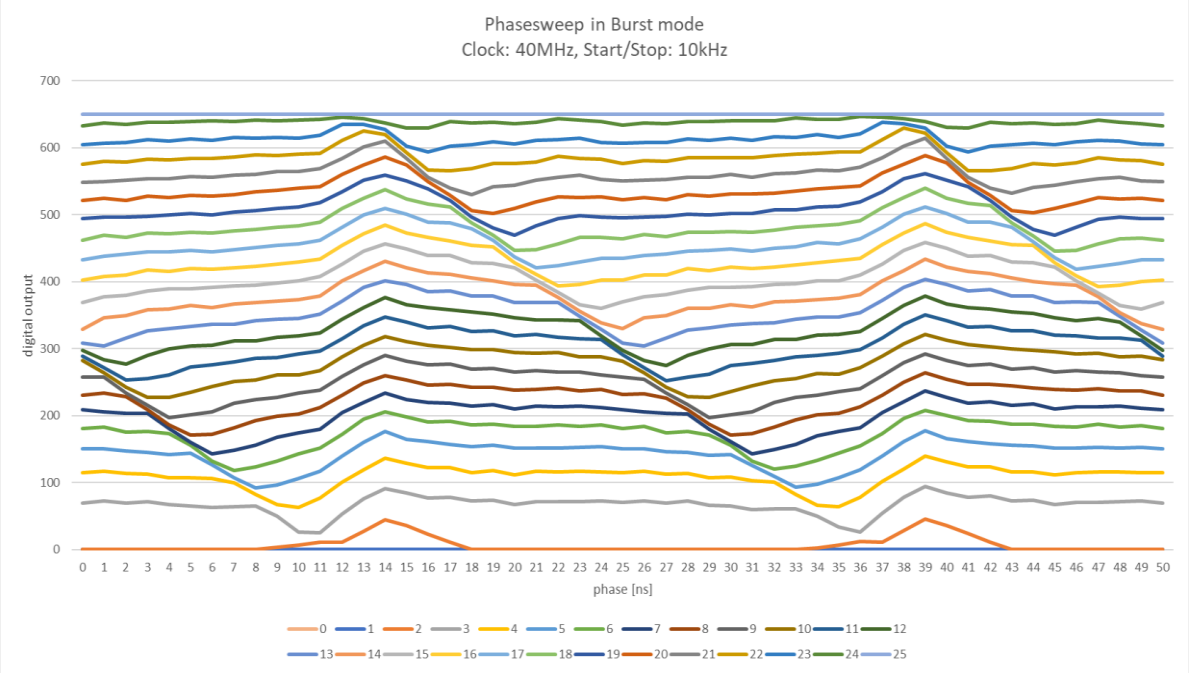

Abbildung 66: Phasensweep, Burst

Das Diagramm in der Abbildung 66 zeigt auf der x-Achse die Änderung der Phase von 0ns bis 50ns in 1ns Schritten und zu der entsprechenden Phase werden die digitalen Ergebnisse ausgewertet. Die einzelnen Kurvenschare zeigen die Verzögerungen des Start- und Stoppsignals von 0ns bis 25ns.

Es ist zu erkennen, dass die Phasenverschiebung Einfluss auf die Ergebnisse hat. Die Messung bei 5ns Delay-Zeit fängt bei einer Phase von 0ns mit dem digitalen Durchschnittswert von 151.9 an und fällt leicht bis zu einer Phase von 5ns. Ab dieser Verzögerung fällt sie stark bis 8ns ab und steigt ab dort wieder bis 14ns an. Je größer die gewählten Verzögerungen gewählt werden, desto kleiner ist der Wert für die Phasenverschiebung bei der das Wandlungsergebnis zu sinken beginnt. Ein möglicher Grund hierfür ist, dass in der Delay Line ein Bereich vorhanden ist, welcher nicht richtig ausgelesen werden kann. Eine weitere Hypothese lautet, dass die Delay-Line des TDC sich auf eine Verzögerungszeit kleiner als 25ns einschwingt. Dadurch gibt es einen gewissen Zeitraum während jeder Periode des Referenztaktes, in dem der TDC nicht sensitiv ist.

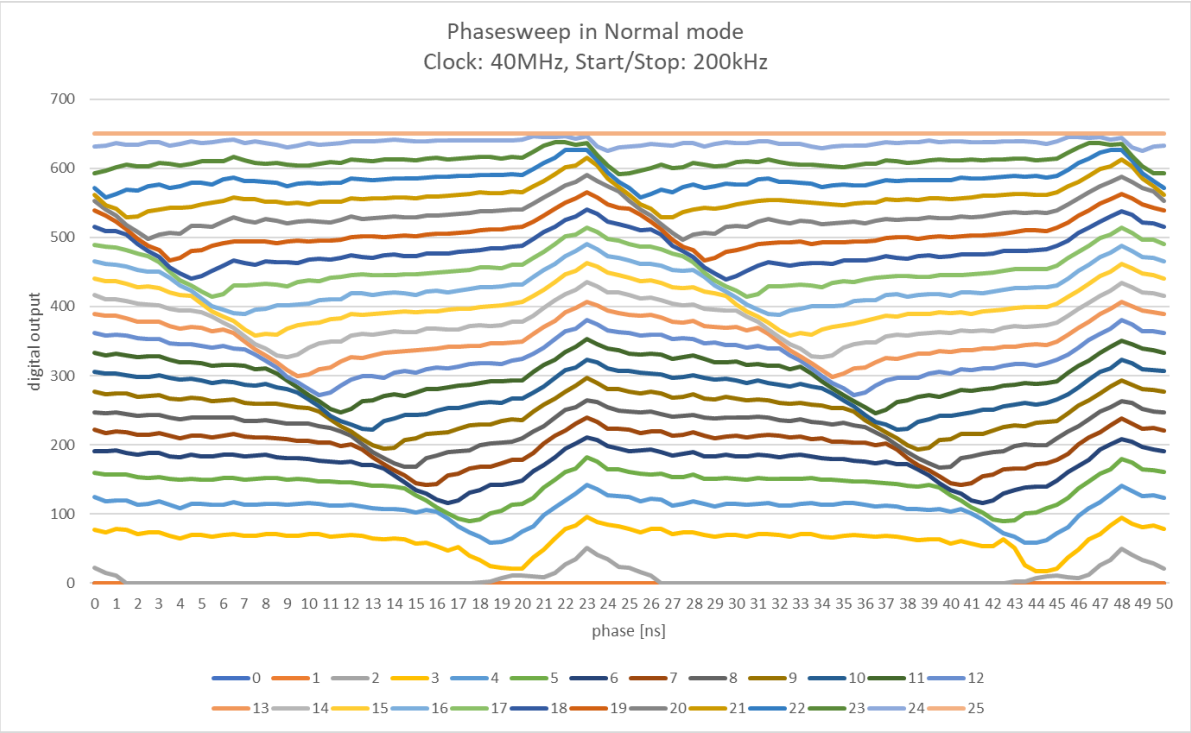

Abbildung 67: Phasensweep, Normal

Abbildung 67 zeigt, dass im normalen Modus die Phasenverschiebung identisch aussieht. Bei diesen Messungen wurde die Phase in 0.5ns Schritten von 0ns bis 50ns verändert. Im Vergleich zwischen den Ergebnissen von Burst und normalen Modus fällt auf, dass die Verläufe der einzelnen Messungen nach rechts verschoben sind. Dies weist darauf hin, dass sich andere Phasen zwischen dem Referenztakt und der steigenden Flanke des StartStop-Signals einstellen. Diese Phasenunterschiede würden auch die Unterschiede zwischen Normalmodus und Burstmodus bei den anderen Messungen erklären.

7.5 Messung 5 - „refFreqSweep“

Bei dieser Messung soll überprüft werden, welchen Einfluss eine Änderung des Referenztaktes auf das Wandlungsergebnisses des TDC hat. Das Diagramm in der Abbildung 68 zeigt Mesungen im Burst-Modus. Hier ist zu beachten, dass die Ergebnisse, je nach Phase unterschiedlich stark schwanken. Bei den Frequenzen für 40MHz und 50MHz erkennt man einen ähnlichen Verlauf. Bei den anderen Kurven sind die Schwankungen geringer.

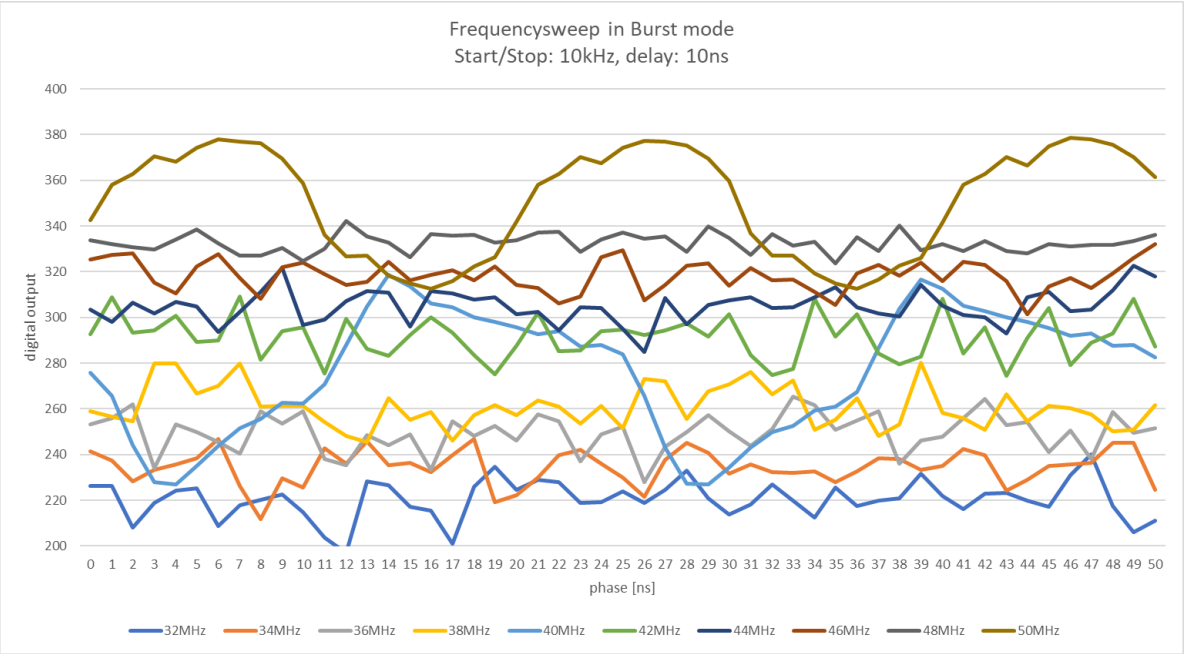

Abbildung 68: Frequenzsweep, Burst

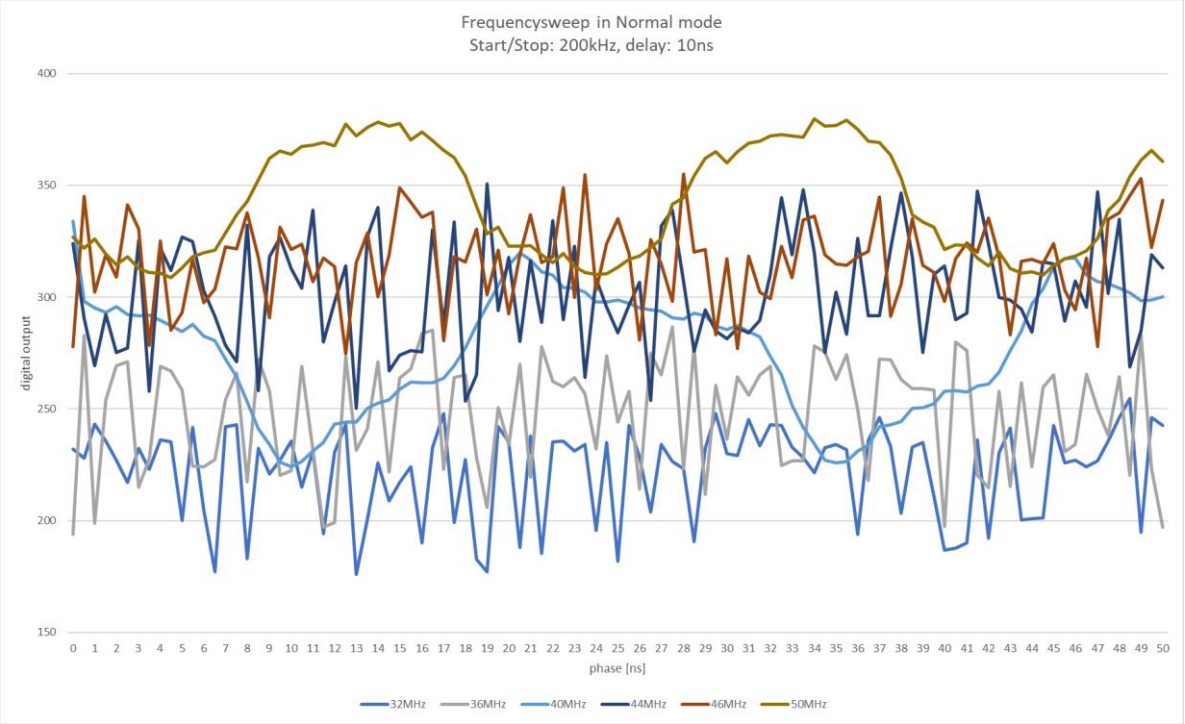

Abbildung 69: Frequenzsweep, Normal

Um die Übersicht zu behalten, wurden im Diagramm 69 nicht alle Messungen für alle aufgenommenen Frequenzen gezeigt. Auch hier im normalen Modus sieht man den gleichen Effekt, dass die Ergebnisse schwanken. Außerdem fällt erneut auf, dass sich anscheinend wieder eine Verschiebung in der Phase zwischen Burst und Normal-Modus einstellt.

8 Fazit

Im Rahmen dieser Bachelorarbeit sollte ein Time-to-Digital Converter auf seine Funktion hin überprüft und seine Wandlungseigenschaften charakterisiert werden. Durch die Entwicklung eines automatisierten Testprogramms sollten aussagekräftige und reproduzierbare Messergebnisse mit wenig Aufwand erstellt werden können. Das automatisierte Messprogramm wurde unter Einbindung der benötigten Messgeräte, des FT2232-56Q Mini-Moduls und der Qt Software mit einer GUI realisiert.

Zusammenfassend lässt sich sagen, dass die festgelegten Ziele dieses Projektes erreicht worden sind. Der TDC wurde zuerst manuell gemessen und erste Messergebnisse erfasst. Anschließend wurde der Test schrittweise automatisiert. Eine anspruchsvolle Aufgabe bestand darin, ein gemeinsames Konzept zu erstellen, welches alle gewünschten Messungen umfasst.

Es ist festgestellt worden, dass der TDC ordnungsgemäß funktioniert, jedoch Abweichungen vom idealen Verhalten aufweist. Die Messergebnisse in dieser Arbeit wiesen an manchen Stellen Auffälligkeiten auf. Durch den Test im Burst und normalen Modus, wurden zwei verschiedene Messverfahren getestet und verglichen. Es ist nachgewiesen worden, dass die Phase einen großen Einfluss auf die Wandlungsergebnisse des TDC besitzt. Variiert die Phase zufällig, werden die Messergebnisse stark verfälscht. Idealerweise sollte nur die Delaylänge bzw. der Start- und Stopppuls einen Einfluss auf die Messungen haben. Die Auswertungen haben jedoch gezeigt, dass zusätzlich die Phasenlage der Start/Stop-Signale zum Referenztakt das Messergebnis stark beeinflusst. Start-, Stopp- und Referenztakt müssen auf der Grundlage von synchronisierten Oszillatoren erzeugt werden.

Abschließend ist festzuhalten, dass das entwickelte Testsystem zukünftig auch für andere entwickelte TDC angewendet werden kann.

Literaturverzeichnis

[1] Krause, Matthias (2018): *Entwicklung eines Delay-Locked Loop basierten Time-to-Digital Converters mit Sub-Gate-Delay Auflösung für eine Time-of-Flight Anwendung in 360nm CMOS Technologie*, Fachhochschule Dortmund.

[2] Lippold, Markus (2018): *Entwurf einer Delay-Locked Loop für die Nutzung als Time-to-Digital Converter in einer Time-of-Flight Anwendung in 350nm CMOS Technologie*, Fachhochschule Dortmund

[3] Pille, Andreas (2021): *Optimierung eines Linear Interpolation Time-to-Digital Converters mit Sub-Gate Delay für eine Time-of-Flight Anwendung*, Fachhochschule Dortmund

[4] Henzler, Stephan (2010): *Time-to-Digital Converters.* Springer Netherlands, 1. Auflage

[5] Tektronix UK Ltd. (Online – Abrufdatum 20.09.2021)

https://de.tek.com/tektronix-and-keithley-digital-multimeter/dmm6500

[6] Conrad Electronic SE (Online - Abrufdatum 22.09.2021)

https://www.conrad.de/de/p/keithley-dmm6500-tisch-multimeter-digital-1674576.html

[7] Keysight Technologies (Online - Abrufdatum 23.09.2021)

https://www.keysight.com/de/de/support/33622A/waveform-generator-120-mhz-2-channel.html

[8] Keysight Technologies (Online – Abrufdatum 23.09.2021)

https://rfmw.em.keysight.com/spdhelpfiles/33500/webhelp-mobile/DE/Advanced/Content/__I_SCPI/00%20scpi_introduction.htm

[9] Qt Software (Online – Abrufdatum 24.09.2021)

https://www.qt.io/download

[10] Future Technology Devices International, pdf-Datei – Abrufdatum 25.09.2021

https://ftdichip.com/wp-content/uploads/2021/02/DS_FT2232H-56Q_Mini_Module.pdf

[11] Future Technology Devices International, pdf-Datei – Abrufdatum 25.09.2021

https://www.ftdichip.com/Support/Documents/ProgramGuides/D2XX_Programmer%27s_Guide(FT_000071).pdf

[12] Future Technology Devices International, pdf-Datei – Abrufdatum 25.09.2021

https://www.ftdichip.com/Support/Documents/AppNotes/AN_135_MPSSE_Basics.pdf

[13] Future Technology Devices International, pdf-Datei – Abrufdatum 25.09.2021

https://ftdichip.com/wp-content/uploads/2020/07/DS_FT2232H.pdf

[14] Qwt User´s Guide (Online - Abrufdatum 24.09.2021)

https://qwt.sourceforge.io/

Danksagung

Ich danke Herrn Prof. Dr. -Ing. Michael Athanassios Karagounis und Herrn Dr. Wolfram Budde, die mir die Möglichkeit gegeben haben, dieses besondere Projekt zu realisieren. Außerdem möchte ich mich bei Herrn Nurullah Özkan bedanken, der mich besonders in meiner Anfangszeit des Projektes im Labor an der Fachhochschule Dortmund unterstützt hat. Ich bedanke mich bei Herrn Felix Schneider, der mir während des Projektes mit seinem Fachwissen zur Seite stand und immer ein offenes Ohr für mich hatte.

Einen abschließenden Dank möchte ich meiner Familie und meiner Frau aussprechen, die mich bei meinem Studium immer unterstützt und motiviert haben.

Anhang

- Projektcode

TDC.pro

```
QT       += core gui

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

CONFIG += c++11

# You can make your code fail to compile if it uses deprecated APIs.
# In order to do so, uncomment the following line.
#DEFINES += QT_DISABLE_DEPRECATED_BEFORE=0x060000    # disables all the
APIs deprecated before Qt 6.0.0

SOURCES += \
    Ftdi.cpp \
    Keysight33600A.cpp \
    TDCMeasurement.cpp \
    keithleydmm6500.cpp \
    main.cpp \
    mainwindow.cpp \
    measurementresult.cpp \
    tdcplot.cpp \
    usbtmcdevice.cpp

HEADERS += \
    Ftdi.h \
    Keysight33600A.h \
    TDCMeasurement.h \
    WinTypes.h \
    ftd2xx.h \
    keithleydmm6500.h \
    mainwindow.h \
    measurementresult.h \
    tdcplot.h \
    usbtmcdevice.h

FORMS += \
    mainwindow.ui


OTHER_FILES += /TDC/build/libftd2xx.a\
               /TDC/build/libftd2xx.so.1.4.22

INCLUDEPATH += /home/kara_local/qwt-6.1.6/src/
LIBS += -L/home/kara_local/ftdi_driver/release/build -
L/home/kara_local/qwt-6.1.6/lib -lftd2xx -lqwt


# Default rules for deployment.
qnx: target.path = /tmp/$${TARGET}/bin
else: unix:!android: target.path = /opt/$${TARGET}/bin
!isEmpty(target.path): INSTALLS += target
```

Mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include "TDCMeasurement.h"
#include "Keysight33600A.h"
#include "keithleydmm6500.h"

#include <qwt_plot_histogram.h>
#include <qwt_plot_curve.h>

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

private slots:
    void on_singleMeasurement_clicked();

    void on_delayConst_clicked();

    void on_sweepDelayPhaseSweep_clicked();

    void on_outputOnClockCH1_clicked();

    void on_outputOffClockCH1_clicked();


    void on_outputOnStartStopCH1_clicked();

    void on_outputOffStartStopCH1_clicked();


    void on_refFreqSweep_clicked();

    void on_sweepDelaylength_clicked();

    void on_sweepDelayPhaseConst_clicked();

    void on_pushButton_2_clicked();

    void on_checkBox_stateChanged(int arg1);

    void on_checkBox_2_stateChanged(int arg1);



    void on_pushButton_clicked();

    void on_pushButton_3_clicked();

private:
```

```
    Ui::MainWindow *ui;
    TDCMeasurement tdc;
};

#endif // MAINWINDOW_H
```

Ftdi.h

```
#ifndef FTDI_H
#define FTDI_H
#include "ftd2xx.h"

class Ftdi
{
public:
    Ftdi();
    ~Ftdi();

    enum GpioByte {
        LOW,
        HIGH
    };

    int open(); // open the ftdi
    int readGpioByte(GpioByte byte); // read the pins successively
    int readGpioBytes(); // read the pins together
protected:
    void clearRcvQueue();

    bool opened;

    FT_HANDLE ftHandle;

};

#endif // FTDI_H
```

Keithleydmm6500.h

```
#ifndef KEITHLEYDMM6500_H
#define KEITHLEYDMM6500_H

#include "usbtmcdevice.h"

class KeithleyDMM6500 : public USBTMCDevice
{
public:
    KeithleyDMM6500();

    double getVoltage();
};

#endif // KEITHLEYDMM6500_H
```

Keysight33600A.h

```
#ifndef KEYSIGHT33600A_H
#define KEYSIGHT33600A_H

#include <usbtmcdevice.h>
#include <string>

const int MAXDEVICE_USB = 16;

class Keysight33600A : public USBTMCDevice
{
public:

    Keysight33600A();

    enum Channel {
        CH1,
        CH2
    };

    void setWaveformSquare(Channel ch, double freq, double vpp, double
offset, double phase, double dcycle, bool sync); // set a square signal
for the measurement
    void setFrequency(Channel ch, double freq); // set the frequency
    void setPhase(Channel ch, double phaseDeg, bool sync); // set the
phase

    void enableBurst(); // activate the burst modus
    void disableBurst(); // disable the burst modus
    void setBusTrigger(Channel ch, unsigned delayNs); // sets the
required settings
    void trigger(); // sent a trigger to activate the signals

    void setOutput(Channel ch, bool on);
protected:

};

#endif // KEYSIGHT33600A_H
```

Measurementresult.h

```
#ifndef MEASUREMENTRESULT_H
#define MEASUREMENTRESULT_H


class MeasurementResult
{
public:
    MeasurementResult();

    double aOutput,phaseNs,delayLengthNs;
    unsigned int dOutput,refFreq,samples;



};

#endif // MEASUREMENTRESULT_H
```

TDCMeasurement.h

```
#ifndef TDCMEASUREMENT_H
#define TDCMEASUREMENT_H
#include "Keysight33600A.h"
#include "Ftdi.h"
#include "keithleydmm6500.h"
#include "measurementresult.h"
#include "QString"
class TDCMeasurement
{
public:
    enum Channel {
        CH1,
        CH2
    };
    TDCMeasurement();
    Keysight33600A *clock, *pulse;

    int init();
    int digitalOutput(bool queueRead = true, bool timeout = false);

    int meanDigitalOutput();


    void SetupClockCH1(double freq, double vpp, double offset, double
phase, double dcycle);
    void SetupClockCH2(double freq, double vpp, double offset, double
phase, double dcycle);
    void SetupStartStopCH1(double freq, double vpp, double offset, double
phase, double dcycle);
    void SetupStartStopCH2(double freq, double vpp, double offset, double
phase, double dcycle);
    void SetupClockCH1Off();
    void SetupClockCH2Off();
    void SetupStartStopCH1Off();
    void SetupStartStopCH2Off();

    void generatorSetupClock(double freq, double vpp, double offset,
double phase, double dcycle);
    void generatorSetupStartStop(double freq, double vpp, double offset,
double phase, double dcycle);

    void setMeasureAnalog(bool enable);
    void setMeasureBurst(bool enable);

    bool isMeasureBurst();


    void writeCSV(std::string filename, std::vector<MeasurementResult>
&data);

    MeasurementResult singleMeasurement(double delayLength,double
phaseNs, int refFreq);

    //1 Startstop-Differenz gesweept, Phase konstant, Referenztakt
konstant
    std::vector<MeasurementResult> sweepDelayLength(double
delayLengthStart, double delayLengthEnd, double delayLengthStep, double
phaseNs, int refFreq, unsigned int nsample);
```

```
    //2 Startstop-Differenz konstant, Phase zufällig (nicht
synchronisiert), Referenztank konstant (Histogramm)
    std::vector<MeasurementResult> delayConst(double delayLength,
unsigned int nsample,double phaseNs, int refFreq);

    //3 Startstop-Differenz gesweept, Phase zufällig (nicht
synchronisiert), Referenztakt konstant
    std::vector<MeasurementResult> sweepDelayPhaseConst(double
delayLengthStart, double delayLengthEnd, double delayLengthStep, double
phaseNs, int refFreq, unsigned int nsample);

    //4  Startstop-Differenz gesweept, Phase gesweept, Referenztakt
konstant (Plot für Variation über der Position im TDC)-
    std::vector<MeasurementResult> sweepDelayPhaseSweep(double
delayLengthStart, double delayLengthEnd, double delayLengthStep, double
phaseNsStart,double phaseNsEnd, double phaseNsStep, int refFreq, unsigned
int nsample);

    //5 Startstop-Differenz konstant, Phase gesweept, Referenztakt
gesweept (Einfluss der Frequenz auf Variation)
    std::vector<MeasurementResult> refFreqSweep(double delayLength,
double refFreqStart, double refFreqEnd, double refFreqStep, double
phaseNsStart,double phaseNsEnd, double phaseNsStep, unsigned int
nsample);

    std::vector<MeasurementResult> lastMeasurement;



    double startStopFreq;

    Keysight33600A wvStartStop, wvClock;
    Ftdi ftdi;


protected:

    KeithleyDMM6500 dmmUControl, dmmAnalogOut;
    bool measureAnalog;
    bool measureBurst;


};
```

#endif // TDCMEASUREMENT_H

Tdcplot.h

```
#ifndef TDCPLOT_H
#define TDCPLOT_H
#include "measurementresult.h"
#include "vector"
#include "qwt.h"
#include "mainwindow.h"
#include "ui_mainwindow.h"
class TDCPlot
{
public:
    TDCPlot();
    static void curveSweepDelay(std::vector<MeasurementResult> data,
QwtPlot *plot);
```

```
    static void curveDelayConst(std::vector<MeasurementResult> data,
QwtPlot *plot);
    static void curveSweepDelayPhaseConst(std::vector<MeasurementResult>
data, QwtPlot *plot);
    static void curveSweepDelayPhaseSweep(std::vector<MeasurementResult>
data, QwtPlot *plot);
    static void curveRefFreqSweep(std::vector<MeasurementResult> data,
QwtPlot *plot);
    // these functions visualize the measurements in diagrams
protected:
    static QColor randomColor(); // generates a random color for each
curve

private:
    static bool cmpDouble(double d1, double d2); // compares the double
results
};

#endif // TDCPLOT_H
```

usbtmcdevice.h

```
#ifndef USBTMCDEVICE_H
#define USBTMCDEVICE_H

#include <string>
#include <fstream>
#include <vector>

class USBTMCDevice
{
public:
    USBTMCDevice();

    enum USBTMCStatus {
        OK = 0,
        NOT_FOUND,
        IO_ERROR,
        NOT_CONNECTED
    };

    USBTMCStatus connect(const std::string &serNr); // connect the
devices ( Keysight and Keithley)
    USBTMCStatus sendCommand(const std::string &cmd); //send one command
    USBTMCStatus sendCommands(const std::vector<std::string> &cmds); //
send more then one command

    USBTMCStatus sendRaw(const std::string &data); // ??

    USBTMCStatus receiveLine(std::string &line); // ??

    bool isConnected();
protected:
    bool connected;
    std::string serNr;
    std::fstream devFile;

    static const unsigned MaxDevice = 16;
};

#endif // USBTMCDEVICE_H
```

Main.cpp

```
#include "mainwindow.h"

#include "usbtmcdevice.h"

#include <QApplication>


int main(int argc, char *argv[])
{
   /*std::setlocale(LC_ALL, "en_US.UTF-8");
    std::setlocale(LC_NUMERIC, "en_US.UTF-8");
    std::locale::global(std::locale("en_US.UTF-8"));*/
    srand(time(NULL));
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    return a.exec();

    return 0;
}
```

Maindwindow.cpp

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <unistd.h>
#include <iostream>
#include <qfile.h>
#include "QString"
#include "vector"
#include "measurementresult.h"
#include "TDCMeasurement.h"
#include "Keysight33600A.h"
#include "keithleydmm6500.h"
#include "sstream"
#include "algorithm"
#include "tdcplot.h"
#include "QFileDialog"
#include "qfile.h"
#include "QMessageBox"
#include "QTime"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);



    tdc.init();
}

MainWindow::~MainWindow()
```

```
{
    delete ui;
}
//duty cycle rausnehmen?
void MainWindow::on_singleMeasurement_clicked()
{
    tdc.startStopFreq = ui->editFreqStartStop->text().toDouble();
    tdc.generatorSetupClock(ui->editFreqClock->text().toDouble(), ui-
>editAmplClock->text().toDouble(), ui->editOffsetClock-
>text().toDouble(), 0, 50); // duty cycle ?
    tdc.generatorSetupStartStop(ui->editFreqStartStop->text().toDouble(),
ui->editAmplStartStop->text().toDouble(), ui->editOffsetStartStop-
>text().toDouble(), 0, 50);
    usleep(50000);

    MeasurementResult single;


    single = tdc.singleMeasurement(ui->editStartStopLengthSingle-
>text().toDouble(), ui->editPhaseSingle->text().toDouble(), ui-
>editReffreqSingle->text().toDouble());

    ui->textEdit->setText(QString::number(single.dOutput));
    ui->textEdit_2->setText(QString::number(single.aOutput));
}

void MainWindow::on_sweepDelaylength_clicked()
{
    tdc.startStopFreq = ui->editFreqStartStop->text().toDouble();
    double timeStep = ui->editStepSweepDelay->text().toDouble();

    double smallestTimeStep;

    if (tdc.isMeasureBurst())
    {
        smallestTimeStep = 1.0;
    }
    else
    {
        smallestTimeStep = 1000000000.0 / tdc.startStopFreq * 0.001 /
360;
    }

    double roundedTimeStep = smallestTimeStep * round(timeStep /
smallestTimeStep);

    if (roundedTimeStep == 0)
    {
        roundedTimeStep = smallestTimeStep;
    }

    ui->editStepSweepDelay->setText(QString::number(roundedTimeStep));

    tdc.generatorSetupClock(ui->editFreqClock->text().toDouble(), ui-
>editAmplClock->text().toDouble(), ui->editOffsetClock-
>text().toDouble(), 0, 50);
    tdc.generatorSetupStartStop(ui->editFreqStartStop->text().toDouble(),
ui->editAmplStartStop->text().toDouble(), ui->editOffsetStartStop-
>text().toDouble(), 0, 50);
    usleep(50000);
    std::vector<MeasurementResult> v = tdc.sweepDelayLength(ui-
>editStartSweepDelay->text().toDouble(), ui->editEndSweepDelay-
```

```
>text().toDouble(), ui->editStepSweepDelay->text().toDouble(), ui-
>editPhaseSweepDelay->text().toDouble(), ui->editReffreqSweepDelay-
>text().toDouble(), ui->editSamplesSweepDelay->text().toDouble());


    TDCPlot::curveSweepDelay(v, ui->qwtPlot);
}

void MainWindow::on_delayConst_clicked()
{
    tdc.startStopFreq = ui->editFreqStartStop->text().toDouble();
    tdc.generatorSetupClock(ui->editFreqClock->text().toDouble(), ui-
>editAmplClock->text().toDouble(), ui->editOffsetClock-
>text().toDouble(), 0, 50);
    tdc.generatorSetupStartStop(ui->editFreqStartStop->text().toDouble(),
ui->editAmplStartStop->text().toDouble(), ui->editOffsetStartStop-
>text().toDouble(), 0, 50);
    usleep(50000);
    std::vector<MeasurementResult> v = tdc.delayConst(ui-
>editDelaylengthDelayConst->text().toDouble(), ui->editSamplesDelayConst-
>text().toInt(), ui->editPhaseDelayConst->text().toDouble(), ui-
>editReffreqDelayConst->text().toDouble());

    TDCPlot::curveDelayConst(v, ui->qwtPlot);
}

void MainWindow::on_sweepDelayPhaseConst_clicked()
{
    QMessageBox::StandardButton reply;
    reply = QMessageBox::question(this, "unsynchronized
measurements","Please disconnect the cable that provides synchronization
of the Waveform generators.Have you disconnected the cable?");
    if (reply == QMessageBox::Yes)
    {

    tdc.startStopFreq = ui->editFreqStartStop->text().toDouble();
    tdc.generatorSetupClock(ui->editFreqClock->text().toDouble(), ui-
>editAmplClock->text().toDouble(), ui->editOffsetClock-
>text().toDouble(), 0, 50);
    tdc.generatorSetupStartStop(ui->editFreqStartStop->text().toDouble(),
ui->editAmplStartStop->text().toDouble(), ui->editOffsetStartStop-
>text().toDouble(), 0, 50);
    usleep(50000);

    std::vector<MeasurementResult> v = tdc.sweepDelayPhaseConst(ui-
>editStartUnsych->text().toDouble(), ui->editEndUnsych-
>text().toDouble(), ui->editStepUnsych->text().toDouble(), ui-
>editPhaseUnsych->text().toDouble(), ui->editReffreqUnsych-
>text().toDouble(), ui->editSamplesUnsych->text().toDouble());

    TDCPlot::curveSweepDelayPhaseConst(v, ui->qwtPlot);
    }
    else
        QApplication::quit();
}

void MainWindow::on_sweepDelayPhaseSweep_clicked()
{
    tdc.startStopFreq = ui->editFreqStartStop->text().toDouble();
    double timeStep = ui->editStepphaseDelayPhaseSweep-
>text().toDouble();
```

```
    double smallestTimeStep;

    if (tdc.isMeasureBurst())
    {
        smallestTimeStep = 1.0;
    }
    else
    {
        smallestTimeStep = 1000000000.0 / tdc.startStopFreq * 0.001 /
360;
    }

    double roundedTimeStep = smallestTimeStep * round(timeStep /
smallestTimeStep);

    if (roundedTimeStep == 0)
    {
        roundedTimeStep = smallestTimeStep;
    }

    ui->editStepphaseDelayPhaseSweep-
>setText(QString::number(roundedTimeStep));

    tdc.generatorSetupClock(ui->editFreqClock->text().toDouble(), ui-
>editAmplClock->text().toDouble(), ui->editOffsetClock-
>text().toDouble(),0,50);
    tdc.generatorSetupStartStop(ui->editFreqStartStop->text().toDouble(),
ui->editAmplStartStop->text().toDouble(), ui->editOffsetStartStop-
>text().toDouble(),0,50);
    std::vector<MeasurementResult> v= tdc.sweepDelayPhaseSweep(ui-
>editStartdelayDelayPhaseSweep->text().toDouble(),
                                                               ui-
>editEnddelayDelayPhaseSweep->text().toDouble(),
                                                               ui-
>editStepdelayDelayPhaseSweep->text().toDouble(),
                                                               ui-
>editStartphaseDelayPhaseSweep->text().toDouble(),
                                                               ui-
>editEndphaseDelayPhaseSweep->text().toDouble(),
                                                               ui-
>editStepphaseDelayPhaseSweep->text().toDouble(),
                                                               ui-
>editReffreqDelayPhaseSweep->text().toDouble(),
                                                               ui-
>editSamplesphaseDelayPhaseSweep->text().toDouble());

    TDCPlot::curveSweepDelayPhaseSweep(v,ui->qwtPlot);

}

void MainWindow::on_refFreqSweep_clicked()
{
    tdc.startStopFreq = ui->editFreqStartStop->text().toDouble();
    double timeStep = ui->editStepPhaseReffreqSweep->text().toDouble();

    double smallestTimeStep;

    if (tdc.isMeasureBurst())
    {
        smallestTimeStep = 1.0;
    }
    else
```

```
    {
        smallestTimeStep = 1000000000.0 / tdc.startStopFreq * 0.001 /
360;
    }

    double roundedTimeStep = smallestTimeStep * round(timeStep /
smallestTimeStep);

    if (roundedTimeStep == 0)
    {
        roundedTimeStep = smallestTimeStep;
    }

    ui->editStepPhaseReffreqSweep-
>setText(QString::number(roundedTimeStep));

    tdc.generatorSetupClock(ui->editFreqClock->text().toDouble(), ui-
>editAmplClock->text().toDouble(), ui->editOffsetClock-
>text().toDouble(),0,50);
    tdc.generatorSetupStartStop(ui->editFreqStartStop->text().toDouble(),
ui->editAmplStartStop->text().toDouble(), ui->editOffsetStartStop-
>text().toDouble(),0,50);
    usleep(50000);


    std::vector<MeasurementResult> v=tdc.refFreqSweep(ui-
>editDelaylengthReffreqSweep->text().toDouble(), ui-
>editStartClockReffreqSweep->text().toDouble(), ui-
>editEndClockReffreqSweep->text().toDouble(), ui-
>editStepClockReffreqSweep->text().toDouble(), ui-
>editStartPhaseReffreqSweep->text().toDouble(), ui-
>editEndPhaseReffreqSweep->text().toDouble(), ui-
>editStepPhaseReffreqSweep->text().toDouble(), ui-
>editSamplesReffreqSweep->text().toDouble());

    TDCPlot::curveRefFreqSweep(v,ui->qwtPlot);
}

void MainWindow::on_outputOnClockCH1_clicked()
{

    tdc.SetupClockCH1(ui->editFreqClock->text().toDouble(),ui-
>editAmplClock->text().toDouble(),ui->editOffsetClock-
>text().toDouble(),0,50);

}


void MainWindow::on_outputOffClockCH1_clicked()
{
    tdc.SetupClockCH1Off();
}



void MainWindow::on_outputOnStartStopCH1_clicked()
{
   tdc.SetupStartStopCH1(ui->editFreqStartStop->text().toDouble(),ui-
>editAmplStartStop->text().toDouble(),ui->editOffsetStartStop-
>text().toDouble(),0,50);
```

```
   tdc.SetupStartStopCH2(ui->editFreqStartStop->text().toDouble(),ui-
>editAmplStartStop->text().toDouble(),ui->editOffsetStartStop-
>text().toDouble(),0,50);
}

void MainWindow::on_outputOffStartStopCH1_clicked()
{
   tdc.SetupStartStopCH1Off();
   tdc.SetupStartStopCH2Off();
}



void MainWindow::on_pushButton_2_clicked()
{
    QString file_name = QFileDialog::getSaveFileName(this, "Open a
file");
    std::vector<MeasurementResult> v = tdc.lastMeasurement;

    //if(ui->checkBox->isChecked())
    //{
    //    tdc.writeCSVWOAvol(file_name.toStdString(),v);
    //}
    //else
        tdc.writeCSV(file_name.toStdString(), v);
}

void MainWindow::on_checkBox_stateChanged(int arg1)
{
    tdc.setMeasureAnalog(ui->checkBox->isChecked());
}


void MainWindow::on_checkBox_2_stateChanged(int arg1)
{
    tdc.setMeasureBurst(ui->checkBox_2->isChecked());
}


void MainWindow::on_pushButton_clicked()
{
    ui->qwtPlot->detachItems();
    ui->qwtPlot->replot();
}

void MainWindow::on_pushButton_3_clicked()
{
    double x[2] = {0,25};
    double y[2] = {0,650};

    QwtPlotCurve* curve = new QwtPlotCurve();

    curve->setSamples(x,y,2);
    curve->attach(ui->qwtPlot);
    ui->qwtPlot->replot();


}
```

Ftdi.cpp

```
#include "Ftdi.h"
#include "WinTypes.h"
#include <unistd.h>
#include <iostream>



Ftdi::Ftdi()
    :opened{false} //initialisiert open auf false
{


}

Ftdi::~Ftdi()
{

    if(opened)
    {
        FT_Close(ftHandle);
    }



}

int Ftdi::open()
{
    FT_STATUS ftStatus;
    DWORD dwNumDevs = 0;
    DWORD dwNumBytesToRead = 0;
    DWORD dwNumBytesRead = 0;

    ftStatus = FT_CreateDeviceInfoList(&dwNumDevs);
    // Get the number of FTDI devices
    if (ftStatus != FT_OK)
    {
        std::cout << "Error in getting the number of devices\n";

        return -1;
    }

    if (dwNumDevs < 1)
    {
        std::cout << "There are no FTDI devices installed\n";

        return -2;
```

```
    }


    ftStatus = FT_Open(0, &ftHandle);
    if (ftStatus != FT_OK)
    {
        std::cout << "Open Failed with error " << ftStatus << std::endl;

        return -3;
    }

    ftStatus |= FT_ResetDevice(ftHandle);
    //Reset USB device
    //Purge USB receive buffer first by reading out all old data from
FT2232H receive buffer
    ftStatus |= FT_GetQueueStatus(ftHandle, &dwNumBytesToRead);
    // Get the number of bytes in the FT2232H
    if ((ftStatus != FT_OK) || (dwNumBytesToRead == 0))
    {
        unsigned char byInputBuffer[0x1000]; //groeße 0x1000
        FT_Read(ftHandle, byInputBuffer, dwNumBytesToRead,
&dwNumBytesRead); // Possible overflow

    }


    //Read out the data from FT2232H receive buffer
    ftStatus |= FT_SetUSBParameters(ftHandle, 65536, 65535);
    //Set USB request transfer sizes to 64K
    ftStatus |= FT_SetChars(ftHandle, false, 0, false, 0);
    //Disable event and error characters
    ftStatus |= FT_SetTimeouts(ftHandle, 0, 5000);
    //Sets the read and write timeouts in milliseconds
    ftStatus |= FT_SetLatencyTimer(ftHandle, 1);
    //Set the latency timer to 1mS (default is 16mS)
    ftStatus |= FT_SetFlowControl(ftHandle, FT_FLOW_RTS_CTS, 0x00, 0x00);
    ftStatus |= FT_SetBitMode(ftHandle, 0x0, 0x00);
    //Reset controller
    ftStatus |= FT_SetBitMode(ftHandle, 0x00, 0x02);
    //Enable MPSSE mode

    if (ftStatus != FT_OK)
    {
        std::cout << "Error in initializing the MPSSE " <<  ftStatus <<
std::endl;
        FT_Close(ftHandle);
        return -5;
    }

    opened = true;
    return 1;

}

void Ftdi::clearRcvQueue()
{
    DWORD dwNumBytesToRead, dwBytesRead;

    FT_GetQueueStatus(ftHandle, &dwNumBytesToRead);

    if (dwNumBytesToRead > 0)
    {
```

```
        unsigned char *bigBuffer = new unsigned char[dwNumBytesToRead];
        FT_Read(ftHandle, bigBuffer, dwNumBytesToRead, &dwBytesRead);

        delete[] bigBuffer;
    }
}

int Ftdi::readGpioBytes()
{
    //FT_STATUS ftStatus;
    WORD buffer = 0x8381;

    DWORD dwNumBytesSent = 0;
    DWORD dwNumBytesToRead = 0;
    DWORD dwNumBytesRead = 0;

    if (!opened)
    {
        return -5;
    }

    clearRcvQueue();
    FT_Write(ftHandle, &buffer, 2, &dwNumBytesSent);

    if (dwNumBytesSent != 2)
    {
        std::cout << "Fehler beim Senden" << std::endl;

        return -1;
    }

    for (int i = 0; i < 100; i++)
    {

        FT_GetQueueStatus(ftHandle, &dwNumBytesToRead);

        if (dwNumBytesToRead == 2)
        {

            FT_Read(ftHandle, &buffer, 2, &dwNumBytesRead);

            if (dwNumBytesRead != 2)
            {
                std::cout << "Fehler" << std::endl;
                return -2;
            }

            return buffer;
        }

        usleep(200);
    }

    return -3;
}

int Ftdi::readGpioByte(GpioByte byte)
{
    FT_STATUS ftStatus;
    unsigned char buffer;
    DWORD dwNumBytesSent = 0;
    DWORD dwNumBytesToRead =0;
```

```
    DWORD dwNumBytesRead = 0;

    if (!opened)
    {
        return -5;
    }

    clearRcvQueue();

    buffer = (byte == GpioByte::LOW) ? 0x81 : 0x83; //low = 0x81, high =
0x83;
    ftStatus = FT_Write(ftHandle, &buffer, 1, &dwNumBytesSent);

    if (dwNumBytesSent != 1)
    {
        std::cout << "Fehler beim Senden" << std::endl;

        return -1;
    }

    for (int i = 0; i < 100; i++)
    {

        ftStatus = FT_GetQueueStatus(ftHandle,&dwNumBytesToRead);

        if (dwNumBytesToRead == 1)
        {
            //std::cout<< "Nummer " << dwNumBytesToRead << std::endl;
            ftStatus = FT_Read(ftHandle, &buffer, 1, &dwNumBytesRead);

            if (dwNumBytesRead != 1)
            {
                std::cout << "Fehler" << std::endl;
                return -2;
            }

            return buffer;
        }

        usleep(200);
    }

    return -13;

}
```

Keithleydmm6500.cpp

```
#include "keithleydmm6500.h"

#include <iostream>

KeithleyDMM6500::KeithleyDMM6500()
{

}

double KeithleyDMM6500::getVoltage()
{
    sendCommand(":SENSE:FUNC 'VOLT:DC'");
    sendCommand("READ?");
```

```
    std::string response;

    if (receiveLine(response) != USBTMCStatus::OK)
    {
        std::cout << "Error measuring voltage" << std::endl;
        return 0;
    }

   // std::cout << "Measure voltage response " << response << std::endl;
    size_t decimal_point = response.find(".");

    if (decimal_point != std::string::npos)
    {
        response = response.replace(decimal_point, 1, ",");
    }

    return std::stod(response);
}
```

Keysight33600A.cpp

```
Kxd#include "Keysight33600A.h"
#include <sstream>
#include <iostream>

Keysight33600A::Keysight33600A()
{

}

void Keysight33600A::setWaveformSquare(Channel ch, double freq, double
vpp, double offset, double phase, double dcycle, bool sync)
{
    std::vector<std::string> cmds;
    std::string channel = (ch == Keysight33600A::Channel::CH1) ? "1" :
"2";

    std::stringstream cmd;


    cmd << "SOURce" << channel << ":APPLy:SQUare " << freq << // SCPI
commands
           "," << vpp <<
           "," << offset;

    cmds.push_back(cmd.str());
    cmd.str("");

    cmd << "SOURce" << channel << ":FUNCtion:SQUare:DCYCle " << dcycle;

    cmds.push_back(cmd.str());
    cmd.str("");

    cmd << "SOURce" << channel << ":PHASe " << phase;
    cmds.push_back(cmd.str());

    if (sync)
    {
        cmds.push_back("SOURce" + channel + ":PHASe:SYNChronize");
    }
```

```
    sendCommands(cmds);


}

void Keysight33600A::setPhase(Channel ch, double phaseDeg, bool sync)
{
    std::vector<std::string> cmds;
    std::string channel = (ch == Keysight33600A::Channel::CH1) ? "1" :
"2";
    std::stringstream cmd;



    if (sync)
    {
        cmds.push_back("SOURce" + channel + ":PHASe:SYNChronize");
    }
    cmd << "SOURce" << channel << ":PHASe " << phaseDeg;
    cmds.push_back(cmd.str());
    sendCommands(cmds);
}

void Keysight33600A::setFrequency(Channel ch, double freq)
{
    std::stringstream cmd;
    std::string channel = (ch == Keysight33600A::Channel::CH1) ? "1" :
"2";

    cmd << "SOURce" << channel << ":FREQuency " << freq;

    sendCommand(cmd.str());
}

void Keysight33600A::setBusTrigger(Channel ch, unsigned delayNs)
{
    std::vector<std::string> cmds;
    std::string channel = (ch == Keysight33600A::Channel::CH1) ? "1" :
"2";

    cmds.push_back("TRIGger" + channel + ":SOURce BUS");
    cmds.push_back("TRIGger" + channel + ":DELay " +
std::to_string(delayNs) + "ns");

    sendCommands(cmds);
}

void Keysight33600A::enableBurst()
{
    std::vector<std::string> cmds;

    cmds.push_back("SOURce1:BURSt:STATe ON");
    cmds.push_back("SOURce2:BURSt:STATe ON");

    sendCommands(cmds);
}

void Keysight33600A::disableBurst()
{
    std::vector<std::string> cmds;

    cmds.push_back("SOURce1:BURSt:STATe OFF");
```

```
    cmds.push_back("SOURce2:BURSt:STATe OFF");

    sendCommands(cmds);
}

void Keysight33600A::trigger()
{
    sendCommand("*TRG");
}

void Keysight33600A::setOutput(Channel ch, bool on)
{
    std::string channel = (ch == Keysight33600A::Channel::CH1) ? "1" :
"2";
    std::string onoff = (on) ? "ON" : "OFF";

    sendCommand("OUTPut" + channel + " " + onoff); }
```

TDCMeasurement.cpp

```
#include "TDCMeasurement.h"
#include "Ftdi.h"
#include "ftd2xx.h"
#include "iostream"
#include "Keysight33600A.h"
#include "unistd.h"
#include "measurementresult.h"
#include "iostream"
#include "vector"
#include "sstream"
#include "QFile"
#include "QTime"
#include "QTextStream"
#include "QMessageBox"
#include "QFileDialog"
#include "QCheckBox"
#include "ui_mainwindow.h"
#include "mainwindow.h"
using namespace std;



TDCMeasurement::TDCMeasurement()
{

    measureAnalog = true;
    measureBurst = true;

    if (wvStartStop.connect("MY59001098") != USBTMCDevice::OK)
    {
        std::cout << "Could not connect to Startstop generator" <<
std::endl;
    }

    if (wvClock.connect("MY59001474") != USBTMCDevice::OK)
    {
        std::cout << "Could not connect to clock generator" << std::endl;
    }

    if (dmmUControl.connect("04414106") != USBTMCDevice::OK)
    {
        std::cout << "Could not connect to DMM for control voltage" <<
std::endl;
```

```
    }

    if (dmmAnalogOut.connect("04414122") != USBTMCDevice::OK)
    {
        std::cout << "Could not connect to DMM for analog output voltage"
<< std::endl;
    }
}


int TDCMeasurement::init()
{
    return ftdi.open();
}

void TDCMeasurement::setMeasureAnalog(bool enable)
{
    this->measureAnalog = enable;
}

void TDCMeasurement::setMeasureBurst(bool enable)
{
    this->measureBurst = enable;
}

bool TDCMeasurement::isMeasureBurst()
{
    return this->measureBurst;
}

int TDCMeasurement::digitalOutput(bool queueRead, bool timeout) //
mehrmals aufrufen
{
    for (int i = 0; i < 100;)
    {
        if (timeout)
        {
            ++i;
        }

        if (queueRead)
        {
            int result = ftdi.readGpioBytes();

            if (result < 0)
            {
                return result;

            }

            if ((result & 0x8000) != 0)
            {
                unsigned int measure = (result & 0x0300) >> 8;

                measure |= (result & 0xff) << 2;

                if (measure > 650)
                {
                    printf("GPIO Bytes: %04x\n", (unsigned)result &
0xffff);
                }
```

```
                return measure;
            }
        }
        else
        {
            int result = ftdi.readGpioByte(Ftdi::GpioByte::HIGH); // High
Byte auslesen

            if (result < 0)
            {
                return result;

            }

            if ((result & 0x80) != 0)
            {
                unsigned int measure = result & 0x03;

                result = ftdi.readGpioByte(Ftdi::GpioByte::LOW);

                if (result < 0)
                {
                    return result;

                }

                measure |= result << 2;

                if (measure > 650)
                {
                    printf("GPIO Bytes: %04x\n", (unsigned)result &
0xffff);
                }

                return measure;
            }
        }

        usleep(200);
    }

    return -1;
}

int TDCMeasurement::meanDigitalOutput()
{

    /*double mean = 0;

    for (int j = 0; j < nsample; j++)
    {

        wvStartStop.trigger();
        usleep(1000);
        int result = digitalOutput();

        if (result > 650 || result < 0)
        {
            std::cout << "Fehler... " << result << std::endl;
            return -2;
        }
```

```
        mean+= result;


    }
    mean /= nsample;

    return mean;*/
    for (int i = 0; i < 100; ++i)
        {
            int result = ftdi.readGpioByte(Ftdi::GpioByte::HIGH); // High
Byte auslesen

            if (result < 0)
            {
                return result;

            }

            if ((result & 0x80) != 0)
            {
                unsigned int measure = result & 0x03;

                result = ftdi.readGpioByte(Ftdi::GpioByte::LOW);

                if (result < 0)
                {
                    return result;

                }

                measure |= result << 2;

                return measure;
            }

            usleep(100);
        }

        return -1;
        std::cout<< "Keine Auswertung" << std::endl;


}


std::vector<MeasurementResult> TDCMeasurement::sweepDelayLength(double
delayLengthStart, double delayLengthEnd, double delayLengthStep, double
phaseNs, int refFreq, unsigned int nsample)
{
    std::vector<MeasurementResult> result_vec;

    if (measureBurst)
    {
        wvStartStop.enableBurst();
        wvStartStop.setBusTrigger(Keysight33600A::CH1, phaseNs);
        usleep(10000);
    }
    else
    {
        wvStartStop.disableBurst();
        wvStartStop.setPhase(Keysight33600A::CH1, (-(phaseNs) *
startStopFreq * 360) / 1000000000.0, true);
```

```
            usleep(10000);
        }

        wvClock.setFrequency(Keysight33600A::CH1, refFreq);
        wvClock.setPhase(Keysight33600A::CH1, 0, true);

        for (double sweep = delayLengthStart; sweep <= delayLengthEnd; sweep
+= delayLengthStep)
        {
            if (measureBurst)
            {
                wvStartStop.setBusTrigger(Keysight33600A::CH2, phaseNs +
sweep);
            }
            else
            {
                double roundedPhase = round(-(phaseNs + sweep) *
startStopFreq * 360 / 1000000000.0 * 1000) / 1000.0;

                std::cout << "sweep " << sweep << " Phase auf 3 NKS: " <<
roundedPhase << std::endl;
                wvStartStop.setPhase(Keysight33600A::CH2, roundedPhase,
true);
            }

            usleep(100000);

            for (int i = 0; i < nsample; i++)
            {
                MeasurementResult s;

                if (measureBurst)
                {
                    wvStartStop.trigger();
                    usleep(1000);
                }
                else
                {
                    //std::cout << "Sleep for " << 2 * 1000000.0 /
startStopFreq << " µs" << std::endl;
                    usleep(2 * 1000000.0 / startStopFreq);
                }

                s.dOutput = digitalOutput();
                if (measureAnalog)
                {
                    s.aOutput = dmmAnalogOut.getVoltage();
                }

                s.refFreq = refFreq;
                s.delayLengthNs = sweep;
                s.phaseNs = phaseNs;

                result_vec.push_back(s);
            }
        }

        lastMeasurement = result_vec;
        return lastMeasurement;
}
```

```
std::vector<MeasurementResult>
TDCMeasurement::sweepDelayPhaseConst(double delayLengthStart, double
delayLengthEnd, double delayLengthStep, double phaseNs, int refFreq,
unsigned int nsample)
{
    std::vector<MeasurementResult> result_vec;

    if (measureBurst)
    {
        wvStartStop.enableBurst();
        wvStartStop.setBusTrigger(Keysight33600A::CH1, phaseNs);
    }
    else
    {
        wvStartStop.disableBurst();
        wvStartStop.setPhase(Keysight33600A::CH1, (-(phaseNs) *
startStopFreq * 360) / 1000000000.0, true);
    }

    wvClock.setFrequency(Keysight33600A::CH1, refFreq);
    wvClock.setPhase(Keysight33600A::CH1, 0, true);

    for (double sweep = delayLengthStart; sweep <= delayLengthEnd; sweep
+= delayLengthStep)
    {
        if (measureBurst)
        {
            wvStartStop.setBusTrigger(Keysight33600A::CH2, phaseNs +
sweep);
        }
        else
        {
            double roundedPhase = round(-(phaseNs + sweep) *
startStopFreq * 360 / 1000000000.0 * 1000) / 1000.0;

            wvStartStop.setPhase(Keysight33600A::CH2, roundedPhase,
true);

        }

        usleep(100000);

        for (int i = 0; i < nsample; i++)
        {
            MeasurementResult s;

            if (measureBurst)
            {
                wvStartStop.trigger();
                usleep(1000);
            }
            else
            {
                usleep(2 * 1000000.0 / startStopFreq);
            }

            s.dOutput = digitalOutput();
            if (measureAnalog)
            {
                s.aOutput = dmmAnalogOut.getVoltage();
            }
```

```
            s.refFreq = refFreq;
            s.delayLengthNs = sweep;
            s.phaseNs = phaseNs;

            result_vec.push_back(s);
        }
    }

    lastMeasurement = result_vec;
    return lastMeasurement;
}


std::vector<MeasurementResult>
TDCMeasurement::sweepDelayPhaseSweep(double delayLengthStart, double
delayLengthEnd, double delayLengthStep, double phaseNsStart, double
phaseNsEnd, double phaseNsStep, int refFreq, unsigned int nsample)
{
    std::vector<MeasurementResult> result_vec;

    if (measureBurst)
    {
        wvStartStop.enableBurst();
    }
    else
    {
        wvStartStop.disableBurst();
    }

    wvClock.setFrequency(Keysight33600A::CH1, refFreq);
    wvClock.setPhase(Keysight33600A::CH1, 0, true);

    for (double pulsewidth = delayLengthStart; pulsewidth <=
delayLengthEnd; pulsewidth += delayLengthStep)
    {
        for (double phase = phaseNsStart; phase <= phaseNsEnd; phase +=
phaseNsStep)
        {

            if (measureBurst)
            {
                wvStartStop.setBusTrigger(Keysight33600A::CH1, 0 +
phase);
                wvStartStop.setBusTrigger(Keysight33600A::CH2, pulsewidth
+ phase);
                usleep(50000);
            }
            else
            {
                double roundedPhase = round(-(phase + pulsewidth) *
startStopFreq * 360 / 1000000000.0 * 1000) / 1000.0;
                wvStartStop.setPhase(Keysight33600A::CH1, (-(phase) *
startStopFreq * 360) / 1000000000.0, true);
                wvStartStop.setPhase(Keysight33600A::CH2, roundedPhase,
true);

                 std::cout << "sweep " << phase << " Phase auf 3 NKS: "
<< roundedPhase << std::endl;
                //wvStartStop.setPhase(Keysight33600A::CH2, ((phase-
pulsewidth) * startStopFreq * 360) / 1000000000.0, true);
            }
```

```
            usleep(100000);


            for (int i = 0; i < nsample; i++)
            {
                if (measureBurst)
                {
                    wvStartStop.trigger();
                    usleep(1000);
                }
                else
                {
                    usleep(2 * 1000000.0 / startStopFreq);
                }

                MeasurementResult s;
                s.dOutput = digitalOutput();
                s.delayLengthNs = pulsewidth;
                s.phaseNs = phase;

                if (measureAnalog)
                {
                    s.aOutput = dmmAnalogOut.getVoltage();
                }

                s.refFreq = refFreq;
                result_vec.push_back(s);
            }
        }
    }

    lastMeasurement = result_vec;
    return lastMeasurement;
}

std::vector<MeasurementResult> TDCMeasurement::refFreqSweep(double
delayLength, double refFreqStart, double refFreqEnd, double refFreqStep,
double phaseNsStart,double phaseNsEnd, double phaseNsStep, unsigned int
nsample)
{
    std::vector<MeasurementResult> result_vec;

    if (measureBurst)
    {
        wvStartStop.enableBurst();
    }
    else
    {
        wvStartStop.disableBurst();
    }

    for (double f = refFreqStart; f <= refFreqEnd; f += refFreqStep)
    {
        wvClock.setFrequency(Keysight33600A::CH1, f);

        for (double phase = phaseNsStart; phase <= phaseNsEnd; phase +=
phaseNsStep)
        {

            if (measureBurst)
            {
                wvStartStop.setBusTrigger(Keysight33600A::CH1, phase);
```

```
                wvStartStop.setBusTrigger(Keysight33600A::CH2, phase +
delayLength);
            }
            else
            {
                double roundedPhase = round(-(phase + delayLength) *
startStopFreq * 360 / 1000000000.0 * 1000) / 1000.0;
                wvStartStop.setPhase(Keysight33600A::CH1, (-(phase) *
startStopFreq * 360) / 1000000000.0, true);
                wvStartStop.setPhase(Keysight33600A::CH2, roundedPhase,
true);

                //wvStartStop.setPhase(Keysight33600A::CH2, ((phase -
delayLength) * startStopFreq * 360) / 1000000000.0, true);
            }


            for (int i = 0; i < nsample; i++)
            {
                MeasurementResult s;

                if (measureBurst)
                {
                    wvStartStop.trigger();
                    usleep(1000);
                }
                else
                {
                    usleep(2 * 1000000.0 / startStopFreq);
                }

                s.dOutput = digitalOutput();
                if (measureAnalog)
                {
                    s.aOutput = dmmAnalogOut.getVoltage();
                }


                s.refFreq = f;
                s.delayLengthNs = delayLength;
                s.phaseNs = phase;

                result_vec.push_back(s);
            }
        }
    }

    lastMeasurement = result_vec;
    return lastMeasurement;
}

std::vector<MeasurementResult> TDCMeasurement::delayConst(double
delayLength, unsigned int nsample,double phaseNs, int refFreq)
{

    std::vector<MeasurementResult> result_vec;

    if (measureBurst)
    {
        wvStartStop.enableBurst();
        wvStartStop.setBusTrigger(Keysight33600A::CH1, phaseNs);
```

```
        wvStartStop.setBusTrigger(Keysight33600A::CH2, phaseNs +
delayLength);

    }
    else
    {
        wvStartStop.disableBurst();
        wvStartStop.setPhase(Keysight33600A::CH1, (-(phaseNs) *
startStopFreq * 360) / 1000000000.0, true);
        double roundedPhase = round(-(phaseNs + delayLength) *
startStopFreq * 360 / 1000000000.0 * 1000) / 1000.0;

        wvStartStop.setPhase(Keysight33600A::CH2, roundedPhase, true);


    }


    wvClock.setFrequency(Keysight33600A::CH1, refFreq);
    usleep(100000);


    for (int i = 0; i < nsample; i++)
    {
        if (measureBurst)
        {
            wvStartStop.trigger();
            usleep(10000);
        }
        else
        {
            usleep(2 * 1000000.0 / startStopFreq);
        }

        MeasurementResult s;
        s.dOutput = digitalOutput();
        if (measureAnalog)
        {
            s.aOutput = dmmAnalogOut.getVoltage();
        }

        s.delayLengthNs = delayLength;
        s.phaseNs = phaseNs;
        s.refFreq = refFreq;

        result_vec.push_back(s);
    }

    lastMeasurement = result_vec;
    return lastMeasurement;
}
MeasurementResult TDCMeasurement::singleMeasurement(double
delayLength,double phaseNs, int refFreq)
{
    cout << "Delaylaenge " << delayLength<< "phase " << phaseNs << "Freq
" << refFreq << "Startstopfreq "<< startStopFreq << endl;
    if (measureBurst)
    {
        wvStartStop.enableBurst();
        wvStartStop.setBusTrigger(Keysight33600A::CH1, phaseNs);
        wvStartStop.setBusTrigger(Keysight33600A::CH2, phaseNs +
delayLength);
```

```
    }
    else
    {
        wvStartStop.disableBurst();

        wvStartStop.setPhase(Keysight33600A::CH1, (-(phaseNs) *
startStopFreq * 360) / 1000000000.0, true);
        double roundedPhase = round(-(phaseNs + delayLength) *
startStopFreq * 360 / 1000000000.0 * 1000) / 1000.0;

        wvStartStop.setPhase(Keysight33600A::CH2, roundedPhase, true);

    }


    wvClock.setFrequency(Keysight33600A::CH1, refFreq);
    wvClock.setPhase(Keysight33600A::CH1, 0, true);

    usleep(100000);

    if (measureBurst)
    {
        wvStartStop.trigger();
        usleep(10000);
    }
    else
    {
        usleep(2 * 1000000.0 / startStopFreq);
    }

    MeasurementResult s;
    s.dOutput = digitalOutput();

    if (measureAnalog)
    {
        s.aOutput = dmmAnalogOut.getVoltage();
    }

    s.delayLengthNs = delayLength;
    s.phaseNs = phaseNs;
    s.refFreq = refFreq;

    return s;
}


void TDCMeasurement::SetupClockCH1(double freq, double vpp, double
offset, double phase, double dcycle)
{
    wvClock.setWaveformSquare(Keysight33600A::CH1,
freq,vpp,offset,phase,dcycle,0);


}

void TDCMeasurement::SetupClockCH2(double freq, double vpp, double
offset, double phase, double dcycle)
{
    wvClock.setWaveformSquare(Keysight33600A::CH2,
freq,vpp,offset,phase,dcycle,0);

}
```

```
void TDCMeasurement::SetupStartStopCH1(double freq, double vpp, double
offset, double phase, double dcycle)
{
    wvStartStop.setWaveformSquare(Keysight33600A::CH1,
freq,vpp,offset,phase,dcycle,0);
}

void TDCMeasurement::SetupStartStopCH2(double freq, double vpp, double
offset, double phase, double dcycle)
{
    wvStartStop.setWaveformSquare(Keysight33600A::CH2,
freq,vpp,offset,phase,dcycle,0);
}

void TDCMeasurement::SetupClockCH1Off()
{
    wvClock.setOutput(Keysight33600A::CH1, 0);
}

void TDCMeasurement::SetupClockCH2Off()
{
    wvClock.setOutput(Keysight33600A::CH2, 0);
}

void TDCMeasurement::SetupStartStopCH1Off()
{
    wvStartStop.setOutput(Keysight33600A::CH1, 0);
}

void TDCMeasurement::SetupStartStopCH2Off()
{
    wvStartStop.setOutput(Keysight33600A::CH2, 0);
}



void TDCMeasurement::generatorSetupClock(double freq, double vpp, double
offset, double phase, double dcycle)
{
    SetupClockCH1(freq,vpp,offset,phase,dcycle);

}

void TDCMeasurement::generatorSetupStartStop(double freq, double vpp,
double offset, double phase, double dcycle)
{
    SetupStartStopCH1(freq,vpp,offset,phase,dcycle);
    SetupStartStopCH2(freq,vpp,offset,phase,dcycle);
}

void TDCMeasurement::writeCSV(std::string filename,
std::vector<MeasurementResult> &data)
{

    ofstream myfile;
    myfile.open(filename);


    myfile << "Delaylaenge,";
    myfile << "DOutput,";
    myfile << "phase,";
```

```
    myfile << "aOutput,";
    myfile << "refFreq\n";

    for (auto& value: data)
    {

        myfile << value.delayLengthNs <<",";
        myfile << value.dOutput <<",";
        myfile << value.phaseNs<< ",";
        myfile << value.aOutput << ",";
        myfile << value.refFreq << endl;

    }

    myfile.close();
}
```

tdcplot.cpp

```
#include "tdcplot.h"
#include "measurementresult.h"
#include "vector"
#include "TDCMeasurement.h"
#include "QString"
#include "TDCMeasurement.h"
#include "Keysight33600A.h"
#include "keithleydmm6500.h"
#include "sstream"
#include "algorithm"
#include "iostream"

TDCPlot::TDCPlot()
{

}

bool TDCPlot::cmpDouble(double d1, double d2)
{
    return std::fabs(d1 - d2) <= 0.00001;
}

QColor TDCPlot::randomColor()
{
    int rand_num1 = rand()% 255;
    int rand_num2 = rand()% 255;
    int rand_num3 = rand()% 255;

    return QColor(rand_num1,rand_num2,rand_num3);
}

void TDCPlot::curveSweepDelay(std::vector<MeasurementResult> data,
QwtPlot *plot)
{
    QwtPlotCurve* curve = new QwtPlotCurve();
    std::list<MeasurementResult> l(data.begin(), data.end());

    QVector<double> delaylength, doutput, mostlikely;
```

```
    QVector<double> aoutput;



    while (!l.empty())
    {

        auto it = l.begin();

        float delay = it->delayLengthNs;
        double avg = 0, avg_analog = 0;
        int navg = 0;

        std::map<int, int> minihisto;
        while (it != l.end())
        {

            if (cmpDouble(it->delayLengthNs, delay))
            {
                avg += it->dOutput;
                avg_analog += it->aOutput;

                if (minihisto.count(it->dOutput) == 0)
                {
                    minihisto.insert(std::pair<int, int>(it->dOutput,
1));
                }
                else
                {
                    minihisto[it->dOutput]++;
                }

                navg++;
                l.erase(it++);
            }
            else
            {
                //std::cout << "Ungleich: " << it->delayLengthNs << " <>
" << delay << " = " << it->delayLengthNs - delay << std::endl;
                it++;
            }

        }

        avg /= navg;
        avg_analog /= navg;

        avg_analog = avg_analog / 3.3 * 650;

        delaylength.push_back(delay);
        doutput.push_back(avg);
        aoutput.push_back(avg_analog);

        int maxcount = 0;
        int maxdig = 0;
        for (auto &p : minihisto)
        {
            if (p.second >= maxcount)
                maxdig = p.first;
        }

        mostlikely.push_back(maxdig);
```

```
    }

    //plot->detachItems();
    curve->setSamples(delaylength, doutput);
    curve->setPen(randomColor());
    curve->attach(plot);

    curve = new QwtPlotCurve;
    curve->setSamples(delaylength, mostlikely);
    curve->setPen(Qt::red);
    curve->attach(plot);

    curve = new QwtPlotCurve;
    curve->setSamples(delaylength, aoutput);
    curve->setPen(Qt::red);
    //curve->attach(plot);

    plot->replot();
    plot->setAxisTitle(QwtPlot::xBottom, "delaylength [ns]");
    plot->setAxisTitle(QwtPlot::yLeft, "digital output");
}

void TDCPlot::curveDelayConst(std::vector<MeasurementResult> data,
QwtPlot *plot)
{
    QwtPlotHistogram* histo = new QwtPlotHistogram();

    double bins[651] = {0.0};
    unsigned smallest = 649;
    unsigned biggest = 0;


    for (auto& value: data)
    {

        int result = value.dOutput;
        if (result > 650 || result < 0)
        {
            std::cout << "Fehler, result no. " << " = " << result <<
std::endl;
            return;
        }


        if (static_cast<unsigned>(result) > biggest)
        {
            biggest = result;
        }

        if (static_cast<unsigned>(result) < smallest)
        {
            smallest = result;
        }

        //measures[i] = result;
        bins[result]++;
    }

    QVector<QwtIntervalSample> samples;

    for (uint i = smallest; i <= biggest; i++)
    {
```

```
        QwtInterval interval(i - 0.5, i + 0.5);
        interval.setBorderFlags( QwtInterval::ExcludeMaximum );

        samples.push_back(QwtIntervalSample(bins[i], interval));
    }

    //plot->detachItems();
    histo->setPen(randomColor());
    histo->setSamples(samples);
    histo->attach(plot);
    plot->replot();

    plot->setAxisTitle(QwtPlot::xBottom, "digital output");
    plot->setAxisTitle(QwtPlot::yLeft, "samples");
   //plot->autoRefresh();
}

void TDCPlot::curveSweepDelayPhaseConst(std::vector<MeasurementResult>
data, QwtPlot *plot)
{
    QwtPlotCurve* curve = new QwtPlotCurve();
    std::list<MeasurementResult> l(data.begin(), data.end());

    QVector<double> delaylength,doutput;

    while (!l.empty())
    {

        auto it = l.begin();

        float delay = it->delayLengthNs;
        double avg = 0;
        int navg = 0;

        while (it != l.end())
        {

            //if (it->delayLengthNs == delay)
            if (cmpDouble(it->delayLengthNs, delay))
            {
                avg += it->dOutput;
                navg++;
                l.erase(it++);
            }
            else
            {
                it++;
            }

        }
        avg /= navg;

        delaylength.push_back(delay);
        doutput.push_back(avg);

    }

    //plot->detachItems();
    curve->setSamples(delaylength, doutput);
    curve->setPen(randomColor());
    curve->attach(plot);
```

```
    plot->replot();
    plot->setAxisTitle(QwtPlot::xBottom, "phase [ns]");
    plot->setAxisTitle(QwtPlot::yLeft, "digital output");
}

void TDCPlot::curveSweepDelayPhaseSweep(std::vector<MeasurementResult>
data, QwtPlot *plot)
{
   //plot->detachItems();

    std::list<MeasurementResult> l(data.begin(), data.end());

    while (!l.empty())
    {

        auto it = l.begin();

        float delay = it->delayLengthNs;
        std::list<MeasurementResult> kurve;

        while (it != l.end())
        {


            //if (it->delayLengthNs == delay)
            if (cmpDouble(it->delayLengthNs, delay))
            {
                kurve.push_back(*it);

                l.erase(it++);
            }
            else
            {
                it++;
            }


        }
        QVector<double> phase, dout;

        while (!kurve.empty())
        {

            auto it = kurve.begin();

            float phaseshift = it->phaseNs;
            double avg = 0;
            int navg = 0;

            while (it != kurve.end())
            {

                //if (it->phaseNs == phaseshift)
                if (cmpDouble(it->phaseNs, phaseshift))
                {
                    avg += it->dOutput;
                    navg++;
                    kurve.erase(it++);
                }
                else
                {
                    it++;
```

```
                }

            }
            avg /= navg;

            phase.push_back(phaseshift);
            dout.push_back(avg);

            QwtPlotCurve* curvete = new QwtPlotCurve();

            curvete->setPen(randomColor());

            curvete->setSamples(phase, dout);
            curvete->attach(plot);
        }

    }

    plot->replot();
    plot->setAxisTitle(QwtPlot::xBottom, "phase [ns]");
    plot->setAxisTitle(QwtPlot::yLeft, "digital output");

}

void TDCPlot::curveRefFreqSweep(std::vector<MeasurementResult> data,
QwtPlot *plot)
{
    plot->detachItems();
    std::list<MeasurementResult> l(data.begin(), data.end());

    while (!l.empty())
    {
        auto it = l.begin();

        float freq = it->refFreq;
        std::list<MeasurementResult> kurve;

        while (it != l.end())
        {


            //if (it->refFreq == freq)
            if (cmpDouble(it->refFreq, freq))
            {
                kurve.push_back(*it);

                l.erase(it++);
            }
            else
            {
                it++;
            }


        }


        QVector<double> phase, avol;

        while (!kurve.empty())
        {
```

```
            auto it = kurve.begin();

            float phaseshift = it->phaseNs;
            double avg = 0;
            int navg = 0;

            while (it != kurve.end())
            {

                //if (it->phaseNs == phaseshift)
                if (cmpDouble(it->phaseNs, phaseshift))
                {
                    avg += it->aOutput;
                    navg++;
                    kurve.erase(it++);
                }
                else
                {
                    it++;
                }

            }
            avg /= navg;

            phase.push_back(phaseshift);
            avol.push_back(avg);

        }

        QwtPlotCurve* curvete = new QwtPlotCurve();

        curvete->setPen(randomColor());

        curvete->setSamples(phase, avol);
        curvete->attach(plot);
    }

    plot->replot();
    plot->setAxisTitle(QwtPlot::xBottom, "phase [ns]");
    plot->setAxisTitle(QwtPlot::yLeft, "voltage [V]");
}
```

usbtmcdevice.cpp

```
#include "usbtmcdevice.h"
#include <fstream>
#include <iostream>
#include <locale>

USBTMCDevice::USBTMCDevice()
    : connected(false)
{

}

USBTMCDevice::USBTMCStatus USBTMCDevice::connect(const std::string
&serNr)
{
    for (int i = 0; i < MaxDevice; i++)
    {
        std::string dev = "/dev/usbtmc" + std::to_string(i);
```

```
        std::fstream devFile;

        devFile.imbue(std::locale("en_US.UTF8"));

        devFile.open(dev);

        if (devFile.good())
        {
            devFile << "*IDN?\r\n";
            std::string line;

            std::getline(devFile, line);


            std::cout << "Response " << i << ": " << line << std::endl;

            if (line.find(serNr) != std::string::npos)
            {
                std::cout << "Found correct device\n";
                this->devFile = std::move(devFile);
                this->serNr = serNr;
                connected = true;

                return USBTMCStatus::OK;
            }
        }
        else
        {
            std::cout << "Device " << i << " not found" << std::endl;
        }
    }

    return USBTMCStatus::NOT_FOUND;
}

USBTMCDevice::USBTMCStatus USBTMCDevice::sendCommand(const std::string
&cmd)
{
    return sendRaw(cmd + "\r\n");
}

USBTMCDevice::USBTMCStatus USBTMCDevice::sendCommands(const
std::vector<std::string> &cmds)
{
    for (auto const &cmd : cmds)
    {
        auto status = sendCommand(cmd);
        if (status != USBTMCStatus::OK)
        {
            std::cout << "Fehler" << std::endl;
            return status;

        }
    }

    return USBTMCStatus::OK;
}

USBTMCDevice::USBTMCStatus USBTMCDevice::sendRaw(const std::string &data)
{
    if (!connected)
    {
```

```
        return USBTMCStatus::NOT_CONNECTED;
    }

    devFile << data;
    devFile.flush();

    std::cout << "Sent: " << data;

    return USBTMCStatus::OK;
}

USBTMCDevice::USBTMCStatus USBTMCDevice::receiveLine(std::string &line)
{
    if (!connected)
    {
        return USBTMCStatus::NOT_CONNECTED;
    }

    std::getline(devFile, line);

    return USBTMCStatus::OK;
}
```