Grafische Benutzeroberfläche – Florian Boemmel

1. Generelles

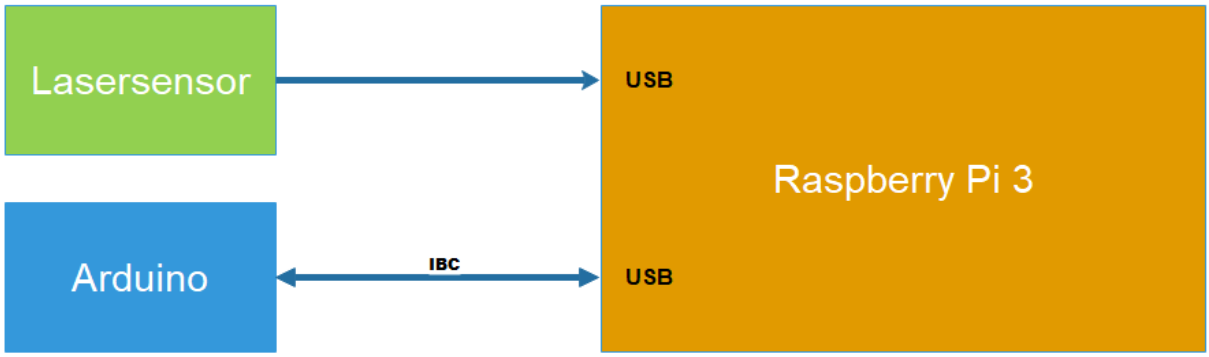
Die grafische Benutzeroberfläche (im folgenden GUI bezeichnet) stellt, abstrakt dargestellt, dass Bindungsglied zwischen Benutzer und dem Fahrzeug da. Über diese, soll die Steuerung des Fahrzeugs erfolgen.

Die GUI soll unter den Aspekten der Skalierbarkeit und der einfachen Erweiterung durch andere Projektmitglieder entwickelt werden. Aus diesem Grund, einigte sich das Projektteam darauf, dass alle Module auf dem Raspberry Pi 3 Model B (im folgenden Pi bezeichnet) in C++ entwickelt werden.

Module stellen hierbei externe Klassen da. Diese werden unabhängig von der GUI entwickelt und müssen anschließend in die GUI eingebunden werden.

Folgende Module existieren:

- Lasersensor
- IBC



2. Entwicklungsumgebung

Während der Anfangsphase des Projekts stand die Zielplattform der GUI noch nicht eindeutig fest. Konkret bedeutete dies, dass das Team zwischen einer Desktop-Anwendung und einer Touch-Anwendung direkt auf dem Pi oder einer App schwankte. Jedoch ist gerade die Zielplattform ein ausschlaggebender Faktor, um das passende GUI-Toolkit auszuwählen.

Auf der Basis der unbekannten Zielplattform, sei es eine Desktop-Anwendung unter Linux / Windows / OSX oder eine Mobile-Anwendung, recherchierte ich über mögliche GUI-Toolkits. Das C++ basierende GUI-Toolkit Qt stach dabei vermehrt heraus. Demnach ist es möglich, auf der Basis eines Projekts alle Zielplattformen zu bedienen.

Qt bietet zusätzlich die Möglichkeit in gewissen Umfang plattformunabhängig zu entwickeln. Konkret bedeutet dies, dass Layout und plattformunabhängige Logik auf jedem Betriebssystem entwickelt und getestet werden kann. Lediglich betriebssystemspezifische Logik kann nur auf dem dazugehörigen Rechner getestet werden. Eine Kompilierung ist jedoch per Cross-Kompilierung möglich.

Weiterhin basiert Qt auf C++. Somit können auf C++ basierende Module ohne weitere Probleme in das Projekt eingefügt werden und erfüllt somit die Voraussetzung des Teams, alle Module auf dem Pi in C++ zu entwickeln.

Ein weiterer Vorteil in Qt liegt in der enorm großen Community und dem einfachen Zugang zu sehr detaillierten Dokumentationen und Beispielen. Qt stellt außerdem auf den gängigsten Plattformen Ihre eigene IDE(QtCreator) mit einem Designer zur Verfügung.

Das Team einigte sich schlussendlich auf eine Touch-Anwendung direkt auf dem Pi. Dazu wurde ein Touchdisplay der Größe 3.2 Zoll direkt auf dem Pi angebracht. Der Pi bietet eine große Auswahl an Möglichkeiten, jedoch ist seine Rechenleistung begrenzt und für einige Tätigkeiten, wie z.B. eine umfangreiche GUI direkt auf ihn zu programmieren eher ungeeignet.

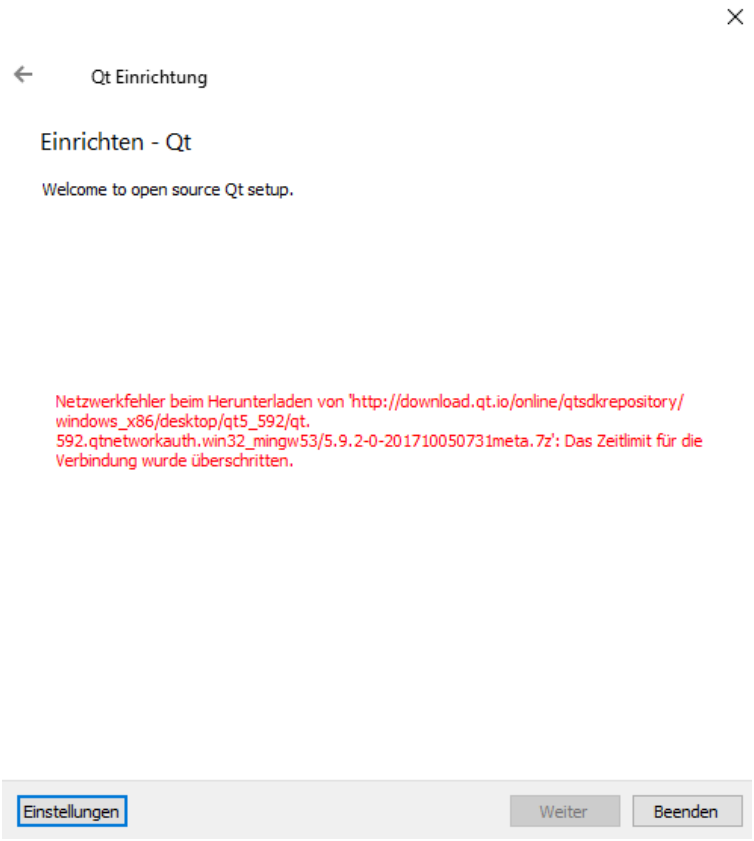
Aus den bereits aufgeführten Gründen wird die GUI in der Sprache C++ mit dem GUI-Toolkit Qt5 realisiert. Dabei möchte ich noch kurz anfügen, dass ich bis dato noch nichts mit Qt gemacht habe und generell noch keine GUI geschrieben habe.

3. Installation & Einrichtung von QtCreator

Qt ist in zwei Versionen verfügbar. Zu einem Open Source und Commercial. Die Unterschiede können unter der Download-Seite eingesehen werden. Ich verwende die kostenlose Open Source Version.

I. Installation unter Windows

Ich möchte hier ausführlicher auf die Installation unter Windows eingehen, da es ein entscheidendes Problem dabei gab. Dies ist auch unter OSX zu beobachten. Downloadet man die Installationdatei über die Downloadseite und führt diese aus, kann es passieren, dass folgende Fehlermeldung während der Installation auftritt:



Auch eine erneute Ausführung der Installation führte zum gleichen Ergebnis. Die einzige Lösung hierfür war es, anstatt der Online-Installation, eine Offline-Installation durchzuführen. Für die Offline-Installation muss zunächst unter der schwer zu findenden Offline-Downloadseite die gewünschte Version gewählt werden und anschließend die Plattform. Danach sollte die Installation reibungslos funktionieren.

Ein letzter wichtiger Punkt ist das Auswählen der zu installierenden Pakete. Unter Windows reicht die von Qt standartmäßig ausgewählten Pakete. Jedoch sollte unter dem Punkt Tools MinGW 5.* ausgewählt werden, falls dieser noch nicht zuvor installiert wurde. Dieser stellt den Standard Compiler unter Windows dar.

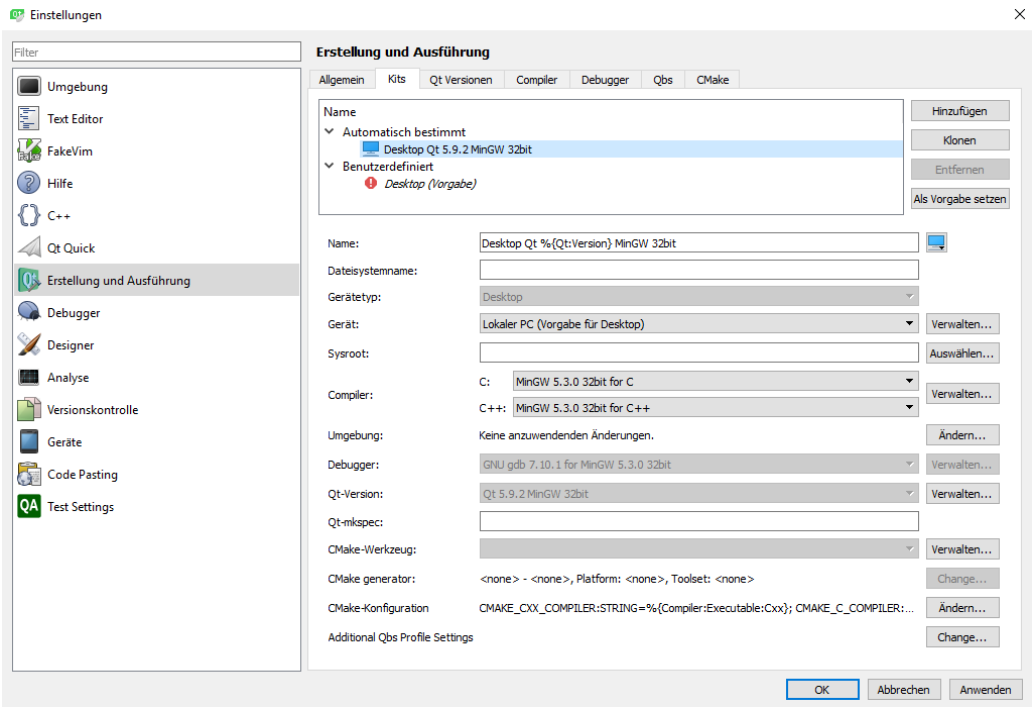
II. Installation unter Linux (Raspberry Pi)

Unter Linux gestaltet sich die Installation etwas einfacher. Dazu müssen lediglich folgenden Kommandos im Terminal ausgeführt werden:

```
sudo apt-get update
sudo apt-get dist-upgrade
sudo apt-get install qt5-default

sudo apt-get install qtcreator

sudo apt-get install libqt5serialport5
sudo apt-get install libqt5serialport5-dev
```

III. Einrichten

Einige erstmalige Einstellungen sind für einen korrekten Kompiliervorgang nötig. Dazu muss ein Kit (Bezeichnung von Qt) eingerichtet werden, falls dies nicht automatisch geschieht. Auch im Falle eines Fehlers beim Kompiliervorgang kann es am nicht korrekt eingestellten Kit liegen.



Dazu muss unter dem Punkt Compiler ein C- sowie C++-Compiler eingestellt sein. Ist das Kit, in diesem Fall „Desktop“ unter dem Reiter Automatisch bestimmt, nicht rot oder gelb markiert, ist das Kit erfolgreich eingestellt worden und eine Kompilierung ist nun möglich. Die Einrichtung unter Linux ist äquivalent. Lediglich die Compiler sind verschieden.

IV. Cross-Kompilierung vs. „Copy And Paste“

Um eine GUI auf einer spezifischen Zielplattform ausführen zu können, muss diese mit einem für die Zielplattform geeigneten Compiler übersetzt werden. Qt bietet die Möglichkeit einer Cross-Kompilierung an.

Jedoch gestaltete sich die Einrichtung eines Cross-Compilers als schwierig. Bereits bei der Beschaffung der richtigen Source-Dateien, passend, für den Pi stellte sich heraus, dass dies ohne externe Tools nicht möglich war. Weiterhin konnte die Dokumentation von Qt mir auch nicht entscheidend weiterhelfen.

Ich probierte aus diesem Grund ein einfaches „Copy and Paste“ aus. Genauer beschrieben, entwickelte ich ein minimales Beispiel auf meinen Windows Computer und pushte dieses anschließend auf GIT und pullte dieses auf den Pi. Unerwartet konnte das Projekt ohne Probleme auf dem Pi geöffnet werden und übersetzt werden.

Ich entschied mich von nun an, die GUI auf einem stärkeren Rechner zu entwickeln und anschließende Tests direkt auf dem Pi durchzuführen. Dies erwies sich im Verlauf des Projekts als vorteilhaft. Einige Einschränkungen gab es jedoch trotzdem.

Plattformspezifische Funktionalitäten mussten entweder auskommentiert oder per Compiler-Schalter deaktiviert werden. Auch nach der Integrierung des Protokolls von Robert war das Problem der plattformspezifischen Funktionalitäten stets gegenwärtig, da in diesem Linux Systemaufrufe getätigt werden. Daraufhin setzte ich ein virtuelles Ubuntu auf, um weiterhin, ohne weitere Compiler Schalter, kompilieren zu können und somit die Entwicklung etwas angenehmer zu gestalten.

4. Anforderungen

/G0101/ **Automatischer Start der Benutzeroberfläche:** Verbindet der Benutzer das Fahrzeug mit einer von Ihm gewählten Stromquelle, bootet der Raspberry Pi direkt in die Benutzeroberfläche des Fahrzeugs und verhindert so eine falsche Bedienmöglichkeit des Fahrzeugs.

/G0102/ **Initialisierung des Fahrzeugs:** Der Benutzer kann über einen Button das Fahrzeug initialisieren. Das bedeutet im konkreten Fall, dass zunächst ein Serieller Port geöffnet wird und das Inter Board Protocoll (IBC) gestartet wird. Weiterführende Steuerungsmöglichkeiten dürfen dem Benutzer zu diesem Zeitpunkt nicht zu Verfügung stehen.

/G0103/ **Moduswahl:** Der Benutzer hat die Möglichkeit zwischen zwei Betriebsmodi auszuwählen:

- Uhrsteuerung
- Controllersteuerung

Zusätzlich muss der Benutzer, ohne einen Modus auszuwählen, die Möglichkeit erhalten, die Raumkartographie zu starten.

/G0104/ **Neustart der Benutzeroberfläche:** Der Benutzer muss über ein Menü die Möglichkeit erhalten, die Benutzeroberfläche neu zu starten. Dies ist insbesondere bei Verbindungsproblemen zum Mikrocontroller unabdingbar.

/G0105/ **Beenden des Systems:** Der Benutzer muss über ein Menü die Möglichkeit erhalten, die Benutzeroberfläche sowie den Raspberry Pi ordnungsgemäß herunterfahren zu können.

/G0106/ **Uhrsteuerung:** Wählt der Benutzer den Modus Uhrsteuerung, muss diesem zunächst eine kurze Anleitung dargestellt werden, wie er die Uhren anzulegen hat. Hat der Benutzer diese Information verstanden, muss er diese bestätigen. Nach der positiven Bestätigung, muss dem Benutzer die Steuerung anhand von Bildern und Animationen verständlich erklärt werden. Zudem muss der Benutzer über einen Button die Möglichkeit gegeben werden, den Raumscan zu starten (/G0111/).

/G0107/ **Controllersteuerung**: Wählt der Benutzer den Modus Controllersteuerung, wird dieser aufgefordert, den Controller griffbereit zu halten. Hat der Benutzer diese Information verstanden, muss er diese bestätigen. Nach der positiven Bestätigung, muss dem Benutzer die Steuerung anhand von Bildern und Animationen verständlich erklärt werden. Zudem muss der Benutzer über einen Button die Möglichkeit gegeben werden, den Raumscan zu starten (/G0111/).

/G0108/ **Navigation:** Der Benutzer muss jederzeit die Möglichkeit erhalten, zur Moduswahl (/G0103/) zurückzukehren und einen anderen Modus wählen zu können. Dabei darf die Navigationstiefe in einem Modus keine Rolle spielen.

/G0109/ **Darstellung der Sensorwerte:** Dem Benutzer muss nach der Wahl, sich die Sensorwerte anzeigen zu lassen, eine Übersicht der vorhandenen Sensoren und deren aktuellen Werten dargestellt werden.

/G0110/ **Fehleranzeige:** Dem Benutzer muss eine Fehleranzeige bereitgestellt werden. Diese muss unabhängig von allen Darstellungen und Benutzereingaben jederzeit gut sichtbar sein. Weiterhin müssen dem Benutzer spezifische Details über einem Fehlerfall dargestellt werden.

/G0111/ **Raumkartographie**: Der Benutzer muss die Möglichkeit erhalten, nach der Wahl eines Modi, die Raumkartographie zu starten. Während die Raumkartographie aktiv ist, müssen dem Benutzer die Sensordaten dargestellt werden (/G0109).

5. Umsetzung

1) Implementierung der GUI

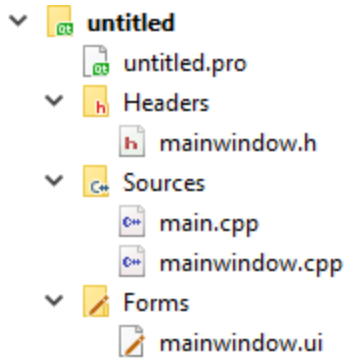
Im folgendem Kapitel wird der Verlauf meiner Implementierung der GUI erläutert und die daraus resultierenden Ergebnisse. Ich weiße an dieser Stelle ausdrücklich darauf hin, dass das Protokoll von Robert Graf sowie der Lasersensor von Anja Strobel in diesem Kapitel noch nicht integriert wurde. Die Integration der beiden Module wird im nächsten Kapitel beschrieben.

I. Projekterstellung

Die im Rahmen des Projekts entwickelte GUI, ist eine Qt-Widget-Anwendung. Diese kann wie folgt generiert werden:

I. Neues Projekt
II. Anwendungen → Qt-Widget-Anwendung
III. Namen für das Projekt vergeben und einen Pfad auswählen
IV. Das passende Kit auswählen (Falls keins vorhanden ist, muss eins, wie unter Punkt 3 (Einrichten), erzeugt werden)
V. Nun kann der Klassenname für das Hauptfenster sowie die Basisklasse eingestellt werden. Dabei wählt man bei der Basisklasse QMainWindow und einen gewünschten Namen für die Klasse und deren Quelldateien.
VI. Schließlich kann noch eine Versionsverwaltung eingestellt werden.

Nun ist das Projekt angelegt und enthält folgende Dateien:



- .pro: Durch ein qmake wird aus diesem ein Makefile
- mainwindow.h: Die Headerdatei des Hauptfensters
- mainwindow.cpp: Die Sourcedatei des Hauptfensters
- mainwindow.ui: Die Formulardatei des Hauptfensters
- main.cpp: Die Main der GUI-Anwendung

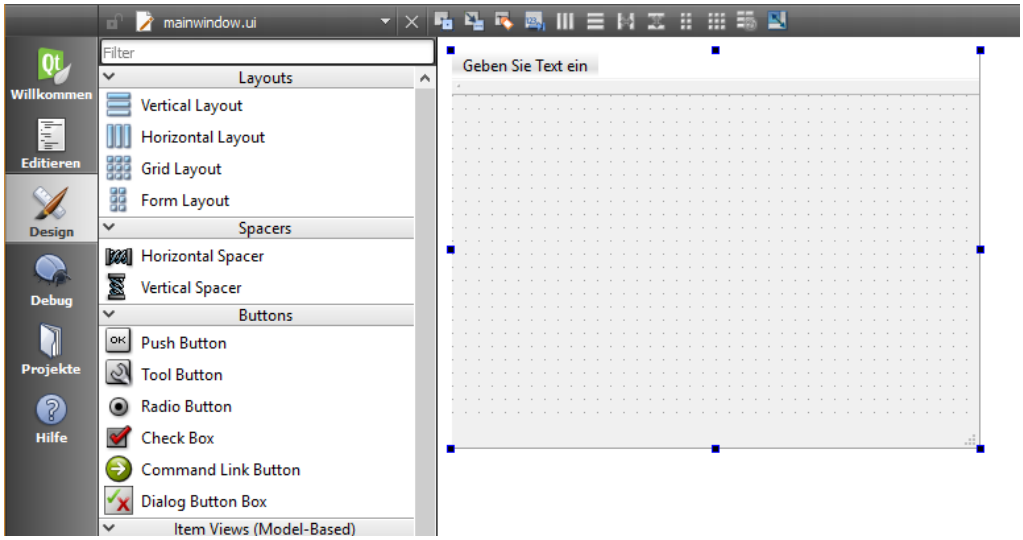
Im Folgenden wird auf die main.cpp genauer eingegangen, um das Grundprinzip von Qt besser zu verstehen:

```cpp
#include "mainwindow.h"
#include <QApplication>

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

    return a.exec();
}
```

Der Ablauf einer Qt-Widgets-Anwendung ist denkbar simpel. Zunächst wird eine Instanz von QApplication erzeugt.

Anschließend wird eine Instanz des Hauptfensters erstellt und mit der Methode show() wird dieses anschließend angezeigt. Würde man show() nicht aufrufen, würde die Anwendung dennoch ausgeführt werden, nur ohne Fenster.

Das letzte Kommando übergibt die Kontrolle des Programms. Die Kontrolle bedeutet hier, dass der Aufruf a.exec() Qt anweist, auf Events zu hören. Ohne diese Anweisung, wären keine Benutzerinteraktionen möglich. a.exec() wird erst beendet, wenn die Qt-Anwendung beendet wird. Somit wird das return erst nach der Beendigung der Qt-Anwendung ausgeführt.
Weiterhin ist eine genauere Betrachtung der Formulardatei des Hauptfensters ratsam:



Öffnet man diese, wird automatisch der integrierte QT-Designer (im Folgenden als Designer bezeichnet) aufgerufen. Dort kann das Hauptfenster geändert werden und Kontroll- sowie Layoutelemente hinzugefügt werden. Formulardateien sind XML-Basiert und können im Qt-Creator nur über den Designer geändert werden.

II. Designer vs. Code

Zu Beginn jedes Projekts, stellt sich die Frage, sollte man den Designer verwenden oder nicht? Dabei gehen die Meinungen sehr weit auseinander.

Gerade bei großen Projekten wird der Designer oft empfohlen und bei kleineren die codebasierende Lösung. Jedoch gibt es auch dort andere Meinungen.

Auch ich überleget lange, welche Herangehensweise ich nutzen möchte für unser Projekt. Es folgten einige Versuche der beiden Möglichkeiten. Ich entschied mich schlussendlich für eine reine Implementierung des Designs und damit gegen die Verwendung des Designers.

III. Layout der GUI

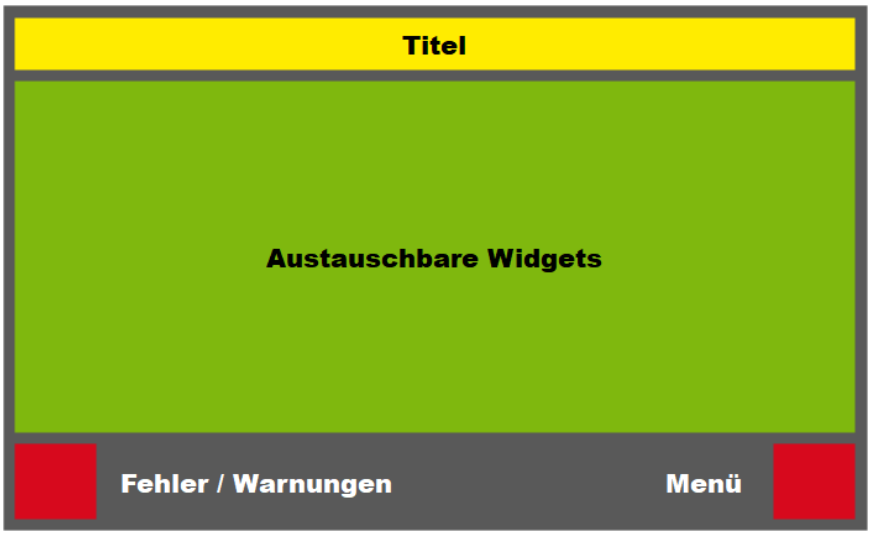
Zunächst musste ich mir ein Grundkonzept des Layouts der GUI überlegen. Wir verwendeten ein 3.2 Zoll großes Touchdisplay (im Folgenden als Display bezeichnet) direkt auf dem Pi. Demnach schränkte bereits das Display das Konzept drastisch ein:

- Die erkannten Eingaben sind ungenau und werden nicht immer gleich erkannt
- Die Größe des Displays ist für die geforderten Anforderungen nicht optimal

Aus dem ersten Punkt folgte die Einschränkung, dass Kontrollelemente eine ausreichende Größe besitzen und gegen eine doppelte Betätigung abgesichert werden müssen.

Aus dem zweiten Punkt folgte die Einschränkung, dass Informationen kompakt dargestellt werden müssen. Dies gilt auch für Kontrollelemente.

Der daraus entstandene Entwurf des Layouts:



Dabei stellt der Titel ein einfacher Schriftzug mit dem Text „StarCar“ dar.

Durch das Konzept der Austauschbaren Widgets wird die Einschränkung der kompakten Informationsdarstellung erfüllt. Im Detail bedeutet dies, das die dargestellten Elemente ein sogenanntes Masterlayout bilden, dass sich nicht ändert und nur die wichtigsten Elemente jederzeit erreichbar sind. Wie der Zugang zu der Fehleranzeige sowie dem Menü.

Weiterhin wurden den wichtigsten Kontrollelementen eine ausreichende Größe zugewiesen und somit wird der Einschränkung der ausreichenden Größe erfüllt. Alle weiteren Kontrollelemente müssen im späteren Entwicklungsprozess an die verfügbare Größe der Austauschbaren Widgets angepasst werden.

IV. Masterlayout

Zu Beginn wurde das Masterlayout entwickelt. Dabei wurde zunächst der Fokus auf die Implementierung des zuvor entworfenen Masterlayouts gelegt. Dabei erstellte ich drei Methoden:

- generateLayout()
- generateStyle()
- setupConnects()

In der ersten Methode wird das Layout des Fensters folgendermaßen generiert:

```
void HomeWindow::generateLayout(){

    centralVBox         = new QVBoxLayout(ui->centralWidget);
    hBox1               = new QHBoxLayout();
    lblHeadline         = new QLabel();
    pButtonAlert        = new QPushButton(QIcon();
    pButtonExit         = new QPushButton(QIcon();

    mainStackedWidget   = new QStackedWidget();

    centralVBox->addSpacing(8);
    centralVBox->addWidget(lblHeadline,0,Qt::AlignHCenter);
    centralVBox->addSpacing(12);
    centralVBox->addWidget(mainStackedWidget);
    centralVBox->addLayout(hBox1);

    hBox1->addSpacing(5);
    hBox1->addWidget(pButtonAlert);
    hBox1->addSpacing(200);
    hBox1->addWidget(pButtonExit);
    hBox1->addSpacing(5);

    centralVBox->addSpacing(5);
}
```

Qt bietet für die Strukturierung von Elementen „Layouts“ an. Ich verwendete QVBoxLayouts und QHBoxLayouts. Dabei stellen diese zu einem eine Box dar, in die Elemente vertikal und zum anderen horizontal angeordnet werden.

In die vertikale Box wird zunächst der Titel in Form eines Labels eingefügt und anschließend ein QStackedWidget. Ein QStackedWidget ist ein Container und seine Hauptfunktionalität besteht darin, dass Widgets in diesem gestapelt werden können. Schließlich wird eine horizontale Box eingefügt. Diese beinhaltet einen Button für die Fehleranzeige und einen für das Menü.

Zusammengefasst bilden diese Elemente das Masterlayout. Alle späteren Widgets werden nur im QStackedWidget angezeigt.

In der zweiten Methode wird das erzeugte Layout und die hinzugefügten Kontrollelemente gestylt:

```
void HomeWindow::generateStyle(){

/*****************************Margins*****************************************/

    ui->centralWidget->setContentsMargins(0,0,0,0);
    centralVBox->setContentsMargins(0,0,0,0);

/*****************************StyleSheets*************************************/

    this->setStyleSheet("QWidget{"
                        "background-color: #2b2b2b;}");

/*****************************Windowstyle*************************************/

    this->setFixedSize(320,240);
    setWindowFlags(Qt::FramelessWindowHint);

/*****************************Button & Label**********************************/

    pButtonExit->setIconSize(QSize(32,32));
    pButtonExit->resize(32,32);

    pButtonAlert->setIconSize(QSize(32,32));
    pButtonAlert->resize(32,32);

    lblHeadline->setText("StarCar");

    lblHeadline->setStyleSheet("QLabel{"
                               "color: yellow;"
                               "font-family: TimesNewRoman;"
                               "font-style: normal;"
                               "font-size: 15pt;"
                               "font-weight: bold;}");

    pButtonExit->setStyleSheet("QPushButton{"
                               "border-radius:  10px;"
                               "border-width:   3px;"
                               "border-color:   black"
                               "border-style:    solid}");

    pButtonAlert->setStyleSheet("QPushButton{"
                                "border-radius:  10px;"
                                "border-width:   5px;"
                                "border-color:   black"
                                "border-style:    solid}");
}
```

Im ersten Schritt werden die Abstände des Masterlayouts gesetzt. Nach den beiden ersten Kommandos werden die Ränder entfernt und somit der gesamte Platz ausgenutzt. Standartmäßig werden 8 Pixel Rand automatisch gesetzt.

Ein ganz wichtiger Punkt ist es, wenn man nicht mit dem Designer arbeitet, dass man die Fenstergröße explizit festlegt. In meinem Fall gleich der Größe des Displays auf dem Pi. Weiterhin muss der Rand des Fensters entfernt werden, da es später im Vollbildmodus laufen soll und der Rand nicht benötigt wird.

Arbeitet man mit oder ohne den Designer, muss man Änderungen am Aussehen von Elementen explizit über das StyleSheet tätigen. Diese sind sehr ähnlich der CSS Syntax. Beispiele hierfür findet man unter der StyleSheet-Dokumentation von Qt.

In der dritten Methode werden die unter Qt als Signal & Slots bezeichneten Verbindungen eingerichtet:

```
void HomeWindow::setupConnects(){

    connect(pButtonExit, SIGNAL(clicked(bool)), this, SLOT(slotShowExitWidget()));
    connect(pButtonAlert, SIGNAL(clicked(bool)), this , SLOT(slotShowAlertWidget()));
}
```

Signals & Slots sind eines der Schlüsselfunktionen von Qt. Diese sind ein Mechanismus von Qt, wie sich verschiedene GUI-Elemente oder Aktionen unterhalten können. Jemand sendet ein Signal aus und ein anderer empfängt dieses. Ein Signal kann z.B. beim Drücken eines Buttons ausgesendet werden. Ein oder mehrere Empfänger, die so genannten Slots, empfangen das Signal und rufen daraufhin eine entsprechende Funktion auf.

Konkret auf dieses Projekt angewandt, wird zunächst der Menübutton mit dem Event „wurde geklickt“ mit der Klasse HomeWindow verknüpft. Bei einem Klickevent wird ab nun der Slot slotShowExitWidget() ausgeführt.

```
void HomeWindow::slotShowExitWidget(){

    exitWidget = new ExitWidget(this);

    connect(exitWidget, SIGNAL(removeWindowformStack()), this, SLOT(removeActiveWidget()));
    addWidgetToMainStackWidget(exitWidget);
}
```

In diesem Slot wird zunächst eine neue Instanz vom Typ ExitWidget erstellt.

Anschließend wird eine weitere Verbindung erstellt. Diese vereinfacht gesagt, löst bei einem Signal removeWindowfromStack den Slot removeActiveWidget aus und entfernt das exitWidget aus dem QStackWidget. Hier sieht man den enormen Vorteil von Signals & Slots. Ein einfaches Signal aus dem neuen Widget informiert das Masterlayout dieses wieder zu entfernen.

```
emit removeWindowformStack();
```

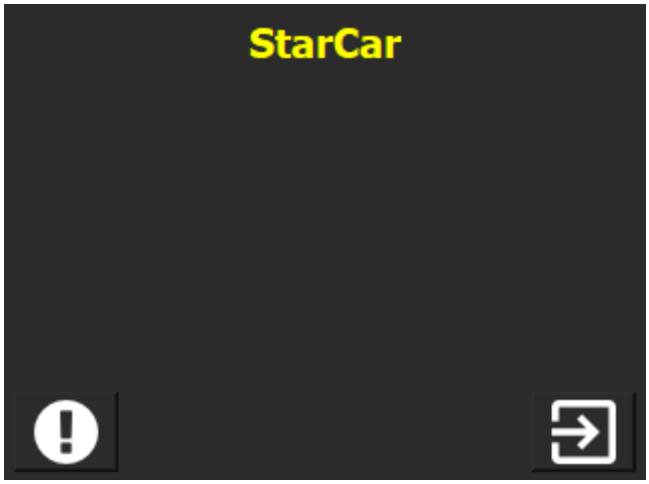
Dazu wird das Schlüsselwort emit verwendet. Auch hier sehr gut erkennbar, wie einfach ein Signal geschickt werden kann aus dem neuen Widget.

Schließlich wird das zuvor erzeugte ExitWidget in das QStackWidget eingefügt und angezeigt.

Äquivalent dazu ist die Vorgehensweise bei dem Button für die Fehleranzeige.

Wie sich im späteren Verlauf der Entwicklung zeigte, erwies sich das Konzept erst das Layout zu genieren, dies zu stylen und schließlich die Verbindungen einzurichten als robust und wird von allen Widgets verwendet.

Das Masterlayout nach der Implementierung:

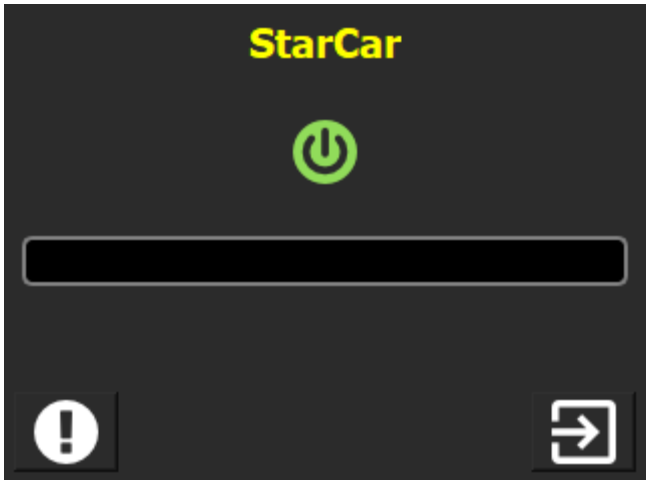


Zu sehen sind der Titel und danach ein freier Platz für die austauschbaren Widgets. Anschließend folgt in der linken unteren Ecke der Button für die Fehleranzeige. Gegenüber befindet sich der Button für das Menü. Auf die dazugehörigen Widgets wird später detaillierter eingegangen.

V. Startfenster

Wird die GUI gestartet wird zunächst wie eben beschreiben das Masterlayout generiert. Anschließend wird automatisch das erste Widget in das Feld der Austauschbaren Widgets geladen.

Das StartWidget:



Das StartWidget beinhaltet einen Button (grüner Button) um die Initialisierung des Fahrzeugs zu starten und eine Fortschrittsanzeige.

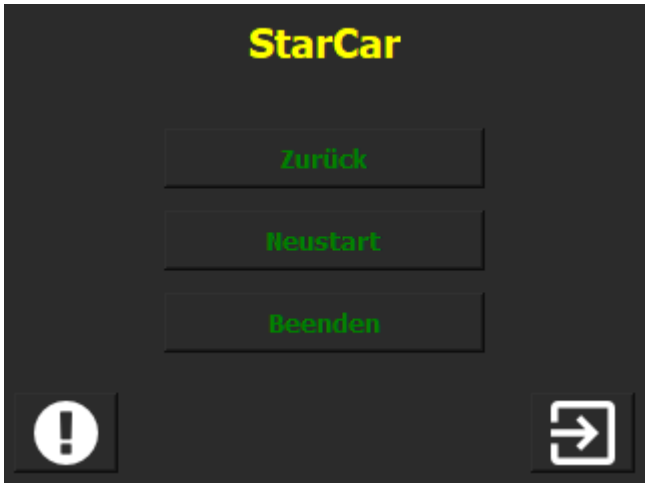
Drückt der Benutzer auf den grünen Button wird ein neuer Thread gestartet. Dieser übernimmt, im späteren Verlauf des Projekts nach der Integrierung des IBCs in die GUI, dessen Initialisierung und öffnet den Seriellen Port zum Arduino. Dieser Prozess muss in einem Thread ausgelagert werden, da sonst während der Initialisierung die GUI einfriert und keine Bedienmöglichkeiten durch den Benutzer möglich sind. Die Fortschrittsanzeige soll im späteren Verlauf in das IBC integriert werden und von dort gesteuert werden.

Über den Button unten links gelangt der Benutzer zur Fehleranzeige. Die Fehleranzeige wurde als Thread realisiert und blinkt bei einem Fehler Rot und bei einer Warnung Orange. Oder beides gleichzeitig.

Über den Button unten rechts gelangt der Benutzer in das Menü.

VI. Menü

Drückt der Benutzer auf den Button Menü, wird das aktuelle Widget nicht aus dem QStackedWidget entfernt, sondern lediglich das ExitWidget auf dem Stapel gelegt und als aktives Widget gesetzt. Dies hat den Vorteil, dass das Widget zuvor in seinem Zustand verbleibt und nicht neu initialisiert werden muss. Das Menü sieht folgendermaßen aus:



Es beinhaltet drei Buttons:

- Zurück
- Neustart
- Beenden

Wählt der Benutzer Zurück, wird das ExitWidget aus dem QStackedWidget entfernt und das zuvor aktive Widget ist wieder sichtbar.

Wählt der Benutzer Neustart, wird die GUI neugestartet. Dazu muss zunächst, im späteren Verlauf des Projekts nach der Integrierung des IBCs in die GUI, dass IBC, falls dieses zu diesem Zeitpunkt bereits initialisiert wurde, gelöscht werden, um den Seriellen Port zu schließen. Danach kann die GUI neugestartet werden. Dazu sind unter Qt lediglich zwei Anweisungen nötig:

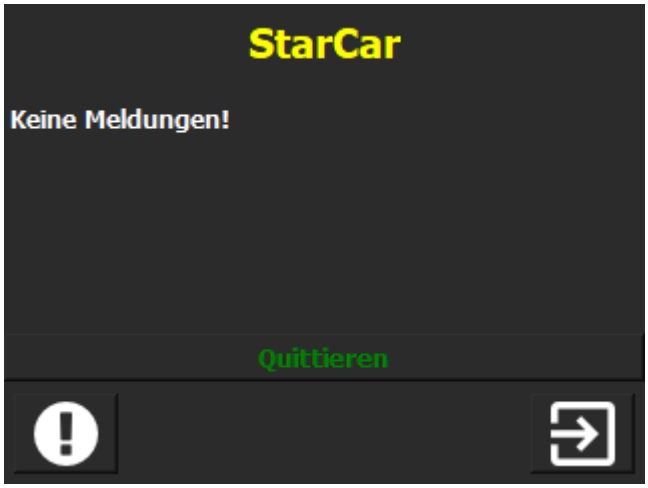
```
qApp->quit();
QProcess::startDetached(qApp->arguments()[0], qApp->arguments());
```

Wählt der Benutzer Beenden aus, wird die GUI beendet und anschließend der Pi heruntergefahren. Eine andere Möglichkeit, um den Pi ordnungsgemäß herunterzufahren, besteht nicht, da die GUI im Vollbildmodus ausgeführt wird und der Benutzer diese nicht verlassen kann. Unter Qt kann dies mit folgenden Kommando realisiert werden:

```
QProcess process;
process.startDetached("shutdown -P now");
```

VII. Fehleranzeige

Drückt der Benutzer auf den Button Fehleranzeige, wird das aktuelle Widget nicht aus dem QStackedWidget entfernt, sondern lediglich das AlertWidget auf dem Stapel gelegt und als aktives Widget gesetzt. Dies hat den Vorteil, dass das Widget zuvor in seinem Zustand verbleibt und nicht neu initialisiert werden muss. Die Fehleranzeige sieht, ohne dass ein Fehler oder eine Warnung aufgetreten ist, folgendermaßen aus:



Dabei ist der Aufbau recht schlicht gehalten. Dieser enthält eine QListView, um Meldungen darzustellen und einen Button zum Quittieren von Meldungen.

Das Quittieren der Meldungen wurde so implementiert, dass die Fehleranzeige aufhört zu blinken und in den neutralen Zustand zurückgeht. Meldungen aber nicht aus der QListView gelöscht werden.

Die QListView ist dabei auf das minimalste reduziert worden. Dies bedeutet, dass weder Ränder noch ein Scroll-Balken zu sehen sind. Jedoch sollten die Meldungen eine bestimmte Anzahl erreichen, erscheint dieser und ein Scrollen wäre prinzipiell möglich. Jedoch gestaltete sich das Scrollen als sehr schwierig auf dem Display.

Die Logik der Fehleranzeige, läuft in einem separaten Thread um wie bereits schon bei der Startseite beschrieben, ein Einfrieren der GUI zu vermeiden. Dieser wird automatisch während des Starts der GUI gestartet. Die Benutzung auf Seitens des Codes ist:

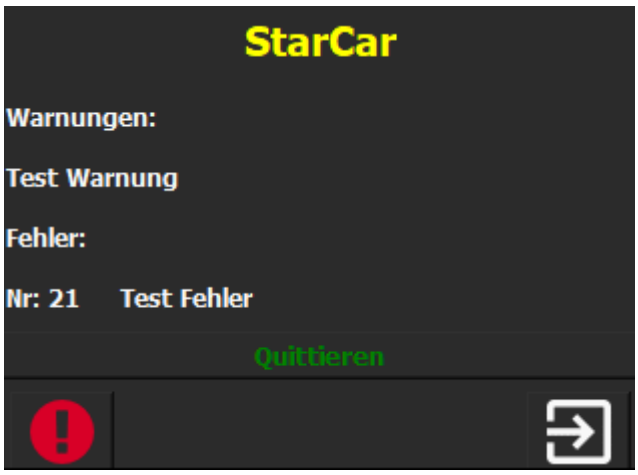
```
this->alertThread->fireWarning();
this->alertThread->fireWarning("Test Warnung");
this->alertThread->fireError("Test Fehler", 21);
```

Es ist an jeder Stelle im Code möglich, Fehlerbehandlungen auch dem Nutzer als Info während des Betriebs zur Verfügung zu stellen. Es gibt hierfür drei Möglichkeiten:

- Warnung / Fehler anzeigen
- Warnung / Fehler mit Beschreibung anzeigen
- Warnung / Fehler mit Beschreibung und Nummer anzeigen

Meldungen werden hierbei dauerhaft gespeichert. Das bedeutet, dass der Nutzer immer alle Meldungen aufgelistet bekommt.

Das bereits dargestellte Code-Beispiel generiert folgende Ausgabe in der GUI:



Zunächst muss darauf hingewiesen werden, dass während der Ausführung der GUI der Button links unten kontinuierlich die Farbe von Orange auf Rot und umgekehrt wechselt.

Es wurde eine Warnung ausgelöst mit der Beschreibung „Test Warnung“. Diese wird unter Warnungen aufgelistet. Analog dazu wurde ein Fehler ausgelöst mit der Beschreibung „Test Fehler“ und der Nummer „21“.

Ein konkretes Beispiel für unser Projekt wäre, der Lasersensor wird vergessen anzustecken. Sobald die GUI den Lasersensor ansprechen würde, würde der Button Rot blinken und nach einem Betätigen des Buttons würde die Fehlermeldung „Lidar not working“ ausgegeben. Gerade während der Entwicklung erwies sich die Fehleranzeige als sehr nützlich.

VIII. Operationsauswahl

Ist die Initialisierung erfolgreich beendet worden, wird das StartWidget aus dem QStackedWidget entfernt und durch das OperationModeWidget ersetzt.

Das OperationModeWidget verfügt über einen zusätzlichen Titel, um dem Benutzer die Auswahlmöglichkeiten etwas verständlicher darzustellen. Direkt darunter befinden sich drei Buttons. Diese sind:

- Uhrsteuerung
- Controllersteuerung
- Sensorwerte

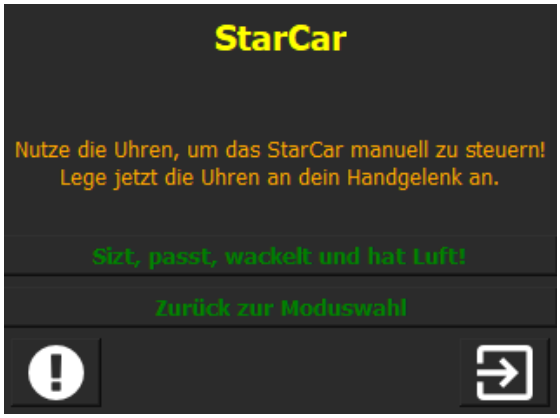
Dieses Widget ist rein für die Auswahl der Modi gedacht. Im Hintergrund werden keine weiteren Tätigkeiten durchgeführt. Der Benutzer muss sich jetzt für einen dieser Modi entscheiden.

Der Button Uhrsteuerung leitet den Benutzer zum ersten Widget der Uhrsteuerung um. Analog dazu der Button Controllersteuerung.

Der Button Sensorwerte leitet den Benutzer auf das Widget des Raumscans um.

IX. Uhrsteuerung

Hat der Benutzer im OperationModeWidget den Button Uhrsteuerung gedrückt, wird das OperationModeWidget entfernt und durch das ClockControlModeWidget ersetzt:



Dabei wird dem Benutzer zunächst ein Infotext angezeigt. Dieser ist während der Ausführung animiert und wechselt seine Größe. Dieser soll dem Benutzer mitteilen, dass er die Uhren anlegen soll und erst danach auf den ersten Button drücken soll. Der zweite Button ermöglicht dem Benutzer wieder zur Auswahl der Modi zu gelangen. Möchte der Benutzer jedoch die Uhrsteuerung starten, drückt dieser den ersten Button.

Im Hintergrund dazu wird, im späteren Verlauf des Projekts nach der Integrierung des IBCs in die GUI, über das IBC eine Mitteilung an den Arduino gesendet, dass der Benutzer die Uhren verwenden möchte und ein Modiwechsel stattfinden muss. Unabhängig von der Mitteilung an den Arduino wird das Widget diesmal nicht entfernt, stattdessen folgendermaßen umgebaut:

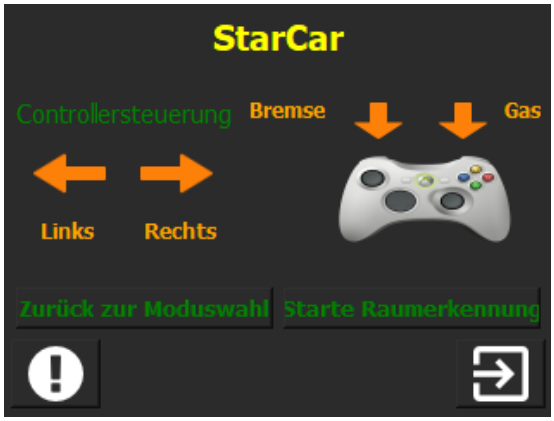
Zunächst wird der Benutzer über einen in der Farbe Grün gewählten Text darüber informiert, dass das Auto nun mit den Uhren steuerbar ist. Daraufhin wird dem Benutzer eine minimale Anleitung dargestellt, wie er mit den Uhren das Auto steuern kann. Leider fehlte der Platz für eine genauere Darstellung. Jedoch sind die Pfeile während der Ausführung animiert. Der Benutzer kann entweder zur Modiwahl zurückkehren oder den Raumscan starten. Eine Steuerung des Autos mit den Uhren ist nun möglich.

X. Controllersteuerung

Hat der Benutzer im OperationModeWidget den Button Controllersteuerung gedrückt, wird das OperationModeWidget entfernt und durch das ControllerControlModeWidget ersetzt:

Dabei wird dem Benutzer zunächst ein Infotext angezeigt. Dieser ist während der Ausführung animiert und wechselt seine Größe. Er soll dem Benutzer mitteilen, dass er den Controller in die Hand nehmen soll und erst danach auf den ersten Button drücken soll. Der zweite Button ermöglicht dem Benutzer wieder zur Auswahl der Modi zu gelangen. Möchte der Benutzer jedoch die Controllersteuerung starten, drückt dieser den ersten Button.

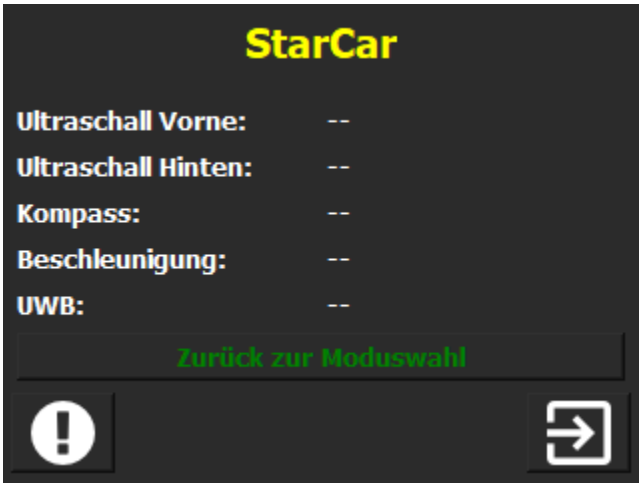
Im Hintergrund dazu wird, im späteren Verlauf des Projekts nach der Integrierung des IBCs in die GUI, über das IBC eine Mitteilung an den Arduino gesendet, dass der Benutzer den Controller verwenden möchte und ein Modiwechsel stattfinden muss. Unabhängig von der Mitteilung an den Arduino wird das Widget diesmal nicht entfernt, stattdessen folgendermaßen umgebaut:



Zunächst wird der Benutzer über einen in der Farbe Grün gewählten Text darüber informiert, dass das Auto nun mit dem Controller steuerbar ist. Daraufhin wird dem Benutzer eine minimale Anleitung dargestellt, wie er mit dem Controller das Auto steuern kann. Leider fehlte der Platz für eine genauere Darstellung. Jedoch sind die Pfeile während der Ausführung animiert. Der Benutzer kann entweder zur Modiwahl zurückkehren oder den Raumscan starten. Eine Steuerung des Autos mit dem Controller ist nun möglich.

XI. Sensorwerte / Raumkartographie

Hat der Benutzer entweder in der Operationsauswahl den Button Sensorwerte oder in der Uhrsteuerung / Controllersteuerung den Button „Starte Raumscan“ gedrückt, wird dem Benutzer das Widget SensorValuesWidget dargestellt:



Zunächst wird dem Benutzer wieder die Möglichkeit gegeben, in die Operationsauswahl zurückzukehren.

Im Hintergrund werden jetzt folgende Operationen kontinuierlich in folgender Reihenfolge ausgeführt:

- Starte die Messung des Lasersensors
- Sende eine Anfrage an den Arduino, alle Sensorwerte zu übermitteln
- Empfange die Sensordaten
- Zeige diese in der GUI an und schreibe diese in die entsprechenden Textdateien

2) Integration Lasersensor und IBC Protokoll

In diesem Kapitel wird der Verlauf der Integration des Lasersensors sowie dem Protokoll beschrieben und dabei entstandene Probleme erläutert.

I. Integration Lasersensor

Der Lasersensor wird direkt an dem Pi über USB angeschlossen. Ich erhielt die Implementierung von Anja Strobel. Diese wurde in C umgesetzt. Die Implementierung des Lasersensors verfügte bereits über eine Funktionalität, die gemessenen Daten in eine Textdatei zu schreiben.

Aus der Forderung des Teams zu Beginn des Projekts, dass alle Module auf dem Pi in C++ entwickelt werden, entwickeltet ich die GUI dementsprechend. Aus diesem Grund habe ich den Code auf C++ portiert und eine Klasse daraus geschrieben, um den Lasersensor sauber in die bereits verfügbare Logik meiner GUI einbinden zu können.

Die Messung wird in der GUI über einen eigenen Thread realisiert, da sonst auch hier ein einfrieren der GUI die Folge wäre. Der Thread wird bereits im Konstruktor des SensorValuesWidget initialisiert und gestartet.

II. Test Lasersensor

Für en ersten Test nach der Integration, wurde zusätzlich ein QTimer erstellt. Dieser wurde auf zehn Sekunden eingestellt. Nach einem Nulldurchgang wird dem Thread signalisiert, dass er keine weiteren Messungen mehr durchführen soll. Anschließend wird der Thread ordnungsgemäß beendet. Nach jeder Messung wurde der Thread für eine Sekunde schlafen gelegt und ein Testdurchgang führte somit zehn Messungen aus.

Bereits der erste Test verlief zufriedenstellend. Es wurden zehn Messungen durchgeführt und alle Messdaten wurden korrekt in die Textdatei geschrieben. Anschließende Test lieferten gleiche Ergebnisse. Der Lasersensor galt somit als erfolgreich integriert und getestet. Auch im späteren Verlauf zeigte dieser keine Auffälligkeiten und arbeitet zuverlässig.

III. Integration IBC Protokoll

Das Protokoll wurde in C++ und von Robert Graf entwickelt. Dieses enthält das Protokoll sowie den Seriellen Port. Der Serielle Port wurde von mir unabhängig vom Protokoll entwickelt und getestet. Genauere Details hierzu sind im Kapitel Serieller Port beschrieben.

Die Integration des Protokolls erfolgte recht spät im Projekt, obwohl dieses ein zentraler Bestandteil des Projekts war. Das Protokoll lies sich aufgrund einer sehr guten Kapselung sehr einfach in die GUI integrieren und bereitete keine Integrationsprobleme.

IV. Test IBC Protokoll

Das Protokoll bestand aus dem Protokoll selbst und einer minimalen Konsolenanwendung zur Veranschaulichung der Funktionsweise sowie Benutzung auf Seitens des Pis. Dabei ist zu erwähnen, dass die Beispielanwendung in einer emulierten Umgebung lief. Genauer gesagt, wurde der Arduino und die Serielle Schnittstelle emuliert und bis dato nicht auf den tatsächlichen Zielgeräten getestet. Ein Test der Beispielanwerbung meinerseits direkt auf dem Pi zeigte das gewünschte Ergebnis.

Anschließend wurde der erste Test mit der GUI und dem integrierten Protokoll auf dem Pi durchgeführt. Auf Seitens des Arduinos wurde das Protokoll ebenfalls integriert und die beiden Geräte über ein USB-Kabel verbunden.

Nach dem Betätigen des Buttons für die Initialisierung im StartWidget wurde der Serielle Port ordnungsgemäß geöffnet und das Protokoll fehlerfrei gestartet. Anschließend wurde über die Operationsauswahl „Uhrsteuerung“ gewählt. Nach dem Betätigen des ersten Buttons, legt die GUI ein Paket mit der ID 101 an. Anschließend sollte das Paket über das Protokoll an den Arduino übermittelt werden. Jedoch trat nun ein Speicherzugriffsfehler auf. Dies konnte auch bei der Controllersteuerung beobachtet werden. Es folgte eine intensive Untersuchung für die mögliche Ursache.

Während der Untersuchung, konnten wir die verursachende Stelle mittels Debugger feststellen. Der Speicherzugriffsfehler trat im Protokoll während dem Versuch einen Mutex zu sperren auf. Weitere lange Untersuchungen folgten bis die Ursache ermittelt werden konnte.

Die Ursache war im Nachhinein betrachtet recht simpel. Es lag nicht am Protokoll selber, sondern an der Implementierung innerhalb der GUI. Dort hatte ich die Referenz auf das Protokoll falsch übergeben. Interessant bei diesem Fall war es zu erkennen, dass der Fehler recht spät erst auftrat und die Suche aus diesem Grund von Anfang an in eine falsche Richtung ging.

Nachdem der Fehler behoben war, konnte das Protokoll fehlerfrei initialisiert werden und die Pakete von der GUI an den Arduino wurden korrekt übermittelt und verarbeitet am Arduino.

Es folgte der Test Sensordaten zu empfangen. Dazu legte ich in der GUI für jeden Sensor eine sogenannte Inbox und die dazugehörigen Pakete an. Das Messintervall wurde durch einen QTimer auf zwei Sekunden festgelegt. Alle zwei Sekunden schickt die GUI eine Anfrage an den Arduino die Sensordaten zu übermitteln. Anschließend werden die empfangenen Daten in die Inboxen geholt und überprüft ob in jeder Inbox Daten enthalten sind. Sind Daten enthalten werden diese in die Label der GUI geschrieben und anschließend in die Textdateien.

Es kamen allerdings zunächst keine Daten an. Es folgten viele weitere Tests, bis schließlich Daten ankamen. Jedoch waren diese nicht korrekt und die verfügbare Zeit bis zur Abgabe wurde immer weniger. Der Grund für die falschen Daten konnte bis heute nicht geklärt werden. Zusammengefasst konnte über das Protokoll zwar Daten an den Arduino übermittelt werden aber nicht zurück. Es war somit möglich, den Modus von der GUI aus zu wechseln.

3) Entwicklung Backup-Protokoll

Nach der Abschlusspräsentation am 12.01.2018 fasste Dominik Scharnagl, Simone Huber und ich den Entschluss eine mögliche Backuplösung bis zur Abschlussvorführung zu entwickeln, mit der Sensorwerte an die GUI übermittelt und dargestellt werden können. Weiterhin wäre dadurch die Raumkartographie vorführbar.

Wir orientierten uns dazu an einem verfügbaren Projekt auf GitHub, dieses Implementiert ein einfaches serielles Protokoll für Arduino und Pi. Angelehnt an diesem Projekt, bauten wir mit der bereits vorhandenen Implementierung des Seriellen Ports, Schritt für Schritt ein neues Protokoll. Die Grundidee hierfür war nicht das einzelne Übermitteln der Sensorwerte, sondern alle Daten vor dem Senden in eine Struktur zusammenzufassen und die gesamte Struktur zu übermitteln.

Dabei konnten wir recht schnell Erfolge mit kleinen Strukturen erreichen. Jedoch ab einer gewissen Größe der Struktur, kam diese nicht mehr korrekt in der GUI an. Eine intensive Untersuchung der verwendeten Datentypen zeigte, dass auf beiden Seiten die gleichen Datentypen nicht die gleiche Größe hatten. Aus diesem Grund wurden die Daten in der GUI auch falsch ausgewertet. Nach einer Anpassung der Datentypen auf beiden Seiten, wurden die Daten dennoch nicht richtig in der GUI ausgewertet. Eine weiter Untersuchung der Größe in Bytes der Struktur auf beiden Seiten lieferte eine unterschiedliche Größe. In der GUI werden ohne das explizite setzen eines Alignments für Strukturen unter gewissen Umständen, sogenannte Schattenfelder hinzugefügt. Diese bewirken, dass die empfangene Struktur zwar korrekt umkopiert werden kann, aber Bytes durch die Schattenfelder verschoben werden. Dies war schließlich auch der Grund, warum kleine Strukturen korrekt umkopiert worden sind, da kein Schattenfeld vorhanden war.

Um Übertragungsfehler zu erkennen, haben wir ein weiteres Feld in die Struktur eingeführt. Der Wert wird durch verodern aller Daten ermittelt. In der GUI wird nach dem umkopieren der Struktur mit der gleichen Methode überprüft, ob die empfangenen Daten richtig sind. So wird verhindert, dass falsch übermittelte Daten zu einem dargestellt und zum anderen in die Textdateien für die Raumkartographie geschrieben werden.

In weiteren Tests zeigte sich allerdings, dass die GUI einfrieret, wenn das Protokoll keine oder zu wenige Daten empfängt. Grund hierfür ist der Verzicht eines Headers sowie einem blockierenden Lesen auf Seiten des Seriellen Ports. Dem entgegenzuwirken, müsste das Protokoll in einem Thread ausgelagert werden. Leider konnte das nicht mehr umgesetzt werden bis zur Abschlussvorführung.

Aufgrund der verhältnismäßig großen zu sendenden Datenmenge, musste das Abrufen der Daten auf ein Intervall von einer Sekunde reduziert werden. Bei einem Intervall von einer halben Sekunde war das präzise Steuern des Fahrzeugs nicht mehr möglich, da das Senden die Motorsteuerung blockierte. Aus diesem Grund musste auch das Intervall einer Messung des Lasersensors auf eine Sekunde reduziert werden.

6. Test

Im Folgenden werden die Abschlusstests ausgehend der definierten Anforderungen mit der aktuellen Implementierung der GUI beschreiben.

1) /T0101/ **Automatischer Start der Benutzeroberfläche:**

   Verbindet der Benutzer das Fahrzeug mit dem Akku, fährt der Pi ordnungsgemäß hoch. Anschließend wird automatisch die GUI im Vollbildmodus gestartet. Der Benutzer kann nur die zulässigen Kontrollelemente in der GUI bedienen und dadurch eine falsche Bedienung des Pis verhindert.

2) /T0102/ **Initialisierung des Fahrzeugs:**

   Nach dem automatischen Start der GUI wird das StartWidget fehlerfrei geladen und dem Benutzer dargestellt.

   Dem Benutzer stehen das Menü, die Fehleranzeige und das Initialisieren des Fahrzeugs zur Verfügung. Drückt der Benutzer auf den grünen Button, um das Fahrzeug zu initialisieren, wird im Hintergrund aufgrund des Wechsels auf das Backup-Protokoll, dieses fehlerfrei initialisiert und der Serielle Port wird fehlerfrei konfiguriert und geöffnet. Weitere Steuermöglichkeiten stehen dem Benutzer nicht zur Verfügung.

3) /T0103/ **Moduswahl:**

   Dem Benutzer wird nach der erfolgreichen Initialisierung des Fahrzeugs das OperationModeWidget fehlerfrei dargestellt. Dieses ermöglicht dem Benutzer eine Auswahl der Modi „Uhrsteuerung “ und „Controllersteuerung“. Weiterhin kann der Benutzer den Raumscan starten. Alle drei Buttons stellen nach dem betätigen, dem Benutzer das dazugehörige Widget fehlerfrei dar.

4) /T0104/ **Neustart der Benutzeroberfläche:**

   Der Benutzer gelangt über den Button „Menü“ in das Menü und kann dort über den Button „Neustart“ die GUI neustarten. Der Neustart funktioniert fehlerfrei.

5) /T0105/ **Beenden des Systems:**

   Der Benutzer gelangt über den Button „Menü“ in das Menü und kann dort über den Button „Beenden“ den Pi ordnungsgemäß herunterfahren. Das herunterfahren funktioniert fehlerfrei.

6) /T0106/ **Uhrsteuerung:**

   Wählt der Benutzer im OperationModeWidget den Modus „Uhrsteuerung“ wird das Widget ClockControlModeWidget fehlerfrei geladen. Der enthaltene Infotext ist vorhanden und wird fehlerfrei animiert. Weiterhin informiert dieser den Benutzer über das Anlegen der Uhren. Der Benutzer kann den Infotext über einen Button bestätigen. Wurde die Bestätigung ausgelöst, wird über das Backup-Protokoll eine Nachricht über den Moduswechsel fehlerfrei an den Arduino übermittelt. Im Anschluss wird das Widget fehlerfrei umgebaut. Animierte Bilder zur Verdeutlichung der Steuerung sind vorhanden und werden fehlerfrei angezeigt. Der Button zum Starten der Raumkartographie ist vorhanden und stellt nach dem betätigen, dem Benutzer fehlerfrei das SensorValuesWidget dar.

7) /T0107/ **Controllersteuerung:**

   Wählt der Benutzer im OperationModeWidget den Modus „Controllersteuerung“ wird das Widget ControllerControlModeWidget fehlerfrei geladen. Der enthaltene Infotext ist vorhanden und wird fehlerfrei animiert.

   Weiterhin informiert dieser den Benutzer über bereithalten des Controllers. Der Benutzer kann den Infotext über einen Button bestätigen. Wurde die Bestätigung ausgelöst, wird über das Backup-Protokoll eine Nachricht über den Moduswechsel fehlerfrei an den Arduino übermittelt. Im Anschluss wird das Widget fehlerfrei umgebaut. Animierte Bilder zur Verdeutlichung der Steuerung sind vorhanden und werden fehlerfrei angezeigt. Der Button zum Starten der Raumkartographie ist vorhanden und stellt nach dem betätigen, dem Benutzer fehlerfrei das SensorValuesWidget dar.

8) /T0108/ **Navigation:**

   Der Benutzer hat nach der Initialisierung des Fahrzeugs, jederzeit die Möglichkeit zurück zur Moduswahl zu gelangen. Die Navigationslogik der GUI arbeitet fehlerfrei.

9) /T0109/ **Darstellung der Sensorwerte:**

   Es sind für die Ausgabe der Daten von den Sensoren alle Felder vorhanden. Die Sensorwerte werden über das Backup-Protokoll empfangen und in der GUI fehlerfrei dargestellt.

10) /T0110/ **Fehleranzeige:**

    Das Signalisieren eines Fehlers oder einer Warnung und der Button, mit dem der Benutzer zur Fehleranzeige gelangt, wurde in einem Button zusammengefasst. Dieser wird bei einem Auftreten eines Fehlers oder und zugleich einer Warnung fehlerfrei animiert. Zudem wurde der Button in das Masterlayout integriert und erfüllt somit die Forderung, jederzeit gut sichtbar in der GUI zu sein. Drückt der Benutzer auf den Button, wird das AlertWidget fehlerfrei geladen. Gleiches gilt für das Auflisten der Meldungen. Der Button zum Quittieren von Meldungen arbeitet ebenfalls fehlerfrei.

11) /T0111/ **Raumkartographie:**

    Wählt der Benutzer den Button „Starte Raumscan“ wird das SensorValuesWidget fehlerfrei geladen. Dieses startet die Messung des Lasersensors und empfängt die Sensordaten. Beides wird fehlerfrei ausgeführt (Fehler während der Übertragung nicht eingeschlossen). Die Darstellung der Sensorwerte und das Wegschreiben der Daten in Textdateien wird ebenfalls fehlerfrei durchgeführt.

7. Ausblick

Die Entwicklung der GUI sowie deren Implementierung konnte ich für mich zufriedenstellen durchführen. Die Statusanzeige in dem StartWidget konnte ich wegen dem Wechsel auf das Backup-Protokoll und daraus resultierenden Zeitgründen nicht mehr integrieren und ist demnach aktuell noch ohne Funktionalität.

Die Animationen der Controller- sowie Uhrensteuerung erfüllen Ihren Zweck, sind aber noch verbesserungsfähig.

Weiterhin benötigt das Backup-Protokoll noch weitere Verbesserungen. Zu einem müsste dieses in einen eigenen Thread ausgelagert werden. Gerade bei einem Verbindungabbruch zwischen Arduino und Pi während der Raumkartographie löst das ein einfrieren der GUI aus und der Pi kann nur über das Trennen der Stromversorgung wieder verwendbar gemacht werden.

Zusammengefasst hat mir das Fach extrem viel Spaß gemacht. In dem Gebiet der Benutzeroberflächen und deren Implementierung war ich schon immer interessiert und konnte nun im Rahmen von diesem Projekt meine eigenen Ideen und Vorstellungen frei umsetzten.

Serieller Port – Florian Boemmel

1. Generelles

In unserem Projekt nutzen wir eine serielle USB-Verbindung zwischen Arduino und Raspberry Pi, um Daten und Befehle zwischen den beiden Geräten auszutauschen. Dieser Abschnitt beschäftigt sich ausschließlich nur mit dem Seriellen Port für die USB-Verbindung zwischen Raspberry Pi und Arduino.

2. Grundlagen

Die Grundlage jeder seriellen Kommunikation auf einem linuxbasiertem Betriebssystem ist das Öffnen und Konfigurieren eines Seriellen Ports. Serielle Ports werden unter Linux durch eine Datei repräsentiert.

1) Seriellen Port bestimmen

Zunächst muss der Port festgestellt werden, an dem der Arduino am Pi erkannt wird. Dazu kann entweder die Arduino IDE benutzt werden, oder das Terminal.

1. Möchte man das Terminal nutzen, muss die Verbindung zum Arduino unbedingt getrennt werden und folgendes Kommando ausgeführt werden:

   ```
   pi@raspberrypi:~$ ls /dev/
   ```

   Nun muss zunächst überprüft werden, ob bereits ein ttyUSB oder ttyACM existiert. Jetzt muss der Arduino verbunden werden. Eine erneute Ausführung des Kommandos sollte jetzt einen weiteren Eintrag liefern (z.B. ttyUSB0). Dieser Eintrag ist nun der Serielle Port zu unserm Arduino.

2. Möchte man die Arduino IDE benutzen, öffnet man diese und verbindet den Arduino mit dem Pi. Anschließend wählt man im Menü:

   ```
   Tools → Serieller Port
   ```

   Hier wird nun der Port angezeigt. Jedoch muss beachtet werden, dass weitere angeschlossene Geräte unter Umständen auch angezeigt werden.

3. Implementierung

Das Implementieren des Seriellen Ports erfolgt mit C und unter der Verwendung der terminos API. Die terminos API unterstützt unterschiedliche Modi um einen Seriellen Port anzusprechen. Die zwei wichtigsten sind:

- **Cannonical Mode:** Dieser Modus ist Zeilenorientiert. Dies bedeutet, dass Eingaben gepuffert und durch den Benutzer bearbeitet werden können, bis ein carriage return (unter Linux CTRL-C) oder ein line feed (Zeilenumbruch) erkannt wird. Anschließend kann ein read(2) ausgeführt werden. Wird von Terminals verwendet.

- **NonCannonical Mode:** Dieser Modus ist im Gegensatz zum Cannonical Mode weder Zeilenorientiert noch werden Eingaben gepuffert oder können vom Benutzer bearbeitet werden. Dies bedeutet, dass ein Input sofort zur Verfügung steht. Zusätzlich muss hier eine Einstellung vorgenommen werden, unter welchen Umständen ein read(2) aufgerufen wird und wie sich dieses verhält.

Ausführliche Informationen über die Seriellen Ports und deren Programmierung können im **"The Serial Programming Guide for POSIX Operating Systems"** nachgelesen werden.

1) Öffnen und Schließen

Zum Öffnen eines Seriellen Ports unter Linux wird der Systemaufruf open(2) verwendet:

```
int fd;
fd = open("/dev/ttyUSB0", O_RDWR | O_NOCTTY);
```

fd: File-Deskriptor
/dev/ttyUSB0: Serieller Port im Verzeichnis /dev
O_RDWR: Serieller Port wird geöffnet für schreiben und lesen
O_NOCTTY: Kein Terminal wird das öffnen kontrollieren

Wurde der Port erfolgreich geöffnet, erhält fd einen positiven Wert. Im Fehlerfall liefert open -1 zurück.

Zum schließen wird close(2) verwendet:

```
close(fd);
```

2) Konfigurieren

Zum Konfigurieren des Seriellen Ports wird, wie schon beschrieben, die terminos API benutzt. Die terminos Struktur sieht wie folgt aus:

```
struct termios
  {
    tcflag_t c_iflag;           /* input mode flags */
    tcflag_t c_oflag;           /* output mode flags */
    tcflag_t c_cflag;           /* control mode flags */
    tcflag_t c_lflag;           /* local mode flags */
    cc_t c_line;                        /* line discipline */
    cc_t c_cc[NCCS];            /* control characters */
    speed_t c_ispeed;           /* input speed */
    speed_t c_ospeed;           /* output speed */
#define _HAVE_STRUCT_TERMIOS_C_ISPEED 1
#define _HAVE_STRUCT_TERMIOS_C_OSPEED 1
  };
```

Nun werden die spezifischen Einstellungen für unser Projekt gesetzt.

```
struct termios SerialPortSettings;

tcgetattr(fd, &SerialPortSettings);             // Get the current attributes of the Serial port

cfsetispeed(&SerialPortSettings,B115200);        // Set Read Speed as 115200
cfsetospeed(&SerialPortSettings,B115200);        // Set Write Speed as 115200

SerialPortSettings.c_cflag &= ~PARENB;          // Disables the Parity Enable bit(PARENB),So No Parity
SerialPortSettings.c_cflag &= ~CSTOPB;          // CSTOPB = 2 Stop bits, here it is cleared so 1 Stop bit
SerialPortSettings.c_cflag &= ~CSIZE;           // Clears the mask for setting the data size ... enables set own data bits... see next line
SerialPortSettings.c_cflag |=  CS8;             // Set the data bits = 8
SerialPortSettings.c_cflag |= (CREAD | CLOCAL); // Enable receiver,Ignore Modem Control lines

cfmakeraw(&SerialPortSettings);                 // Setup Raw-Mode automatically
```

Für weitere Informationen und einer detaillierten Beschreibung der verwendeten sowie möglichen weiteren Einstellungen kann das unter Punkt 4 referenzierte Dokument verwendet werden.

Ein letzter Schritt setzt die Einstellungen in der terminos Struktur zu dem Seriellen Port:

```
tcsetattr(fd,TCSANOW,&SerialPortSettings));
```

Die Funktion liefert im Erfolgsfall eine 0 zurück. Danach ist der Serielle Port konfiguriert und für die Übertragung und das Empfangen von Daten eingerichtet.

3) Daten Schreiben

Das Schreiben auf dem Seriellen Port, wird durch den Systemaufruf write(2) realisiert.

```
write(fd, data, size);    // use write() to send data to port
                          // "fd"                  - file descriptor pointing to the opened serial port
                          // "write_buffer"        - address of the buffer containing data
                          // "sizeof(write_buffer)" - No of bytes to write
                          // returns the actual bytes written to the port
```

Dabei wird write() der File-Deskriptor, einem Puffer vom Typ const void * und eine Größe der zu schreibenden Daten in Bytes übergeben.
Die Größe der zu schreibenden Daten in Bytes gibt an, wie viele Bytes auf dem Puffer geschrieben werden sollen.

write() liefert im Erfolgsfall die Anzahl der geschriebenen Bytes zurück. Im Fehlerfall wird -1 zurückgegeben und 0 bedeutet, dass keine Daten geschrieben wurden.

4) Daten Lesen

Das Lesen auf dem Seriellen Port, wird durch den Systemaufruf read(2) realisiert.

```
read(fd, data , maxsize);        // Read the data
                                     // Write it to the location at <data>
                                     // Write <maxsize> bytes at maximum to not overrun buffersizes
```

Dabei wird read() der File-Deskriptor, einem Puffer vom Typ void * und eine Größe der zu lesenden Daten in Bytes übergeben. Die Größe der zu lesenden Daten in Bytes gibt an, wie viele Bytes aus dem File-Deskriptor in den übergebenen Puffer gelesen und anschließend geschrieben werden sollen.

read() liefert im Erfolgsfall die Anzahl der gelesenen Bytes zurück. Im Fehlerfall wird -1 zurückgegeben und 0 bedeutet, das Ende der Datei ist erreicht.

4. Probleme

1) Wechsel des Seriellen Ports

Zu Beginn des Projekts verwendeten wir einen Arduino Uno. Im späteren Verlauf wechselten wir jedoch auf einen Arduino Mega. Während der Debug-Tätigkeiten fiel immer wieder das Problem mit dem Seriellen Port auf. Manchmal wurde der Arduino unter ttyUSB0 erkannt und ein paar Tage später wieder unter ttyACM0 oder andersherum.

Schlussendlich stellte sich heraus, dass der Grund hierfür der Wechsel der Arduinos war. Einmal wurde mit dem Uno gearbeitet und ein anderes Mal mit dem Mega. Der Uno bekommt durch den Raspberry Pi den Port ttyUSB0 zugewiesen. Im Gegensatz dazu bekommt der Mega den Port ttyACM0 zugewiesen. Mögliche Gründe hierfür konnte ich noch nicht ermitteln.

2) Öffnen des Seriellen Ports

Zu Beginn der Implementierung des Seriellen Ports, kam es sporadisch vor, dass der Port zwar ordnungsgemäß geöffnet wurde, jedoch anschließend nicht fehlerfrei arbeitete. Nach intensiver Recherche wurde ich fündig. In einigen Fällen, kann es vorkommen, dass das Programm zu schnell weiterarbeitet und z.B. ein write(2) zu früh ausführt. Ein usleep(3) von 200 Millisekunden direkt nach dem Öffnen löste das Problem dauerhaft.

3) Neustart des Arduino

Während der ersten Tests viel auf, dass keine Daten an den Arduino gesendet werden konnten. Ich untersuchte dies ausgiebig und stellte anschließend fest, dass nach dem unter Punkt 6 beschriebenen Tätigkeiten der Arduino neustartet. Genauer gesagt geschieht dies direkt nach dem Funktionsaufruf:

```
tcsetattr(fd,TCSANOW,&SerialPortSettings));
```

Eine Lösung hierfür ist denkbar einfach und mehrfach in den Foren als einzige Lösung bekannt. Man muss auf den Neustart des Arduinos warten. Eine sichere Zeitspanne ist dabei drei Sekunden. Realisierbar unter Linux mit sleep(3).

4) Schließen des Seriellen Ports

Es ist unabdingbar den Seriellen Port am Ende des Programms wieder zu schließen, um Ihn anschließend beim erneuten Starten des Programms wieder öffnen zu können. Das bereitete gerade während der Entwicklungsphase der Benutzeroberfläche einige Probleme. Stürzte die Benutzeroberfläche während eines Testlauf ab, konnte der Serielle Port nicht mehr geöffnet werden und es blieb nichts anderes übrig, als jedes Mal die USB-Verbindung zum Arduino zu trennen und wiederherzustellen.

5. Ausblick

Die geforderten Anforderungen wurden umgesetzt und der Serielle Port erweist sich als robust und bereitete keine Probleme. Dies wurde durch ausgiebige Tests bestätigt.

Jedoch wäre eine genauere Untersuchung der gesamten terminos Struktur ein weiterer möglicher Schritt. In diesem könnten eventuelle Verbesserungen in dem Gebiet der Performance erreicht werden.

Zusätzlich könnte noch eine Logik eingebaut werden, die die vorhandenen Ports überprüft und so selber den richtigen für den Arduino auswählt. Aktuell ist dieser im Code fest implementiert und muss bei einem Wechsel des Ports geändert und neu kompiliert werden.