CCA – COMPETENCE CENTRE

HTL Anichstraße

DIPLOMARBEIT

Grafische Oberfläche für Signalgenerator FPGA

HÖHERE TECHNISCHE BUNDESLEHR- UND VERSUCHSANSTALT

ANICHSTRASSE

Abteilung

ELEKTRONIK UND TECHNISCHE INFORMATIK

Ausgeführt im Schuljahr 2020/21 von:

Plunser Fabio 5BHEL

Rieser David 5BHEL

Betreuer/Betreuerin:

Mag. DI Klingler Felix

Projektpartner: INNIO / Jenbacher Werke

Innsbruck, am 26.3.2021

Abgabevermerk:

Datum:

Betreuer/in:

ERKLÄRUNG DER EIGENSTÄNDIGKEIT DER ARBEIT

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche erkenntlich gemacht habe.

Innsbruck 24.3
Ort, Datum

Rieser David
Verfasser: Rieser David

Innsbruck, 24.03.2020
Ort, Datum

Plunser
Verfasser: Plunser Fabio

GENDERERKLÄRUNG

Aus Gründen der besseren Lesbarkeit wird in dieser Diplomarbeit die Sprachform des generischen Maskulinums angewendet. Es wird an dieser Stelle darauf hingewiesen, dass die ausschließliche Verwendung der männlichen Form geschlechtsunabhängig verstanden werden soll.

ZUSAMMENFASSUNG DES PROJEKTERGEBNISSES

Kurzfassung

Die vorliegende Diplomschrift beschäftigt sich mit der Erstellung einer grafischen Benutzeroberfläche zur Ansteuerung eines FPGA, der verwendet wird, um die Motorsteuerung der Gasmotoren zu testen.
Die Firma INNIO / Jenbacher hat den FPGA von der technischen Universität Wien entwickeln lassen. Zur Ansteuerung des FPGA wurde eine Excel-Benutzeroberfläche erstellt, die jedoch nicht den Ansprüchen der Firma entspricht.

Die Hauptaufgabe besteht darin, in der Programmiersprache Python mit PyQt, mittels Qt-Designer und Pyserial eine grafische Benutzeroberfläche zu erstellen, die einfach zu verwenden ist und alle gewünschten Funktionen bereitstellt.
Ebenfalls soll für die GUI eine Anleitung erstellt werden und der VHDL Code des FPGA verstanden und dokumentiert werden.

ABSTRACT

The present diploma thesis deals with the creation of a graphical user interface to control an FPGA, which is used to test the engine control of gas engines. INNIO / Jenbacher had the FPGA developed by the Technical University of Vienna. An Excel user interface was created to control the FPGA, but it does not meet the company's requirements.

The main task is to create an easy-to-use graphical user interface in the Python programming language with PyQt, Qt Designer and Pyserial and contains all the desired functions. A manual should also be created for the GUI. Furthermore, the VHDL code of the FPGA should be understood and documented.

PROJEKTERGEBNIS

Das Ergebnis dieser Arbeit besteht aus einer grafischen Benutzeroberfläche, einer Anleitung für diese grafische Benutzeroberfläche und der Dokumentation des bereitgestellten VHDL-Codes.

Bei der grafischen Benutzeroberfläche handelt es sich um eine GUI zur Ansteuerung eines FPGA, der einen Gasmotor simuliert. Die GUI besitzt folgende Eigenschaften:

- Speicherbare Einstellungen
- Möglichkeit die Drehzahl zu ändern auch während der Motor läuft
- Darstellung der Drehzahlkurve
- eingebauter Editor zu Änderung von importierten Signalen
- automatische Erkennung an welchem Port der FPGA angeschlossen ist
- übersichtliches Design
- und vieles mehr

Abbildung 1: Projektergebnis

DANKSAGUNG

An dieser Stelle möchten wir uns bei allen bedanken, die uns bei einer erfolgreichen Umsetzung der Diplomarbeit geholfen haben.
Als erstes möchten wir uns recht herzlich bei unseren Eltern bedanken, die uns die gesamte Zeit lang unterstützt haben.
Weiterhin danken wir unserem Projektpartner INNIO / Jenbacher, insbesondere Hr. Perktold Michael und Hr. Kopecek Herbert, die uns diese Diplomarbeit ermöglichten. Auch bei unserem Betreuer seitens INNIO, Hr. Martin Gyurkó, möchten wir uns für seine Hilfe bei der Festlegung der Projektergebnisse, der Hilfe bei der grafischen Benutzeroberfläche und vielem mehr, herzlich bedanken.

Ebenfalls danken wir unserem Betreuungslehrer Mag. DI. Klingler Felix für seine Geduld, Hilfsbereitschaft und Unterstützung während der gesamten Umsetzung der Arbeit.

Ein abschließender Dank an alle Lehrer und Mitschüler, die bei der Ausarbeitung mitgewirkt haben und an alle Korrekturleser der finalen Diplomarbeit.

Inhaltsverzeichnis

1 EINLEITUNG

Die Firma INNIO / Jenbacher entwickelt und produziert Gasmotoren zur Stromerzeugung. Um diese Motoren zu steuern, verwendet die Firma sogenannte „SAFI“-Geräte, diese können zwei Zylinder komplett steuern und überwachen. Diese SAFI's werden von einer anderen Firma hergestellt, daher sind die Pläne dieser Geräte für INNIO / Jenbacher nicht einsehbar.
Fehler können daher im praktischen Umfeld schlecht kontrolliert, und die realen Konditionen, im Prüfstand, schwer nachgebaut werden. Um diese SAFI ohne großen Gasmotor testen zu können, wurde mithilfe der technischen Universität Wien ein Signalgenerator, basierend auf einem FPGA entwickelt, wobei die genaue Funktionsweise und der VHDL-Code des FPGAs nicht dokumentiert wurde.
Der Signalgenerator besitzt folgende Eigenschaften:

1. Er simuliert bis zu 24 Zylinder eines Motors mit vorgegebenen Verbrennungssignalen.
2. Er kann auf Knopfdruck diese Signale mit einem Rausch-Signal überlagern, um das Verhalten der „SAFI“-Geräte unter reelen Konditionen und mit Störsignalen zu testen.
3. Er erzeugt Geschwindigkeitssignale, und winkel- und zeitsynchron dazu die Verbrennungssignale.
4. Für jeden Zylinder werden folgende Signale mithilfe eines Digital-Analog Wandler erzeugt und an das SAFI Gerät weitergeleitet:
    a Druckkurve
    b Klopfsignal
    c Hochspannungssignal
    d Abgastemperatur

Als Digital-Analog-Wandler werden Digilent PMOD-DA4 verwendet. Diese werden mithilfe eines SPI-Bus (Serial-Peripheral-Interface-Bus) an dem FPGA angeschlossen. Um diesen FPGA zu steuern, wurde von der TU Wien eine Excel-Datei erstellt, die die serielle Schnittstelle beschreibt und ausliest. Da diese Excel Datei unzuverlässig und schwierig zu verwenden ist, die Aufgabe des Projektes eine neue grafische Benutzeroberfläche, zu erstellen. Diese basiert auf der Programmiersprache Python und muss alle Features der Excel übernehmen und noch beliebig erweitert werden können. Somit ist die Zielsetzung des Projektes, eine grafische Benutzeroberfläche, zu erstellen, die einen FPGA ansteuert und die Daten ausliest. Weiterhin soll eine Dokumentation des VHDL Codes des FPGAs erstellt werden, um die Funktionsweise zu dokumentieren.

2 VERTIEFENDE AUFGABENSTELLUNG

2.1 Rieser David

Ein Teil der Aufgabe ist, es den Datenaustausch zwischen Excel und FPGA, durch Dokumentation des FPGA-Codes zu verstehen. Dabei sollte herausgefunden werden:

- Welche Befehle es gibt
- Was diese bewirken
- Wie empfangene Daten seitens der Excel interpretiert werden.

Diese Erkenntnisse sollen dann in der GUI mithilfe von PySerial implementiert werden. Ausserdem sollen die Verbindungen der GUI thread-sicher implementiert werden.

2.2 Plunser Fabio

Um die GUI auf Basis der Programmiersprache Python mit PyQT und Pyserial erstellen zu können, werden folgende Punkte abgearbeitet:

- PyQT, PySerial kennenlernen
- Excel Script verstehen
- mit PyQT die GUI erstellen
- mit Pyserial eine serielle Kommunikation mit dem FPGA herstellen

DOKUMENTATION DER ARBEIT

Die Quellenangaben für diese Arbeit befinden sich im Literaturverzeichnis, wobei die Fußnoten auf das Literaturverzeichnis referenzieren.

3 THEORIE UND VORWISSEN

3.1 FPGA und VHDL

Ein **Field Programmable Gate Array (FPGA)** ist ein integrierter Schaltkreis, welcher mithilfe einer logischen Schaltung programmiert werden kann. Anders als bei Computern und Mikrocontrollern bezieht sich der Begriff Programmierung hier darauf, dass im FPGA intern logische Gatter verbunden werden, um die Funktion des Programms in Hardware zu implementieren. Dies wird mittels einer Hardwarebeschreibungssprache wie **VHDL** formuliert.[1]

Very High Speed Integrated Circuit Hardware Description Language (VHDL) ist eine Hardwarebeschreibungssprache. Mit VHDL wird das gewünschte Verhalten einer Schaltung beschrieben. Dies kann mithilfe von concurrent Statements (parrallele Schaltungsstruktur) oder mit sequential Statements (sequentieller Schaltungsstruktur) erreicht werden.

[1] Vgl. **[1]**

3.1.1 AsciiDoc

Die Dokumentation des VHDL-Code soll mithilfe von AsciiDoc erstellt werden. AsciiDoc ist eine Auszeichnungssprache welche es ermöglicht ein Textdokument z.B. in eine PDF zu kompilieren.

```
= Introduction to AsciiDoc
Doc Writer <doc@example.com>

A preface about https://asciidoc.org[AsciiDoc].

== First Section

* item 1
* item 2

[source,ruby]
puts "Hello, World!"
```

Codeausschnitt 1: Beispiel AsciiDoc von https://asciidoctor.org/docs/what-is-asciidoc/

3.2 SPI

Das Bus-System **Serial Peripheral Interface**, das vom Halbleiterhersteller Motorola entwickelt wurde, stellt einen Standard für einen synchron seriellen Datenbus dar, mit dem digitale Schaltungen nach dem Master-Slave-Prinzip miteinander verbunden werden können.

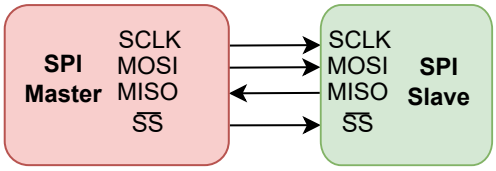


Abbildung 2: SPI-single-slave

3.2.1 SPI-Eigenschaften

Das SPI-Protokoll besitzt folgende Eigenschaften

- Drei gemeinsame Leitungen, an denen jeder Teilnehmer angeschlossen ist:
  1. SCLK auch SCK, wird vom Master zu Synchronisation ausgegeben
  2. MOSI oder SIMO (*Master Output, Slave Input*)
  3. MISO oder SOMI (*Master Inputm Slave Output*)
- Eine oder mehrere Chip-Select Leitungen mit der Beziehcnung: $\overline{\mathrm{SS}}$
- Vollduplexfähig
- Unterscheidliche Taktfrequenzen bis in den MHz-Bereich sind zulässig.

3.2.2 SPI-Funktion und Protokollablauf

An dem Bus können so viele Teilnehmer angeschlossen werden, wie Slave-Select-Leitungen vorhanden sind. Der Master legt mit der Leitung Slave Select $\overline{\mathrm{SS}}$ fest, mit welchem Slave er kommunizieren will. Wird sie gegen Masse gezogen, ist der jeweilige Slave aktiv und lauscht an der MOSI Leitung und oder legt seine Daten im Takt von SCLK and MISO. Es wird ein Wort vom Master zum Slave und ein anderes Wort vom Slave zum Master transportiert.
Ein Protokoll für die Datenübertragung ist nicht festgelegt.[2]

[2] Vgl. **[4]**

3.3 UART

Universal Asynchronous Receiver Transmitter ist eine elektronische Schaltung, die dazu dient, eine digitale serielle Schnittstelle zu realisieren. Eine UART-Schnittstelle dient zum Senden und Empfangen von Daten über eine Datenleitung und bildet den Standard der seriellen Schnittstellen an PCs und Mikrocontrollern.

3.3.1 Asynchrone Datenübertragung

Bei der asynchronen Betriebsweise wird das Taktsignal **nicht** über eine eigene Steuerleitung übertragen. Stattdessen synchronisiert sich der Empfänger über die Länge des Frames, vermittelt durch die Vorderflanke des neuen Start-Bits nach dem letzten empfangenen Stop-Bit, sowie die eingestellte Baudrate. Da der Beginn einer Übertragung mit dem Start-Bit zu beliebigen Zeitpunkten erfolgen kann, wird diese serielle Schnittstelle als asynchron bezeichnet. Um eine Synchronisation gewährleisten zu können, ist die Anzahl der übertragbaren Datenbits innerhalb eines Frames eingeschränkt. Würde mehr als ein Byte in einen Frame verpackt werden, könnte die Synchronisation verloren gehen, was zu Fehlinterpretationen des Datenstromes und somit zu einer fehlerhaften Übertragung führen könnte. Wenn in einer Sendepause keine Daten zu übertragen sind, so legt der Sender die Leitung auf die Polarität des Stop-Bits. Weil der Empfänger sich mit jedem übertragenen Frame neu synchronisiert, ist es nicht notwendig, dass zwischen dem übertragenen Frame ein zeitlicher Zusammenhang besteht. Nur für die Dauer eines einzelnen Frame müssen Sender und Empfänger synchron arbeiten, nicht länger.[3]

[3] Vgl. **[5]**

3.3.2 UART-Übertragung-Darstellung

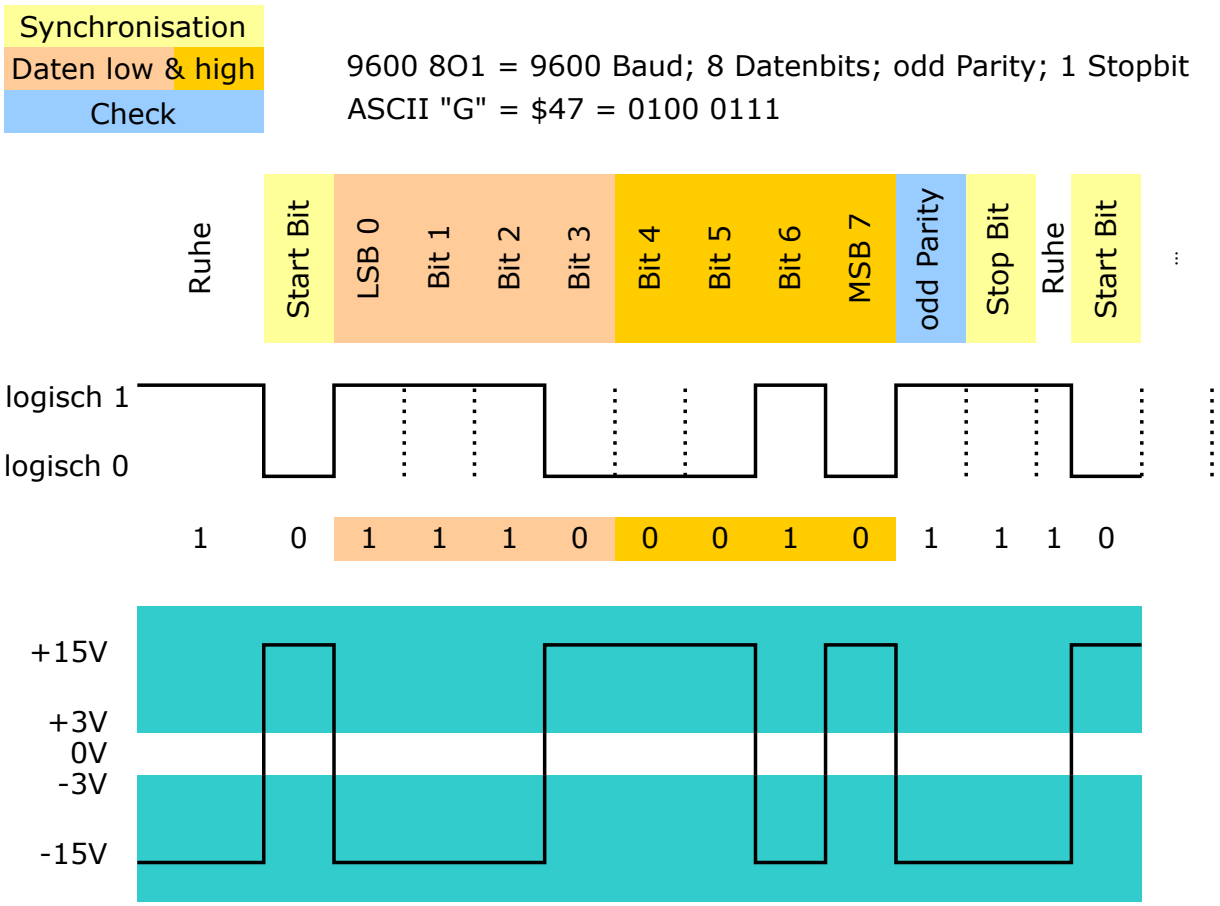


Abbildung 3: UART-Übertragung-Darstellung[4]

[4]Quelle: https://upload.wikimedia.org/wikipedia/commons/f/f3/RS-232_timing.svg

3.4 Digital-Analog-Wandler PMOD-DA4

Ein Digital-Analog-Wandler wird verwendet, um digitale Signale in analoge Signale umzusetzen. Er erzeugt aus einem zeit- und wertdiskreten digitalen Signal, ein kontinuierliches analoges Signal. [5]
Der PMOD-DA4 funktioniert mithilfe einer Widerstands-Kette, die die Versorgungsspannung teilt.

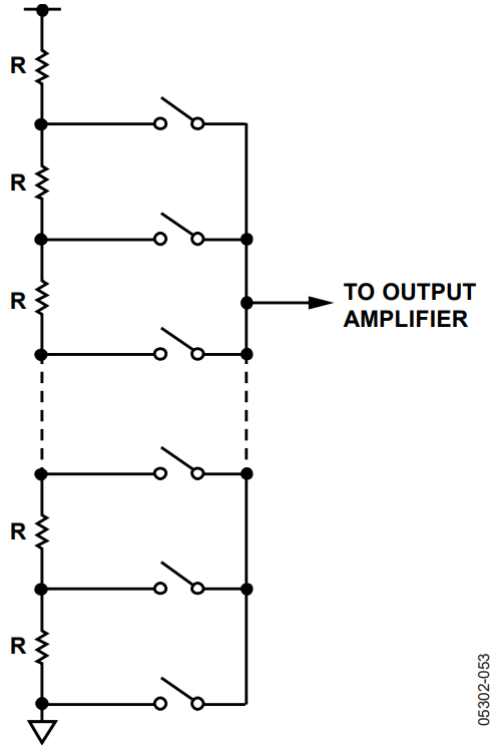


Abbildung 4: DAW-Funktionsweise

Da die Widerstände alle den gleichen Wert haben fällt bei jedem Widerstand eine Spannung von $U_R = U_V \cdot \frac{1}{\text{Anzahl Widerstände}}$ ab. Wenn nun ein digitaler Wert mit SPI übertragen wird, konvertiert der DAW diesen, indem er den Schalter, der diesem Digitalwert entspricht, schließt und alle anderen öffnet.
Daraus ergibt sich folgende Gleichung: $U_A = U_R \cdot \text{Digitalwert} = U_V \cdot \frac{\text{Digitalwert}}{\text{Anzahl Widerstände}}$

[5] Vgl. **[6]**

3.5 Threading

Threading bezeichnet die Möglichkeit verschiedene Prozess eines Programms unabhängig und parallel auszuführen.
Wenn Threads eingestetzt werden muss jedoch bedacht werden, dass diese gleichzeitig auf die gleichen Ressourcen zugreifen können, was zu Problemen führen kann. Um dies zu verhindern werden Thread-sichere Ressourcen verwendet wie z.B.:

- Events
  Ein Thread kann warten bis ein anderer Thread mithilfe des Event signalisiert das ein Ereignis aufgetreten ist.
- Message-Queues
  Message Queues sind FIFO-Buffers (First-In First-Out). Man kann sie sich wie eine Liste vorstellen, in der Daten nur am Anfang hinzufügt und am Ende entnommen werden können.

Im folgenden Beispiel wurde die Struktur eines fiktiven GUI-Programms skizziert,
welches mit einem externen Gerät eine aktive Verbindung besitzt.
Thread 1 und 2 verwalten das Rendern und die Funktionen der Front-End während Thread 3 mit dem externen Gerät kommuniziert.
Um maximale Datenrate zu sichern liest Thread 3 nur die Daten aus und sendet sie mithilfe einer Message-Queue, welche 5 Datensätze beinhalten kann, an die Threads 4 und 5, welche die erhaltenen Daten verarbeiten.
Wenn nun relevante Daten empfangen werden, signalisieren Thread 4 und 5 der Front-End das dem Nutzer diese Daten angezeigt werden sollen.

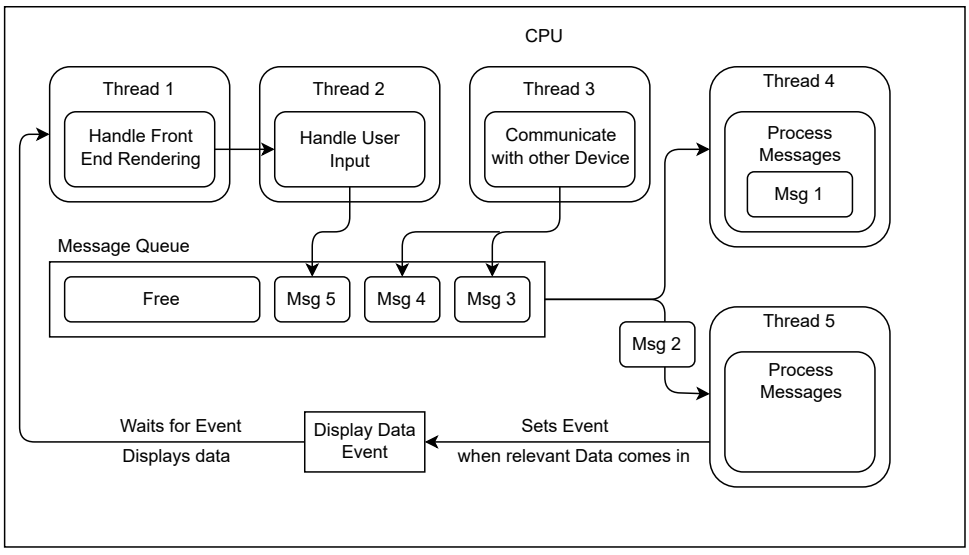


Abbildung 5: Threads, Queues und Events

3.6 Python

Python ist eine objektorientierte, interpreter Programmiersprache. Sie hat den Anspruch, einen gut lesbaren, knappen Programmierstil zu fördern. So müssen beispielsweise Code-Blöcke eingerückt werden, um sie hierarchisch zu strukturieren.
Die Programmiersprache muss auf dem typischen Windows und MACOS Computer zusätzlich installiert werden. Sie bietet die Möglichkeit, aus einem großen Onlineangebot an Paketen unterschiedliche Funktionen zu downloaden und zu importieren. So werden, zum Beispiel, in dieser Arbeit die Pakete PyQt5 und Pyserial verwendet.

3.6.1 Interpreter

Ein Interpreter verarbeitet den Quellcode eines Projekts. Er geht den Sourcecode Zeile für Zeile durch und verifiziert die syntaktische Richtigkeit des Programms. Wenn das Programm keine Fehler enthält, verarbeitet er den Sourcecode in Byte-Code, der dann von der Python-Virtual-Machine ausgeführt wird. Die Python Virtual Machine kann plattformunabhängig diesen Byte-Code ausführen, da der Byte-Code auf jedem System, mit den für das System richtigen Assembler-Befehlen implementiert wird.[6]

3.6.2 Python Packages

Um eine Aufteilung des Programms in unterschiedliche Dateien zu erreichen, kann ein Projekt folgendermaßen als ein *Package* realisiert werden.

```
parent/
    __init__.py
    one/
        __init__.py
    two/
        __init__.py
    three/
        __init__.py
```

Abbildung 6: Python-Package

Ordner welche eine *__init__.py* Datei beinhalten, sind sogenante *Packages bzw. Pakete*. Eine .py Datei die sich in einem dieser Pakte befindet, wird *Module bzw. Modul* genannt.[7]
Module können in Python relativ importiert werden. Das bedeutet, es wird nicht der gesamte Pfad der importierten Datei benötigt, sondern der Weg von der Datei, worin der Import stattfindet, zu der Datei, die importiert werden soll.

[6] Vgl. **[7]**
[7] Vgl. **[8]**

3.6.3 Pakete installieren

Um Pakete wie PyQt5 zu installieren, wird **pip** verwendet. Dieser *package installer* wird bei der Installation von Python mit installiert.
Folgender Befehl muss in der Windows-Eingabeaufforderung ausgeführt werden um PyQt5 zu installieren.

```
1 python -m pip install PyQt5
2
```

Codeausschnitt 2: Paket-installieren-Beispiel

3.6.4 PyQt5

PyQt5[8] besteht aus plattformübergreifenden C++ Bibliotheken des Qt-Toolkits für die Erstellung grafischer Benutzeroberflächen. Qt wird von der „The Qt Company“ entwickelt. [9]

3.6.5 Qt

Qt ist ein Widget-Toolkit zum Erstellen grafischer Benutzeroberflächen sowie plattformübergreifender Anwendungen. Diese können auf verschiedenen Software- und Hardwareplattformen wie Linux, Windows, MACOS, Android oder eingebettete Systeme mit geringer oder keiner Änderung der zugrunde liegenden Codebasis laufen, während sie immer noch eine native Anwendung mit nativen Funktionen und Geschwindigkeit sind.[10]

[8] Vgl. **[10]**
[9] Vgl. **[12]**
[10] Vgl. **[12]**

3.6.6 Qt Designer

Um die GUI zu designen, wird der Qt Designer verwendet.

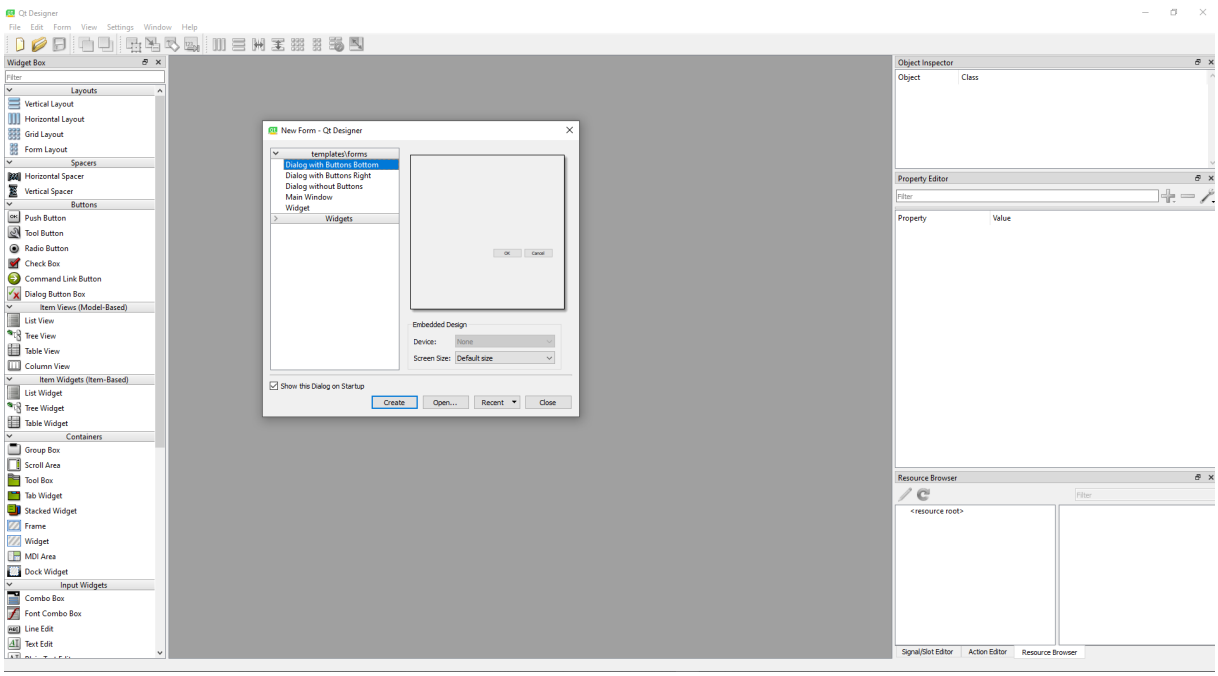


Abbildung 7: PyQt5-QtDesigner

Der Qt Designer ist eine Open-Source Variante des Programms Qt Creator, der „The Qt Company Firma“. Der Qt Creator wird von vielen verschiedenen Großkonzernen verwendet, um ihre Webseiten und grafischen Benutzeroberflächen zu gestalten.[11]
Mit dem Qt Designer Software kann eine GUI „gezeichnet“ werden. Der QtDesigner erstellt beim Speichern eine **.ui**-Datei, die auf XML basiert, die mithilfe von PyQt5 bzw. dem automatisch installierten Paket *pyuic5*[12] in eine **.py** Datei umgewandelt wird, um diese dann im Python-Programm verwenden zu können.
Die Umwandlung wird mit folgendem Befehl durchgeführt.

Codeausschnitt 3: pyuic5 bsp. captionpos

```
1 pyuic5 xyz.ui -o xyz.py
2
```

Im Qt Designer ist es weiterhin möglich, interne *Signale/Slots* zu verknüpfen, sodass für einfache Funktionen kein extra Python Code benötigt wird. Zum Beispiel wird ein Knopf mit einer Anzeige so verknüpft, dass sich die Anzeige bei einem Knopfdruck ändert.
Weiterhin können mit **QWidget** unabhängige Python Programme dargestellt werden.

[11] Vgl. **[13]**
[12] Vgl. **[15]**

3.6.6.1 Extensible Markup Language (XML)

XML ist eine Auszeichnungssprache zur Darstellung hierarchisch strukturierter Daten im Format einer Textdatei. Im Fall des Qt Designers, wird XML verwendet, da es wie HTML (Hypertext Markup Language) die Möglichkeit bietet, eine Struktur aufzubauen, die dann mit CSS weiter designed werden kann. Jedoch können in XML, im Unterschied zu HTML, dafür eigene Befehle eingeführt werden, die sich der Qt Designer zu Nutzen macht. [13]

3.6.6.2 Cascading Style Sheets (CSS)

Bei CSS handelt es sich um eine Gestaltungs- und Formatierungssprache, mit der das Aussehen von HTML und XML basierten Programmen bestimmt wird. Im Fall des QtDesigners, besitzt jeder Block eigene Befehle, wie dieser Block zu welchem Zeitpunkt bzw. bei welcher Funktion aussehen soll.
Um zum Beispiel einen Knopf richtig zu färben, wird folgender Code eingefügt. [14]

```
1  QPushButton {
2      border: 2px solid rgb(52, 59, 72);
3      border-radius: 5px;
4      background-color: rgb(52, 59, 72);
5  }
6  QPushButton:hover {
7      background-color: rgb(57, 65, 80);
8      border: 2px solid rgb(61, 70, 86);
9  }
10 QPushButton:pressed {
11     background-color: rgb(35, 40, 49);
12     border: 2px solid rgb(43, 50, 61);
13 }
14 QPushButton:disabled{
15     color: gray
16 }
```

Codeausschnitt 4: CSS bsp.

3.6.7 PySerial

PySerial ist ein Python Modul, welches es ermöglicht, über die USB-Schnittstellen eines PCs per UART (Universal Asynchronous Receiver Transmitter) mit anderen Geräten, z.B. Mikrocontrollern zu kommunizieren. [15]

[13] Vgl. **[16]**
[14] Vgl. **[17]**
[15] Vgl. **[21]**

4 HARDWARE

4.1 Vorgaben

4.1.1 FPGA

Von der Fimra INNIO / Jenbacher wurde das FPGA Entwicklungsboard **AES-A7EV-7A50T-G** mit einem **Xilinx Artix 750t XC7A50T** FPGA bereitgestellt:

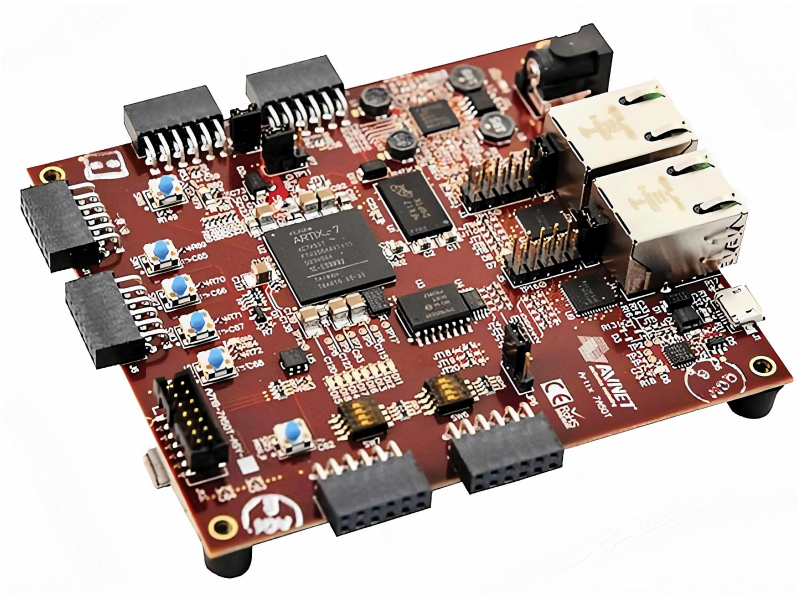


Abbildung 8: FPGA-Entwicklungsboard-AES-A7EV-7A50T-G

Dieser besitzt folgende Features:

- Xilinx XC7A50T-1FTG256C
- 256 MB DDR3 SDRAM
- 32 MB of QSPI Flash
- Dual 10/100 Ethernet interface
- 6 Digilent kompatible Pmod Interfaces
  die 48 I/O Pins ermöglicht
- USB-UART interface
- 512b EEPROM mit SHA-256 authentication engine
- 16 AMS inputs
- 10 user LEDS
- 8 user DIP switches
- 5 user push-button switches
- 200 MHz LVCMOS oscillator (system clock)

4.1.1.1 Aufbau des Entwicklungsboards anhand eines Block Diagramms

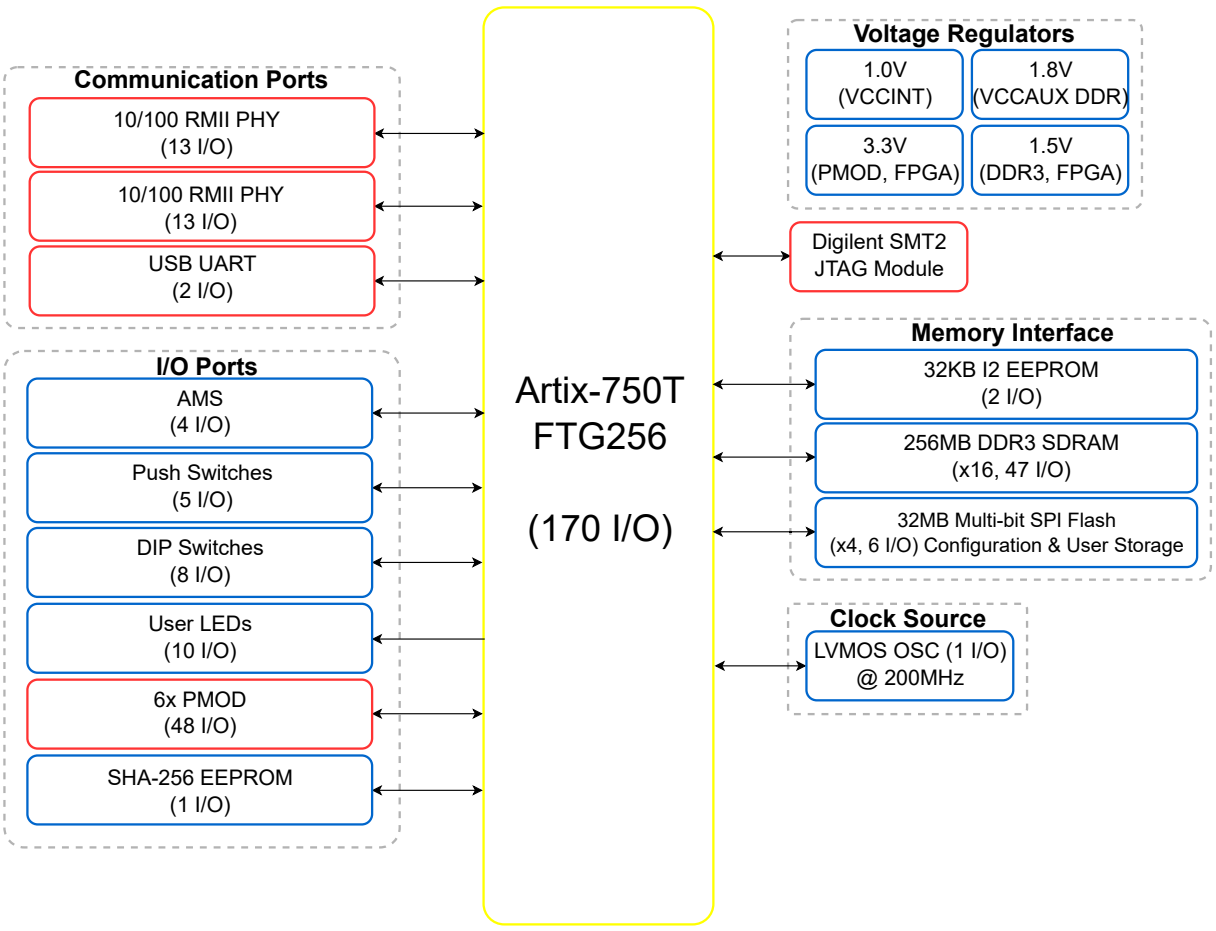


Abbildung 9: 7A50T-Block-Diagramm

Da dieses „Artix®-7 50T FPGA Evaluation Kit“ nicht mehr hergestellt wird, gibt es nur sehr limitierte Dokumentationen über das Entwicklungsboard.

4.1.2 Digital-Analog-Wandler Digilent Pmod

Weiters wurden 12 Digilent Pmod-DA4 Digital-Analog-Wandler bereitgestellt.

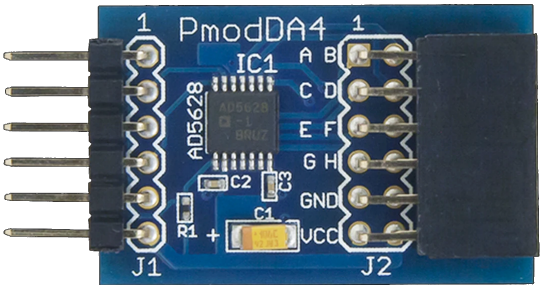


Abbildung 10: Pmod-DA4

An einen I/O-Port des Entwicklungsboards können 2 Pmods angeschlossen werden, diese besitzen wieder 8 Ausgangskanäle. Somit ergeben sich 6 Kanäle * 2 PMODS * 8 Kanäle = 96 Ausgangskanäle.
Auf dem ersten I/O-Port werden zusätzlichen zu den drei benötigten Signalen, eines Zylinders, das CAM-Reset und das Trigger-Signal generiert. Bei den restlichen I/O-Ports werden nur die folgenden drei Signale generiert.

- **Cylinder Pressure**(Zylinder Druck)
- **Knocksensor**(Klopfsensor)
- **Ignition Voltage**(Zündungs Spannung)

Die restlichen Pins werden für die Versorgung der PMOD's verwendet. [16] Dadurch ergibt sich, dass man damit 24 Zylinder simulieren kann.

[16] Da die benötigten Signale in der grafischen Benutzeroberfläche und im VHDL Code nur als englische Begriffe vorkommen, werden in der restlichen Dokumentation hauptsächlich die englischen Begriffe verwendet.

4.1.2.1 Pmod DA4-Schematic

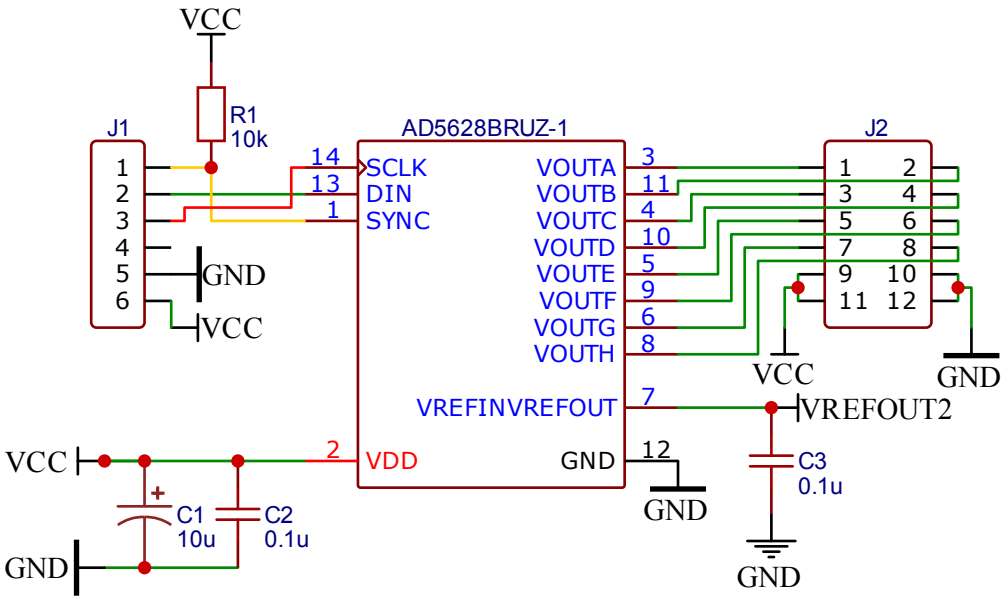


Abbildung 11: PMOD-DA4-Schematic

4.1.2.2 Pmod DA4-Eigenschaften

Der PMOD-DA4 besitzt folgende Eigenschaften:

- Acht unabhängige 12-Bit DAW
- Hochgeschwindigkeits DSP (Digital Signal Processor) kompatibel
- Power-Down-Funktion
- Energieeffizient
- 6-poliger Pmod-Anschluss mit SPI-Schnittstelle

Der Pmod DA4 verwendet einen **Analog Devices AD5628-1** um ein Signal-DAW bereitzustellen, das 8 Kanäle gleichzeitig ausgeben kann.
Der Pmod DA4 kommuniziert mit dem Host Board über das SPI Protokoll (SPI Erklärung siehe: 3.2). [17]

[17] Pmod DA4 Datenblatt: A.6

4.2 FPGA

4.2.1 FPGA-Resourcen

Die FPGA-Ressourcen können über die offizielle Webseite von Xilinx der Xilinx Artix 7 Serie abgerufen werden. Unter dem Product Selection Guide ist diese Information Verfügbar. [18]

Logic Resources	Logic Cells	52,160
	Slices	8,150
	CLB-Flip-Flops	65,200
Memory Resources	Maximum Distributed RAM	600kB
	Block RAM/FIFO (36kB each)	75
	Total Block RAM	2700
Clock Resources	CMTs (1 MMCM + 1 PLL)	5
I/O Resources	Maximum Single-Ended I/O	250
	Maximum Differential I/O Pairs	120
Embedded Hard IP Resources	DSP Slices	120
	PCIe Gen2	1
	Analog Mixed Signal (AMS) / XADC	1
	Configuration AES / HMAC Blocks	1
	GTP Transceivers (6.6 Gb/s MaxRate)	4
Speed Grades	Commercial Temp (C)	-1,-2
	Extended Temp (E)	2L,-3
	Industrial Temp (I)	-1,-2,-1L

Tabelle 1: FPGA-Ressourcen-des-XC7A50T

Wir können leider nicht direkt die Hardware-Last des VHDL-Codes berechnen, da wir keinen Zugriff auf die Vivado-Software von Xilinx mit der verbundenen Toolchain, die dann das Modell in wirklicher Hardware synthetisiert, haben.

[18] PDF im Anhang: A.6

4.3 Ideen für die Hardware

Da der FPGA und die Digital Analog Wandler direkt an SAFI's angeschlossen werden und dadurch nicht gegen Hochspannungen geschützt sind, wurden ein paar Ideen entwickelt, um die Hardware zu schützen.

4.3.1 Schutz des FPGAs durch Optokoppler

Der FPGA muss gegen Spitzenwerte von 40kV und 80A geschützt werden. Die Idee ist es den FPGA mit Optokoppler vor diesen Hochspannungen zu trennen. Jedoch wurde wegen Zeitmangel nur ein Blockschaltbild angefertigt.

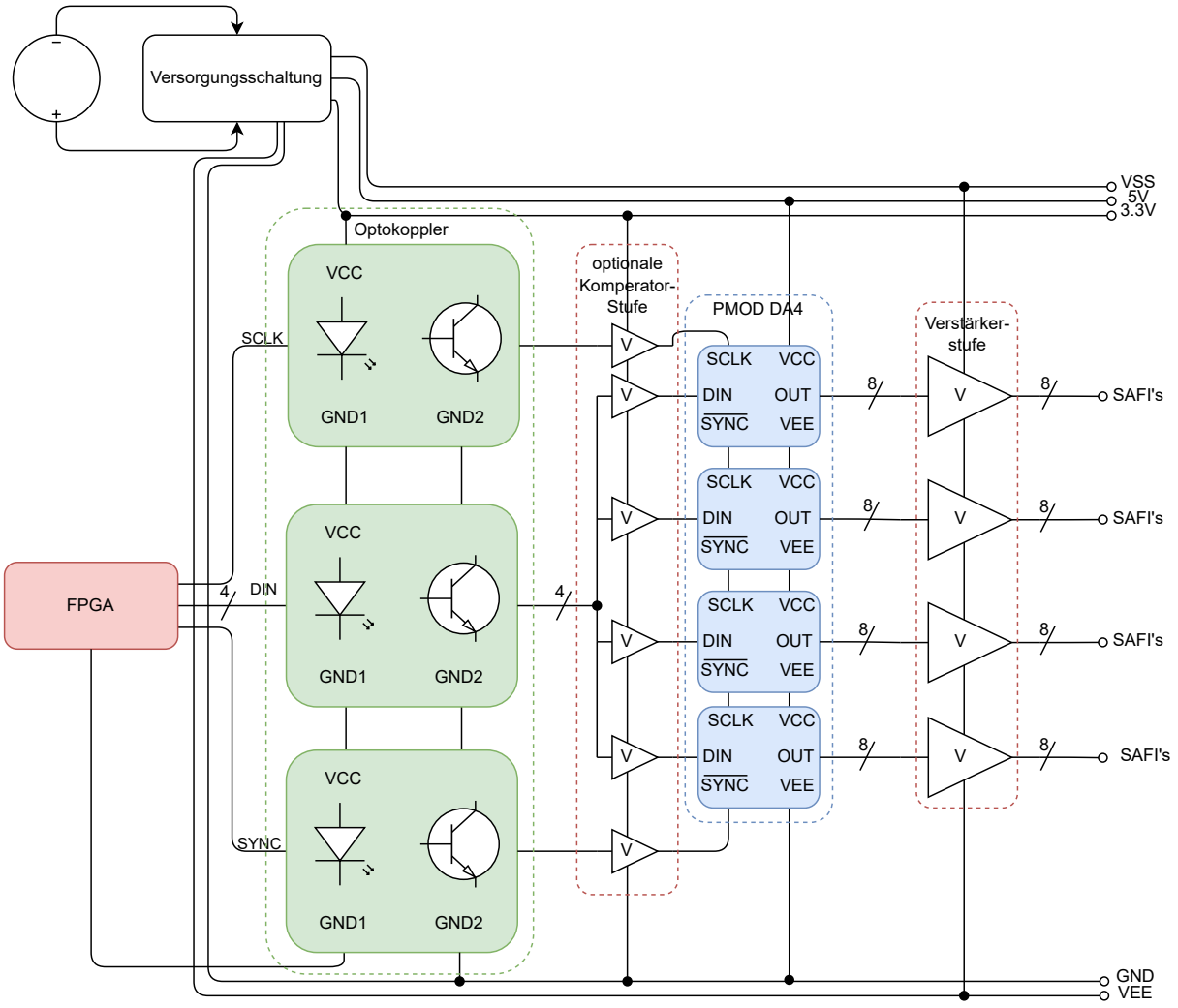


Abbildung 12: Blockschaltbild-Schutz des FPGAs mit Optokoppler

5 SOFTWARE

5.1 Vorgaben

Von der Firma INNIO / Jenbacher wurde die Excel Datei als Vorgabe bereitgestellt.

5.1.1 Excel-Datei

Die folgenden Bilder zeigen die Hauptseite der Excel und das zweite Fenster „Engine Configuration“. Im Fenster Engine Configuration finden alle Einstellungen statt.

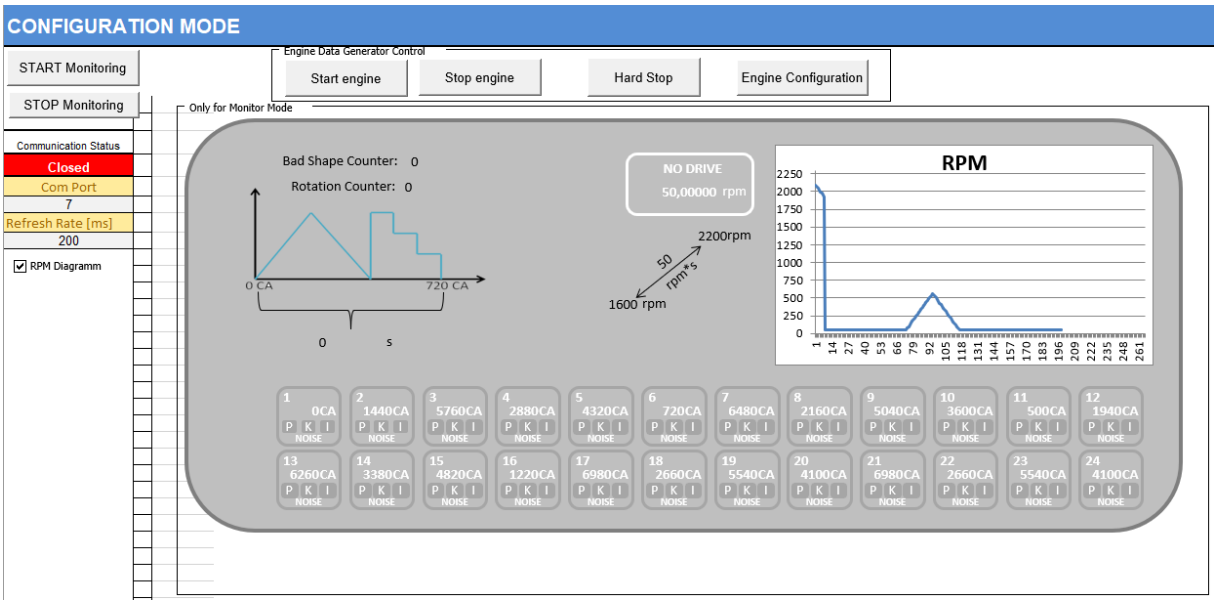


Abbildung 13: Vorgaben-Excel-1

Die Abbildung 12 zeigt die Hauptseite der Excel-Datei.
Diese zeigt an:

- Wie viele Zylinder aktiv sind
- Bei welchem Zündwinkel die Zylinder gezündet werden
- die RPM (Revolutions per Minute) also Drehzahlanzeige
- den Bad Shape Counter - dieser zählt, wie oft die beabsichtigten fehlerhaften Kurven im Signalverlauf vorgekommen sind
- Rotation Counter - dieser zählt wie viele Motorzyklen durchgelaufen sind.

Auf der Hauptseite können folgende Einstellungen getroffen werden:

- COM Port: An welchem COM-Port der FPGA angeschlossen ist
- Refresh Rate: die Übertragungsrate der UART als $\frac{1}{\text{Baudrate}}$
- ob das RPM Diagramm aktiviert sein soll.

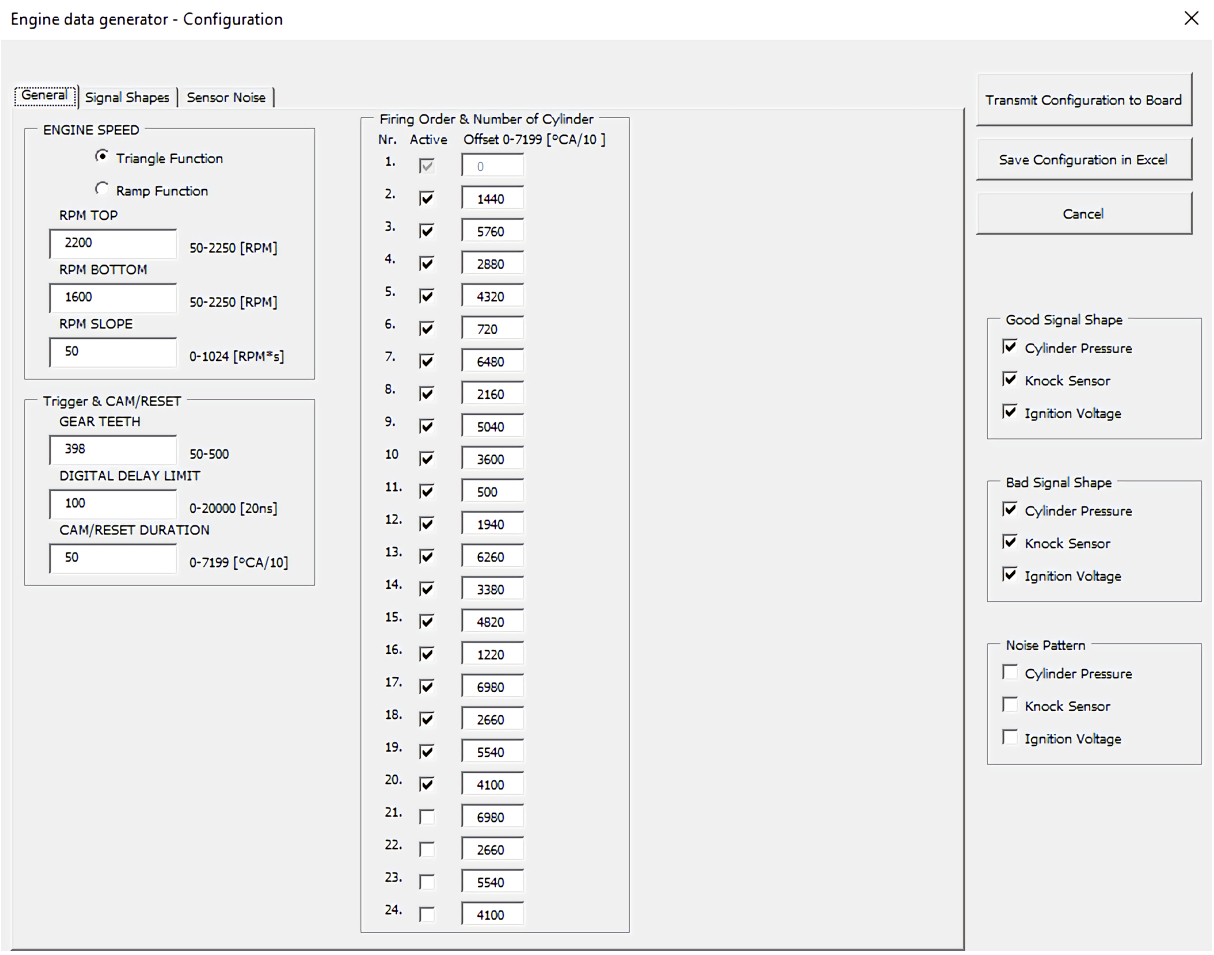


Abbildung 14: Vorgaben-Excel-2

Die wichtigsten Einstellungen finden im Fenster „Engine Configuration“ statt. Dort gibt es 3 Unterkategorieren, General, Signal Shapes, Sensor Noise. Es kann auf jeder Seite die Konfiguration dem FPGA übermittelt, die Konfiguration gespeichert, die Good und Bad Signal Shapes und das Noise Pattern aktiviert und deaktiviert werden. In der Abbildung 14 werden die General Einstellungen gezeigt, die bestehen aus:

- Welche Zylinder aktiviert sind
- Den Zündwinkel der Zylinder
- Maximal- Minimaldrehzahl
- Steigung der Drehzahl
- Gear Teeth
- Digital Delay limit
- CAM/Reset duration

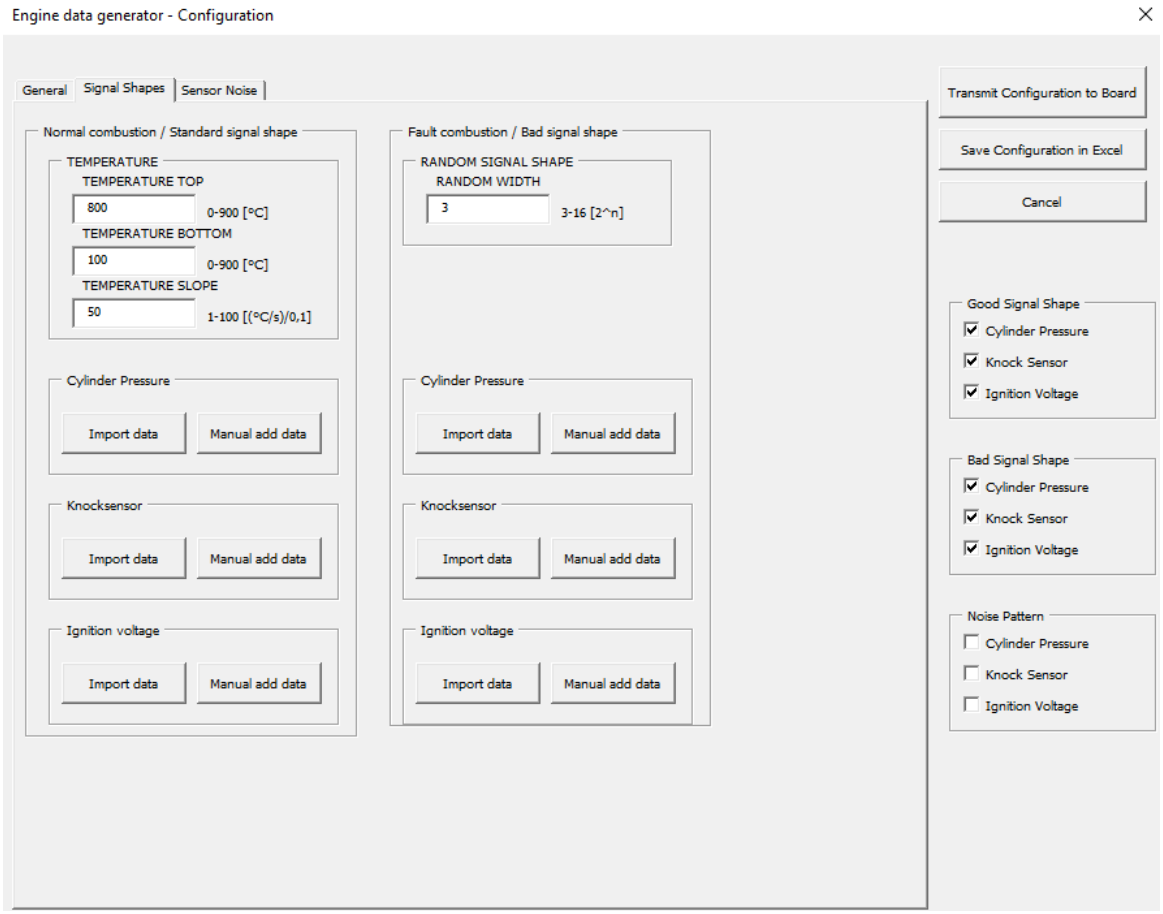


Abbildung 15: Vorgaben-Excel-3

Unter Signal Shapes gibt es folgende Einstellungen:

- Maximale- Minimale Abgastemperatur
- Steigung der Abgastemperatur
- Importierung selbsterstellter Signale
- Random Signal Shape

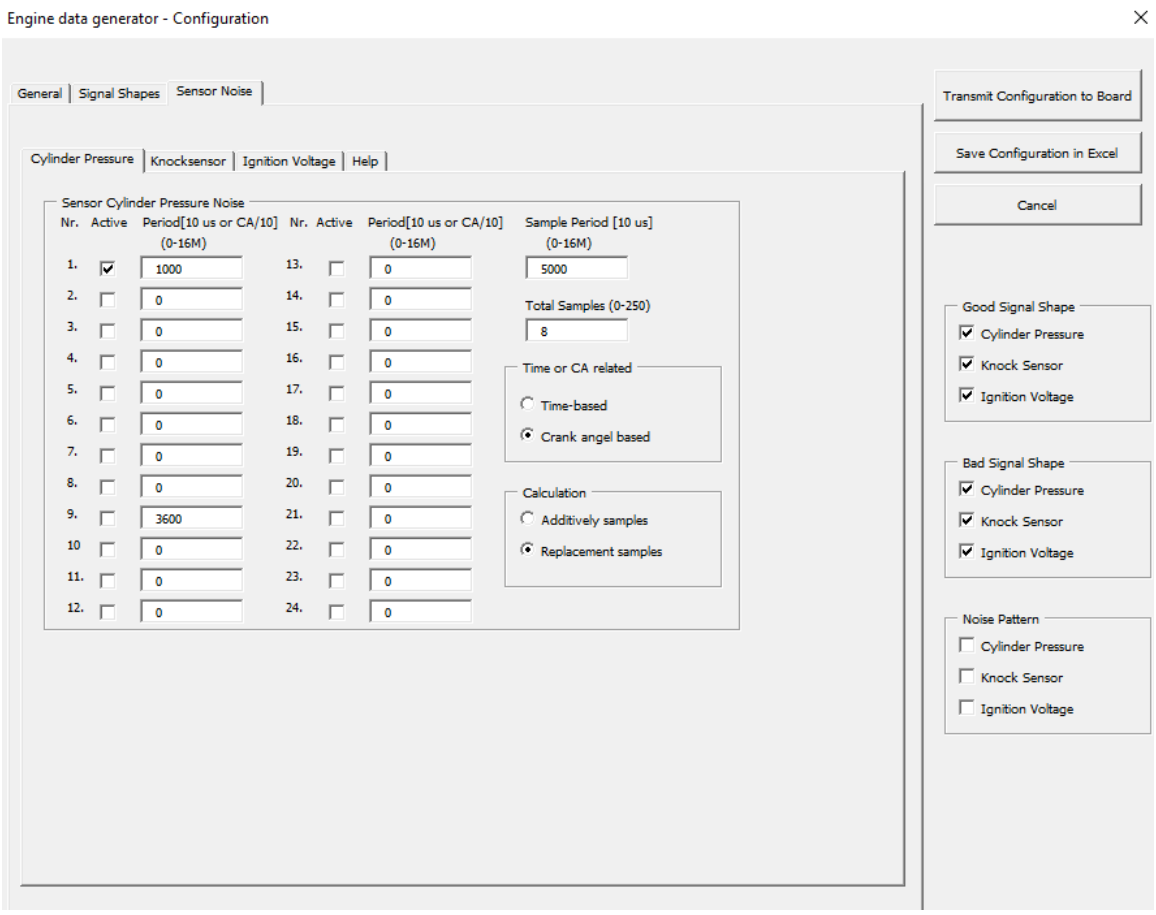


Abbildung 16: Vorgaben-Excel-4

Unter Sensor Noise gibt es nochmal die einzelnen Signale Cylinder Pressure, Knocksensor und Ignition Voltage als Unterkategorieren. Diese Einstellungen bewirken, dass der Signalgenerator realistische und zufällige Störungen an den jeweiligen aktivierten Zylinder ausgibt.

- Active: Auf welchen Zylinder Störungen aktiviert sind.
- Period: Wie lange eine Störung auftreten soll.
- Sample Period
- Total Samples
- Time or CA related: Auf welchem Bezugspunkt sollen die Störungen berechnet werden.
- Calculation: Wie sollen die Störungen berechnet werden.

All diese Funktionen werden von der Python GUI übernommen und teilweise verbessert. Zusätzlich wird in der GUI eine Darstellung der Signale implementiert und die Möglichkeit, die Drehzahl während des Motorzyklus zu verändern.

5.1.2 VHDL-Code der TU Wien

Da die TU-Wien Eigentümer dieses Codes ist, dürfen wir diesen nicht zeigen. Jedoch dürfen wir unsere Ausarbeitung, Erklärung und Dokumentation wiedergegeben.

5.2 VHDL

5.2.1 Blockschaltbild

Der VHDL-Code ist aus mehreren Komponenten und Prozessen aufgebaut. Diese werden im Topmodul alle verbunden.

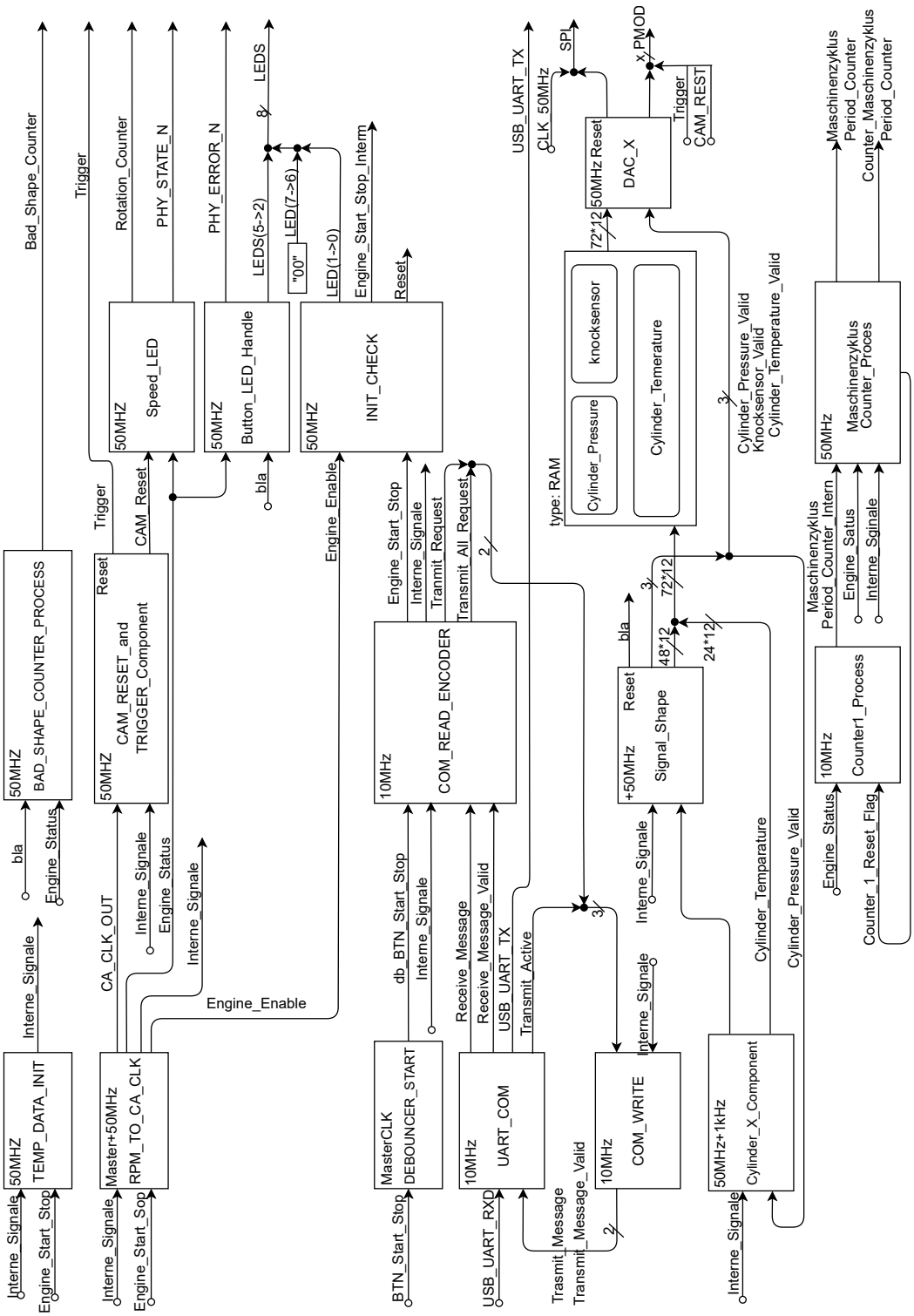


Abbildung 17: VHDL-Blockschaltbild

5.2.2 Aufteilung

Die wichtigsten Teile des VHDL-Code können in UART-Kommunikation, Signal-Generierung, Rauschsignal-Generierung und SPI-Kommunikation unterteilt werden. Die hier nicht besprochenen Komponenten und Prozesse sind nicht relevant für die Funktionsweise des VHDL-Codes und werden daher nicht aufgeführt.

5.2.3 UART-Kommunikation

Die UART-Kommunikation ist die Kommunikation, die mit dem PC stattfindet. Diese wird mit dem COM_READ_ENCODER-Prozess, COM_WRITE-Prozess und der UART_COM-Komponente, die wiederum aus der UART_TX-Komponente und der UART_RX-Komponente besteht, zusammengesetzt.
Die Kommunikation wird mithilfe von 32 Bit großen Datenpaketen implementiert. Hier werden die ersten 7 Bit für die „Escape Sequence“ verwendet, die angibt welcher Datentyp übertragen wird und die restlichen Bits werden für die wirklichen Daten verwendet.

5.2.3.1 UART_RX

Die UART_RX-Komponente empfängt die Bits, die seriell über die UART vom PC gesendet werden und verwandelt diese in 32-Bit (4-Byte) große Datenpakete die, dann an den COM_READ_ENCODER weitergeleitet werden.

5.2.3.2 COM_READ_ENCODER

Die COM_READ_ENCODER-Komponente analysiert die, von der UART_RX-Komponente, empfangenen Datenpakete und verändert entweder interne Werte oder befiehlt der COM_WRITE-Komponente, Daten an den PC zu senden.
Eine Tabelle der Escape-Seqeunces und den damit verbundenen Datentypen folgt auf der nächsten Seite:

Datentyp	Datenbits	Gruppe	Escape Sequences	Escape Sequence [bin]
RPM_BOTTOM, RPM_TOP	24	Nein	1	0000001
Acceleration	25	Nein	3	0000011
GEAR_TEETH	25	Nein	4	0000100
DIGITAL_DELAY_LIMIT	25	Nein	5	0000101
CAM_RESET_DURATION	25	Nein	6	0000110
CA_OFFSET_CYL	24	Ja	7-30	0000111
T_high	2	Nein	31	0011111
T_low	12	Nein	32	0100000
T_Speed	25	Nein	33	0100001
random_width	25	Nein	34	0100010
RAM 0 bis 6	25	Ja	38-44	0100110
Zylinder-Druck/Klopfsensor: Rausch-Periode	24	Ja	45-68	0101101
Zylinder-Druck: Rausch-Aktiv-Flaggen	24	Nein	69	1000101
Zylinder-Druck: Rausch-Sample-Periode	24	Nein	70	1000110
Zylinder-Druck: Rausch-Sample-Count	8	Nein	71	1000111
Zylinder-Druck: Rausch-Config	2	Nein	72	1001000
RAM 7	25	Nein	73	1001001
Klopfsensor: Rausch-Aktiv-Flaggen	24	Nein	74	1001010
Klopfsensor: Rausch-Sample-Periode	24	Nein	75	1001011
Klopfsensor: Rausch-Sample-Count	8	Nein	76	1001100
Klopfsensor: Rausch-Config	2	Nein	77	1001101
Zündspannung: Rausch-Periode	24	Ja	78-101	1001110
RAM 8	25	Nein	102	1100110
Zündspannung: Rausch-Aktiv-Flaggen	24	Nein	103	1100111
Zündspannung: Rausch-Sample-Periode	24	Nein	104	1101000
Zündspannung: Rausch-Sample-Count	8	Nein	105	1101001
Zündspannung: Rausch-Config	2	Nein	106	1101010
TransmitAllRequest	0	Nein	124	1111100
TransmitRequest	0	Nein	125	1111101
Engine_Start_Stop, Hard_Stop	2	Nein	127	1111111
Engine_Start_Stop	1	Nein	127	1111111

Tabelle 2: VHDL- Receive Befehle

5.2.3.3 UART_TX

Die UART_TX-Komponente sendet die Datenpakete von der COM_Write-Komponente seriell über die UART.

5.2.3.4 COM_WRITE

Die COM_WRITE-Komponente wird mithilfe der COM_READ_ENCODER-Komponente gesteuert. Die COM_WRITE-Komponente kann, auf Befehl von der COM_READ_ENCODER-Komponente, mithilfe der UART_TX-Komponente Daten an den PC senden.

Datentyp	Datenbits	Gruppe	Escape-Sequence	Escape-Sequence binär
RPM_BOTTOM	25	Nein	1	0000001
RPM_TOP	25	Nein	2	0000010
GEAR_TEETH	25	Nein	4	0000100
DIGITAL_DELAY_LIMIT	25	Nein	5	0000101
CAM_RESET_DURATION	25	Nein	6	0000110
CA_OFFSET_CYL	24	Ja	7 bis 30	0000111
T_high	12	Nein	31	0011111
T_low	12	Nein	32	0100000
T_Speed	25	Nein	33	0100001
random_width	25	Nein	34	0100010
CYL_ACTIVE	24	Nein	35	0100011
Engine_Status	4	Nein	36	0100100
Bad_Shape_Counter	25	Nein	45	0101101
Rotation_Counter	25	Nein	47	0101111
Maschinenzyklus_ Period_Counter	25	Nein	48	0110000
Counter_Maschinenzyklus_ Period_Counter	25	Nein	49	0110001
Zylinder-Druck Rauschen-Aktiv-Flaggen	24	Nein	69	1000101
Klopfsensor Rauschen-Aktiv-Flaggen	24	Nein	74	1001010
Zündspannung Rauschen-Aktiv-Flaggen	24	Nein	103	1100111
RPM_STATUS	28	Nein	120 bis 127	1111XXX

Tabelle 3: VHDL- Transmit Befehle

5.2.4 Signal-Generierung

Die Signal-Generierung erfolgt mithilfe der „signal_shape“-, „rpm_to_CA_CLK“-, „CAM_RESET_generation“- und „cylinder“-Komponenten.
Durch Zeitmangel und Komplexität der Komponente, wurde der Vorgang der Signal-Generation nicht genauer analysiert.

5.2.5 Rauschsignal-Generierung

Die Rauschsignal-Generierung wird mithilfe der „signal_noise“-Komponente realisiert. Diese generieren, abhängig davon, wie die COM_READ_ENCODER-Komponente die Parameter eingestellt hat, Rauschsignale. Die mit den Signalen der Signal-Generierung überlagert werden.
Durch Zeitmangel und Komplexität der Komponente, wurde der Vorgang der Rauschsignal-Generierung nicht genauer analysiert.

5.2.6 SPI-Kommunikation

Die SPI-Kommunikation wird mithilfe von 12 „dac_PMOD“-Komponenten realisiert. Jede Komponente steuert einen angeschlossenen PMOD mit den Signalen, die von der Signalgeneration berechnet werden.

5.3 Python

5.3.1 Anforderung des Python Programms

Das Python Programm soll übersichtlich und einfach zu verstehen gestaltet werden.
Sodass, wenn nötig, noch weiter Methoden, ohne große Hinderungen, hinzugefügt werden können.
Die GUI soll einfach zu verwenden sein und alle Methoden der Excel, wenn möglich in verbesserter Form, beinhalten.
Zusätzlich wird das Programm nach Fertigstellung in ein **.exe**-Programm (executable), also in ein normal ausführbares Programm, mit Hilfe des Pakets **PyInstaller** [19], umgewandelt.

5.3.2 Lösungsansatz

Durch die Anforderungen ergibt sich folgender Lösungsansatz.
Das Python Programm wird aufgeteilt in **Front-End** (GUI mit QTDesigner) und **Back-End** (PySerial verbunden mit dem FPGA).
Diese werden in weitere Komponenten unterteilt.

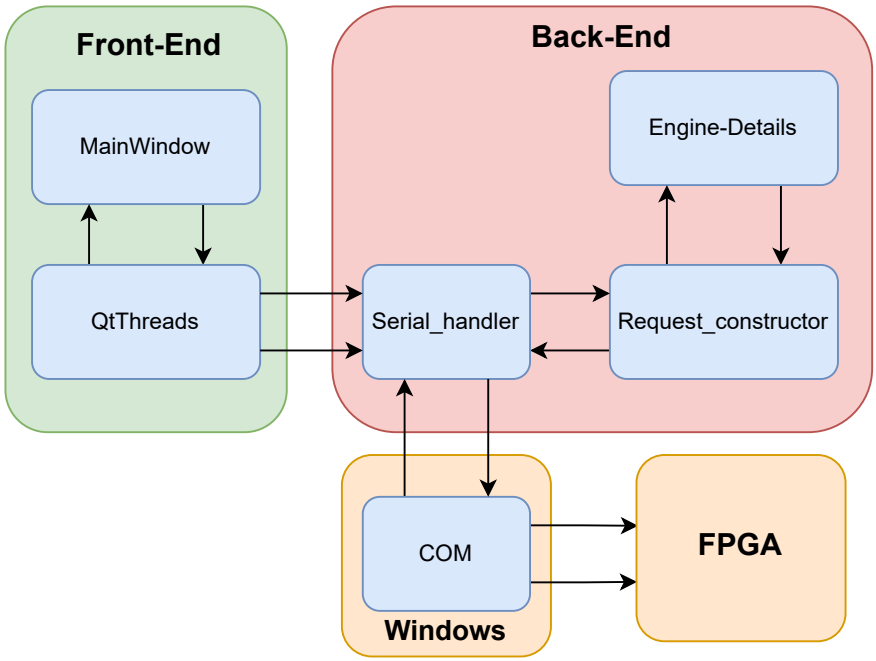


Abbildung 18: Python-Block-Diagramm

Dieser Aufbau kann auch als die Ordnerstruktur des Python Programms angesehen werden und wird in Abbildung 19 noch genauer dargestellt.

[19] Vgl. **[18]**

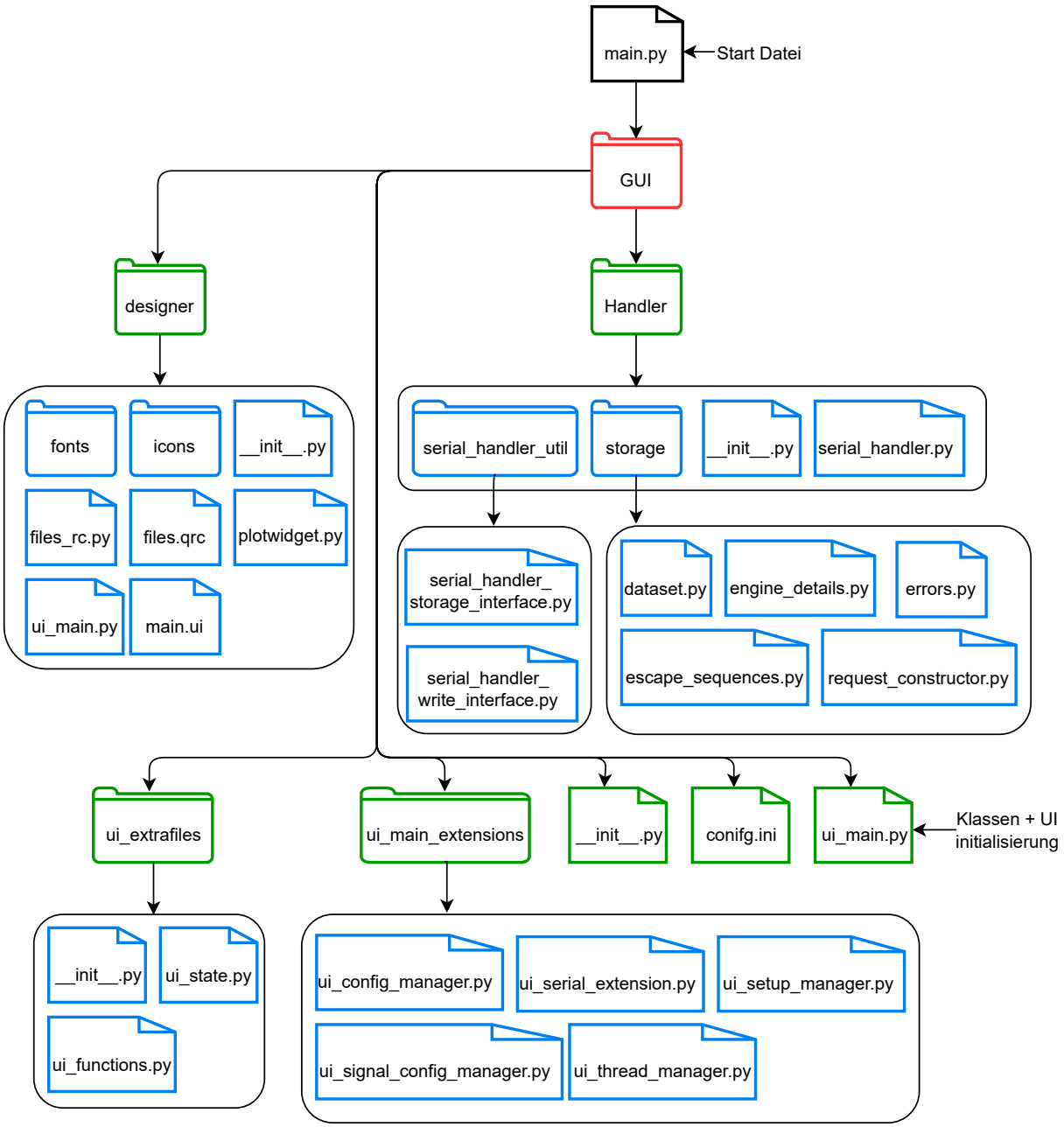


Abbildung 19: Python-Datei-Struktur

Die Ordnerstruktur ist essentiell für ein übersichtliches Programm und für die Umwandlung in ein „.exe“-Programm. Hierfür ist es erforderlich, eine Datei *main.py*, hierarchisch gesehen, über den gesamten Aufbau zu haben, die letztendlich die Applikation startet.

Die grünen Ordner sind Pakete, die die darunter stehenden blauen Dateien beinhalten. Die blauen Dateien sind die einzelnen Module.

Um diese Aufteilung zu erreichen und dafür zu sorgen, dass die einzelnen Dateien miteinander richtig kommunizieren, wird das gesamte Python Projekt als ein *Package* und die einzelnen Dateien als *Module* realisiert werden. Das wird erreicht, indem man in jedem Unterordner eine leere Datei mit dem Namen **__init__.py** hinzufügt. Dadurch können Klassen und Methodeen untereinander, relativ, importiert werden.[20]

Ein weitere wichtiger Teil um Daten untereinander auszutauschen, ist das **self** und die Vererbung der Klassen.
Auf die einzelnen Komponenten wird auf den nächsten Seiten vertieft eingegangen.

5.3.3 Umsetzung

Verwendete Programmierungsumgebung und Pakete

Für die Programmierung wurde der Text-Editor Visual Studio Code Version 1.53.2 und Atom 1.54.0 verwendet.
Es wurde die zu der Zeit neueste Python-Version 3.9.2 mit folgenden Paketen verwendet:

Paket	Version
PyQt5	5.15.2
PySerial	3.5
Pyinstaller	4.2
Numpy	1.19.3
Pyqtgraph	0.11.0
Qt Designer	5.11.1
pylint	2.7.0

Tabelle 4: Verwendete Python Pakete

5.3.3.1 Fluss- und Klassendiagramm

Das Flussdiagramm zeigt den vereinfachten Aufbau der Threads und wie diese miteinander kommunizieren und Daten mit dem FPGA ausgetauscht werden.

Das Klassendiagramm wurde mithilfe von dem Paket *pyreverse* erstellt. Da dieses jedoch viel zu groß ist, um es auf einem DIN A4 Blatt darzustellen, wurde es verkleinert erneut gezeichnet. Somit sind in der Graphik die meisten Methoden der Klassen nicht enthalten.

[20] Vgl.**[19]**

Flussdiagramm

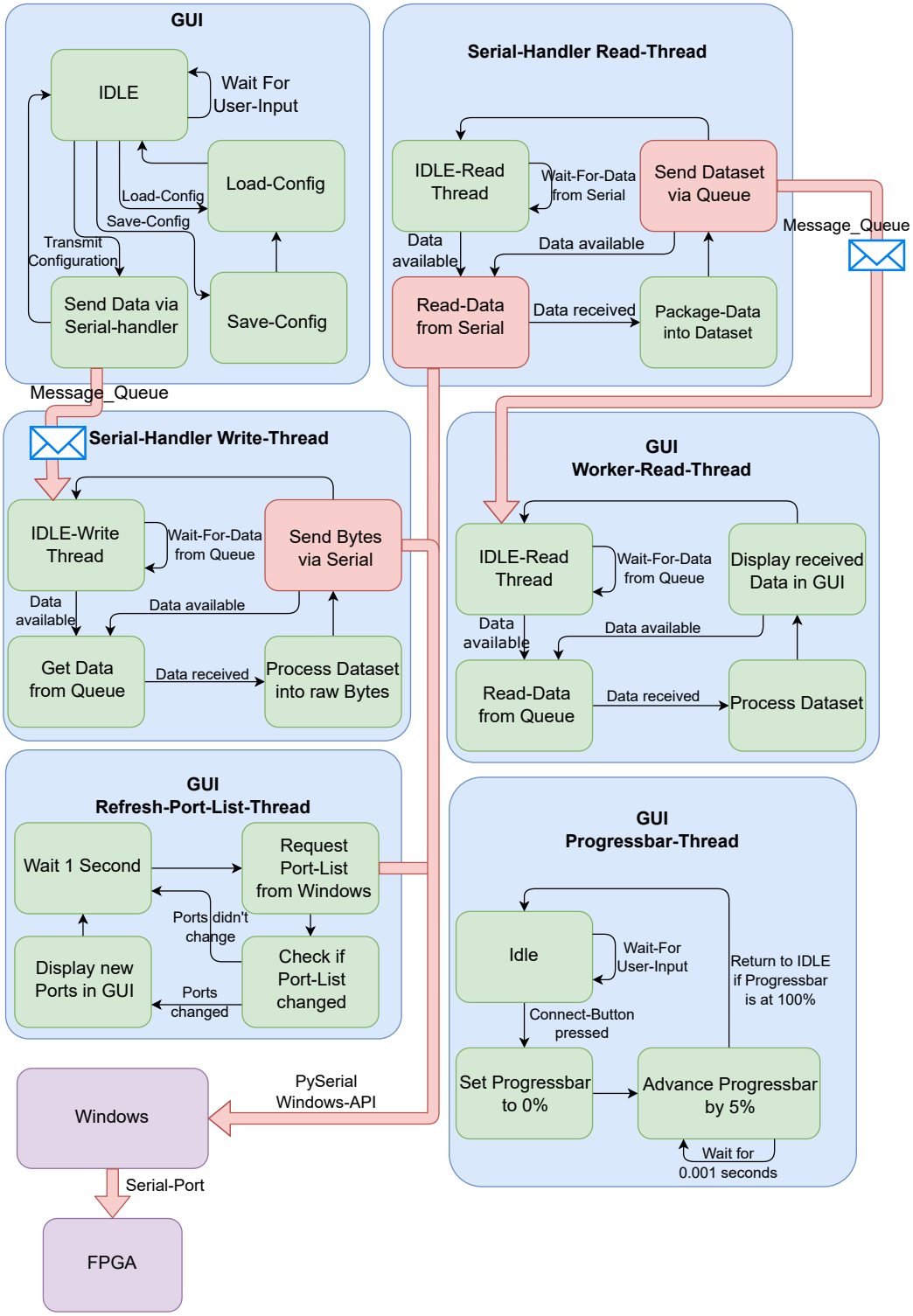


Abbildung 20: Flussdiagramm

Klassendiagramm

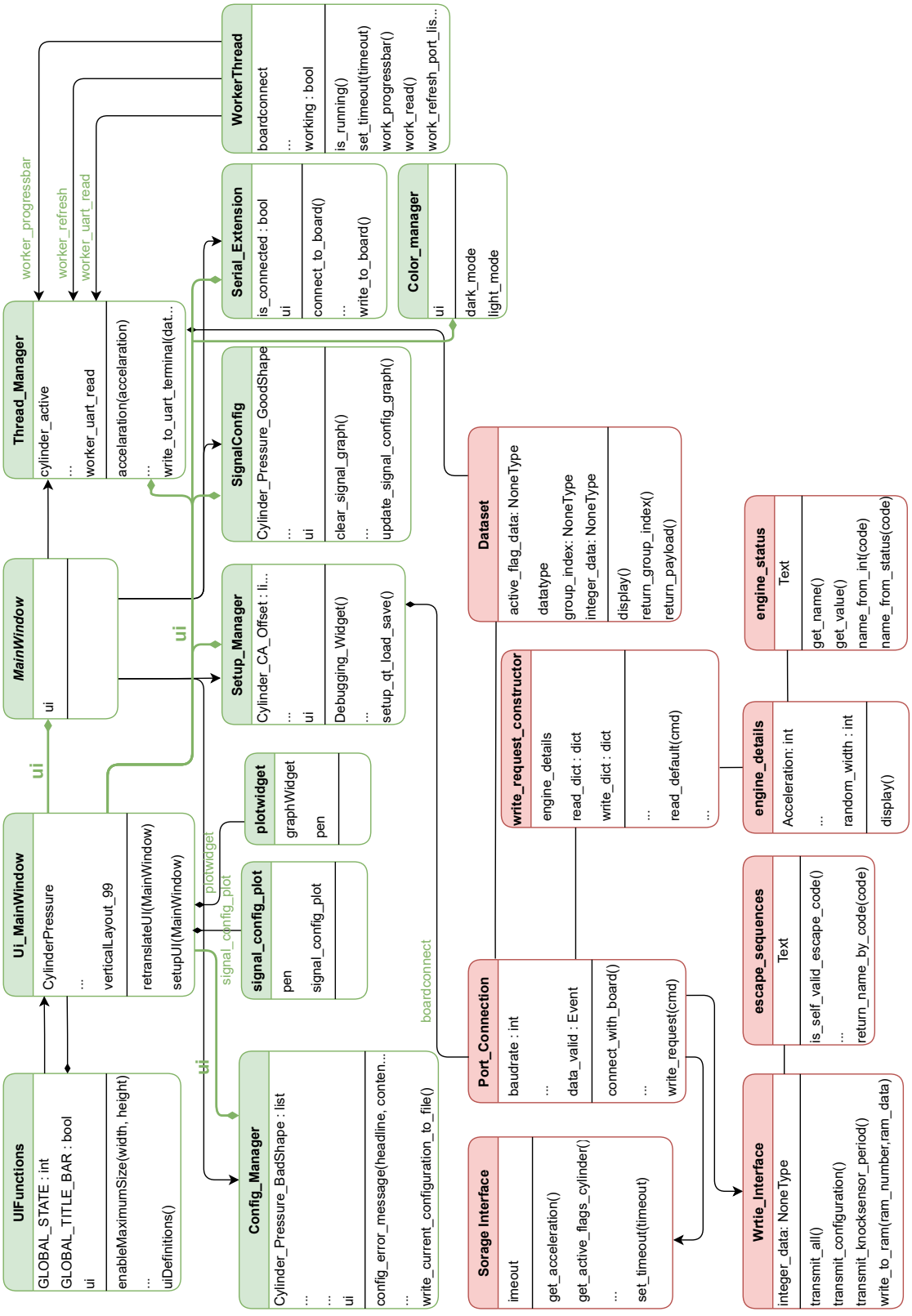


Abbildung 21: Klassendiagramm

Initialisierung

Wie bereits erwähnt, muss es für die Umwandlung in eine Windows Applikation eine Startdatei geben.
Diese Datei erfüllt in diesem Fall auch noch die Aufgabe, Python Pakete zu installieren, falls diese noch nicht vorhanden sind. Weiterhin wird die Installation des Treibers für den FPGA passend zur Betriebssystemarchitektur, falls noch nicht vorhanden, ausgeführt. Wenn dies erledigt ist, wird die GUI durch den Methodenaufruf *run_app()* gestartet.
Diese Methode wird in der Haupt-Initialisierungsdatei der grafischen Benutzeroberfläche *ui_main.py* definiert.

main.py

```
1  if __name__ == "__main__":
2    import os, platform, subprocess
3    try:
4      import serial
5      import PyQt5
6      import pyqtgraph
7    except ImportError:
8      os.system("python -m pip install pyserial")
9      os.system("python -m pip install PyQt5")
10     # Install a specific Verion of Numpy
11     # because PyQT5 installs the wrong version automatically!
12     os.system("python -m pip install numpy==1.19.3")
13     os.system("python -m pip install pyqtgraph")
14     import serial
15     import PyQt5
16     import pyqtgraph
17   #Pseudo-Code Begin
18     if(FPGA_drive != installed):
19       os.system(f"driver\\CP210xVCPInstaller_{os_x6486}.exe")
20   #Pseudo-Code End
21   from gui.ui_main import run_app
22     run_app()
23
```

Codeausschnitt 5: GUI-main.py

ui_main.py

Die Datei ui_main.py ist die Haupt-Initialisierungsdatei. Sie beinhaltet die Initialisierung der GUI und die Verknüpfung der Klassen durch Vererbung. Diese Vererbung basiert darauf, dass die ui_main.py die Methoden der Vaterklassen aufrufen kann.[21] Um die grafische Benutzeroberfläche zu starten, wird die Methode *run_app()* definiert. Diese initialisiert die QApplication und definiert die MainWindow-Klasse. Die MainWindow-Klasse verknüpft alle anderen Klassen und initialisiert die *ui_main.py* Datei des Designers, welche die GUI enthält. Ebenfalls wird festgelegt, falls ein Betriebsystembefehl auftritt, der die Schließung der GUI anfordert, wird die Methode *close_app()* ausgeführt, um eine sichere Schließung des Programms zu gewährleisten.
Weiterhin wird die *setup_mainwindow()* Methode aus dem *ui_setup_manager* ausgeführt, die jegliche Initialisierungen startet.

```
1  # Exit the Application safely, with exit button
2  def close_app(app, window):
3    window.write_config()  #save all settings in config
4    app.exec_()        #close application
5  # Create QApplication
6  def run_app():
7    app = QApplication(sys.argv)
8    winddow = MainWindow()
9    sys.exit(close_app(app, window)) #for closing the app, when system
       exit command gets raised
10 # Create Mainwindow-Class with multiple Parent-Classes to inherit
       their Methods
11 # This lets us write Code for this class in other Files and break our
       Code into Chunks
12 class MainWindow(QMainWindow, Config_Manager, Setup_Manager,
       Thread_Manager, Serial_Extension, SignalConfig):
13   # Call QMainWindow-Parent-Constructor and initialize UI-Elements
14   def __init__(self):
15     QMainWindow.__init__(self)
16   # Ui_MainWindow is created class from QTDesigner
17     self.ui = Ui_MainWindow()
18     self.ui.setupUi(self) # Initialize class
19   # Initializing Methode from ui_setup_manager.py
20     self.setup_mainwindow()
21     self.show() # "render" GUI
22
```

Codeausschnitt 6: GUI-ui_main.py-Klassen Parenting

[21] Vgl. **[26]**

QApplication

QApplication[22] initialisiert die Applikation und besitzt folgende Eigenschaften:

- Es initialisiert die Anwendung mit den Desktop-Einstellungen des Benutzers wie palette(), font() und doubleClickInterval(). Diese Eigenschaften werden verfolgt, falls der Benutzer den Desktop global ändert, z. B. über eine Art Systemsteuerung.
- Es führt eine Ereignisbehandlung durch, dh es empfängt Ereignisse vom zugrunde liegenden Fenstersystem und sendet sie an die entsprechenden Widgets. Mit sendEvent() und postEvent() können Sie Ihre eigenen Ereignisse an Widgets senden.
- Es analysiert allgemeine Befehlszeilenargumente und legt den internen Status entsprechend fest. Weitere Informationen finden Sie in der Konstruktordokumentation unten.
- Es definiert das Erscheinungsbild der Anwendung, das in einem QStyle-Objekt gekapselt ist. Dies kann zur Laufzeit mit setStyle() geändert werden.
- Es ermöglicht die Lokalisierung von Zeichenfolgen, die für den Benutzer über translate () sichtbar sind.
- Es bietet einige magische Objekte wie den Desktop() und die Zwischenablage().
- Es kennt die Fenster der Anwendung. Sie können mit widgetAt() fragen, welches Widget sich an einer bestimmten Position befindet, eine Liste der topLevelWidgets() und closeAllWindows() usw. abrufen.
- Es verwaltet die Mauszeigerbehandlung der Anwendung, siehe setOverrideCursor().

5.3.3.2 Front-End

Wie bereits bei 3.6.4 erwähnt, wird für die Erstellung der grafischen Benutzeroberfläche PyQt5 mit dem Qt Designer verwendet.
Um eine Idee zu bekommen, wie man den Qt Designer verwendet und wie man damit eine gut aussehende und funktionelle GUI erstellt, wird einerseits die offizielle Dokumentation[23]herangezogen, andererseits wurden vom GitHub Projekt von Wanderson-Magalhaes: „Simple_PySide_Base“ [24] grundsätzliche Funktionen interpretiert.
Dieses GitHub-Projekt wird mit PySide statt mit PyQt5 realisiert. Der Unterschied der beiden Module liegt in der Lizenzierung und der Definitionen und Methoden der API-Aufrufe.

[22] Vgl. **[22]**
[23] Vgl. **[23]**
[24] Vgl. **[24]**

5.3.3.2.1 Designer

Der Unterordner Designer, beinhaltet alle Methoden und Definitionen, die der QT-Designer verwendet und erzeugt. Verwendete Daten sind die Icons, Fonts und die Datei „files.qrc“. In der *files.qrc* Datei sind die Pfade der Icons gespeichert, sodass diese im Qt-Designer relativ importiert werden können.
Wie bereits bei 3.6.6 erwähnt wurde, ist diese Datei in XML geschrieben und beinhaltet die gesamte Information der entworfenen GUI des Qt Designers.
Dadurch, dass diese Datei das Design beinhaltet, besitzt diese einen 30.000 Zeilen Code und somit kann nur ein kleiner Ausschnitt gezeigt werden.

main.ui

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <ui version="4.0">
3      <author>Fabio Plunser</author>
4      <class>MainWindow</class>
5      <widget class="QMainWindow" name="MainWindow">
6      <property name="geometry">
7      <rect>
8      <x>0</x>
9      <y>0</y>
10     <width>1100</width>
11     <height>810</height>
12     </rect>
13     </property>
```

Codeausschnitt 7: main.ui-Anfang

Die **main.ui** wird dann wie bei 3.6.6 in eine **.py** Datei umgewandelt. Diese besitzt dann als Pythoncode die gesamte Information, der im Qt Desinger entworfenen, grafischen Benutzeroberfläche.

designer_ui_main.py

```
1  from PyQt5 import QtCore, QtGui, QtWidgets
2
3  class Ui_MainWindow(object):
4    def setupUi(self, MainWindow):
5      MainWindow.setObjectName("MainWindow")
6      MainWindow.resize(1100, 810)
7      sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.
     Expanding, QtWidgets.QSizePolicy.Expanding)
8      sizePolicy.setHorizontalStretch(0)
9      sizePolicy.setVerticalStretch(0)
10     sizePolicy.setHeightForWidth(MainWindow.sizePolicy().
     hasHeightForWidth())
11     MainWindow.setSizePolicy(sizePolicy)
12     MainWindow.setMinimumSize(QtCore.QSize(1100, 800))
13
```

Codeausschnitt 8: designer_ui_main.py

Weiterhin besitzt diese wichtige Imports:

```
14329 from .plotwidget import plotwidget
14330 from . import files_rc
14331
```

Codeausschnitt 9: designer_ui_main.py

Import Erklärungen

Importiert wird die „files.qrc“-Datei, die ebenfalls mithilfe von pyuci5 in eine Pythondatei umgewandelt wurde. Weiterhin werden aus der Datei plotwidget.py zwei Graphen importiert, diese werden als *Promoted Widgets* verwendet. Einer der Graphen stellt die Drehzahlkurve dar, der andere wird verwendet, um im Editor die importierten Signale zu verändern.

Promoted Widgets

Im QtDesigner können eigene *Widgets*, mit eigenen Funktionen, implementiert werden. Diese werden Promoted Widgets genannt.
Sie sind essentiell eigene Klassen, die im QtDesigner integriert werden.[25]

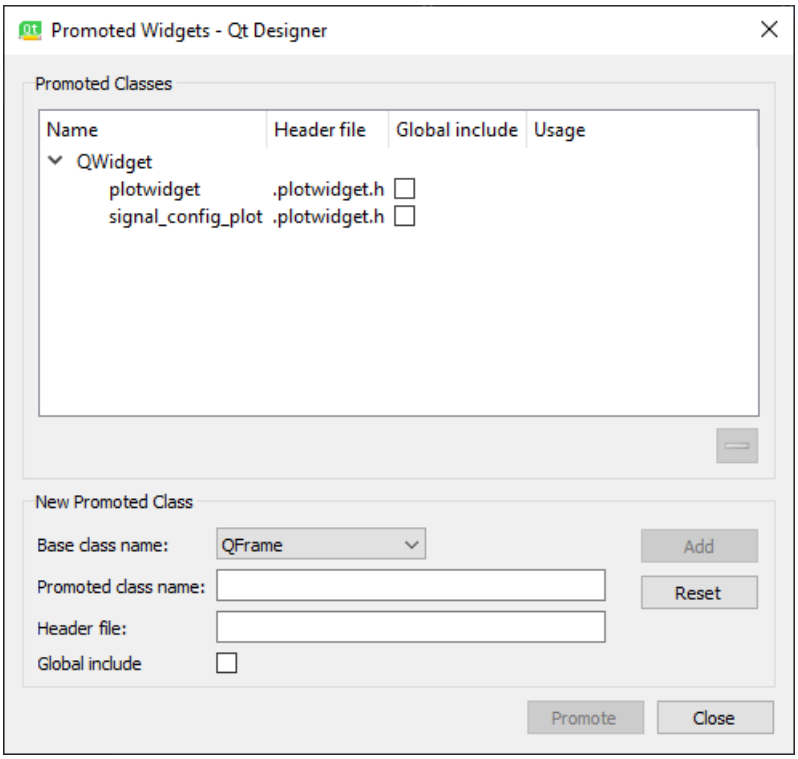


Abbildung 22: QtDesigner-Promoted-Widgets

Im Fall der grafischen Benutzeroberfläche, dieser Arbeit, wurden zwei Graphen mit *PyQt-Graph* als Promoted Widgets integriert. Beide Graphen werden in der Datei *plowdget.py* definiert und als Widget hinzugefügt. In der Datei *setup_manger* werden diese dann beim Start initialisiert und ausgeführt.

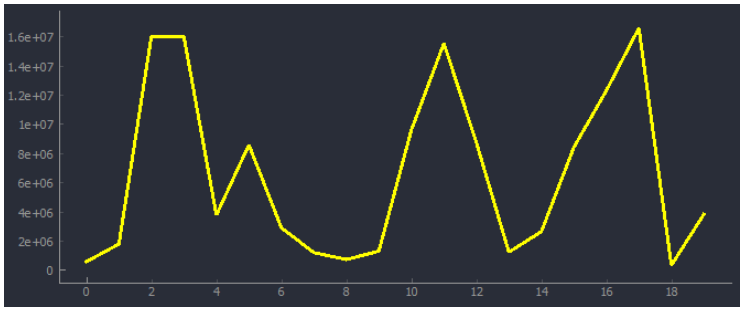


Abbildung 23: Promoted-Widget-1

[25] Vgl. **[25]**

Beide Graphen werden in der Datei *plotwidget.py* definiert.

plotwidget.py

```
1  from pyqtgraph import PlotWidget, plot
2  import pyqtgraph as pg
3
4  #define class plotwidget to import into gui
5  class plotwidget(QWidget):
6    def __init__(self, parent=None):
7      QWidget.__init__(self, parent)
8
9      #define how the wighet should look
10     vertical_layout = QVBoxLayout()
11     self.setLayout(vertical_layout)
12     self.graphWidget = pg.PlotWidget()
13     self.graphWidget.setBackground((41, 45, 56))
14     self.pen = pg.mkPen(color='y', width=3)
15     self.layout().addWidget(self.graphWidget)
16     self.setSizePolicy(QSizePolicy.Fixed, QSizePolicy.Fixed)
17
18 #Define class signal_config_plot to import into gui
19 class signal_config_plot(QWidget):
20   def __init__(self, parent=None):
21     QWidget.__init__(self, parent)
22     #define how the wighet should look
23     vertical_layout = QVBoxLayout()
24     self.setLayout(vertical_layout)
25     self.signal_config_plot = pg.PlotWidget()
26     self.signal_config_plot.setBackground((41, 45, 56))
27     self.pen = pg.mkPen(color='y', width=3)
28     self.layout().addWidget(self.signal_config_plot)
29     self.setSizePolicy(QSizePolicy.Preferred, QSizePolicy.Preferred)
30
```

Codeausschnitt 10: plotwidget.py Graphen

Diese Kurven werden in *setup_manager.py*[26] initialisiert. Der RPM-Graph wird in *ui_thread_manager.py*[27] und der signal_config-Graph in *ui_signal_config_manager.py*[28] beschrieben.

[26] siehe: 5.3.3.2.3
[27] siehe: 5.3.3.2.7
[28] siehe: 5.3.3.2.5

Qt Designer Verbindungen

Für sehr einfache Funktionen, wie Felder aktivieren, gibt es im QtDesigner eingebaute *Slots/Signals* die die Programmierung derartiger Funktionen selbst übernehmen. Für die GUI werden diese Slots hauptsächlich zur Aktivierung und Deaktivierung bestimmter Felder verwendet.

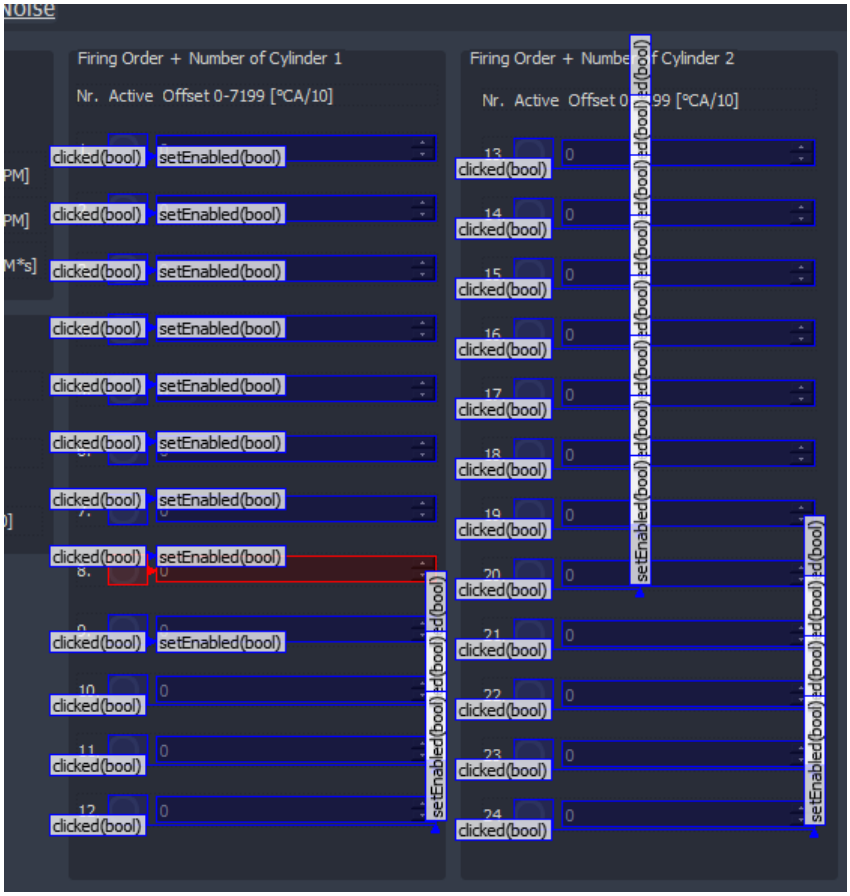


Abbildung 24: QtDesigner-Slots

Um die Benutzerfreundlichkeit der grafischen Benutzeroberfläche zu verbessern, sind viele „Sicherheiten“ eingebaut. Zum Beispiel kann man erst Einstellungen vornehmen, wenn ein FPGA oder Mikrocontroller an den PC angeschlossen ist. Weiterhin wird beim Verbinden zum ausgewählten Board und bei der Übertragung ein Ladebalken gestartet, der dem Benutzer signalisiert, dass eine Übertragung gestartet wurde bzw. eine Verbindung aufgebaut wird.

UI_extrafiles

In dem Ordner UI_extrafiles befindet sich die Datei *UI_functions.py*. Diese sorgt für eine komplette Kontrolle über das Aussehen der GUI. Sie sorgen für folgende Funktionen:

- die Windows-Titelzeile entfernt
- die Windows-Icons/Symbole entfernt
- Größe des Fensters veränderbar
- eigene Icons/Symbole hinzugefügt

Weiterhin werden für die eigenen Symbole, die Methoden zur Minimierung, Maximierung und Schließung des Programms definiert.

5.3.3.2.2 UI_Functions

Diese Datei beinhaltet folgende Methoden:

Methoden	Methodesaufgaben
maximize_restore	Maximierung und Rücksetzung auf originale Größe
enableMaximumSize	Setzen der maximalen Größe
removeTitleBar	Wenn wahr $\rightarrow$ Titelzeile wird entfernt
uiDefinitions	Schatten hinzufügen, Doppelklick zum Maximieren, Minimieren und Schließen der Applikation mit dem Schließen-Symbol

Tabelle 5: UI_Functions_Methoden

UI_main_extensions

In dem Unterordner UI_main_extensions befinden sich die Dateien, die die Threads initialliseren und das Back-End mit dem Front-End verknüpfen. Sie verwenden die Daten, die vom serial_handler[29] bereitgestellt werden.
Die Dateien wurden nach ihren grundsätzliche Methoden folgerndermaßen unterteilt:

- UI_config_manager
- UI_serial_extension
- UI_setup_manager
- UI_signal_config_manager
- UI_thread_manager
- UI_color_manager

[29] 5.3.3.3.1

5.3.3.2.3 UI_setup_manager

Die Komponente *UI_setup_manager*, initialisiert alle wichtigen Arrays, Knopfverbindungen und vieles mehr.
Der UI_setup_manager wird in folgende Methoden unterteilt:

Methode	Methodesaufgaben
setup_mainwindow	Ruft die Setup-Funktionen bei Startup auf
misc_setup	Beinhaltet: Initialisierung der Threads, Back-End/Serielle Verbindung, UIFunctions, Graphen, standard deaktivierte/aktivierte Knöpfe
init_logging	Initialisierung des Logging des Programms
mousePressEvent	
expertMode	Debugging Seite in GUI aktivieren/deaktivieren.
Debugging_Widget	Setzen eines Textes, um dem Nutzer zu signalisieren, dass er auf der Debugging Seite ist. Diese Seite kann nur ausgewählt werden, wenn der Expertmode aktiviert ist
open_manual	Knöpfe verbinden, um die Anleitung der GUI zu öffnen
setup_qt_load_save	Beschreiben und Laden von Einstellungen im Windows Register
connect_buttons	Verbindungen aller Knöpfe mit den entsprechenden Methoden im Programm herstellen
setup_arrays	Benötigte Arrays initialisieren

Tabelle 6: UI_setup_manager Methoden

setup_mainwindow()

Diese Methode wird beim Start der GUI in *ui_main.py* ausgeführt. Sie führt verschiedene andere Methoden aus, um alles beim Start Benötigte, zu initialisieren.

```
1  def setup_mainwindow(self):
2    # Initialize Logging-Package
3    self.init_logging()
4    # Setup miscellaneous variables
5    self.misc_setup()
6    # Setup Arrays to make Iteration in ui_config_manager easier
7    self.setup_arrays()
8    # Read Config
9    self.config_setup() # => ui_config_manager.py
10   self.connect_buttons()
11
12   self.setup_qt_load_save()
13   self.fill_list_with_config_data()
14
```

Codeausschnitt 11: UI_setup_manager_setup_mainwindow()

misc_setup()

Diese Methode erfüllt verschiedene Einstellungen und Initialsierungen. Sie wird innerhalb der *setup_mainwindow()* aufgerufen.

```
1  def misc_setup(self):
2    UIFunctions.removeTitleBar(True)
3    ...
4    self.ui.comboBox_Ports.setDisabled(True)
5    ...
6    self.boardconnect = Port_Connection()
7    ...
8    self.plot_x = [0]
9    self.plot_y = []
10   self.init_threads() # Initialize Threads => ui_thread_manager.py
11   self.start_thread(3) # =>Start Threads ui_thread_manager.py
12   self.curve1 = self.ui.plotwidget.graphWidget.plot(pen=self.ui.
      plotwidget.pen, name="Data 1", width=25)
13   for s in escape_sequences:
14     self.ui.comboBox_debugging_reqeusts.addItem(s.name)
15
```

Codeausschnitt 12: UI_setup_manager_misc_setup()

init_logging()

Diese Methode wird beim Start in der Methode *setup_mainwindow()* ausgeführt. In der Methode wird ein Logger erstellt und ein neues Logging-Level definiert. Der Logger speichert alle Vorgänge in eine Datei ab, um bei einem aufgetretenen Fehler, diesen relativ einfach zu finden.

```
1  def init_logging(self):
2    # Set the Level of the Root-Logger to NOTSET so all Messages will
       be
3    # processed by the classhild-Loggers
4    getLogger().setLevel(0)
5    # Add custom Log-Level
6    SERIAL_LEVEL_NO = 25
7    addLevelName(SERIAL_LEVEL_NO,"Serial")
8    # Add custom Function to Logger-Class for Serial-Level
9    def serial(self, message, *args, **kws):
10     if self.isEnabledFor(SERIAL_LEVEL_NO):
11       self._log(SERIAL_LEVEL_NO, message, args, kws)
12   Logger.serial = serial
13   ...
14   # Create Seperator in Log for new Session
15   self.logger.info("New Session")
16
```

Codeausschnitt 13: UI_setup_manager_init_logging()

setup_qt_load_save()

Diese Methode wird beim der Start der GUI in der Methode *setup_mainwindow()* aufgerufen. Sie erstellt im Windows Register einen neuen Eintrag um Einstellungen zu speichern. Es wird gespeichert, ob die Debugging Seite gezeigt wird oder nicht.

```
1  def setup_qt_load_save(self):
2    self.settings = QSettings("MyQtApp", "SAFIGUI")
3    try:
4      self.move(self.settings.value("window position"))
5      if self.settings.value("Expert Mode") == "true":
6        self.ui.radioButton_Expert_Mode.setChecked(True)
7        self.ui.pushButton_Debugging.show()
8      else:
9        self.ui.pushButton_Debugging.hide()
10   except: pass
11
```

Codeausschnitt 14: UI_setup_manager_setup_qt_load_save()

connect_buttons()

Diese Methode verknüpft alle Knöpfe mit zugehörigen Methoden. Sie wird im *setup_manager* ausgeführt. Ebenfalls wird die lokale Funktion moveWindow() definiert. Diese sorgt dafür, dass man das Fenster mit der eigen erstellten Titelzeile verschieben kann.

```
1  def connect_buttons(self):
2    def moveWindow(event):
3      if UIFunctions.returStatus() == 1:
4        UIFunctions.maximize_restore(self)
5
6      if event.buttons() == Qt.LeftButton:
7        self.move(self.pos() + event.globalPos() - self.dragPos)
8        self.settings.setValue(
9          "window position", (self.pos() + event.globalPos() - self.
       dragPos)
10       )
11       self.dragPos = event.globalPos()
12       event.accept()
13   self.ui.frame_label_top_btns.mouseMoveEvent = moveWindow
14   self.ui.stackedWidget.setCurrentWidget(self.ui.page_home)
15   self.ui.label_top_info_2.setText("Home")
16   self.ui.tabWidget.setCurrentWidget(self.ui.tab_general)
17   self.ui.tabWidget_2.setCurrentWidget(self.ui.CylinderPressure)
18   self.ui.comboBox_Ports.setDisabled(True)
19   self.ui.pushButton_connect.setDisabled(True)
20   self.ui.pushButton_home.setDisabled(True)
21   self.ui.pushButton_enigne_config.setDisabled(True)
22   self.ui.pushButton_signal_config.setDisabled(True)
23   self.ui.pushButton_Debugging.setDisabled(True)
24   ...
25
```

Codeausschnitt 15: UI_setup_manager_connect_buttons()

setup_arrays

Diese Methode wird beim der Start der GUI in der Methode *setup_mainwindow()* ausgeführt. Sie initialisiert alle benötigten Arrays.

```
1  def setup_arrays(self):
2    self.cylinder_active = [2] * 24
3    self.cylinder_press_active = [2] * 24
4    self.knock_active = [2] * 24
5    self.ignition_active = [2] * 24
6    self.offset = [0] * 24
7    self.offset2 = [0] * 24
8
9    self.Cylinders_active = [
10     self.ui.frame_Monitor_Zylinder_1,
11     ...
12     self.ui.frame_Monitor_Zylinder_24,
13   ]
14   ...
15   ...
16
```

Codeausschnitt 16: UI_setup_manager_setup_arrays()

5.3.3.2.4 UI_config_manager

Die Komponente UI_config_manager verwaltet das Speichern der Einstellungen, die der Benutzer tätigt, damit man diese später weiter verwendet werden können. Dazu werden alle Eingabefelder abgefragt und mithilfe vom Paket *ConfigParser*[30]in eine Konfigurationsdatei namens *Conifg.ini* gespeichert. Der Speichervorgang geschieht bei der Schließung der GUI und bei der Übertragung zum FPGA.
Weiterhin kann die gesamte Konfiguration exportiert und importiert werden, sodass unterschiedliche Einstellungen schnell ausgetauscht werden können. Das Laden der Konfiguration geschieht beim Start des Programms und bei der Übertragung zum FPGA.
Dieser Manager hat ebenfalls die wichtige Aufgabe, die Flags für die korrekte Übertragung der Einstellungen zu setzen.
Er wird in folgende Methoden unterteilt:

Methode	Methodesaufgaben
config_setup	Initialisierung des ConfigParsers und der Arrays
set_ram	Schicken der 7200 Zeichen langen Good und Bad Shapes, die in den BRAM des FPGA geschrieben werden
write_config	Speicherung aller Einstellungen
load_config	Laden aller Einstellung; setzen der Einstellungsfelder der GUI; setzen der Flags für die Übertragung
import_data	Importieren von den Störsignalen
read_config_file	Einlesen der Störsignale, aus der Config.ini, sodass diese in den Signal_Config-Graph importiert werden können
ram_data_to_int	Konvertierung der eingelesenen Strings zu entsprechender Ganz Zahl
export_config_data	Exportieren der Signale, des Signal_Config Graphen
write_current_configuration_to_file	Exportieren der Konfiguration in eine Programm externe Datei
load_config_file	Importieren einer Konfiguration aus einer Programm externen Datei
set_p_k_i	Flags setzen, welche Zylinder aktiviert sein sollen, an welchem Zylinder Knocksensor und/oder Ignitionvoltage Störungen auftreten sollen
transmit_configuration_button	Konfiguration an den FPGA senden

Tabelle 7: UI_config_manager Methoden

[30]Vgl.**[27]**

Codeauschnitte des UI_config_managers

config_setup()

Diese Methode wird im UI_setup_manager mit der Methode *setup_window()* ausgeführt.

```
1  from configparser import ConfigParser
2  class Config_Manager:
3    # Unused Constructor => Needed for Code-Analyser
4    def __init__(self):
5      self.ui = Ui_MainWindow()
6      self.ui.setupUi(self)
7    def config_setup(self):
8      # Create Data-Class to store the current Configuration
9      self.config = ConfigParser()
10     # Initialize Arrays for RAM-Data
11     self.Cylinder_Pressure_GoodShape = []
12     self.Cylinder_Pressure_BadShape = []
13     ...
14     self.load_config()
15
```

Codeausschnitt 17: UI_config_manager_config_setup()

set_ram

Die Methode *set_ram* verwendet die *write_to_ram* Methode, die im *serial_handler_write_interface* definiert ist. Die Störsignale werden von der *import_data* in die entsprechenden Arrays geschrieben.

```
1  def set_ram(self):
2    self.boardconnect.write_to_ram(0, self.Cylinder_Pressure_GoodShape)
3    self.boardconnect.write_to_ram(3, self.Cylinder_Pressure_BadShape)
4
5    self.boardconnect.write_to_ram(1, self.Knocksensor_GoodShape)
6    self.boardconnect.write_to_ram(4, self.Knocksensor_BadShape)
7
8    self.boardconnect.write_to_ram(2, self.IgnitionVoltage_GoodShape)
9    self.boardconnect.write_to_ram(5, self.IgnitionVoltage_BadShape)
10
```

Codeausschnitt 18: UI_config_manager_set_ram()

write_config()

Diese Methode wird ausgeführt, bei der Schließung der Programms und vor dem der Start der Übertragung.

```
1  def write_config(self, file="gui\config.ini"):
2    #saving config into the config.ini file
3    # list of all sections in config
4    sections = ["EngineSpeed settings",
5          "Trigger-CAM/Reset settings",
6          ...
7          "RAM_IgnitionVoltage_DATA_Bad_Shape"
8    ]
9    # Add Sections
10   for section in sections:
11     if not self.config.has_section(section):
12       self.config.add_section(section)
13
14
15   #EngineSpeed
16   self.config.set("EngineSpeed settings", "Triagnle_Function", f"{
       self.ui.radioButton_Triangle_Function.isChecked()}")
17   ...
18   #Trigger-CAM/Reset
19   self.config.set("Trigger-CAM/Reset settings", "Gear_teeth", f"{self
       .ui.spinBox_GEAR_TEETH.value()}")
20   ...
21   #General
22   for i in range(0,24):
23     self.config.set("General_Cylinder_on_off", f"General_Cylinder_{i}
       ", f"{self.General_Zylinder_checkBoxes[i].isChecked()}")
24   for i in range(0,24):
25     self.config.set("General_Cylinder_offest_value", f"
       General_Cylinder_{i}", f"{self.General_Zylinder_spinBoxes[i].value
       ()}")
26
27   #open config file and write config
28   with open(f"{file}", "w") as f:
29     self.config.write(f)
30
```

Codeausschnitt 19: UI_config_manager_write_config()

load_config()

Diese Methode wird beim start des Programms und bei der Übertragung, da sie die Flags setzt, ausgeführt.

```
1  def load_config(self, file="gui\config.ini"):
2    self.General_Cylinder = []
3    ...
4    self.config.read(file)
5    if self.config.has_section("EngineSpeed settings"):
6      #Engine Speed Settings
7      self.ui.radioButton_Triangle_Function.setChecked(self.config.
     getboolean("EngineSpeed settings", "triagnle_function"))
8      self.ui.radioButton_Ramp_Function.setChecked(self.config.
     getboolean("EngineSpeed settings", "ramp_function"))
9    ...
10   if self.config.has_section("Trigger-CAM/Reset settings"):
11     #Trigger-CAM/Reset settings
12     gear_teeth = self.config.getint("Trigger-CAM/Reset settings", "
     gear_teeth")
13     self.boardconnect.set_gear_teeth(gear_teeth)
14     self.ui.spinBox_GEAR_TEETH.setValue(gear_teeth)
15   ...
16   if self.config.has_section("Pressure_Cylinder_on_off"):
17     for i in range(0, 24):
18       active_flags = self.config.getboolean("Pressure_Cylinder_on_off
     ", f"pressure_cylinder_{i}")
19       self.Cylinder_Pressure_Zylinders_checkBoxes[i].setChecked(
     active_flags)
20       self.Cylinder_Pressure_Zylinders_spinBoxes[i].setEnabled(
     active_flags)
21       if active_flags == True:
22         self.Pressure_Cylinder.append(1)
23       else:
24         self.Pressure_Cylinder.append(0)
25       ...
26     self.boardconnect.set_active_flags_cylinder_pressure(self.
     Pressure_Cylinder)
27   ...
28
```

Codeausschnitt 20: UI_config_manager_load_config()

import_data()

Diese Methode wird ausgeführt, wenn ein Benutzer auf einen Import-Knopf klickt. Sie liest eine Datei ein, in der sich das zu importierende Störsignal befindet.

```
1  def import_data(self, type="Cylinder_Press",good_bad=0):
2    if type in ["Cylinder_Pressure","Knocksensor","IgnitionVoltage"]:
3      # Open FileDialog and get selected file, with 2 possible file
       formates (csv, txt, ini)
4      userFile = QFileDialog.getOpenFileName(self, "Open File", "", "
       (*.txt);;(*.ini)")[0]
5      if userFile != "":
6        # Read data from the selected File
7        if self.ui.lineEdit_shapes_delimiter.text() == "":
8          data = self.read_config_file(file=userFile,header=self.ui.
       lineEdit_shapes_column.text())
9        else:
10         data = self.read_config_file(file=userFile,header=self.ui.
       lineEdit_shapes_column.text(), delimiter=self.ui.
       lineEdit_shapes_delimiter.text())
11       if len(data) > 0:
12         if type == "Cylinder_Pressure":
13           if not good_bad:
14             section_name = "RAM_Cylinder_Pressure_DATA_Good_Shape"
15             self.Cylinder_Pressure_GoodShape = data
16           ...
17           else:
18             section_name = "RAM_IgnitionVoltage_DATA_Bad_Shape"
19             self.IgnitionVoltage_BadShape = data
20
21         for i in range(0,len(data)):
22           self.config.set(section_name, f"{i}",f"{data[i]}")
23         QMessageBox.information(self, "Data was imported","Data was
       imported")
24       else:
25         self.config_error_message("Unable to use File","The supplied
       File contains " + str(len(data)) + " Values, Max-Length is 7200")
26   else:
27     self.config_error_message("Import Data Error","Couldn't determine
       which data to import")
28
```

Codeausschnitt 21: UI_config_manager_import_data()

set_p_k_i

Diese Methode wird bei der Übertragung ausgeführt und setzt die aktiven Zylinder.

```
1  def set_p_k_i(self):
2    self.set_cylinder_pressure_active(self.Pressure_Cylinder)
3    self.set_knocksensor_active(self.Knocksensor_Cylinder)
4    self.set_ignition_voltage_active(self.IgnitionVoltage_Cylinder)
5
```

Codeausschnitt 22: UI_config_manager_set_p_k_i()

transmit_configuration_button

Diese Methode wird ausgeführt, wenn der Benutzer auf den „Transmit Configuration“ Knopf klickt.

```
1  def transmit_configuration_button(self):
2    self.write_config()
3    self.load_config()
4
5    self.ui.label_Transmit_progressbar.setText("Writing RAM / Shapes")
6    self.set_ram()
7
8    value, text = self.boardconnect.transmit_configuration()
9    for i in range(0,len(value)):
10     self.ui.progressBar_Transmit_Configuration.setValue(value[i])
11     self.ui.label_Transmit_progressbar.setText(text[i])
12
13   self.ui.progressBar_Transmit_Configuration.setValue(0)
14   self.ui.label_Transmit_progressbar.setText("")
15   if self.worker_uart_read.monitor == True:
16     self.boardconnect.write_request(124)
17     self.set_p_k_i()
18
```

Codeausschnitt 23: UI_config_manager_transmit_configuration_button()

Sie schreibt zuerst die Einstellungen in die Konfigurations-Datei und liest diese danach ein, da beim Einlesen die Flags gesetzt werden. Es werden die Störsignale (Good und Bad Shapes) vorbereitet um sie in den RAM des FPGA zu laden. Danach werden mithilfe der Methode *self.boardconnect.transmit_configuration()*, welche im *serial_handler_write_interface* definiert ist, alle Daten übertragen.

5.3.3.2.5 UI_signal_config_manager

Die Komponente UI_signal_config_manager verwaltet die zweite Graphik der GUI, womit der Verlauf importierter Störsignale verändert werden kann.
Die Komponente setzt sich aus folgenden Methoden zusammen:

Metode	Methodesaufgaben
clear_signal_graph	den Graphen löschen
clear_table	Tabelle löschen
get_index_value_from_ram_data	aus der Config.ini Störsignal importieren
fill_list_with_config_data	*get_index_value_from_ram_data* zu importierendes Signal übergeben
update_signal_configß_graph	zu importierendes Signal in den Graph schreiben
import_graph	Störsignal aus Programm externer Datei importieren
export_graph	Verändertes Störsignal exportieren

Tabelle 8: UI_signal_config_manager Methoden

get_index_value_from_ram_data()

Diese Methode wird in der Methode *fill_list_with_config_data* ausgeführt, um die Daten des gewünschten Störsignals aus der Konfigurationsdatei in die Tabelle zu importieren.

```
1  def get_index_value_from_ram_data(self, section_name):
2    #Create needed Lists
3    index = []
4    value = []
5    #Clear Graph
6    self.clear_signal_graph()
7    self.clear_table()
8    #Check if Ram section is available
9    if self.config.has_section(section_name):
10     #Check if Data in section is available
11     if self.config.has_option(section_name, "0"):
12       #Iterate through RAM data
13       self.ui.tableWidget.setRowCount(7200)
14       for i in range(0, 7200):
15         index.append(i)
16         value.append(self.config.getint(section_name, f"{i}"))
17         self.ui.tableWidget.setItem(i, 0, QTableWidgetItem(str(value[
     i])))
18
```

Codeausschnitt 24: UI_signal_config_manager_get_index_value_from_ram_data()

fill_list_with_config_data()

Diese Methode wird ausgeführt, wenn ein Benutzer ein Störsignal aus der Konfigurationsdatei importieren will.

```
1  def fill_list_with_config_data(self):
2    self.ui.tableWidget.clear()
3    self.clear_signal_graph()
4    if self.ui.comboBox_shapes.currentText() == "cylinder_pressure_good
       ":
5      self.get_index_value_from_ram_data("
       RAM_Cylinder_Pressure_DATA_Good_Shape")
6
7
8    elif self.ui.comboBox_shapes.currentText() == "
       cylinder_pressure_bad":
9      self.get_index_value_from_ram_data("
       RAM_Cylinder_Pressure_DATA_Bad_Shape")
10   ...
11
```

Codeausschnitt 25: UI_signal_config_manager_fill_list_with_config_data()

update_signal_config_graph()

Die Methode *update_signal_config_graph* wird ausgeführt, wenn der Benutzer auf den Knopf zum Graphen updaten drückt, wenn der Nutzer einen Datenpunkt in der Tabelle geändert hat.

```
1  def update_signal_config_graph(self):
2    ..
3    #iterate over changes
4    for i in range(0 , self.ui.tableWidget.rowCount()):
5      item = self.ui.tableWidget.item(i, 0)
6      try:
7        temp = int(item.text())
8      except:
9        return
10     if 0 <= temp <= 4095:
11       value.append(temp)
12       index.append(i)
13     else:
14       return
15   self.curve2.setData(index, value)
16
```

Codeausschnitt 26: UI_signal_config_manager_update_signal_config_graph()

import_graph()

Mit dieser Methode kann ein Störsignal auf der Signal_Config Seite von einer Programm externen Datei importiert werden.
Sie wird ausgeführt, wenn ein Benutzer auf einen Knopf klickt.

```
1  def import_graph(self):
2  #create needed list and "global values"
3    index = []
4    #Get slected userfile
5    userFile = QFileDialog.getOpenFileName(self, "Open File", "", "(*.
       txt);;(*.ini)")[0]
6    if userFile != "":
7      if self.ui.lineEdit_shapes_delimiter.text() == "":
8        value = self.read_config_file(file=userFile,header=self.ui.
       lineEdit_signal_config_column.text())
9      else:
10       value = self.read_config_file(file=userFile,header=self.ui.
       lineEdit_signal_config_column.text(), delimiter=self.ui.
       lineEdit_signal_config_delimiter.text())
11     ...
12     #Import data to config, to transmit to board
13     if self.ui.comboBox_shapes.currentText() == "
       cylinder_pressure_good":
14       section_name = "RAM_Cylinder_Pressure_DATA_Good_Shape"
15       self.Cylinder_Pressure_GoodShape = value
16       if len(value) > 0:
17       #clear graph and list
18       self.clear_signal_graph()
19       self.clear_table()
20       for i in range(0,len(value)):
21         index.append(i)
22       self.ui.tableWidget.setRowCount(len(index))
23       for i in range(0, len(index)):
24         self.ui.tableWidget.setItem(i, 0, QTableWidgetItem(str(value[
       i])))
25         #self.ui.tableWidget.insertRow(self.ui.tableWidget.rowCount()
       )
26       self.curve2.setData(index, value)
27
```

Codeausschnitt 27: UI_signal_config_manager_import_graph()

export_graph()

Mit dieser Methode kann ein Störsignal auf der Signal_Config Seite von einer Programm externen Datei exportiert werden.
Sie wird ausgeführt, wenn ein Benutzer auf einen Knopf klickt.

```
1  def export_graph(self):
2    userFile = QFileDialog.getSaveFileName(self, "Open File", "", "(*.
       txt);;(*.ini)")[0]
3    if userFile != "":
4      # Append the current Graph to an Array
5      data = []
6      for i in range(0, self.ui.tableWidget.rowCount()-1):
7        data.append(f'{int(int(self.ui.tableWidget.item(i,0).text())
       /10)}')
8      # Write the Array to a File
9      self.export_config_data(file=userFile,export_data=data,header=
       self.ui.lineEdit_signal_config_column.text(),delimiter=self.ui.
       lineEdit_signal_config_delimiter.text())
10
```

Codeausschnitt 28: UI_signal_config_manager_export_graph()

5.3.3.2.6 UI_serial_extension

Die Komponente UI_serial_extension implementiert die Methoden, um das Back-End zu steuern.
Die *UI_serial_extension* wird aus den folgenden Methoden zusammengesetzt:

Methode	Methodesaufgaben
list_ports_and_connect_to_board	Aktualisiert die Dropdown-Liste der verbundenen COM-Ports in der GUI
connect_to_board	Wird verwendet um dem serial_handler zu kommunizieren, dass dieser sich, mit dem im GUI ausgewählten Port, verbinden soll. Aktiviert/Deaktiviert Knöpfe für den Benutzer.
write_to_board	Wird für die Seite Debugging verwendet, um manuell Daten an den FPGA zu senden. Die Daten kann man mithilfe von 4 Eingabefelder genau spezifizieren.
send_debug_request	Wird für die Seite Debugging verwendet, um ein Datenset an den FPGA zu senden. Hier kann man aus einem Drop-Down-Menü den Datentyp auswählen und die restlichen Daten werden aus dem request_construtor geholt.
disconnect_from_board	Verbindung vom FPGA trennen und entsprechende Felder und Knöpfe für den Benutzer aktivieren/-deaktivieren.

Tabelle 9: UI_serial_extension Methoden

list_ports_and_connect_to_board()

Die *list_ports_and_connect_to_board*-Methode wird vom *Refresh_Port_List*-Thread in einem definierten Intervall aufgerufen, um die Liste an angeschlossenen Boards zu aktualisieren:

```
1  def list_ports_and_connect_to_board(self):
2    try:
3    # Get updated List of Ports from OS
4      corr_port_found, ports = self.boardconnect.find_all_Ports()
5    # Pseudo-Code
6    set_buttons_and_port_list(ports)
7    # If the correct-Port was found, try connecting to it
8    if(corr_port_found):
9     # Catch any Errors
10    try:
11     # Connect to Board
12     self.boardconnect.set_com_port(ports[0])
13     self.boardconnect.connect_to_board()
14  # Pseudo Code
15  except:
16   LOG ERROR
17
```

Codeausschnitt 29: UI_serial_extension_list_ports_and_connect_to_board()

connect_to_board()

Connect_to_board wird aufgerufen, wenn der Benutzer auf den “Connect“-Button klickt und ein Gerät am PC angeschlossen ist.
Daraufhin werden die restlichen Knöpfe aktiviert bzw. deaktiviert. Diese werden ausgewählt, je nach dem welche Funktionen gerade benötigt werden. Besonders wird Acht gegeben auf Knöpfe, die bei mehrmaliger Verwendung das Programm zum Absturz bringen könnten. Weiterhin werden die entsprechenden GUI-Threads zum Empfangen der Datensätze gestartet:

```
1  def connect_to_board(self):
2   try:
3    if self.boardconnect.connect_to_board():
4     # Pseudo-Code
5     disable_connect_buttons()
6     enable_disconnect_and_write_buttons()
7        # Set Flag
8        self.is_connected = True
9        # Start Reading-Thread
10       self.start_thread(1)
11       # Start Progress-Bar-Thread
12       self.start_thread(2)
13  except:
14   # Pseudo-Code
15   LOR ERROR
16
```

Codeausschnitt 30: UI_serial_extension_connect_to_board()

write_to_board()

Write_to_board wurde implementiert damit ein Nutzer zu Debugging-Zwecken manuell Daten senden kann. Da hier User-Input direkt an den FPGA gesendet wird müssen spezielle Vorsichtsmaßnahmen getroffen werden, da PySerial nur 8-Bit-Integer senden kann:

```
1  def write_to_board(self):
2   # Initialize Dataset
3    data = bytearray([0,0,0,0])
4    temp = [-1,-1,-1,-1]
5   # Try getting the Values from the Input-Text-Boxes
6   try:
7    # Pseudo-Code
8    temp = get_values_from_text_boxes()
9   except:
10   # Pseudo-Code
11   LOG ERROR
12   return
13   # Get the Data which would normally be sent with the Escape-
      Sequence
14   stored_request = self.boardconnect.return_raw_request(data[0] >> 1)
15  for i in range(1,4):
16   # Check if all the Int's are 8 Bytes (0 < x <= 255) long => bytes()
       -Conversion in Serial-Handler would throw Error if not
17   if 0 <= temp[i] <= 255:
18    data[i] = temp[i]
19   else:
20    # If an Int is not in the bytes()-Range fill in the Value which
       would normally be sent using the request_construtor
21    data[i] = stored_request[i]
22  # Send the Data to the Serial-Handler so it will be sent
23  self.boardconnect.write_request(data)
24
```

Codeausschnitt 31: UI_serial_extension_write_to_board()

send_debug_request()

Send_debug_request wurde implementiert, damit ein normaler User manuell Daten an den FPGA senden kann. Hier kann er aus einem Drop-Down-Menü mit den Namen der Datentypen auswählen, was gesendet werden soll und dies wird dann automatisch mit den richtigen Daten aufgefüllt.

```
1  # Executed when the Send-Request-Button in the Debugging-Panel is
       pressed
2  # Gets the currently selected Text in the Dropdown-Menu next to it
3  # Then it converts that Text to the corresponding Escape-sequence and
       tells
4  # the Serial-Handler to send that Escape-Sequence with Data from the
       request-constructor.
5  def send_debug_request(self):
6
7    # Translate the String in the Dropdown-Menu to the corresponding
       Escape-sequence and
8    # tell the Serial-Handler to send it with the stored Data
9    self.boardconnect.write_request(escape_sequences.
       return_code_by_name(self.ui.comboBox_debugging_reqeusts.
       currentText()))
10
```

Codeausschnitt 32: UI_serial_extension_send_debug_request()

disconnect_from_board()

Disconnect_to_board wird aufgerufen, wenn der User auf den “Disconnet“-Button klickt und ein Gerät verbunden ist.
Hier werden dann die entsprechenden Buttons aktiviert/deaktiviert.
Außerdem werden die entsprechenden GUI-Threads und der serial_handler gestoppt:

```
1  def disconnect_from_board(self):
2   # Pseudo-Code
3   disable_buttons()
4   enable_buttons()
5    # Stop the Refresh_Port_List and ProgressBar-Threads
6    self.stop_comm_threads()
7    # Tell Serial-Handler to close the Connection
8    self.boardconnect.stop_thread_and_connection()
9    # Reset Connection-Flag
10   self.is_connected = False
11
```

Codeausschnitt 33: UI_serial_extension_disconnect_from_board()

5.3.3.2.7 UI_thread_manager

Die Komponente UI_thread_manager verwaltet und initialisiert die PyQt-Threads am Front-End. Hierfür wird eine “WorkerThread“-Klasse erstellt, welche in einem sogenannten QThread bewegt wird. Diesem WorkerThread kann dann eine Subroutine zugewiesen werden, die in einem separaten Thread ausgeführt wird, wenn man den QThread startet. Außerdem können Signale mit dem WorkerThread verbunden werden, die dann emittiert werden können, um Daten an den Main-Thread und die GUI zu senden. Um zu erkennen, ob ein QThread läuft und um den Threads zu signalisieren das sie stoppen sollen, wurde eine “working“-Variable implementiert, da die von PyQt implementierte “running“-Variable und “stop“-Funktion in unseren Tests nicht korrekt funktioniert haben.

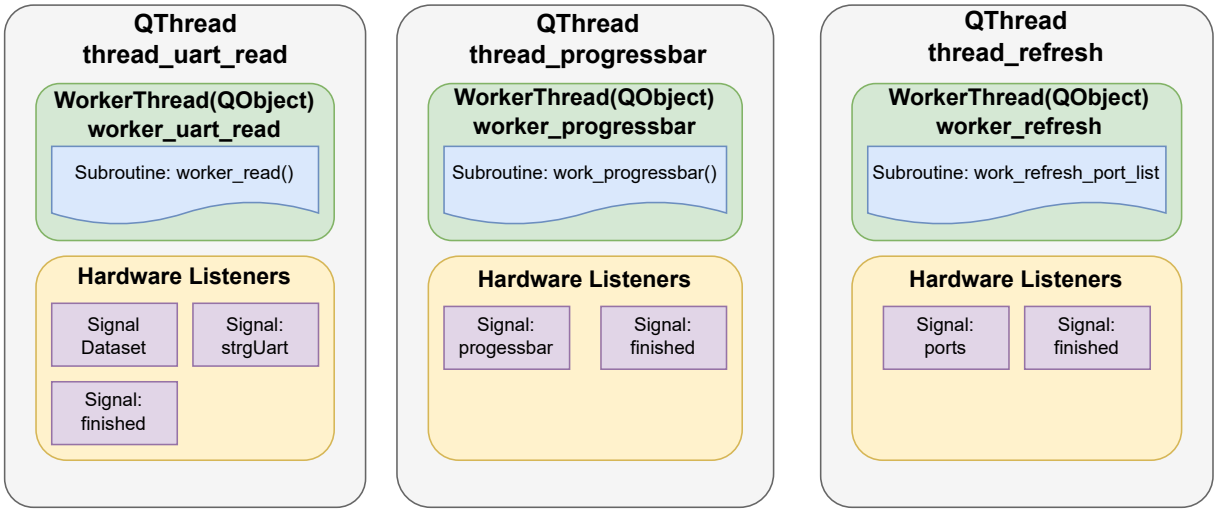


Abbildung 25: Qt-Threads

Der UI_thread_manager setzt sich aus folgenden Methoden zusammen:

Methode	Methodesaufgaben
start_thread	Initialisiert und startet den angegebenen Thread
stop_all_threads	Befiehlt allen Threads, dass sie stoppen sollen
stop_comm_threads	Befiehlt dem “uart_read“-, und “progressBar“-Thread, dass sie stoppen sollen
stop_refresh_thread	Befiehlt dem “refresh_port_list“-Thread, dass er stoppen soll
hard_stop_all_threads	Stoppt alle Threads
hard_stop_comm_threads	Stoppt den “uart_read“-, und “progressBar“-Thread
hard_stop_thread	Stoppt den angegeben Thread
handle_dataset	Verarbeitet ein einkommendes Dataset vom “uart_read“-Thread
write_to_uart_terminal	Sendet die String-Representation eines Datensets an das Terminal im Debugging-Fenster
progessBar_setValue	Aktualisiert die Fortschritts-Anzeige in der GUI
refresh_port_list	Hilfsfunktion für den “refresh_port_list“-Thread

Tabelle 10: UI_thread_manager

WorkerThread

Bei der Deklarierung der WorkerThread-Klasse müssen die Signale die verbunden werden können, definiert werden.
Diese führen auf der GUI-Seite eine Callback-Funktion mit Argumenten des spezifizierten Datentypes aus.

```
1 class WorkerThread(QObject):
2   finished = pyqtSignal()
3   strgUart = pyqtSignal(str)
4   progressBar = pyqtSignal(int)
5   ports = pyqtSignal(list, bool)
6   received_dataset = pyqtSignal(Dataset)
```

Codeausschnitt 34: WorkerThread-Klasse

Innerhalb der WorkerThread Klasse werden folgende die Subroutinen für die Threads definiert:

start_thread

Mithilfe der start_thread-Funktion können Threads einzeln gestartet werden:

```
1  def start_thread(self,num_thread):
2   if (num_thread == 1) and (not self.worker_uart_read.is_running()):
3    # init Reading Thread
4    self.worker_uart_read.working = True
5    self.worker_uart_read = WorkerThread(self.boardconnect)
6    self.thread_uart_read = QThread()
7    self.worker_uart_read.moveToThread(self.thread_uart_read)
8    self.thread_uart_read.started.connect(self.worker_uart_read.
        work_read)
9    self.worker_uart_read.strgUart.connect(self.write_to_uart_terminal)
10   self.worker_uart_read.received_dataset.connect(self.handle_dataset)
11   self.worker_uart_read.finished.connect(self.thread_uart_read.quit)
12   self.worker_uart_read.finished.connect(self.worker_uart_read.
        deleteLater)
13   self.thread_uart_read.finished.connect(self.thread_uart_read.
        deleteLater)
14   self.thread_uart_read.start()
15  elif (num_thread == 2) and (not self.worker_progressbar.is_running()
       ):
16   # Init ProgressBar Thread
17   self.worker_progressbar.working = True
18   ...
19   self.thread_progressbar.finished.connect(self.thread_progressbar.
        deleteLater)
20   self.thread_progressbar.start()
21  elif (num_thread == 3) and (not self.worker_refresh.is_running()):
22   # Init Refresh_Port_List-Thread
23   self.worker_refresh.working = True
24   self.worker_refresh = WorkerThread(self.boardconnect)
25   ...
26   self.thread_refresh.finished.connect(self.thread_refresh.
        deleteLater)
27   self.thread_refresh.start()
```

Codeausschnitt 35: UI_thread_manager_start_thread - Funktion

work_read

Der Worker *work_read* liest ununterbrochen von der Datenpaket Message Queue vom “serial_handler“ ein und sendet diese an den Main-Thread.
Ausserdem sendet er im Monitor-Mode in einem fixen Interval einen Transmit-Befehl an den FPGA.

```
def work_read(self):
  time.sleep(2)
  self.running = True

  while self.working:
    # Block until flag is set or abort after timeout duration
    if self.boardconnect.data_valid.wait(self.timeout):
      # Read until there is no more data available
      while not self.boardconnect.data_transmitter.empty():
        # Get read data from Serial-HandlerS
        dataset = self.boardconnect.data_transmitter.get()
        # Pass dataset as String to Debug-Terminal
        self.strgUart.emit(dataset.display())
        # Pass dataset to GUI to set Values accordingly
        self.received_dataset.emit(dataset)
      # Reset flag after all data was read
      self.boardconnect.data_valid.clear()
    elif self.monitor:
      # Send Transmit-Request if no data is coming in
      self.boardconnect.write_request(125)

  self.running = False
 self.finished.emit()
```

Codeausschnitt 36: UI_thread_manager_work_read-Subroutine

work_progressbar

Work_Progressbar wird verwendet um beim Verbindungsaufbau dem Nutzer zu signalisieren, ab welchem Zeitpunkt er die GUI sicher verwenden kann.

```
def work_progressbar(self):
  self.completed = 0
  self.running = True
  while (self.completed < 100) and (self.working):
    self.completed += 5
    self.progressBar.emit(self.completed)
    time.sleep(0.001)
  self.running = False
  self.finished.emit()
```

Codeausschnitt 37: UI_thread_manager_work_progressbar-Subroutine

work_refresh_port_list

Work_refresh_port_list aktualisiert die COM-Port Liste der GUI in einem fixen Interval.

```
1  def work_refresh_port_list(self):
2    ports = []
3    self.running = True
4    while self.working:
5      corr_port_found, ports_temp = self.boardconnect.find_all_Ports()
6
7      if ports != ports_temp:
8        ports = ports_temp
9        self.ports.emit(ports, corr_port_found)
10
11     time.sleep(1)
12   self.running = False
13   self.finished.emit()
```

Codeausschnitt 38: UI_thread_manager_work_refresh_port_list-Subroutine

refresh_port_list

Die *refresh_port_list*-Funktion wird beim Start der GUI und mithilfe des *refresh_port_list*-Thread in einem definierten Interval ausgeführt:

```
1  def refresh_port_list(self, ports, corr_port_found):
2   # Check if connected Board was unplugged
3   if ((self.is_connected) and (self.boardconnect.port not in ports)):
4    self.disconnect_from_board()
5   # Check if any Ports are connected
6   if len(ports) > 0:
7    self.ui.comboBox_Ports.setDisabled(False)
8    self.ui.pushButton_connect.setDisabled(False)
9    self.ui.comboBox_Ports.clear()
10   if corr_port_found == True:
11    self.ui.label_port_found.setText(ports[0])
12   # List all Ports in the Combo_Box
13   for i in range(0, len(ports)):
14    self.ui.comboBox_Ports.addItem(ports[i])
15  # If no Port is connected raise Error
16  else:
17   self.logger.warn("No Ports were found")
```

Codeausschnitt 39: UI_thread_manager_refresh_port_list-Funktion

handle_dataset

Handle_dataset wird an das *received_dataset*-Signal des *uart_read*-Thread verbunden und somit jedes Mal ausgeführt, wenn ein neues Datenset vom *serial_handler* an die GUI geschickt wird.
Hier wird je nach Datentyp in der GUI ein Feld geändert:

```
1  def handle_dataset(self,dataset):
2    if (dataset.datatype is Datatype.RPM_STATUS):
3      data_int = dataset.return_payload()
4    # Set current RPM-Text-Field
5      self.rpm_status(data_int)
6    # Append RPM to Graph
7      self.plot_x.append(self.plot_x[-1] + 1)
8      self.plot_y.append(data_int)
9    # Update Graph
10     if (self.plotting):
11       self.update_graph(self.plot_x[:-1], self.plot_y)
12   elif dataset.datatype is Datatype.CYL_ACTIVE_FLAGS:
13     self.set_cylinders_active(dataset.return_payload())
14  ...
15   elif dataset.datatype is Datatype.ROTATION_COUNTER:
16     self.rotation_counter(dataset.return_payload())
17
18   else:
19     pass
```

Codeausschnitt 40: UI_thread_manager_handle_dataset

5.3.3.3 Back-End

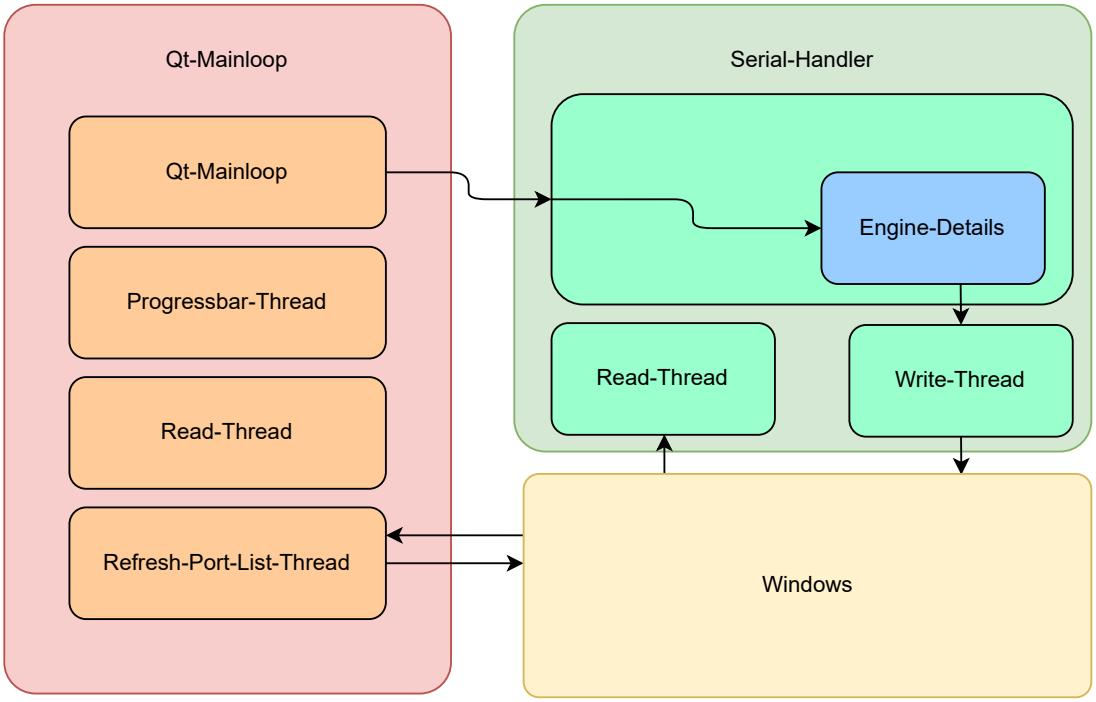


Abbildung 26: Thread-Blockschaltbild

Das Back-End des Python Programms befasst sich mit der Kommunikation mit dem FPGA und der Thread-sicheren Kommunikation des Back-Ends mit dem Front-End.

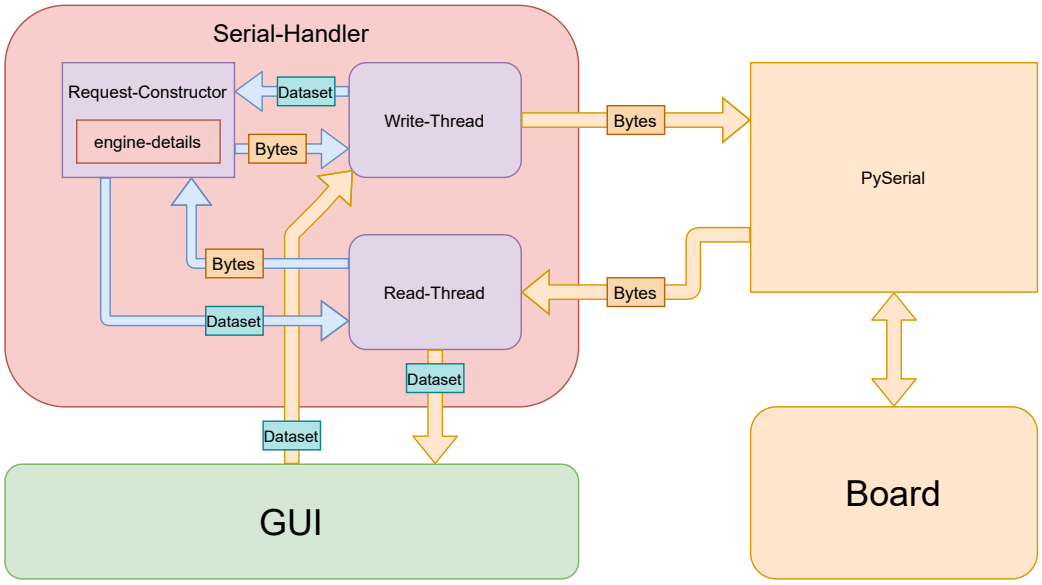


Abbildung 27: Back-End-Blockschaltbild

5.3.3.3.1 serial_handler

Der Serial_handler ermöglicht die Kommunikation mit dem FPGA über UART mithilfe von PySerial.
Um die Kommunikation mit einer guten Geschwindigkeit zu ermöglichen wird, der Serial_handler mithilfe von Threads, Queues, und blockierenden Funktionsaufrufen implementiert.
Hier wurde ausserdem ein Logger implementiert, der die Daten welche über die serielle Schnittstelle gesendet werden, in einer File abspeichert, damit man im Nachhinein sicherstellen kann das der Datenverkehr richtig verlaufen ist.
Der serial_handler setzt sich aus folgenden Methoden zusammen:

Methode	Methodesaufgaben
__init__	Initialisiert den serial_handler
find_all_Ports	Fragt das Betriebssystem nach den angeschlossenen Ports und macht eine schnelle Analyse ob das FPGA angeschlossen ist.
set_com_port	Setzt den COM-Port
connect_with_board	Öffnet die Schnittstelle mit dem durch die „set_com_port“-Methode gesetzten Port.
__handle_read__	Private Methode die als Subroutine für den Read-Thread verwendet wird.
__handle_write__	Private Methode die als Subroutine für den Write-Thread verwendet wird.
write_request	Bevorzugte Methode um Daten an den FPGA zu schicken.
__write_four_bytes__	Private Methode um ein Datenset an den FPGA zu senden. Wird vom Write-Thread verwendet.
return_raw_request	Konstruiert ein Datenset mit der gegebenen Escape-Sequence mit Daten aus der engine-details-Klasse und gibt diese zurück.
start_thread	Initialisiert den Read-, und Write-Thread und startet sie daraufhin.
__init_thread__	Private Methode zum Initialisieren der Threads.
__start_thread__	Private Methode zum Starten der Threads.
stop_thread_and_connection	Schliesst die Verbindung mit dem Board und stoppt den Read-, und Write-Thread.
stop_thread	Stoppt Read-, und Write-Thread.
stop_connection	Schliesst die Verbindung mit dem verbundenen Board.

Tabelle 11: UI_thread_manager

__init__()

Im Konstruktor des serial_handler werden mehrere Felder initialisiert:

- Der Serial-Logger
- Die Liste der verbundenen Ports
- Der Input-Buffer
- Das serielle Interface
- Der request_constructor
- Die Message-Queue die die GUI verwendet um Datenpakete zu senden.
- Die Read-, und Write-Threads

```
1  def __init__(self, port="", baud=115200, timeout=0.05):
2    # Initialize Logger
3    self.__logger__ = getLogger("SerialLogger")
4
5    # Keep Track of the connected COM-Ports
6    self.__ports__ = []
7
8    # Buffer the incoming data
9    self.__input_buffer__ = bytearray([])
10
11   # "Initialize" the Serial Interface
12   self.__ser__ = None
13   # Keep Track of Communication specific Parameters
14   self.port = port
15   self.baudrate = baud
16
17   # Create Request-Constructor to outsource Integer-Logic into
       seperate File
18   self.__request_constructor__ = write_request_constructor()
19
20   # Create Thread-sage "List" to receive Commands from other Threads
21   self.__cmd_receiver__ = Queue()
22   self.__cmd_valid__ = Event()
23
24   # Create Thread for all Communications with the FPGA
25   # The Thread can only be active if the Serial-Connection is open
26   self.__terminate_thread__ = False
27   self.__write_thread__ = Thread(target=self.__handle_write__,daemon=
       True)
28   self.__read_thread__ = Thread(target=self.__handle_read__,daemon=
       True)
29
```

Codeausschnitt 41: serial_handler__init__()

find_all_Ports()

Find_all_Ports sendet eine Anfrage an das Betriebssystem welches die angeschlossenen Ports zurückgibt.
Die zurückgegebenen Ports werden dann durchsucht um zu sehen ob der FPGA angeschlossen ist:

```
1  from serial.tools import list_ports
2  # Iterate through all the connected COM-Ports on the PC and return a
       Flag
3  # indicating if the desired Port was found and a List containing all
       found Ports
4  def find_all_Ports(self, search_string="Silicon Labs"):
5    # Create empty list to store all available Ports
6    ports = []
7    # Get all the connected Ports
8    portData = list_ports.comports()
9    # Create three empty lists to store ports
10   temp_ports = []
11   correct_ports = []
12   other_ports = []
13   # Iterate through the connected Ports
14   # => Doesn't loop if there are no Ports connected
15   for i in range(0, len(portData)):
16     port_name = str(portData[i])
17
18     if search_string in port_name:
19       correct_ports.append(port_name.split(" ")[0])
20     else:
21       other_ports.append(port_name.split(" ")[0])
22
23   # append both the Lists of Ports to the List which is returned
24   temp_ports.extend(correct_ports)
25   temp_ports.extend(other_ports)
26   # Override internally stored Port-List if the connected Ports
       changed
27   if temp_ports != self.__ports__:
28     self.__ports__ = temp_ports
29
30   # Return True if the correct Port was found
31   # Also return a List of all connected Ports
32   if len(correct_ports) == 1:
33     return True, self.__ports__
34   else:
35     return False, self.__ports__
36
```

Codeausschnitt 42: serial_handler_find_all_Ports()

set_com_port()

Mithilfe von set_com_port kann man den Port, mit welchem sich der serial_handler verbindet, verändern.
Um hier möglichst viele Fehler zu vermeiden müssen mehrere Sachen kontrolliert werden:

- Ob der Port ein String ist und nicht leer ist
- Ob der Port am PC angeschlossen ist

```
1  # Sets the COM-Port used by this class and frees up the old Port-
       Connection
2  # Doesn't automatically connect to the new Port
3  # Can raise InvalidPortError
4  def set_com_port(self, com_port_v):
5    if (type(com_port_v) == str) and (com_port_v != "") and (com_port_v
       in self.__ports__):
6      self.__logger__.info("Set COM-Port to " + com_port_v)
7      self.port = com_port_v
8    else:
9      raise InvalidPortError("set_com_port()&Something went terribly
       wrong\r\n")
10
```

Codeausschnitt 43: serial_handler_set_com_port()

connect_with_board()

Connect_with_board verbindet die GUI mit einem Port und setzt die Flagge zum Stoppen der Threads zurück. Wenn man versucht sich zu verbinden müssen mehrere Fälle abgesichert werden:

- Wenn noch kein Port verbunden ist
- Wenn die Threads noch laufen
- Wenn schon ein Port verbunden ist, man sich aber mit einem anderen Port verbinden will.

```
1  # Open Serial Port with needed settings to connect to the FPGA
2  def connect_with_board(self) -> bool:
3    # Check if Serial-Connection is not initialized
4    if (self.__ser__ is None):
5      try:
6        self.__ser__ = Serial(
7          port=self.port,
8          baudrate=self.baudrate,
9          bytesize=EIGHTBITS,
10         parity=PARITY_NONE,
11         stopbits=STOPBITS_ONE,
12         timeout=self.timeout,
13       )
14       self.start_thread()
15       return True
16     exce
17   elif ((self.__write_thread__.is_alive()) or (self.__read_thread__.
      is_alive())):)
18   elif (self.port != ""):
19     # Close the Port so it can be modified
20     self.stop_connection()
21     # Reset Terminate-Flag for the Thread
22     self.__terminate_thread__ = False
23     try:
24       self.__ser__.port = self.port
25       self.__ser__.baudrate = self.baudrate
26       self.__ser__.open()
27       self.start_thread()
28       return True
29     except Exception as e:
30       pass
31   else:
32     pass
33   return False
34
```

Codeausschnitt 44: serial_handler_connect_with_board()

Read-Thread

Die __handle_read__-Subroutine wird im Read-Thread ausgeführt.
Um hier eine sichere Ausführung zu garantieren, muss man überprüfen, ob das Board die ganze Zeit verbunden ist. Da ein User das Board einfach abstecken kann und somit kein sicherer Verbindungsabbau gewährleistet ist.
Wenn nun ein verbundenes Gerät Daten an die GUI sendet, müssen diese gebuffered werden, wenn ein komplettes Datenpaket angekommen ist, wird dieses mithilfe des request_constructor verarbeitet und an die GUI gesendet.

```
1  def __handle_read__(self):
2    # Check if the neccesary conditions are met for the Thread to start
      /run
3    while (self.__ser__.is_open) and (not self.__terminate_thread__):
4      # Try-Except to catch unplugging of the board
5      try:
6        # Reads all Bytes if there are any or blocks until at least 4
     Bytes are available
7        # Add the read Bytes to the Input-Buffer
8        self.__input_buffer__.extend(self.__ser__.read(self.__ser__.
     in_waiting or 4))
9      # Log Error if the Board was unplugged
10     except:
11       self.__logger__.critical("Board was unplugged while Thread was
     still active")
12     # Execute Handler-Code if Something was read
13     else:
14       # Check if there already is a full dataset available
15       while len(self.__input_buffer__) >= 4:
16         # Handle the first 4 Bytes
17         dataset = self.__request_constructor__.handle_read(self.
     __input_buffer__[0:4])
18
19         self.__log_read_cmd__(self.__input_buffer__[0:4],dataset)
20
21         # Remove the first 4 Bytes from the Buffer
22         del self.__input_buffer__[0:4]
23         if self.data_transmitter.full() != True:
24           # The Escape-Sequence is transmitted as a custom Dataset-
     Object
25           self.data_transmitter.put(dataset)
26           self.data_valid.set()
27
```

Codeausschnitt 45: serial_handler__handle_read__()

Write-Thread

Die __handle_write__-Subroutine wird im Write-Thread ausgeführt.
Wie im Read-Thread muss hier überprüft werden, ob das Board durchgehend verbunden ist, um eine sichere Verarbeitung zu garantieren. Wenn nun die GUI Daten über die Message-Queue schickt, sendet der Thread diese an das Board. Hier können mehrere Typen von Daten geschickt werden:

- Ein Integer als Escape-Sequence welche vom request_constructor aufgefüllt wird
- Der Bytes-, oder Bytearray-Datentyp welcher direkt mit PySerial gesendet werden kann.
  Hier wird sichergestellt das die Daten 4 Byte lang ist.
- Ein Datenset

Wird ein Datentyp übertragen der hier nicht vorkommt wird ein Error geworfen.

```
1  def __handle_write__(self):
2    # Check if the neccesary conditions are met for the Thread to start
      /run
3    while (self.__ser__.is_open) and (not self.__terminate_thread__):
4      # Wait for an incoming Command using a Flag and a small Delay
5      if self.__cmd_valid__.wait(self.timeout):
6        # Work through all Commands if there are multiple
7        while (self.__cmd_receiver__.empty() is False):
8          cmd = self.__cmd_receiver__.get()
9          if ((type(cmd) is int) and (0 <= cmd <= 255)):
10           # Turn Integer into Command using the request_constructor
11           self.__write_four_bytes__(self.__request_constructor__.
     construct_request(cmd))
12         elif (((type(cmd) is bytes) or (type(cmd) is bytearray)) and
     (len(cmd) == 4)):
13           # Directly write commnad
14           self.__write_four_bytes__(cmd)
15         elif ((type(cmd) is list) and (len(cmd) == 4)):
16           # Directly write commnad
17           self.__write_four_bytes__(bytes(cmd))
18         elif (type(cmd) is Dataset):
19           self.__write_four_bytes__(self.__request_constructor__.
     construct_literal_request(cmd.dataset.value))
20         else:#Invalid Data-Type=>Assume to shut down
21           self.__cmd_receiver__.task_done()
22           self.__cmd_valid__.clear()
23           break #Tell the UI that one Command is processed
24         self.__cmd_receiver__.task_done()
25       self.__cmd_valid__.clear()
26
```

Codeausschnitt 46: serial_handler__handle_write__()

write_request()

Write_request kann von der GUI verwendet werden um ein Datenpaket an den Write-Thread zu senden:

```
1  # automatically construct Request with the specified Escape_seqeunce
       and send it to the board
2  def write_request(self,cmd):
3    # Put Command in the Queue
4    self.__cmd_receiver__.put(cmd)
5    # Set Flag so the Write_Thread will become active
6    self.__cmd_valid__.set()
```

Codeausschnitt 47: serial_handler_write_request()

__write_four_bytes__()

Die Methode __write_four_bytes__ ist die Einzige, die Daten an das Board senden kann.Sie kann ausserdem nur vom Write-Thread aufgerufen werden, daher müssen hier keine Sicherheitschecks gemacht werden. Zur Sicherheit wird der Code in einem Try-Catch-Block ausgeführt, falls die Verbindung mit dem Board unterbrochen wird:

```
1  # Write one Command (= 4 Bytes) to the FPGA safely
2  def __write_four_bytes__(self, cmd) -> bool:
3    # Check if the Port is open and the Command is valid
4    try:
5      # Log every Command which is sent to log.json
6      self.__log_write_cmd__(cmd,escape_sequences.return_name_by_code(
       int(cmd[0])>>1))
7      # Write the Data and return the Number of Bytes written
8      return self.__ser__.write(cmd)
9    except Exception:
10     self.__logger__.critical("Tried writing to closed Port")
11     raise PortNotOpenError("__write_four_bytes__()&Port is not open:
       __write_four_bytes__")
12       return 0
13
```

Codeausschnitt 48: find_all_Ports()

return_raw_request()

Return_raw_request wurde für das Debugging-Fenster implementiert, um die Datenpakete welche manuell gesendet werden können, wenn diese unvollständig sind, aufzufüllen.

```
1 # Return a raw Request as a Bytes-Array ( used for Debugging Panel )
2 def return_raw_request(self,escape_sequence):
3   return self.__request_constructor__.construct_request(
     escape_sequence)
4
```

Codeausschnitt 49: serial_handler_return_raw_request_()

start_thread()

Start_thread startet Read-, und Write-Thread indem es zwei private Methoden (diese werden später erklärkt) aufruft:

```
1 def start_thread(self):
2   self.__logger__.info("Starting Communication-Thread")
3   self.__init_thread__()
4   self.__start_thread__()
5
```

Codeausschnitt 50: serial_handler_start_thread()

__init_thread__()

Die private Methode __init_thread__, initialisiert die beiden Threads für den Fall, dass die Threads abgestürzt sind, z.B. wenn das verbundene Board ohne Vorwarnung ausgesteckt wurde:

```
1  def __init_thread__(self):
2    # Check if the Write-Thread is already running
3    if (not self.__write_thread__.is_alive()):
4      # If not, create the Thread
5      self.__write_thread__ = Thread(target=self.__handle_write__,
      daemon=True)
6    # Check if the Read-Thread is already running
7    if (not self.__read_thread__.is_alive()):
8      # If not, create the Thread
9      self.__read_thread__ = Thread(target=self.__handle_read__,daemon=
      True)
10
```

Codeausschnitt 51: serial_handler__init_thread__()

__start_thread__()

Die private Methode __start_thread__ überprüft zuerst, ob die Threads nicht im Betrieb sind, und startet Read-, und Write-Thread daraufhin.

```
1 def __start_thread__(self):
2   # Check if the Serial-Port is open, as the Thread can only be
3   # successfully started when the Serial-Port is open
4   if (self.__ser__.is_open):
5     if (not self.__write_thread__.is_alive()):
6       self.__write_thread__.start()
7     if (not self.__read_thread__.is_alive()):
8       self.__read_thread__.start()
9
```

Codeausschnitt 52: serial_handler__start_thread__()

stop_thread_and_connection()

Stop_thread_and_connection stoppt zuerst Read-, und Write-Thread und schließt dann die Verbindung mit dem Board:

```
1 # Closes Port-connection and the Communication-Thread
2 def stop_thread_and_connection(self):
3   # Stop the Read-, and Write-Thread
4   self.stop_thread()
5   # Disable any further communication with the FPGA
6   self.stop_connection()
7
```

Codeausschnitt 53: serial_handler_stop_thread_and_connection()

stop_thread()

Stop_thread setzt die Flagge zum Stoppen der Threads und wartet dann bis beide Threads gestoppt haben:

```
1  # Stop the Thread that is communicating with the FPGA
2  # Can raise RunTimeError as far as i know
3  def stop_thread(self):
4    if (self.__write_thread__.is_alive()) or (self.__read_thread__.
     is_alive()):
5      self.__terminate_thread__ = True
6
7      self.__read_thread__.join()
8      self.__write_thread__.join()
9      self.__logger__.debug("Threads stopped")
10
```

Codeausschnitt 54: serial_handler_stop_thread()

stop_connection()

Stop_connection schließt die Verbindung mit dem Board auf einem sicheren Weg. Dafür wird überprüft dass:

- Die Verbindung überhaupt offen ist
- Read-, und Write-Thread gestoppt sind

```
1  # Closes Port-connection if it isn't being used by the Communication
     Thread
2  def stop_connection(self):
3    if self.__ser__ is not None:
4      # Check if Serial-Connection was initialized, is open and
5      # not being used by the Communication-Thread
6      if (self.__ser__.is_open == True) and (not self.__write_thread__.
     is_alive()) and (not self.__read_thread__.is_alive()):
7        # Disable any further communication with the FPGA
8        self.__ser__.close()
9        self.__logger__.warn("Stopped Connection")
10
```

Codeausschnitt 55: serial_handler_stop_connection()

5.3.3.3.2 serial_handler_storage_interface

Das serial_handler_storage_interface sorgt dafür, dass die GUI auf die Werte, die im Request_constructor gespeichert werden, zugreifen kann, da dieser im serial_handler als privates Feld implementiert ist.
Das Interface implementiert Getter-, und Setter-Methoden für die Werte die im request_constructor gespeichert werden. Die Getter geben einfach den Wert zurück, während die Setter noch einige Sicherheitschecks durchlaufen, bevor sie den Wert setzen.

```
1 def get_engine_start_stop(self):
2   return self.__request_constructor__.engine_details.
     Engine_Start_Stop
3
4 def get_engine_enable(self):
5   return self.__request_constructor__.engine_details.Engine_Enable
6
7 def get_engine_status(self):
8   return self.__request_constructor__.engine_details.Engine_Status
9
```

Codeausschnitt 56: serial_handler_serial_handler_storage_interface - Getter

```
1 def set_engine_start_stop(self,engine_start_stop):
2   if ((type(engine_start_stop) == int) and
3   ( 0 <= engine_start_stop <= ((2 ** 1) - 1))):
4     self.__request_constructor__.engine_details.Engine_Start_Stop =
     engine_start_stop
5
6 def set_engine_enable(self,engine_enable):
7   if ((type(engine_enable) == int) and
8   ( 0 <= engine_enable <= ((2 ** 1) - 1))):
9     self.__request_constructor__.engine_details.Engine_Enable =
     engine_enable
10
11 def set_engine_status(self,engine_status):
12   if (type(engine_status) == engine_status):
13     self.__request_constructor__.engine_details.Engine_Status =
     engine_status
14
```

Codeausschnitt 57: serial_handler_serial_handler_storage_interface - Setter

5.3.3.3.3 serial_handler_write_interface

Das serial_handler_write_interface wurde implementiert, damit die GUI komplizierte Anfragen ganz einfach mithilfe einer Methode an den serial_handler schicken kann.
Im nachfolgenden Codeausschnitt ein Beispiel, das die gesamte Konfiguration auf einmal zum FPGA sendet:

```
1  # Transmits the Engine-Configuration to the board
2  def transmit_configuration(self):
3    # Return-Array for the UI's Progressbar
4    dataset_strings = []
5    # Create an empty array to easily store the ranges of the required
6    # Escape-Sequences
7    escape_sequences = []
8    escape_sequences.extend(range(1,36))
9    escape_sequences.extend(range(45,73))
10   escape_sequences.extend(range(74,102))
11   escape_sequences.extend(range(103,107))
12   escape_sequences.extend(range(126,128))
13
14   # Transmit all of the Escape-Sequences which belong to the
      configuration
15   for x in escape_sequences:
16     # Get the Data from the UI and return it in a 4-Byte-Array
17     cmd = self.__request_constructor__.construct_request(x)
18     # Append the 4-Byte-Array to the Return-Array as a String
19     # for it to be displayed in the UI with the Progressbar
20     dataset_strings.append(str(cmd))
21     # Pass the Data to the Serial-Handler's Write-Thread to send it
22     self.write_request(cmd)
23
24   # Return Escape_seqeunces and Dataset-Names for Progressbar
25   return(escape_sequences, dataset_strings)
26
```

Codeausschnitt 58: serial_handler_write_interface - Beispiel

5.3.3.3.4 Dataset

Die „Dataset“-Klasse wurde zusammen mit dem „datatype“-Enum zur einfacheren Handhabung von erhaltenen Datenpaketen entwickelt.
In einem Dataset werden Daten mit ihrem Datentyp zusammen gespeichert.
Der Datentyp wird in zwei Teile aufgeteilt: Der Datengruppe und den Gruppenindex.
Diese müssen beide gespeichert werden, da manche Datentypen Gruppen bilden z.B. die CA-Offset-Werte der Zylinder sind eine Gruppe mit 24 Indizes. Die wirklichen Daten sind entweder ein Integer-Wert oder ein Array von Flags.
Der Datentyp gibt an, welche Daten sich in dem dataset befinden (hier ein Auschnitt):

```
1  class Dataset:
2    def __init__(self,datatype_v,integer_data=None,active_flag_data=
       None,group_index=None):
3      # Create Fields for the actual Data
4      self.integer_data = integer_data
5      self.active_flag_data = active_flag_data
6      # Specifies which Index inside a Group the data belongs to
7      self.group_index = group_index
8      try:
9        # Specifies the group of data this Dataset belongs to
10       self.datatype = Datatype(datatype_v)
11     except:
12       # If the Datatype is not defined indicate an Error
13       self.datatype = Datatype(0)
```

Codeausschnitt 59: serial_handler_storage_dataset Klassen-Deklaration

```
1  class Datatype(Enum):
2    NOT_DEFINED = 0
3    RPM_BOTTOM = 1
4    RPM_TOP = 2
5    ACCELERATION = 3
6    GEAR_TEETH = 4
7    DIGITAL_DELAY_LIMIT = 5
8    CAM_RESET_DURATION = 6
9    ...
10   ...
11
```

Codeausschnitt 60: serial_handler_storage_dataset

Der Group-Index gibt bei einem Gruppen-Datentyp an, welchem spezifischen Datentyp die Daten innerhalb einer Datentyp-Gruppe angehören:

```
1  # Function-Definitions to get the correct Index of the Function in
       the Dictionary
2  # for Read-, and Write-Escape-Sequences
3  def get_write_escape_sequence_group(self,escape_sequence):
4    if ((escape_sequence >= 7) and (escape_sequence <= 30)):
5      # Return CA-Offset-Cylinder-Group
6      return 7
7    elif (((escape_sequence >= 38) and (escape_sequence <= 44)) | (
       escape_sequence == 73) | (escape_sequence == 102)):
8      # Return Write-To-Ram-Group
9      return 38
10   elif ((escape_sequence >= 45) and (escape_sequence <= 68)):
11     # Return Period-Knocksensor/Period-Cylinder-Group
12     return 45
13   elif ((escape_sequence >= 78) and (escape_sequence <= 101)):
14     # Return Period-Ignition-Voltage-Group
15     return 78
16   elif ((escape_sequence >= 1) and (escape_sequence <= 106)) or ((
       escape_sequence >= 124) and (escape_sequence <= 127)):
17     # Return the Escape-Sequence because it is valid and can be sent
18     return escape_sequence
19   else:
20     # Return 0 to indicate that the Escape-Sequence is not valid and
       cannot be sent
21     return 0
22
```

Codeausschnitt 61: serial_handler_storage_request_constructor

Um Fehler auf höhreren Ebenen vorzubeugen wird sichergestellt, dass Daten korrekt bereitgestellt wurden. Falls keine Daten bereitgestellt wurden, wird ein Fehler-Ereignis ausgeloest, das mit “catch“-Anweisungen verarbeitbar ist

```
1 def return_payload(self):
2   if (self.integer_data is not None):
3     return self.integer_data
4   elif (self.active_flag_data is not None) and (len(self.
     active_flag_data) == 24):
5     return self.active_flag_data
6   else:
7     raise ValueError("No Data or corrupted Data was given")
```

Codeausschnitt 62: serial_handler_storage_dataset

5.3.3.3.5 engine_details

Die „engine_details“-Klasse wurde entwickelt, um einheitlichen Zugang auf alle gespeicherten Motordaten zu haben und diese einfach initialisieren zu können.
Hier ein Auschnitt:

```
1  class engine_details:
2    def __init__(self):
3      self.Engine_Start_Stop = 0
4      self.Engine_Enable = 0
5      # 2-Bit Status-Variable
6      self.Engine_Status = engine_status(0)
7      self.RPM_TOP = 0
8      self.RPM_BOTTOM = 0
9      self.Acceleration = 0
10     ...
11
```

Codeausschnitt 63: serial_handler_storage_engine_details

Der „Engine-Status“ wird im VHDL-Code mithilfe eines 2-Bit Datentypes definiert. Hierfür wurde im Python-Code mithilfe der „engine_status“-Enumeration dieser Datentyp mit Namen für den jeweiligen “engine-status“ nachgebaut:

```
1  class engine_status(Enum):
2    NoDrive = 0
3    DriveUp = 1
4    Drive = 2
5    DriveDown = 3
6    # Helper Function for GUI to display the Name correctly
7    def name_from_int(code):
8      if (code == 0):
9        return "No Drive"
10     elif (code == 1):
11       return "Drive Up"
12     elif (code == 2):
13       return "Drive"
14     elif (code == 3):
15       return "Drive Down"
16     else:
17       # Theoretically impossible to reach but never say never
18       raise ValueError("Invalid Engine-Status")
19
```

Codeausschnitt 64: serial_handler_storage_engine_details

5.3.3.3.6 request_constructor

Der request_constructor wurde entwickelt, um die eingehenden und ausgehenden Datenpakete einfach und modular zu verarbeiten.
Dies wird mithilfe eines Dictionaries gemacht, welches die
Escape_sequence auf die jeweilige Verarbeitungs-Methode „mappt“.
Dies muss seperat für Read-, und Write-Pakete implementiert werden, da die Escape_sequences verschiede Daten enthalten.
Im request_constructor wird ausserdem ein “engine_details“-Objekt instanziert, um dem request_constructor einfachen Zugriff auf die nötigen Daten zu ermöglichen.

```
1  class write_request_constructor():
2
3    def __init__(self):
4      # Create Data-Container-Object
5      # Stores all important Variables relating to the Engine
6      self.engine_details = engine_details()
7
8      # Store all Write-Functions inside a Dictionary to avoid endless
9      # If-Elif-Statements and improve Performance
10     self.write_dict = {
11       1 : self.set_rpm,
12       2 : self.set_rpm_top,
13       3 : self.set_acceleration,
14       4 : self.set_gear_teeth,
15       ...
16     }
17
18     # Store all Read-Functions inside a Dictionary to avoid endless
19     # If-Elif-Statements and improve Performance
20     self.read_dict = {
21       1 : self.read_rpm_bottom,
22       2 : self.read_rpm_top,
23       3 : self.read_acceleration,
24       4 : self.read_gear_teeth,
25       ...
26     }
27
```

Codeausschnitt 65: serial_handler_storage_request_constructor

Dann wurde eine einfache Methode zum Konstruieren von Datensets implementiert, die einfach die entsprechende Methode im Dictionary aufruft. Falls eine Anfrage auf eine Escape-Sequence hereinkommt, die nicht im request_constructor implementiert ist, wird als Default ein Array mit Nullen zurückgegeben:

```
1  # Turns an Escape-sequence into a 4-Byte Command ready to be sent
2  # by the Serial-Handler
3  def construct_request(self,escape_sequence):
4    # Check if a correct escape_seqeunce was given
5    if escape_sequence != 0:
6      # Convert List to Bytes (could theoretically raise ValueError-
       Exception
7      # if a Value in the List is bigger than 255 => is practically not
       possible
8      # because every Function is written so that Values can only be
       between 0 and 255)
9      return bytes(self.write_dict.get(escape_sequence,lambda x
       :[0,0,0,0])(escape_sequence))
10   else:
11     # Send back array which will be detected by serial_handler and
       not be send
12     return []
13
```

Codeausschnitt 66: serial_handler_storage_request_constructor

Hier eine Beispiel-Methode aus dem Dictionary, welche mithilfe von Bit-Shifts die Daten in die richtige Form für den FPGA bringt:

```
1  def set_rpm(self,escape_sequence):
2   cmd = []
3   # RPM_TOP and RPM_BOTTOM are 12 Bit Integers and must therefore be
      split
4   # => RPM_TOP = 23 downto 12
5   # => RPM_BOTTOM = 11 downto 0
6   cmd.append(escape_sequence<<1)
7   cmd.append((self.engine_details.RPM_BOTTOM>>4)&((2**8)-1))
8   cmd.append((((self.engine_details.RPM_BOTTOM<<4)&((2**8)-(2**4)))) +
        ((self.engine_details.RPM_TOP>>8)&((2**4)-1)))
9   cmd.append((self.engine_details.RPM_TOP)&((2**8)-1))
10  return cmd
```

Codeausschnitt 67: serial_handler_storage_request_constructor

Bei den Verarbeitungen von einkommenden Datenpaketen wird zuerst der Gruppen-Index der Daten berechnet, da alle Datentypen welche zur gleichen Gruppe mithilfe der gleichen Funktion verarbeitet werden können.
Damit muss für jede Gruppe nur eine Handler-Methode erstellt werden:

```
1  def handle_read(self,cmd):
2    if (((type(cmd) == bytearray) or (type(cmd) == bytes)) and (len(cmd
     ) == 4)):
3      escape_sequence_group = self.get_read_escape_sequence_group((cmd
     [0]>>1)&127)
4      return self.read_dict.get(escape_sequence_group,self.read_default
     )(cmd)
5    return Dataset(0)
6
7  def get_read_escape_sequence_group(self,escape_sequence):
8    if (escape_sequence >= 7) and (escape_sequence <= 30):
9      return 7
10   elif (escape_sequence >= 120) and (escape_sequence <= 127):
11     return 120
12   elif (escape_sequence >= 1) and (escape_sequence <= 106):
13     return escape_sequence
14   else:
15     return 0
16
```

Codeausschnitt 68: serial_handler_storage_request_constructor

Hier eine Beispiel-Methode aus dem Dictionary, welche mithilfe von Bit-Masken die Nutzdaten aus dem gelesenen Datenpaket rekonstruiert:

```
1  def read_rpm_top(self,cmd):
2   Rpm_Top =  ((cmd[0] & ((2**1)-1)) * (2**24)) +\
3      ((cmd[1] & ((2**8)-1)) * (2**16)) +\
4      ((cmd[2] & ((2**8)-1)) * (2**8)) +\
5      ( cmd[3] & ((2**8)-1))
6   return Dataset(((cmd[0]>>1) & ((2**7)-1)),integer_data=Rpm_Top)
```

Codeausschnitt 69: serial_handler_storage_request_constructor

Zur Sicherheit wurde ausserdem eine Rückfall-Methode implementiert, die aufgerufen wird, wenn die Escape-Sequence im Dictionary nicht existiert:

```
1 def read_default(self,cmd):
2   data = ((cmd[0] & ((2**1)-1)) * (2**24)) +\
3       ((cmd[1] & ((2**8)-1)) * (2**16)) +\
4       ((cmd[2] & ((2**8)-1)) * (2**8)) +\
5       ( cmd[3] & ((2**8)-1))
6   # print("Unknown data was read : " + str(cmd))
7   return Dataset((cmd[0] >> 1) & ((2**7)-1),integer_data=data)
```

Codeausschnitt 70: Default-Read-Methode

5.3.4 Testung des Back-End

Um das Back-End zu testen, wurde:

- ein Arduino-Programm, das serielle Daten unverändert zurückschickt.
- ein Arduino-Programm, das Zufallszahlen seriell zurückschickt.
- mehrere Unit-Tests in Python entwickelt.

5.3.4.1 Arduino Repeater Test

Hier wurde ein Arduino-Program geschrieben, welches Daten die der Arduino erhält, unverändert wieder zurück an die GUI sendet.

```
1  uint8_t x = 0;
2  void setup() {
3          // initialize digital pin LED_BUILTIN as an output.
4          pinMode(LED_BUILTIN, OUTPUT);
5          // Setup Serial-Interface
6          Serial.begin(115200);
7          // Setup Random-Number-Generator
8          randomSeed(analogRead(0));
9  }
10 void send_back();
11
12 // the loop function runs over and over again forever
13 void loop() {
14         send_back();
15 }
16 void send_back(){
17         digitalWrite(LED_BUILTIN, LOW);
18         if (Serial.available() > 0){
19                 digitalWrite(LED_BUILTIN, HIGH);
20                 x = Serial.read();
21                 Serial.write(x);
22         }
23 }
```

Codeausschnitt 71: Arduino Repeater Code

5.3.4.2 Arduino Zufallszahlen Test

Hier wurde ein Arduino-Program aufgesetzt, dass Zufallsdaten zurückschickt (und die Onboard-LED einschaltet), wenn ein Datenpaket empfangen wird.

```
uint8_t x = 0;
uint8_t counter = 0;
#define NUMBER_COMMANDS 44
uint8_t cmds[NUMBER_COMMANDS] =
{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,
25,26,27,28,29,30,31,32,33,34,35,44,45,47,48,49,69,74,105,120};

void setup() {
        // initialize digital pin LED_BUILTIN as an output.
        pinMode(LED_BUILTIN, OUTPUT);
        // Setup Serial-Interface
        Serial.begin(115200);
        // Setup Random-Number-Generator
        randomSeed(analogRead(0));
}

void send_random();

// the loop function runs over and over again forever
void loop() {
        send_random();
}
void send_random(){
        if (Serial.available() > 0){
                x = Serial.read();
                if ((x == 248) || (x == 250)) {
                        digitalWrite(LED_BUILTIN, HIGH);
                        counter = 0;
                        while (counter < NUMBER_COMMANDS){
                                Serial.write(2 * cmds[counter]);
                                Serial.write(random(254));
                                Serial.write(random(254));
                                Serial.write(random(254));
                                counter ++;
                        }
                        digitalWrite(LED_BUILTIN, LOW);
                }
        }
}
```

Codeausschnitt 72: Arduino Zufallszahlen Code

5.3.4.3 Python Transmit Test

Dieser Test kann mit dem FPGA oder einem der Arduino-Programme verwendet werden. Hier kann man mithilfe der “read_requests“-Variable einstellen, wie oft ein “Transmit“-Befehl gesendet wird. Die Antworten werden dann in der Command-Line angezeigt:

```
1  def test_serial_handler():
2    read_requests = 10000
3    try:
4      boardconnect = Port_Connection(port="COM9")
5      if boardconnect.connect_with_board():
6            time.sleep(2)
7            start = timer()
8            switch_byte = 0
9            counter = 0
10           write_requests = 0
11           while counter < read_requests:
12             print("Writing : " + str(switch_byte + 124))
13             boardconnect.write_request(switch_byte + 124)
14             write_requests = write_requests + 1
15             while True:
16                   if (boardconnect.data_valid.wait(0.01)):
17                     switch_byte = (1 if (switch_byte == 0) else 0)
18                     while not boardconnect.data_transmitter.empty():
19                       print(boardconnect.data_transmitter.get().
                             display())
20                       boardconnect.data_transmitter.get()
21                       boardconnect.data_valid.clear()
22                   else:
23                     counter = counter + 1
24                     break
25           print("Written " + str(write_requests) +
26           "Requests and received " + str(read_requests) +
27           " Answers in " + str(timedelta(seconds=timer()-start)))
28   except Exception as e:
29         print("Error : " + e.__str__())
30         boardconnect.stop_thread_and_connection()
31         boardconnect.stop_thread_and_connection()
```

Codeausschnitt 73: Transmit Test

5.3.4.3.1 Python Konfigurations-Test

Beim Konfigurations-Test wird in jedem Schleifendurchlauf jeder Parameter neu gesetzt und dann wieder abgefragt:

```
def test_transmit_config():
  try:
    boardconnect = Port_Connection(port="COM7")
    if boardconnect.connect_with_board():
      time.sleep(2)
      total_err = 0
      for i in range(0,10):
        err = 0
        total = 0
        boardconnect.set_engine_start_stop(i)
        ...
        boardconnect.set_ignition_voltage_config(i)
        for x in range(0,24):
          boardconnect.set_cylinder_offset_cycles(i,x)
          ...
          boardconnect.set_active_flags_ignition_voltage(i)
        boardconnect.transmit_configuration()
        boardconnect.write_request(124)
        if boardconnect.data_valid.wait(0.1):
          while not boardconnect.data_transmitter.empty():
            dataset = boardconnect.data_transmitter.get()
            total = total + 1
            if (dataset.return_payload() != i):
              err = err + 1
              print("Error on " + dataset.datatype.name)
              print(str(dataset.return_payload()))
              print("Value = " + str(i))
              boardconnect.data_valid.clear()
          if (err == 0):
            print("No Errors on " + str(i))
          else:
            total_err = total_err + err
        else:
          print("Critical : No Response")
    print("Total Errors " + str(total_err))
  except Exception as e:
    boardconnect.stop_thread_and_connection()
  boardconnect.stop_thread_and_connection()
```

Codeausschnitt 74: Konfigurations-Test Code

5.3.4.3.2 Python BRAM-Test

Nachdem der Konfigurations-Test funktioniert, musste sichergestellt werden, dass auch das Übertragen von Daten an den internen BRAM des FPGA funktioniert.
Leider kann man dies nicht direkt am FPGA testen, da es keine Möglichkeit gibt den BRAM auszulesen. Anstatt dessen müssen die Kommandos an das Arduino-Repeater-Programms geschickt werden und dann am Back-End zurück gerechnet werden:

```
1  def test_write_to_ram():
2    try:
3      boardconnect = Port_Connection(port="COM10")
4      if boardconnect.connect_with_board():
5        time.sleep(2)
6        err = 0
7        total = 0
8        data = []
9        for i in range(0,4095):
10         data.append(i)
11         rams = [0,1,2,3,4,5,7,8]
12         for i in rams:
13         boardconnect.write_to_ram(i,data)
14         if boardconnect.data_valid.wait(0.1):
15           y = 0
16           time.sleep(2)
17           while not boardconnect.data_transmitter.empty():
18             dataset = boardconnect.data_transmitter.get()
19             total = total + 1
20             if (dataset.return_payload()!=(((y*(2**12))+(data[y])))):
21               print("Error on "+str(y)+" on RAM "+str(rams[i]))
22               print(dataset.return_payload())
23               print(((y*(2**12))+(data[y])))
24               err = err + 1
25               y = y + 1
26             boardconnect.data_valid.clear()
27         else:
28           print("Nothing received")
29           print(str(err) + " Errors out of " + str(total) + " Values"
                 )
30   except Exception as e:
31     print("Error : " + e.__str__())
32     boardconnect.stop_thread_and_connection()
33     raise
34   boardconnect.stop_thread_and_connection()
```

Codeausschnitt 75: BRAM-Test Code

5.3.5 Anleitung der graphischen Benutzeroberfläche

5.3.5.1 Einleitung

Das Programm wurde speziell für den „Xilinx Artix 750t“ FPGA entwickelt. Es ist nur mit dem Windows 7/10 Betriebssystem kompatibel.

Für das Programm **muss** Python (Version 3) auf dem PC installiert sein.[31]
Für bessere Lesbarkeit in ausgedruckter Form, wird in der Anleitung der „LightMode“/Helle Modus der GUI verwendet.

5.3.5.2 Erster Start

Beim ersten Start des Programms wird überprüft, ob der Treiber für den FPGA bereits installiert ist. Wenn der Treiber **nicht** installiert ist, wird die Treiberinstallation ausgeführt.

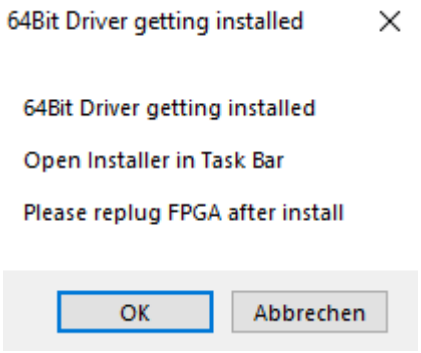


Abbildung 28: Anleitung-Treiberfenster

Um den Treiber zu installieren, auf **OK** klicken.
Bitte stecken Sie nun den FPGA ein bzw. stecken sie in aus und wieder ein, sodass der Treiber auch initialisiert wird.
Danach kann das Programm erneut gestartet werden.

[31] Python download: https://www.python.org/downloads/

5.3.5.3 Hauptfenster

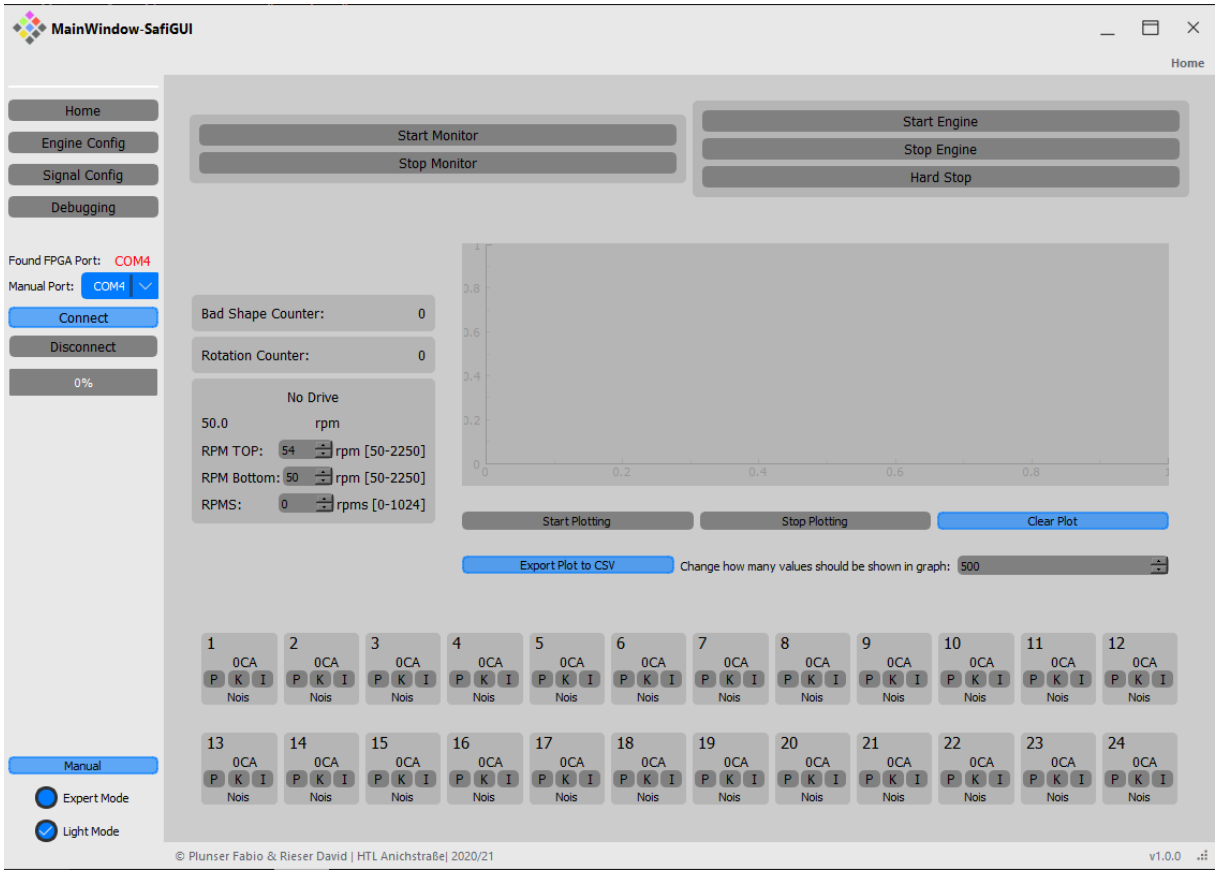


Abbildung 29: GUI-Hauptfenster

Das Hauptfenster ist das Fenster, das beim Start des Programms erscheint.
Es stellt folgende Funktionen dar:

- RPM: Drehzahlkurve als Graph
- RPM-TOP: Maximale Drehzahl
- RPM-Bottom: Minimale Drehzahl
- RPMS $\rightarrow$ RPM Slope: Steigung der RPM Kurve
- Aktivierte Zylinder
- CA: Zündwinkel der Zylinder
- Ob Störungen (P,K,I) für diesen Zylinder aktiviert sind.

Wie in der Abbildungs zu sehen ist, sind die meisten Knöpfe deaktiviert, da kein Gerät mit der GUI verbunden ist. Somit muss zuerst ein Gerät verbunden werden, bevor man die graphische Benutzeroberfläche verwenden kann.

5.3.5.3.1 Seitliches Menü

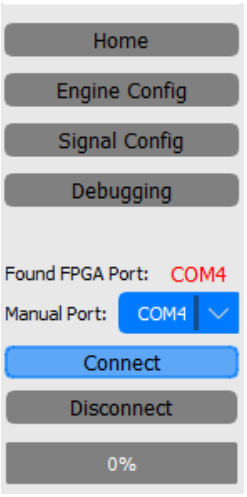


Abbildung 30: Anleitung-seitliches-Menü

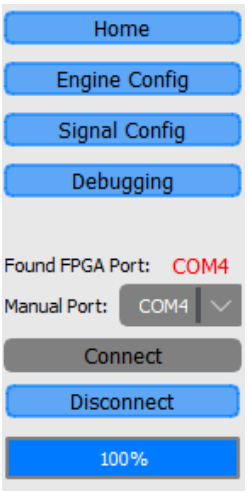


Abbildung 31: Anleitung-seitliches-Menü-2

Das seitliche Menü besitzt folgende Funktionen:

- Zwischen Seiten wechseln
- Anzeige an welchem Port der FPGA angeschlossen ist.
- Auflistung aller verbundenen COM-Ports
- Verbindungsaufbau mit einem angeschlossenen Gerät
- Verbindungsabbau mit einem Gerät

Die oberen vier Knöpfe *Home*, *Engine Config*, *Signal Config*, *Debugging* sind die einzelnen Seiten.
Darunter wird neben *Found FPGA Port* der COM-Port angezeigt, an dem der FPGA angeschlossen ist. Die GUI erlaubt es auch andere Geräte, wie den FPGA anzuschließen, daher wird der Anschluss des FPGAs einzeln angezeigt und in der Port auswahl automatisch ausgewählt. Neben *Manual Port* kann der automatisch ausgewählte Port überschrieben und somit ein anders Gerät verbunden werden. Der Knopf *Connect* startet die Verbindung. Der Knopf *Disconnect* startet die Terminierung der Verbindung.
Unter dem Disconnect-Knopf befindet sich eine Statusanzeige um dem Nutzer zu signalisieren, ob die Verbindung und Initialisierung vollständig ist.

5.3.5.3.2 Home-Seite

Wenn ein Gerät verbunden ist, wird der Start-Monitor Knopf aktiviert. Dieser startet das Auslesen des Geräts.

Abbildung 32: Anleitung-Home-Seite-1

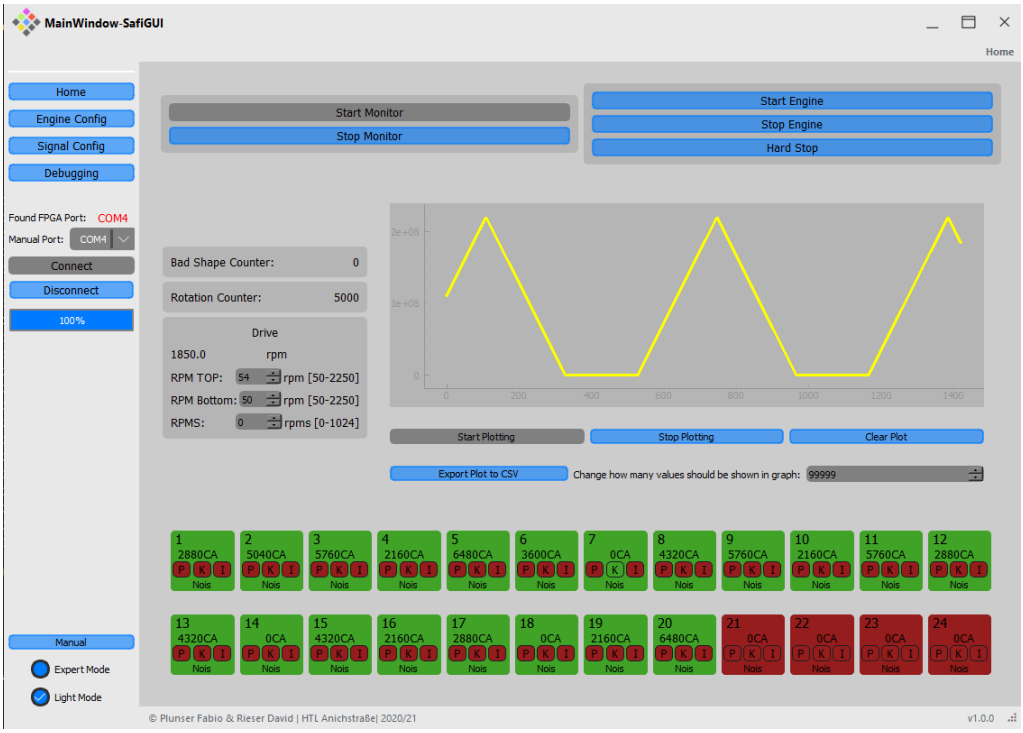


Abbildung 33: Anleitung-Anleitung-Home-Seite-1

Die graphische Benutzeroberfläche stellt dann die eingelesen Daten dar. Man sieht, dass in diesem Fall 20 Zylinder aktiviert sind und und bei Zylinder 7 ist eine Störung beim Klopfsignal aktiviert.

Start-Stop-Hard-Stop-Engine und RPM-Darstellung

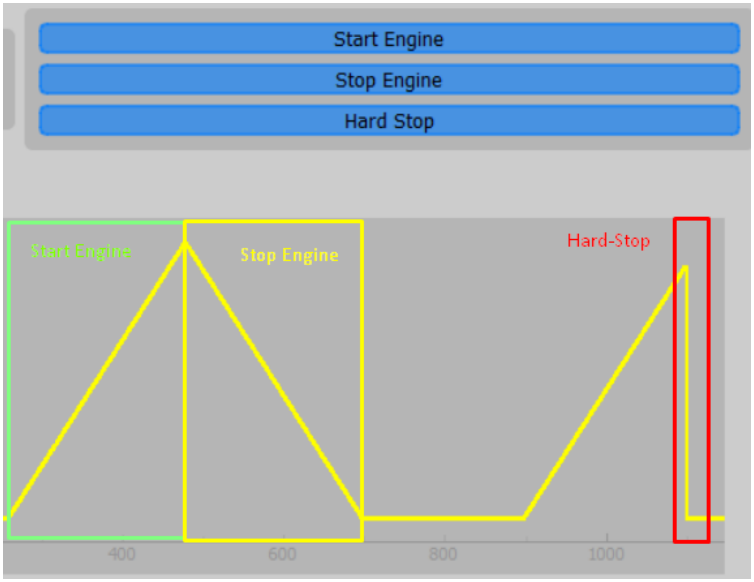


Abbildung 34: Anleitung-Start-Stop-Botton

- Start Engine: Startet den „Motor“ mit der konfigurierten RPM Kurve und Beschleunigung.
- Stop Engine: Stoppt den „Motor“ mit der eingestellen Beschleunigung.
- Hard Stop: Stoppt sofort den „Motor“.

Die RPM Kurve wird dementsprechend im Graphen dargestellt.

RPM Information Home-Seite

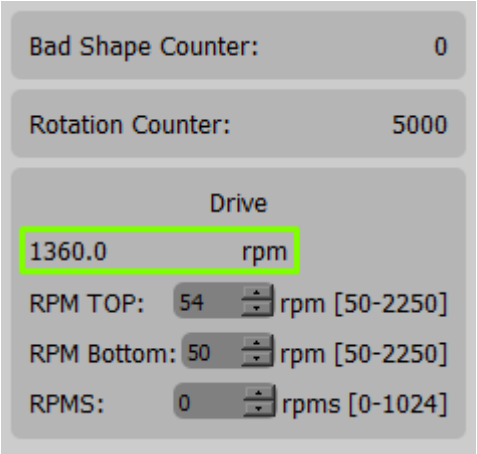


Abbildung 35: RPM-Darstellung

- Bad Shape Counter: Wenn eine Störungen in der Konfiguration aktiviert ist, zählt dieser Counter, wie viele „Böse“ Kurven, also Störungen aufgetreten sind.
- Zyklus Counter: Zählt wie, oft ein Motorzyklus durchgeführt wurde.
- RPM: Zeigt die aktuelle Drehzahl an.
- RPM TOP/BOTTOM: Maximale/Minimale Drehzahl des „Motors“.
- RPMS: Steht für RPM Slope = RPM Steigung, es wird damit die Beschleunigung des „Motors“ eingestellt und kann im Betrieb geändert werden.

Zylinder-Darstellung

Hier werden die aktivierten Zylinder, aktive Störungen (schwarzes Kästchen) für Druck, Klopf, Zündung und Zündwinkel dargestellt. In diesem Fall sieht man, dass 20 Zylinder und eine Störung bei Zylinder 7 aktiviert sind.

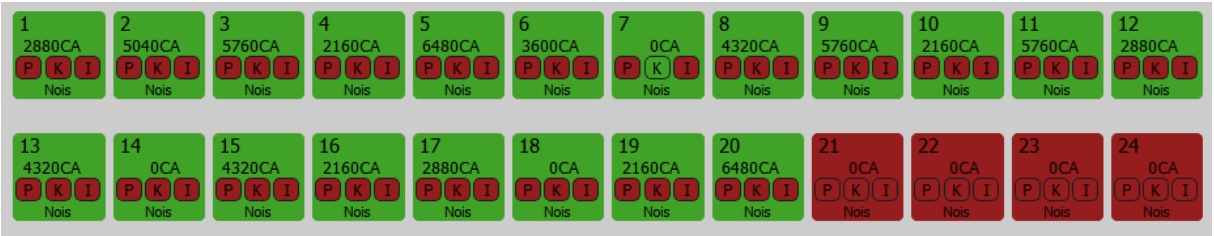


Abbildung 36: Anleitung-Home-Zylinder-Darstellung

5.3.5.4 Engine-Config

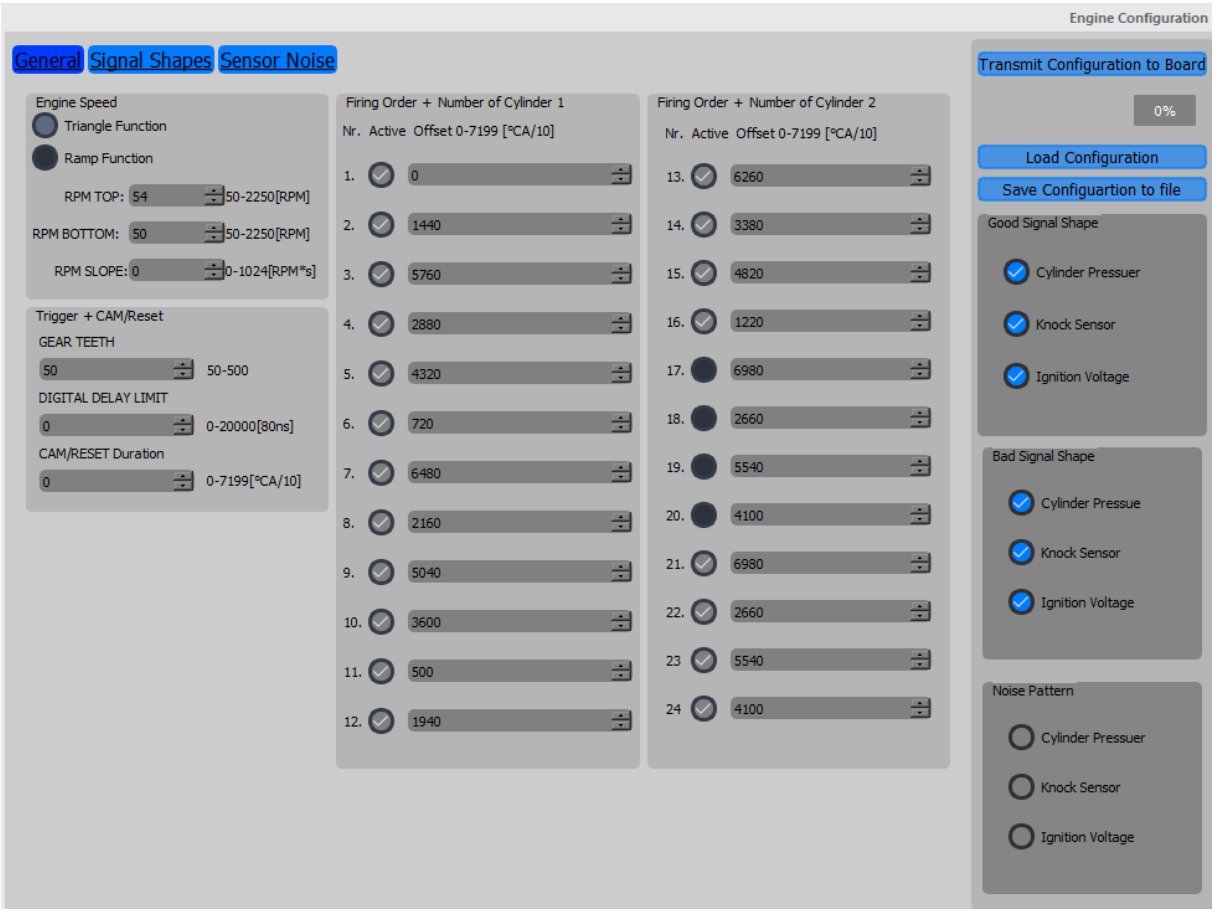


Abbildung 37: EngineConfig-1

Auf der Seite *Engine-Config* können alle Einstellungen des FPGA konfiguriert werden. Für bessere Übersicht werden die Einstellungen unterteilt in **General**, **Signal Shapes**, **Sensor Noise** und das rechte Menü.

Unter *General* befinden sich die Einstellungen der RPM Kurver, Zylinder, Trigger, CAM Shapes. Weiterhin sind dort die Knöpfe zur Übertragung der Konfiguration zum FPGA, Laden und Speichern einer Konfiguration.

Unter *Signal Shapes* befinden sich die Einstellungen für die Temperatur und die Import-Knöpfe für die Good und Bad Shapes.

Unter *Sensor Noise* befinden sich die Störgrößeneinstellungen der Druck, Klopfsensor und Zündungsausgänge.

5.3.5.4.1 General

Engine-Speed

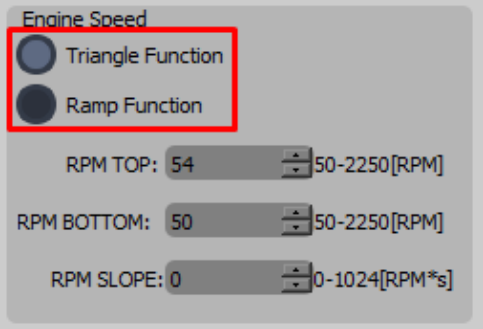


Abbildung 38: Anleitung-Engine-Config-1

Im Bereich Engine Speed wird die RPM Kurve eingestellt. Die markierten Einstellungen wurden von der Vorlage für diese GUI übernommen, jedoch haben diese keinerlei Funktion, da diese Einstellungen im Code des FPGAs nicht vorhanden sind. Wofür die RPM Einstellungen da sind, wurde bei, *RPM-Darstellung und Live-Daten-Änderung* bereits erklärt.

Trigger+CAM/Reset

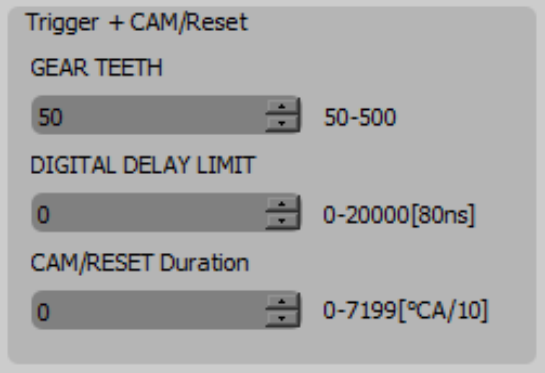


Abbildung 39: Anleitung-Engine-Config-2

Trigger und CAM-Reset sind spezielle Signale, die für die SAFI generiert werden.

Zylinder aktivieren und Zündwinkel

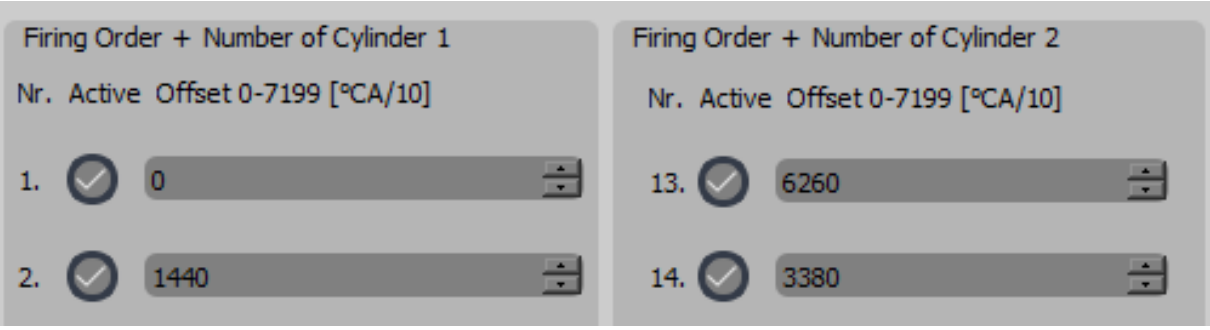


Abbildung 40: Anleitung-Engine-Config-3

Hier können die Zylinder aktiviert/deaktiviert und der Zündwinkel eingestellt werden.

Konfiguration-Speichern-Laden-Übertragen

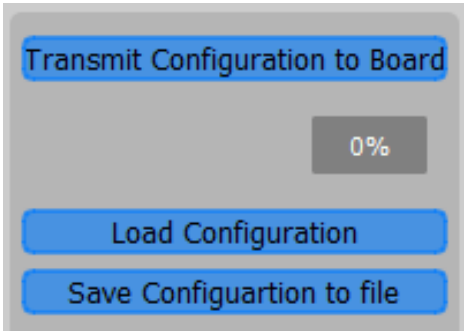


Abbildung 41: Anleitung-Engine-Config-4

Knopf	Funktion
Transmit Configuration to Board	Konfiguration zum FPGA übertragen
Load Configuration	Laden einer Konfiguration aus externer Datei
Save Configuration to file	Speichern der Konfiguration in eine externe Datei

Tabelle 12: Anleitung-Engine-Config

Good/Bad-Signal-Shapes

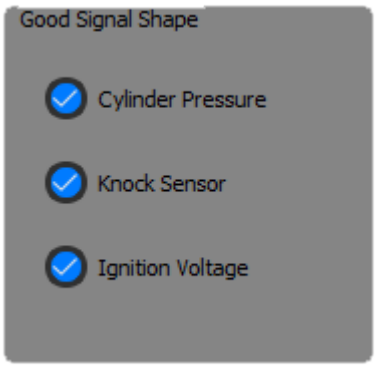


Abbildung 42: Anleitung-Engine-Config-5

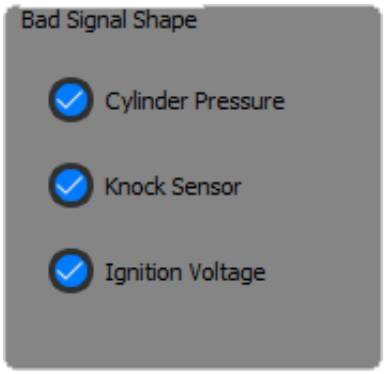


Abbildung 43: Anleitung-Engine-Config-6

Hier kann eingestellt werden, ob die importierten Good- und Bad Shapes für die einzelnen Signale verwendet werden sollen.

Noise-Pattern

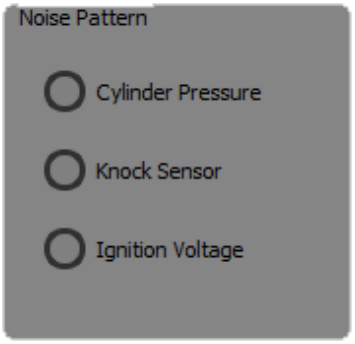


Abbildung 44: Anleitung-Engine-7

Hier kann eingestellt werden, ob die importierten Good- und Bad Shapes für die einzelnen Signale verwendet werden sollen.

5.3.5.4.2 Signal-Shapes

Abbildung 45: Engine-Config-Signal-Shapes-1

Auf der Seite Signal-Shapes, innerhalb von Engine-Config, können die Temperatur und die Random Signale eingestellt werden. Weiterhin können hier, wie auf der Seite *Signal Config*, die Good und Bad Shapes importiert werden.

Temperature

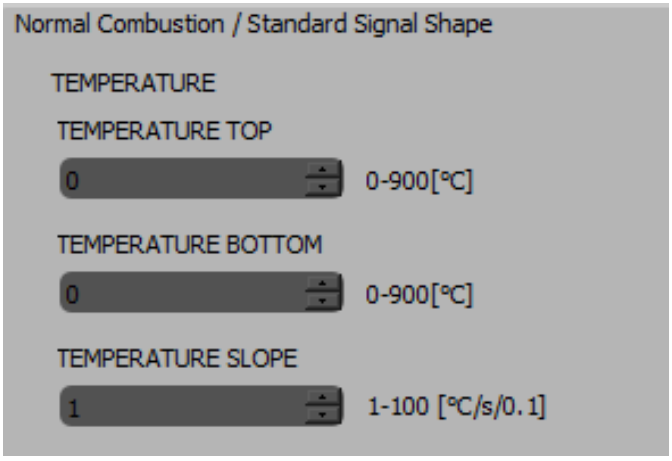


Abbildung 46: Engine-Config-Signal-Shapes-2

Unter Temperatur kann das Maximum, Minimum und die Steigung der Temperatur-Signale eingestellt werden.

Standard Signal Shape Imports

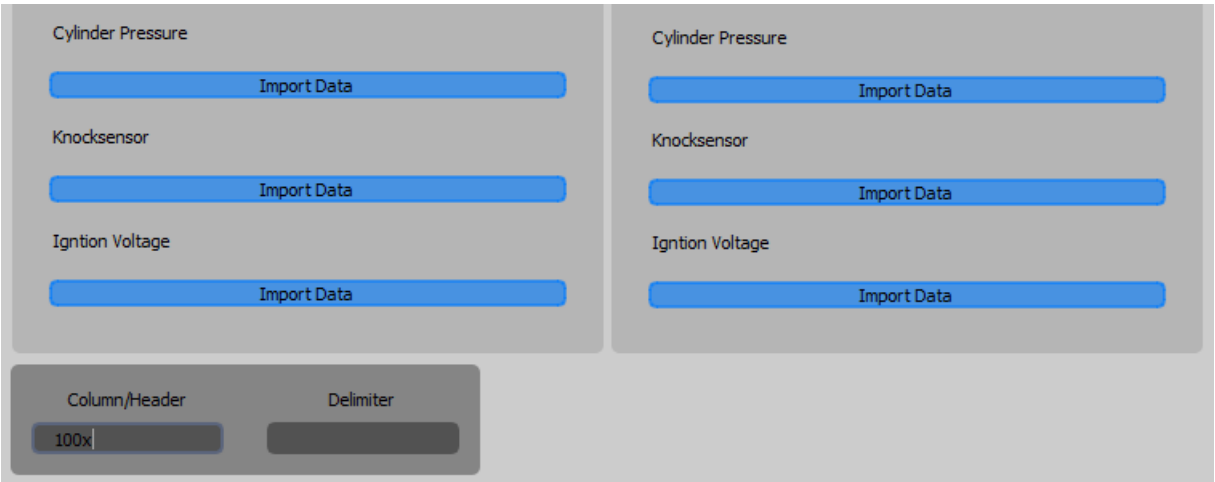


Abbildung 47: Engine-Config-Signal-Shapes-3

Um Good- und Bad Shapes zu importieren, muss ein Header und Delimiter eingegeben werden.
In diesem Fall wurde ein Header passend zur Beispieldatei in Abbildung *Engine-Config-Signal-Shapes-File-Beispiel* eingegeben.
Der Delimiter für das Beispiel ist ein Leerzeichen.

Nötige Struktur des Import File

Die Daten benötigt einen Header, ansonsten kann die Datei beliebig viele Spalten besizten. Falls die Datei mehrere Spalten hat, sollten diese per Tab getrennt werden.

Weiterhin sollte die Datei genau 7200 Datenzeilen beinhalten also mit Header 7201 Zeilen besitzen.
Die Datei kann weniger oder mehr haben. Bei weniger wird der Rest mit Nullen aufgefüllt und bei mehr abgeschnitten (nur beim Engine-Config import).

DegCA	V	100X	200X
0	1,4346	143,46	286,92
0,1	1,3486	134,86	269,72

Abbildung 48: Engine-Config-Signal-Shapes-File-Beispiel

```
Test
50
2
```

Abbildung 49: Engine-Config-Signal-Shapes-File-Beispiel-2

5.3.5.4.3 Sensor-Noise

Abbildung 50: Engine-Config-Noise

Auf dieser Seite können Störsignale, für die unterschiedlichen Zylinder aktiviert und eingestellt werden. Es kann konfiguriert werden, an welchem Zylinder, für wie lange, an welchem Sensor, ein Störsignal auftreten soll. Weiterhin wird bei *Time or CA based* eingestellt, ob die Periodendauer abhängig von der Zeit oder vom Kurbelwellenwinkel ist. Bei *Calculation* kann eingestellt werden, wie der FPGA das Signal berechnet.

5.3.5.5 SignalConfig

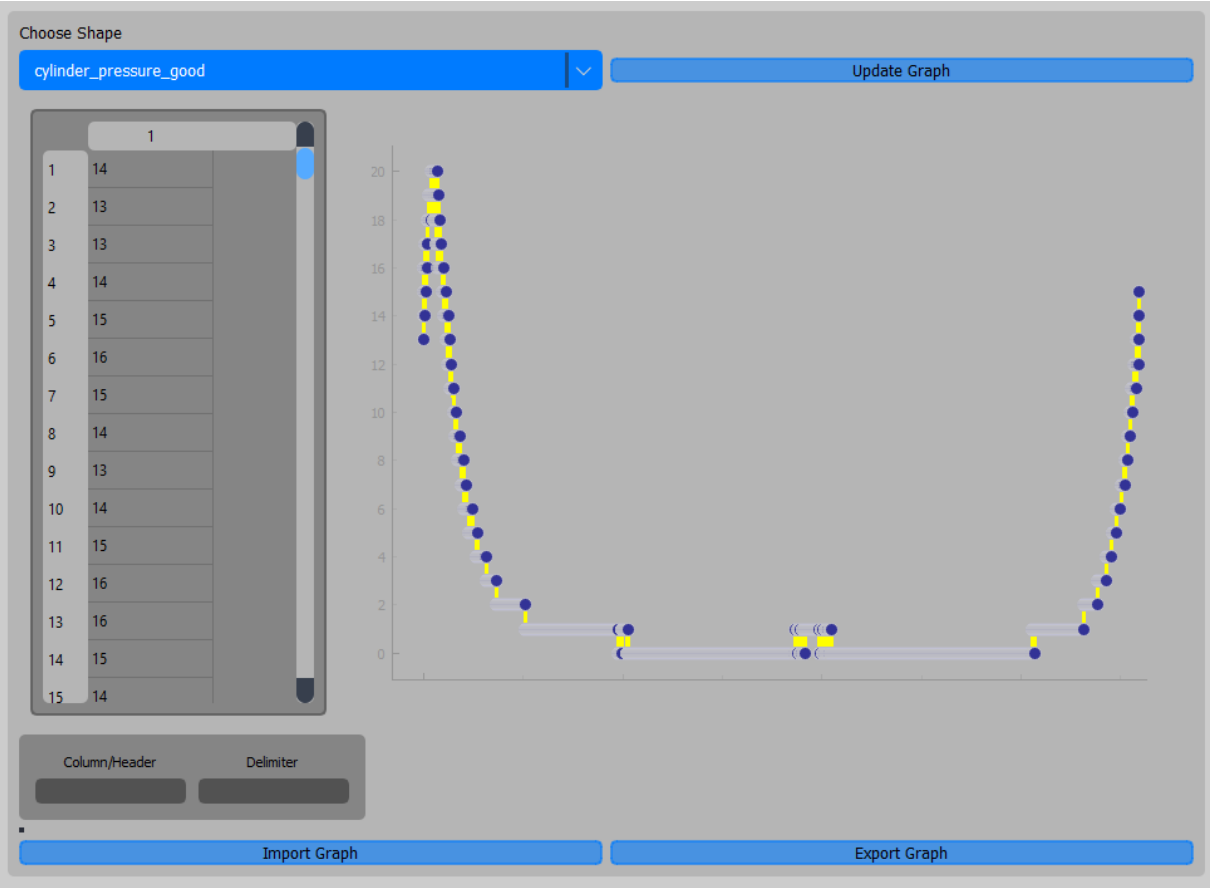


Abbildung 51: Seite-SignalConfig

Auf der Seite SignalConfig befindet sich ein kleiner Signal-Editor. Dort können die Good und Bad-Shapes aus der Konfigurationsdatei oder aus einer Programm externen Datei importiert werden. Diese werden dann im Graphen dargestellt und die einzelnen Datenpunkte können in der Liste bearbeitet werden. Wenn ein Datenpunkt bearbeitet wurde, muss man auf den “Update Graph” Knopf klicken. Ebenfalls kann die Kurve exportiert werden.

5.3.5.6 Debugging

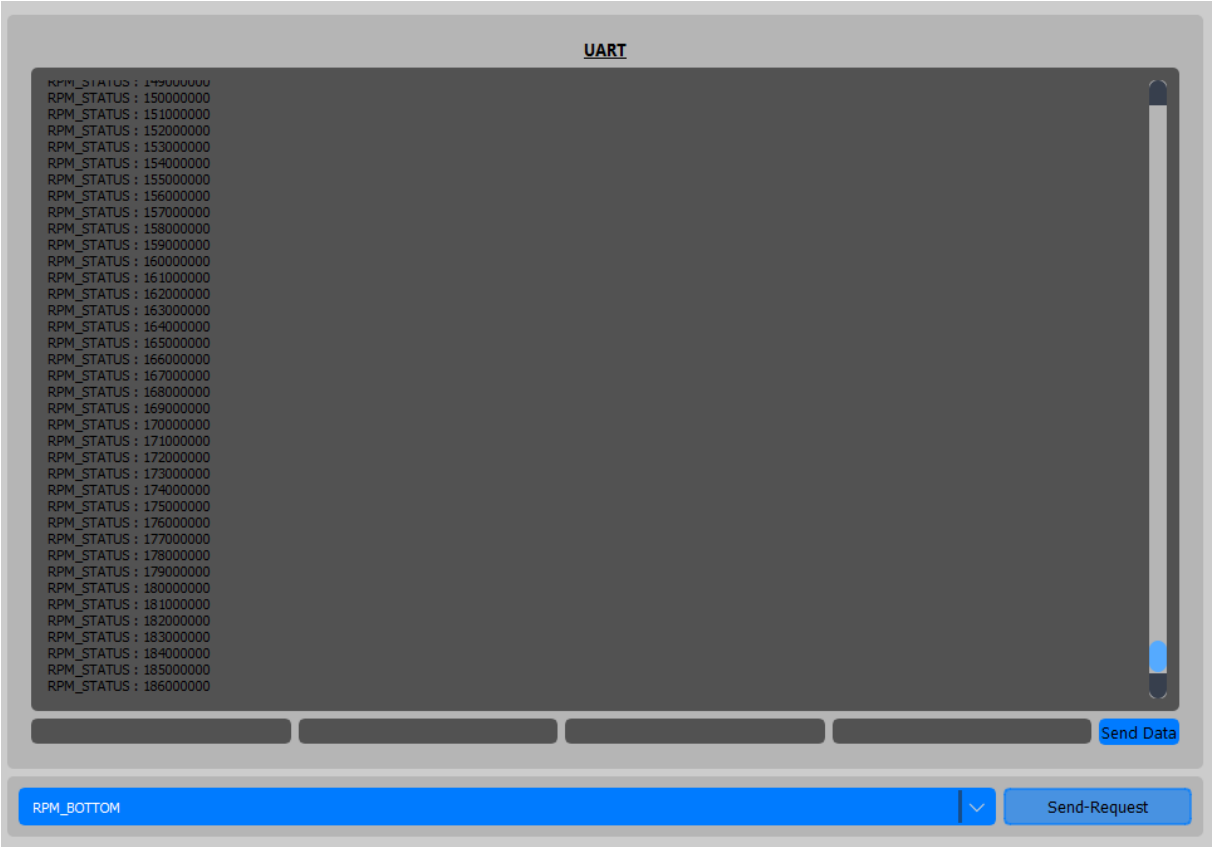


Abbildung 52: Seite-Debugging

Die Debugging Seite steht nur zur Auswahl wenn der Expert-Mode aktiviert ist. Auf dieser Seite sieht man alle empfangene Daten und man kann direkt Befehle an den FPGA schicken. Es können Befehle selbst geschrieben werden, dabei muss auf die richtige Formatierung geachtet werden, oder man wählt einen Befehl aus und schickt diesen. Somit ist diese Seite mit bedacht zu verwenden.

Abbildungsverzeichnis

TABELLENVERZEICHNIS

CODEVERZEICHNIS

LITERATURVERZEICHNIS

[1] **FPGA Theorie** [Online]
https://www.nandland.com/articles/what-is-an-fpga-what-is-an-asic.html

[2] **VHDL** [Online]
https://vhdlguide.readthedocs.io/en/latest/
https://www.ics.uci.edu/~jmoorkan/vhdlref/Synario%20VHDL%20Manual.pdf

[3] **AsciiDoc** [Online]
https://asciidoc.org/
https://asciidoctor.org/docs/what-is-asciidoc/

[4] **SPI-Dokumentation** [Online] 2012
https://www.ti.com/lit/ug/

[5] **UART-Dokumentation** [Online] 2010
https://www.ti.com/lit/

[6] **DAW** [Online]
https://www.ti.com/lit/

[7] **Dev-Insider** |*chrissikraus, Stephan Augsten*|Interperter [Online] 31.08.2018
https://www.dev-insider.de/der-unterschied-von-compiler-und-interpreter-a-742282/

[8] **Python Software Foundation** Packages [Online]
https://docs.python.org/3/reference/import.html#regular-packages

[9] **Python Software Foundation** pip [Online]
https://pypi.org/project/pip/

[10] **pip-PyQt5** *PhilThompson* PyQt5 [Online]
https://pypi.org/project/PyQt5/

[11] **Riverbank Computing Limited, The Qt Company** PyQt5 Dokumentation [Online]
https://www.riverbankcomputing.com/static/Docs/PyQt5/

[12] **The Qt Company** Qt [Online] 20.06.1995
https://www.qt.io/

[13] **The Qt Company** Qt-Creator [Online] 03.2009
https://www.qt.io/product/development-tools

[14] **The Qt Company** Qt-Dokumentation [Online]
https://doc.qt.io/

[15] **Riverbank Computing Limited, The Qt Company** pyuic5 [Online]
https://www.riverbankcomputing.com/static/Docs/PyQt5/designer.html

[16] **Mozilla and individual contributors** XML Dokumentation [Online]
https://developer.mozilla.org/en-US/docs/Web/XML

[17] **Mozilla and individual contributors** CSS Dokumentation [Online]
https://developer.mozilla.org/en-US/docs/Web/CSS

[18] **PyInstaller-readthedocs** Dokumentation [Online]
https://pypi.org/project/pyinstaller/

[19] **Python Software Foundation** Module [Online]
https://docs.python.org/3/tutorial/modules.html

[20] **Python Software Foundation** Threading [Online]
https://docs.python.org/3.6/library/threading.html

[21] **Pyserial-readhthedocs** Pyerial Dokumentation [Online]
https://pyserial.readthedocs.io/en/latest/pyserial_api.html

[22] **TheQtCompany** QApplication Dokumentation [Online]
https://doc.qt.io/qt-5/qapplication.html

[23] **TheQtCompany** QtDeisgner Anleitung[Online]
https://doc.qt.io/qt-5/qtdesigner-manual.html

[24] **GitHub** *Wanderosn Magalhaes* PySide_Base [Online] 9.5.2020
https://github.com/Wanderson-Magalhaes/Simple_PySide_Base

[25] **The Qt Company** Qt-Widgets-Dokumentation [Online]
https://doc.qt.io/archives/qt-4.8/designer-using-custom-widgets.html

[26] **GeeksforGeeks** Python-Parenting [Online] 23 Jan. 2020
https://www.geeksforgeeks.org/python-call-parent-class-method/

[27] **Python Software Foundation** ConfigParser Dokumentation [Online]
https://docs.python.org/3/library/configparser.html

[28] **The Qt Company** Qt-Threads-Dokumentation [Online]
https://doc.qt.io/qt-5/qthread.html

ABKÜRZUNGSVERZEICHNIS

[1] FPGA ..... Field Programmable Gate Array = Fieldprogrammierbares Gate-Array

[2] GUI ..... Graphical User Interface = Grafische Benutzeroberfläche

[3] CAM ..... CAM-Shaft = Knockenwelle

[4] SAFI ..... Sensor Actor Functional Interface

[5] RPM ..... Revolutions per Minute = Umdrehungen pro Minute

[6] CA ..... CAM-Shaft Angle in radiant = Knockenwellen-Winkel in Radiant

[7] DegCA ..... CAM-Shaft Angle in degrees = Knockenwellen-Winkel in Grad

[8] SPI ..... Serial Peripheral Interface

[9] DAW ..... Digital Analog Wandler

[10] PMOD ..... Peripheral Module Interface

[11] DA4 ..... Digital Analog Wandler von Digilent

[12] VHDL ..... Very High Speed Integrated Circuit Hardware Description Language

[13] UI ..... User Inteface

[14] PKI ..... Pressure, Knocksensor, Ignition Voltage = Druck, Klopfsensor, Zündspannung

[15] RAM ..... Random Access Memory

[16] BRAM ..... Block Random Access Memory

[17] RPMS ..... Revolutions p Minute Slope = Änderung der Umdrehungen pro Minute

[18] QSPI ..... Quad Serial Peripheral Interface

[19] DDR3 ..... Double Data Rate 3

[20] UART ..... Universal Asynchronous Receiver-Transmitter

[21] UART_RX ..... UART Receiver

[22] UART_TX ..... UART Transmitter

[23] USB ..... Universal Serial Bus

[24] EEPROM ..... Electrically Erasable Programmable Read Only Memory

[25] SHA-256 ..... Secure Hashing Algorithm with 256-Bit Key = Sicherer Hash-Algorithmus mit 256-Byte Key

[26] VCC ..... Voltage at the common collector = Versorgungsspannung

[27] JTAG ..... Joint Test Action Group

[28] I/O ..... Input / Output

[29] GND ..... Ground

[30] DSP ..... Digital Signal Processor

[31] FIFO ..... First In First Out (Buffer)

[32] COM ..... Communication = Kommunikation

[33] TU ..... Technische Universität

[34] CSS ..... Cascading Stylesheets

[35] XML ..... Extensible Markup Language

DATENDISK VERZEICHNIS

Dokumentation

Diplomarbeit_Plunser_Rieser.pdf — Dokumentation der Diplomarbeit

Sourcecode

Programm.zip — Gesamter Code der graphischen Benutzeroberfläche

Setup.exe

SafiGUISetup.exe — Installer der GUI

Anhang

A.1 Schlussfolgerung/Projekterfahrung

Die Entwicklung der grafischen Benutzeroberfläche gestaltete sich als außerordentlich lehrreich. Durch die Erstverwendung von GitHub, Qt und Pyserial, konnte viel gelernt werden. Dadurch wurden Fähigkeiten im Design und Programmierung erweitert, vor allem im bezug, wie sollte ein Programm a ufgebaut sein, wie werden Threads korrekt implementiert, wie entwirft man eine GUI.
Die Innio GmbH&Co OG Jenbach, ist sehr zufrieden mit dem Ergebnis.

A.2 Projektterminplanung

Datum	Wer	Meilenstein
28.09.2020	Gemeinsam	Überlegung, wie das Python Programm aufgebaut wird
5.11.2020	Rieser	Analyse VHDl
5.11.2020	Plunser	Grunddesign der GUI
20.01.2021	Rieser	Backend der GUI
20.01.2021	Plunser	Frontend der GUI
28.02.2021	Rieser	Dokumentation und Fehlerbeseitigung VHDL
28.02.2021	Plunser	Dokumentation und Fehlerbeseitung GUI/Python Code
24.03.2021	Gemeinsam	Fertigstellung aller Dokumentationen

Tabelle 13: Projektterminplanung

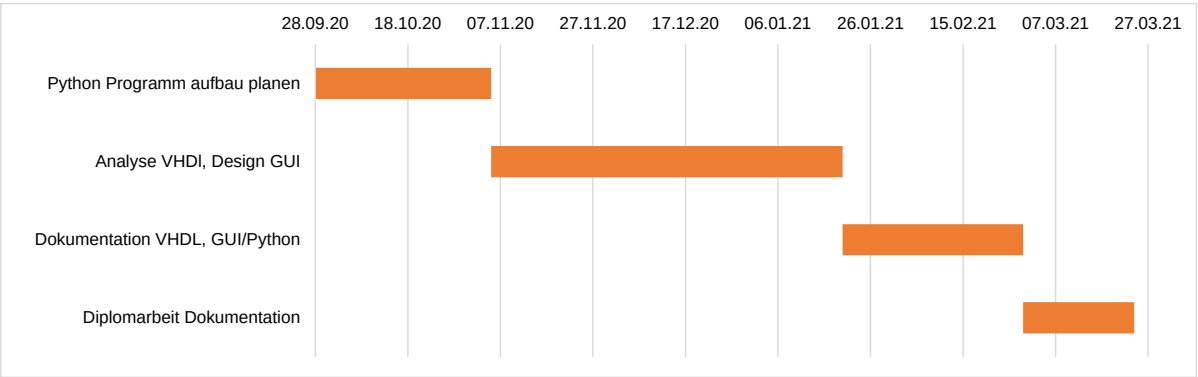


Abbildung 53: Gantt-Diagramm

A.3 Arbeitsnachweis Diplomarbeit

Plunser Fabio			
Datum[T:M:J]	Uhrzeit[h:min]	Stunden[h]	Beschreibung
28.9.2020	18:00-18:30	0.30	Austestung von GitHub
29.9.2020	18:00-19:00	1:00	Grundriss der LaTeXDokumentation
30.9.2020	16:00-18:30	2:30	Python imports ausgebessert, Serial Handler grundriss.
1.10.2020	18:00-19:00	2:00	GUI Update, Python imports, Ordner Sturktur
2.10.2020	18:00-19:00	2:00	Python imports repariert, durch relative imports
3.10.2020	16:00-20:00	4:00	GUI QThreads, UART Start Stop, Serial update, GUI design update
5.10.2020	18:00-18:30	0:30	HTL Logo hinzugefügt, bug fixes, Ordner Struktur erweitert
6.10.2020	19:00-23:00	4:00	GUI komplett neu gestaltet
7.10.2020	20:00-21:30	1:30	GUI erweitert mit Icons
8.10.2020	16:00-22:00	6:00	Python main.py komplett neu, GUI neuer Style
9.10.2020	16:00-22:00	6:00	RPM Graph implementiert, bug fixes
10.10.2020	16:00-20:00	4:00	QThread für Graphen und Serial, FPGA treiber automatisch installieren
11.10.2020	16:30-17:00	0:30	Bug Fixes
12.10.2020	17:00-18:00	1:00	gitignore
15.10.2020	17:00-18:00	1:00	request_constructor repariert, FPGA Port automatisch finden fix, bug fixes
16.10.2020	17:00-18:00	1:00	bug fixes, GUI erweitert und geupdated
17.10.2020	16:00-21:00	5:00	viele bug fixes, Serial Handler verbessert, automatische Listung aller angeschlossenen USB-COM anschlüsse repariert, automatische Verbindung mit richtigen COM-Port repariert.
18.10.2020	16:00-22:00	6:00	fixed UartRead/ListRefresh/ListRefresh/Listallports
20.10.2020	15:00-20:00	5:00	bug fixes, main.ui update
21.10.2020	17:00-23:00	5:00	PyQt in mehrere Dateien aufteilen, QtThread versucht zu reparieren
22.10.2020		5:00	main.ui update, Zylinder Arrays, viele GUI änderungen
23.10.2020	15:00-20:00	5:00	fixed Listung der Ports, bug fixes
25.10.2020	17:00-23:00	5:00	importieren von daten, konfiguration speicherung und laden bei start Grundriss
27.10.2020	16:00-22:00	5:00	Bug Fixes
28.10.2020	16:00-22:00	5:00	Bug Fixes
29.10.2020	15:00-20:00	5:00	versucht Skalierung bei unterschiedlichen Monitorauflösungen zu reparieren
30.10.2020	15:00-20:00	5:00	Speicherung der Konfiguration erweitert, Bug Fixes
31.10.2020	16:00-22:00	5:00	Konfiguration Speicherung bug fixes, Threads fixes, updates, erweiterungen
1.11.2020	17:00-23:00	5:00	GUI Struktur update, Threads Fixes, Updates, Erweiterungen
2.11.2020	16:00-23:00	6:00	Python import fixes, bug fixes
3.11.2020	16:00-22:00	5:00	Bug Fixes
4.11.2020	20:00-23:00	3:00	Funktion Erweiterungen, Bug Fixes
15.11.2020	20:00-23:00	3:00	Funktion Erweiterungen, Bug Fixes
9.12.2020	20:00-23:00	3:00	GUI Update
26.12.2020	16:00-21:30	5:30	Anleitung der GUI entwurf
2.1.2021	16:00-21:00	5:00	GUI changes, Bug Fixes
3.1.2021	18:00-22:00	4:00	Diplomarbeit LaTeXVorlage entwurf
5.1.2021	18:00-22:00	4:00	Diplomarbeit LaTeXVorlage erweitern

9.1.2021	17:00-22:00	5:00	Diplomarbeit LaTeXVorlage erweitern
10.1.2021	17:00-22:00	5:00	Diplomarbeit LaTeXVorlage erweitern
20.1.2021	17:00-19:00	2:00	Wer dokumentiert was
22.1.2021	17:00-19:00	2:00	Python fix
23.1.2021	17:00-19:00	2:00	Diplomarbeit Dokumentation
24.1.2021	17:00-19:00	2:00	Diplomarbeit Dokumentation
8.2.2021	17:00-20:00	3:00	Diplomarbeit Dokumentation
10.2.2021	17:00-20:00	3:00	Diplomarbeit Dokumentation
12.2.201	14:00-16:00	2:00	Diplomarbeit Dokumentation
14.2.2021	14:00-16:00	2:00	Diplomarbeit Dokumentation
16.2.2021	14:00-16:00	2:00	Diplomarbeit LaTeXBug Fixes
18.2.2021	16:00-18:00	2:00	Diplomarbeit Dokumentation
19.2.2021	16:00-18:00	2:00	Diplomarbeit Dokumentation
20.2.2021	16:00-18:00	2:00	Diplomarbeit Dokumentation
22.2.2021	16:00-18:00	2:00	Diplomarbeit Dokumentation
26.2.2021	16:00-18:00	2:00	Diplomarbeit Dokumentation
1.3.2021	16:00-18:00	2:00	Diplomarbeit Dokumentation
10.3.2021	16:00-18:00	2:00	Diplomarbeit Dokumentation
15.3.2021	16:00-18:00	2:00	Diplomarbeit Dokumentation
25.3.2021	16:00-18:00	2:00	Fertigstellung der Diplomarbeit Dokumentation
		Summe: $\sum = 194$	

Tabelle 14: Arbeitsnachweis-Plunser Fabio

Rieser David			
Datum [T:M:J]	Uhrzeit	Stunden [h]	Beschreibung
28.9.2020	18:00-18:30	0:30	Erstellung main.py mit Pseudo Code und Initialisierung des GitHub Projekts
29.9.2020	18:00-20:00	2:00	Einbindung des VHDL-Code in das Github Projekt und erste Top-Down-Analyse
30.9.2020	12:00-20:00	8:00	Erstellung des Blockschaltbild des VHDL-Code auf Papier
1.10.2020	14:00-16:30	2:30	Übertragen des Blockschaltbild ins digitale
2.10.2020	14:00-17:00	3:00	Entwicklung - Schutz des FPGA mit Oktokopplern
3.10.2020	18:00-19:00	1:00	Erst-Analyse der Komponenten für die UART im VHDL-Code
4.10.2020	20:00-23:00	3:00	Vertiefende Analyse in UART_RX
5.10.2020	20:00-23:00	3:00	Vertiefende Analyse in UART_TX
6.10.2020	20:00-23:30	3:30	Vertiefende Analyse in COM_READ_ENCODER-Prozess
7.10.2020	20:00-24:00	4:00	Vertiefende Analyse in COM_WRITE-Prozess
8.10.2020	16:00-18:00	3:00	Abgleich der Analyse des VHDL-Code mit der Excel
9.10.2020	16:00-21:00	5:00	AsciiDoc Dokumentation von UART_RX und UART_TX
10.10.2020	16:00-21:30	5:30	AsciiDoc Dokumentation von COM_WRITE-Prozess und COM_READ_ENCODER-Prozess
11.10.2020	16:00-21:00	5:00	Analyse der VHDL-Signal-Generation
12.10.2020	16:00-21:30	5:30	AsciiDoc Dokumentation der VHDL-Signal-Generation
13.10.2020	16:00-21:30	5:30	Analyse der anderen VHDL-Rauschsignal-Generation
14.10.2020	16:00-21:30	5:00	AsciiDoc Dokumentation VHDL-Rauschsignal-Generation
15.10.2020	14:00-19:00	5:00	Analyse der anderen VHDL-SPI-Kommunikation
16.10.2020	16:00-20:00	4:30	AsciiDoc Dokumentation der VHDL-SPI-Kommunikation
20.10.2020	16:00-21:45	5:45	Erstellung von Python Equivalent zu Escape-Sequences in VHDL
21.10.2020	12:00-13:00	1:30	Recherche um das Program thread-sicher zu machen
22.10.2020	12:00-16:00	4:00	Serial-Handler neu gemacht mit blockierenden Calls und Message-Queues
23.10.2020	14:00-19:00	5:00	Erstellung request_constructor und Dataset
28.10.2020	14:00-17:30	3:30	Debugging von Back-End mit FPGA
29.10.2020	16:00-20:30	4:30	Erstellung von ui_serial_extension.py zur Verbindung von Front-End mit Back-End
30.10.2020	16:00-18:00	2:00	Verbindung der Message-Queues mit dem Front-End
1.11.2020	16:00-19:30	3:30	Debuggen der Verbindung von Front-End mit Back-End
3.11.2020	16:00-19:30	3:30	Erstellung der Test-Skript am Arduino
4.11.2020	15:00-20:30	5:30	Debugging des Back-End mit dem Arduino
15.11.2020	16:00-20:15	4:15	Debugging von QThreads
16.11.2020	16:00-18:45	2:45	Treiber Installation umgeschrieben, da sie die GUI nicht starten lässt
17.11.2020	16:00-18:00	2:00	Erstellung der Unit-Tests in Python
20.11.2020	16:00-17:30	1:30	Recherche über AsciiDoc und Start der VHDL-Dokumentation
25.11.2020	16:00-17:00	1:00	Recherce Logging
5.12.2020	15:00-20:30	5:30	Erstellung Logging
12.12.2020	16:00-17:30	1:30	Recherche über Konfigurations-Datei Handling in Python
28.12.2020	16:00-21:15	5:15	Erstellung von config_manager.py
2.1.2020	16:00-21:00	5:00	Erstellung von import_graph auf Nachfrage von INNIO
3.1.2020	16:00-19:00	3:00	Debugging von import_graph
10.1.2020	16:00-18:00	2:00	Implementierung von export_graph
20.11.2020	16:00-18:00	2:00	Absprache Diplomschrift - Wer macht was?
1.2.2020	16:00-19:00	3:00	Diplomarbeit Dokumentation - Vorwissen VHDL, AsciiDoc, DAW

8.2.2020	16:00-20:00	4:00	Diplomarbeit Dokumentation - VHDL
20.2.2020	16:00-21:00	5:00	Diplomarbeit Dokumentation - Back-End
1.3.2020	16:00-21:30	5:30	Diplomarbeit Dokumentation - ui_config_manager und ui_serial_extension
2.3.2020	16:00-20:00	4:00	Diplomarbeit Dokumentation - Arduino-Tests
10.3.2020	16:00-18:00	2:00	Diplomarbeit Dokumentation - Korrektur Fehler
12.3.2020	13:00-13:30	0:30	Diplomarbeit Dokumentation - Abkuerzungsverzeichnis
13.3.2020	16:00-20:30	4:30	Diplomarbeit Dokumentation - Korrekturlesen
14.3.2020	16:00-19:30	3:30	Diplomarbeit Dokumentation - Korrekturlesen
15.3.2020	16:00-21:00	5:00	Diplomarbeit Dokumentation - Korrketur der LaTeXFormattierung
25.3.2020	16:00-19:00	3:00	Fertigstellung Diplomarbeit Dokumentation
		Summe: $\sum = 204 : 15$	

Tabelle 15: Arbeitsnachweis-Rieser David

A.4 Pflichtenheft

P11 Safi Tester - Erweiterung

HTL Innsbruck - Projektplan

Martin Gyurkó

Version 1.0, 2020-05-20: valami

Table of Contents

INNIO Jenbacher
GmbH & Co OG
A-6200 Jenbach, Österreich

1. Vorgeschichte

Wir haben schon gemeinsam mit der TU-Wien ein Projekt gehabt, bei dem als wichtiges Nebenprodukt der Signalgenerator für Prüfstand P11 entwickelt wurde.

Dieser Signalgenerator hat folgende Eigenschaften:

1. Er bedient bis zu 24 Zylinder eines simulierten Motors mit plausiblen oder gewollt fehlerhaften, reproduzierbaren Verbrennungssignalen.
2. Er erzeugt Geschwindigkeitssignale, und winkel- und zeitsynchron dazu die Verbrennungssignale.
3. Für jeden Zylinder werden folgende Signale erzeugt über DA Wandler:
    a. Druckkurve
    b. Klopfsignal
    c. Hochspannungssignal
    d. Abgastemperatur

Diese DA Wandler werden mittels in VHDL programmiertem Xilinx Artix FPGA bedient über mehrere SPI Busse.

Der FPGA ist auf einem Probepanel untergebracht, woran viele Digilent P-Mod DA Wandler angesteckt sind.

Es existiert der VHDL Sourcecode für das Xilinx FPGA Projekt und die Bediensoftware in Form von Excel Tabelle und USB Treiber.

2. Zielsetzung des neuen Projekts

Hauptziel ist es im Rahmen des Sommers die Bedienung des Testers zu erleichtern sowie in Hardware als auch in Software. Weiters muss die Hardware vor elektromagnetischen Einflüssen geschützt werden.

2.1. Bedienung mit Python Script

1. Bedienung über excel Script kennen lernen
2. Kennenlernen des VHDL Codes und Reverse Engineering
3. Funktionen in ein Dokument genau verfassen, um Protokoll genau zu dokumentieren, und Weiterentwicklungen zu erleichtern
    a. Registerzuweisungen
    b. Nachrichten-Protokoll
    c. Beispiele
4. Planung einer Bedienungssoftware GUI und Bedienungsablauf anhand des excel Dokuments.
5. Implementierung des GUI und Kommunikation in Python 3.8 mit PyQt5 und Py-Serial
    a. Minimale Funktionen zum Testen des Protokolls implementieren
    b. Implementierung erweiterter Funktionen mit Herunterladen von
        i. guten
        ii. schlechten Test-Kurven.

2.2. Hardware und Code Entwicklung

1. Kennenlernen auch mit Hilfe von Reverse Engineering
    a. der Hardware
    b. des VHDL Codes und
    c. des Nachrichtenprotokolls.
2. [Wenn genügend Zeit]: Fehlerursachenanalyse und Fehlerbehebung
    a. Feststellen, warum nicht 24 Zylinder möglich sind.
        i. Es ist zur Zeit leider nicht korrekt möglich die Zylinder 17,18,19,20 korrekt zu benutzen.
    b. Behebung des Fehlers

   i. Optimierung des VHDL codes
   ii. Testen und Tests dokumentieren.
c. Dokumentation des neuen Codes
   i. mit Blockschaltbild!
   ii. Beispiele

3. [Wenn genügend Zeit] Anbindung an Diane in P11
   a. Zum Starten und Stoppen der Speed Simulation
4. [Wenn genügend Zeit] Einbau des Geräts in ein fertig kaufbares Gehäuse.
   a. Gehäuseauswahl
      i. Mit Möglichkeit der internen Erweiterung mit Verstärkern
   b. Keine neue Platinenentwicklung! (Zu umständlich, zur Zeit nicht durchführbar.)
   c. Ziel ist eine gute Abschirmung der Ausgänge zu bauen, um EMV Einflüssen der Spulen und Ventile standzuhalten.
   d. Anschlüsse zum Gehäuse müssen steckbar sein
   e. Das Versorgungs-, Erdungs- und Schirmungskonzept dokumentieren

2.3. Inbetriebnahme am P11 durch Innio

Vorerst mit

1. Klopf-/ Drucksignale
2. Temperatursignal
3. Speed Signalen

A.5 Rechtliche Regelung

Innio GmbH&Co OG Jenbach bleiben Eigentümer des VHDL-Codes und der Vorlagen die uns zur Analyse gegeben wurden.
Wir sind und bleiben die Eigentümer des von uns geschriebenen Programms und der Dokumentation.
Wir gewähren Innio GmbH&Co OG Jenbach uneingeschränkte Nutzungsrechte für das Programm und die Dokumentation einschließlich Änderung, Vermarktung und Verwendung.

Artix-7 FPGAs

		Transceiver Optimization at the Lowest Cost and Highest DSP Bandwidth (1.0V, 0.95V, 0.9V)							
	Part Number	XC7A12T	XC7A15T	XC7A25T	XC7A35T	XC7A50T	XC7A75T	XC7A100T	XC7A200T
Logic Resources	Logic Cells	12,800	16,640	23,360	33,280	52,160	75,520	101,440	215,360
	Slices	2,000	2,600	3,650	5,200	8,150	11,800	15,850	33,650
	CLB Flip-Flops	16,000	20,800	29,200	41,600	65,200	94,400	126,800	269,200
Memory Resources	Maximum Distributed RAM (Kb)	171	200	313	400	600	892	1,188	2,888
	Block RAM/FIFO w/ ECC (36 Kb each)	20	25	45	50	75	105	135	365
	Total Block RAM (Kb)	720	900	1,620	1,800	2,700	3,780	4,860	13,140
Clock Resources	CMTs (1 MMCM + 1 PLL)	3	5	3	5	5	6	6	10
I/O Resources	Maximum Single-Ended I/O	150	250	150	250	250	300	300	500
	Maximum Differential I/O Pairs	72	120	72	120	120	144	144	240
Embedded Hard IP Resources	DSP Slices	40	45	80	90	120	180	240	740
	PCIe® Gen2[1]	1	1	1	1	1	1	1	1
	Analog Mixed Signal (AMS) / XADC	1	1	1	1	1	1	1	1
	Configuration AES / HMAC Blocks	1	1	1	1	1	1	1	1
	GTP Transceivers (6.6 Gb/s Max Rate)[2]	2	4	4	4	4	8	8	16
Speed Grades	Commercial Temp (C)	-1, -2	-1, -2	-1, -2	-1, -2	-1, -2	-1, -2	-1, -2	-1, -2
	Extended Temp (E)	-2L, -3	-2L, -3	-2L, -3	-2L, -3	-2L, -3	-2L, -3	-2L, -3	-2L, -3
	Industrial Temp (I)	-1, -2, -1L	-1, -2, -1L	-1, -2, -1L	-1, -2, -1L	-1, -2, -1L	-1, -2, -1L	-1, -2, -1L	-1, -2, -1L

	Package[3], [4]	Dimensions (mm)	Ball Pitch (mm)	**Available User I/O**: 3.3V SelectIO™ HR I/O (GTP Transceivers)							
	CPG236	10 x 10	0.5		106 (2)		106 (2)	106 (2)			
	CPG238	10 x 10	0.5	112 (2)		112 (2)					
	CSG324	15 x 15	0.8		210 (0)		210 (0)	210 (0)	210 (0)	210 (0)	
	CSG325	15 x 15	0.8	150 (2)	150 (4)	150 (4)	150 (4)	150 (4)			
	FTG256	17 x 17	1.0		170 (0)		170 (0)	170 (0)	170 (0)	170 (0)	
	SBG484	19 x 19	0.8								285 (4)
Footprint Compatible	FGG484[5]	23 x 23	1.0		250 (4)		250 (4)	250 (4)	285 (4)	285 (4)	
	FBG484[5]	23 x 23	1.0								285 (4)
Footprint Compatible	FGG676[6]	27 x 27	1.0						300 (8)	300 (8)	
	FBG676[6]	27 x 27	1.0								400 (8)
	FFG1156	35 x 35	1.0								500 (16)

Notes:
1. Supports PCI Express Base 2.1 specification at Gen1 and Gen2 data rates.
2. Represents the maximum number of transceivers available. Note that the majority of devices are available without transceivers. See the Package section of this table for details.
3. Leaded package option available for all packages. See DS180, *7 Series FPGAs Overview* for package details.
4. Device migration is available within the Artix-7 family for like packages but is not supported between other 7 series families.
5. Devices in FGG484 and FBG484 are footprint compatible.
6. Devices in FGG676 and FBG676 are footprint compatible.

Pmode DA4 Datenblatt

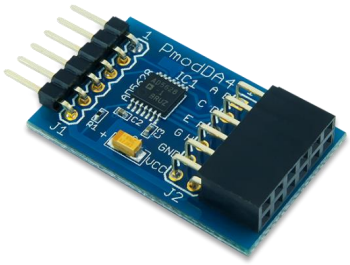
1300 Henley Court
Pullman, WA 99163
509.334.6306
www.digilentinc.com

Pmod DA4™ Reference Manual

Revised November 28, 2016
This manual applies to the Pmod DA4 rev. B

Overview

The Digilent Pmod DA4 is an octal, 12-bit digital-to-analog converter module.



The Pmod DA4.

Features include:

- Eight channel, 12-bit DAC
- Capable of eight simultaneous outputs
- High-speed DSP compatible
- Power-down function capability
- Low power consumption
- Small PCB size for flexible designs 1.2“ × 0.8” (3.0 cm × 2.0 cm)
- 6-pin Pmod connector with SPI interface
- Follows Digilent Interface Specification Type 2
- Example code available in resource center

1 Functional Description

The Pmod DA4 utilizes Analog Devices AD5628-1 to provide a signal DAC capable of 8 simultaneous output channels. With its internal reference voltage of 1.25V and a gain of 2, a resolution of approximately 1 mV can be achieved.

2 Interfacing with the Pmod

The Pmod DA4 communicates with the host board via the SPI protocol. By driving the Chip Select (CS) line to a logic level low voltage, users may send a series of 32 bits of information with the data clocked into the appropriate register on the falling edge of the Serial Clock (SCLK). Once the 32nd bit of information has been clocked in, the command that was sent in the data stream is executed.

An example data stream of how the 32 bits are to be sent to the module is provided below from the AD5628 datasheet.

DB31 (MSB)				Command Bits				Address Bits			
X	X	X	X	C3	C2	C1	C0	A3	A2	A1	A0
Data Bits											
D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
DB0 (LSB)											
X	X	X	X	X	X	X	X				

Table 1. Example data stream from the AD5628 datasheet.

Users should note that the Pmod DA4 by default attempts to use an external reference voltage. However, as there is no external reference voltage provided on the Pmod, users must write to the register to program the chip to use its internal reference voltage of 1.25V. The data stream required to change this is provided below:

Internal REF Register (DB0)	Action
0	Reference off (default)
1	Reference on

Table 2. Internal reference register from the AD5628 datasheet (table 10).

DB31 to DB28	DB27	DB26	DB25	DB24	
X	1	0	0	0	
Don't Cares	Command bits (C3 to C0)				
DB23	**DB22**	**DB21**	**DB20**	**DB19 to DB1**	**DB0**
X	X	X	X	X	1/0
Address bits (A3 to A0) - don't cares				Don't cares	Internal REF register

Table 3. 32-bit input shift register contents for reference set-up command from the AD5628 datasheet (table 11).

The various commands and addresses available for the on-board AD5628 can be found on page 22 of its datasheet.

2.1 Pinout Description Table

Pin	Signal	Description
1	~CS	Chip Select
2	MOSI	Master-Out-Slave-In
3	(NC)	Not Connected
4	SCLK	Serial Clock
5	GND	Power Supply Ground
6	VCC	Power Supply (3.3V/5V)

Any external power applied to the Pmod DA4 must be within 2.7V and 5.5V; however, it is recommended that Pmod is operated at 3.3V.

3 Physical Dimensions

The pins on the pin header are spaced 100 mil apart. The PCB is 1.2 inches long on the sides parallel to the pins on the pin header and 0.8 inches long on the sides perpendicular to the pin header.