

Fachhochschule Dortmund

University of Applied Sciences and Arts

Bachelorarbeit

zur Erlangung des Akademischen Grades

Bachelor of Engineering

Konfiguration und Inbetriebnahme des FTDI 2232H

Mini Moduls als I²C Schnittstelle

Nurullah Özkan

Matrikel Nr.: 7089461

Erstprüfer: Prof. Dr.-Ing. Michael Karagounis

Zweitprüfer: Dipl.-Ing. Rolf Paulus

Abgabedatum: Dortmund, 11.03.2019

Kurzzusammenfassung

In dieser Arbeit wird der Aufbau einer Verbindung zwischen einem Linux Rechner unter Verwendung der Programmierumgebung QT-Creator und der D2XX Bibliothek erläutert. Anschließend wird das Mini Modul als I2C-Schnittstelle konfiguriert, und für die Kommunikation mit dem Speicherbaustein EEPROM 24LC256 über das I2C Protokoll verwendet. Zur Umsetzung dieser Ziele wurde eine GUI zur Steuerung des Mini Moduls programmiert und eine Testplatine für die Platzierung der benötigten Bauteile erstellt.

Die mit der GUI programmierte Applikationssoftware erlaubte zunächst nur die Aktivierung von einzelnen LEDs, die auf der Platine angebracht und mit dem Mini-Modul verbunden waren. Schließlich wurde die GUI und die Applikationssoftware um die Ansteuerung des Mini Moduls als I²C-Schnittstelle erweitert, so dass eine I²C Datenübertragung gestartet und die empfangenen Daten ausgewertet werden konnten. Als letztes wurde der Datentransfer anhand eines Oszilloskops überwacht und analysiert.

Abstract

This thesis explains the connection of a Linux computer to the FTDI FT2232H Mini Module via the programming environment QT-Creator, the D2XX library and the USB port. Based on the I2C protocol, an I2C interface is developed which enables communication with the EEPROM 24LC256 memory module. A GUI has been programmed for the control of the FTDI FT2232H Mini Module. In addition, a circuit board has been designed for test purposes.

The GUI is further developed so that the LEDs, which are mounted on the board, can be controlled. In addition, the GUI can be used to configure the I2C interface, start data transmission and evaluate data. The data transfer is monitored and analyzed using an oscilloscope.

Inhaltsverzeichnis

Tabellenverzeichnis	IV
Abbildungsverzeichnis	V
Abkürzungsverzeichnis	VII
1 Einleitung	1
2 FTDI FT 2232H und D2XX Treiber	2
2.1 FTDI FT 2232H Mini Modul	2
2.1.1 PIN Belegung und Versorgung	4
2.2 D2XX Installation unter Linux	8
3 Qt Creator	12
3.1 Erstellung eines Projektes	12
3.2 D2XX einbinden	14
4 Das I ² C Protokoll	16
4.1 Serielle Schnittstelle	16
4.2 I ² C-Bus	16
4.3 Elektrische Spezifizierung	17
4.4 Datenübertragung	18
4.4.1 Startbedingung	19
4.4.2 Stoppbedingung	20
4.5 Adressierung	21
4.5.1 Read und Write Bit	21
4.5.2 7-Bit Adressierung	21
4.5.3 Lesen und Schreiben bei 7-Bit-Adressierung	22
4.5.4 10-Bit-Adressierung	23
5 EEPROM 24LC256	25
5.1 Funktion	25
5.2 Die Adresseingänge	26

5.3	Serial Data (SDA).....	26
5.4	Serial Clock (SCL)	26
5.5	Write-Protection (WP).....	26
5.6	Die Adressierung	26
5.7	Schreibvorgang.....	27
5.8	Lesevorgang.....	29
6	Konfiguration und Inbetriebnahme	31
6.1	MPSSE.....	31
6.1.1	Pinbelegung bei Nutzung der MPSSE	31
6.1.2	Datenrate (Clock)	31
6.1.3	Konfiguration der Pins	33
6.2	Die Schaltung	35
7	Programmierung des Moduls	37
7.1	Verwendete D2XX Methoden	37
7.1.1	FT_GetDeviceInfoList	37
7.1.2	FT_Open.....	37
7.1.3	FT_Close	37
7.1.4	FT_Write	38
7.1.5	FT_SetBitMode	38
7.1.6	FT_ListDevices	39
7.1.7	FT_ResetDevice	39
7.1.8	FT_GetQueueStatus	39
7.1.9	FT_SetUSBParameters.....	39
7.1.10	FT_SetChars.....	39
7.1.11	FT_SetTimeouts	39
7.1.12	FT_SetLatencyTimer	40
7.2	Test Programm unter Qt-Creator	40
7.3	Startseite der grafischen Benutzeroberfläche	41

7.4	Control Panel für die Steuerung der LEDs	42
7.4.1	Test Button erstellen.....	43
7.4.2	Konstruktor und Destruktor	46
7.4.3	I/O Schalter	46
7.4.4	Pin Steuerung	47
7.4.5	Lauflicht	47
7.4.6	Counter	49
7.4.7	HEX/BIN Konverter	50
7.4.8	Counter / SpinBox	51
7.4.9	Beenden	51
7.4.10	Bild Einfügen	51
7.5	Control Panel für die I ² C-Kommunikation.....	52
7.6	Schreiben	53
7.7	Lesen.....	70
8	Fazit.....	74
9	Danksagung.....	75
10	Literaturverzeichnis	76
11	Anhang.....	79

Tabellenverzeichnis

Tabelle 1: Mini Modul Kanal 2 (CN2) [2]	5
Tabelle 2: Mini Modul Kanal 3 (CN3) [2]	6
Tabelle 3: MPSSE Pinbelegung	31
Tabelle 4: Anfangszustände und Datenrichtung der MPSSE-Schnittstelle.....	34
Tabelle 5: SCL-Taktung	55
Tabelle 6: EEPROM Adresse einstellen.....	56
Tabelle 7: Screenshots (schreiben) bei 100 kHz Taktung	69
Tabelle 8: Screenshots (lesen) bei 100 kHz Taktung	73

Abbildungsverzeichnis

Abbildung 1: FTDI 2232H Mini Modul [1]	2
Abbildung 2: Virtual Com Port [2].....	3
Abbildung 3: Zugriff mittels einer Anwendung zu USB über D2XX [2].....	3
Abbildung 4: FTDI mechanische Details [2]	4
Abbildung 5: Kanal 2 (CN2)	7
Abbildung 6: Kanal 3 (CN3) [2].....	8
Abbildung 7: Aufbau der Verbindung.....	8
Abbildung 8: Download D2XX Treiber [5]	9
Abbildung 9: Release Ordner	10
Abbildung 10: Terminal Ausgabe	11
Abbildung 11: Neues Projekt anlegen	13
Abbildung 12: Neues Projekt anlegen	13
Abbildung 13: Compiler Auswahl.....	14
Abbildung 14: FTDI_Test.pro	15
Abbildung 15: I ² C-System.....	18
Abbildung 16: Startbedingung.....	19
Abbildung 17: Stoppbedingung.....	20
Abbildung 18: Format der 7-Bit-Slave-Adresse.....	22
Abbildung 19: Schreibvorgang mit 7-Bit-Adressierung	23
Abbildung 20: Lesevorgang mit 7-Bit-Adressierung	23
Abbildung 21: Format der 7-Bit-Slave-Adresse.....	24
Abbildung 22: EEPROM 24LC256.....	25
Abbildung 23: Bitmuster für die Adressierung des 24LC256.....	27
Abbildung 24: Byte Write	28
Abbildung 25: Page-Write.....	28
Abbildung 26: Aktuelle Adresse lesen	29
Abbildung 27: Bestimmte Adresse lesen.....	30
Abbildung 28: Sequentieller Lesevorgang	30
Abbildung 29: Zwei-und Dreiphasentaktung [6].....	32
Abbildung 30: Schaltung FTDI - EEPROM.....	35
Abbildung 31: Platine Vor- und Rückderseite	35
Abbildung 32: Ergebnis des Programms	41
Abbildung 33: Fehlermeldung.....	42

Abbildung 34: Startseite	42
Abbildung 35: FTDI Control Panel	43
Abbildung 36: Menü Qt Creator	44
Abbildung 37: Erstellung eines Buttons	45
Abbildung 38: Funktion des Buttons wählen	45
Abbildung 39: I2C-Kommunikation	53
Abbildung 40: Fehlermeldung- Can't open FT2232H	57
Abbildung 41: Startbedingung. SCL (Blau) SDA (Gelb)	62
Abbildung 42: ACK-Bit (Adresse A0)	64
Abbildung 43: NACK-Bit (Adresse A4)	65
Abbildung 44: Fehlermeldung- NACK	65
Abbildung 45: Stoppbedingung	67
Abbildung 46: Daten übertagen	68

Abkürzungsverzeichnis

FTDI	Future Technology Devices International
D2XX	Direct driver interface via FTD2XX
MPSSE	Multi Protocol Synchronous Serial Engine
VCC	Spannungsregler
V3V3	3.3 V DC Betriebsspannung
VIO	Eingangsspannung
USB	Universal Serial Bus
COM	Communication port
Mb/s	Megabyte pro Sekunde
FIFO	First in first out
I2C	Inter-Integrated Circuit
VCP	Virtual COM Port
PC	Personal Computer
DLL	Dynamic Link Library
CN	Channel
GCC	GNU Compiler Collection
GUI	Graphical User Interface
LED	Light emitting diode
I2C	Inter-Integrated Circuit
SATA	Serial Advanced Technology Attachment
SDA	Serial Data
SCL	Serial Clock
ACK	Acknowledge
NACK	Not Acknowledge
MSB	Most Significant Bit
LSB	Least Significant Bit
R/W	Read / Write
WP	Write-Protection
DO	DO, Serial Data / Adress Output
DI	Serial Data Input
SK	Serial Clock
GPIO	General purpose I/O
SPI	Serial Peripheral Interface

JTAG

Joint Test Action Group

1 Einleitung

Im Rahmen dieses Projektes soll der Aufbau einer Verbindung zwischen einem Linux Rechner und dem FT2232H Mini Modul über USB mittels der Programmierumgebung QT-Creator und der *D2XX* Bibliothek erreicht sowie beschrieben werden. Das Ziel ist es, die bereits deklarierten Methoden von der *D2XX* Bibliothek anzupassen und anschließend mit selbst entwickelten Funktionen so zu erweitern, sodass das Modul angesprochen und gesteuert werden kann.

Dieses Projekt wird im Labor für integrierten Schaltungsentwurf durchgeführt und soll zukünftig ermöglichen, selbst entworfene ASICs, welche über eine I²C Schnittstelle verfügen, von einem Linux PC aus zu konfigurieren.

Das *FTDI FT 2232H Mini Module* ist ein USB zu Dual Serial/FIFO (*first in first out*) Entwicklungsmodul. Das Modul verfügt über zwei Kanäle (*CN1 und CN2*), die für synchrone oder asynchrone Geräte konfiguriert werden können. Das Modul ermöglicht es, eine Verbindung zu anderen Geräten über die serielle Schnittstelle aufzubauen, um mit ihnen zu kommunizieren bzw. sie zu steuern. Die integrierte MPSSE Maschine, welche auch für die serielle Kommunikation zuständig ist, kann eine von Taktungsrate bis zu 60 MHz erreichen. [2]

Die serielle Schnittstelle bzw. die MPSSE Maschine wird unter Beachtung der Spezifikation des I²C-Protokolls als I²C-Schnittstelle konfiguriert. Die I²C-Schnittstelle wird zum Nachweis der gewünschten Funktionalität zur Kommunikation mit einem EEPROM 24LC256 Speicherbaustein genutzt. Für diesen Test wurde eine Platine mit allen benötigten Bauelemente entworfen. Abschließend wurde eine GUI mit Hilfe von QT programmiert, Zunächst bestand das Ziel darin über die GUI LEDs ein- und auszuschalten. Anschließend wurde die GUI so weiterentwickelt, dass das Mini Modul als I²C Schnittstelle konfiguriert, sowie eine Datenübertragung gestartet und Daten ausgewertet werden konnten. Zuletzt wurde der Datentransfer anhand eines Oszilloskops überwacht und analysiert.

2 FTDI FT 2232H und D2XX Treiber

2.1 FTDI FT 2232H Mini Modul



Abbildung 1: FTDI 2232H Mini Modul [1]

Das *FTDI 2232H* Mini Modul (Abbildung 1) ist ein elektronisches System, mit dessen Hilfe Daten von einem PC auf verschiedene serielle Busse übertragen werden können. Außerdem verfügt das Modul über einen USB 2.0 Adapter, über den die Daten mit einer Höchstgeschwindigkeit von 480 Mb/s Übertragungsrate zwischen PC und FTDI Baustein gesendet werden können. Zusätzlich besitzt das zwei Modul Kanäle, die entweder als synchrone oder asynchrone serielle Schnittstelle oder aber auch als parallele FIFO Schnittstelle konfiguriert werden können. Die beiden Kanäle können von der sogenannten MPSSE Zustands Maschine so angesteuert werden, dass die Abbildung verschiedener serieller Protokolle wie z.B. SPI, JTAG und im Falle dieser Arbeit I²C möglich wird. [2]

Für den Aufbau der Verbindung zwischen Mini-Modul und PC wird ein USB Treiber benötigt. Der Treiber wird vom Hersteller frei zur Verfügung gestellt. Der Treiber sowie die Datenblätter

sind auf der Internetseite des Herstellers¹ direkt zu finden und können bei Bedarf heruntergeladen und installiert werden. Um das Modul erfolgreich programmieren zu können, muss das Modul zunächst mit einem Rechner verbunden werden. Die Verbindung kann direkt über den Virtual COM Port (VCP) aufgebaut werden (Abbildung 2). Dies ermöglicht dem Benutzer, an einem standarmäßigen PC-Anschluss über die USB Schnittstelle zu kommunizieren. Darüber hinaus gibt es noch die Möglichkeit, die Verbindung über den D2XX Treiber auszubauen. Auf diese Methode wird im späteren Verlauf der Arbeit näher eingegangen. Der Treiber kann aus einer Anwendungssoftware mit Hilfe einer DLL (*Dynamic Link Library*) direkt angesprochen werden (Abbildung 3). Im Rahmen des Projektes wird die Applikationssoftware unter Verwendung der Programmierumgebung QT-Creator entwickelt, die wir in den nächsten Kapiteln detaillierter betrachten. [2]

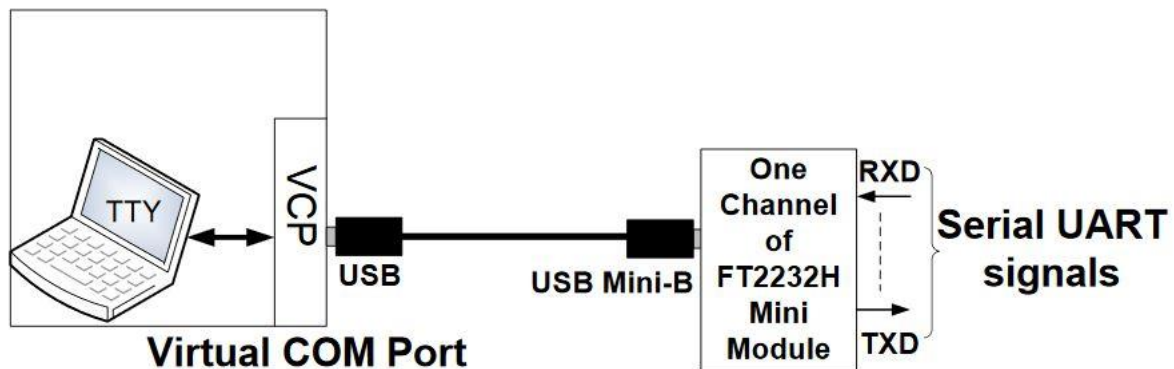


Abbildung 2: Virtual Com Port [2]

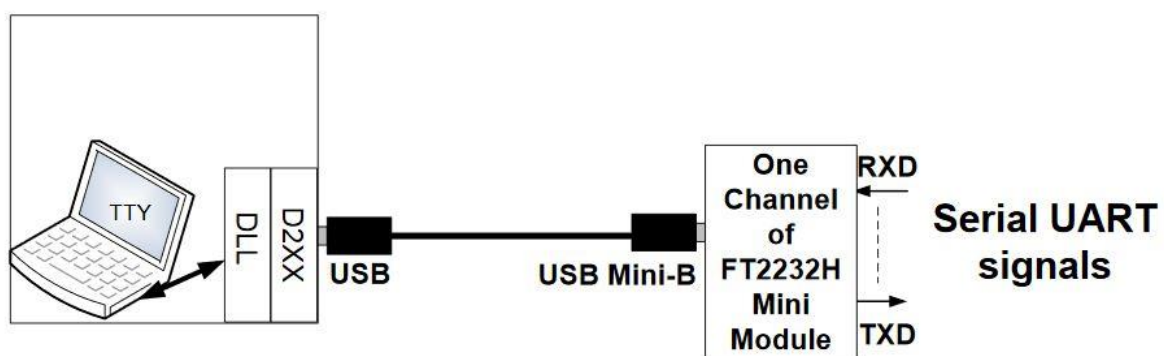


Abbildung 3: Zugriff mittels einer Anwendung zu USB über D2XX [2]

¹ <https://ftdichip.com>

2.1.1 PIN Belegung und Versorgung

Die PIN Belegung bzw. die elektrischen Verbindungen werden in Abbildung 4 dargestellt.

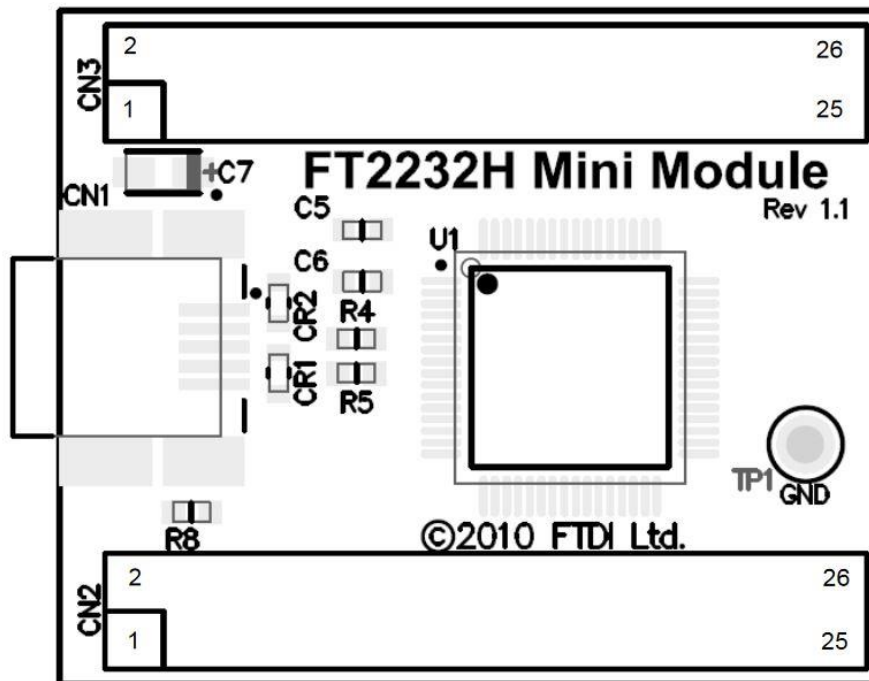


Abbildung 4: FTDI mechanische Details [2]

Das Modul hat zwei Leisten mit jeweils 26 Pins. Die Belegung der einzelnen Pins kann aus den Tabellen 1 und 2 entnommen werden. [2]

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

Tabelle 1: Mini Modul Kanal 2 (CN2) [2]

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

Tabelle 2: Mini Modul Kanal 3 (CN3) [2]

Das Modul bietet zwei Konfigurationsmöglichkeiten. Bei der ersten Option kann das Modul an eine Spannungsquelle angeschlossen und darüber versorgt werden. Bei der zweiten Möglichkeit wird das Modul direkt über den USB Bus versorgt. In diesem Projekt wurde entschieden, dass das Modul über den USB Bus versorgt werden soll. Zunächst müssen einige Pins miteinander verbunden werden:

In Abbildung 5 ist die Beschaltung des Kanals 2 (CN2) und in Abbildung 6 die Beschaltung des Kanals 3 (CN3) dargestellt. In der Tabelle 1 werden die Bedeutungen der einzelnen Pins des Kanals 2 und in der Tabelle 2 die Pins des Kanals 3 erklärt.

Zur Versorgung über USB muss VBUS (Pin 1 CN3) über einen Jumper mit VCC (Pin3 CN3) verbunden werden. Die Spannungsversorgung für die Ein- bzw. Ausgänge (VIO) ist laut Schaltplan nicht angeschlossen und muss daher ebenfalls mit einer Spannung verbunden werden. Dies kann mit einer Verbindung des Spannungsausgangs VCC3V3 (Pin 1, 3 oder 5 CN2) mit VIO (Pin 11 oder 21 CN2) erfolgen (siehe Abbildung 6). [2]

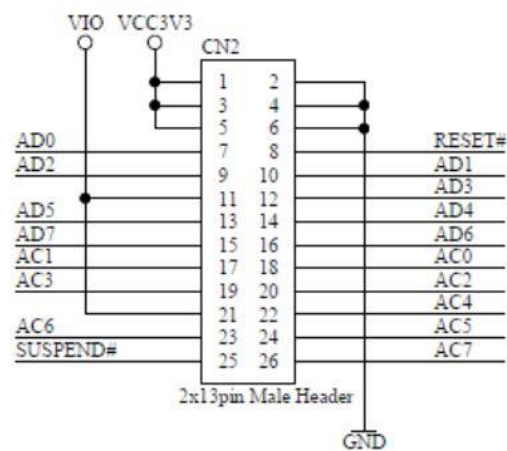


Abbildung 5: Kanal 2 (CN2)

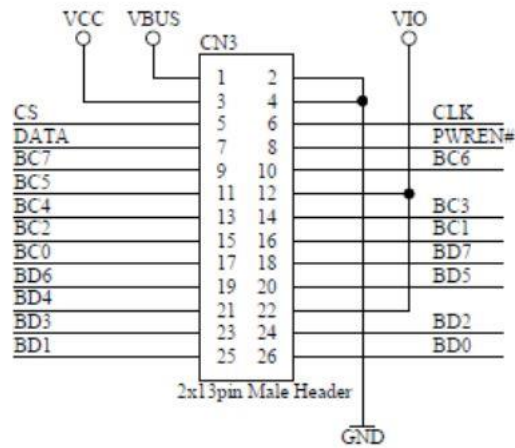


Abbildung 6: Kanal 3 (CN3) [2]

2.2 D2XX Installation unter Linux

Der von FTDI bereitgestellte D2XX Treiber ermöglicht den Zugriff auf das USB Gerät bzw. auf das FTDI 2232H Mini Modul durch die Einbindung der entsprechenden D2XX.dll in der verwendeten Programmierumgebung. Dies vermindert den Aufwand für die Softwareentwicklung, da die für die Verbindung benötigten Methoden bereits von FTDI entwickelt worden sind und einfach in das Programm integriert werden können. Um die Verbindung über den USB Port aufbauen zu können, muss zuerst der Treiber unter Linux erfolgreich installiert werden. In Abbildung 7 wird ein Aufbau der Verbindung dargestellt. Sollte der Treiber nicht wie vorgesehen installiert sein, kann auch keine Verbindung aufgebaut und das Modul nicht angesprochen werden. Daher wird die Installation nachfolgend detaillierter beschrieben. [2][3]



Abbildung 7: Aufbau der Verbindung

Zunächst wird der D2XX Treiber von der Internetseite von FTDI² heruntergeladen (Abbildung 8). Für das Projekt wurde die Version 1.4.6, die am 29.06.2017 veröffentlicht wurde, verwendet. Anschließend wird die Datei *libftd2xx-x64-1.4.6.tgz* gespeichert.


Operating System	Release Date	Processor Architecture					Comments
		x86 (32-bit)	x64 (64-bit)	ARM	MIPS	SH4	
Windows*	2017-08-30	2.12.28	2.12.28	-	-	-	WHQL Certified. Includes VCP and D2XX. Available as a setup executable Please read the Release Notes and Installation Guides.
Windows RT	2014-07-04	1.0.2	-	1.0.2	-	-	A guide to support the driver (AN_271) is available here
Linux	2017-06-29	1.4.6	1.4.6	1.4.6 ARMv5 soft-float 1.4.6 ARMv5 soft-float uClibc 1.4.6 ARMv6 hard-float (suits Raspberry Pi) 1.4.6 ARMv7 hard-float 1.4.6 ARMv8 hard-float	1.4.6 MIPS32 soft-float 1.4.6 MIPS32 hard-float 1.4.6 MIPS openwrt-uclibc		If unsure which ARM version to use, compare the output of <code>readelf</code> and <code>file</code> commands on a system binary with the content of <code>release/build/libftd2xx.txt</code> in each package. ReadMe 

Abbildung 8: Download D2XX Treiber [5]

Nun wird ein Terminal unter Linux geöffnet. Um die Installation erfolgreich zu beenden, müssen folgende Befehle in das Terminal eingegeben werden:

```
cd Download // Downloadordner öffnen
ls // die Datei „libftd2xx-x64-1.4.6.tgz“ müsste nun angezeigt werden
tar -xvf lib* // Treiberdatei wird extrahiert
```

Nun müsste die Datei extrahiert worden sein und die jeweiligen Dateien im Terminal ausgegeben werden. Ein Überblick der Dateien und der extrahierten Ordnerstruktur des Treiberpakets ist in Abbildung 10 dargestellt.

```
sudo cp release/build/lib* /usr/local/ftdi /* kopiert die Dateien in den lokalen Benutzerordner */
cd /usr/local/ftdi //Wechsel in den lokalen FTDI Ordner wird aufgerufen
// Überprüfen, ob die Dateien erfolgreich kopiert worden sind
sudo ln -s libftd2xx.so.1.4.6 libftd2xx.so // Verknüpfung der Treiberbibliothek
```

² <http://www.ftdichip.com/Drivers/D2XX.htm>

sudo chmod libftd2xx.so.1.4.6

// Modus ändern, damit jeder Zugriff darauf hat

An dieser Stelle wird das FTDI-Modul angeschlossen. Das Modul wird von Linux erkannt, es kann angesprochen werden, und es kann eine Seriennummer abgefragt werden. Im weiteren Verlauf der Untersuchungen ist jedoch ein Fehlverhalten aufgetreten, welches auf ein Problem bei der Treiberinstallation hinweist. Wenn in der Programmierumgebung ein Befehl verwendet worden ist, mit dem z.B Ports als Ausgänge gesetzt werden können, wurde eine Fehlermeldung ausgeworfen, und die Verbindung zum Modul brach ab. Der Grund für dieses Fehlverhalten besteht darin, dass Linux die Standardtreiber `ftdi_sio` und `usbserial` lädt sobald das FTDI Modul angeschlossen wird. Diese Treiber sind aber mit FTDI inkompatibel und verhindern den Aufbau der Verbindung. Die Treiber sind einfach zu deinstallieren. Ohne Adminrechte ist der Vorgang jedoch nicht durchzuführen. Folgende Befehle beschreiben die Deinstallation: [4]

Sudo rmmod ftdi_sio

Sudo rmmod usbserial

Nach der Deinstallation der Standardtreiber sollte sich bei der Verbindung des Mini-Moduls die in Abbildung 10 dargestellte Ausgabe einstellen.

Jetzt besteht die Möglichkeit, direkt über das Terminal, die mitgelieferten Testprogramme auszuführen. Das Ziel des Projektes ist es aber, das Modul über Qt-Creator anzusprechen und zu steuern.



```
oezkan_local@IMESBM14:~
File Edit View Search Terminal Help
USB Serial support registered for FTDI USB Serial Device
ftdi_sio 2-1:1.0: FTDI USB Serial Device converter detected
usb 2-1: Detected FT2232H
usb 2-1: Number of endpoints 2
usb 2-1: Endpoint 1 MaxPacketSize 512
usb 2-1: Endpoint 2 MaxPacketSize 512
usb 2-1: Setting MaxPacketSize 512
usb 2-1: FTDI USB Serial Device converter now attached to ttyUSB0
ftdi_sio 2-1:1.1: FTDI USB Serial Device converter detected
usb 2-1: Detected FT2232H
usb 2-1: Number of endpoints 2
usb 2-1: Endpoint 1 MaxPacketSize 512
usb 2-1: Endpoint 2 MaxPacketSize 512
usb 2-1: Setting MaxPacketSize 512
usb 2-1: FTDI USB Serial Device converter now attached to ttyUSB1
usbcore: registered new interface driver ftdi_sio
ftdi_sio: v1.5.0:USB FTDI Serial Converters Driver
usbcore: deregistering interface driver ftdi_sio
ftdi_sio ttyUSB1: FTDI USB Serial Device converter now disconnected from ttyUSB1
ftdi_sio 2-1:1.1: device disconnected
ftdi_sio ttyUSB0: FTDI USB Serial Device converter now disconnected from ttyUSB0
ftdi_sio 2-1:1.0: device disconnected
USB Serial deregistering driver FTDI USB Serial Device
[oezkan_local@IMESBM14 ~]$
```

Abbildung 9: Terminal Ausgabe

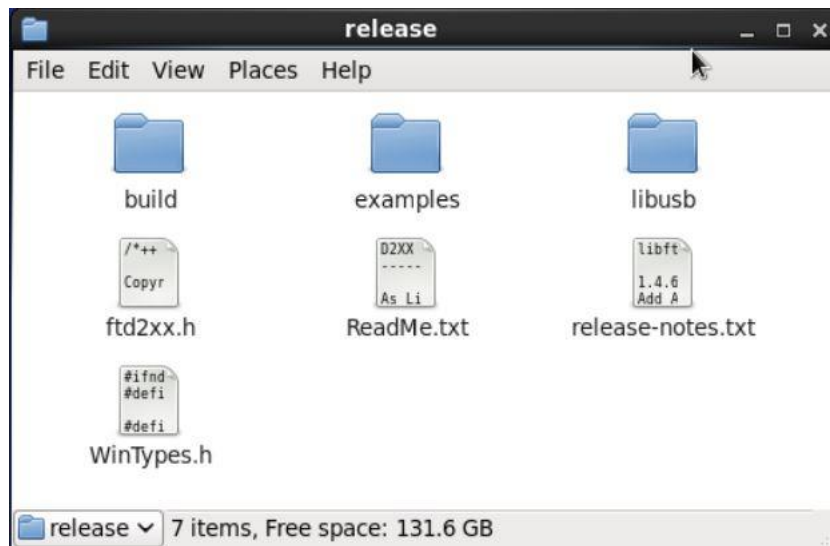


Abbildung 10: Release Ordner

3 Qt Creator

In diesem Projekt wird zur Entwicklung der C++ Programme die Programmierumgebung *Qt-Creator* verwendet. Die Programmierumgebung ist plattformunabhängig. *Qt-Creator* ist ideal für die GUI Programmierung und es erleichtert die Entwicklungsarbeit. Es kann von der Webseite des Herstellers³ heruntergeladen werden. Nach der Installation müssen noch entsprechende Packages, wie der „GCC“ Compiler installiert werden.

3.1 Erstellung eines Projektes

Zunächst wird der Qt Creator gestartet. Nun gibt es die Möglichkeit, ein neues Projekt anzulegen oder ein vorhandenes Projekt zu öffnen. Um ein neues Projekt anzulegen, wird „*New Projekt*“ angeklickt. Jetzt muss ausgewählt werden, was für eine Anwendung programmiert wird. Da zum jetzigen Zeitpunkt nur eine Verbindung zum *FTDI-Modul* aufgebaut werden soll, wird die Konsolen-Anwendung gewählt. Anschließend muss festgelegt werden, wie das Projekt heißen soll. In diesem Fall heißt das Projekt „*FTDI_Test*“. Im Anschluss daran wird der Speicherort festgelegt. Empfehlenswert ist der lokale Ordner. Diese bisherigen Schritte werden graphisch in Abbildung 11 und Abbildung 12 dargestellt.

³ <https://www.qt.io/download>

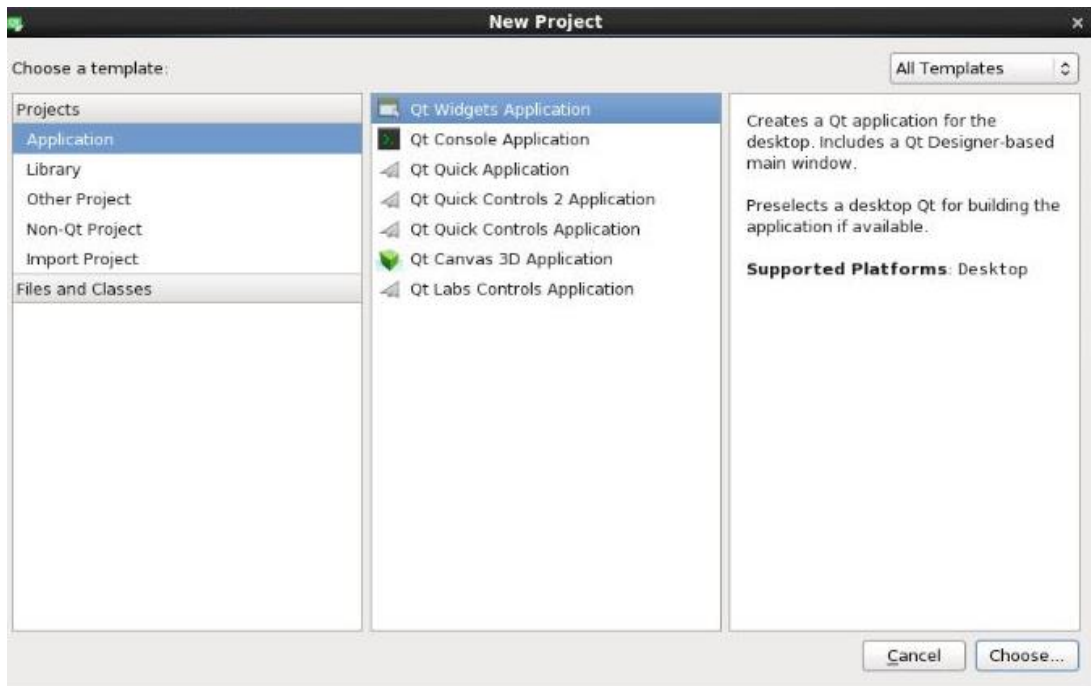


Abbildung 11: Neues Projekt anlegen

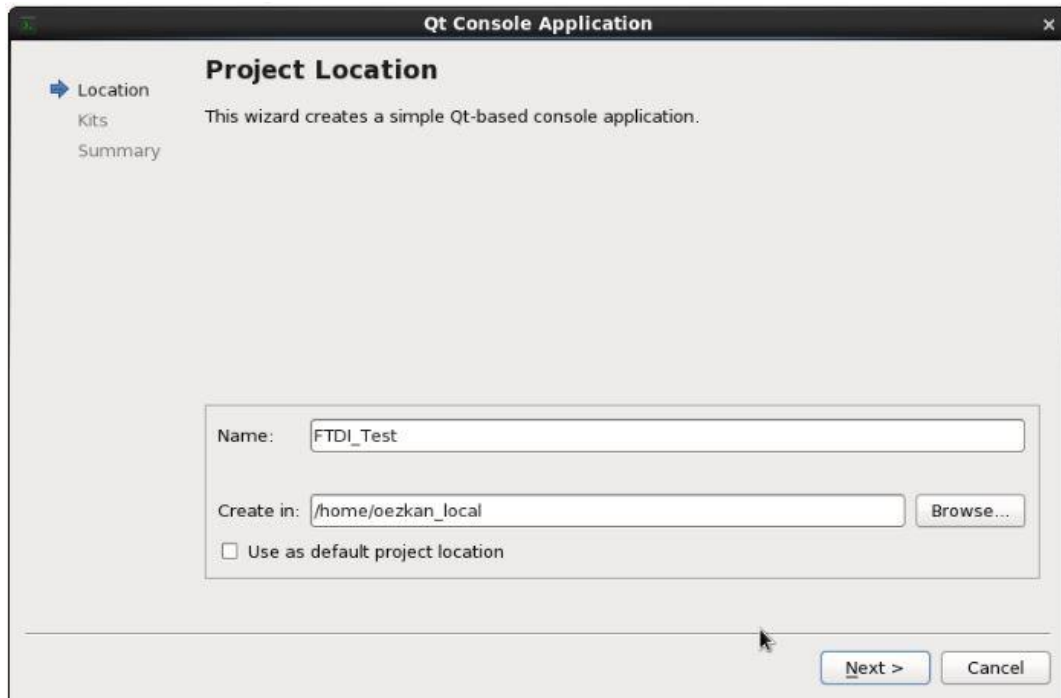


Abbildung 12: Neues Projekt anlegen

Im nächsten Schritt fragt die Programmierumgebung, welchen Compiler das Programm verwenden soll. Dazu muss nichts verändert werden, da der Pfad automatisch auf den „GCC“ Compiler gesetzt wird, wenn der gcc Compiler vor der Erstellung des Projektes bereits installiert wurde. Dies wird in Abbildung 13 dargestellt. Der gewählte Compiler kann selbstverständlich im Laufe des Projektes unter Einstellungen bzw. unter Manage Kits jederzeit geändert werden.



Abbildung 13: Compiler Auswahl

Qt generiert eine Übersicht über das was bisher eingestellt wurde. Abschließend wird der Button „finish“ geklickt, um die Generierung des Projektes zu initiieren.

Inzwischen wurde im lokalen Verzeichnis bereits ein Ordner mit den Namen „*FTDI_Test*“ angelegt. Um die *D2XX* Treiber korrekt nutzen zu können, werden die Inhalte von dem bereits entpackten „Release“ in den Projektordner kopiert. Außerdem müssen in der Programmierumgebung *Qt-Creator* gewisse Einstellungen durchgeführt werden. Welche im nächsten Kapitel erläutert werden. [14]

3.2 D2XX einbinden

Als erstes wird die *FTDI_Test.pro* Datei geöffnet. In dieser Datei kann die Nutzung einer seriellen Schnittstelle definiert werden. Gleichzeitig werden in dieser Datei die Pfade auf die Header Dateien sowie die *D2XX* Bibliothek definiert. Die serielle Schnittstelle mit dem Befehl „*QT += serialport*“ eingebunden. Als nächstes werden die Header Dateien definiert. Hierfür wird hinter „*HEADERS+=/*“ der Pfad, in der sich die Header Dateien befinden, eingefügt. Im vorherigen Kapitel wurde erwähnt, dass die Dateien aus dem Verzeichnis „*Release*“ in den

Projektordner „FTDI_Test“ kopiert wurden. Es gibt zwei Header Dateien, die eingebunden werden müssen, damit das Modul korrekt angesprochen werden kann, nämlich die „ftd2xx.h“ und die „wintypes.h“ Datei. Anschließend wird die Bibliothek eingebunden. Die Dateien befinden sich ebenfalls im „Release“ bzw. im „build“ Ordner. Es handelt sich um die Dateien „libftdxx.a“ und „libftdxx.so.1.4.6“, welche unter „OTHER_FILES+=“ eingegeben werden müssen. In Abbildung 14 wird gezeigt, wie die .pro Datei schließlich auszusehen hat.

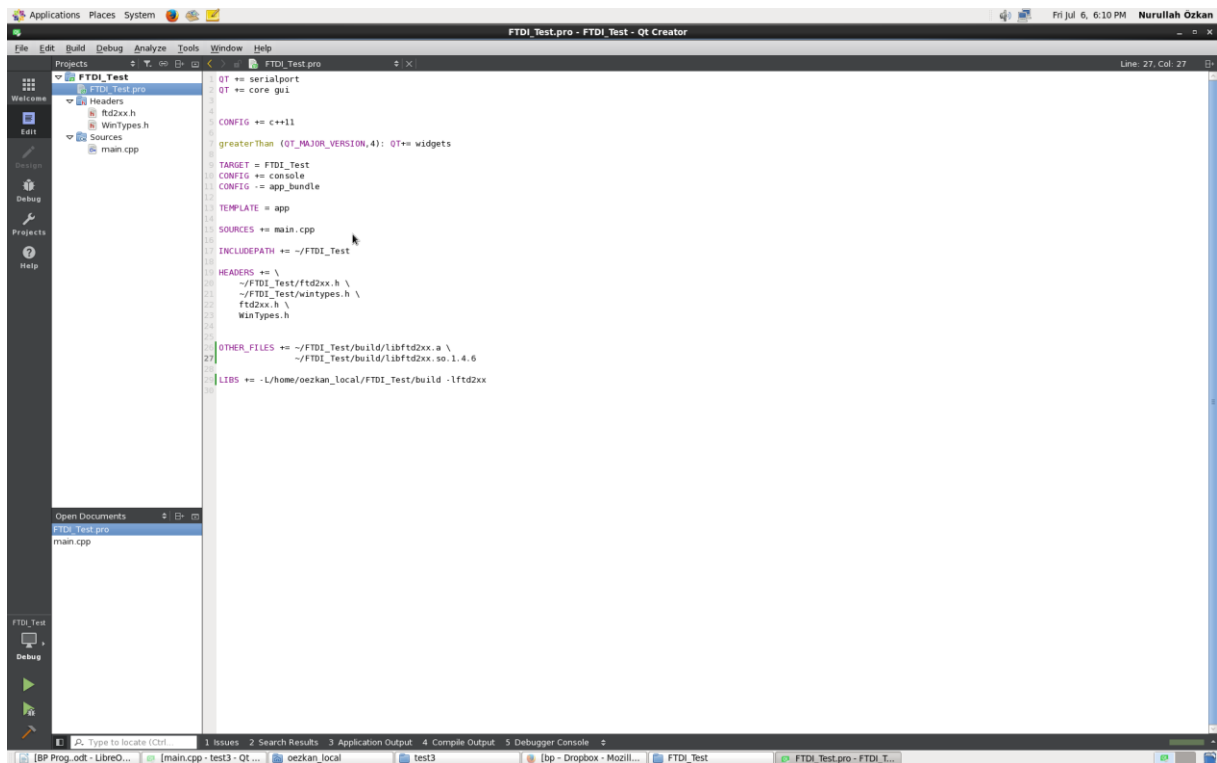


Abbildung 14: FTDI_Test.pro

Mit einem Rechtsklick auf die Datei „Headers“ im linken Fenster, öffnet sich ein Dropdownmenü. Anschließend werden unter „Add existing Files“ die beiden Header Dateien, die sich im selben Ordner wie das Projekt befinden, gewählt. Jetzt sind alle Dateien korrekt eingebunden und es kann mit der Programmierung begonnen werden.

4 Das I²C Protokoll

4.1 Serielle Schnittstelle

Serielle Schnittstellen gehören in der Informationstechnik zu den bevorzugten Verfahren für die Implementierung von Bussystemen. Bei dieser Kommunikationsart werden alle Bits eines Datensatzes seriell bzw. nacheinander über eine einzelne Leitung geschickt. Im Gegensatz zur seriellen Schnittstelle gibt es auch parallele Schnittstellen. Bei dieser Schnittstellenart werden alle Bits eines Datenwortes gleichzeitig über eine entsprechende Anzahl an Leitungen gesendet. Bei sehr hohen Übertragungsraten entstehen jedoch Synchronisationsprobleme zwischen den einzelnen Bits. Außerdem ergibt sich auf Grund der höheren Leitungszahl auch ein höherer Aufwand und ein größerer Platzbedarf auf einer Platine oder einem ASIC. Aus diesen Gründen werden in der Informationstechnik hauptsächlich serielle Schnittstellen eingesetzt und der Aufwand und die Kosten für eine parallele Kommunikation vermieden. Mit der seriellen Kommunikation können ebenfalls hohe Übertragungsraten erreicht werden. Beispielsweise erfolgt die Übertragung innerhalb eines Rechners für den Festplattenanschluss SATA⁴ oder die Verbindung zu den externen Geräten über die serielle Schnittstelle USB. [11][12]

4.2 I²C-Bus

Der I²C-Bus gehört auch zu der seriellen-Bus Familie. I²C = IIC ist die Abkürzung für Inter-Integrated Circuit. Er wurde, für die Kommunikation von Mikrocontrollern und ICs auf einer Platine von Philips im Jahr 1982 entwickelt und wurde für die Kommunikation der einzelnen integrierten Bauteile in einem Fernseher genutzt. I²C wird daher hauptsächlich für die geräteinterne Kommunikation verwendet und ist nicht für die Kommunikation über lange Leitungen gedacht. Bisher wurden 4 Versionen veröffentlicht. In der ersten Version konnten Übertragungsraten von 100 kBit in Fast-Mode sogar 400 kBit erreicht werden. In der letzten Version kann unidirektional eine Übertragungsrate bis zu 5 MBit erzielt werden.

Da der Bus ursprünglich von Philips entwickelt bzw. lizenziert wurde, wird er von anderen Firmen als TWI (Two-Wire Interface) bezeichnet. Jedoch sind I2C und TWI Bussystem-technisch betrachtet identisch.

⁴ Serial Advanced Technology Attachment

Der I²C-Bus ist ein Master-Slave-Bus und wird mit zwei Leitungen SDA (Serial Data) und SCL (Serial Clock) realisiert. Diese zwei Leitungen dienen bidirektional dem Informationsaustausch. Alle Teilnehmer dieses Bus-System haben eine eigene Adresse. Die Bausteine auf diesem Bus-System können je nach Zweck, Daten senden oder Daten empfangen. Eine Tastatur oder ein Temperatursensor können beispielsweise meist nur Daten senden während ein LCD-Display nur Daten empfangen kann. Ein Speicherbaustein dagegen kann Daten sowohl senden als auch empfangen. Derartige Peripherie-Bausteine werden generell im Slave-Modus betrieben. Ein Microcontroller agiert hingegen in den meisten Fällen als Master, welcher in beide Richtungen kommuniziert. Grundsätzlich kann jeder Teilnehmer des I²C-Bus-Systems sowohl als Master als auch als Slave operieren. Es macht aber sicherlich keinen Sinn, einen EEPROM-Baustein als Master zu konfigurieren, da diesem Baustein im Gegensatz zu einem Mikrocontroller die lokale Intelligenz fehlt. Da I²C über Multimastereigenschaften verfügt können auf diesem Bus-System mehrere Einheiten als Master operieren. Selbstverständlich kann zu einem Zeitpunkt nur ein Master auf dem Bus die Kommunikation steuern. Der Master generiert die Taktimpulse auf der SCL-Leitung und bestimmt die Datenrichtung. Der Master kann mit Angabe der Adresse jeden Slave auf dem Bus-System ansprechen, Daten senden oder empfangen. [1] [11]

4.3 Elektrische Spezifizierung

Die beiden Datenleitungen SDA und SCL werden mittels Pull-Up-Widerstandes (R_v) auf HIGH-Pegel gehalten. Der Wert dieser Pull-Up-Widerstände ist von der Versorgungsspannung, von der Taktrate und von der Buskapazität abhängig. Bei +5 Volt sollte der Wert des Widerstandes mindestens $2k\Omega$ betragen. Alle Teilnehmer dieses Bus-Systems verfügen über ein Open-Drain oder Open-Collector Schaltung damit die entsprechende Leitung auf Masse heruntergezogen werden kann. Dadurch ergibt sich eine sogenannte Wired-AND Schaltung. In der unteren Abbildung 15 wird die Struktur I²C-System dargestellt. [1]

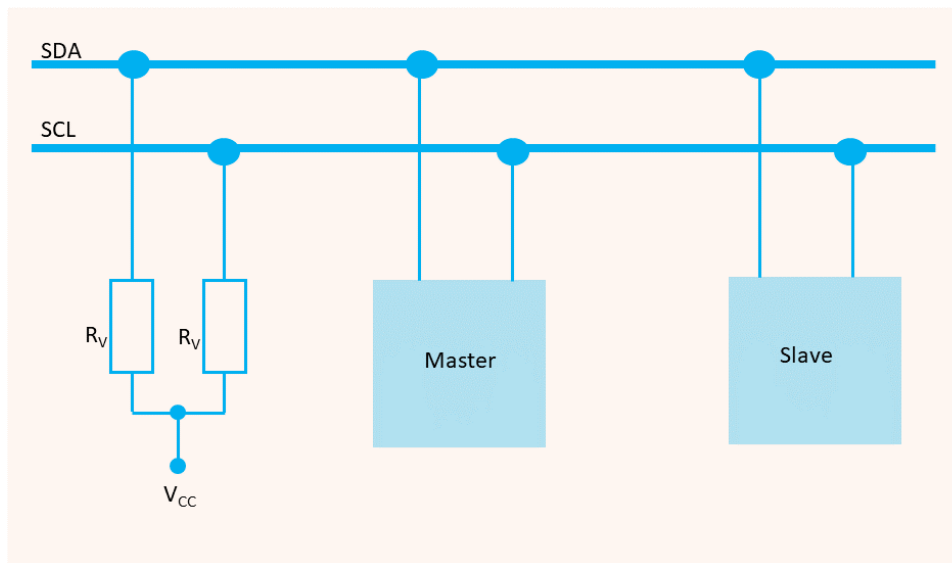


Abbildung 15: I²C-System

4.4 Datenübertragung

Auf einem I²C-Bus können die angeschlossenen Buseinheiten mit verschiedenen Geschwindigkeiten bzw. Taktfrequenzen kommunizieren. Die Kommunikation erfolgt zwar seriell, das Datenprotokoll ist aber Byte-orientiert. Daher erfolgt die Datenübertragung mit der kleinsten Speichereinheit, mit der Peripherie-Geräte Daten untereinander austauschen. I²C entspricht einem Master-Slave Bus. Das bedeutet, dass eine Kommunikation auf dem Bus nur vom Master initiiert werden kann. Der adressierte Slave kann daraufhin als Sender oder Empfänger agieren. Wiederrum kann ein Master auch als Sender oder Empfänger handeln.

Für eine Übertragung eines Byte-Paketes sind 8 Clock Impulse auf der SCL-Leitung erforderlich. Die Clock Impulse werden vom Master generiert. Um sicher zu stellen, dass der adressierte Slave die gesendeten Daten korrekt erhalten hat, wird vom Master noch ein 9.Clock-Impuls generiert. So hat der Slave die Möglichkeit zu bestätigen, dass er die Daten korrekt empfangen hat, was in der Kommunikationstechnik als Handshaking bezeichnet wird. Im I²C-System wird der 9. Clock-Impuls als Acknowledge-Bit abgekürzt ACK bezeichnet. Wenn also der Master ein Byte-Paket sendet, muss der Slave den Empfang des Byte-Paketes mit Acknowledge bestätigen. Das Acknowledge muss wiederum vom Master gelesen werden. Sollte der Master kein Acknowledge bekommen, wird die Datenübertragung beendet. Hat das ACK-Bit den Wert 1 (wird auch als NACK = Not Acknowledge bezeichnet) wird die Datenübertragung beendet. Hat das ACK-Bit den Wert 0, so wird die Datenübertragung fortgesetzt. Um die Kommunikation starten zu können, muss den Teilnehmern erst signalisiert

werden, dass der Master die Kommunikation beginnen möchte, um ihnen die Gelegenheit zu geben, sich auf den Datenstrom des Masters zu synchronisieren. Hierfür wird vom Master eine Startbedingung erzeugt, welche im nächsten Abschnitt weiter erläutert wird. [1] [12]

4.4.1 Startbedingung

Wenn gerade kein Teilnehmer kommuniziert und sich der Bus im Ruhezustand befindet, liegt an den SDA- und SCL-Leitungen, aufgrund der Pull-Up-Widerständen eine HIGH-Pegel an. Sobald ein Teilnehmer als Master eine Kommunikation starten möchte, zieht er den Pegel der SDA-Leitung auf LOW runter. Die SCL-Leitung bleibt währenddessen weiterhin auf HIGH-Pegel. Da während der regulären Datenkommunikation der Pegel der SDA Leitung sich nur während der Low-Phase des Taktsignales SCL ändern darf, stellt die Startbedingung eine Ausnahme dar, welche von allen Busteilnehmern detektiert werden kann. Die Teilnehmer verfolgen nun die Datenkommunikation und greifen nur lesend auf den Bus zu. Erst wenn der Bus vom Master freigegeben wird, darf ein ein anderer Busteilnehmer eine neue Datenübertragung beginnen. In der unten aufgeführten Abbildung 16 wird die Startbedingung veranschaulicht. [1]

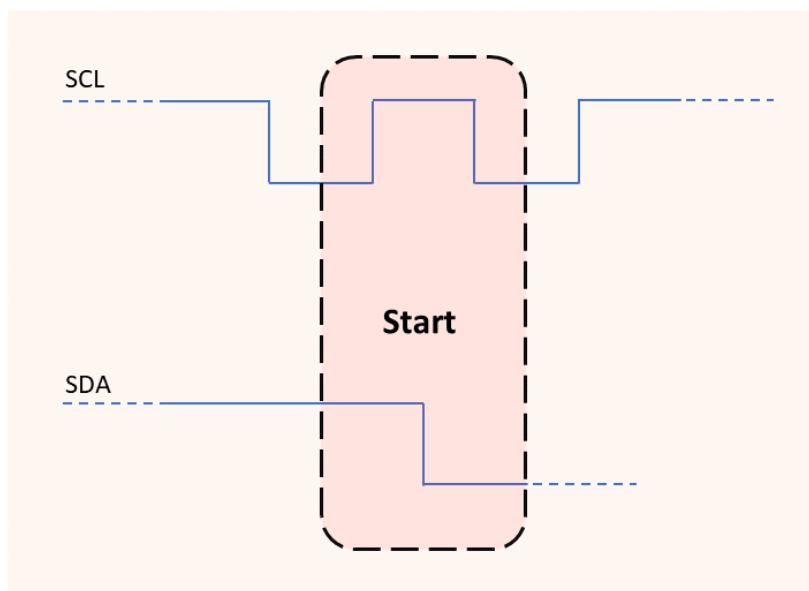


Abbildung 16: Startbedingung

4.4.2 Stoppbedingung

Um die Datenübertragung zu beenden, muss eine Stoppbedingung erzeugt werden. Es ist zu beachten, dass kein anderer Master die Stoppbedingung erzeugen darf als der, der auch die Startbedingung generiert hat. Bei der Stoppbedingung setzt der Master, die SDA-Leitung auf HIGH-Pegel während sich der SCL Pegel ebenfalls auf High Pegel befindet. Ähnlich wie bei der Startbedingung wird diese Abweichung vom regulären Kommunikationsfall, bei dem die SDA Leitung Ihren Pegel nur während der Low-Phase des SCL Signals wechseln darf, von allen Busteilnehmern detektiert und das Ende der Kommunikation erkannt. Somit erkennen alle Busteilnehmer, dass die Kommunikation beendet wurde. Nun kann jeder beliebige Master durch das Setzen einer Startbedingung eine neue Kommunikation starten. In der unten aufgeführten Abbildung 17 wird die Stoppbedingung veranschaulicht.

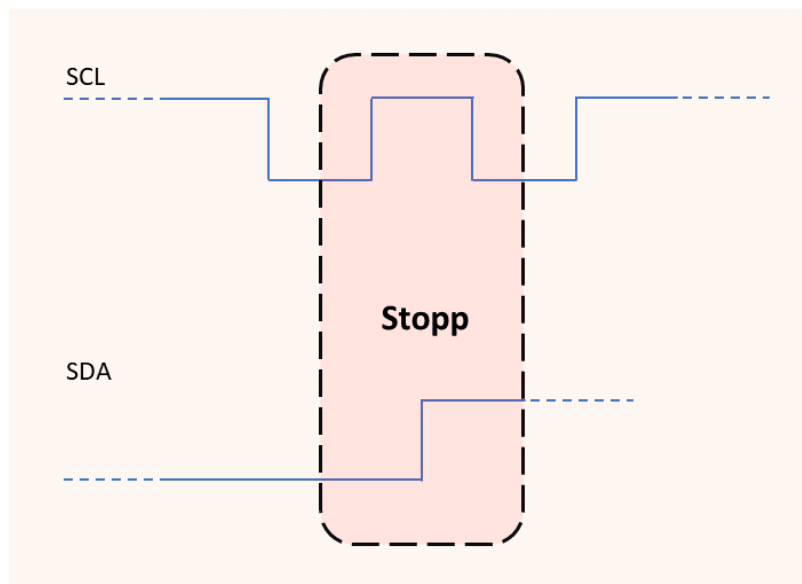


Abbildung 17: Stoppbedingung

Eine Besonderheit stellt die wiederholte Startbedingung (Repeated-Start) dar. Bei der wiederholten Startbedingung wird die Kommunikation nicht mit einer Stoppbedingung beendet. Stattdessen erzeugt der Master erneut eine Startbedingung. Somit kann der Master bei der Reinitialisierung der Kommunikation etwas Zeit einsparen. [1][11]

4.5 Adressierung

Nach der Erzeugung einer Startbedingung muss der Master zur Identifizierung des Bausteins die eindeutige Slave-Adresse übertragen. Bausteine, die nur als Master agieren benötigen keine Adresse. Bei der Datenübertragung wird das höchstwertige Bit MSB (Most Significant Bit) eines Byte-Pakets zuerst übertragen. Das LSB (Least Significant Bit) wird dementsprechend als letztes übertragen. Bei der ersten I²C-Version waren nur 7-Bit-Adressen vorgesehen. Mit 7 Bit lassen sich 128 Bausteine ansteuern. Allerdings wurden bereits bestimmte Adressen reserviert, sodass die Anzahl der Bausteine statt auf 128 auf 112 beschränkt ist. Da aber im komplexen System deutlich mehr Adressen benötigt werden, hat man die Länge des Adressfeldes auf 10 Bit erweitert. Damit können bis zu 1024 weitere Bausteine also insgesamt 1136 Bausteine adressiert werden, da 7-Bit-Adressen und 10-Bit-Adressen gleichzeitig verwendet werden können. [1]

4.5.1 Read und Write Bit

Nach den Adressbits folgt ein weiteres Bit, welches die Datenrichtung angibt. Ist dementsprechend das 8. Bit eine 1 weiß der Slave (in dem Fall der Sender), dass der Master (in dem Fall der Empfänger) Daten lesen möchte. Andernfalls weiß der Slave (in dem Fall der Empfänger), dass der Master (in dem Fall der Sender) Daten schreiben möchte.

4.5.2 7-Bit Adressierung

Das 7-Bit Adressformat wird in der Abbildung 18 veranschaulicht. In der Regel besteht die Slave-Adresse aus zwei Teilen. Generell sind die ersten 4-Bits (7-4) fest definierte Adressen des Bausteintyps. Die restlichen 3 Bits (3-1) können separat vergeben werden. Einige Bausteine bekommen in der Produktion eine feste Adresse, die nicht geändert werden kann. Andere Speicherbausteine lassen sich wiederum durch Jumper oder Lötbrücken einstellen. Bausteine, die nicht adressiert werden, warten bis zur nächsten Startbedingung. Der Baustein, welcher adressiert wurde, zieht das ACK-Bit herunter. In dem Fall wird die Datenübertragung vom Master fortgesetzt. Ansonsten wird die Verbindung unterbrochen. [1]

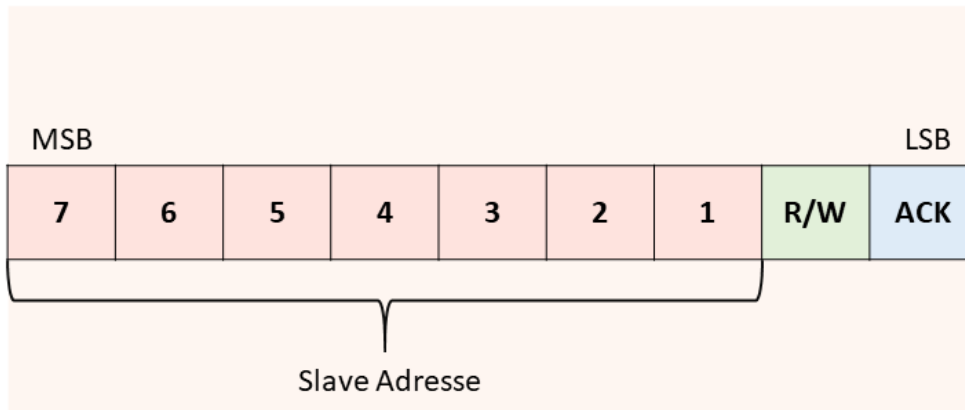


Abbildung 18: Format der 7-Bit-Slave-Adresse

4.5.3 Lesen und Schreiben bei 7-Bit-Adressierung

Wie bereits im vorherigen Unterkapitel erläutert, sind die ersten 7-Bit des ersten Byte-Pakets die Adresse des jeweiligen Bausteins. Das 8. Bit bestimmt die Datenrichtung. (1 = Lesen, 0=Schreiben). An der 9. Stelle kommt das ACK-Bit, welches vom Slave herunter gezogen wird. Damit bestätigt der Slave, dass er angesprochen wurde und dass er an der Kommunikation teilnehmen wird. Ab dieser Stelle können eine beliebige Anzahl von Byte-Pakete mit jeweils einem ACK gesendet oder empfangen werden. Sollte nach einem Byte-Paket kein ACK erfolgen, wird die Kommunikation vom Master abgebrochen. Um die Kommunikation erfolgreich beenden zu können, muss am Ende jedes Transfers die Stoppbedingung erzeugt werden.

Nach diesem Prinzip kann der Master Daten senden oder empfangen bzw. Daten lesen oder schreiben. Um die beiden Transfertypen besser verstehen zu können, werden beide Szenarien im Folgenden genauer erläutert. [1]

1. Master sendet Daten

In diesem Fall bleibt die Datenrichtung bestehen, bis der Master eine Stoppbedingung erzeugt (siehe Abbildung 19). Es kann aber auch sein, dass der Master, um etwas Zeit zu sparen, die erneute Startbedingung erzeugt, um dann einen anderen Baustein zu adressieren.

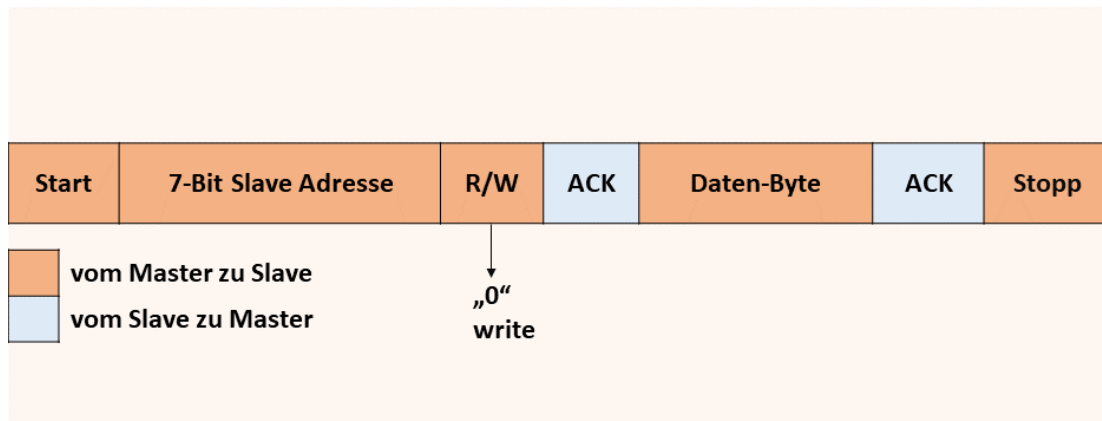


Abbildung 19: Schreibvorgang mit 7-Bit-Adressierung

2. Master empfängt Daten

Um Daten vom Slave lesen zu können, wird die 7-Bit-Adresse des jeweiligen Slaves gesendet. Das 8. Bit muss in dem Fall den Wert „1“ entsprechen. Der Slave erzeugt ein ACK-Bit und die Datenrichtung wird anschließend umgekehrt. Der Master wechselt in den Empfangsmodus, während der Slave in den Sendemodus wechselt. Ab diesem Zeitpunkt wird das ACK-Bit nach Erhalt von jedem Byte-Paket vom Master erzeugt. Wenn der Master die Kommunikation beenden möchte generiert er kein ACK-Bit mehr, sondern die Kommunikation wird mit einem NACK-Bit gefolgt von der Stoppbedingung beendet. Siehe dazu die folgende Abbildung. [1]

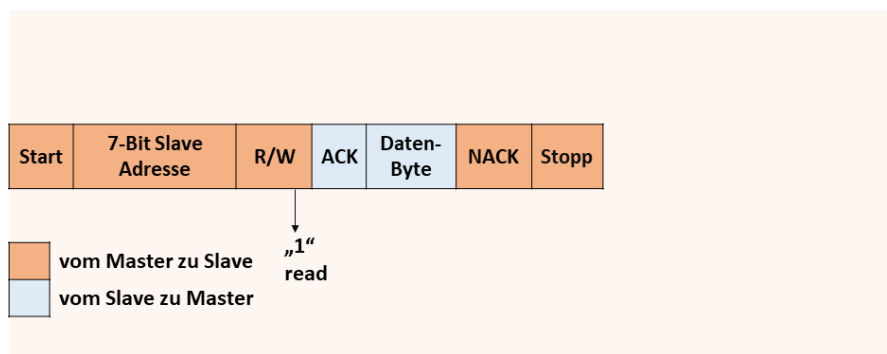


Abbildung 20: Lesevorgang mit 7-Bit-Adressierung

4.5.4 10-Bit-Adressierung

Für den Fall, dass in einem I²C-System keine 7-Bit-Adresse mehr zur Verfügung steht, wurde 10-Bit-Adressmodus definiert. Beide Verfahren sind miteinander kombinierbar. Anders formuliert heißt das, dass es nicht dringend notwendig ist, dass in einem System nur 10-Bit oder 7-Bit-Adressen vorkommen müssen. Es können beispielsweise in einem System 10-Bit-Adressen vergeben werden, obwohl es weniger als 112 Teilnehmer hat. Diesbezüglich gibt es keine Vorschriften und jedem Anwender ist es selbst überlassen, wie er die Adressen vergibt.

Bei einer 10-Bit-Adressierung wird die Slave-Adresse in zwei separaten Byte-Paketen gesendet. Damit alle Busteilnehmer wissen, dass nun eine 10-Bit-Adresse folgt, wird ein Code eingesetzt. Im ersten Byte-Paket sind die ersten 5 Bits mit dem Code „11110“ belegt. Die letzten beiden Bits enthalten den ersten Teil der Slave-Adresse. Folglich kommt nach der Bestimmung der Datenrichtung und dem ACK-Bit das zweite Byte-Paket. In diesem Byte-Paket bilden alle 8 Bits die Slave-Adresse. Dieses Format wird in Abbildung 21 veranschaulicht. Ab dieser Stelle geht es wie bei der 7-Bit-Adressierung weiter. Aus diesem Grund wird der Lese- und Schreibvorgang nicht nochmal erläutert. [1]

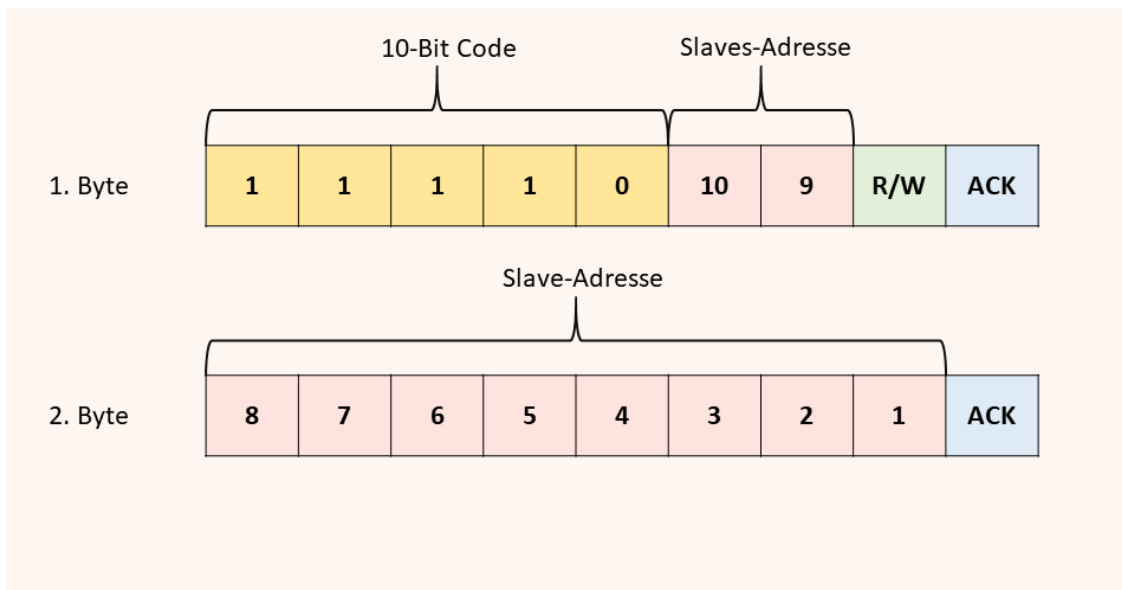


Abbildung 21: Format der 7-Bit-Slave-Adresse

5 EEPROM 24LC256

Zum Test der I2C Schnittstelle wurde der Speicherbaustein EEPROM 24LC256 von der Firma Microchip Technology Inc. verwendet. Dieser Baustein hat insgesamt 8 Pins und ist mit der I²C-Schnittstelle kompatibel. Er verfügt über eine elektronisch löschbare Speicherkapazität von 32 K x 8-Bit. Außerdem kann er mit einer Spannung im Bereich von 1,8V-5,5V betrieben werden. Die drei Adressleitungen ermöglichen die Verbindung von bis zu 8 Geräte am selben Bus (siehe Abbildung 22). [13]

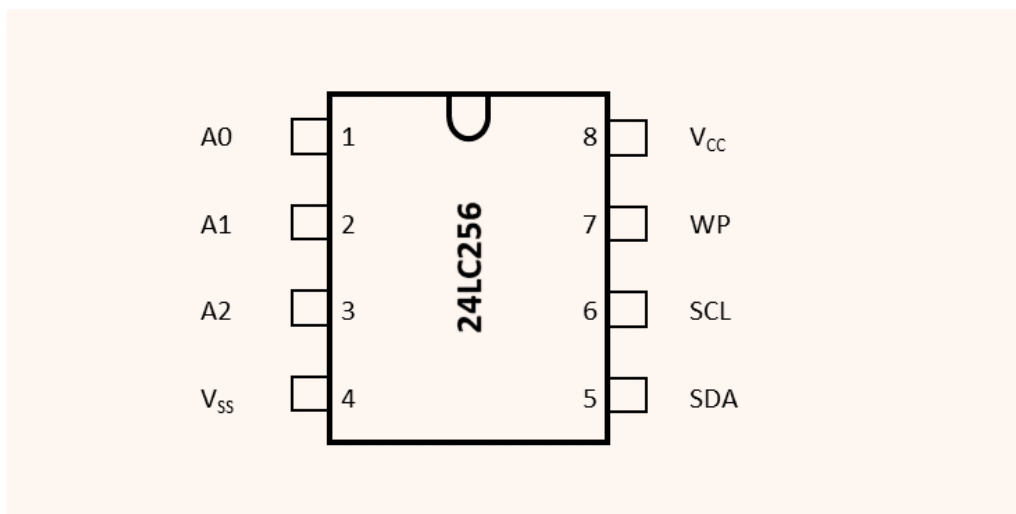


Abbildung 22: EEPROM 24LC256

5.1 Funktion

Der 24LC256 Speicherbaustein unterstützt das I²C-Protokoll und kann als Slave Daten senden oder empfangen. Der Bus muss demnach von einem Master gesteuert werden. Der Master generiert dann den Takt auf der SCL-Leitung und gegebenenfalls Daten auf der SDA-Leitung. Außerdem erzeugt der Master die Start- und Stoppbedingungen und entscheidet, ob der Baustein Daten senden oder empfangen soll. Die Datenübertragung kann bei einer Taktfrequenz von bis zu 400 kHz stattfinden.

Eine wichtige Besonderheit gibt es jedoch bei dem ACK-Bit. Es ist zu beachten, dass der Baustein kein ACK-Bit generiert, wenn gerade ein interner Programmierzyklus ausgeführt wird. Es wird daher empfohlen nach jedem Lese- oder Schreibvorgang fünf Millisekunden zu warten. [13]

5.2 Die Adresseingänge

Die Eingänge A0, A1 und A2 werden verwendet, um die Adresse des Bausteins einzustellen. Die Pegel dieser Eingänge werden mit den Bits in der Slave-Adresse verglichen. Der Chip erzeugt ein ACK-Bit, wenn der Vergleich wahr ist. Wenn die Adresseingänge nicht belegt sind, werden die Eingänge intern zum V_{SS} (Ground) gezogen. Bei einem HIGH-Pegel werden die internen Pull-Downs deaktiviert. [13]

5.3 Serial Data (SDA)

SDA ist ein bidirektionaler Pin, der zur Übertragung von Adressen und Daten verwendet wird. Wegen der inneren Open-Drain Schaltung benötigt die SDA-Leitung noch einen Pull-Up-Widerstand. Es wird empfohlen bei einer 100 kHz Taktung 10 k Ω und bei einer 400kHz Taktung 2 k Ω Widerstand zu nehmen. [13]

5.4 Serial Clock (SCL)

Dieser Eingang wird verwendet, um die Datenübertragung vom und zum Gerät zu synchronisieren. Der Serial-Clock kann nur vom Master erzeugt werden. [13] [10]

5.5 Write-Protection (WP)

Bei einem HIGH-Pegel wird „*Write-Protect*“ aktiviert. Das bedeutet, dass der Baustein schreibgeschützt ist und alle Schreibvorgänge sperrt. Lesevorgänge sind davon nicht betroffen. Bei einem LOW-Pegel ist der Normalbetrieb aktiviert. In diesem Betrieb können Daten gesendet oder empfangen werden. Wird der Pin nicht angeschlossen, wird der Baustein vom internen Schaltkreis in einem ungeschützten Zustand gehalten. In diesem Zustand kann es weder gesendet noch empfangen werden. [13]

5.6 Die Adressierung

Ein Adressbyte ist das erste Byte, welches nach der Startbedingung vom Master übertragen wird. Damit der 24LC256 richtig angesprochen werden kann muss das Byte-Paket ein 4-Bit Steuercode beinhalten. Bei diesem Speicherbaustein ist der 4-Bit Code als „1010“ deklariert. Die nächsten drei Bits des Steuerbytes sind die Chip-Auswahlbits (A2, A1, A0). Das letzte Bit in dem Byte-Paket bestimmt die Richtung, wobei die nächsten zwei Bytes die genaue Adresse

des ersten Datenbytes definieren. Da der Speicherbaustein eine Gesamtspeicherkapazität von 32 K x 8 Bit hat und dementsprechend 2^{15} Bits verarbeiten kann, werden von LSB angefangen 15 Bits eingelesen. Der MSB wird nicht berücksichtigt. Dies wird in der folgenden Abbildung nochmal veranschaulicht. [13]

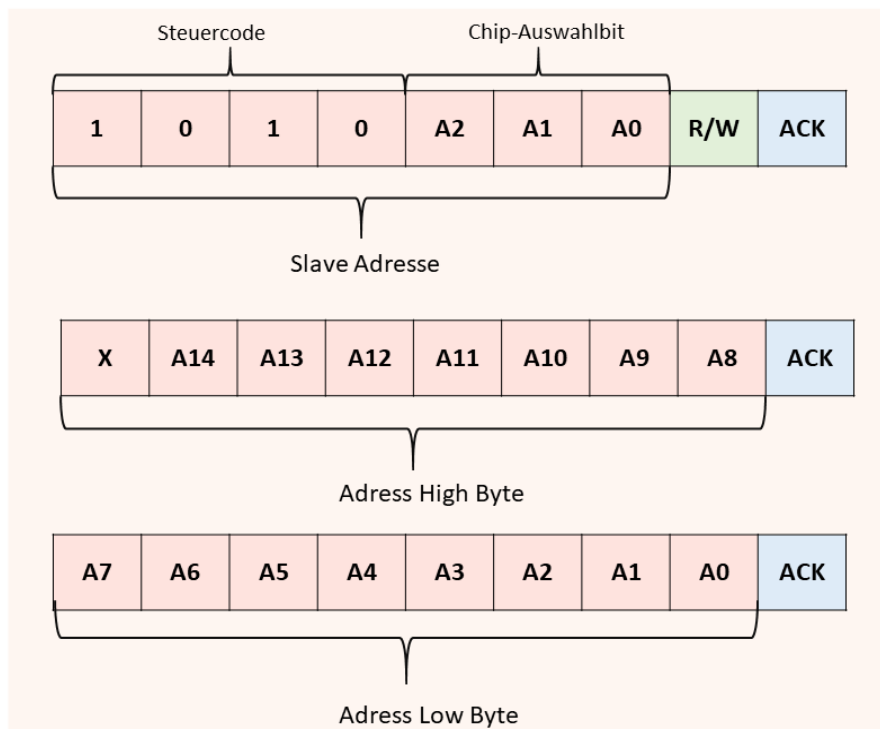


Abbildung 23: Bitmuster für die Adressierung des 24LC256

5.7 Schreibvorgang

Nach dem die Startbedingung vom Master erzeugt wird, das Adressbyte übertragen werden. Wenn das LSB des Adressbytes eine „0“ ist, weiß der 24LC256 Speicherbaustein, dass der Master Daten senden möchte. Es wird erst der Zustand der WP-Leitung überprüft. Bei einem HIGH-Pegel an der WP-Leitung erzeugt der Slave zwar ein ACK-Bit aber startet nicht den internen Schreibvorgang. Bei einem HIGH-Pegel an der WP-Leitung wird der innere Schreibvorgang des Speicherbausteins gestartet, wobei jedoch die empfangenen Daten erst mal nur vorgehalten und nicht im nichtflüchtigen Speicher geschrieben werden. Nachdem die Adressbytes übermittelt wurden, kommt als nächstes das Datenbyte. Als letztes kommt die vom Master generierte Stoppbedingung. Der 24LC256 Speicherbaustein führt erst nach Detektion der Stopp-Bedingung die Speicherung der Daten. Aus diesem Grund benötigt er 5 Millisekunden Zeit, bis er den nächsten Lese-Schreibvorgang annehmen kann.

Es ist unbedingt zu beachten, dass nach jedem erfolgreich übermittelten Byte ein ACK-Bit folgen muss. Der Slave hält den ACK-Bit solange unten, bis er für den Empfang des nächsten Byte-Paketes bereit ist (siehe Abbildung 24). Wenn überhaupt kein ACK-Bit erzeugt wird, bricht der Master die Datenübertragung ab. [10][13]

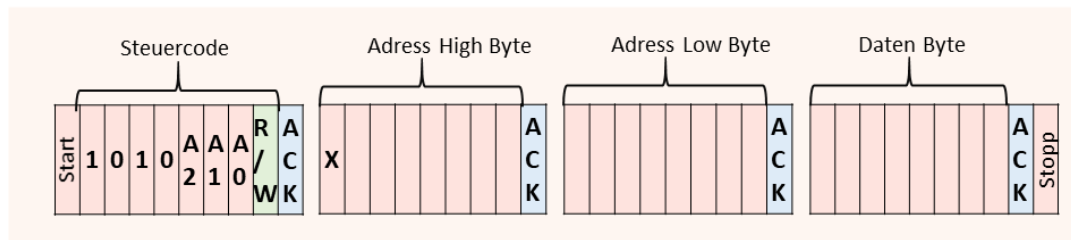


Abbildung 24: Byte Write

Der 24LC256 Speicherbaustein besitzt einen 2^6 -Bit großen Zwischenspeicher. Das heißt, dass der Master nach dem Senden des Daten-Bytes noch 63 weitere Bytes senden könnte. Die Adresse wird dabei immer um eins inkrementiert. Sendet der Master mehr als 64 Bytes, werden die Daten, die sich im Zwischenspeicher befinden, überschrieben. Erst nach Erhalt der Stoppbedingung beginnt der Speicherbaustein die Daten, die im Zwischenspeicher liegen, abzuspeichern (siehe Abbildung 25). Dabei ist zu beachten, dass die WP-Leitung auf einem LOW-Pegel liegen muss. [13]

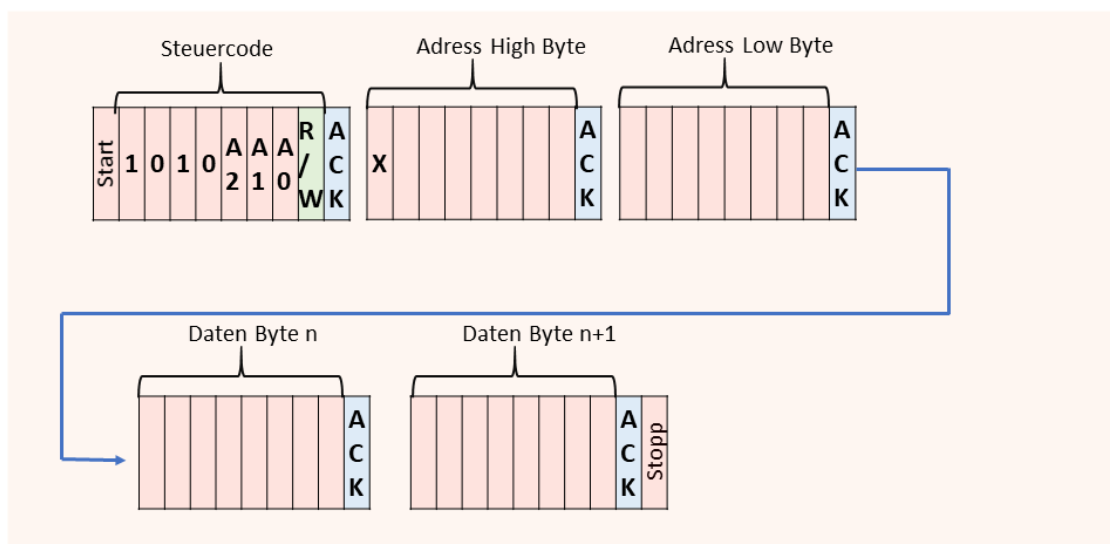


Abbildung 25: Page-Write

5.8 Lesevorgang

Lesevorgänge werden auf die gleiche Weise wie Schreibvorgänge initiiert, mit der Ausnahme, dass das R/W Bit hier auf „1“ gesetzt ist. Es gibt drei grundlegende Arten von Lesevorgängen. Es kann eine aktuelle, bestimmte oder sequenzielle Adresse gelesen werden.

Der 24LC256 Speicherbaustein enthält einen Adresszähler, der die Adresse des zuletzt aufgerufenen Wortes aufbewahrt. Bei jedem weiteren Lesevorgang wird die Adresse um eins inkrementiert. So kann der Inhalt, der sich in der aktuellen Adresse befindet vom Master empfangen werden. Nachdem der Master die Daten empfangen hat, wird vom ihm kein ACK-Bit erzeugt. Anschließend kommt die vom Master erzeugte Stoppbedingung (siehe Abbildung 26).

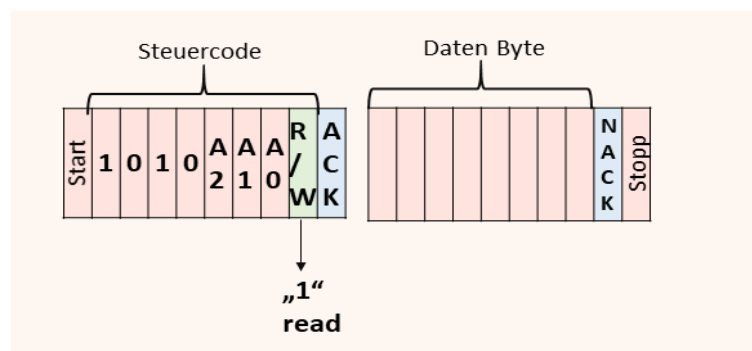


Abbildung 26: Aktuelle Adresse lesen

Durch die folgende Methode kann der Master auf einen definierten beliebigen Speicherplatz zugreifen. Um diese Art von Leseoperation auszuführen, muss zuerst die genaue Datenadresse eingestellt werden. Dabei wird das R/W-Bit als „0“ gesetzt. Nachdem die Datenadresse vollständig übermittelt wurde, erzeugt der Master eine neue Startbedingung. An dieser Stelle wird das R/W-Bit als „1“ gesetzt. Wenn der Master die Daten empfangen hat, wird vom Master kein ACK-Bit erzeugt. Anschließend kommt die vom Master erzeugte Stoppbedingung (siehe Abbildung 27). [13]

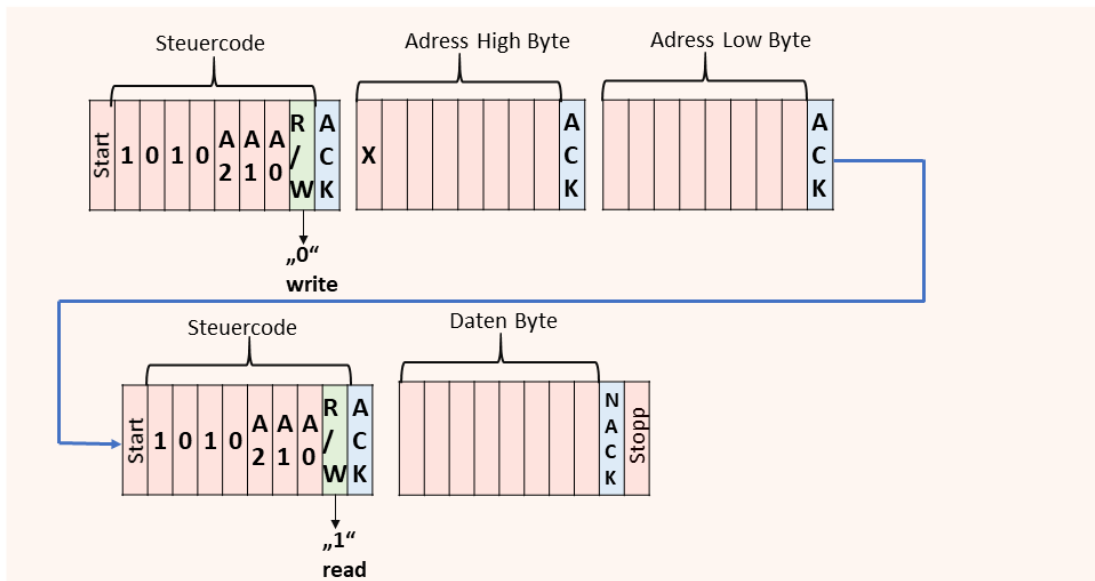


Abbildung 27: Bestimmte Adresse lesen

Sequentielle Lesevorgänge werden auf dieselbe Weise wie Lesevorgänge auf definierte Speicherstellen initiiert, mit der Ausnahme, dass der Master nach der Übertragung des ersten Datenbytes ein ACK-Bit erzeugt. Solange nach jedem Datenbyte der Master ein ACK-Bit erzeugt, werden die Daten fortlaufend vom Slave an den Master gesendet. Dabei wird die Adresse jedes Mal um eins inkrementiert. Mit einer Stoppbedingung wird der Datentransfer beendet (siehe Abbildung 28).

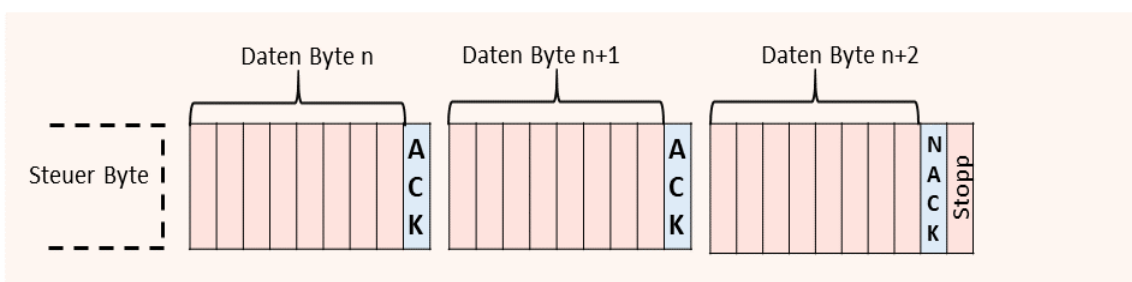


Abbildung 28: Sequentieller Lesevorgang

6 Konfiguration und Inbetriebnahme

6.1 MPSSE

Die Multiprotokoll-Synchronous-Serial-Engine (MPSSE) von FTDI bietet eine flexible Möglichkeit serielle Geräte an einen USB-Anschluss anzuschließen. Durch die Verwendung des "Multiprotokoll" Ansatzes, ist es möglich die MPSSE für die Kommunikation mit vielen verschiedenen Arten von synchronen Geräten zu ermöglichen. SPI, I²C und JTAG sind demnach auch mit der MPSSE realisierbar. Datenformatierung und Taktsynchronisation können in vielfältiger Weise konfiguriert werden, um beinahe jede Anforderung zu erfüllen. Neben den seriellen Datenpins stehen zusätzliche GPIO-Signale zur Verfügung.

Die MPSSE ist ein Master-Controller für die ausgewählte synchrone Schnittstelle. Als solches generiert er den Takt und alle erforderlichen Schnittstellenauswahl- sowie Chipauswahlsignale. [9][8][6]

6.1.1 Pinbelegung bei Nutzung der MPSSE

Für jeden MPSSE-Kanal stehen 8 Pins zur Verfügung. Tabelle 3 gibt die Pinbelegung bei Verwendung des Bausteins für die I2C Kommunikation wieder. In Tabelle 1 kann die Pinbelegung von allen Pins nachgelesen werden. [4]

Kanal 2 (CN2)		Kanal 3 (CN3)		Daten- richtung	Beschreibung
Pin	Pin Name	Pin	Pin Name		
7	ADBUS0	26	BDBUS0	Output	SK, Serial Clock
10	ADBUS1	25	BDBUS1	Output	DO, Serial Data / Adress Output
9	ADBUS2	24	BDBUS2	Input	DI, Serial Data Input
14	ADBUS4	21	BDBUS4	Output	GPIO, Write Protect control Output

Tabelle 3: MPSSE Pinbelegung

6.1.2 Datenrate (Clock)

Bei einer Kommunikation muss die korrekte Übertragungsgeschwindigkeit eingestellt werden. Ansonsten wird das Zielgerät nicht fehlerfrei angesprochen. Nach der Aktivierung des MPSSE-Modus müssen als nächstes die MPSSE-Einstellungen initialisiert werden. Die Software baut

einen Puffer mit mehreren MPSSE-Befehlen auf und sendet sie dann mit der Funktion „*FT_Write D2xx*“ an den Chip. Die erste an das Gerät gesendete Einstellung entspricht der Deaktivierung des internen Taktteilers, sodass die MPSSE einen 60-MHz-Takt von der internen Taktschaltung des *FT232H* empfängt. Zusätzlich gibt es für die Taktung zwei Modi. Die Zwei- und Dreiphasentaktung. Bei einer Dreiphasentaktung besteht jedes Bit aus drei Takthalbzyklen. Im dreiphasigen Modus bleiben die Daten stattdessen für drei Halbtaktperioden auf der Leitung und der Takt wechselt zweimal. In *Abbildung 29* werden die Modi veranschaulicht. Beim üblichen zweiphasigen Taktmodus werden die Daten mit der fallenden Flanke des Taktsignals SCL auf den Bus gelegt und für eine SCLTaktperiode auf der Leitung gehalten. Der Takt wechselt in diesem Fall seinen Pegel in etwa der Mitte des Bitdauer des Datenbits. Bei einer Dreiphasentaktung verändert sich der SDA Signalpegel bei einem SCL Low-Pegel und zwar in etwa zur Hälfte der Halbperiodendauer.

Das Ergebnis zeigt, dass die Daten auf beiden Taktflanken stabil sind, welches für I2C eine Voraussetzung ist. In der unteren Darstellung ist die Dreiphasentaktung aktiviert. Das SDA Signal ist im Falle der Dreiphasentaktung um etwa 90° bzw. um etwa ein Viertel der Periodendauer im Vergleich zum Signal bei Zweiphasentaktung phasenverschoben. Es erscheint daher wie vom I2C-Protokoll gefordert sowohl eine steigende als auch eine fallende Flanke, während die Daten-Leitung stabil ist. [9][6]

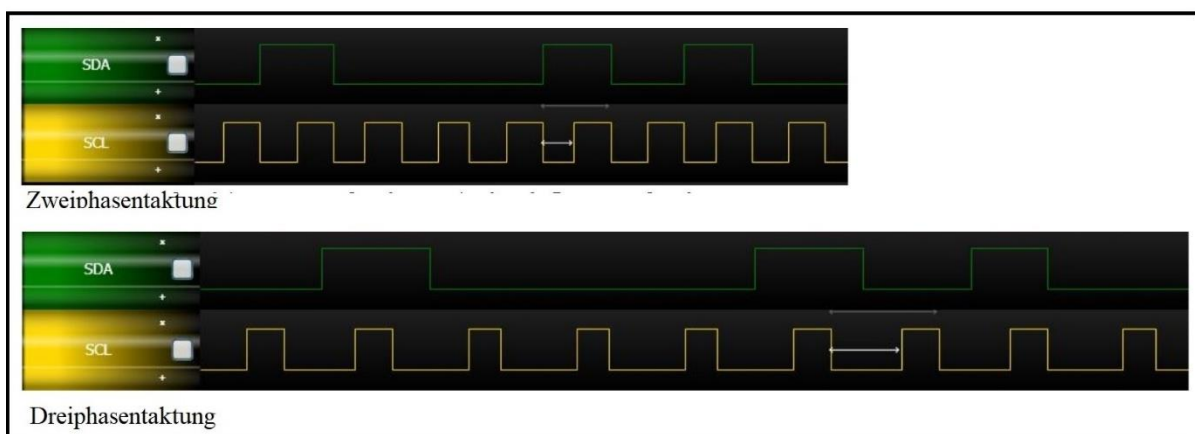


Abbildung 29: Zwei- und Dreiphasentaktung [6]

Schließlich wird die Taktfrequenz eingestellt. Dafür muss zunächst der Wert des Clockdividers ermittelt werden, was mit der unten dargestellten Formel ermittelt werden kann. Die Taktfrequenz (Clock) soll zunächst auf 100 kHz gesetzt werden. Dies wird durch Teilung des 60 MHz-Takts erzeugt, welcher intern an die MPSSE geliefert wird.

$$\text{Clock} = \frac{60 \text{ MHz}}{(1 + \text{Clockdivider}) * 2}$$

$$100 * 10^3 \text{ Hz} = \frac{60 * 10^6 \text{ Hz}}{(1 + \text{Clockdivider}) * 2}$$

Die Formel wird nun nach Clockdivider umgeformt.

$$100 * 10^3 \text{ Hz} * (1 + \text{Clockdivider}) * 2 = 60 * 10^6 \text{ Hz}$$

$$\text{Clockdivider} + 1 = \frac{60 * 10^6 \text{ Hz}}{100 * 10^3 \text{ Hz} * 2}$$

$$\text{Clockdivider} = \frac{60 * 10^6 \text{ Hz}}{100 * 10^3 \text{ Hz} * 2} - 1$$

$$\text{Clockdivider} = 299$$

Bevor der endgültige Wert für den Teiler bestimmt werden kann, muss jedoch auch die Auswirkung des 3-Phasen-Taktmodus berücksichtigt werden. Die obige Standardberechnung nimmt den normalen Zweiphasenmodus an. Wie oben dargestellt ist ein Zyklus im Dreiphasenmodus tatsächlich um 1/2 Periodendauerlänger als der Zweiphasenzyklus. Die gesamte Dauer eines Zyklusses entspricht d.h. 3/2 einer Taktperiode. Damit die Übertragungsrate beim Dreiphasenmodus genauso groß ist wie beim Zweiphasenmodus, muss die Periodendauer mit dem Faktor 2/3 multipliziert werden. Was einer Erhöhung der Taktfrequenz um den Faktor 3/2 entspricht. Der CLOCKDIVIDER-Wert muss also mit dem Faktor 2/3 multipliziert werden, um eine Erhöhung der Frequenz um den Faktor 3/2 zu erzielen. [6]

$$\text{Clockdivider} = 299 * \frac{2}{3} \approx 200$$

6.1.3 Konfiguration der Pins

In diesem Kapitel wird erläutert, wie die Anfangszustände und die Datenrichtung der jeweiligen Pins, festgelegt wird. Dafür wird zuerst die Tabelle 4 und der Quellcode betrachtet.

Pin Name	Signal	Datenrichtung		Anfangszustand	
ADBUS0	SK	1 LSB	output	1 LSB	High
ADBUS1	DO	1	output	0	Low
ADBUS2	DI	0	input	0	z
ADBUS3	GPIO0	1	output	1	High
ADBUS4	GPIO1	1	output	0	Low
ADBUS5	GPIO2	1	output	0	Low
ADBUS6	GPIO3	1	output	1	High
ADBUS7	GPIO4	1 MSB	output	1 MSB	High

Tabelle 4: Anfangszustände und Datenrichtung der MPSSE-Schnittstelle

```
OutputBuffer[dwNumBytesToSend++] = 0x80
```

```
OutputBuffer[dwNumBytesToSend++] = 0xC9;
```

```
OutputBuffer[dwNumBytesToSend++] = 0xFB;
```

Es ist zu beachten, dass die Reihenfolge des Quellcodes nicht vertauscht werden darf. Zunächst wird der Port mit dem Befehl „0x80“ ausgewählt. Als nächstes wird der Anfangszustand des jeweiligen Pins festgelegt. Der Hex-Wert „0xC9“ entspricht in Binär dem Wert „11001001“. Nun wird der Wert angefangen mit LSB in die Spalte „Anfangszustand“ eingetragen. Die Pins die eine „1“ zugewiesen bekommen, haben einen HIGH-Pegel und Pins mit „0“ dementsprechend einen LOW-Pegel.

In der nächsten Zeile des Quellcodes wird die Datenrichtung festgelegt. Der Hex-Wert „0xFB“ entspricht in Binär dem Wert „11111011“. Auch hier wird der Wert angefangen mit LSB in die Spalte „Datenrichtung“ eingetragen. Die Pins die eine „1“ zugewiesen bekommen haben, agieren als Output und bei „0“ agiert der Pin als ein Eingang. Es ist zu beachten, dass ein als Ausgang festgelegter Pin kein Anfangszustand hat. [8]

6.2 Die Schaltung

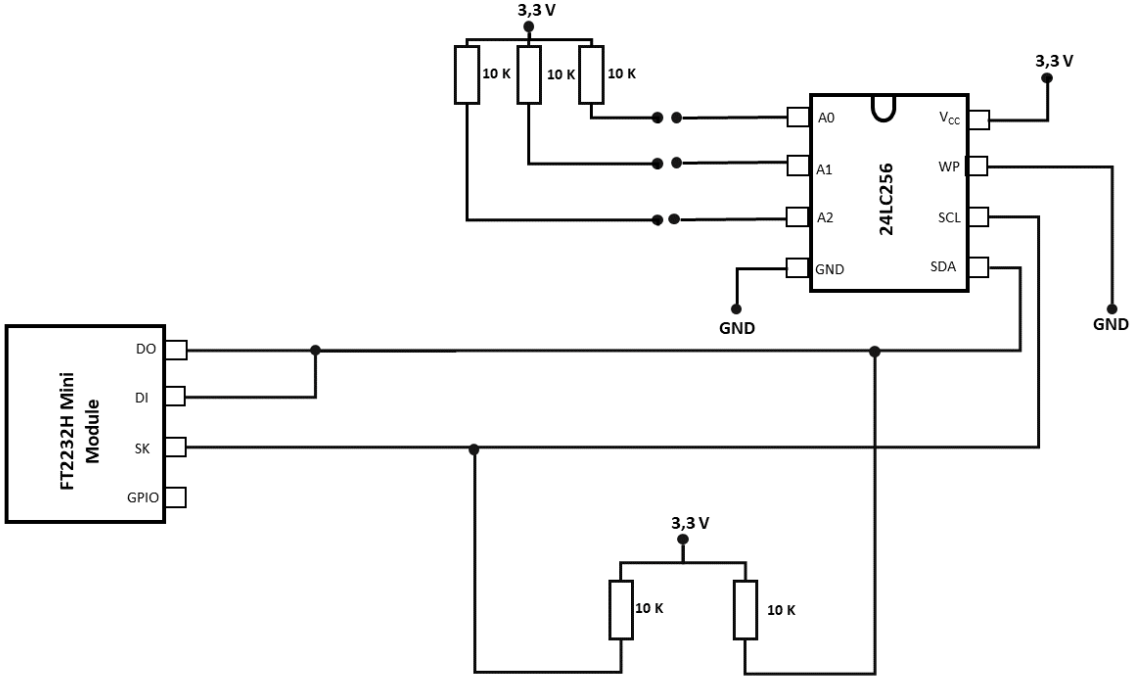


Abbildung 30: Schaltung FTDI - EEPROM

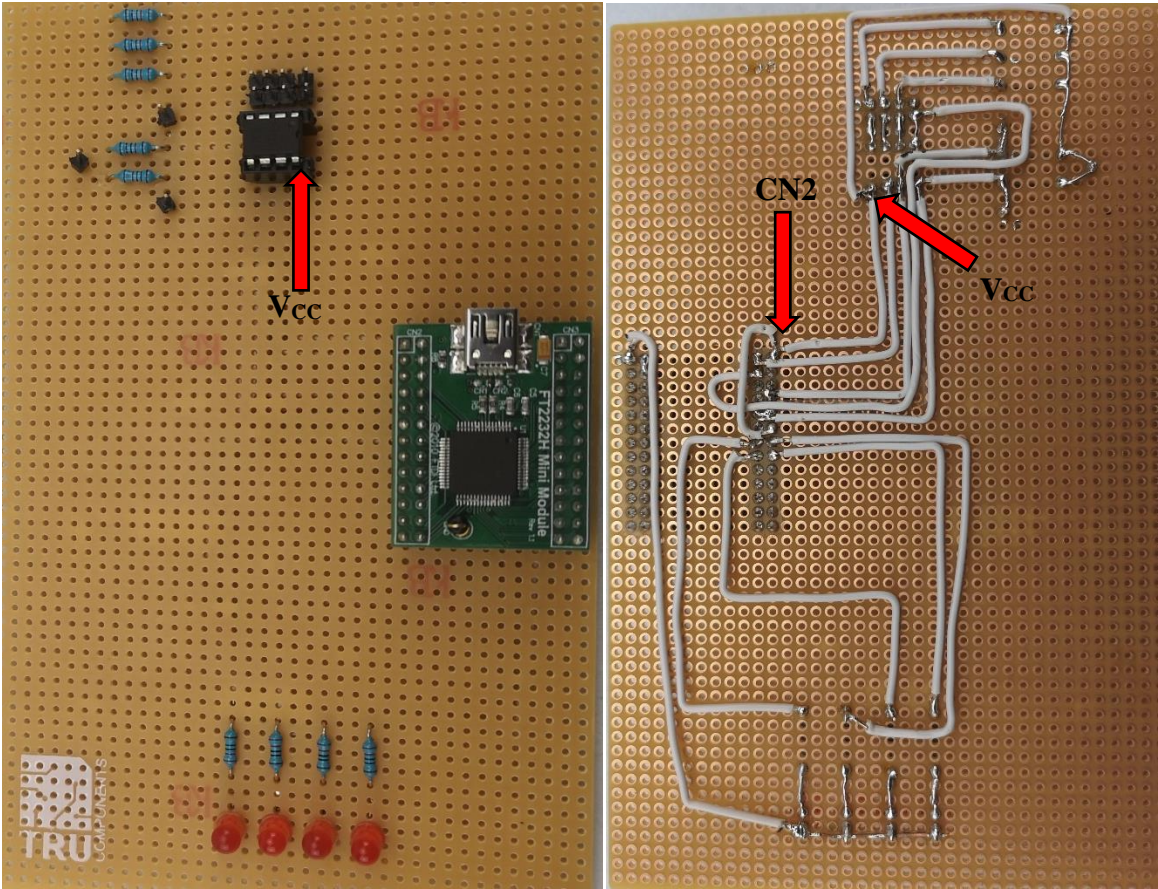


Abbildung 31: Platine Vor- und Rückderseite

Die Abbildung 30 veranschaulicht die Verbindungen, welche zwischen dem des FTDI Baustein und dem seriellen EEPROM Speicherbaustein bestehen. In Abbildung 31 wird die selbst erstellte Platine dargestellt.

Zunächst wird der SCL-Pin vom 24LC256 Speicherbaustein mit dem SK-Pin des FTDI Moduls verbunden. Dadurch kann die Datenübertragung synchronisiert werden. Das Clocksignal wird vom FT2232H Mini Modul generiert und an den 24LC256 Baustein geführt. Der Taktwert wird vom internen Takteiler FT2232H Mini Modul bestimmt und kann bis zu 30 MHz betragen. Das Taktsignal erfordert einen Pull-up-Widerstand für die Spannungsversorgung V_{CC} (typisch $10\text{ K}\Omega$ für 100 kHz und $2\text{ K}\Omega$ für 400 kHz). DO und DI werden zunächst zusammen verdrahtet und mit dem SDA-Pin des 24LC256 Speicherbausteins für die Datenübertragung verbunden. DO agiert als Ausgangspin, um serielle Daten oder Adressen von FT2232H Mini Module an 24LC256 Speicherbaustein zu übertragen. DI agiert als Eingangspin, um serielle Dateneingangspin von 24LC256-Speicherbaustein an FT2232H Mini Modul zu empfangen. GPIO wird zunächst nicht verbunden. [14]

Da nun die Schaltung betriebsbereit ist, kann mit der Programmierung begonnen werden.

7 Programmierung des Moduls

7.1 Verwendete D2XX Methoden

Um das Modul korrekt anzusprechen und steuern zu können, müssen die einzelnen Treibermethoden korrekt und in richtiger Reihenfolge aufgerufen werden. Ansonsten kann die MPSSE nicht korrekt konfiguriert werden, sodass die I2C-Schnittstelle nicht aktiviert wird. FTDI liefert sehr viele Methoden. Die wichtigsten bzw. die bei der Programmierung verwendeten Methoden werden unten näher beschrieben. Die Liste mit allen verfügbaren Methoden sind im „*D2XX Programmers guide*“ zu finden⁵. [3]

7.1.1 FT_GetDeviceInfoList

Mit der Funktion können hardware-spezifische Infos abgefragt werden. Die Funktion kann nur dann genutzt werden, wenn das Gerät auch angeschlossen ist. [3]

7.1.2 FT_Open

Diese Methode öffnet das Gerät. Hierbei ist zu beachten, dass mit dem Befehl das Gerät nur geöffnet wird. Die Methode kann auch genutzt werden, um zu überprüfen, ob der D2XX Treiber korrekt installiert wurde. Denn wenn der Treiber oder die Einstellungen im Qt-Creator nicht korrekt sind, kann das Modul nicht geöffnet werden. Daher ist die Nutzung dieser Methode der einfachste Weg, um zu überprüfen, ob alles korrekt konfiguriert wurde. [3]

7.1.3 FT_Close

Diese Methode wird verwendet, um die bestehende Verbindung zu beenden. Sollte die Methode *FT_Open* verwendet werden, muss die Verbindung mit *FT_Close* beendet werden. Die Auslassung des *FT_Close* Befehls führt zwar zu keiner Fehlermeldung, kann in einem komplexen Programm, in dem mehrmals *FT_Open* verwendet wird, jedoch zu Problemen führen. Beispielsweise wird mit der Methode *FT_Open* Befehl das Modul geöffnet und

5

[http://www.ftdichip.com/Support/Documents/ProgramGuides/D2XX_Programmer's_Guide\(FT_000071\).pdf](http://www.ftdichip.com/Support/Documents/ProgramGuides/D2XX_Programmer's_Guide(FT_000071).pdf)

anschließend ein Pin auf High gesetzt. Sobald der Compiler aus der Schleife aussteigt, ohne dass die *FT_Close* Methode aufgerufen wurde, kann kein weiterer Pin ein oder ausgeschaltet werden. [3]

7.1.4 FT_Write

Diese Methode wird verwendet, um auf das Modul Daten zu übertragen. Die Daten können in Form eines Buffers oder als einzelne Bits gesendet werden. Die Form der Übertragung muss bei jeder Verwendung von *FT_Write* auch festgelegt werden. [3]

7.1.5 FT_SetBitMode

Diese Methode wird verwendet, um die Datenflussrichtung zu bestimmen. Dementsprechend wird festgelegt, welche Pins Ein- oder Ausgänge sind. Mit dem Befehl „*FT_SetBitMode(ftHandle, 0xFF, FT_BITMODE_ASYNC_BITBANG);*“ werden 8 Pins als Ausgang konfiguriert. Zudem kann auch bestimmt werden, in welchem Modus die Daten übertragen werden sollen (vgl. FTDI Chip, 2012b). Es kann zwischen folgenden Modi unterschieden werden: [3]

0x0 = *Reset*

0x1 = *Asynchronous Bit Bang*

0x2 = *MPSSE (FT2232, FT2232H, FT4232H and FT232H devices only)*

0x4 = *Synchronous Bit Bang (FT232R, FT245R, FT2232, FT2232H, FT4232H and FT232H devices only)*

0x8 = *MCU Host Bus Emulation Mode (FT2232, FT 2232H, FT4232H and FT232H devices only)*

0x10 = *Fast Opto-Isolated Serial Mode (FT2232, FT2232H, FT4232H and FT232H devices only)*

0x20 = *CBUS Bit Bang Mode (FT232R and FT232H devices only)*

0x40 = *Single Channel Synchronous 245 FIFO Mode (FT2232H and FT232H devices only)*

7.1.6 FT_ListDevices

Diese Methode wird verwendet, um die Informationen zu den derzeit verbundenen Geräten aufzurufen. Diese Funktion kann Informationen, wie bspw. die Anzahl der verbundenen Geräte, die Seriennummer des Geräts, die Zeichenfolgen der Gerätebeschreibung oder die Standort-IDs des angeschlossenen Geräts zurückgeben. [3]

7.1.7 FT_ResetDevice

Diese Methode dient dazu, um das angeschlossene Gerät zurückzusetzen. [3]

7.1.8 FT_GetQueueStatus

Um die Anzahl der Bytes, die sich in dem Zwischenspeicher befinden, abzurufen, wird diese Methode benutzt. [3]

7.1.9 FT_SetUSBParameters

Diese Methode bewirkt, dass die Übertragungsgrößen von der Standardübertragungsgröße von 4096 Byte auf einen anderen Wert geändert werden. Um die Anforderungen der Anwendung besser zu erfüllen. Die Übertragungsgrößen müssen einem Vielfachen von 64 Byte entsprechen und zwischen 64 Byte und 64 KB liegen. Wenn *FT_SetUSBParameters* aufgerufen wird, sind die Änderungen sofort wirksam und alle Daten, die zum Zeitpunkt der Änderung im Treiber gespeichert waren, gehen verloren. [3]

7.1.10 FT_SetChars

Diese Methode wird benutzt, um Sonderzeichen für das Gerät festzulegen. [3]

7.1.11 FT_SetTimeouts

Diese Methode legt die Lese- und Schreib-Timeouts für das Gerät fest. [3]

7.1.12 FT_SetLatencyTimer

Diese Methode wird verwendet, um die Latenzzeit festzulegen. Dieses Timeout ist programmierbar und kann im Abstand von 1 ms zwischen 2 ms und 255 ms eingestellt werden. Dadurch kann das Gerät besser für Protokolle optimiert werden, die kürzere Antwortzeiten von kurzen Datenpaketen erfordern. [3]

7.2 Test Programm unter Qt-Creator

Für einen einfach ersten Test wird die folgende Main-Datei in der Qt Entwicklungsumgebung Qt-Creator verwendet. Zunächst werden die benötigten Header eingebunden. In diesem Fall sind es die Folgenden:

```
+++++
#include <wintypes.h>
#include <ftd2xx.h>
#include <stdio.h>
#include <iostream>
#include <stdlib.h>
#include <assert.h>
+++++
```

Als erstes wird überprüft, was für ein Modul angeschlossen ist. Dafür kann auch das von FTDI mitgelieferte Testprogramm gestartet werden. Der Programmcode wird weiter unten aufgeführt.

```
+++++
int main()
{
    FT_STATUS ftStatus;
    DWORD numDevs;
    if (numDevs > 0) {
        // get the device information list
        ftStatus = FT_GetDeviceInfoList(devInfo, &numDevs);
        if (ftStatus == FT_OK) {
            printf("Dev %d:\n", numDevs);
            printf("  Flags=0x%x\n", devInfo[numDevs].Flags);
            printf("  Type=0x%x\n", devInfo[numDevs].Type);
            printf("  ID=0x%x\n", devInfo[numDevs].ID);
            printf("  LocId=0x%x\n", devInfo[numDevs].LocId);
            printf("  SerialNumber=%s\n", devInfo[numDevs].SerialNumber);
            printf("  Description=%s\n", devInfo[numDevs].Description);
            printf("  ftHandle=0x%x\n", devInfo[numDevs].ftHandle);
        }
    }
}
```

```
    return 0;
}
```

+++++

Das Programm führt eine detaillierte Abfrage bezüglich der angeschlossenen FTDI Module durch und erstellt darüber hinaus eine Liste mit den Ergebnissen. Die Liste wird in *Abbildung 32* dargestellt.

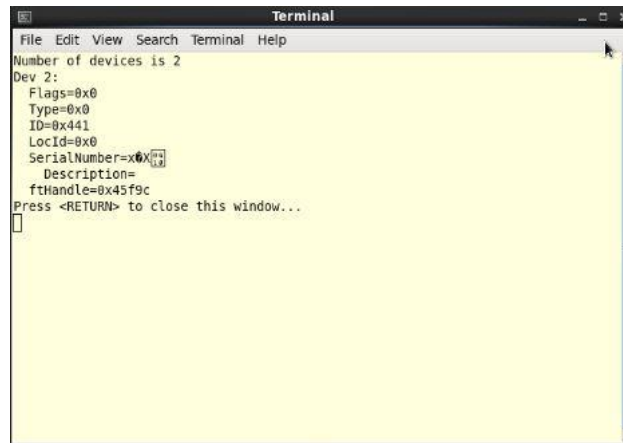


Abbildung 32: Ergebnis des Programms

7.3 Startseite der grafischen Benutzeroberfläche

Wenn das Programm ausgeführt wird, bekommt der Benutzer erstmal die Startseite (siehe *Abbildung 34*) zu sehen. An dieser Stelle muss der Benutzer sich einloggen und Benutzername sowie Passwort in die dafür vorgesehenen Felder eintragen. Als nächstes kann die Benutzeroberfläche der I²C oder LED Steuerung geöffnet werden. Sobald einer der Tasten angeklickt wird, überprüft das Programm als erstes den Benutzernamen und das Passwort. Wenn die Eingaben korrekt sind, wird das gewünschte Fenster geöffnet. Andernfalls bekommt der Benutzer eine Fehlermeldung (siehe *Abbildung 33*).

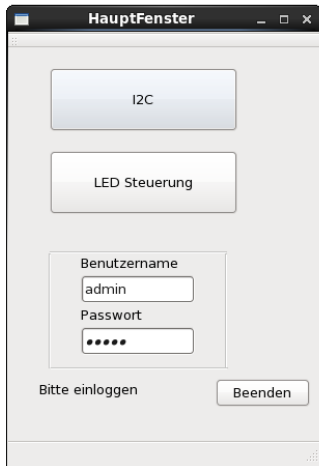


Abbildung 34: Startseite



Abbildung 33: Fehlermeldung

7.4 Control Panel für die Steuerung der LEDs

Die erste Etappe des Projektes vor der eigentlichen Inbetriebnahme der I²C Schnittstelle bestand darin eine GUI für die Steuerung der LEDs unter Verwendung des FTDI Moduls in der Entwicklungsumgebung Qt-Creator zu programmieren. Dadurch sollte nachgewiesen werden, dass die Verbindung zu dem Modul steht und dass das Modul auch korrekt angesprochen wird. In Abbildung 35 ist die GUI abgebildet. Die jeweiligen Funktionsbereiche sind mit Rot markiert und umrandet. In den kommenden Kapiteln werden die Funktionen der Testumgebung näher betrachtet und erklärt.

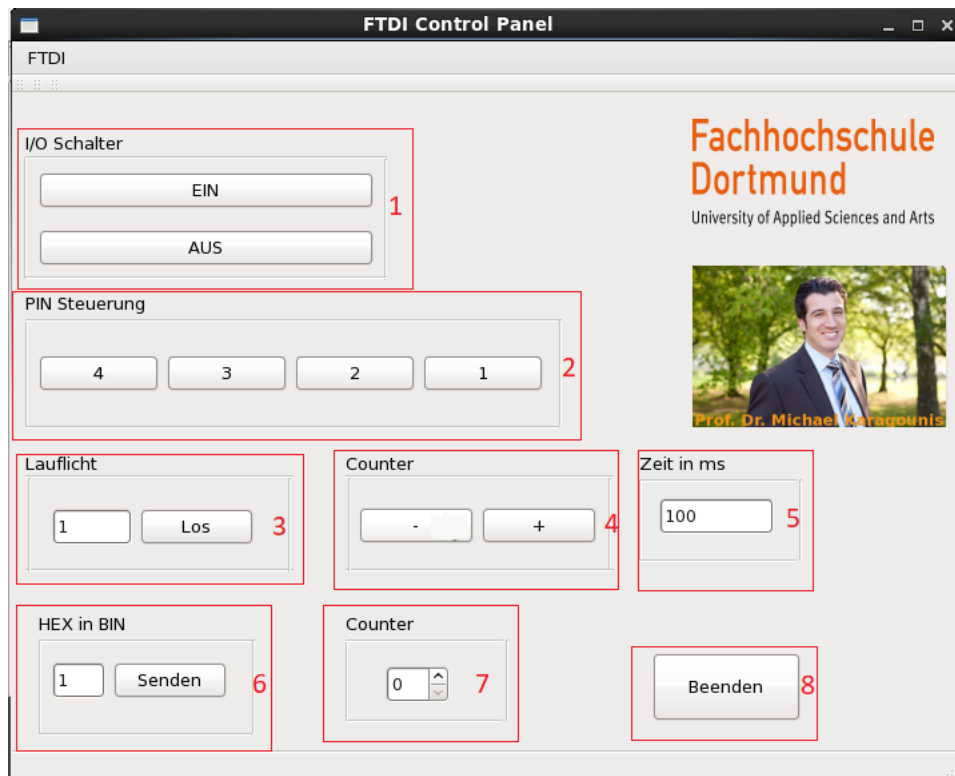


Abbildung 35: FTDI Control Panel

7.4.1 Test Button erstellen

In Kapitel 3.1 wurde bereits beschrieben, wie das Projekt angelegt wird. Daher wird dieser Teil übersprungen. Beachtet werden sollte allerdings, dass am Anfang bzw. bei der Anlegung des Projektes anstatt *Qt Console Application* die Anwendung *Qt Quick Application* ausgewählt wird. Nachdem das Projekt erstellt wurde, werden wieder die Bibliotheken und die benötigten Dateien von *Treiber-Releases* in den neu erstellten Ordner kopiert. Nun kann unter *Mainwindow.ui* die GUI frei gestaltet werden. Links im Menü (siehe Abbildung 36) können verschiedene Funktionen mit der Maus ausgewählt und per Drag&Drop der GUI hinzugefügt werden.

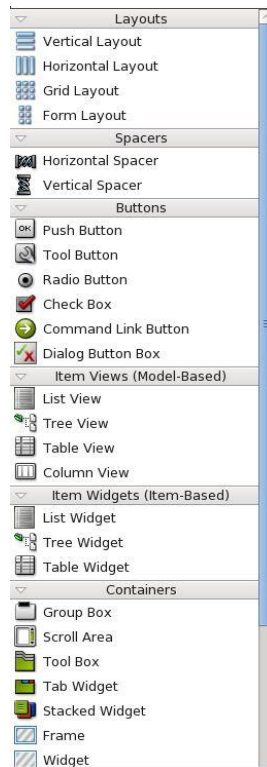


Abbildung 36: Menü Qt Creator

Um beispielsweise einen Button hinzuzufügen, wird unter der Kategorie „*Button*“ der *PushButton* gewählt und in die GUI hinübergezogen. Nun kann der Name, die Größe und weitere Einstellungen im Menü bearbeitet werden. Der Name kann auch durch einen Doppelklick auf den Button geändert werden. In diesem Fall wird der Button *Test* genannt.

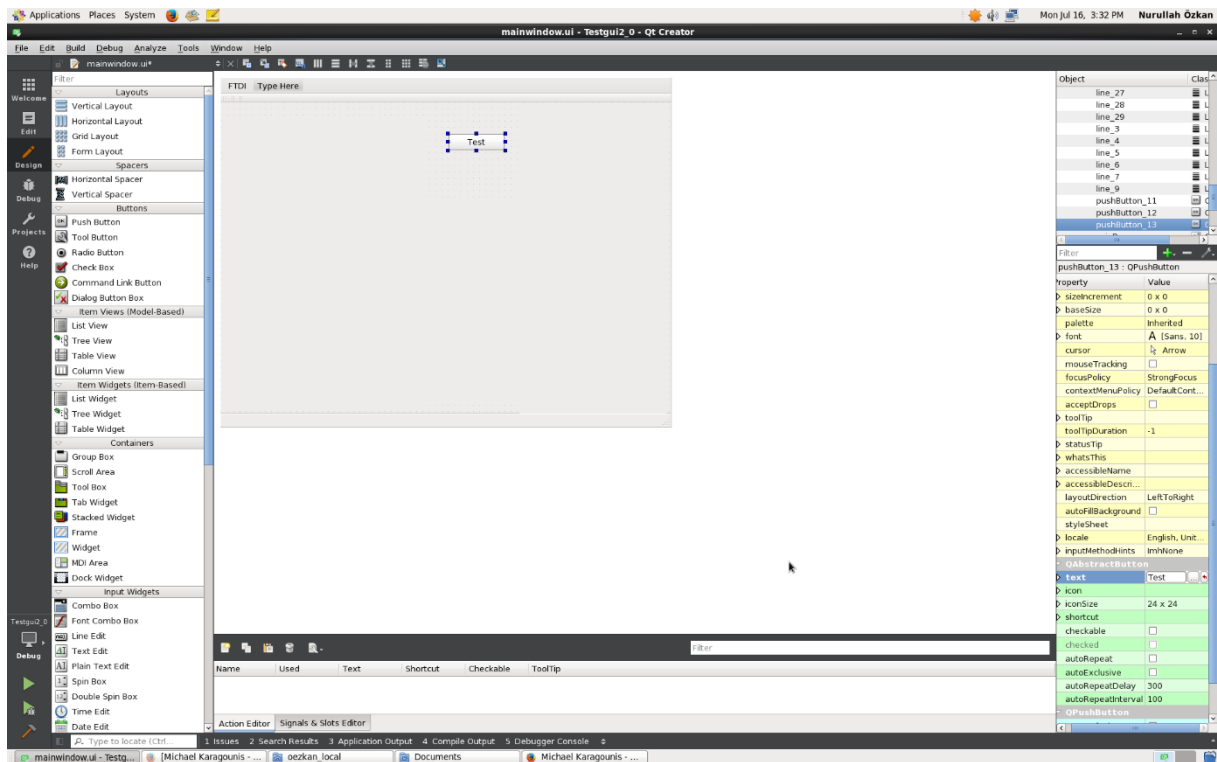


Abbildung 37: Erstellung eines Buttons

Der Programmierer kann jetzt entscheiden, was mit dem Button passieren soll bzw. welche Funktion der Button erfüllen soll (siehe Abbildung 37). Qt bietet in diesem Zusammenhang sehr viele Möglichkeiten. Um eine Funktion hinzuzufügen, wird durch einen Rechtsklick auf den betreffenden Button und anschließend über „go to slot“ die Funktion ausgewählt.

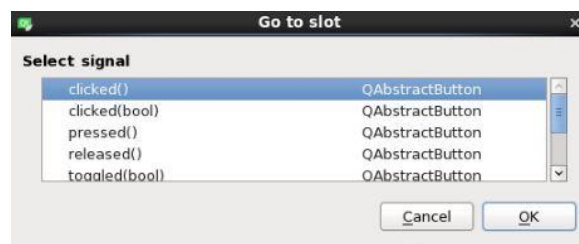


Abbildung 38: Funktion des Buttons wählen

Der Button ruft die entsprechende Funktion auf, sobald er angeklickt wird. Es öffnet sich dann in die Datei `mainwindow.cpp` wo der Programmierer den Code für die betreffende Funktion eingeben kann. Wenn der Button während der Programmlaufzeit angeklickt wird, startet die Ausführung des Codes, welcher in der entsprechenden Funktion hinterlegt worden ist.

7.4.2 Konstruktor und Destruktor

Um die Verbindung mit dem FTDI Mini-Modul nicht bei jedem Schreib-/Lesezugriff neu aufbauen bzw. trennen zu müssen, wird beim Start die Verbindung im Konstruktor der Klasse `SteuerungLed` aufgebaut. Die für die Verbindung benötigten Variablen werden als Globalvariablen deklariert. Anschließend wird die Variable `portNumber` an die Funktion `FT_Open` übergeben. In der folgenden Zeile werden alle 8 Pins als Ausgang gesetzt und der `FT_BITMODE_ASYNC_BITBANG` gestartet. Sobald das Fenster geschlossen wurde, löscht der Destruktor die Verbindung, sodass bei Bedarf wieder eine neue Verbindung aufgebaut werden kann. Hierfür wird der untere Quellcode verwendet

```
+++++
SteuerungLed::SteuerungLed(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::SteuerungLed)
{
    ftStatu = FT_Open(portNumber, &ftHandl);
    ftStatu = FT_SetBitMode(ftHandl,
                            0xFF, // sets all 8 pins as outputs
                            FT_BITMODE_SYNC_BITBANG);

    ui->setupUi(this);
}

SteuerungLed::~SteuerungLed()
{
    FT_Close(ftHandl);
    delete ui;
}
+++++
```

7.4.3 I/O Schalter

Der in der Abbildung 35 mit der Nr. 1 gekennzeichnete Bereich der Bedienoberfläche dient zur Ein- sowie Ausschaltung der einzelnen Pins und der LED. Was sich hinter den beiden Buttons verbirgt, wird unten ausführlich erläutert.

```
+++++
void SteuerungLed::on_ein_clicked()
{
    outputData = 0xFF;
    ftStatu = FT_Write(ftHandl, &outputData, 1, &bytesWritten);
}
+++++
```


+++++

Der Hex-Wert „FF“ wird der Variablen *outputData* zugewiesen. Das heißt, dass alle 8 Pins und somit alle 8 LED eingeschaltet werden. Als nächstes werden die ermittelten Werte an die Funktion *FT_Write* übergeben, welche die Daten wiederum an die variable *ftStatu* weitergibt. Genau an dieser Stelle erfolgt eine Übertragung der Daten über die bereits deklarierte Schnittstelle an das Modul. Jetzt sollten alle LEDs leuchten. Dieser Programmablauf gilt für den Button EIN und gilt auch für den Button AUS mit der Ausnahme, dass anstatt dem Hex-Wert *FF* an die Variable *outputData* der Wert *00* übergeben wird.

7.4.4 Pin Steuerung

Der in der Abbildung 35 mit der Nr. 2 gekennzeichnete Bereich der Bedienoberfläche dient zur unabhängigen Steuerung der einzelnen LEDs, welche an den Pins des Moduls angeschlossen sind. Der Quellcode für die Buttons ist dem Quellcode der Ein- und Ausschalter sehr ähnlich. Die LEDs werden von rechts beginnend nummeriert und der Reihe nach hochgezählt. Das heißt, dass die LED ganz rechts die Nummer 1 bekommt usw. Auf dem Button werden die Nummern der einzelnen LED abgebildet. Wird der Button mit der Nummer 1 angeklickt, wird eine „0x01<<4“ an die Funktion *outputData* übergeben. Es wird jeweils um vier Bits nach links geschiftet, weil die ersten vier Pins für die I²C-Verbindung benötigt werden. Es könnte auch „0x16“ übergeben werden. Bei jedem Button ist dementsprechend ein Wert hinterlegt worden, der bei einem Mausklick an die Funktion *outputData* übergeben wird.

7.4.5 Lauflicht

Der in der Abbildung 35 mit der Nr. 3 gekennzeichnete Bereich der Bedienoberfläche dient dazu, dass die LEDs der Reihe nach ein- und ausgeschaltet. Die LEDs werden von rechts nach links jeweils nacheinander mit einer gewissen Zeitverzögerung ein- und ausgeschaltet. Sobald die 4. LED eingeschaltet ist, wird die Laufrichtung gewechselt. Im linken Feld kann nun eingegeben werden, wie oft das Licht hin und her laufen soll. Die Zeitverzögerung lässt sich unter der mit der Nr. 5 gekennzeichneten Fläche in Millisekunden einstellen. Dieser Wert beeinflusst wie lange das Programm warten soll, nach dem eine LED eingeschaltet worden ist bevor die nächste LED eingeschaltet wird. Der Quellcode für das Teilprogramm wird unten aufgeführt.

```

+++++
void SteuerungLed::on_los_clicked()
{

    QString str, str2 ;
    bool ok;

    str = ui->lauflicht->text();
    int z = str.toInt(&ok, 16); // String in Hex umwandeln

    str2 = ui->zeit->text();
    int zeit = str2.toInt(&ok, 10); // String in Hex umwandeln
    zeit=zeit*1000;

    for(z;z>0;z--)
    {
        int z1=1;

        for(z1;z1<9;z1*=2)
        { // variable hochzählen
            outputData = z1<<4;
            ftStatu = FT_Write(ftHandl, &outputData, 1, &bytesWritten);
            usleep(zeit);

            if(z1>7)
            {
                for(int z2=8;z2>1;z2/=2) // variable runterzählen
                {

                    if(z2==8)
                    {

                        continue;

                    }
                    else
                    {

                        outputData = z2<<4;
                        ftStatu = FT_Write(ftHandl, &outputData, 1,
&bytesWritten);
                        usleep(zeit);

                    }

                }

            }

            outputData = 1<<4;
            ftStatu = FT_Write(ftHandl, &outputData, 1, &bytesWritten);
        }
    }
}
}
}
}
+++++

```

Da bei diesem Programmteil die Daten von einem Textfeld eingelesen werden, wird hier mit der Bibliothek *QString* gearbeitet. Das Modul kann jedoch mit einem String nichts anfangen.

Daher muss der eingelesene *String* vor der Übertragung erst in einen *Hex* Wert konvertiert werden. Wenn in dem Feld der Buchstabe „A“ eingegeben wird, interpretiert das Programm das Zeichen als Hex-Wert, und überträgt die Dezimalzahl 10. Da die LEDs mit einer gewissen Zeitverzögerung ein- und ausgeschaltet werden soll, muss auch die Zeit eingelesen und ebenfalls konvertiert werden. Als nächstes werden zwei ineinander verschachtelte *for-Schleifen* mit einer *if-Abfrage* gebildet. Die *if-Abfrage* prüft, ob die 4. LED eingeschaltet ist. Sollte dies der Fall sein, sorgt die nächste *for-Schleife* dafür, dass die Laufrichtung gewechselt wird. Die erste *for-Schleife* prüft, wie oft das Licht hin und her läuft. Dafür wird der Wert, der im Feld eingetragen wurde, eingelesen und heruntergezählt. Um die LED nacheinander einschalten zu können, müssen die Werte {1,2,4,8} übertragen werden. Die Laufrichtung wird umgekehrt in dem nach der Aktivierung der 4. LED die Werte {8,4,2,1} übertragen werden. Daher fängt die nächste Schleife mit eins beginnend an hochzuzählen. Der Wert wird nach der Übertragung mit 2 multipliziert. Die *if-Abfrage* prüft, ob der Wert die Acht erreicht hat. Sollte dies der Falls sein, wird die nächste Schleife den Wert herunterzählen, in dem der Wert jeweils durch zwei dividiert wird, bis der Wert eins erreicht wurde. Da am Ende der ersten Schleife und am Anfang der zweiten Schleife die acht jeweils vorkommt, sorgt die nächste *if-Abfrage* dafür, dass die Acht nur einmal übertragen wird. Die *for-Schleife* dividiert den Wert, bis der Wert eins erreicht wurde und berichtet das Programm dann ab. Daher wird im letzten Schritt die Eins nochmal übertragen und bleibt bei diesem Wert stehen.

7.4.6 Counter

Der in der Abbildung 35 mit der Nr. 4 gekennzeichnete Bereich der Bedienoberfläche zählt bis 16 hoch oder herunter. Der Wert wird jeweils binär übertragen. Bei diesem Teilprogramm werden die LEDs ebenfalls mit einer Verzögerungszeit geschaltet. Der Quellcode für das Teilprogramm wird unten aufgeführt.

```

+++++
void SteuerungLed::on_hoch_clicked()
{
    QString str2;
    bool ok;
    str2 = ui->zeit->text();
    int zeit = str2.toInt(&ok, 10); // String in Hex umwandeln
    zeit=zeit*1000;

    for(int i=0;i<16;i++ )
    {
        outputData = i<<4;
    }
}

```

```

    ftStatu = FT_Write(ftHandl, &outputData, 1, &bytesWritten);
    usleep(zeit);
}

}

+++++
```

Die *for-Schleife* zählt den Wert hoch bis der Wert 15 erreicht ist. Beim Wert 16 wird die Schleife beendet. Während des Zählvorgangs wird der Wert jeweils übertragen. Beim Herunterzählen wird innerhalb der *for-Schleife* der Wert mit 15 beginnend dekrementiert. Der Wert wird ebenfalls jeweils übertragen.

7.4.7 HEX/BIN Konverter

Der in der Abbildung 35 mit der Nr. 6 gekennzeichnete Bereich der Bedienoberfläche liest zunächst den Wert als String ein und interpretiert ihn als Hex-Wert. Der entsprechende Wert wird dann in binärer Darstellung durch die LEDs angezeigt. Der Quellcode für das Teilprogramm wird unten aufgeführt.

```

+++++
void SteuerungLed::on_hexbin_2_clicked()
{
    QString str ;
    bool ok;
    str = ui->hexbin->text();
    int hex = str.toInt(&ok, 16); // String in Integer umwandeln
    outputData = hex<<4;
    ftStatu = FT_Write(ftHandl, &outputData, 1, &bytesWritten);
}

+++++
```

Nachdem der zu konvertierende Wert im Feld eingetragen wurde, kann die Eingabe bestätigt werden, indem der Button angeklickt wird oder die Taste „Enter“ betätigt wird. Im Qt-Creator wird daher der Quellcode auch in die Funktion „*on_enter_clicked()*“ kopiert. Beide Funktionen besitzen den gleichen Code und unterscheiden sich nur in Bezug auf die Aufrufmethode.

7.4.8 Counter / SpinBox

Der in der Abbildung 35 mit der Nr. 7 gekennzeichnete Bereich der Bedienoberfläche kann eingelesene Dezimalwerte übertragen und hoch- oder runterzählen. Der Quellcode für das Teilprogramm wird unten aufgeführt.

```
+++++
void SteuerungLed::on_spinBox_valueChanged(int arg1)
{
    int inhalt;
    inhalt=ui->spinBox->value();

    outputData = inhalt<<4;
    ftStatu = FT_Write(ftHandl, &outputData, 1, &bytesWritten);
}
+++++
```

Zunächst wird der Wert eingelesen, sobald der Wert im Feld geändert wurde. Anschließend wird der Wert übertragen und binär dargestellt. Es kann auch von dem eingegeben Wert beginnend, zwischen 0 – 15 mit den Pfeiltasten hoch oder heruntergezählt werden.

7.4.9 Beenden

Der in der Abbildung 35 mit der Nr. 8 gekennzeichnete Bereich der Bedienoberfläche schließt das Fenster und beendet somit das Programm. Der Quellcode für das Teilprogramm wird unten aufgeführt.

```
+++++
void SteuerungLed::on_beenden_clicked()
{
    this->hide();
}
+++++
```

7.4.10 Bild Einfügen

Um in die GUI ein Bild integrieren zu können, wird unter *Display Widgets* die Funktion *QLabel* in der GUI platziert. Anschließend wird das Textfeld entfernt und die Größe optimiert. Als

nächstes wird der Name des *QLabels* geändert. In diesem Fall wird der Name auf „*label_pic*“ gesetzt. Um ein Bild einfügen, zu können, wird die Datei „*mainwindow.cpp*“ geöffnet. Hier wird erst mal die Bibliothek „*QDesktopServices*“ eingebunden. Es wird in der Funktion „**MainWindow::MainWindow(QWidget *parent)**“ festgelegt, wo das Bild gespeichert ist und wie die Skalierung sein soll. Zunächst wird mit „**QPixmap pix("/home/local/ ");**“ der Speicherort des Bildes angegeben und der Variable „*pix*“ zugewiesen. Nun wird das Bild an die „*label_pic*“ zugewiesen. Mit „**ui->label_pic->setScaledContents(true);**“ wird das Bild automatisch skaliert. [14]

7.5 Control Panel für die I²C-Kommunikation

Das Hauptziel dieses Projektes besteht darin, eine GUI für die Steuerung der I2C-Kommunikation auf dem FTDI Board, in der Entwicklungsumgebung Qt-Creator, zu programmieren.

Um den Quellcode besser nachvollziehen zu können, wird der Programmablauf bzw. die Vorgehensweise Schritt für Schritt erklärt. Der vollständige Quellcode ist im Anhang wieder zu finden.

```

+++++
SteuerungI2C::SteuerungI2C(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::SteuerungI2C)
{
    DWORD devIndex = 0;
    char Buf[64];
    ftStatus = FT_ListDevices((PVOID)devIndex, &Buf,
FT_LIST_BY_INDEX|FT_OPEN_BY_SERIAL_NUMBER);
    ftStatus = FT_Open(0, &ftHandle); //Modul öffnen

    ui->setupUi(this);

    QPixmap pix("/home/oezkan_local/Downloads/fhdo.png"); //Speicherort
    ui->label_pic->setPixmap(pix);
    ui->label_pic->setScaledContents(true); //skalierung
}

SteuerungI2C::~SteuerungI2C()
{
    FT_Close(ftHandle); //verbindung wird abgebrochen
    delete ui;
}
+++++

```

Als nächstes gibt es fünf Felder, in denen Werte eingefügt werden können. Was genau in diese Felder eingetragen wird und die jeweiligen Funktionen werden in den kommenden Abschnitten näher betrachtet. Rechts gibt es noch ein Statusdisplay, mit dessen Hilfe der Benutzer über aktuelle Prozesse informiert wird. Die Verbindung wird erst abgebrochen, wenn die „Beenden“ Taste angeklickt oder das Fenster geschlossen wird.

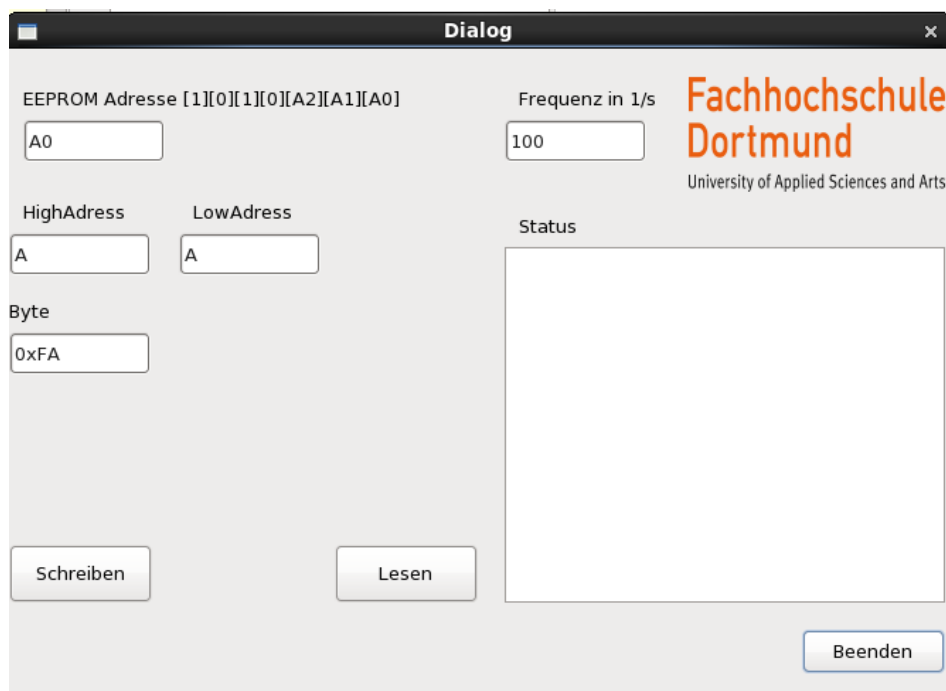


Abbildung 39: I2C-Kommunikation

7.6 Schreiben

Sobald die Taste „Schreiben“ angeklickt wird ruft das Programm die dazugehörige Funktion auf. Die Funktion heißt „`on_schreiben_clicked()`“. Als nächstes wird von dieser Funktion aus, eine weitere Funktion aufgerufen. Diese Funktion `Eingabenauswerten()` soll die Eingabefelder auslesen bzw. auswerten.

```

+++++
void SteuerungI2C::on_schreiben_clicked()
{
    Eingabenauswerten();
}

void SteuerungI2C::Eingabenauswerten()
{

```

```

scltohex = ui->scl->text();
scltosend = scltohex.toInt(&ok, 10); // String in Hex umwandeln
dwClockDivisor=Frequenzrechner(scltosend);
+++++

```

Zunächst wird das Feld „Frequenz in 1/s“ eingelesen und anschließend vom Typ String in eine Integer-Variable bzw. in eine Dezimalzahl konvertiert. Nun kann die SCL-Frequenz berechnet werden. Dafür wird die Funktion „**Frequenzrechner**“ aufgerufen bzw. der Wert (*scltosend*) an die Funktion **Frequenzrechner** übergeben.

```

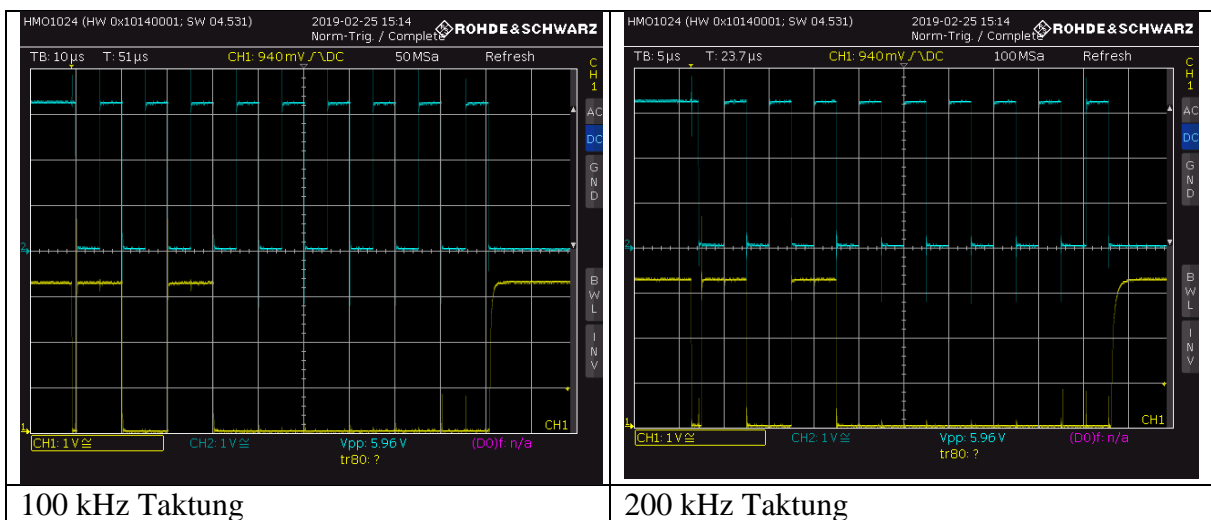
+++++

int Frequenzrechner(int f)
{
    f=((60*1000000)/(f*1000*2))-1;
    //f=f*2/3; //bei dreiphasentaktung !!!MPSSE dementsprechend auf 8C
    return f;
}

+++++

```

Die Formel für die Frequenz ist in Kapitel 6.1.2 bereits hergeleitet worden. Nun wird das Ergebnis der Rechnung zurückgegeben. Der zurückgegebene Wert wird anschließend in der Globalvariable „dwClockdivisor“ gespeichert (Siehe Tabelle 5).



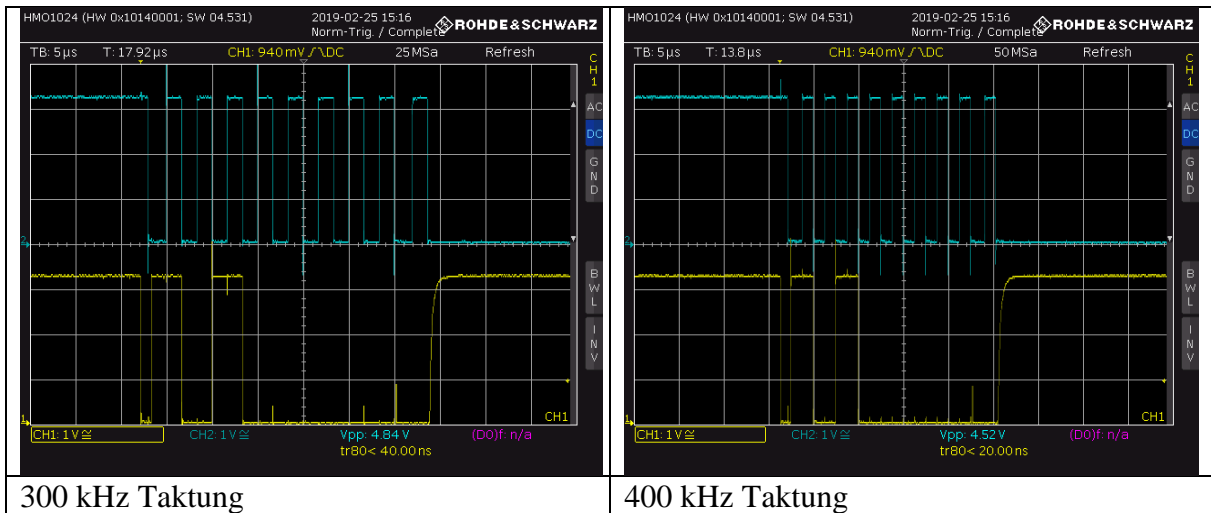


Tabelle 5: SCL-Taktung

+++++

```

eepromtohex = ui->eeprom->text();
eepromadress = eepromtohex.toInt(&ok, 16); // String in Hex
hadresstohehex = ui->highadress->text();
highadress = hadresstohehex.toInt(&ok, 16);
adresstohehex = ui->adresse->text();
sadresse = adresstohehex.toInt(&ok, 16);
bytetohehex = ui->byteadress->text();
bytetosend = bytetohehex.toInt(&ok, 16);

```

```

ByteAddressHigh=highadress;
ByteAddressLow = sadresse;
ByteDataToBeSend=bytetosend;
EepromAdresse=eepromadress;
check2=1;
zahl++;

```

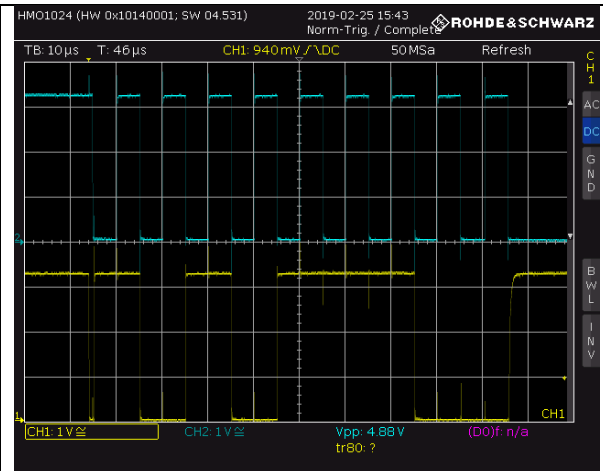
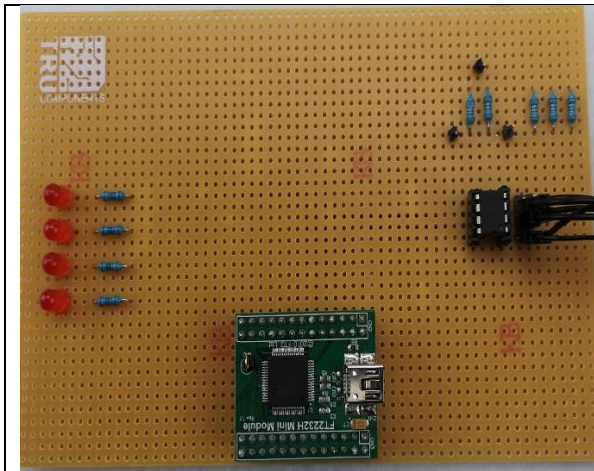
```

MPSSE();
}

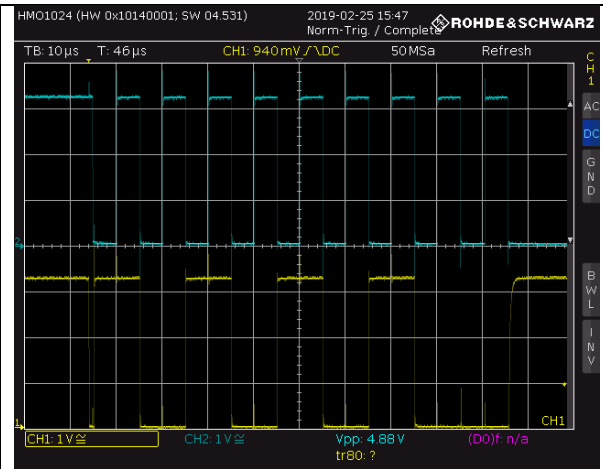
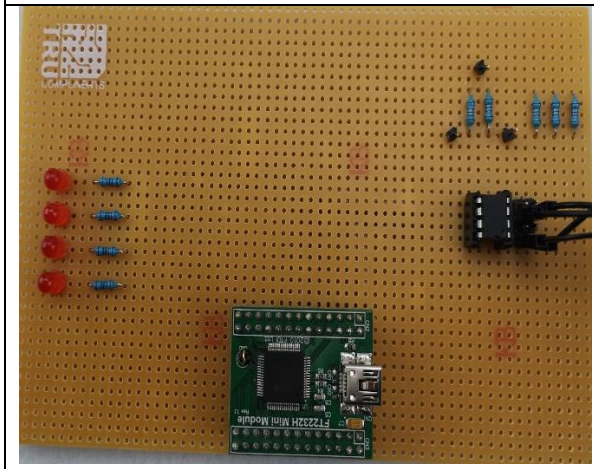
```

+++++

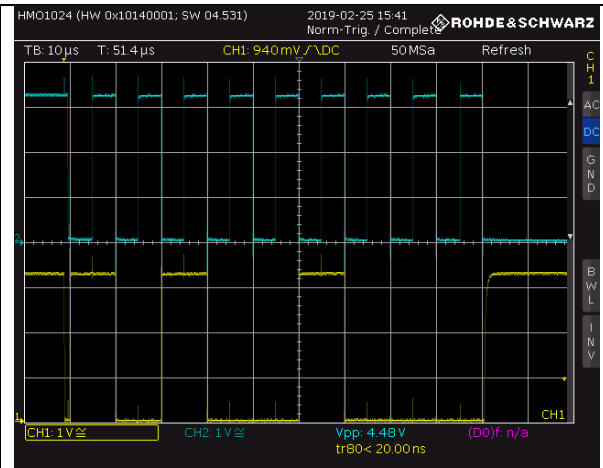
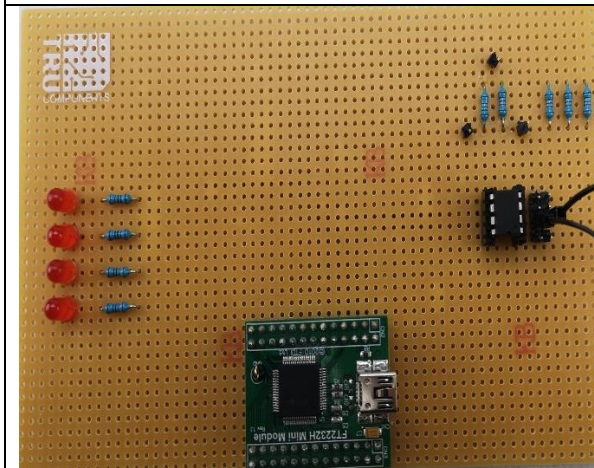
Jetzt wird der Reihe nach, die Adresse des EEPROMs, Adresshighbyte, Adresslowbyte und das Datenbyte ermittelt, in dem die aus der GUI erhaltenen String Variablen als Hex-Wert interpretiert werden und in Integer konvertiert werden. Anschließend werden alle Werte in den globalen Variablen gespeichert. Zum Schluss wird die Funktion „MPSSE“ aufgerufen. Durch Setzen der vorgesehenen Jumper kann die jeweilige EEPROM Adresse eingestellt werden. In der folgenden Tabelle werden ein paar Beispiele vorgeführt.



Adresse 0xAE



Adresse 0xAA



Adresse 0xA4

Tabelle 6: EEPROM Adresse einstellen

```

+++++
void SteuerungI2C:: MPSSE(void)
{
    DWORD dwCount;

if (ftStatus != FT_OK)
{
    printf("\n\nCan't open FT232H device! \n");
}
}

```

```

ui->display->setTextColor(Qt::red);
ui->display->insertPlainText("Can't open FT2232H device! \n");
QMessageBox::information(this,"ERROR","Can't open FT2232H device! \n");

return;
}
+++++

```

In der Funktion „MPSSE()“ wird zuerst überprüft, ob „FT_OK“ den Wert „TRUE“ besitzt. Wenn das nicht der Fall ist, also das Modul nicht geöffnet werden kann, wird eine Fehlermeldung generiert. Der Benutzer wird im Statusdisplay darüber informiert, dass das Modul nicht geöffnet werden konnte (siehe Abbildung 40).

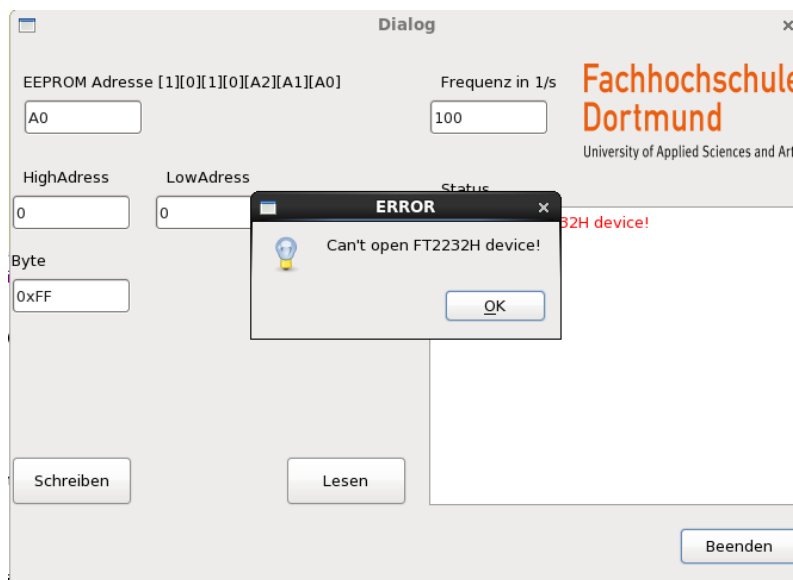


Abbildung 40: Fehlermeldung- Can't open FT2232H

```

+++++

else
{
    // Port opened successfully

    if (ftStatus == FT_OK&&check1==1)
    {
        ui->display->setTextColor(Qt::green);
        printf("\n\n\nSuccessfully open FT2232H device! \n");
        ui->display->insertPlainText("Successfully open FT2232H device! \n");

        check1=0;
    }
}
+++++

```

Liefert „FT_OK“ den Wert „TRUE“, bedeutet dies, dass das Modul erfolgreich geöffnet wurde. Im Statusdisplay wird der Benutzer über den Vorgang informiert.

```

+++++
ftStatus |= FT_ResetDevice(ftHandle); //Reset USB device
ftStatus |= FT_GetQueueStatus(ftHandle, &dwNumInputBuffer);
if ((ftStatus == FT_OK) && (dwNumInputBuffer > 0))
FT_Read(ftHandle, &InputBuffer, dwNumInputBuffer, &dwNumBytesRead);
ftStatus |= FT_SetUSBParameters(ftHandle, 65536, 65535);
ftStatus |= FT_SetChars(ftHandle, false, 0, false, 0
ftStatus |= FT_SetTimeouts(ftHandle, 0, 5000);
ftStatus |= FT_SetLatencyTimer(ftHandle, 16);
ftStatus |= FT_SetBitMode(ftHandle, 0x0, 0x00); //Reset
controller
ftStatus |= FT_SetBitMode(ftHandle, 0x0, 0x02);

if (ftStatus != FT_OK)
{
printf("fail on initialize FT2232H device 1 ! \n");
return;
}
usleep(50);

```

```

+++++

```

Jetzt werden der Reihe nach die für die Initialisierung erforderlichen Methoden aufgerufen. Was die einzelnen Methoden bewirken wurde bereits in Kapitel 7.1 erläutert. Deswegen wird hier nicht näher drauf eingegangen. Am Ende wird überprüft, ob der Vorgang erfolgreich abgeschlossen wurde. Sollte dies nicht der Fall sein, bekommt der Benutzer eine Fehlermeldung und das Programm wird beendet.

```

+++++

```

```

////////////////////////////////////
// Synchronize the MPSSE interface by sending bad command ;@0xAA;
////////////////////////////////////
OutputBuffer[dwNumBytesToSend++] = 0xAA; //Add BAD command
;@0xAA;
ftStatus = FT_Write(ftHandle, OutputBuffer, dwNumBytesToSend,
&dwNumBytesSent); // Send off the BAD commands
dwNumBytesToSend = 0; //Clear output buffer
do{
ftStatus = FT_GetQueueStatus(ftHandle, &dwNumInputBuffer);
}while ((dwNumInputBuffer == 0) && (ftStatus == FT_OK));
bool bCommandEchod = false;
ftStatus = FT_Read(ftHandle, &InputBuffer, dwNumInputBuffer,
&dwNumBytesRead); //Read out the data from input buffer
for (dwCount = 0; dwCount < dwNumBytesRead - 1; dwCount++) //Check
if Bad command and echo command received
{
if ((InputBuffer[dwCount] == BYTE('\xFA')) &&
(InputBuffer[dwCount+1] == BYTE('\xAA')))
{
bCommandEchod = true;
break;
}
}
if (bCommandEchod == false)
{
ui->display->setTextColor(Qt::red);
}

```

```

        ui->display->setText("fail to synchronize MPSSE with command '0xAA'
\n");
        printf("fail to synchronize MPSSE with command '0xAA' \n");
        return; /*Error, can't receive echo command , fail to synchronize
MPSSE interface;*/
    }

    //////////////////////////////////////
    // Synchronize the MPSSE interface by sending bad command ;@0xAB;
    //////////////////////////////////////
    //dwNumBytesToSend = 0;          //Clear output buffer

    OutputBuffer[dwNumBytesToSend++] = 0xAB;    //Send BAD command ;@0xAB;
    ftStatus = FT_Write(ftHandle, OutputBuffer, dwNumBytesToSend,
&dwNumBytesSent); // Send off the BAD commands
    dwNumBytesToSend = 0;          //Clear output buffer
    do{
        ftStatus = FT_GetQueueStatus(ftHandle, &dwNumInputBuffer);
    }while ((dwNumInputBuffer == 0) && (ftStatus == FT_OK));
    bCommandEchod = false;
    ftStatus = FT_Read(ftHandle, &InputBuffer, dwNumInputBuffer,
&dwNumBytesRead); //Read out the data from input buffer
    for (dwCount = 0;dwCount < dwNumBytesRead - 1; dwCount++)
    {
        if ((InputBuffer[dwCount] == BYTE('\xFA')) &&
(InputBuffer[dwCount+1] == BYTE( '\xAB')))
        {
            bCommandEchod = true;
            break;
        }
    }
    if (bCommandEchod == false)
    {
        ui->display->setTextColor(Qt::red);
        ui->display->setText("fail to synchronize MPSSE with command '0xAB'
\n");
        printf("fail to synchronize MPSSE with command '0xAB' \n");
        return;
        /*Error, can't receive echo command , fail to synchronize MPSSE
interface;*/
    }

    if (ftStatus == FT_OK&&check==1)
    {
        printf("MPSSE synchronized with BAD command \n");

        ui->display->setTextColor(Qt::green);
        ui->display->insertPlainText("\n MPSSE synchronized with BAD command
\n");
        ui->display-
>insertPlainText("\n_____ \n\n");
        check=0;
    }
}

```

+++++

An dieser Stelle wird die MPSSE-Schnittstelle mit einem „Bad Command“ synchronisiert. Wenn ein fehlerhafter Befehl erkannt wird, gibt die MPSSE den Wert „0xFA“ zurück, gefolgt von dem Byte, das den fehlerhaften Befehl verursacht hat. In dem Fall ist es der Wert „0xAA“. Die Verwendung der Erkennung des fehlerhaften Befehls ist die empfohlene Methode zur

Bestimmung, ob die MPSSE mit dem Anwendungsprogramm synchron ist. Durch das Senden eines fehlerhaften Befehls und die Suche nach „0xFA“ kann die Anwendung feststellen, ob eine Kommunikation mit der MPSSE möglich ist. Aus diesem Grund wird das Byte „0xAA“ geschrieben. Anschließend wird der Output Buffer gelöscht.

Zunächst wird der Input Buffer solange überprüft, bis alle Bits empfangen wurden. Dann wird der Buffer gelesen. Nun wird überprüft, ob das erstempfangene Byte dem Wert „0xFA“ und das zweitempfangene Byte dem Wert „0xAA“ entspricht. Wenn dies der Fall ist, so ist die Synchronisierung erfolgreich gewesen. Ansonsten wird das Programm eine Fehlermeldung ausgeben und den Benutzer im Statusdisplay über den Vorgang informieren. Für die Synchronisation reicht es in der Regel, wenn nur ein „Bad Command“ gesendet wird. Allerdings empfiehlt FTDI zwei „Bad Commands“ zu senden. Daher wird zusätzlich nochmal ein „Bad Command“ gesendet und dieselbe Prozedur mit dem Wert „0xAB“ wiederholt. [8]

```

+++++
//Configure the MPSSE for I2C
//MPSSE dementsprechend auf 8C

OutputBuffer[dwNumBytesToSend++] = 0x8A;
OutputBuffer[dwNumBytesToSend++] = 0x97;
OutputBuffer[dwNumBytesToSend++] = 0x8D; //MPSSE dementsprechend auf 8C

ftStatus = FT_Write(ftHandle, OutputBuffer, dwNumBytesToSend,
&dwNumBytesSent); // Send off the commands

OutputBuffer[dwNumBytesToSend++] = 0x86;
OutputBuffer[dwNumBytesToSend++] = dwClockDivisor & 0xFF;
OutputBuffer[dwNumBytesToSend++] = (dwClockDivisor >> 8) & 0xFF;
ftStatus = FT_Write(ftHandle, OutputBuffer, dwNumBytesToSend,
&dwNumBytesSent); // Send off the commands
dwNumBytesToSend = 0; //Clear output buffer

OutputBuffer[dwNumBytesToSend++] = 0x80; //Command to set
directions of lower 8 pins and force value on bits set as output
OutputBuffer[dwNumBytesToSend++] = '\x03';
OutputBuffer[dwNumBytesToSend++] = '\x13';

usleep(20); //Delay for a while

//Turn off loop back in case
OutputBuffer[dwNumBytesToSend++] = 0x85;
ftStatus = FT_Write(ftHandle, OutputBuffer, dwNumBytesToSend,
&dwNumBytesSent); // Send off the commands
dwNumBytesToSend = 0; //Clear output buffer
usleep(30); //Delay for a while
}}
+++++

```

Als nächstes wird die MPSSE konfiguriert. Die erste an das Gerät gesendete Einstellung ist die Deaktivierung des internen 5-Teiler-Takteilers, sodass die MPSSE einen 60-MHz-Takt von der internen Taktschaltung des *FT232H* empfängt. Danach wird die adaptive Taktung deaktiviert. Anschließend wird die Dreiphasentaktung ebenfalls deaktiviert. In der nächsten Zeile werden die Befehle, die sich in im Outbuffer befinden zum Gerät gesendet. Nach dem der Output Buffer gelöscht wurde, folgt der Befehl zum Setzen des Clockdivisors. Da der Outputbuffer maximal ein Byte aufnehmen kann und der Clockdivisor aus 2 Bytes bestehen kann, wird das Datenwort in zwei Teile geteilt und an den Output Buffer übergeben. Es wird also erst das HIGH-Byte und dann das LOW-Byte gesendet. Jetzt werden die Befehle geschrieben und der Output Buffer entleert. Mit dem nächsten Befehl wird der Port (ADBUS) ausgewählt. Dann werden die SCL- und DO-Leitung mit den Anfangszustand „1“ als Output gesetzt. Für eine ausführliche Erklärung der Konfiguration der Pins ist Kapitel 6.1.3 nachzuschlagen. Zum Schluss muss der „Loopback“ deaktiviert werden. Wenn Loopback aktiviert ist, sind die DI / DO- und DO / DI-Pins intern verbunden, um die Datenübertragung ohne externes Gerät zu testen. [8]

```

+++++
        HighSpeedSetI2CStart();

void HighSpeedSetI2CStart(void)
{
    DWORD dwCount;
    for(dwCount=0; dwCount < 4; dwCount++)
    {
        OutputBuffer[dwNumBytesToSend++] = 0x80;
        OutputBuffer[dwNumBytesToSend++] = '\x03';
        OutputBuffer[dwNumBytesToSend++] = '\x13';
    }

    for(dwCount=0; dwCount < 4; dwCount++)
    {
        OutputBuffer[dwNumBytesToSend++] = 0x80;
        OutputBuffer[dwNumBytesToSend++] = '\x01';
        OutputBuffer[dwNumBytesToSend++] = '\x13';
    }

    OutputBuffer[dwNumBytesToSend++] = 0x80;
    OutputBuffer[dwNumBytesToSend++] = '\x08';
    OutputBuffer[dwNumBytesToSend++] = '\x0B';
}

+++++

```

Jetzt wird die Funktion „`HighSpeedSetI2CStart()`“ aufgerufen. Mit der Funktion wird an der SCL- und SDA-Leitung die Startbedingung erzeugt. Die Befehle werden viermal

wiederholt, um sicherzustellen, dass die minimale Dauer der Start-Setup-Zeit d.h. 600 Nanosekunden, erreicht wird. Als erstes sind die SCL- und SDA-Leitungen auf HIGH-Pegel. Anschließend geht SDA-Leitung runter, während die SCL-Leitung oben bleibt. Danach geht auch die SCL-Leitung runter. Sodass die Startbedingung erzeugt wird. Die Startbedingung kann anhand eines Oszilloskops überprüft werden (siehe Abbildung 41).

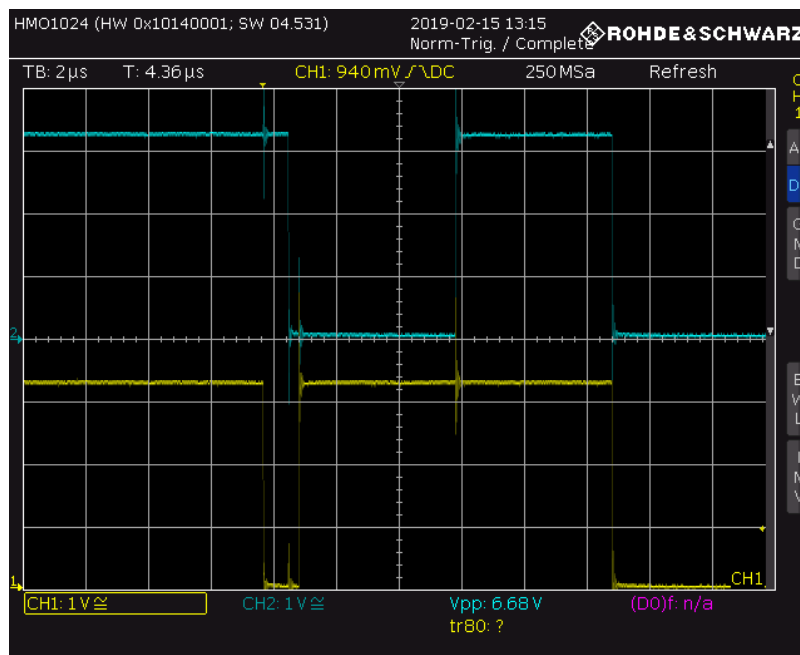


Abbildung 41: Startbedingung. SCL (Blau) SDA (Gelb)

```

+++++
    bSucceed = SendByteAndCheckACK(EepromAdresse); //Send control byte
and check the ACK bit
    bSucceed = SendByteAndCheckACK(ByteAddressHigh);
    bSucceed = SendByteAndCheckACK(ByteAddressLow);
    bSucceed = SendByteAndCheckACK(ByteDataToBeSend);
+++++

```

+++++

Nach dem die Startbedingung erzeugt wurde, können nun Datenübertragen werden. Daher wird die Funktion „SendByteAndCheckACK“ aufgerufen und jeweils die Daten, die gesendet werden sollen, an die Funktion übergeben.

+++++

```

bool SteuerungI2C:: SendByteAndCheckACK (BYTE dwDataSend)
{
    FT_STATUS ftStatus = FT_OK;

```



```

OutputBuffer[dwNumBytesToSend++] = MSB_FALLING_EDGE_CLOCK_BYTE_OUT;
//clock data byte out on "Cve Clock Edge MSB first
OutputBuffer[dwNumBytesToSend++] = '\x00';
OutputBuffer[dwNumBytesToSend++] = '\x00'; //Data length of 0x0000 means
1 byte data to clock out
OutputBuffer[dwNumBytesToSend++] = dwDataSend;

OutputBuffer[dwNumBytesToSend++] = 0x80;
OutputBuffer[dwNumBytesToSend++] = '\x00';
OutputBuffer[dwNumBytesToSend++] = '\x11';

OutputBuffer[dwNumBytesToSend++] = MSB_RISING_EDGE_CLOCK_BIT_IN;
//Command to scan in acknowledge bit , -ve clock Edge MSB first
OutputBuffer[dwNumBytesToSend++] = '\x0'; //Length of 0x0 means to scan
in 1 bit

OutputBuffer[dwNumBytesToSend++] = 0x87;
ftStatus = FT_Write(ftHandle, OutputBuffer, dwNumBytesToSend,
&dwNumBytesSent); //Send off the commands
dwNumBytesToSend = 0;
ftStatus = FT_Read(ftHandle, InputBuffer, 1, &dwNumBytesRead); //Read
one byte from device receive buffer
if ((ftStatus != FT_OK) || (dwNumBytesRead == 0))
{
    printf("fail to get ACK when send control byte 1 [Program Section]
\n");

    ui->display->setTextColor(Qt::red);
    ui->display->insertPlainText(" \n fail to get ACK when send control
byte 1 [Program Section] \n");
    QMessageBox::information(this,"ERROR","Fail to get ACK when send
control byte 1 [Program Section] \n");
    if(check2==1)
    {
        // QMessageBox::information(this,"ERROR","\n fail to get ACK when send
control byte 2 [Program Section] \n");
        check2=0;
    }
    return FALSE; //Error, can't get the ACK bit from EEPROM
}
else
{
    if (((InputBuffer[0] & BYTE('\x1')) != BYTE('\x0')) ) //Check
ACK bit 0 on data byte read out
    {
        printf("fail to get ACK when send control byte 2 [Program Section]
\n");
        ui->display->setTextColor(Qt::red);
        ui->display->insertPlainText(" \n fail to get ACK when send control
byte 1 [Program Section] \n");
        if(check2==1)
        {
            QMessageBox::information(this,"ERROR","\n fail to get ACK when send
control byte 2 [Program Section] \n");
            check2=0;
        }
        return FALSE; //Error, can't get the ACK bit from EEPROM
    }
}
OutputBuffer[dwNumBytesToSend++] = 0x80;
OutputBuffer[dwNumBytesToSend++] = '\x08';
OutputBuffer[dwNumBytesToSend++] = '\x0B';

```

```

return TRUE;
}

```

+++++

Nun wird eingestellt, dass das MSB zuerst gesendet wird. Dadurch werden Bytes auf DI / DO zwischen 1 und 65536, abhängig von den Length-Bytes ausgegeben. Eine Länge von „0x0000“ führt zu 1 Byte und eine Länge von „0xffff“ zu 65536 Byte. Die Daten werden mit dem MSB zuerst versendet. Das 7. Bit des ersten Bytes wird auf DI / DO gesetzt. Dann wird der CLK-Pin getaktet und bei der abfallenden Flanke des CLK-Pins das nächste Bit gesetzt. Es werden keine Daten in das Gerät DO / DI getaktet. [8]

Nach jedem gesendeten Byte muss ein ACK Seitens des Slaves erfolgen. Deswegen werden die SDA- und SCL-Leitung auf „0“ gesetzt. Anschließend wird die SCL-Leitung als Ausgang und die SDA-Leitung als Eingang gesetzt. Da die SDA-Leitung als Eingang geschaltet wird, entspricht das Ausgangssignal einem hochohmigen Zustand „z“. Die SDA-Leitung wird wegen des Pull-Up-Widerstands auf „1“ gezogen. An dieser Stelle wird die SDA-Leitung vom Speicherbaustein runtergezogen, um den Erhalt des empfangenen Bytes zu bestätigen. Zum Vergleich wird in der Abbildung 42 ein ACK und in der Abbildung 43 ein NACK dargestellt.

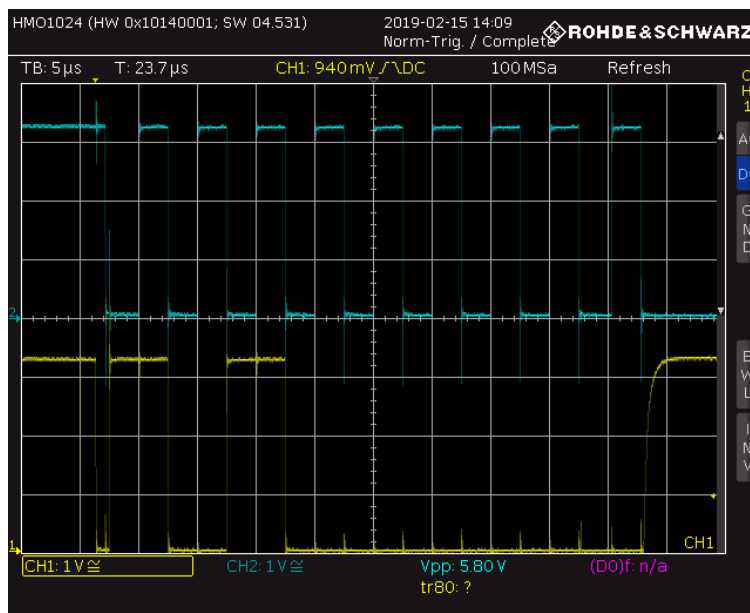


Abbildung 42: ACK-Bit (Adresse A0)

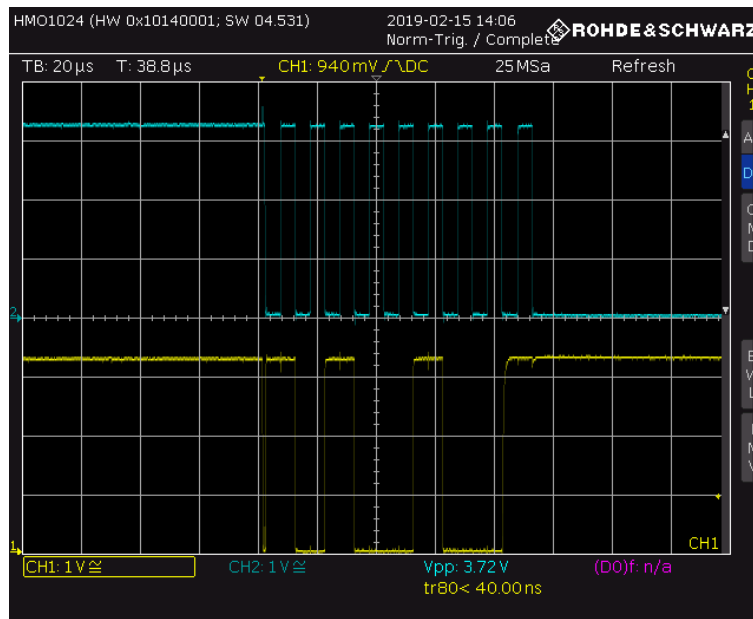


Abbildung 43: NACK-Bit (Adresse A4)

Der nächste Befehl dient dazu, dass letzte Bit also das ACK-Bit zu empfangen. Daher wird bei einer Länge von „0x00“ das 1. Bit und bei einer Länge von „0x07“ das 8. Bit gelesen. Bevor die Daten geschrieben werden, folgt noch ein letzter Befehl. Dieser Befehl sorgt dafür, dass der Inhalt des Input Buffers an das Anwendungsprogramm übergeben wird. Damit kann das Programm auswerten, ob ein ACK erzeugt wurde. Sollte dies der Fall sein wird der Datentransfer fortgesetzt. Ansonsten wird das Programm eine Fehlermeldung geben und anschließend die Kommunikation beenden (siehe Abbildung 44). Sshließlich werden die SDA- und SCL-Leitung wieder als Ausgang konfiguriert und beiden wieder der Wert „0“ zugewiesen.

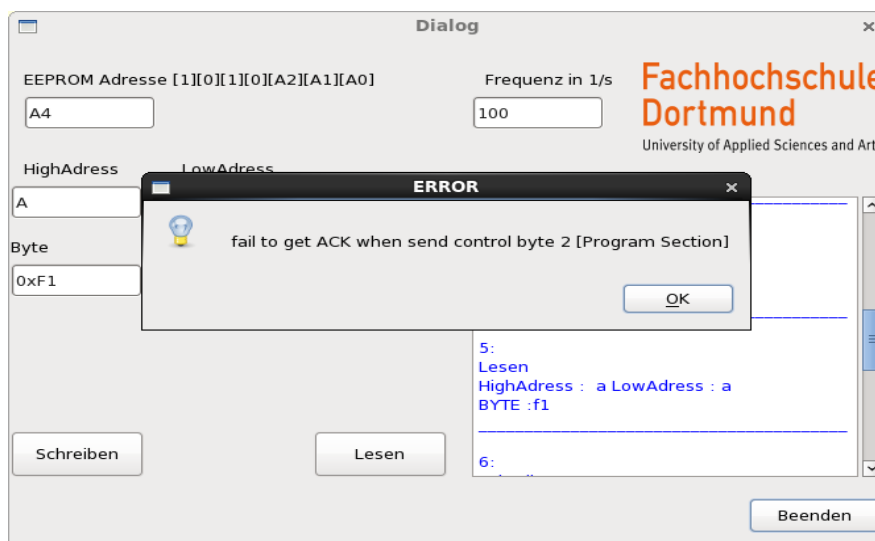


Abbildung 44: Fehlermeldung- NACK

Um den Datentransfer erfolgreich beenden zu können, wird die Funktion „HighSpeedSetI2CStop“ aufgerufen.

```
+++++
HighSpeedSetI2CStop();

void HighSpeedSetI2CStop(void)
{
    DWORD dwCount;
    for(dwCount=0; dwCount<4; dwCount++)
    {
        OutputBuffer[dwNumBytesToSend++] = 0x80;
        OutputBuffer[dwNumBytesToSend++] = '\x01';
        OutputBuffer[dwNumBytesToSend++] = '\x13'
    }
    for(dwCount=0; dwCount<4; dwCount++)
    {
        OutputBuffer[dwNumBytesToSend++] = 0x80;
        OutputBuffer[dwNumBytesToSend++] = '\x03'
        OutputBuffer[dwNumBytesToSend++] = '\x13';
    }
    //Tristate the SCL, SDA pins
    OutputBuffer[dwNumBytesToSend++] = 0x80;
    OutputBuffer[dwNumBytesToSend++] = '\x00';
    OutputBuffer[dwNumBytesToSend++] = '\x10';
}

ftStatus = FT_Write(ftHandle, OutputBuffer, dwNumBytesToSend,
&dwNumBytesSent);
dwNumBytesToSend = 0; //Clear output buffer
+++++
```

Mit der Funktion wird an der SCL- und SDA-Leitung die Stoppbedingung erzeugt. Die Befehle werden viermal wiederholt, um sicherzustellen, dass die minimale Dauer der Stopp-Setup-Zeit d.h. 600 Nanosekunden erreicht wird. Als erstes wird die SCL-Leitung auf „1“ und die SDA-Leitung auf „0“ gezogen. Als nächstes werden beide Leitungen als Ausgang festgelegt. Anschließend geht SDA-Leitung auf High-Pegel. Somit wird die Stoppbedingung erzeugt. Zum Schluss werden beide Leitungen auf Low runtergezogen und als Eingänge konfiguriert. Somit werden beide Leitungen wegen den Pull-Up-Widerständen hochgezogen. Die Stoppbedingung kann anhand eines Oszilloskops überprüft werden (siehe Abbildung 45).

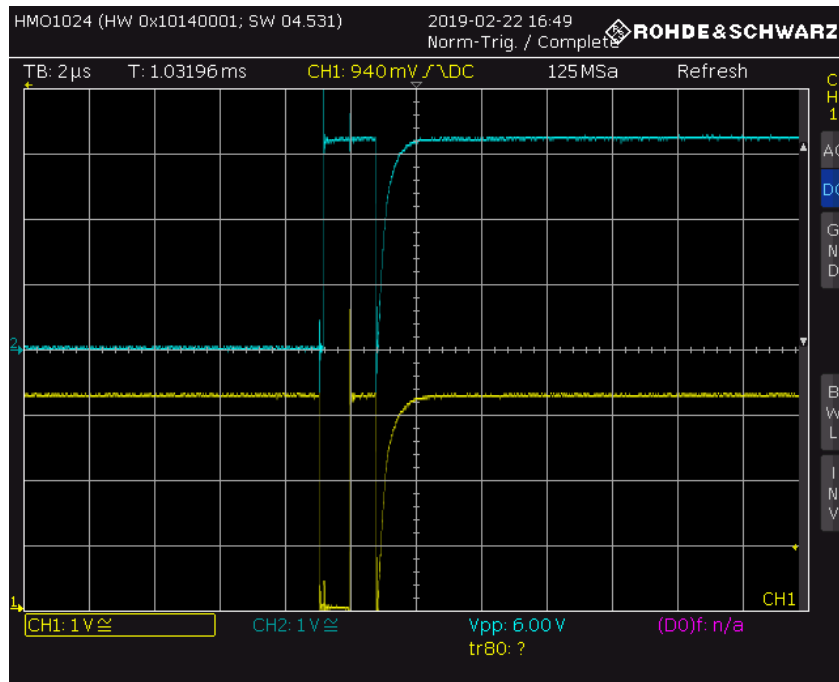


Abbildung 45: Stoppbedingung

Dadurch wird der Datentransfer beendet. Zu diesem Zeitpunkt überträgt der Speicherbaustein die empfangenen Dateien vom Zwischenspeicher zum ausgewählten Speicherort. Währenddessen darf kein neuer Transfer stattfinden. Daher wird erst 10 Millisekunden gewartet. Jetzt werden die Daten zusammengefasst und im Statusdisplay ausgegeben (siehe Abbildung 46).

```

+++++
    usleep(10000);

    qDebug() << "SCHREIBEN";
    qDebug() << "BYTE Adresse: " << ByteAddressLow << " : " << ByteDataToBeSend;

    if (check2 == 1)
    {
        QString str, str1, str2, str3;
        str = "HighAddress : " + str1.setNum(ByteAddressHigh, 16) + " LowAddress : "
+ str.setNum(ByteAddressLow, 16) + "\n";
        str2 = "BYTE : " + str2.setNum(ByteDataToBeSend, 16) + "\n";
        str3 = str3.setNum(zahl) + ": \nSchreiben \n";
        ui->display->setTextColor(Qt::blue);
        ui->display->insertPlainText(str3);
        ui->display->insertPlainText(str + str2);
        ui->display-
>insertPlainText("_____ \n\n");
    }
+++++

```

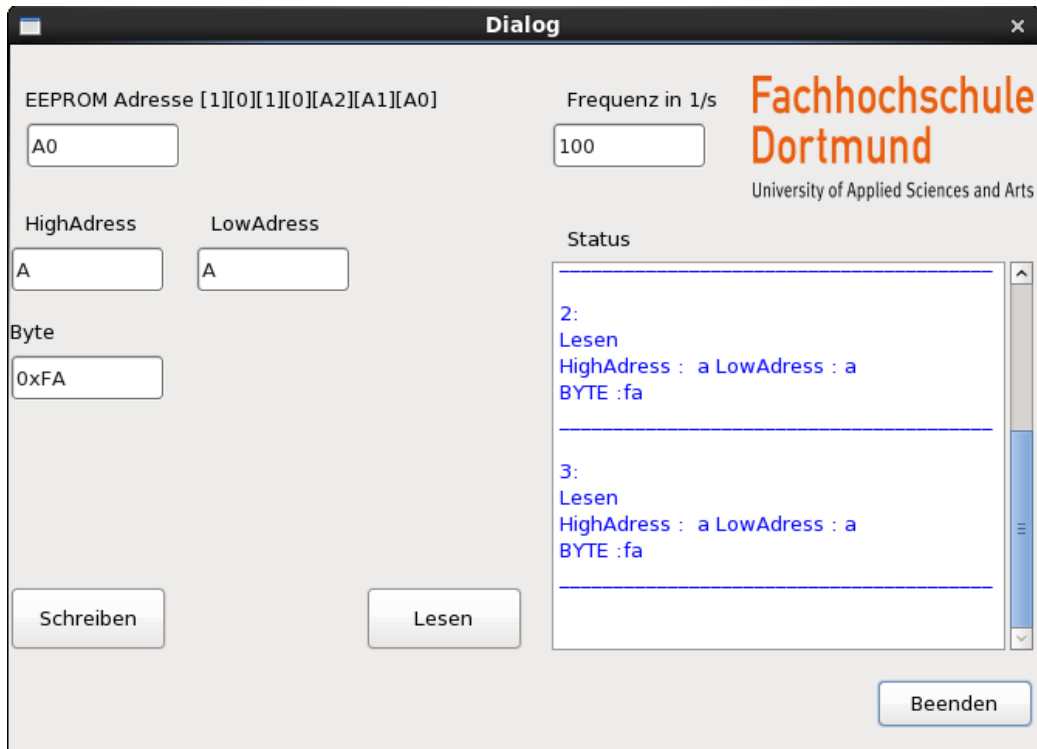


Abbildung 46: Daten übertagen.

In der folgenden Tabelle werden die Screenshots vom Oszilloskop dargestellt.

<p>Startbedingung</p>	<p>EEPROM Adresse 0xA0</p>
<p>Adress High Byte 0x0A</p>	<p>Adress Low Byte 0x0A</p>
<p>Daten Byte 0x8A</p>	<p>Stoppbedingung</p>

Tabelle 7: Screenshots (schreiben) bei 100 kHz Taktung

7.7 Lesen

Wird die Schaltfläche „Lesen“ angeklickt, wird zunächst die dazu gehörige Funktion aufgerufen. Die Funktion heißt „`on_lesen_clicked`“. Als nächstes wird von der Funktion aus, eine weitere Funktion aufgerufen. Diese Funktion soll die Eingabefelder auslesen bzw. auswerten. Da die Funktionsweise der Funktion bereits im vorherigem unter Kapitel erläutert wurde, wird hier nicht drauf eingegangen.

```
+++++
void SteuerungI2C::on_lesen_clicked()
{
    qDebug() << "Lesen Beginnt" << endl;
    Eingabenauswerten();

    ftStatus = FT_GetQueueStatus(ftHandle, &dwNumInputBuffer);
    if ((ftStatus == FT_OK) && (dwNumInputBuffer > 0))
        FT_Read(ftHandle, &InputBuffer, dwNumInputBuffer, &dwNumBytesRead);
    HighSpeedSetI2CStart();
    bSucceed = SendByteAndCheckACK(EepromAdresse);
    bSucceed = SendByteAndCheckACK(ByteAddressHigh);
    bSucceed = SendByteAndCheckACK(ByteAddressLow

    HighSpeedSetI2CStart();
    bSucceed = SendByteAndCheckACK(EepromAdresse+1);
}
+++++
```

Dazu wird die Anzahl der Bytes, die sich im Zwischenspeicher befinden ausgelesen. Anschließend werden alle Bytes gelesen, bis der Zwischenspeicher leer ist. Danach wird eine Startbedingung erzeugt. Im nächsten Schritt wird die EEPROM-Adresse sowie ByteAdressHigh und ByteAdressLow gesendet. Durch diese Vorgehensweise wurde die Speicheradresse des auszulesenden gewünschten Bytes übergeben. Nun wird eine erneute Startbedingung erzeugt und dem Speicherbaustein wird mitgeteilt, dass das FTDI Modul die Daten empfangen bzw. auslesen möchte. Nach Erhalt des ACK-Bits wird die Datenrichtung umgedreht.

```
+++++
OutputBuffer[dwNumBytesToSend++] = 0x80;
OutputBuffer[dwNumBytesToSend++] = '\x00';
OutputBuffer[dwNumBytesToSend++] = '\x11';
OutputBuffer[dwNumBytesToSend++] = MSB_FALLING_EDGE_CLOCK_BYTE_IN;
OutputBuffer[dwNumBytesToSend++] = '\x00';
+++++
```



```

OutputBuffer[dwNumBytesToSend++] = '\x00';
OutputBuffer[dwNumBytesToSend++] = MSB_RISING_EDGE_CLOCK_BIT_IN;
OutputBuffer[dwNumBytesToSend++] = '\x0';
OutputBuffer[dwNumBytesToSend++] = 0x87;
ftStatus = FT_Write(ftHandle, OutputBuffer, dwNumBytesToSend,
&dwNumBytesSent); //Send off the commands
dwNumBytesToSend = 0; //Clear output buffer
//Read two bytes from device receive
buffer, first byte is data read from EEPROM, second byte is ACK bit
ftStatus = FT_Read(ftHandle, InputBuffer, 2, &dwNumBytesRead);
ByteDataRead= InputBuffer[0]; //Return the data read from EEPROM

OutputBuffer[dwNumBytesToSend++] = 0x80;
OutputBuffer[dwNumBytesToSend++] = '\x08';
OutputBuffer[dwNumBytesToSend++] = '\x0B';
HighSpeedSetI2CStop();
ftStatus = FT_Write(ftHandle, OutputBuffer, dwNumBytesToSend,
&dwNumBytesSent);
dwNumBytesToSend = 0; //Clear output buffer

```

+++++

Ab dieser Stelle geht es genau wie bei der Funktion „SendByteAndCheckACK()“. Der einzige Unterschied besteht, dass am Ende anstatt einem Byte zwei Bytes gelesen werden.

+++++

```

usleep(10000);

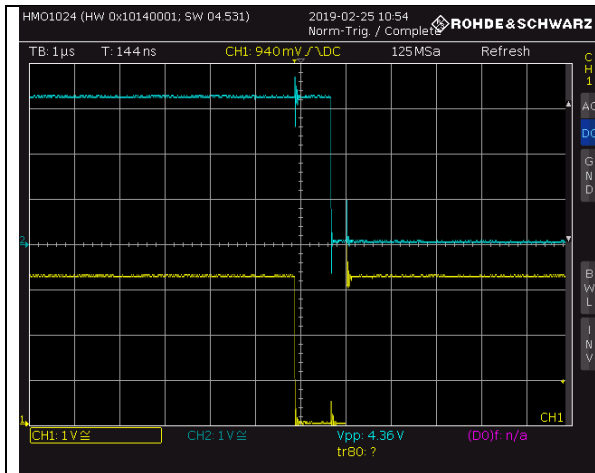
qDebug() << "LESEN";
qDebug() << "BYTE Adresse: " << ByteAddressLow << " : " << ByteDataRead;
qDebug() << "Lesen zu ende" << endl;
qDebug() << ". . ." << endl;

if (check2==1)
{
QString str, str1, str2, str3;
str = "HighAddress : " + str1.setNum(ByteAddressHigh, 16) + " LowAddress : "
+ str.setNum(ByteAddressLow, 16) + "\n";
str2 = "BYTE : " + str2.setNum(ByteDataRead, 16) + "\n";
str3 = str3.setNum(zahl) + ": \nLesen \n";
ui->display->setTextColor(Qt::blue);
ui->display->insertPlainText(str3);
ui->display->insertPlainText(str+str2);
ui->display->insertPlainText("_____ \n\n");
}

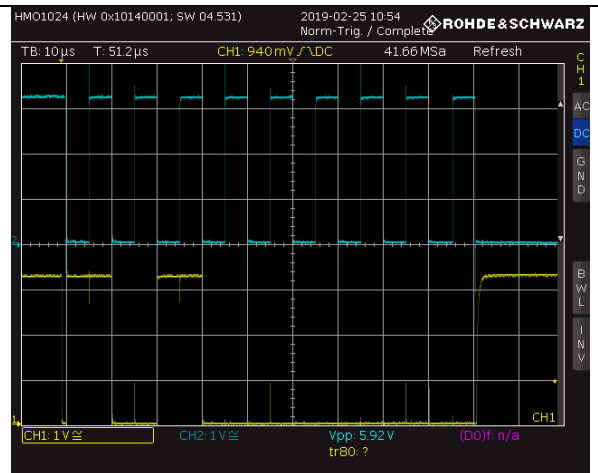
```

+++++

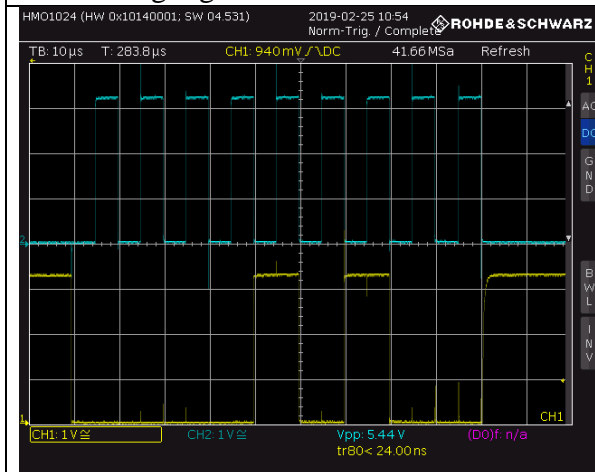
Diesmal werden die empfangenen und die gesendeten Daten zusammengefasst und im Statusdisplay ausgegeben. In der folgenden Tabelle werden die Oszilloskop Screenshots dargestellt.



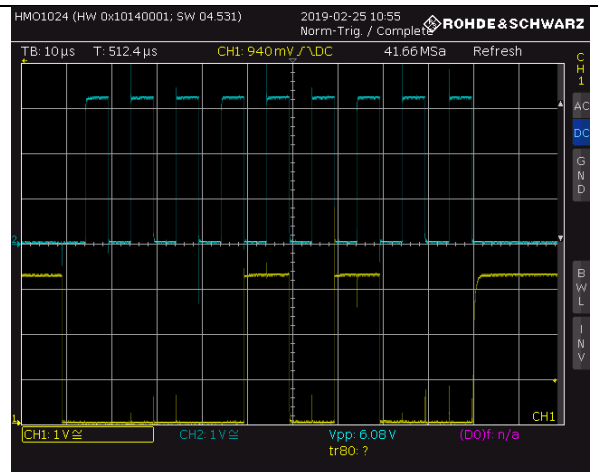
Startbedingung



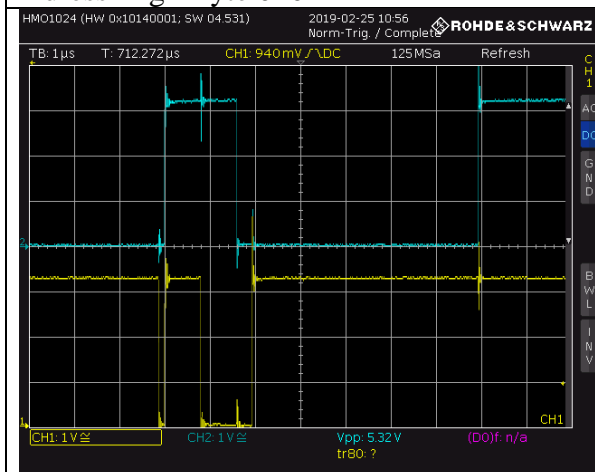
EEPROM Adresse 0xA0



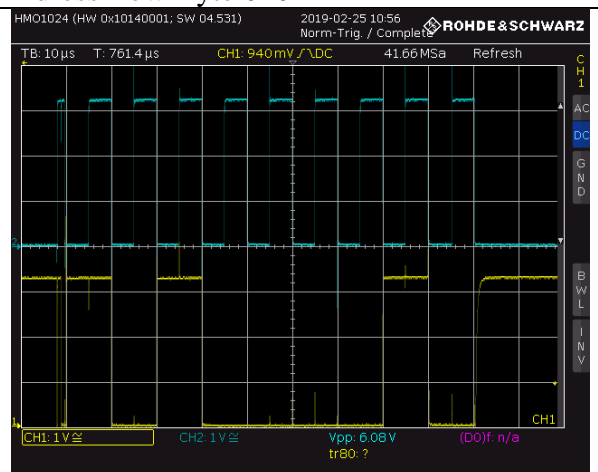
Adress High Byte 0x0A



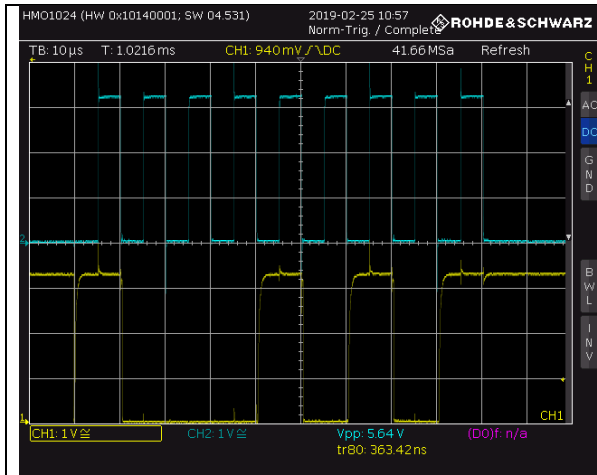
Adress Low Byte 0x0A



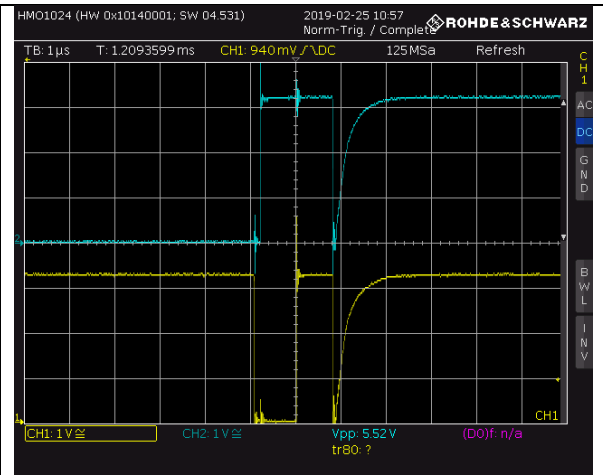
Erneut Startbedingung



EEPROM Adresse 0xA0 8. Bit Lesen



EEPROM Sendet Daten (NACK vom Master)



Stoppbedingung

Tabelle 8: Screenshots (lesen) bei 100 kHz Taktung

8 Fazit

Ziel der vorliegenden Studie war es, das FTDI Modul über einen Linux-Rechner so anzusteuern, dass es sich wie eine I²C-Schnittstelle verhält. Für den Aufbau der Verbindung sollte der mitgelieferte *D2XX* Treiber verwendet werden. Der Treiber sollte wiederum so installiert werden, dass er in der Programmierumgebung *Qt-Creator* miteingebunden werden kann. Anschließend sollte ein selbstentwickeltes Programm bzw. GUI erstellt werden, um das Modul zu steuern. Es sollte auch möglich sein, mit der entwickelten GUI eine I²C-Kommunikation zu konfigurieren. Zusätzlich sollte mittels der GUI eine Kommunikation zu einem oder mehreren Speicherbausteinen möglich sein.

Zusammenfassend lässt sich sagen, dass die im Rahmen des Projektes angesetzten Ziele erfolgreich erreicht wurden. Der Treiber wurde wie vorgesehen unter Linux installiert. Einer der schwierigsten Aufgaben war es, das Modul über die Programmierumgebung *Qt-Creator* anzusprechen. Nichtsdestotrotz wurde der Treiber erfolgreich installiert und anschließend die Schnittstelle definiert. Die für die Steuerung erforderlichen Methoden wurden sorgfältig erarbeitet.

Allerdings war der Aufbau der I²C-Schnittstelle eine Herausforderung. Dafür war eine gründliche Recherche nötig und Beispiel Programme mussten gründlich analysiert werden. Alle Fehler wurden gefunden und beseitigt. Zusätzlich wurde eine GUI entwickelt, welche die Nutzung der verschiedenen Funktionen ermöglicht. Um die erstellten Funktionen zu testen, wurde eine Platine mit 4 LEDs aufgebaut. Dadurch konnte sofort überprüft werden, in welchem Zustand sich die jeweiligen Pins befinden. Anschließend wurde die Platine weiterentwickelt, sodass mit der entwickelten GUI nun sowohl die LEDs als auch die I²C-Kommunikation gesteuert werden konnte. Mithilfe eines Oszilloskops konnte die I²C-Kommunikation überwacht bzw. es konnte direkt analysiert werden, ob die generierten Signale der I²C-Spezifikation entsprechen.

Schlussfolgernd kann gesagt werden, dass mit der vorgestellten Methodik viele weitere Projekte entwickelt werden können. Die GUI bzw. der Quellcode kann verwendet werden, um noch mehr Features zu erzielen. Es ist durchaus möglich, das Modul in weiteren Studienarbeiten weiter zu erweitern.

9 Danksagung

Ich bedanke mich herzlich bei Herrn Prof. Dr. -Ing Michael Karagounis, der mir die Möglichkeit gegeben hat, dieses großartige Projekt zu realisieren. Außerdem bedanke ich mich bei Herrn Dipl. -Ing. Rolf Paulus, der mich stets tatkräftig unterstützt und betreut hat. Zuletzt möchte ich noch einen Dank an Herrn M. Eng. Andreas Stiller aussprechen, der immer ein offenes Ohr für mich hatte.

10 Literaturverzeichnis

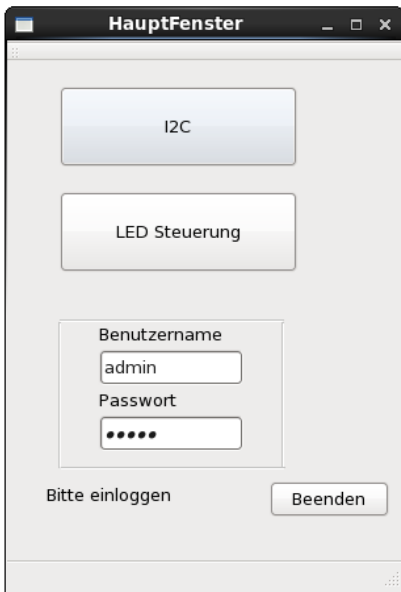
- [1] Elektor (1995). *I²C-Bus angewandt. Chips und Schaltungen* (2. Aufl.). Schaltungspraxis Aachen: Elektor-Verlag.
- [2] FTDI Chip: (2012). *FT2232H Mini Module. USB Hi-Speed FT2232H Future Evaluation Module. Datasheet*. [online PDF] [Zugriff am: 13.02.2019.] Verfügbar unter: Technology <http://www.ftdichip.com/Support/Documents/Devices> International Ltd. [DataSheets/Modules/DS_FT2232H_Mini_Module.pdf](http://www.ftdichip.com/Support/Documents/DataSheets/Modules/DS_FT2232H_Mini_Module.pdf).
- [3] FTDI Chip: (2012b). *Software Application Development D2XX Future Programmer's Guide*. [online PDF] [Zugriff am: 14.02.2019.] Verfügbar unter: Technology <http://www.ftdichip.com/Support/Documents/Devices> International Ltd [ProgramGuides/D2XX_Programmer's_Guide\(FT_000071\).pdf](http://www.ftdichip.com/Support/Documents/ProgramGuides/D2XX_Programmer's_Guide(FT_000071).pdf).
- [4] FTDI Chip: (ohne Jahr). *FT2232H. Dual High Speed USB to Multipurpose UART/FIFO*. [online PDF] [Zugriff am: 08.02.2019.] Verfügbar unter: Technology <http://www.ftdichip.com/Support/Documents/DataSheets/ICs/Devices> International Ltd. [DS_FT2232H.pdf](http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT2232H.pdf).
- [5] FTDI Chip: (2011). *D2XX Drivers*. [online] [Zugriff am: 12.02.2019.] Verfügbar unter: Technology <http://www.ftdichip.com/Drivers/D2XX.htm>. Devices International Ltd.

- [6] FTDI Chip: (2013). *Application Note AN_255 USB to I2C Example using the FT232H and FT201X devices*. [online PDF] [Zugriff am 19.02.2019] Verfügbar unter: https://www.ftdichip.com/Support/Documents/AppNotes/AN_255_USB%20to%20I2C%20Example%20using%20the%20FT232H%20and%20FT201X%20devices.pdf
- Future Technology Devices International Ltd.
- [7] FTDI Chip: (2013). *USB to I2C Example using the FT232H and FT201X devices* [online PDF] [Zugriff am 17.02.2019] Verfügbar unter: https://www.ftdichip.com/Support/Documents/AppNotes/AN_255_USB%20to%20I2C%20Example%20using%20the%20FT232H%20and%20FT201X%20devices.pdf
- Future Technology Devices International Ltd.
- [8] FTDI Chip: (2010). *AN_135 FTDI MPSSE Basics* [online PDF] [Zugriff am 18.02.2019] Verfügbar unter: https://www.ftdichip.com/Support/Documents/AppNotes/AN_135_MPSSE_Basics.pdf
- Future Technology Devices International Ltd.
- [9] FTDI Chip: (2011). *Command Processor for MPSSE and MCU Host Bus Emulation Modes* [online PDF] [Zugriff am 19.02.2019] Verfügbar unter: https://www.ftdichip.com/Support/Documents/AppNotes/AN_108_Command_Processor_for_MPSSE_and_MCU_Host_Bus_Emulation_Modes.pdf
- Future Technology Devices International Ltd.

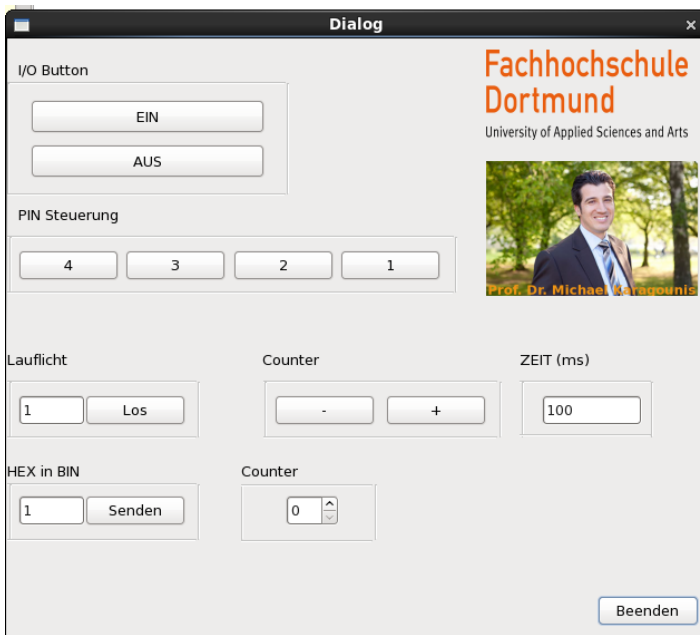
- [10] FTDI Chip: (2011). *Interfacing FT2232H Hi-Speed Devices To I2C Bus Modes* [online PDF] [Zugriff am 20.02.2019] Verfügbar unter:
 Future Technologies International Ltd. https://www.ftdichip.com/Support/Documents/AppNotes/AN_113_FTDI_Hi_Speed_USB_To_I2C_Example.pdf
- [11] Klaas, W. (2015). *Bussysteme in der Praxis*. Haar bei München: Franzis Verlag.
- [12] Leicht, R. (2005). *Das große 51er Anwendungsbuch. Vom einfachen Blinkprogramm bis zur Ansteuerung eines Graphik-LCD-Moduls über den I²C-Bus*. Poing: Franzis Verlag.
- [13] Microchip Technology Inc. (2004) *256K I2C™ CMOS Serial EEPROM* [online PDF] [Zugriff am 12.02.2019] Verfügbar unter: <http://ww1.microchip.com/downloads/en/devicedoc/21203m.pdf>
- [14] Qt Creator: Qt Documentation (ohne Jahr). [online] [Zugriff am 05.02.2019] Verfügbar unter: <http://doc.qt.io/qtcreator/creator-tutorials.html>.

11 Anhang

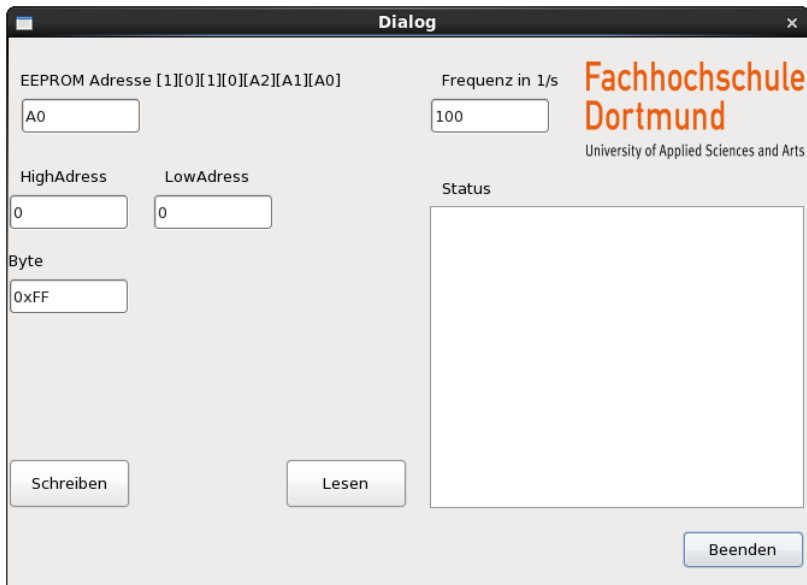
Startseite



LED Steuerung



I²C Steuerung



FTDI_I2C.pro

```
QT += core gui
QT += serialport
CONFIG += c++11
CONFIG -= app_bundle
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = FTDI_I2C
CONFIG += console
TEMPLATE = app
SOURCES += main.cpp \
    hauptfenster.cpp \
    steuerungi2c.cpp \
    steuerungled.cpp
INCLUDEPATH += ~/FTDI_I2C
HEADERS += hauptfenster.h \
    steuerungi2c.h \
    steuerungled.h \
    ftd2xx.h \
    winTypes.h \
    wintypes.h
HEADERS += \
    ~/FTDI_I2C/ftd2xx.h \
    ~/FTDI_I2C/wintypes.h \
    ftd2xx.h
FORMS += hauptfenster.ui \
    steuerungi2c.ui \
    steuerungled.ui
OTHER_FILES += ~/FTDI_I2C/build/libftd2xx.a \
    ~/FTDI_I2C/build/libftd2xx.so.1.4.6
LIBS += -L/home/oezkan_local/FTDI_I2C/build -lftd2xx
```

Hauptfenster.h

```
#ifndef HAUPTFENSTER_H
#define HAUPTFENSTER_H
#include <QMainWindow>
namespace Ui {
class HauptFenster;

```

```

}
class HauptFenster : public QMainWindow
{
    Q_OBJECT
public:
    explicit HauptFenster(QWidget *parent = 0);
    ~HauptFenster();
private slots:
    void on_i2c_clicked();
    void on_beenden_clicked();
    void on_led_clicked();
    bool einloggen();
private:
    Ui::HauptFenster *ui;
};
#endif // HAUPTFENSTER_H

```

Steuerungi2c.h

```

#ifndef STEUERUNGI2C_H
#define STEUERUNGI2C_H

#include <QDialog>
#include <ftd2xx.h>
#include <WinTypes.h>

namespace Ui {
class SteuerungI2C;
}

class SteuerungI2C : public QDialog
{
    Q_OBJECT

public:
    explicit SteuerungI2C(QWidget *parent = 0);
    ~SteuerungI2C();

private slots:
    void on_beenden_clicked();

    void on_schreiben_clicked();

    void Eingabenauswerten();

    void MPSSE();

    void on_lesen_clicked();

    bool SendByteAndCheckACK(BYTE dwDataSend);

private:
    Ui::SteuerungI2C *ui;
};

#endif // STEUERUNGI2C_H

```

Steuerungled.h

```

#ifndef STEUERUNGLED_H
#define STEUERUNGLED_H

```

```

#include <QDialog>

namespace Ui {
class SteuerungLed;
}

class SteuerungLed : public QDialog
{
    Q_OBJECT

public:
    explicit SteuerungLed(QWidget *parent = 0);
    ~SteuerungLed();

private slots:
    void on_beenden_clicked();

    void on_ein_clicked();

    void on_AUS_clicked();

    void on_eins_clicked();

    void on_zwei_clicked();

    void on_drei_clicked();

    void on_vier_clicked();

    void on_los_clicked();

    void on_hoch_clicked();

    void on_runter_clicked();

    void on_hexbin_2_clicked();

    void on_spinBox_valueChanged(int arg1);

    void on_bild_clicked();

private:
    Ui::SteuerungLed *ui;
};

#endif // STEUERUNGLED_H

```

Main.cpp

```

#include "hauptfenster.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    HauptFenster w;
    w.show();

    return a.exec();
}

```

```
}
```

Hauptfenster.cpp

```
#include "hauptfenster.h"
#include "ui_hauptfenster.h"
#include <steuerungi2c.h>
#include <steuerungled.h>
#include<QTextStream>
#include<QMessageBox>
#include<QTextBrowser>

QString name="admin",pw="admin";
int zeit=1000000;

HauptFenster::HauptFenster(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::HauptFenster)
{
    ui->setupUi(this);
    ui->status->setText("Bitte einloggen");
}

HauptFenster::~HauptFenster()
{
    delete ui;
}

bool HauptFenster::einloggen()
{
    if((ui->benutzer->text()==name) && (ui->passwort->text()==pw))
    {
        ui->status->setText("Einloggen erfolgreich");
        return true;
    }
    else
    {
        ui->status->setText("Einloggen fehlgeschlagen");
        QMessageBox::information(this,"ERROR","Benutzername oder Passwort falsch");
        return false;
    }
}

void HauptFenster::on_i2c_clicked()
{
    if(einloggen()==true)
    {
        SteuerungI2C i2c;
        this->hide();
        i2c.setModal(true);
        i2c.exec();
    }
}

void HauptFenster::on_beenden_clicked()
{

```

```

        this->hide();
    }

void HauptFenster::on_led_clicked()
{
    if(einloggen()==true)
    {
        SteuerungLed led;
        this->hide();
        led.setModal(true);
        led.exec();
    }
}

```

Steuerungi2c.cpp

```

#include "steuerungi2c.h"
#include "ui_steuerungi2c.h"

#include "ftd2xx.h"
#include "WinTypes.h"
#include <stdio.h>
#include <iostream>
#include <stdlib.h>
#include <assert.h>
#include <time.h>
#include <QDebug>
#include <unistd.h>
#include<QFile>           // text read
#include<QTextStream>
#include<QMessageBox>
#include<QTextBrowser>

const BYTE MSB_VEDGE_CLOCK_IN_BIT = '\x22';
const BYTE MSB_EDGE_CLOCK_OUT_BYTE = '\x11';
const BYTE MSB_EDGE_CLOCK_IN_BYTE = '\x24';
const BYTE MSB_FALLING_EDGE_CLOCK_BYTE_IN = '\x24';
const BYTE MSB_FALLING_EDGE_CLOCK_BYTE_OUT = '\x11';
const BYTE MSB_RISING_EDGE_CLOCK_BIT_IN = '\x22';

QString eeepromtohex, adresstohex, bytetohex, scltohex, hadresstohex ;
bool ok, check=1, check1=1, check2=1;
int scltosend, eepromadress, sadresse, bytetosend, highadress, zahl=0;

BOOL bSucceed = TRUE;
BYTE ByteAddressHigh;
BYTE ByteAddressLow; //Set read address is 0x0080 as example
BYTE ByteDataRead; //Data to be read from EEPROM
//Purge USB receive buffer first before read operation
BYTE ByteDataToBeSend;
BYTE EepromAdresse;//data programmed and read

FT_STATUS ftStatus;           //Status defined in D2XX to indicate
operation result
FT_HANDLE ftHandle;          //Handle of FT2232H device port
BYTE OutputBuffer[1024];     //Buffer to hold MPSSE commands and data to
be sent to FT2232H
BYTE InputBuffer[1024];     //Buffer to hold Data bytes to be read from
FT2232H

```

```

//DWORD dwClockDivisor = 0x004A;          //Value of clock divisor, SCL
Frequency = 60/((1+0x004A)*2) (MHz) = 400khz
DWORD dwClockDivisor;    //100khz=12B
DWORD dwNumBytesToSend = 0;    //Index of output buffer
DWORD dwNumBytesSent = 0,      dwNumBytesRead = 0, dwNumInputBuffer = 0;

SteuerungI2C::SteuerungI2C(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::SteuerungI2C)
{

    //BYTE I2CConvertDes[] = "USB Serial Converter A";    //Device port
description

    DWORD devIndex = 0;
    char Buf[64];
    ftStatus = FT_ListDevices((PVOID)devIndex,&Buf,
FT_LIST_BY_INDEX|FT_OPEN_BY_SERIAL_NUMBER);
    ftStatus = FT_Open(0,&ftHandle);

    ui->setupUi(this);

    QPixmap pix("/home/oezkan_local/Downloads/fhdo.png");
    ui->label_pic->setPixmap(pix);
    ui->label_pic->setScaledContents(true);
}

SteuerungI2C::~~SteuerungI2C()
{
    FT_Close(ftHandle);
    delete ui;
}

void HighSpeedSetI2CStart(void)
{
    DWORD dwCount;
    for(dwCount=0; dwCount < 4; dwCount++)    // Repeat commands to ensure the
minimum period of the start hold time ie 600ns is achieved
    {
        OutputBuffer[dwNumBytesToSend++] = 0x80;    //Command to set directions
of lower 8 pins and force value on bits set as output
        OutputBuffer[dwNumBytesToSend++] = '\x03';    //Set SDA, SCL high, WP
disabled by SK, DO at bit ;@1;-, GPIOL0 at bit ;@0;-
        OutputBuffer[dwNumBytesToSend++] = '\x13';    //Set SK,DO,GPIOL0 pins
as output with bit ;-1;- , other pins as input with bit ;@0;-
    }
    for(dwCount=0; dwCount < 4; dwCount++)    // Repeat commands to ensure the
minimum period of the start setup time ie 600ns is achieved
    {
        OutputBuffer[dwNumBytesToSend++] = 0x80;    //Command to set
directions of lower 8 pins and force value on bits set as output
        OutputBuffer[dwNumBytesToSend++] = '\x01';    //Set SDA low, SCL high,
WP disabled by SK at bit ;@1;- , DO, GPIOL0 at bit ;@0;-
        OutputBuffer[dwNumBytesToSend++] = '\x13';    //Set SK,DO,GPIOL0 pins
as output with bit ;-1;- , other pins as input with bit ;@0;-
    }
}

```

```

OutputBuffer[dwNumBytesToSend++] = 0x80;      //Command to set directions of
lower 8 pins and force value on bits set as output
OutputBuffer[dwNumBytesToSend++] = '\x08';    //10000011Set SDA, SCL low
high, WP disabled by SK, DO, GPIOL0 at bit ;@0;
OutputBuffer[dwNumBytesToSend++] = '\x0B';    //11111011Set SK,DO,GPIOL0
pins as output with bit ;_1;_, other pins as input with bit ;@0;
}

void HighSpeedSetI2CStop(void)
{
DWORD dwCount;
for(dwCount=0; dwCount<4; dwCount++)    // Repeat commands to ensure the
minimum period of the stop setup time ie 600ns is achieved
{
    OutputBuffer[dwNumBytesToSend++] = 0x80;    //Command to set
directions of lower 8 pins and force value on bits set as output
    OutputBuffer[dwNumBytesToSend++] = '\x01';    //Set SDA low, SCL high,
WP disabled by SK at bit ;@1;_, DO, GPIOL0 at bit ;@0;
    OutputBuffer[dwNumBytesToSend++] = '\x13';    //Set SK,DO,GPIOL0 pins
as output with bit ;_1;_, other pins as input with bit ;@0;
}
for(dwCount=0; dwCount<4; dwCount++)    // Repeat commands to ensure the
minimum period of the stop hold time ie 600ns is achieved
{
    OutputBuffer[dwNumBytesToSend++] = 0x80;    //Command to set
directions of lower 8 pins and force value on bits set as output
    OutputBuffer[dwNumBytesToSend++] = '\x03';    //Set SDA, SCL high, WP
disabled by SK, DO at bit ;@1;_, GPIOL0 at bit ;@0;
    OutputBuffer[dwNumBytesToSend++] = '\x13';    //Set SK,DO,GPIOL0 pins
as output with bit ;_1;_, other pins as input with bit ;@0;
}
//Tristate the SCL, SDA pins
OutputBuffer[dwNumBytesToSend++] = 0x80;    //Command to set directions of
lower 8 pins and force value on bits set as output
OutputBuffer[dwNumBytesToSend++] = '\x00';    //Set WP disabled by GPIOL0
at bit 0;
OutputBuffer[dwNumBytesToSend++] = '\x10';    //Set GPIOL0 pins as output
with bit ;_1;_, SK, DO and other pins as input with bit ;@0;
}

bool SteuerungI2C:: SendByteAndCheckACK(BYTE dwDataSend)
{
FT_STATUS ftStatus = FT_OK;
OutputBuffer[dwNumBytesToSend++] = MSB_FALLING_EDGE_CLOCK_BYTE_OUT;
//clock data byte out on "Cve Clock Edge MSB first
OutputBuffer[dwNumBytesToSend++] = '\x00';
OutputBuffer[dwNumBytesToSend++] = '\x00';    //Data length of 0x0000 means
1 byte data to clock out
OutputBuffer[dwNumBytesToSend++] = dwDataSend;    //Set control byte, bit
4-7 of ;@1010;_ as 24LC02 control code, bit 1-3 as block select bits
//which is don;t care here, bit 0 of ;@0;_ represent Write operation
//Get Acknowledge bit from EEPROM

OutputBuffer[dwNumBytesToSend++] = 0x80;    //Command to set directions of
lower 8 pins and force value on bits set as output
OutputBuffer[dwNumBytesToSend++] = '\x00';    //Set SCL low, WP disabled
by SK, GPIOL0 at bit ;@0;
OutputBuffer[dwNumBytesToSend++] = '\x11';    //Set SK, GPIOL0 pins as
output with bit ;_1;_, DO and other pins as input with bit ;@0;
}

```



```

OutputBuffer[dwNumBytesToSend++] = MSB_RISING_EDGE_CLOCK_BIT_IN;
//Command to scan in acknowledge bit , -ve clock Edge MSB first
OutputBuffer[dwNumBytesToSend++] = '\x0'; //Length of 0x0 means to scan
in 1 bit

OutputBuffer[dwNumBytesToSend++] = 0x87; //Send answer back immediate
command
ftStatus = FT_Write(ftHandle, OutputBuffer, dwNumBytesToSend,
&dwNumBytesSent); //Send off the commands
dwNumBytesToSend = 0; //Clear output buffer

//Check if ACK bit received, may need to read more times to get ACK bit or
fail if timeout
ftStatus = FT_Read(ftHandle, InputBuffer, 1, &dwNumBytesRead); //Read
one byte from device receive buffer
if ((ftStatus != FT_OK) || (dwNumBytesRead == 0))
{
    printf("fail to get ACK when send control byte 1 [Program Section]
\n");

    ui->display->setTextColor(Qt::red);
    ui->display->insertPlainText(" \n fail to get ACK when send control
byte 1 [Program Section] \n");
    QMessageBox::information(this,"ERROR","Fail to get ACK when send
control byte 1 [Program Section] \n");
    if(check2==1)
    {
        // QMessageBox::information(this,"ERROR","\n fail to get ACK when send
control byte 2 [Program Section] \n");
        check2=0;
    }
    return FALSE; //Error, can't get the ACK bit from EEPROM
}
else
{
    if (((InputBuffer[0] & BYTE('\x1')) != BYTE('\x0')) ) //Check
ACK bit 0 on data byte read out
    {
        printf("fail to get ACK when send control byte 2 [Program Section]
\n");
        ui->display->setTextColor(Qt::red);
        ui->display->insertPlainText(" \n fail to get ACK when send control
byte 1 [Program Section] \n");
        if(check2==1)
        {
            QMessageBox::information(this,"ERROR","\n fail to get ACK when send
control byte 2 [Program Section] \n");
            check2=0;
        }
        return FALSE; //Error, can't get the ACK bit from EEPROM
    }
}

OutputBuffer[dwNumBytesToSend++] = 0x80; //Command to set directions of
lower 8 pins and force value on bits set as output
OutputBuffer[dwNumBytesToSend++] = '\x08'; //Set SDA high, SCL low, WP
disabled by SK at bit '0', DO, GPIOL0 at bit '1'
OutputBuffer[dwNumBytesToSend++] = '\x0B'; //Set SK,DO,GPIOL0 pins as
output with bit ;_1;_, other pins as input with bit ;@0;_

return TRUE;

```

```

}

void SteuerungI2C:: MPSSE(void)
{
    DWORD dwCount;

if (ftStatus != FT_OK)
{
    printf("\n\n\nCan't open FT2232H device! \n");

    ui->display->setTextColor(Qt::red);
    ui->display->insertPlainText("Can't open FT2232H device! \n");
    QMessageBox::information(this,"ERROR","Can't open FT2232H device! \n");

    return;
}
else
{
    // Port opened successfully

    if (ftStatus == FT_OK&&check1==1)
    {
        ui->display->setTextColor(Qt::green);
        printf("\n\n\nSuccessfully open FT2232H device! \n");
        ui->display->insertPlainText("Successfully open FT2232H device! \n");

        check1=0;
    }

    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, &dwNumInputBuffer); // Get
the number of bytes in the FT2232H receive buffer
    if ((ftStatus == FT_OK) && (dwNumInputBuffer > 0))
        FT_Read(ftHandle, &InputBuffer, dwNumInputBuffer,
&dwNumBytesRead); //Read out the data from FT2232H receive buffer
    ftStatus |= FT_SetUSBParameters(ftHandle, 65536, 65535); //Set USB
request transfer size
    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 for the FT2232H
    ftStatus |= FT_SetLatencyTimer(ftHandle, 16); //Set the latency
timer
    ftStatus |= FT_SetBitMode(ftHandle, 0x0, 0x00); //Reset
controller
    ftStatus |= FT_SetBitMode(ftHandle, 0x0, 0x02); //Enable MPSSE
mode

    if (ftStatus != FT_OK)
    {
        printf("fail on initialize FT2232H device 1 ! \n");
        return;
    }
    usleep(50); // Wait for all the USB stuff to complete and work

    ////////////////////////////////////////
    // Synchronize the MPSSE interface by sending bad command ;@0xAA;
    ////////////////////////////////////////

```

```

    OutputBuffer[dwNumBytesToSend++] = 0xAA;          //Add BAD command
;@0xAA;
    ftStatus = FT_Write(ftHandle, OutputBuffer, dwNumBytesToSend,
&dwNumBytesSent); // Send off the BAD commands
    dwNumBytesToSend = 0;          //Clear output buffer
    do{
        ftStatus = FT_GetQueueStatus(ftHandle, &dwNumInputBuffer); //
Get the number of bytes in the device input buffer
    }while ((dwNumInputBuffer == 0) && (ftStatus == FT_OK)); //or
Timeout

    bool bCommandEchod = false;
    ftStatus = FT_Read(ftHandle, &InputBuffer, dwNumInputBuffer,
&dwNumBytesRead); //Read out the data from input buffer
    for (dwCount = 0; dwCount < dwNumBytesRead - 1; dwCount++) //Check
if Bad command and echo command received
    {
        if ((InputBuffer[dwCount] == BYTE('\xFA')) &&
(InputBuffer[dwCount+1] == BYTE('\xAA')))
        {
            bCommandEchod = true;
            break;
        }
    }
    if (bCommandEchod == false)
    {
        ui->display->setTextColor(Qt::red);
        ui->display->setText("fail to synchronize MPSSE with command '0xAA'
\n");
        printf("fail to synchronize MPSSE with command '0xAA' \n");
        return; /*Error, can;t receive echo command , fail to synchronize
MPSSE interface;*/
    }

    ////////////////////////////////////////////////////////////////////
    // Synchronize the MPSSE interface by sending bad command ;@0xAB;
    ////////////////////////////////////////////////////////////////////
    //dwNumBytesToSend = 0;          //Clear output buffer

    OutputBuffer[dwNumBytesToSend++] = 0xAB; //Send BAD command ;@0xAB;
    ftStatus = FT_Write(ftHandle, OutputBuffer, dwNumBytesToSend,
&dwNumBytesSent); // Send off the BAD commands
    dwNumBytesToSend = 0;          //Clear output buffer
    do{
        ftStatus = FT_GetQueueStatus(ftHandle, &dwNumInputBuffer); //Get
the number of bytes in the device input buffer
    }while ((dwNumInputBuffer == 0) && (ftStatus == FT_OK)); //or Timeout
    bCommandEchod = false;
    ftStatus = FT_Read(ftHandle, &InputBuffer, dwNumInputBuffer,
&dwNumBytesRead); //Read out the data from input buffer
    for (dwCount = 0;dwCount < dwNumBytesRead - 1; dwCount++) //Check if
Bad command and echo command received
    {
        if ((InputBuffer[dwCount] == BYTE('\xFA')) &&
(InputBuffer[dwCount+1] == BYTE( '\xAB')))
        {
            bCommandEchod = true;
            break;
        }
    }
    if (bCommandEchod == false)
    {
        ui->display->setTextColor(Qt::red);

```

```

        ui->display->setText("fail to synchronize MPSSE with command '0xAB'
\n");
        printf("fail to synchronize MPSSE with command '0xAB' \n");
        return;
        /*Error, can't receive echo command , fail to synchronize MPSSE
interface;*/
    }

    if (ftStatus == FT_OK&&check==1)
    {
        printf("MPSSE synchronized with BAD command \n");

        ui->display->setTextColor(Qt::green);
        ui->display->insertPlainText("\n MPSSE synchronized with BAD command
\n");
        ui->display-
>insertPlainText("\n_____ \n\n");
        check=0;
    }

    ////////////////////////////////////////
    //Configure the MPSSE for I2C
    ////////////////////////////////////////

    OutputBuffer[dwNumBytesToSend++] = 0x8A;    //Ensure disable clock
divide by 5 for 60Mhz master clock
    OutputBuffer[dwNumBytesToSend++] = 0x97;    //Ensure turn off adaptive
clocking
    OutputBuffer[dwNumBytesToSend++] = 0x8D;    //8C for Enable 3 phase
data clock, used by I2C to allow data on both clock edges
    ftStatus = FT_Write(ftHandle, OutputBuffer, dwNumBytesToSend,
&dwNumBytesSent); // Send off the commands

    // The SK clock frequency can be worked out by below algorithm with
divide by 5 set as off
    // SK frequency = 60MHz /((1 + [(1 +0xValueH*256) OR 0xValueL])*2)
    OutputBuffer[dwNumBytesToSend++] = 0x86;    //Command to set
clock divisor
    OutputBuffer[dwNumBytesToSend++] = dwClockDivisor & 0xFF;    //Set
0xValueL of clock divisor
    OutputBuffer[dwNumBytesToSend++] = (dwClockDivisor >> 8) & 0xFF;
//Set 0xValueH of clock divisor
    ftStatus = FT_Write(ftHandle, OutputBuffer, dwNumBytesToSend,
&dwNumBytesSent); // Send off the commands
    dwNumBytesToSend = 0;    //Clear output buffer

    dwNumBytesToSend = 0;    //Clear output buffer
    OutputBuffer[dwNumBytesToSend++] = 0x80;    //Command to set
directions of lower 8 pins and force value on bits set as output
    OutputBuffer[dwNumBytesToSend++] = '\x03';    //Set SDA, SCL high, WP
disabled by SK, DO at bit ;1; , GPIOL0 at bit ;0;
    OutputBuffer[dwNumBytesToSend++] = '\x13';    //Set SK,DO,GPIOL0 pins
as output with bit ;1; , other pins as input with bit ;0;

    usleep(20);    //Delay for a while

    //Turn off loop back in case
    OutputBuffer[dwNumBytesToSend++] = 0x85;    //Command to turn off
loop back of TDI/TDO connection

```

```

    ftStatus = FT_Write(ftHandle, OutputBuffer, dwNumBytesToSend,
&dwNumBytesSent);    // Send off the commands
    dwNumBytesToSend = 0;    //Clear output buffer
    usleep(30);    //Delay for a while
}
}

void SteuerungI2C::on_beenden_clicked()
{
    this->hide();
}

int Frequenzrechner(int f)
{
    f=((60*1000000)/(f*1000*2))-1;

    //f=f*2/3; //bei dreiphasentaktung !!!MPSSE dementsprechend auf 8C
ändern!!!
    qDebug()<<f;
    return f;
}

void SteuerungI2C::Eingabenauswerten()
{
    MPSSE();
    scltohex = ui->scl->text();
    scltosend = scltohex.toInt(&ok, 10); // String in Hex umwandeln

    eeepromtohex = ui->eeeprom->text();
    eeepromadress = eeepromtohex.toInt(&ok, 16); // String in Hex umwandeln

    hadresstohex = ui->highaddress->text();
    highadress = hadresstohex.toInt(&ok, 16); // String in Hex umwandeln

    adresstohex = ui->adresse->text();
    sadresse = adresstohex.toInt(&ok, 16); // String in Hex umwandeln

    bytetohex = ui->byteadress->text();
    bytetosend = bytetohex.toInt(&ok, 16); // String in Hex umwandeln

    dwClockDivisor=Frequenzrechner(scltosend);

    ByteAddressHigh=highadress;
    ByteAddressLow = sadresse; //Set read address is 0x0080 as
example
    ByteDataToBeSend=bytetosend;
    EepromAdresse=eeepromadress;
    check2=1;
    zahl++;
}

void SteuerungI2C::on_schreiben_clicked()
{

```

```

    Eingabenauswerten();
    HighSpeedSetI2CStart();
    bSucceed = SendByteAndCheckACK(EepromAdresse); //Send control byte
and check the ACK bit
    bSucceed = SendByteAndCheckACK(ByteAddressHigh); // Send word
high address byte and check the ACK bit
    bSucceed = SendByteAndCheckACK(ByteAddressLow); // Send word
low address byte and check the ACK bit
    bSucceed = SendByteAndCheckACK(ByteDataToBeSend); // Send data
byte and check the ACK bit
    HighSpeedSetI2CStop(); // Set I2C
Stop Condition

    ftStatus = FT_Write(ftHandle, OutputBuffer, dwNumBytesToSend,
&dwNumBytesSent);
    dwNumBytesToSend = 0; //Clear output buffer

    usleep(10000);

    qDebug() << "SCHREIBEN";
    qDebug() << "BYTE Adresse: " << ByteAddressLow << " : " << ByteDataToBeSend;

    if(check2==1)
    {
        QString str, str1, str2, str3;
        str = "HighAddress : " + str1.setNum(ByteAddressHigh, 16) + " LowAddress :
"+str.setNum(ByteAddressLow, 16) + "\n";
        str2 = "BYTE : " + str2.setNum(ByteDataToBeSend, 16) + "\n";
        str3 = str3.setNum(zahl) + ": \nSchreiben \n";
        ui->display->setTextColor(Qt::blue);
        ui->display->insertPlainText(str3);
        ui->display->insertPlainText(str+str2);
        ui->display-
>insertPlainText("_____ \n\n");
    }

    qDebug() << " " << endl;
    qDebug() << "Schreiben zu ende" << endl;
    qDebug() << ". . ." << endl;
}

void SteuerungI2C::onlesen_clicked()
{
    qDebug() << "Lesen Beginnt" << endl;
    Eingabenauswerten();

    ftStatus = FT_GetQueueStatus(ftHandle, &dwNumInputBuffer); //
Get the number of bytes in the device receive buffer
    if ((ftStatus == FT_OK) && (dwNumInputBuffer > 0))
        FT_Read(ftHandle, &InputBuffer, dwNumInputBuffer, &dwNumBytesRead);
//Read out all the data from receive buffer
    HighSpeedSetI2CStart(); //Set START condition for I2C
communication
    bSucceed = SendByteAndCheckACK(EepromAdresse); //Set control byte
and check ACK bit.bit 4 - 7 of control byte is control code,
// bit 1 - 3 of " 111 " as
block select bits , bit 0 of " 0 " represent Write operation
    bSucceed = SendByteAndCheckACK(ByteAddressHigh); //Send high
address byte and check if ACK bit is received

```

```

        bSucceed = SendByteAndCheckACK(ByteAddressLow);    //Send low
address byte and check if ACK bit is received

        HighSpeedSetI2CStart();        //Set START condition for I2C
communication
        bSucceed = SendByteAndCheckACK(EepromAdresse+1);    //Set control
byte and check ACK bit. bit 4- 7 as „1010 “ of control byte is control
code,
                                                // bit 1-3 of „111“ as block
select bits, bit 0 as „ 1“represent Read operation

        ///////////////////////////////////////////////////////////////////
        // Read the data from 24LC256 with no ACK bit check
        ///////////////////////////////////////////////////////////////////
        OutputBuffer[dwNumBytesToSend++] = 0x80;    //Command to set
directions of lower 8 pins and force value on bits set as output
        OutputBuffer[dwNumBytesToSend++] = '\x00';    //Set SCL low, WP
disabled by SK, GPIOI0 at bit „“
        OutputBuffer[dwNumBytesToSend++] = '\x11';    //Set SK, GPIOI0 pins
as output with bit “”, DO and other pins as input with bit „“
        OutputBuffer[dwNumBytesToSend++] = MSB_FALLING_EDGE_CLOCK_BYTE_IN;
//Comand to clock data byte in on - ve Clock Edge MSB first
        OutputBuffer[dwNumBytesToSend++] = '\x00';
        OutputBuffer[dwNumBytesToSend++] = '\x00';    //Data length of
0x0000 means 1 byte data to clock in
        OutputBuffer[dwNumBytesToSend++] = MSB_RISING_EDGE_CLOCK_BIT_IN;
//Command to scan in acknowledge bit ,-ve clock Edge MSB first
        OutputBuffer[dwNumBytesToSend++] = '\x0';    //Length of 0 means to
scan in 1 bit
        OutputBuffer[dwNumBytesToSend++] = 0x87;    //Send answer back
immediate command
        ftStatus = FT_Write(ftHandle, OutputBuffer, dwNumBytesToSend,
&dwNumBytesSent);    //Send off the commands
        dwNumBytesToSend = 0;    //Clear output buffer
                                                //Read two bytes from device receive
buffer,first byte is data read from EEPROM, second byte is ACK bit
        ftStatus = FT_Read(ftHandle, InputBuffer, 2, &dwNumBytesRead);
        ByteDataRead= InputBuffer[0];    //Return the data read from EEPROM

        OutputBuffer[dwNumBytesToSend++] = 0x80;    //Command to set
directions of lower 8 pins and force value on bits set as output
        OutputBuffer[dwNumBytesToSend++] = '\x08';    //Set SDA high, SCL
low, WP disabled by SK at bit '0', DO, GPIOI0 at bit '1'
        OutputBuffer[dwNumBytesToSend++] = '\x0B';    //Set SK,DO,GPIOI0 pins
as output with bit “”, other pins as input with bit „“
        HighSpeedSetI2CStop();        //Set STOP condition for
I2C communication
        //Send off the commands
        ftStatus = FT_Write(ftHandle, OutputBuffer, dwNumBytesToSend,
&dwNumBytesSent);
        dwNumBytesToSend = 0;    //Clear output buffer

        usleep(10000);

        qDebug() << "LESEN";
        qDebug() << "BYTE Adresse: " << ByteAddressLow << " : " << ByteDataRead;
        qDebug() << "Lesen zu ende" << endl;
        qDebug() << ". . ." << endl;

        if (check2==1)

```

```

    {
        QString str, str1, str2, str3;
        str= "HighAddress : "+str1.setNum(ByteAddressHigh,16)+" LowAddress :
"+str.setNum(ByteAddressLow,16)+"\n";
        str2="BYTE :"+str2.setNum(ByteDataRead,16)+"\n";
        str3=str3.setNum(zahl)+" : \nLesen \n";
        ui->display->setTextColor(Qt::blue);
        ui->display->insertPlainText(str3);
        ui->display->insertPlainText(str+str2);
        ui->display-
>insertPlainText("_____ \n\n");
    }
}

```

Steuerungled.cpp

```

#include <WinTypes.h>
#include <ftd2xx.h>
#include "steuerungled.h"
#include "ui_steuerungled.h"
#include <steuerungi2c.h>
#include <QApplication>
#include <stdio.h>
#include <iostream>
#include <stdlib.h>
#include <assert.h>
#include <QDebug>
#include<QString>
#include <time.h>
#include<QMessageBox>
#include<QDesktopServices>
#include<QUrl>

using namespace std;
FT_STATUS ftStatu = FT_OK;
FT_HANDLE ftHandl;
DWORD bytesWritten;
UCHAR outputData=0;
int portNumber=0;

SteuerungLed::SteuerungLed(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::SteuerungLed)
{
    ftStatu = FT_Open(portNumber, &ftHandl);
    ftStatu = FT_SetBitMode(ftHandl,
                            0xFF, // sets all 8 pins as outputs
                            FT_BITMODE_SYNC_BITBANG);

    ui->setupUi(this);

    QPixmap pix("/home/oezkan_local/Downloads/fhdo.png");
    QPixmap prof("/home/oezkan_local/Downloads/Prof.png");
    ui->label_pic->setPixmap(pix);
    ui->label_pic->setScaledContents(true);
    ui->bild->setIcon(prof);
    ui->bild->setIconSize(QSize(190,200));
}

```



```

}

SteuerungLed::~SteuerungLed()
{
    FT_Close(ftHandl);
    delete ui;
}

void SteuerungLed::on_beenden_clicked()
{
    this->hide();
}

void SteuerungLed::on_ein_clicked()
{
    outputData = 0xFF;
    ftStatu = FT_Write(ftHandl, &outputData, 1, &bytesWritten);
}

void SteuerungLed::on_AUS_clicked()
{
    outputData = 0x00;
    ftStatu = FT_Write(ftHandl, &outputData, 1, &bytesWritten);
}

void SteuerungLed::on_eins_clicked()
{
    outputData = 0x01<<4;
    ftStatu = FT_Write(ftHandl, &outputData, 1, &bytesWritten);
}

void SteuerungLed::on_zwei_clicked()
{
    outputData = 0x02<<4;
    ftStatu = FT_Write(ftHandl, &outputData, 1, &bytesWritten);
}

void SteuerungLed::on_drei_clicked()
{
    outputData = 0x04<<4;
    ftStatu = FT_Write(ftHandl, &outputData, 1, &bytesWritten);
}

void SteuerungLed::on_vier_clicked()
{
    outputData = 0x08<<4;
    ftStatu = FT_Write(ftHandl, &outputData, 1, &bytesWritten);
}

void SteuerungLed::on_los_clicked()
{
    QString str, str2 ;
    bool ok;

    str = ui->lauflicht->text();
    int z = str.toInt(&ok, 16); // String in Hex umwandeln

    str2 = ui->zeit->text();

```

```

int zeit = str2.toInt(&ok, 10); // String in Hex umwandeln
zeit=zeit*1000;

for(z;z>0;z--)
{
int z1=1;

for(z1;z1<9;z1*=2)
{ // variable hochzählen
outputData = z1<<4;
ftStatu = FT_Write(ftHandl, &outputData, 1, &bytesWritten);
usleep(zeit);

if(z1>7)
{
for(int z2=8;z2>1;z2/=2) // variable runterzählen
{

if(z2==8)
{

continue;

}
else
{

outputData = z2<<4;
ftStatu = FT_Write(ftHandl, &outputData, 1,
&bytesWritten);
usleep(zeit);

}

}

outputData = 1<<4;
ftStatu = FT_Write(ftHandl, &outputData, 1, &bytesWritten);
}
}
}
}

void SteuerungLed::on_hoch_clicked()
{

QString str2;
bool ok;
str2 = ui->zeit->text();
int zeit = str2.toInt(&ok, 10); // String in Hex umwandeln
zeit=zeit*1000;

for(int i=0;i<16;i++ )
{
outputData = i<<4;
ftStatu = FT_Write(ftHandl, &outputData, 1, &bytesWritten);
usleep(zeit);
}

}

void SteuerungLed::on_runter_clicked()
{
QString str2;

```

```

        bool ok;
        str2 = ui->zeit->text();
        int zeit = str2.toInt(&ok, 10); // String in Hex umwandeln
        zeit=zeit*1000;
        for(int i=16;i>=0;i-- )
    {
        outputData = i<<4;
        ftStatu = FT_Write(ftHandl, &outputData, 1, &bytesWritten);
        usleep(zeit);
    }
}

void SteuerungLed::on_hexbin_2_clicked()
{
    QString str ;
    bool ok;
    str = ui->hexbin->text();
    int hex = str.toInt(&ok, 16); // String in Hex umwandeln
    outputData = hex<<4;
    ftStatu = FT_Write(ftHandl, &outputData, 1, &bytesWritten);
}

void SteuerungLed::on_spinBox_valueChanged(int arg1)
{
    int inhalt;
    inhalt=ui->spinBox->value();

    outputData = inhalt<<4;
    ftStatu = FT_Write(ftHandl, &outputData, 1, &bytesWritten);
}

void SteuerungLed::on_bild_clicked()
{
    QString link ="https://www.fh-
dortmund.de/de/fb/3/personen/lehr/karagounis/index.php";
    QDesktopServices::openUrl(QUrl(link));
}

```

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

Ort, Datum

Nurullah Özkan