## Einleitung

## Software

## Abstract

## Zusammenfassung

Es soll eine multiplattformfähige Software (Linux, macOS, Windows) entwickelt werden, die die Daten des Prüfstands für den Benutzer grafisch ausgibt. Die Bedienung sollte der des alten Programms ähneln, ein Diagramm soll wie gehabt nach der abgeschlossenen Messung angezeigt werden, das ausgedruckt und abgespeichert werden kann.

Die Software für den µC soll alle Sensordaten einlesen und über ein eigenes Protokoll an die GUI-Applikation übermitteln.

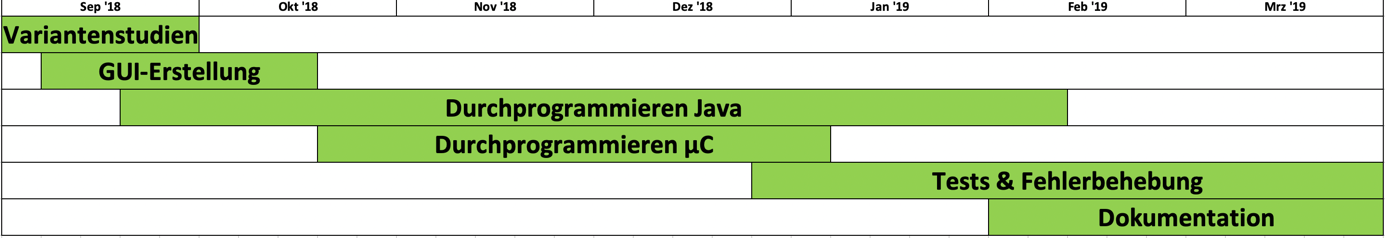
## Aufgabenstellung

Ziel ist es, eine Software für den Benutzer zur Bedienung des Prüfstandes zu schreiben. Diese soll die Messung ausführen und automatisiert steuern, d.h. der Benutzer soll ab dem Zeitpunkt der Messung nur mehr das Motorrad bedienen müssen. Nach abgeschlossener Messung sollen die Messdaten verarbeitet und in einem Diagramm ausgegeben werden.

Der µC soll die Messwerte der Sensoren auswerten und auf Anfrage der GUI-Applikation an diese übertragen. Das System muss echtzeitfähig sein und das Intervall zwischen den Messpunkten soll sich auf ein Minimum beschränken.

## Zeitplan

Um die Erfüllung der Diplomarbeit zu gewährleisten, ist es notwendig sich im Vorfeld einen Terminplan zu erstellen und diesem Folge zu leisten.



## Variantenstudien

## GUI-Applikation

## Plattform

Folgende Varianten habe ich in Betracht gezogen:

* App für Smartphone & Tablet
* Stationärer Touch-Screen mit angeschlossenem Raspberry Pi
* Applikation für PC

## Smartphone & Tablet

Mobile Geräte werden immer beliebter und leistungsstärker, eine Variante besteht im Entwickeln einer App für Android und iOS.

*Vorteile:*

* Mobil
* Schnelle Inbetriebnahme

*Nachteile:*

* Es müssten zwei Apps geschrieben werden (Android und iOS)
* iOS AppStore-Veröffentlichung kostenpflichtig

## Raspberry Pi

Eine weitere Möglichkeit besteht in der Montage eines stationären Touchscreens, der über einen Raspberry Pi das Programm ausführt.

*Vorteile:*

* Einfache Handhabung für den User
* Schnelle Inbetriebnahme

*Nachteile:*

* Begrenzte Leistung
* Zusätzliche Kosten des Systems

## PC

*Vorteile:*

* Hohe Performance
* Großer Monitor

*Nachteile:*

* Eventuell langsame Inbetriebnahme
* Nicht handlich

## Programmiersprache

Im Falle einer App für Smartphone und Tablet standen Java (für Android), Objective-C und Swift (für iOS) zur Auswahl. Für iOS wäre Swift gewählt worden, allerdings hätte dies eine hohe Einarbeitungszeit erfordert, da diese Sprache in der Schule nicht gelehrt wurde. Zusätzlich besteht das Problem, dass Swift noch recht jung ist und es daher noch relativ schwierig ist, für spezifische Probleme im Internet Lösungen zu finden.

Für eine PC-Applikation standen C, C++ und Java zur Verfügung. C wäre nicht geeignet, da diese Sprache nicht objektorientiert ist. Java hat den Vorteil, dass das Programm multiplattformfähig ist, sprich auf den drei wichtigsten Betriebssystemen – Linux, macOS und Windows – ausführbar sein wird. Zusätzlich gestaltet sich das Designen der GUI mit Hilfe des Editors für Java Swing in der Netbeans IDE als relativ einfach.

## µC-Software

## Entwicklungsboard

## Anforderungen an den µC

|  |  |
| --- | --- |
| Sensor | Benötigte Peripherie |
| Motordrehzahl | Digitaler Eingang (ext. Interrupt) |
| Walzendrehzahl | Digitaler Eingang (ext. Interrupt) |
| Motortemperatur (Thermoelement) | Analoger Eingang oder I2C |
| Abgastemperatur (Thermoelement) | Analoger Eingang oder I2C |
| Umgebungswerte (BMP-180) | I2C oder SPI |

Tabelle : Anforderungen an den µC

Zusätzlich wird noch mindestens ein UART zur Kommunikation mit dem PC benötigt. Ein optionaler weiterer UART-Port würde als Debugging-Schnittstelle verwendet werden.

## Arduino MEGA & Arduino DUE

*Vorteile:*

* Hohe Rechenleistung
* Großer Speicher
* 4x UART
* 2x I2C
* CAN-Bus (nur DUE)

*Nachteile:*

* Großer Platzbedarf
* Teuer

Abbildung 1: Arduino MEGA & Arduino DUE, Quelle: Eigene Fotografie

* 3,3V Pegel durch ARM-Architektur (nur DUE)

## Arduino UNO & Arduino NANO

*Vorteile:*

* Klein
* Günstig
* µC tauschbar (nur UNO, THT-Edition)

*Nachteile:*

* Nur ein UART
* Kleinerer Speicher

Abbildung 2: Arduino UNO & Arduino NANO, Quelle: Eigene Fotografie

* Geringere Performance

## Kommunikationsmethoden

## Ethernet

Mit den sogenannten „Ethernet-Shields“ oder den speziellen Arduino Ethernet Boards ist es möglich mit einem Arduino Daten über das Netzwerk zu übertragen. Dadurch spart man sich das Verbinden mit einem USB-Kabel, es setzt aber eine Ethernet-Leitung zur Elektronik und eine Verbindung des Computers ins Netzwerk voraus.

Abbildung 3: Arduino Ethernet, Quelle: Eigene Fotografie

## UART

Deutlich einfacher lässt sich die Kommunikation mit UART lösen. Für Java gibt es Bibliotheken, mit denen das Realisieren der Kommunikation einfach vonstatten geht. Auch ist man bei einer UART-Verbindung nicht an die Netzwerkgeschwindigkeit gekoppelt, bzw. im Falle von WiFi an die Signalstärke.

## Programmiersprachen

Die verbreitetsten Programmiersprachen zur µC Programmierung sind C und Assembler.

Eine weitere Option besteht mit der „Arduino-Language“, die sich als C++ mit der eingebundenen *Arduino.h*-Library versteht und das Programmieren von µCs deutlich vereinfachen soll.

## Zusammenfassung

Als Plattform für die PC-Applikation wurde der PC gewählt, da er die höchste Leistung bietet und im Gegensatz zur mobilen Lösung nur eine Anwendung geschrieben werden muss.

Als Sprache dient Java um die Multiplattformfähigkeit zu gewährleisten. Ein weiterer Vorteil: Sollten Einplatinencomputer wie der Raspberry Pi zukünftig in der Lage sein die erforderliche Leistung zu bringen, wird es möglich sein, das Java-Programm auf diesen auszuführen.

Die Wahl des Entwicklungsboard ist auf den Arduino UNO Rev.3 gefallen, da die Spezifikationen des darin verbauten ATmega328p für unsere Zwecke ausreicht. Der NANO, der denselben µC verbaut hat, wurde nicht verwendet, da der Formfaktor des UNO besser zu handeln ist.

Als Kommunikationsmethode wurde UART gewählt, da die einwandfreie Kommunikation über das Netzwerk von der Geschwindigkeit, bzw. im Falle von W-LAN von der Signalstärke abhängt. Unser Programm setzt aber eine sehr schnelle Kommunikation voraus. Außerdem würde ein solches System keine weiteren Vorteile bringen, da sich im Normalfall Prüfer und Prüfstand im selben Raum befinden.

Programmiert wird der µC mit C++ und der Arduino.h-Library, da die Programmierung der GUI-Applikation sehr zeitintensiv werden wird und an dieser Stelle der Aufwand verringert werden kann. Dies bietet auch den Vorteil, dass Bibliotheken von Sensor-Herstellern (zB von Adafruit für den BMP-180) eingebunden werden können, die die Programmierung noch weiter vereinfachen.

Andere Möglichkeiten, wie zB einen Raspberry Pi mit FreeRTOS (Free Realtime Operating System) als echtzeitfähiges System zu betreiben, wurden außer Acht gelassen, da sie viel Einarbeitungszeit bedeutet hätten.

## PC-Applikation

## UML-Diagramm (vereinfacht)

Zur Übersichtlichkeit wurden JFrame-Objekte (JButton, JMenu, etc.) entfernt, wie auch die von der abstrakten Klasse *Request* abgeleiteten Klassen. In der Klasse data.*Config* wurden die Getter und Setter entfernt.

## GUI-Klassen

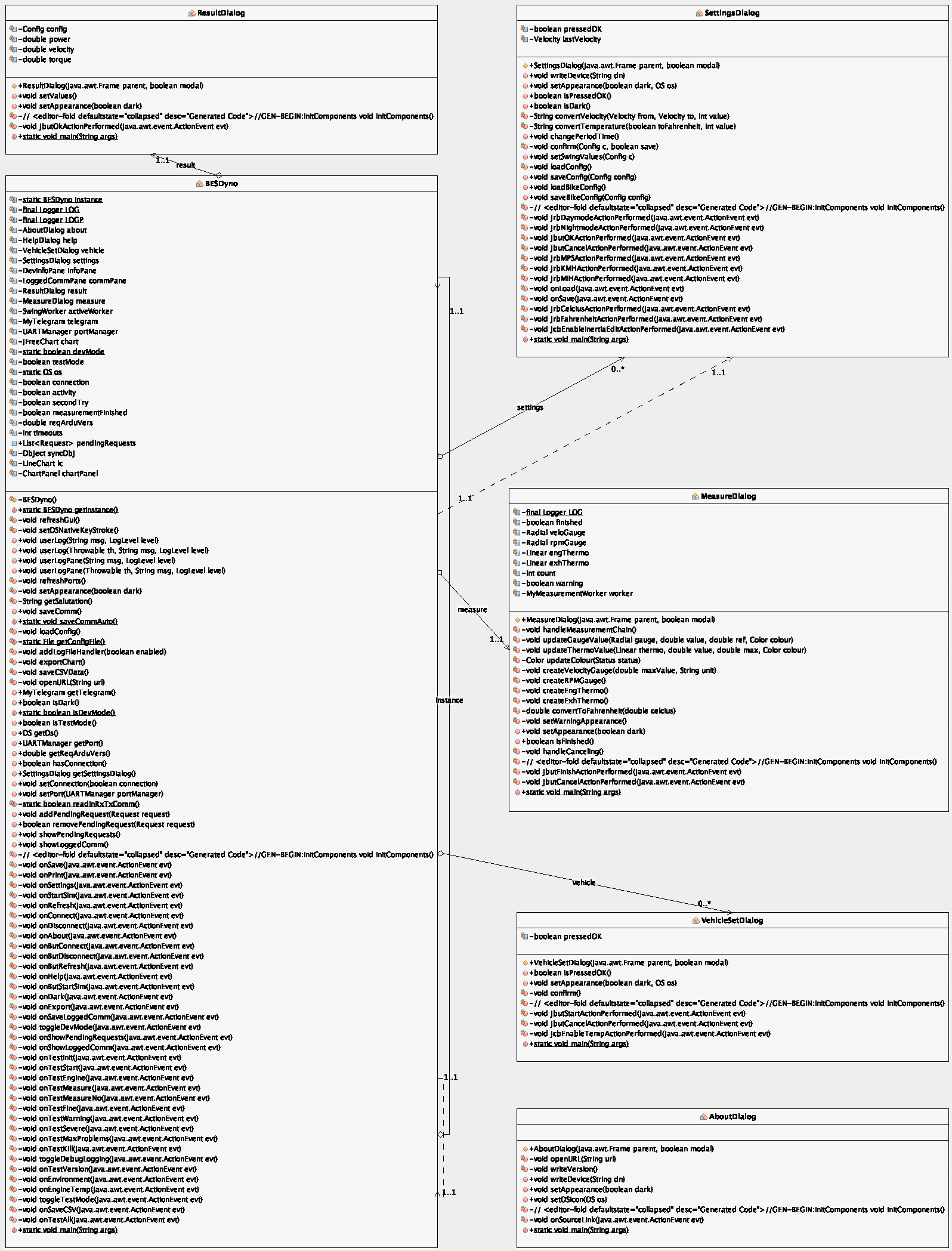


Abbildung 4: UML-Diagramm der GUI-Klassen, Quelle: Generiert mit easyUML

## Datenerhaltungsklassen

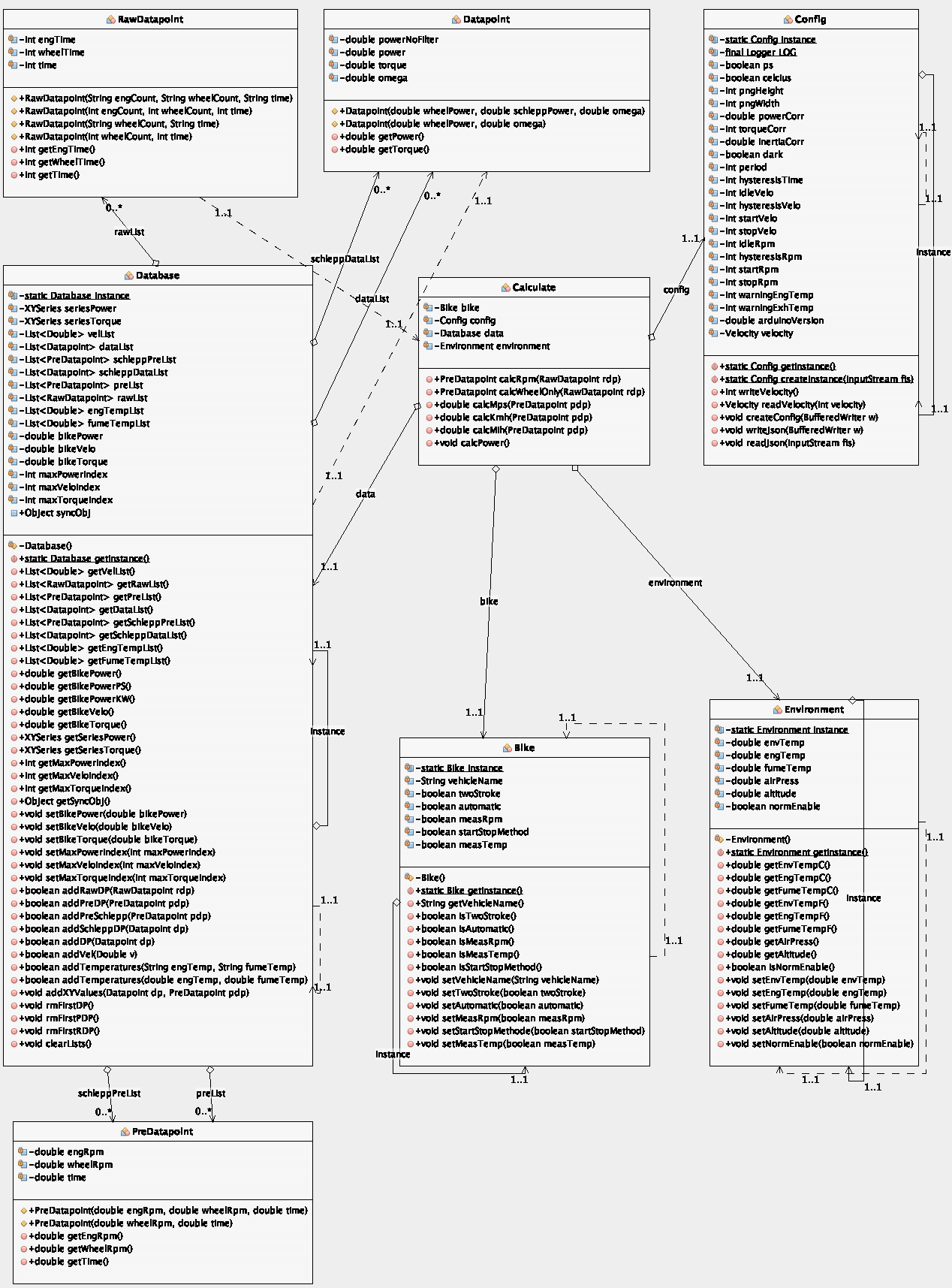


Abbildung 5: UML-Diagramm der Datenerhaltungsklassen, Quelle: Generiert mit easyUML

## Klassen zur Kommunikation und Berechnung

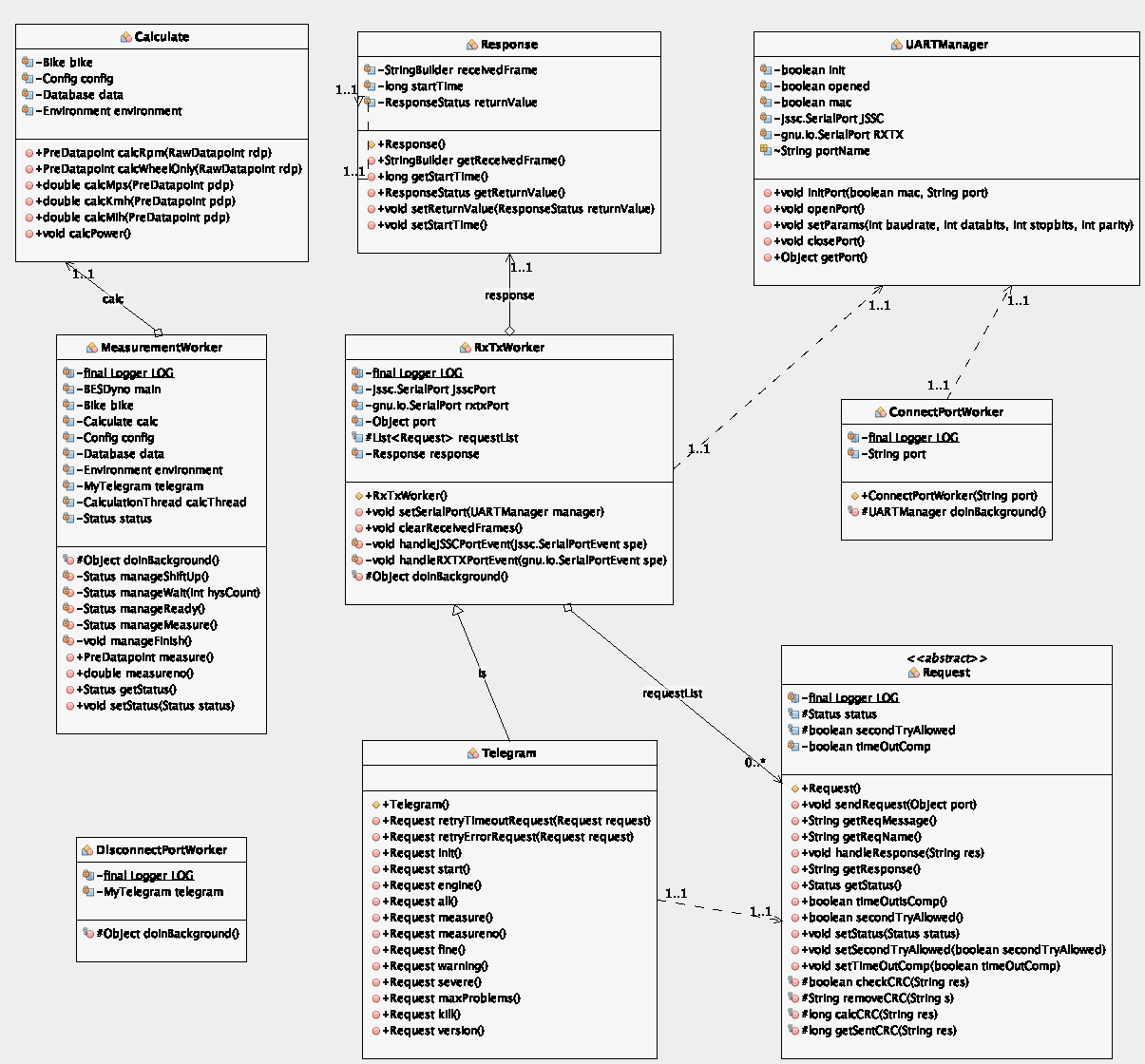


Abbildung 6: UML-Diagramm der Klassen für Kommunikation und Berechnung, Quelle: Generiert mit easyUML

## Graphical User Interface

## About



Abbildung 7: AboutDialog, Quelle: Screenshot

Jedes Programm sollte einen About-Dialog besitzen, in dem allgemeine Informationen über das Programm, rechtliches und Copyright-Informationen Die jLabels wurden mit HTML-Tags[[1]](#footnote-1) ausgestattet, die eine Formatierung der Labels erlauben. Am Anfang wird mit Tag <html> geöffnet und am Ende mit </html> geschlossen. Mit <b> lässt sich eine Textstelle fett darstellen und mit <br> fügt man einen Zeilenumbruch ein.

Der jButton „Quellcode“ öffnet im Standardbrowser einen Link zum GitHub-Repository des Projekts.

## Hauptfenster

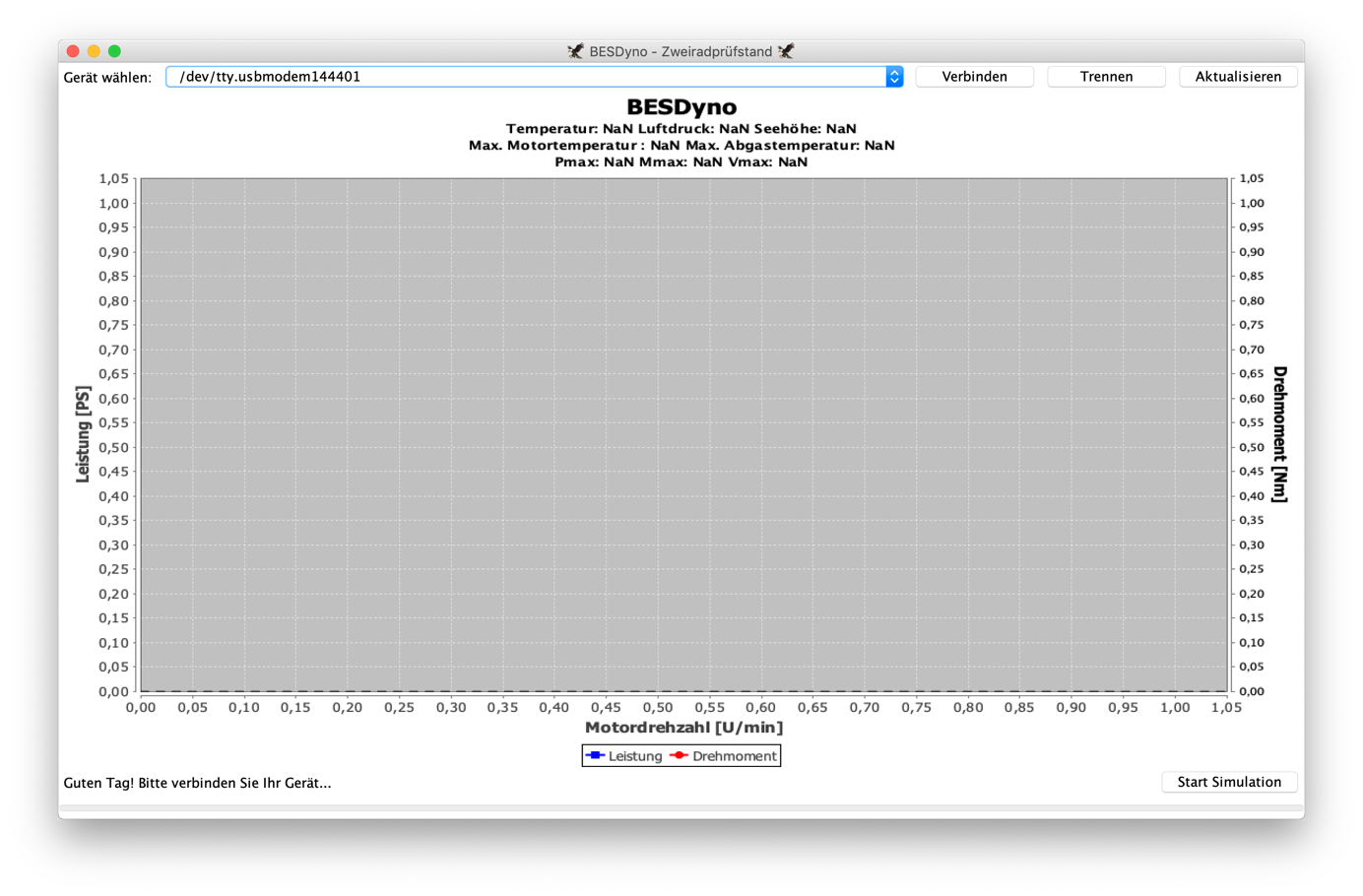


Abbildung 8: Main-GUI, Quelle: Screenshot

Wenn das Programm startet, wird dieses Fenster als erstes angezeigt.

In der oberen Leiste kann man sich mit seinem Endgerät verbinden und unten wird ein Status-Text in einem jTextField angezeigt. Die eigentliche Bedienung findet über das Menü in der Titelleiste statt, das den Vorteil bietet, dass mit Tastaturkürzeln gearbeitet werden kann.

In der Mitte wird beim Start ein leeres Diagramm angezeigt, dass nach Abschluss der Messung mit Daten gefüllt wird.

## Einstellungen

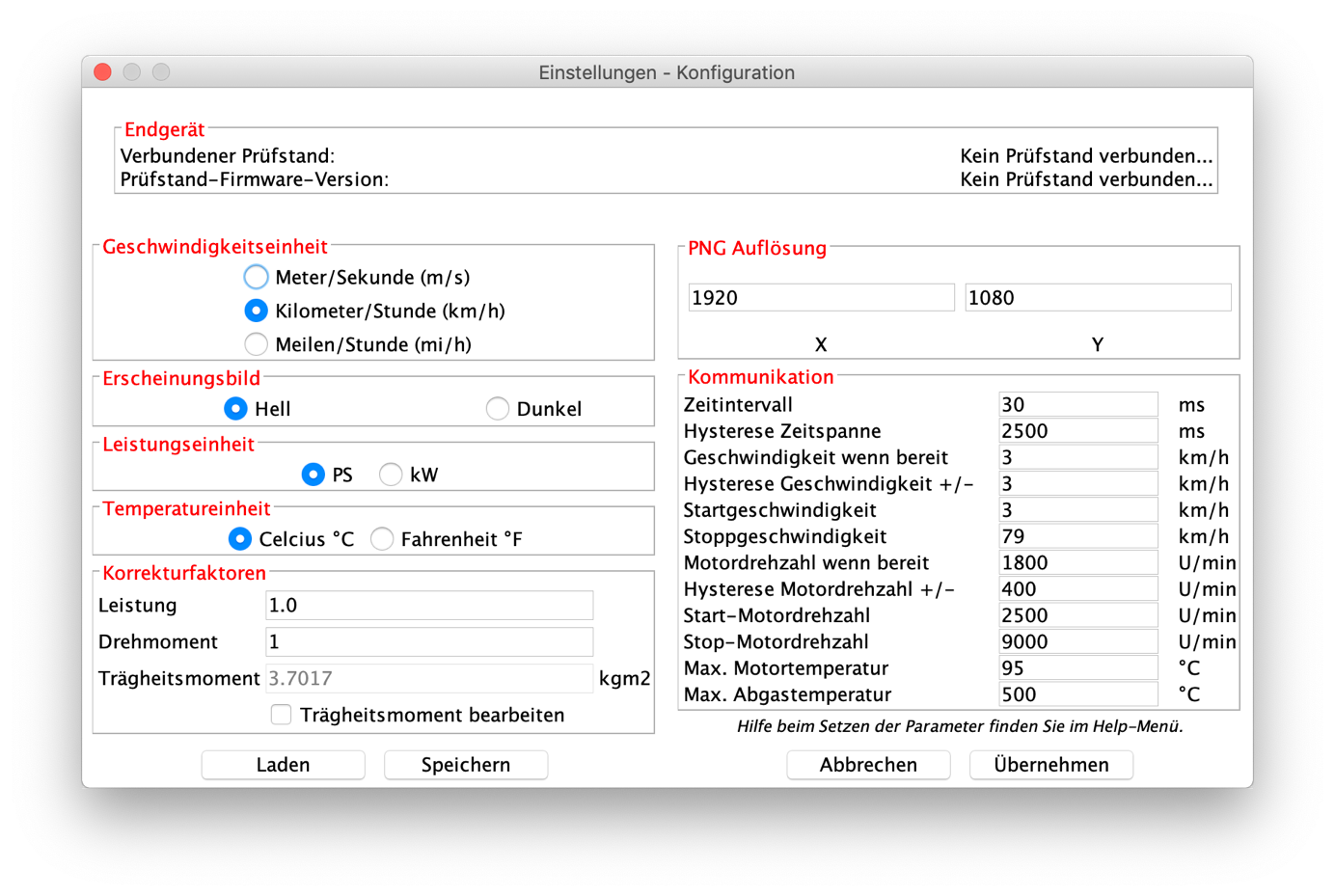


Abbildung 9: SettingsDialog, Quelle: Screenshot

*Im Einstellungsfenster werden für die Messung wichtige Parameter festgelegt:*

|  |  |
| --- | --- |
| Parameter | Funktion |
| Zeitintervall | Zeit, die zwischen zwei Messpunkten vergehen soll. Bei leistungsstarken Rechnern kann sie sehr klein sein. |
| Hysterese Zeitspanne | Gibt an, wie lange das Einpendeln bei einer bestimmten Drehzahl dauern soll. |
| Geschwindigkeit, wenn bereit | Bei dieser Geschwindigkeit pendelt sich das Programm ein. (Leerlaufgeschwindigkeit) |
| Hysterese Geschwindigkeit | Einpendeln: Leerlaufgeschwindigkeit ± Hysterese Geschwindigkeit |
| Startgeschwindigkeit | Ab dieser Geschwindigkeit werden Daten für das Diagramm erfasst. |
| Stoppgeschwindigkeit | Ab dieser Geschwindigkeit stoppt die Messung. |
| Motordrehzahl, wenn bereit | Bei dieser Drehzahl pendelt sich das Programm ein. (Leerlaufdrehzahl) |
| Hysterese Motordrehzahl | Einpendeln: Leerlaufdrehzahl ± Hysterese Motordrehzahl |
| Start-Motordrehzahl | Ab dieser Drehzahl werden Daten für das Diagramm erfasst. |
| Stop-Motordrehzahl | Ab dieser Drehzahl stoppt die Messung. |
| Max. Motortemperatur | Überschreitet während der Messung der Motor diesen Wert, gibt es eine Warnung. |
| Max. Abgastemperatur | Überschreitet während der Messung der Krümmer diesen Wert, gibt es eine Warnung. |

Tabelle : Einstellungs-Parameter

*Anmerkung: Wenn die Motordrehzahl erfasst wird, bezieht sich das Programm nur auf die eingegeben Drehzahlen.*

## Mess-Fenster

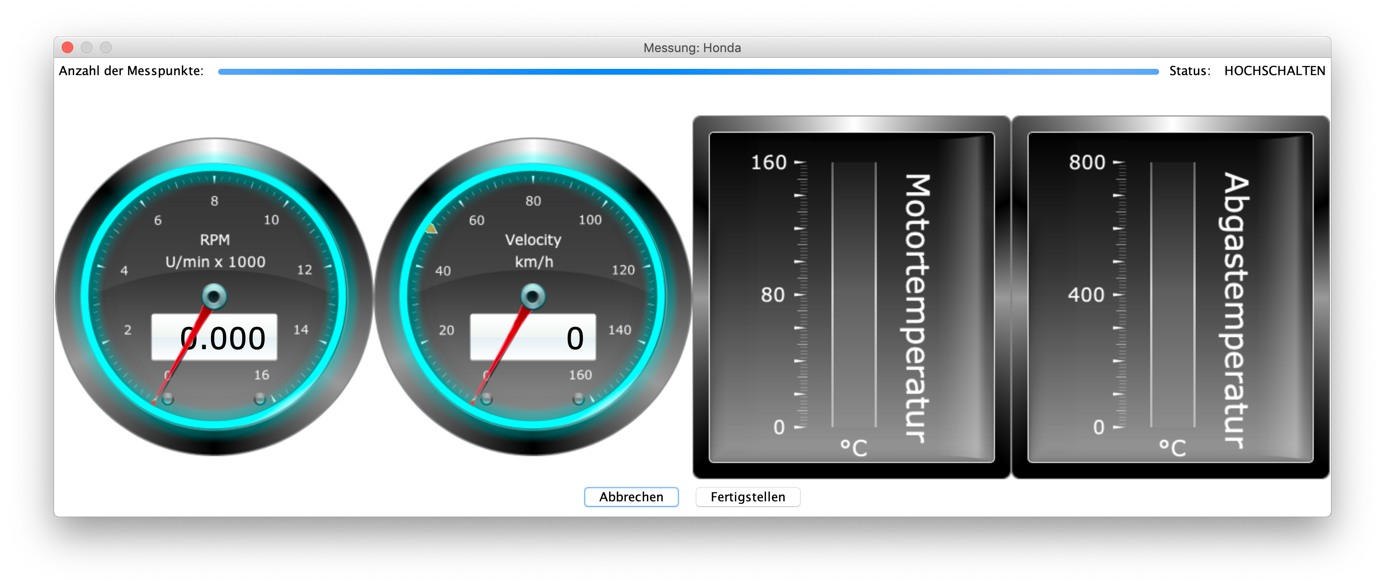


Abbildung 10: MeasureDialog mit Temperaturmessung, Quelle: Screenshot



Abbildung 11: MeasureDialog ohne Temperaturmessung, Quelle: Screenshot

Bevor man zu diesem Fenster gelangt, muss noch in einem separaten Dialogfenster der Name des Motorrads, die Taktanzahl des Motors und ob man Motordrehzahl und die Temperaturen messen möchte, eingegeben werden.

Danach gelangt man zum MeasureDialog der die aktuelle Drehzahl des Motors und die aktuelle Geschwindigkeit des Walze anzeigt. Die Aufgabe des Nutzers wird textuell über den Tachos und farblich (Blau – im Bild zu sehen, Gelb, Grün, Rot) ausgegeben.

Wenn die Temperatur währen des gesamten Messvorgangs gemessen wird, überprüft das Programm, ob sie die im Einstellungsdialogfenster eingegebenen Werte übersteigt. Ist dies der Fall, färbt sich als Warnung die gesamte Benutzeroberfläche rot.

Nach abgeschlossener Messung wird ein Dialogfenster mit den Maximalwerten des Motorrads angezeigt. Wenn dieses geschlossen wird, gelangt man wieder zurück zum Hauptfenster und das Diagramm wird mit den gemessenen Werten gefüllt.

Die Tachometer/Thermometer passen sich je nach Einheit (m/s, mi/h, km/h bzw. °C, °F) in ihrer Skalierung an.

## Menü in der Titelleiste

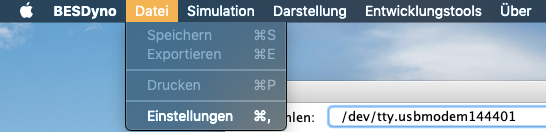


Abbildung 12: Datei-Menü, Quelle: Screenshot

Über den Menü-Punkt „Datei“ können die Werte als CSV-Datei gespeichert, das Diagramm exportiert oder gedruckt, sowie das Einstellungsdialogfenster aufgerufen werden.



Abbildung 13: Simulation-Menü, Quelle: Screenshot

Über den Menü-Punkt „Simulation“ kann die Simulation gestartet und die Umweltdaten bzw. Motorradtemperaturen aktualisiert werden. Unter dem jSeperator wird die Verbindung mit dem Endgerät gesteuert.



Abbildung 14: Darstellung-Menü, Quelle: Screenshot

Im Menü „Darstellung“ gibt es nur den Punkt „Dark-Mode“ mit dem die gesamte Oberfläche auf ein dunkles Theme geschaltet wird. Die Hintergründe werden Dunkelgrau, der Text wird weiß.

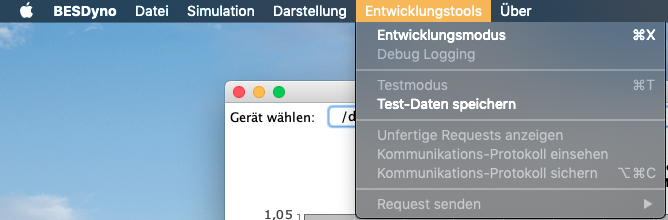


Abbildung 15: Entwicklungstools-Menü, Quelle: Screenshot

Im Menü-Punkt „Entwicklungstools“ finden sich Debugging-Tools, die sich während der Entwicklung des Programms als sehr nützlich erwiesen haben.

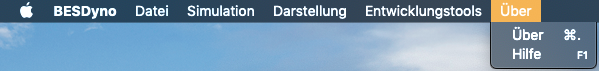


Abbildung 16: Über-Menü, Quelle: Screenshot

Im Menü-Punkt „Über“ findet sich das ‚Über das Programm‘-Dialogfenster, sowie eine Hilfe, die als Online-Handbuch realisiert wird.

## SteelSeries (Tachometer & Thermometer)

Für die Anzeige der Tachometer und Thermometer im MeasureDialog habe ich nach einer Bibliothek gesucht, die einfach zu bedienen und optisch zufriedenstellen ist.

Fündig geworden bin ich auf dem Blog[[2]](#footnote-2) von Gerrit Grunwald: <https://harmoniccode.blogspot.com>.

Er hat die SteelSeries-Library für JavaFX, Java Swing und JavaScript ins Leben gerufen.

GitHub-Repository: <https://github.com/HanSolo/SteelSeries-Swing>

Da die Dokumentation zu dieser Bibliothek auf dem Blog schwer zu finden war (es handelt sich nur um Blog-Einträge zu verschiedensten Zeitpunkten), habe ich eine Website gesucht, die diese Bibliothek verwendet und den JavaScript-Quelltext eingesehen, um zu verstehen, wie diese Bibliothek funktioniert:

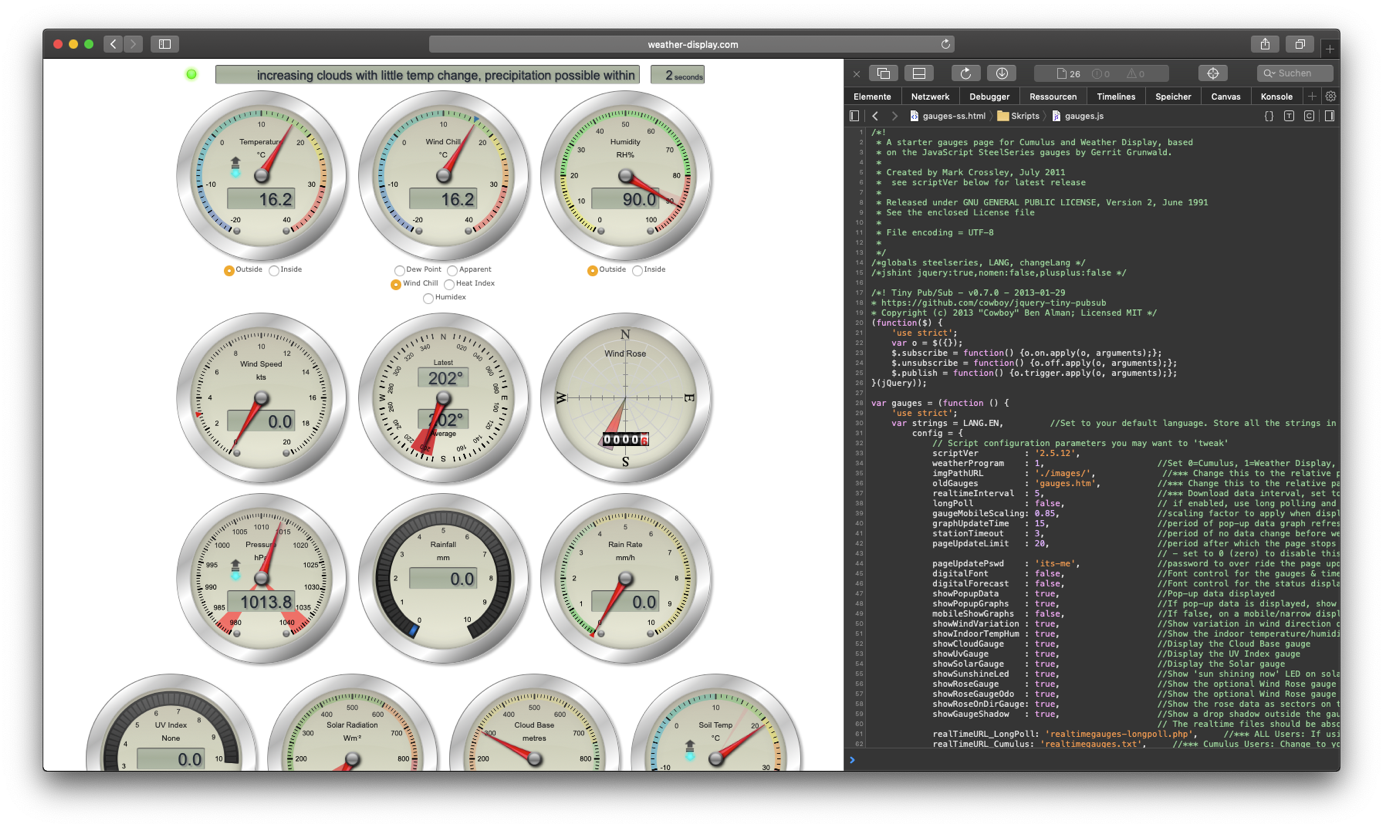


Abbildung 17: <http://www.weather-display.com/windy/gb/gauges-ss.html>, Abgerufen am 19.02.2019, Quelle: Screenshot

Herausgefunden habe ich die Aufgaben folgender Methoden:

Bsp: private final Radial gauge = new Radial();

|  |  |
| --- | --- |
| Methode | Beschreibung |
| gauge.setTitle() | Title des Gauge |
| gauge.setUnitString() | Textfeld für die physikalische Einheit |
| gauge.setFrameDesign() | Wählt aus einer enum das Design aus |
| gauge.setLedVisible() | Aktiviert/Deaktiviert „LED“ |
| gauge.setGlowVisible() | Aktiviert/Deaktiviert „glow“ um den Gauge |
| gauge.setGlowColor() | Farbe des „Glowings“ |
| gauge.setKnobStyle() | Stil des „Knopfes“ in der Mitte aus enum |
| gauge.setKnobType() | Weitere Anpassung des „Knopfes“ |
| gauge.setMaxValue() | Maximalwert der Skala |
| gauge.setMajorTickSpacing() | Abstand zwischen den großen, beschrifteten Strichen |
| gauge.setMinorTickSpacing() | Abstand zwischen den kleinen Strichen |
| gauge.setTickmarkColor() | Farbe der Striche |
| gauge.setTicklabelOrientation() | Orientierung der Beschriftung (Horizontal, Radial, etc.) |
| gauge.setLabelNumberFormat() | Format der Beschriftung (Normal, Scientific, Prozent, etc.) |
| gauge.setThreasholdColor() | Farbe des benutzerdefinierten Markers |
| gauge.setThreasholdType() | Design des Markers (Pfeil oder Dreieck) |
| gauge.setThreasholdVisible() | Aktiviert/Deaktiviert Marker |
| gauge.setValue() | Neuer Wert wird ausgegeben |
| gauge.setValueAnimated() | Neuer Wert wird ausgegeben und vom aktuellen Stand animiert „angefahren“ |
| gauge.setLcdValue() | Neuer Wert für die Digitalanzeige am Gauge |
| setThreashold() | Neuer Wert für den Marker |

Tabelle : Methoden der Bibliothek "SteelSeries"

Über diese Methoden lassen sich die anzeigen personalisieren.

Die Bibliothek verfügt über verschiedenste Klassen für die verschiedenen Anzeigen, verwendet habe ich Radial() für die Tachometer und Linear() für die Thermometer.

## JFreeChart (Leistungsdiagramm)

Um das Leistungsdiagramm zu erstellen, wurde die Bibliothek JFreeChart verwendet. Das „Free“ in JFreeChart ist allerdings nur relativ, da die Bibliothek zwar gratis verwendet werden darf, man aber keine Code-Beispiele erhält, diese erhalten nur zahlende Kunden. Aus diesem Grund habe ich das Diagramm meines Vorgängers übernommen und an mein Programm angepasst.

Die folgende Dokumentation der Methoden stammt aus der Diplomarbeit von Levin Messing[[3]](#footnote-3):

|  |  |
| --- | --- |
| Methode | Beschreibung |
| ChartFactory.createXYLineChart() | Erstellt ein JFreeChart Objekt. Legende, Dataset, Orientierung und Überschrift werden gesetzt. |
| XYPlot.setDataset() | Fügt ein Dataset dem Plot eines JFreeChart-Objektes hinzu. |
| XYPlot.setRangeAxis() | Fügt eine Y-Achse dem Diagramm hinzu. |
| XYPlot.mapDatasetToRangeAxis() | Weist einem Dataset die zu verwendende Y-Achse zu. |
| XYPlot.setRenderer() | Setzt den Renderer, der für das Design zuständig ist. |
| XYPlot.addRangeMarker() | Fügt einen Marker hinzu, der bestimmte Werte markiert. |
| Dataset.addSeries() | Fügt ein Series Objekt dem Dataset Objekt hinzu. |
| ValueMarker.setStroke() | Setzt das Design eines Markers |
| ValueMarker.setLabelTextAnchor() | Setzt die Position vom Text des Markers |
| ValueMarker.setValue() | Setzt den Wert den der Marker hervorheben soll. |
| XYLineAndShapeRenderer.setSeriesStroke() | Setzt das Design der Linie. |
| JFreeChart.addSubtitle() | Fügt einen Untertitel hinzu. |

Tabelle : JFreeChart XYLineChartMethoden, Quelle: Messing, Levin: Diplomarbeit: Zweiradprüfstand. 2017. S.30

## JSON (Lesen und Schreiben der Config-File)

Um Daten im Allgemeinen und Konfigurationsdaten im Speziellen zu speichern, wird gerne das JSON-Format verwendet. Dies bietet den Vorteil, dass es auch für den Menschen lesbar ist, da es auf dem Key-Value-Prinzip basiert.

Leider ist die offizielle Bibliothek javax.json[[4]](#footnote-4) nur in der Java Enterprise Edition verfügbar und muss nachinstalliert werden. Zusätzlich zur javax.json.jar wird noch die javax.json-api-*versionsnummer*.jar benötigt.

|  |  |
| --- | --- |
| Methode | Beschreibung |
| JsonObjectBuilder.add(key, value) | Fügt dem JSON-Objekt einen Wert hinzu: Variable „key“ ist ein String und value kann jeder Datentyp sein. |
| JsonObjectBuilder.build() | Gibt ein JSON-Objekt zurück. |
| JsonReader.readObject() | Liest ein JSON-Objekt aus einem String oder einer Datei ein. |
| JsonObject.getBoolean(key) | Gibt den hinterlegten Boolean-Wert von „key“ zurück. |
| JsonObject.getInt(key) | Gibt den hinterlegten Integer-Wert von „key“ zurück. |
| JsonObject.getJsonNumber(key) | Wird für double benötigt: Gibt JSON-Number zurück. |
| JsonNumber.doubleValue() | Konvertiert JSON-Number in double |

Tabelle : javax.json Methoden

Als Datei geschrieben wird mittels eines BufferdWriter w: w.write(jsonObj.toString());

Eingelesen wird mit einem FileInputStream fis: Json.createReader(fis);

## Kommunikation

## Protokoll

Für die Kommunikation zwischen dem µC und der PC-Applikation wurde ein Protokoll definiert. Es sieht sich als einfaches Request/Response-System bzw. basiert auf dem Master/Slave-Prinzip. Der Master (PC) muss eine Anfrage stellen um eine Antwort vom Slave (µC) zu erhalten. Das Protokoll ist verbindungslos und die Datenübertragung erfolgt textuell.

## Aufbau

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Startzeichen | Daten | Trennzeichen | Prüfsumme | Stoppzeichen |
| „:“ | data#data#data | „>“ | CRC32 | „;“ |

Tabelle : Aufbau des Protokolls

Wenn die Response aus mehreren Daten besteht, wird zwischen den Daten ein Hash „#“ als Trennzeichen verwendet. Die CRC32-Prüfsumme wird über die Daten generiert, ohne Berücksichtigung der Steuerzeichen („:“, „>“, „;“), mit Ausnahme des Hash („#“), der sich als Teil der Daten sieht.

Beispiel: Request „ALL“:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Startzeichen | Daten | Trennzeichen | Prüfsumme | Stoppzeichen |
| „:“ | 0#0#0.00#0.00#800144 | „>“ | 51141376 | „;“ |

Tabelle : Beispiel des Protokoll-Aufbaus

Weitere Informationen zu den Daten gibt es unter dem Punkt *4.6.6.1.3 Requests & Responses*.

## Ablauf

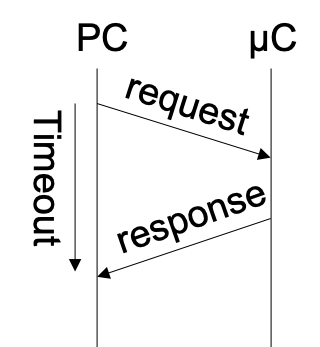


Abbildung 18: Ablauf des Protokolls, Quelle: Eigene Zeichnung

Der PC sendet eine Request aus und speichert den Zeitpunkt des Versendens. Der µC antwortet mit der gewünschten Response. Sollte die Response nicht in einer vorgegebenen Zeit eintreffen, wird bei manchen Requests die Anfrage erneut gestellt, bei anderen einfach verworfen. Die Länge des Timeouts unterscheidet sich auch von Request zu Request.

|  |  |  |
| --- | --- | --- |
| Request | Erneute Anfrage (Timeout) | Länge des Timeouts (ms) |
| INIT | ja | 5000 |
| VERSION | ja | 5000 |
| START | ja | 1000 |
| ENGINE | ja | 1000 |
| KILL | nein | 1000 |
| ALL | nein | 1000 |
| MEASURE | nein | 1000 |
| MEASURENO | nein | 1000 |
| FINE | nein | 1000 |
| WARNING | nein | 1000 |
| SEVERE | nein | 1000 |
| MAXPROBLEMS | nein | 1000 |

Abbildung 19: Timeout-Handling der Requests

Auf eine weitere Überprüfungsmaßnahme außer dem CRC – wie zB das Idle-RQ-Verfahren – wurde verzichtet, da es nicht notwendig ist. Fehlerhafte Pakete werden einfach verworfen, bzw. nicht ankommende Pakete werden ignoriert und das Programm geht zum nächsten Messpunkt.

## Requests & Responses

|  |  |  |  |
| --- | --- | --- | --- |
| Request | Request-String | Response (Data) | Aufgabe am µC |
| INIT | i | BESDyno | - |
| VERSION | p | Versionsnummer | - |
| START | s | Umgebungstemperatur und -luftdruck | Auslesen der Werte des BMP180 |
| ENGINE | e | Motor- und Abgastemperatur | Auslesen der Thermoelemente am ADC. |
| KILL | k | „KILL“ | Rücksetzen des Timers |
| ALL | a | Motor- und Walzendrehzahl,  Motor- und Abgastemperatur und Zeit | Messen der Periodendauer an D2 & D3, Auslesen von A0 & A1 (Thermoelemente) |
| MEASURE | m | Motor- und Walzendrehzahl, Zeit | Messen der Periodendauer an D2 & D3 |
| MEASURENO | n | Walzendrehzahl, Zeit | Messen der Periodendauer an D3 |
| FINE | f | „FINE“ | Setzt grüne LED |
| WARNING | w | „WARNING“ | Setzt gelbe LED |
| SEVERE | v | „SEVERE“ | Setzt rote LED |
| MAXPROBLEMS | x | „MAXPROBLEMS“ | Setzt gelbe und rote LED |
| *DEBUG* | *d* | *Für den Menschen lesbaren Text im Terminal aller Messwerte* | *Alles wird gemessen* |

Tabelle : Arten der Requests & Responses

|  |  |
| --- | --- |
| Request | Response-Beispiel (mit gleichen Werten) |
| INIT | :BESDyno>CRC; |
| VERSION | :1.00>CRC; |
| START | :22.88#989.00#420.00>CRC; |
| ENGINE | :55.32#439.21>CRC; |
| KILL | :KILL>CRC; |
| ALL | :5350#1220#55.32#439.21#800144>CRC; |
| MEASURE | :5350#1220#800144>CRC; |
| MEASURENO | :1220#800144>CRC; |
| FINE | :FINE>CRC; |
| WARNING | :WARNING>CRC; |
| SEVERE | :SEVERE>CRC; |
| MAXPROBLEMS | :MAXPROBLRMS>CRC; |
| DEBUG | Temperatur: 22.88  Luftdruck: 989.00  Höhenmeter: 420.00  A0 Motor: 55.32  A1 Abgas: 439.21  D2 Motor: 5350  D3 Walze: 1220  Zeit (µs): 800144 |

Tabelle : Beispiele für Responses

## Prüfsumme

Um eine fehlerfreie Kommunikation sicherzustellen, wird eine Prüfsumme verwendet. Für die Berechnung der CRC32-Prüfsumme werden die Daten des Protokolls, ohne Steuerzeichen, aber inklusive des Hash („#“), herangezogen.

Für dieses Protokoll wird der CRC-CCITT (0xFFFFFFF) mit dem Polynom 0x4C11DB7 verwendet.

Auf dem Arduino wird eine CRC-Tabelle im Flash-Speicher abgelegt, dies wird über die Bibliothek avr/pgmspace.h mit dem Schlüsselwort PROGMEM realisiert.[[5]](#footnote-5)

Im Java-Programm wird die Bibliothek java.util.zip.CRC32 verwendet. Diese wurde eigentlich zum Zippen entwickelt, aber arbeitet mit den gleichen Werten wie der Algorithmus am Arduino. Sie bietet die Methoden crc.update(Byte[] b) und crc.getValue(), die die Prüfsumme als Long-Variable zurückgibt.

Da es für die Diplomarbeit zu aufwendig gewesen wäre die CRC-Berechnung selbst zu programmieren, habe ich im Internet nach Lösungen gesucht und im Arduino-Forum gefunden[[6]](#footnote-6). Zur Kontrolle wurde ein Online-CRC-Rechner[[7]](#footnote-7) herangezogen:

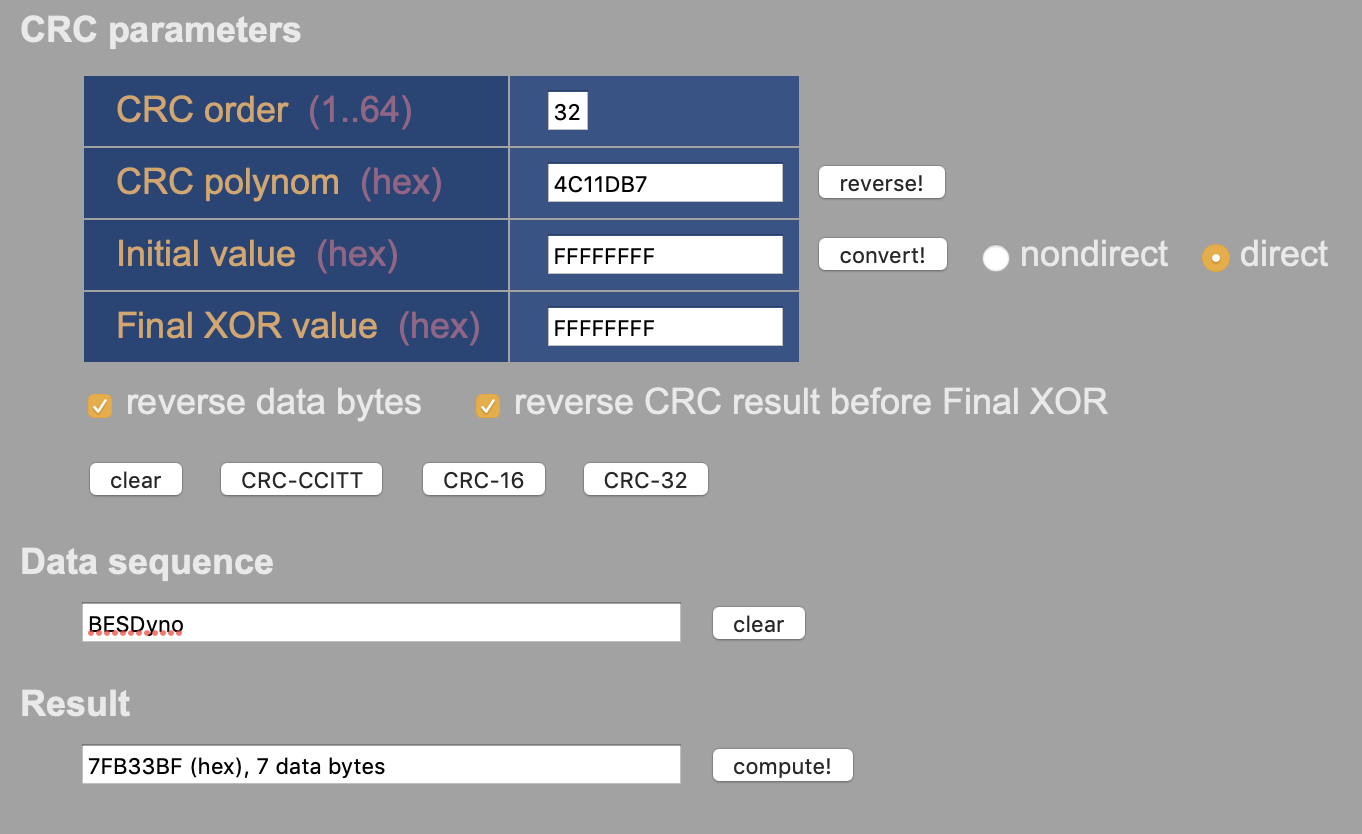


Abbildung 20: Online-CRC-Rechner, http://www.zorc.breitbandkatze.de/crc.html, Quelle: Screenshot

## Behandlung von Fehlern

Wie im Punkt 4.6.6.1.2: Ablauf bereits erwähnt, hängt die Behandlung von Fehlern vom gesendeten Request ab.

Requests, die nur einmal in der Messkette abgefragt werden (INIT, VERSION, START & ENGINE), werden im Falle eines Fehlers (Timeout, CRC-Fehler oder Fehler bei der Überprüfung der Plausibilität) der Request erneut gesendet. Alle anderen Requests zur kontinuierlichen Messung, bzw. zum Setzen der Status-LEDs wird im Falle einer fehlerhaften Response verworfen. Mehrfachantworten, die eventuell auftreten könnten, werden vom Programm ignoriert.

## Serielle Schnittstelle

Die Kommunikation zwischen dem µC und dem PC erfolgt über die serielle Schnittstelle des Computers. Um dies so einfach wie möglich umzusetzen gibt es Bibliotheken für Java.

## JSSC

Eine dieser Bibliotheken ist JSSC[[8]](#footnote-8) (Java Simple Serial Connector). Mit den Methoden openPort() und setParams() wird ein Port zur Kommunikation geöffnet. Mit den folgenden Parametern erfolgt die Kommunikation in diesem Projekt:

* Baudrate: 57600
* Databits: 8
* Stopbits: 1
* Parity: NONE

## RxTxComm

Eine weitere Bibliothek zur Realisierung der Kommunikation ist die deutlich ältere RxTxComm Library. Diese funktioniert ähnlich wie die JSSC Library, allerdings mit anderem Syntax.

## Probleme der RxTxComm

Damit die Bibliothek richtig läuft, muss die binäre Bibliothek für das Betriebssystem manuell geladen werden. Zuerst muss aus der JAR-Datei des Projektes die binären Bibliotheken (\*.so (Linux), \*.jnilib (macOS), \*.dll (Windows)) mit Hilfe der Methode class.getRessourceAsStream()[[9]](#footnote-9) in ein Verzeichnis extrahiert werden. Anschließend die Bibliothek mit System.setProperty(“java.library.path“, path);[[10]](#footnote-10) und System.loadLibrary(“rxtxSerial“); [[11]](#footnote-11) in die Java-VM geladen werden.

Ein weiteres Problem mit der RxTxComm besteht darin, dass wenn bei einem geöffneten Port das Schnittstellenkabel abgezogen wird, das Programm einfriert und die Prozessorauslastung gegen 100% geht. Zu erwarten wäre in diesem Fall eine Exception, die aber ausbleibt – in der JSSC Library wurde dieses Problem deutlich eleganter gelöst.

## Verbindungsaufbau

Obwohl es sich um ein verbindungsloses Protokoll handelt, das ohne Keep-Alive-Packages, etc. auskommt, wird am Anfang überprüft, ob es sich um das richtige Gerät handelt, das angeschlossen ist. Unmittelbar nach dem Öffnen des Ports wird die Request INIT versendet, sollte nichts zurückkommen, handelt es sich um das falsche oder ein kaputtes Gerät. Wenn die Request INIT abgeschlossen ist, folgt die Request VERSION, um sicherzustellen, dass die aktuelle Software am Arduino alle vom Java-Programm benötigten Funktionen unterstützt.

Wenn dieser Vorgang abgeschlossen ist, kann das Gerät verwendet werden.

## Messvorgang

## Messkette

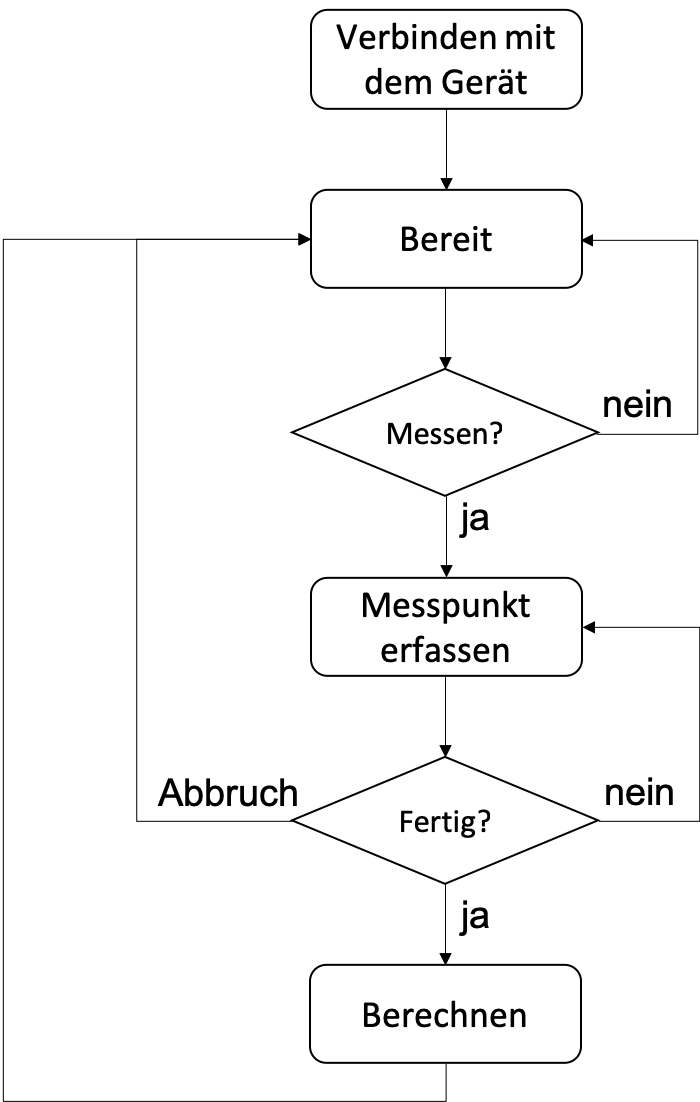
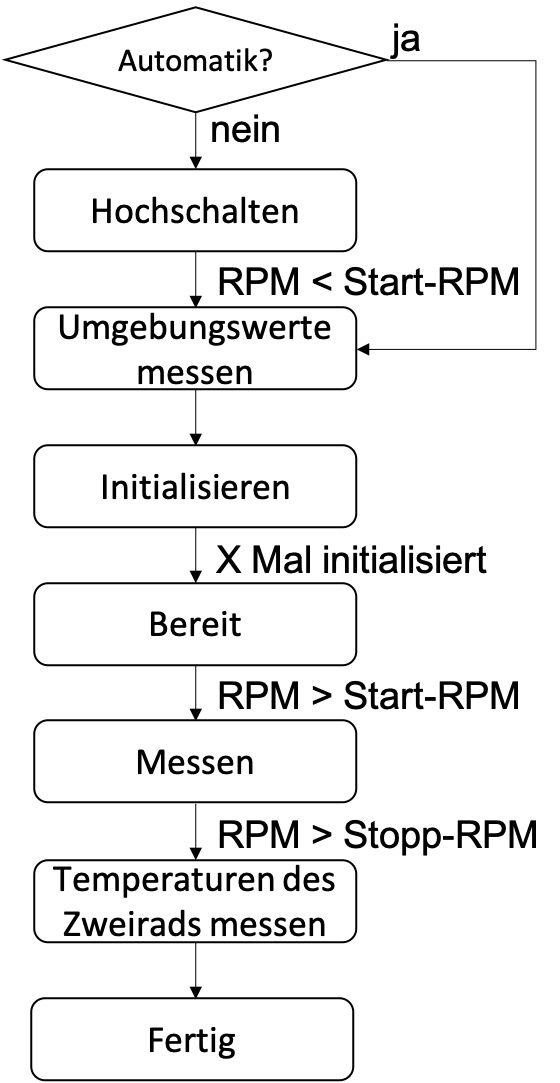
Die folgenden beiden Diagramme stellen den Ablauf der Messung dar. Zum einen den Ablauf im Allgemeinen (Low-Level) und zum anderen den Ablauf in den Punkten „Messpunkt erfassen“ und „Fertig?“, die noch einen internen Ablauf besitzen (High-Level).

Abbildung 21: Messablauf High-Level, Quelle: Eigene Zeichnung

Abbildung 22: Messablauf Low-Level, Quelle: Eigene Zeichnung

## Datenerfassung und -sicherung

Wenn eine Response zu den Requests ALL, MEASURE oder MEASURENO eintrifft, werden die unberechneten Daten als Objekt RawDatapoint in einer Collection gespeichert. In der Klasse, die die Request versendet, die Berechnungen vornimmt und die Live-Anzeigen aktualisiert (measure.MeasurementWorker) wird immer der letzte Wert der Collection verwendet. Da auf die Collection rawList zwei Threads zugreifen, muss dies in einem synchronized-Kontext geschehen. Die Daten werden für die Live-Anzeigen umgerechnet und als Objekt PreDatapoint in der preList gespeichert. Wenn die Messung abgeschlossen ist, wird mit der Klasse measure.Calculate mit den Daten der preList die finalen Daten für das Diagramm berechnet und als Objekt Datapoint in der dataList gespeichert. Wenn mit der Request ALL die Motorradtemperaturen kontinuierlich gemessen werden, werden auch diese in einer Collection gespeichert.

All diese Collections finde sich in der Klasse data.Database wieder, die nach dem Entwurfsmuster Singleton programmiert wurde.

## Auswahl der Collections in Java

In Java gibt es viele praktikablere Alternativen zu einem Array um Daten zu speichern. Besonders hervorzuheben wäre die **Listen**, über die der Wert mit einem Index abgefragt werden kann, und die **Maps**, die nach dem Key-Value-Prinzip funktionieren.

Für unsere Zwecke ist eine Liste das idealste, da wir kontinuierliche Messwerte nacheinander abspeichern wollen. Hier unterscheidet man wieder zwischen LinkedList und ArrayList, die für verschiedene Zwecke geeignet sind.

Folgende Informationen habe ich aus einer stackoverflow-Frage[[12]](#footnote-12):

**ArrayList:**

* Schneller Zugriff auf zufällige Objekte (O(1))
* Möglicherweise langsam beim Hinzufügen, da wenn die Liste voll ist, sie einmal verdoppelt wird und die Daten in die neue Liste kopiert werden müssen (O(1), worst case: O(n))

**LinkedList:**

* Schneller Zugriff mit Iterator (O(1), worst case: O(n))
* Schnell beim Hinzufügen von Daten (O(1))

**Auswahl:**

Für die Collections rawList und preList habe ich als ArrayList realisiert, da immer der letzte Wert der Collection benötigt wird und die ArrayList für random access schneller ist.

Die Collection dataList habe ich aber als LinkedList angelegt, da in nur Daten nacheinander hinzugefügt werden und anschließend darüber iteriert wird.

## Berechnungen

## Live-Anzeigen

Für die Live-Anzeigen müssen die vorhandenen Daten in die Geschwindigkeit der Walze und die Drehzahl des Motors umgerechnet werden.

Wir erhalten vom µC die Periodendauer der Signale des Motors und der Walze (tMotor, tWalze) in µs. Die Frequenz f definiert sich als Kehrwert zur Periodendauer (1/t).

X ist der Faktor zur Kompensation der Taktanzahl. Bei einem Zweitaktmotor wird bei jeder Umdrehung gezündet, daher beträgt x=1. Bei einem Viertaktmotor wird bei jeder zweiten Umdrehung gezündet und x=2.

## Numerische Simulation

Am Ende des Messvorgangs wird mit den erhaltenen Werten die Leistung zu jedem Punkt berechnet. Dafür benötigt wird auch der zeitliche Abstand zwischen den Messpunkten. Der µC übermittelt mit jedem Telegramm die aktuell vergangene Zeit seit dem Beginn der Messung. Da die Werte numerisch simuliert werden, muss der erste Wert verworfen werden, da immer der vorherige Wert benötigt wird.

## Tabellarische Berechnung[[13]](#footnote-13)

|  |  |
| --- | --- |
| Wert | Formel |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Tabelle : Tabellarische Berechnung der Leistung

## Multithreading

Damit in einem Programm mehrere Prozesse parallel ablaufen können, muss mit Threads gearbeitet werden, die ihre Aufgaben jeweils einem anderen Prozessorkern zuweisen. In Java gibt es dafür mehrere Möglichkeiten, folgende habe ich verwendet:

* **SwingWorker** für die Kommunikation
* **SwingWorker** für den Messablauf
* **Thread** für die abschließende Berechnung

## SwingWorker allgemein

Der SwingWorker bietet den Vorteil, dass während der Worker läuft, Daten mit anderen Threads einfach geteilt werden können. Dies wird über die Methoden publish(), die die Werte wegschickt und process(), die die Werte empfängt, realisiert.

Die Methode done() wird nach dem Beenden des Workers aufgerufen und mit der Methode get() kann ein eventuell vorhandener Rückgabewert evaluiert werden.

Der auszuführende Code wird in die Methode doInBackground() geschrieben.

Es handelt sich hierbei um Override-Methoden, die Klasse des Worker muss vorher von der Klasse SwingWorker abgeleitet werden.

## RxTxWorker

Im RxTxWorker wird der Kommunikationsablauf gesteuert. Über eine Collection werden dem Worker abzuarbeitende Requests übergeben, die an den µC gesendet wird. Der Zeitpunkt des Sendens wird gespeichert und dient für den Timeout. Die Response wird über ein serialEvent als Byte-Array abgefangen und mit einem StringBuilder umgewandelt. Anschließend wird jede Response passend weiterverarbeitet.

Der RxTxWorker läuft, sobald der Port geöffnet wird beendet, wenn der Port geschlossen wurde. Während des Prozesses wird über die Methode publish() die Request und deren Status übermittelt, der die Werte WAITINGTOSEND, WAITINGFORRESPONSE, ERROR, TIMEOUT und OK annehmen kann. Je nach Status wird in der Methode process() anders darauf reagiert.

Da die einzelnen Requests abgeleitete Klassen der abstrakten Klasse serial.requests.Request sind, kann über das Schlüsselwort instanceof abgefragt werden, um welche Request es sich handelt, um speziell darauf zu reagieren.

## MeasurementWorker

Der MeasurementWorker arbeitet die Messkette aus Punkt 4.6.7.1 ab. Der Worker startet, sobald die Messkette beginnt und endet, wenn die Berechnung abgeschlossen ist oder die Messung abgebrochen wurde.

Je nachdem welche Daten der Benutz messen will, werden die Requests ALL, MEASURE oder MEASURENO gesendet. Aus der Collection rawList wird der letzte Wert herangezogen und wie in Punkt 4.6.7.3.1 umgerechnet. Diese Werte werden in der preList gespeichert und als Objekt DialData mit der Methode publish() der GUI übergeben. Das Objekt DialData enthält die Drehzahl, die Geschwindigkeit, den aktuellen Status (für die Farbe der Tachos), die erforderliche Drehzahl, die erforderliche Geschwindigkeit und die Motorradtemperaturen. Abschließend wird noch anhand dieser Daten überprüft, ob in der Messkette ein Schritt weitergegangen werden kann.

Die Methode process() ist in einer inneren Klasse von MeasureDialog definiert und setzt die Werte und Farben der Tachos und der Status-Leiste.

## Maßnahmen zur einwandfreien Plattformunabhängigkeit

## Bibliotheken für die serielle Kommunikation

Wie man in der folgenden Profiling-Analyse sehen kann, verbraucht die Bibliothek JSSC sehr viel Prozessorzeit auf macOS.

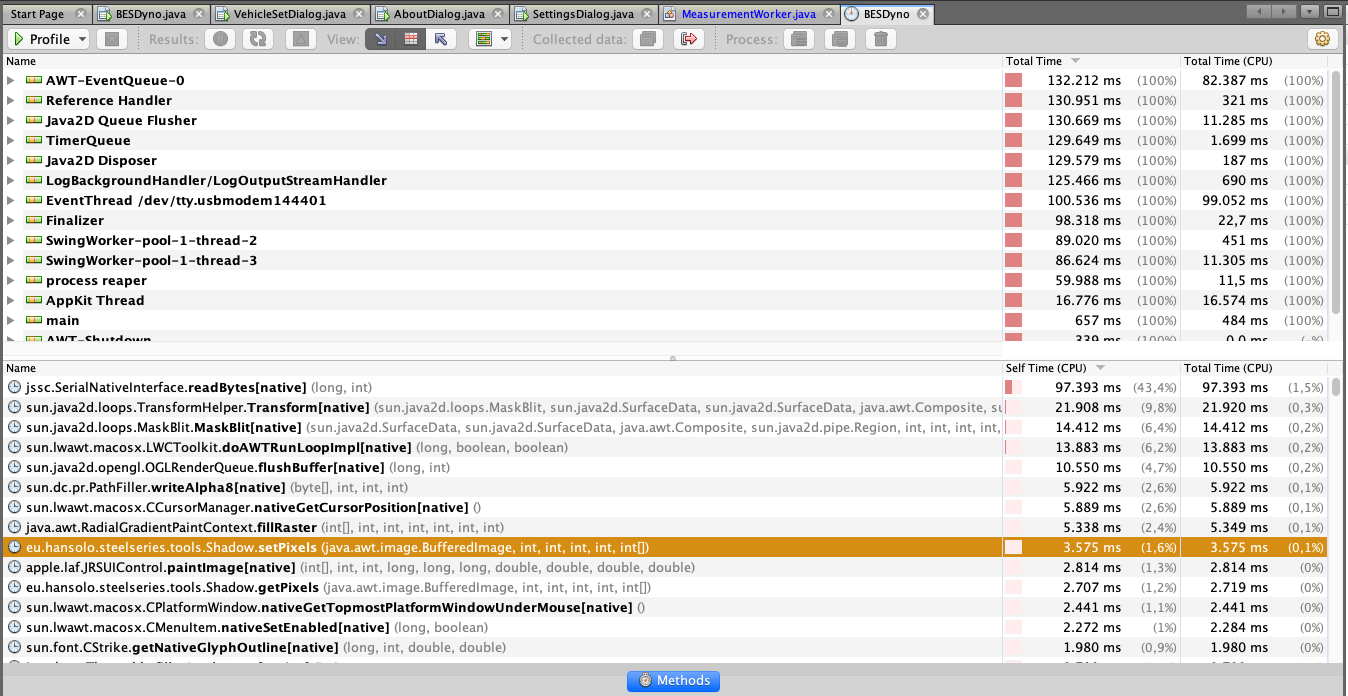


Abbildung 23: Profiling-Analyse des Projektes auf macOS mit Netbeans, Quelle: Screenshot

Daher verwende ich beide Bibliotheken (JSSC und RxTxComm), die in ein Wrapper-Objekt serial.UARTManager verpackt sind und je nach Betriebssystem auf die JSSC-Library (Linux und Windows) oder auf die RxTxComm (macOS) zugreift.

Dieses Problem ist recht interessant, da die Bibliothek auf allen Betriebssystemen funktionieren sollte und offenbar für macOS unzureichend performant programmiert ist. Der Nachteil dabei: Wenn das Programm auf macOS ausgeführt werden soll, hat man mit den Problemen der RxTxComm (Punkt 4.6.6.2.2.1) zu kämpfen.

## OS native Look & Feel

Ein Programm sollte auf einem Betriebssystem aussehen, wie man es gewohnt ist. Dafür wird am Beginn des Programms abgefragt, um welches Betriebssystem es sich handelt und anschließend das korrekte Look & Feel aktiviert.

Für macOS wird, wie in diesem System üblich, zusätzlich die Menüleiste in die Titelleiste verschoben.

Wenn die Abfrage des Betriebssystems nicht aufgrund anderer Punkte notwendig gewesen wäre, hätte man sich dies auch sparen können, da mit der folgenden Methode automatisch das richtige Look & Feel aktiviert wird:

UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());

Für die Menüleiste in macOS:

System.setProperty("apple.laf.useScreenMenuBar", "true");

## Dateinamen

Der Vorgänger von macOS war vor der Jahrtausendwende Mac OS 9. Zu dieser Zeit war das Trennzeichen für Dateipfade noch nicht der Slash („/“), sondern der Doppelpunkt („:“). Aus historischen Gründen ist es auch heutzutage nicht möglich einen Doppelpunkt in einem Dateinamen auf einem macOS-Betriebssystem zu verwenden.

Ein weiteres Problem bereitet uns Windows, da auf diesem Betriebssystem das Trennzeichen der Backslash („\“) ist. Dies lässt sich aber einfach umgehen: Das Attribut File.seperator fügt automatisch je nach Betriebssystem den Slash (Linux und macOS) oder den Backslash (Windows) ein.

## Tastaturkürzel

Früher hatte die Maus eines Apple-Computers nur eine Taste, den Rechtsklick übernahm die Taste „control“. Daher sind auch heutzutage die Tastaturkürzel, die bei anderen Betriebssystemen unter der „ctrl“-Taste zu finden sind, bei macOS unter der „command“-Taste.

Am Beginn des Programms werden je nach Betriebssystem die Tastaturkürzel unter die „ctrl“-Taste (InputEvent.CTRL\_MASK) oder für macOS unter die „command“-Taste (InputEvent.META\_MASK) gelegt.

## Nennenswerte Features der Software

## Einheiten

Im Einstellungsdialogfenster kann der Benutzer wählen, welche Einheiten er bevorzugt. Im ganzen Programm werden dem User dann nur mehr diese Einheiten ausgegeben.

Gewählt werden können:

* Leistung: PS oder kW
* Geschwindigkeit: m/s, km/h oder mi/h
* Temperatur: °C oder °F

## Exportieren und Laden von Config-Files

Da für verschiedene Motorräder unterschiedliche Konfigurationen (Start-Drehzahl, Leerlaufdrehzahl, etc.) erforderlich sind, bietet das Programm, die Config-File zu exportieren und vorhandene Config-Files zu laden. Es handelt sich dabei ebenfalls um eine Datei im JSON-Format.

## Analoge Anzeigen der Bibliothek SteelSeries

Die Bibliothek SteelSeries bietet optisch ansprechende analoge Anzeigen. Diese sind sehr Ressourcenschonend und passen sich je nach verwendeter Einheit (m/s, km/h, mi/h) in ihrer Skalierung an. Die Realisierung der Anzeigen wäre auch mit JFreeChart möglich gewesen, allerdings ist die Handhabung von SteelSeries deutlich einfacher. Außerdem konnten die Anzeigen von JFreeChart optisch nicht überzeugen.

## Live-Anzeige der Motor- und Abgastemperatur

Es ist möglich, die Motor- und Abgastemperatur kontinuierlich zu messen. Dadurch ist es dem Benutzer möglich, die Messung im optimalen Temperaturfenster des Motors zu starten. Sollte eine der Temperaturen den im Einstellungsdialogfenster eingegebenen Maximalwert übersteigen, wird der Benutzer gewarnt, in dem sich die gesamte Oberfläche des Programms rot färbt. Dadurch kann es nicht zu unverhofften Schäden am Motorrad kommen.

## Dark-Mode

Mit der neuesten Version von macOS 10.14 „Mojave“ wurde die Möglichkeit eingeführt, die Oberfläche systemweit zu verdunkeln. Damit sich das Programm besser in diese Umgebung einfügt, kann unter dem Menü-Punkt „Darstellung“ der „Dark-Mode“ aktiviert werden. Entwicklungs- und Testmodus

## CSV-Dateien

Während der Testphase war es notwendig die Rohdaten auf Plausibilität zu überprüfen. Daher können die Listen aus der Klasse Database als CSV-Dateien exportiert werden.

## Aufgetretene Probleme und Verbesserungsmöglichkeiten

## Singleton

Die Klassen data.Bike, data.Config, data.Database und main.BESDyno sind nach dem Entwurfsmuster Singleton entwickelt worden. Dabei muss darauf geachtet werden, dass das Objekt einer Klasse auf jeden Fall nur einmal vorkommt. Um dies zu gewährleisten, wird der Konstruktor auf private gesetzt.

In der public static void main() wurde anfangs dabei vergessen, getInstance() statt den Konstruktor aufzurufen, was dazu führte, dass das Objekt zweimal erzeugt wurde.

## Kommunikation (Timeout)

In der Anfangsphase des Projektes wurde vergessen für die Abarbeitung der Requests ein Timeout einzubauen. Dies hat dazu geführt, dass wenn eine Response nicht eintrifft, das Programm “ewig” auf diese wartet. Somit würde auch das Programm hängen bleiben, wenn man zum Beispiel das Verbindungskabel abziehen würde.

## Messvorgang JSSC

Java ist zwar eine plattformunabhängige Programmiersprache, aber die serielle Kommunikation ist je nach Betriebssystem anders realisiert. Dafür sind Bibliotheken wie JSSC zuständig. Bei den ersten Tests auf macOS stellte sich heraus, dass das Programm relativ viele Ressourcen benötigt (in Abbildung xy zu sehen). Aufgrund dieser Erkenntnis wurde zusätzlich die Bibliothek RxTxComm implementiert. Die Kommunikation findet nun über eine Wrapper-Klasse statt, die je nach Betriebssystem auf JSSC oder RxTxComm zugreift. Dies führte aber nicht zur gewünschten Verbesserung und das Programm läuft auf macOS – verursacht durch die serielle Kommunikation – nach wie vor schwach (Abbildung yz).

## Berechnung

## JFreeChart

## SteelSeries

## µC-Programmierung

## Mikrocontroller

## Peripherie

## UART

## ADC

## I2C

## Messungen

## Thermoelemente: Motor- & Abgastemperaturen

Die beiden Thermoelemente sind an Messverstärker angeschlossen, die abhängig von der Temperatur eine Spannung zwischen 0V und 5V ausgibt. Um diese Werte mit dem µC einzulesen, wird der interne ADC verwendet. Dafür wird die Funktion analogRead(pin); verwendet, die einen Wert zwischen 0 und 1024 zurückgibt. Dieser Wert wird wie folgt umgerechnet:

## BMP180: Umgebungstemperatur & -luftdruck

Der BMP180 ist ein intelligenter Sensor, er verfügt über eine I2C-Schnittstelle. Um die Programmierung zu vereinfachen, wird eine Bibliothek von Adafruit verwendet, die die Methoden bmp.readTemperature(), bmp.readPressure() und bmp.readAltitude() verfügt. Es handelt sich hierbei um eine Bibliothek für C++, daher werden die Methoden über ein vorher angelegtes Objekt aufgerufen.

## Drehzahlen

## Variante 1: Zählen der Impulse

Eine Möglichkeit zur Messung der Drehzahl besteht im Zählen der am Digital-Pin in einer bestimmten Zeit ankommenden Flanken. Mit attachInterrupt(pin, ISR(), RISING); wird bei jeder ankommenden steigenden Flanke in die ISR (Interrupt Service Routine) gesprungen. Dort wird der Wert einer Zählvariable inkrementell erhöht. Wenn vom PC eine Anfrage zur Messung der Drehzahl ankommt, werden die Werte übermittelt und die Zählvariablen wieder zurückgesetzt.

## Variante 2: Messen der Periodendauer

Variante Nr. 1 hat sich als nicht praktikabel herausgestellt, da die Messung zu ungenau wird. Ein Zweitakt-Verbrennungsmotor gibt nur einen Impuls pro Umdrehung, daher muss zur genauen Berechnung die Zeit zwischen zwei Messpunkten relativ groß sein (>100ms). Das führt dazu, dass man zu wenig Messpunkte erhält und die Kurve unbrauchbar wird.

Daher haben wir uns für folgende Variante entschieden: Mit der Funktion pulseInLong(pin); wird die Periodendauer des ankommenden Signals in Mikrosekunden gemessen. Die Drehzahl ist die Frequenz des Signals, die mit berechnet wird.

Bei einer Anfrage des PCs nach den Drehzahlen wird die seit Beginn der Messung vergangene Zeit gesendet. Diese kann durch die Requests START und KILL zurückgesetzt werden.

## CRC-Berechnung

## Status-LEDs, optische Rückmeldung an den User

Um die Verwendung des Gesamtsystems noch weiter zu vereinfachen, wurden auf der Oberseite des Gehäuses drei Status LEDs angebracht.

|  |  |
| --- | --- |
| LED leuchtet: | Ursache |
| Alle | Gerät wurde noch nicht verbunden |
| Grün | Alles in Ordnung |
| Gelb | Inkorrekte Erfassung der Drehzahl oder Problem im Java-Programm |
| Rot | Sensor nicht (zB BMP) nicht verfügbar. |

Tabelle : Status-LEDs

Links neben den Status-LEDs befinden sich zwei gelbe LEDs zur Kontrolle der korrekten Erfassung der beiden Drehzahlen. Immer wenn ein Interrupt durch eine Flanke im Arduino ausgelöst wird, wird die entsprechende LED getoggelt. Dadurch ist erkennbar, ob das Programm richtig arbeitet.

## Wichtigste Funktionen der Arduino.h

## micros();

Die Funktion micros(); liefert die aktuelle Systemzeit in µs zurück. Am Anfang des Programms wird diese Funktion aufgerufen und der Wert in eine Variable gespeichert. Um die vergangene Zeit seit Beginn der Messung zu erhalten, muss man nur die Funktion erneut aufrufen und den Startwert abziehen.

## attachInterrupt();

Mit attachInterrupt(pin, ISR(), RISING); wird der externe Interrupt aktiviert. Die für einen Interrupt zur Verfügung stehenden Pins am Arduino UNO sind D2 und D3. Der erste Parameter ist die Nummer des Pins, der zweite die Funktion in die gesprungen werden soll und der dritte bescheibt, auf welche Art von Flanken reagiert werden soll (RISING, FALLING, CHANGE).

## pulseInLong();

## pinMode();

## analogRead();

## Aufgetretene Probleme und Verbesserungsmöglichkeiten

## Elektronik

## Mechanik

## Anhang

## Literaturverzeichnis

## Abbildungsverzeichnis

## Tabellenverzeichnis

1. Vgl: (Stackoverflow, 2011) [↑](#footnote-ref-1)
2. Vgl: (Grunwald, 2012) [↑](#footnote-ref-2)
3. (Messing, 2017) [↑](#footnote-ref-3)
4. Vgl: (Stackoverflow, 2016) [↑](#footnote-ref-4)
5. Vgl: (Arduino Referenz, 2019) [↑](#footnote-ref-5)
6. Vgl: (Arduino Forum, 2012) [↑](#footnote-ref-6)
7. (Sven's Home Page, 2001) [↑](#footnote-ref-7)
8. Vgl: (Javadox, 2013) [↑](#footnote-ref-8)
9. Vgl: (Sokolov, 2014) [↑](#footnote-ref-9)
10. Vgl: (Stackoverflow, 2011) [↑](#footnote-ref-10)
11. Vgl: (Stackoverflow, 2009) [↑](#footnote-ref-11)
12. (Stackoverflow, 2008) [↑](#footnote-ref-12)
13. (Tinauer, 2017) [↑](#footnote-ref-13)