



StarCar

Ein Projekt im Rahmen der Veranstaltung
Datenverarbeitung in der Technik

von

Annkathrin Bauer, Mehmet Billor, Florian Boemmel, Robert Graf, Simoe
Huber, Dominik Scharnagl, Anja Strobel
(Gruppe 2)



1. Inhaltsverzeichnis

1. Inhaltsverzeichnis	2
2. Überblick.....	7
3. Das Team	8
4. Dokumentation.....	9
5. Projektentstehung	10
6. Projektplanung und Projektauftrag	11
7. Plakaterstellung	12
8. Software.....	14
8.1. Programmiersprachen	14
8.2. Programme	14
9. Hardware	15
9.1. Firmware des Arduino.....	15
9.1.1. Anforderungen	15
9.1.2. Analyse / Design.....	15
9.1.3. Implementierung.....	16
9.1.4. Test	19
9.2. Aufsetzen des Raspberry Pi 3.....	19
9.3. Installation des Displays.....	20
9.4. „Sabotage“ / Verschwinden von Teilen	21
10. Sensoren / Aktuatoren	22
10.1. Motor	22
10.1.1. Anforderungen	22
10.1.2. Analyse / Design.....	22
10.1.3. Implementierung.....	23
10.1.4. Test	25
10.2. Servo	26
10.2.1. Anforderungen	26
10.2.2. Analyse / Design.....	26
10.2.3. Implementierung.....	27
10.2.4. Test	31
10.3. Ultraschallsensor.....	31
10.3.1. Funktionalität	31
10.3.2. Inbetriebnahme & Programmierung.....	32



10.3.3. Stopp-Bedingung	32
10.3.4. Ausblick	33
10.4. Kompass Sensor	34
10.4.1. Überblick	34
10.4.2. Verkabelung	34
10.4.3. Implementierung und Fehlersuche	34
10.5. Beschleunigungssensor.....	37
10.5.1. Überblick	37
10.5.2. Verkabelung	38
10.5.3. Implementierung und Fehlersuche	38
10.6. Sensoren – Übertragung	39
10.6.1. Übertragungsansatz	39
10.6.2. Überprüfung der Sensorfunktionalität.....	39
10.6.3. Erstellung des Bitframes	40
10.6.4. Bildung einer Struktur	41
10.7. UWB	41
10.7.1. Anforderungen	41
10.7.2. Berechnung und Funktionalität.....	41
10.7.3. Grundeinstellungen der DW1000-Module	42
10.7.4. Inbetriebnahme und Probleme	43
10.7.4. Fazit	45
10.8. LIDAR	45
10.8.1. Spezifikationen	45
10.8.2. Inbetriebnahme.....	46
10.8.3. Verwendung	46
10.8.4. Sourcecode	46
10.8.5. Beobachtungen	48
11. Steuerung	49
11.1. Steuerung mittels Xbox 360 USB Controller	49
11.1.1. Anforderungen	49
11.1.2. Analyse / Design.....	50
11.1.3. Implementierung.....	51
11.1.4. Test	53
11.2. Steuerung mittels eZ430-Chronos-Watch	53
11.2.1. Anforderungen	54
11.2.2. Analyse / Design.....	58



11.2.3. Implementierung.....	63
11.2.4. Test	70
12. Serieller Port – Raspberry Pi 3 Serieller Port.....	72
12.1. Generelles	72
12.2. Grundlagen.....	72
12.2.1. Seriellen Port bestimmen.....	73
12.3. Implementierung	73
12.3.1. Öffnen und Schließen.....	74
12.3.2. Konfigurieren.....	74
12.3.3. Daten Schreiben	75
12.3.4. Daten Lesen.....	76
12.4. Probleme	76
12.4.1. Wechsel des seriellen Ports	76
12.4.2. Öffnen des seriellen Ports	76
12.4.3. Neustart von Arduino	77
12.4.4. Schließen des seriellen Ports.....	77
12.5. Ausblick	77
13. Kommunikationsprotokoll	78
13.1. Versionsübersicht	78
13.2. Anforderungs- und Konzeptliste	79
13.3. Entwicklungsvorgang	82
13.3.1. [V0.0] Erste Versuche	82
13.3.2. [V0.1] Kontrollstrukturen Pi	83
13.3.3. [V0.2] Protokollablauf	87
13.3.4. [V0.3] Konzeptionelle Fehler im Protokollablauf	89
13.3.5. [V0.4] Kontrollstrukturen Arduino	95
13.3.6. [V1.0] Integrierung und Fehleranalyse.....	98
13.3.7. [V1.1] Fehlerbehebung	99
13.3.8. [V1.2] Richtigstellung Protokollablauf	100
13.3.9. [V1.3] Finales Debugging	102
13.4. Fazit	102
14. Grafische Benutzeroberfläche	104
14.1. Generelles	104
14.2. Entwicklungsumgebung	104
14.3. Installation & Einrichtung von QtCreator	105
14.3.1. Installation unter Windows	105



14.3.2. Installation unter Linux (Raspberry Pi).....	107
14.3.3. Einrichten	107
14.3.4. Cross-Kompilierung vs. „Copy And Paste“	108
14.4. Anforderungen.....	108
14.5. Umsetzung	110
14.5.1. Implementierung der GUI	110
14.5.2. Integration Lasersensor und IBC Protokoll	125
14.5.3. Entwicklung Backup-Protokoll	127
14.6. Test.....	128
14.7. Ausblick	131
15. Raumdarstellung.....	132
15.1. Voraussetzungen.....	132
15.2. Datenübertragung.....	132
15.3. Verarbeitung der Daten in Matlab.....	133
15.3.1. Kompass	134
15.3.2. Ultraschall.....	134
15.3.3. LIDAR	134
15.3.4. Beschleunigungssensoren	135
15.3.5. Ergebnis	135
15.4. Probleme und Ausblick	136
16. Zusammenbau	137
16.1. Montage Fahrzeug - Grundgerüst.....	137
16.2. Montage der einzelnen Sensoren / Steuerungen / Boards	137
16.3. Spannungsversorgung.....	138
16.4. Übersicht verwendeter Teile	139
17. Verkabelung.....	140
17.1. Verkabelung Daten	140
17.2. Verkabelung Gesamtübersicht	141
18. Ausblick	143
19. Stundenzettel.....	144
19.1. Annkathrin Bauer.....	144
19.2. Mehmet Billor	145
19.3. Florian Boemmel	146
19.4. Robert Graf.....	147
19.5. Simone Huber.....	148
19.6. Dominik Scharnagl.....	149



19.7. Anja Strobel	150
20. Abschließende Eindrücke	151
21. Abbildungsverzeichnis	152
22. Literaturverzeichnis	153



2. Überblick

Dieser Bericht soll einen Überblick der getätigten Arbeiten im Rahmen der Veranstaltung „Datenverarbeitung in der Technik“ für die Gruppe StarCar darstellen. Das Fach wurde von Herrn Professor Dr. Richard Roth und Herrn Matthias Altmann betreut.

Das Endergebnis des Projektes war ein Fahrzeug, welches eigenständig zusammengebaut wurde. Es wurden verschiedenste Sensoren verwendet, um Abstände, Orientierung und Hindernisse zu erkennen. Das Fahrzeug kann über zwei verschiedene Auswahloptionen gesteuert werden. Einerseits mit einem Xbox-Controller andererseits mit zwei Smart-Watches, mit welchen eine Gestensteuerung entwickelt worden ist.

Des Weiteren ist auf dem Fahrzeug ein Display verbaut worden, auf welchem eine Benutzeroberfläche dargestellt wird. Diese ermöglicht dem Benutzer die Auswahl des gewünschten Fahrmodus, sowie die Darstellung der erfassten Sensordaten und veranlasst die Übertragung der Sensorwerte an einen PC, auf welchem eine Raumerkennung mittels MATLAB visualisiert werden kann.

Im Folgenden wird auf die einzelnen Bereiche detailliert eingegangen und aufgezeigt wer für die Durchführung des Teilbereiches verantwortlich war.



3. Das Team

Die Projektgruppe besteht aus 7 Studenten des Studiengangs Technische Informatik an der Ostbayerischen technischen Hochschule Regensburg. Die Studenten befinden sich alle im 6. oder 7. Semester.

Der Projektzeitraum war vom 6. Oktober 2017 bis zum 19. Januar 2018. Bis zum 2. Februar 2018 war noch Zeit, diese Dokumentation zu erstellen.

An dem Projekt waren beteiligt:

- Anja Strobel
- Annkathrin Bauer
- Dominik Scharnagl
- Florian Boemmel
- Mehmet Billor
- Robert Graf
- Simone Huber



4. Dokumentation

Die Dokumentation wurde am Ende des Projektes erstellt, wobei im Projektzeitraum bereits jedes Mitglied seine Tätigkeiten dokumentierte und auftretende Probleme und Umsetzungsstrategien notierte, um bei der Erstellung dieser Dokumentation auf diese Notizen zurückgreifen zu können.

Die einzelnen Teilbereiche der Dokumentation wurden von den einzelnen Teammitgliedern selbst erstellt und anschließend zusammengeführt. Um aufzuzeigen, welches Teammitglied ein Kapitel verfasst hat, wird zu Beginn jedes größeren Kapitels der Verfasser aufgeführt. Teamübergreifende Kapitel wurden von Simone Huber verfasst, welche auch die Zusammenführung der einzelnen Komponente durchführte. Sowie auch die Zeitpläne für die Dokumentation erstellte, hierzu stellten alle Teammitglieder ihre Zeitpläne im GitHub-Projekt zur Verfügung und wurden dann in ein einheitliches Schema für die Auflistung in der Dokumentation gebracht.

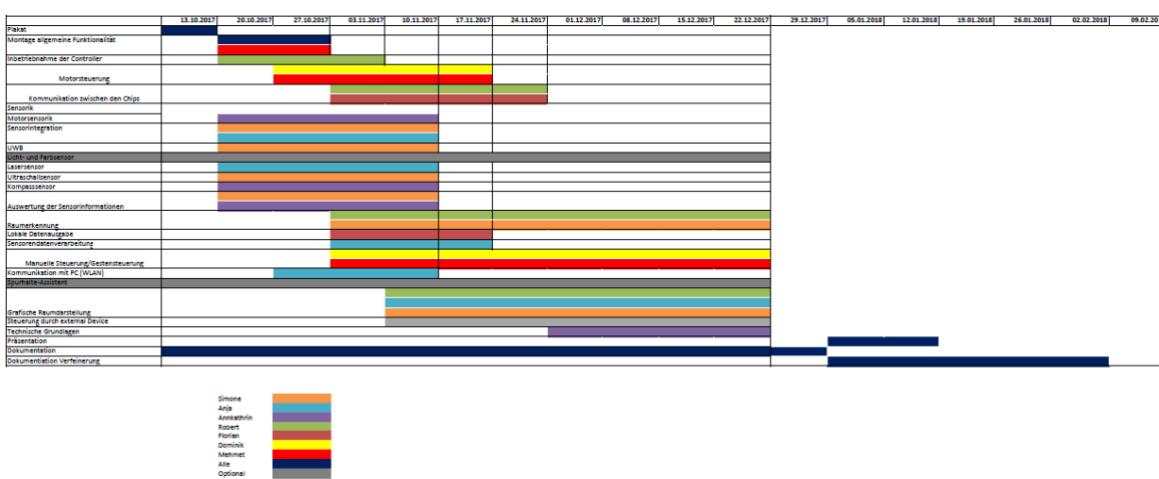
Nach der Zusammenstellung der Dokumentation wurde, diese von allen Teammitgliedern nochmals überarbeitet und etwaige Fehler behoben.



5. Projektentstehung

Der erste und zugleich entscheidende Punkt war es sich für eine Projektidee zu entscheiden. Schnell war das Team sich einig mit einem der angebotenen Fahrzeuge zu arbeiten und ein eigenes fahrendes Auto zu entwickeln.

Nun war es wichtig die einzelnen Bestandteile des Fahrzeuges festzulegen. Im gemeinsamen Austausch wurde sich dann darauf geeinigt, dass das Fahrzeug im Idealfall über zwei Steuerungsoptionen fahren kann oder autonom, des Weiteren soll anhand von Sensoren der Raum kartographiert werden können und für den Benutzer eine grafische Oberfläche entwickelt werden.



Zu Beginn des Projektes wurde gemeinsam ein Zeitplan für das Projekt erstellt und grafisch dargestellt, um so einerseits die gewünschten Deadlines niedergeschrieben zu haben, der aber auch als Orientierung für die Durchführung der einzelnen Teilprojekte diente.



6. Projektplanung und Projektauftrag

Zu Beginn des Projektes war es essentiell einzelne Aufgabenbereiche festzulegen, welche den entsprechenden Interessen und Fähigkeiten der Teammitglieder entsprachen. Hierzu wurde ein gemeinsames Meeting einberufen und im gemeinsamen Austausch über Priorisierungen die Themenbereiche festgelegt.

Es wurde nicht explizit ein Teamleiter festgelegt, da es im Team eine gute Kommunikation gab, wer welche Aufgaben erledigt und von welchem Mitglied notwendige Unterlagen an die Dozenten gesendet werden, diesbezüglich wurde sich auf eine gemeinsame Kommunikationsplattform für den Austausch wichtiger Informationen festgelegt und über diese dann immer rechtzeitig abgesprochen würde, welches Teammitglied sich einzelner organisatorischer Aufgaben annimmt. So wurden alle Abgabefristen zeitgerecht erfüllt und sorgten für keine Unruhen im Team oder Verärgerung der Dozenten.

Bei der Projektplanung wurde sich auch sofort auf eine Plattform zum Austausch aller softwaretechnischen Inhalte geeinigt. Hierzu wurde ein GitHub-Projekt erstellt. Einige Team-Mitglieder hatten bereits zuvor mit der Plattform GitHub gearbeitet und unterstützten die Teammitglieder, die diese zuvor noch nicht genutzt haben, bei der Verwendung.

Im Rahmen der Projektplanung wurde ein grafischer Zeitplan erstellt, in welchem die einzelnen Projektteile erfasst wurden und mit den entsprechenden Teammitgliedern hinterlegt wurden.

Es wurde auch eine Umfrage unter den Teammitgliedern durchgeführt, zu welchen wöchentlichen Terminen Meetings stattfinden können. Es wurde sich letztendlich darauf geeinigt, dass wöchentlich freitags dieses Meeting stattfinden sollte, da an diesem Tag auch die Dozenten anwesend waren und über wichtige Abgaben oder andere Informationen mitteilten.

In der Regel waren diese Meetings auch von allen besucht. Falls ein Teammitglied diese Meetings einmal nicht wahrnehmen konnte, wurde dies immer rechtzeitig mitgeteilt.



7. Plakaterstellung

Zeitnahe zum Beginn des Projektes sollte für dieses ein Plakat erstellt werden. Man einigte sich darauf im Zuge der Plakaterstellung auch ein Logo zu entwerfen, welches für das weitere Projekt verwendet werden kann und dem Teamnamen entspricht. In einem gemeinsamen Brainstorming wurden mögliche Ideen zusammengetragen und sich auf eine Vorstellung geeinigt. Das Logo wurde zuerst gezeichnet und dann am Computer weiterbearbeitet.

Das Logo sollte schlicht, aber eindeutig sein. Am Ende entschied sich das Team gemeinsam auf einen Entwurf von Anja Strobel, welcher das Projekt gut präsentierte.

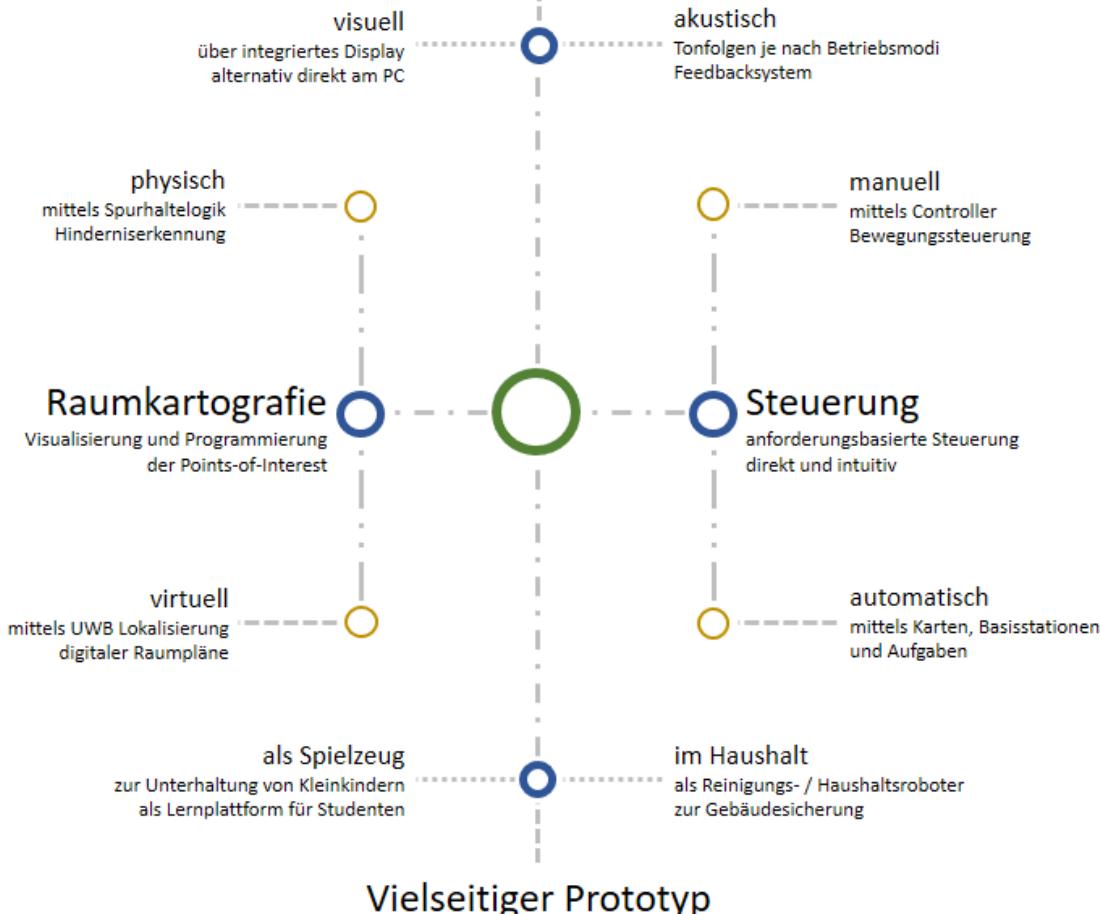


Während das Logo entwickelt wurde, kümmerte sich Dominik Scharnagl bereits um die Erstellung des Plakats. Hierbei hatte er freie gestalterische Möglichkeiten vom Team gewährt bekommen. Er entschied sich für eine innovative Gestaltung des Plakats. Bevor das Plakat an die Dozenten gesendet wurde, schickte Dominik Scharnagl das Plakat an das Team um Rücksprache zu halten. Das Team fand die innovative Darstellung des Projektes sehr kreativ und ansprechend und gab somit ihr Einverständnis für das Plakat ab. Somit war die Erstellung des Plakats abgeschlossen.





Interaktiv



Vielseitiger Prototyp

Ziel des Prototyps ist es, ein multifunktionales teilautonomes Fahrzeug für den Heimgebrauch zu schaffen.
Das Projekt „StarCar“ soll somit richtungweisende Techniken vereinen und in heterogenen Szenarien einsetzen.



Team 02 – StarCar

Anja Strobel · Annkathrin Bauer · Dominik Scharnagl · Florian Boemmel
Mehmet Billor · Robert Graf · Simone Huber





8. Software

8.1. Programmiersprachen

Zu Beginn des Projektes wurde sich auf eine gemeinsame Programmiersprache geeinigt, diese war C/C++. Im Laufe des Projektes war es nötig einige Teile des Projektes in anderen Programmiersprachen durchzuführen, diese waren Python und C#. Die Sprachwahl für das Display, wird in Kapitel [14.2.](#) genauer diskutiert.

8.2. Programme

Bei der Entstehung des Projektes wurde eine Vielzahl von Programmen verwendet um unterschiedlichste Analysen und Tests durchzuführen wie auch als Entwicklungsplattform zur Programmierung oder zum Debugging.

Es wurden folgende Programme verwendet:

- Arduino IDE 1.8.5
- Visual Studio 2017 15.3.1 + Arduino IDE Erweiterung
- IAR Embedded Workbench 6.50.1
- Device Monitoring Studio 7.81
- Advanced Serial Port Terminal 6.0
- PicoScope 6
- Fritzing 0.9.3
- Simple Motor Control Center 1.2.0.0
- MATLAB R2017b
- PuTTY
- PSFtp
- Urg Viewer
- CuteCom
- DecaRangeRTLS



9. Hardware

9.1. Firmware des Arduino

Ersteller: Dominik Scharnagl

9.1.1. Anforderungen

Jeder für das Projekt zum Einsatz kommende Sensor wie auch die Ansteuerung der Motorsteuerung und des Servos soll in einen Arduino integriert werden. Die Firmware des Arduino soll dabei so modular aufgebaut sein, dass es zu jedem Zeitpunkt möglich sein soll, eine der am Board angeschlossenen Peripherien softwaretechnisch „abzustecken“. Ein weiterer Aspekt soll zudem berücksichtigen, dass manche Module die Arduino-typische Setup-Sequenz durchlaufen müssen, bevor sie ihre zyklische Arbeit während der Loop-Sequenz aufnehmen können.

Das Hauptaugenmerk soll bei der Implementierung auf die Datenkapselung in eine zentrale Klasse liegen. Über dieses Objekt sollen alle anfallenden Daten während einer Loop-Sequenz und über Loop-Sequenzen hinaus zwischengespeichert und anderen Modulen zur Verfügung gestellt werden. Der besondere Vorteil dabei soll darin liegen, dass kein anderes Modul eine spezielle Referenz auf ein anderes Modul als sich selbst benötigt, um bestimmte Daten abzurufen. Diese lose Kopplung (engl. loose coupling) soll zudem zur leichteren Modularisierung der einzelnen Komponenten der StarCar-Firmware führen.

Aufgrund der Größe des Teams und der Anzahl der diversen Komponenten, die in das System einfließen, wird jedes Teammitglied seine eigene kleine „Insellösung“ anfertigen. Es gilt deshalb, eine möglichst einfache Möglichkeit zu schaffen, diese Lösungen ohne große Änderungen des Codes in die Firmware zu integrieren.

9.1.2. Analyse / Design

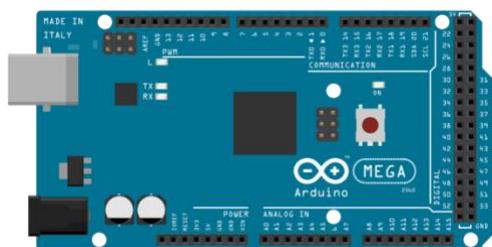
Eine Arduino Firmware besteht typischerweise genau aus einer Datei mit der Dateiendung INO. Per Definition muss in dieser Datei eine Methode namens „setup“ und eine Methode namens „loop“ existieren – beide Signaturen sind parameterlos und ohne Rückgabewert. Während die „setup“-Methode nur einmalig beim Anschließen des Arduinos an eine Stromquelle aufgerufen wird, wird die „loop“-Methode im Stil eines üblichen „Main-Loop“ kontinuierlich immer wieder aufgerufen. Kommt es zu einem Software- / Hardwarefehler, der entsprechend schwerwiegend ist, dann startet der Arduino von Neuem und durchläuft wiederholt die „setup“-Methode und anschließend kontinuierlich die „loop“-Methode.





Zu Debugging-Zwecken wird in nahezu jeder Beispielfirmware mindestens die serielle Kommunikation konfiguriert, indem der Klasse „Serial“ mittels „begin“-Aufruf die gewünschte Baudrate mitgeteilt wird. Dieses Vorgehen ist deshalb so üblich, da es für den Arduino häufig die einfachste / einzige Möglichkeit ist, ein Feedback an den Entwickler zu geben, insbesondere auch daher, weil die Arduino IDE kein Debugging der Firmware unterstützt.

Prinzipiell bietet die Arduino Plattform nahezu alles an, was man als C/C++ Entwickler kennt. Bis auf größere Frameworks wie MFC, Qt und Ähnlichem lassen sich alle bekannten Elemente der STL in einer Arduino Firmware integrieren und verwenden. Die Community zur Plattform bietet darüber hinaus auch ein äußerst umfangreiches Sortiment an zusätzlichen Arduino „Libraries“. Diese „einfachen Ordner“ mit entsprechenden C-Header- und C-Source-Dateien lassen sich einfach in ein Arduino Projekt per Include Anweisung einbinden und ohne weiteres verwenden. Zu beachten ist dabei, dass eine eingebundene Library entsprechend Einfluss auf die Größe der Firmware hat und der Speicher des Arduinos beschränkt ist. So stellt der Arduino Uno einen Flash-Speicher mit einer Größe von 32 KB bereit, der für die Firmware, dem Bootloader und Benutzerdaten ausreichen muss.



Im Rahmen unserer Projektarbeit zielten wir darauf ab, dass der vom Arduino Uno bereitgestellte Flash-Speicher unseren Anforderungen genügen sollte. Trotz dieser groben Einschätzung haben wir sichergestellt, dass wir eine Alternative haben, falls der Flash

Speicher des Arduino Uno nicht ausreichen sollte. Diese Alternative wäre ein wesentlich leistungsstärkerer Arduino Mega mit einer Flash-Speicherkapazität von 256 KB. Ebenso stand zu Beginn des Projektes noch nicht fest, wie viele PINs wir für alle Komponenten konkret benötigen und ob dementsprechend die Anzahl der digitalen wie auch analogen PINs des Arduino Uno für unsere Ansprüche ausreichen würde. Ein weiterer, nicht ganz klarer Punkt war auch die Leistung der Plattform im Allgemeinen. Aufgrund der Absicherung über den Arduino Mega sahen wir jedoch keine Probleme bei der nun folgenden Implementierung der Firmware.

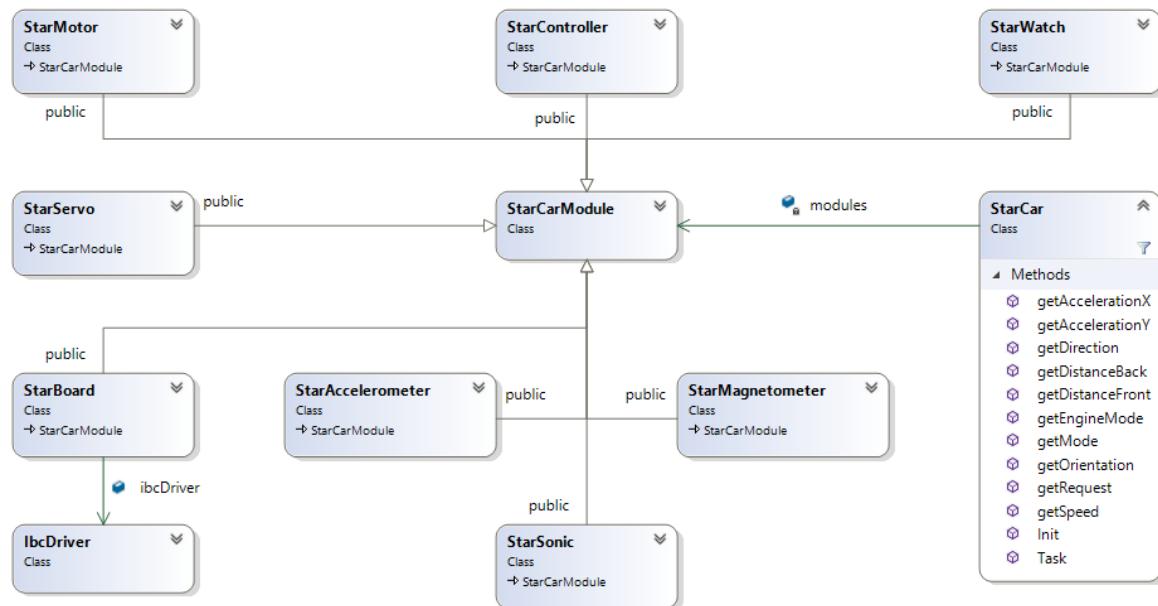
9.1.3. Implementierung

Die von uns an die Firmware gestellten Ansprüche, insbesondere die der Datenkapselung und der losen Kopplung realisierten wir schrittweise. Während die Ansteuerung der Motorsteuerung zu Beginn direkt in der Loop-Methode stattgefunden hat, lagerten wir bereits zum Zeitpunkt der Servo-Integration den Code in eine „StarMotor“-Klasse aus. Nachdem wir die Steuerungslogik für den Servo ebenfalls fertiggestellt hatten, verschoben wir auch diese in eine entsprechende Klasse namens „StarServo“. Hierbei implementierten



wir den notwendigen Setup-Code in einer Setup-Methode in der jeweiligen Klasse. Eine entsprechende, klassenspezifische Loop-Methode war bis dato noch nicht vorgesehen, da jede Klasse bisher immer nur wenige Test-Routinen implementierte, welche im „Main-Loop“ des Arduinos aufgerufen wurden.

Bei der weiteren Implementierung der Firmware lernten wir das generische USB-Interface der „USB Host Shield Library 2.0“ kennen. Die dabei von der „USB“-Klasse bereitgestellte API implementierte eine Art „Init-Task“-Pattern. Während man die Klasse über die Init-Methode in der Setup-Routine des Arduino initialisieren musste, arbeitete man in der Loop-Methode stets mit dem Aufruf der „Task“-Methode. Nur wenn dieses Pattern eingehalten wurde, konnten spezifische USB-Treiber wie der „XBOXUSB“ Treiber der Library mit der Klasse verlinkt und entsprechend verwendet werden. Dieses Pattern hat uns so sehr zugesagt, dass wir es auf unsere Klassen adaptierten. Dementsprechend rief ab diesem Zeitpunkt die Arduino Setup-Methode auf jeder Klasse eine Init-Methode und die Arduino Loop-Methode auf jeder Klasse eine Task-Methode auf. Nachdem wir weitere Kapselungen von allgemeinen Board-Funktionen vornehmen wollten, implementierten wir zu den bisherigen Klassen die „StarBoard“-Klasse. Aufgabe dieser ist es, sich um das allgemeine Setup und den Ablauf von generellen Dingen wie die Steuerung der LEDs des Fahrzeugs zu kümmern. So implementiert die Klasse zum Beispiel die Lichteffekte bei der Erkennung von Hindernissen, die Lichtsignale während der Fahrt und während der Startup-Sequenz der Firmware.



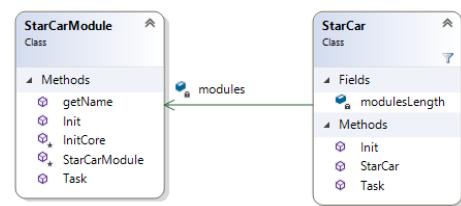
An dem Punkt, an dem wir den Xbox 360 USB Controller in die Firmware integrierten, stellten wir im Rahmen der „StarController“-Klasse fest, dass wir ein entsprechendes Objekt zur Datenkapselung benötigten. Aus dieser (bereits vorab geplanten) Anforderung heraus entstand die „StarCar“-Klasse. Sie soll die bei der Steuerung mittels Controller die für die Motor- und Servosteuerung relevanten Parameter liefern. Dementsprechend stellt die „StarCar“-Klasse Methoden zum Lesen und Schreiben der aktuellen Beschleunigung als



auch der aktuellen Richtung bereit. Im weiteren Verlauf der Logik mussten die Klassen „StarServo“ und „StarMotor“ entsprechend einen Verweis auf das „StarCar“-Objekt erhalten, um die vom „StarController“-Objekt gespeicherten Steuerungswerte zu verarbeiten. Dies geschah, indem wir das globale „StarCar“-Objekt beim Aufruf der Task-Methode dem entsprechenden Modul mitgaben.

Nach der Integration der Sensorik in entsprechende Klassen wie beispielsweise „StarSonic“ für die Ansteuerung eines Ultraschallsensors, konnten wir uns sicher sein, dass die Schnittstellen der einzelnen Module zueinander kompatibel sind. Somit konnten wir in einem letzten Schritt die Abstraktion der einzelnen Module weiter ausbauen und implementierten die Basisklasse „StarModule“. Jedes Modul wurde entsprechend von dieser Klasse abgeleitet (siehe Abbildung oben).

Zum Abschluss der Abstraktion änderten wir die Rolle der „StarCar“-Klasse von der einfachen Datenklasse hin zur Controllerklasse. Somit sah ab diesem Zeitpunkt die Architektur der Firmware so aus, dass in der INO Datei alle Module in einem Array von Modul-Zeigern referenziert und an das „StarCar“-Objekt übergeben wurden. In der Setup-Routine des Arduinos wurden schließlich nur noch der Init des „StarCar“-Objektes wie auch dessen Task Methode in der Loop-Routine des Arduinos aufgerufen. Die von den Modulen notwendige Referenz auf das „StarCar“-Objekt übergab ab diesem Zeitpunkt das Objekt selbst über seinen „this“-Zeiger.



Arduino Uno stürzt regelmäßig ab

Trotz unserer anfänglichen Bedenken, dass die Ressourcen des Uno nicht ausreichen könnten, konnten wir den Arduino Uno sehr lange für unser Projekt einsetzen. Als kleinen Makel seiner geringen Leistungstellten wir schnell fest, dass häufige und umfangreiche Ausgaben von Debugging-Nachrichten über die Serial-Klasse schnell zum Absturz des Arduino Uno führten. Bis wir aber diese Ursache für die anfänglichen Abstürze ausmachen konnten, mussten wir einiges an Nachforschungsarbeit investieren.

Nachdem uns die Arduino IDE wie auch das Visual Studio mit der Arduino IDE Erweiterung die prozentuale Auslastung des Flash-Speichers beim Flashen mitteilte, beobachteten wir im Verlauf der Implementierung regelmäßig, wie sich die Auslastung des Speichers änderte. Bereits ab der Mitte des Projektes erreichten wir 50% der maximalen Kapazität. Nach der Integration aller Sensoren waren es schließlich 80%.

Besonders erwähnenswert dabei ist, dass sich bei dieser Auslastung die Firmware manchmal nicht mehr auf den Arduino flashen ließ. Funktionierte das Flashen, kam es jedoch während der aktiven Steuerung des Fahrzeuges schnell zum Absturz des Arduino. Demnach gingen wir davon aus, dass der Arduino Uno nicht länger unseren Ansprüchen



gerecht wird und wir versuchten es mit dem Arduino Mega aus privaten Mitteln von Dominik Scharnagl. Nach Austausch der Boards stellten wir keinerlei Probleme durch Abstürze mehr fest und konnten ebenso auch wieder diverse Debugging-Benachrichtigungen aktivieren.

9.1.4. Test

Die Firmware des Arduinos an sich musste kaum bis gar nicht getestet werden, da aufgrund der Modularisierung der Firmware jedes Modul separat und sehr intensiv getestet wurde. Somit kann man sagen, dass die Firmware-Tests bereits durch die Funktionstests der einzelnen Module durchgeführt wurden.

9.2. Aufsetzen des Raspberry Pi 3

Ersteller: Mehmet Billor

Der Raspberry wurde für das Projekt neu aufgesetzt und Raspbian „Strech with Desktop“ verwendet. Das System wurde auf eine 8GB Speicherkarte installiert. Um das Image auf die Speicherkarte aufzuspielen wurde der W32DiskImager verwendet. Nach der Installation wurde das Standardpasswort geändert, eine feste IP-Adresse eingestellt und folgende Packages zusätzlich installiert:

Package #1

Package #2

...

Package #n

Quelle Raspbian: <https://www.raspberrypi.org/downloads/raspbian/>

Raspberry Pi startet nicht

Der Raspberry Pi wurde zunächst mit einem Monitor und einer Tastatur angeschlossen. Nachdem das Image auf die Speicherkarte geflasht wurde, hätte es normalerweise starten müssen. Jedoch war kein Bild zu erkennen. Die erste Vermutung war, dass beim Flashen des Images etwas schiefgegangen war, deswegen wurde das Image neu geflasht. Jedoch trat das gleiche Problem wieder auf. Da dieser Vorgang nicht im Labor standfand musste die weitere Bearbeitung des Problems auf das Labor vertagt werden. Im Labor wurde zunächst eine andere Speicherkarte ausprobiert, jedoch wieder ohne Erfolg. Anschließend wurde die Karte bei einem anderen Raspberry Pi ausprobiert. Die Raspberry fuhr nun hoch. Dadurch wurde das Problem gefunden. Der erste Raspberry Pi war defekt.



Raspberry Pi für SSH einrichten

Um sich mit per SSH auf den Raspberry zu verbinden, benötigte der Raspberry eine IP-Adresse. Das Problem war, dass der Pi sich mit dem „TI-Roboter“ Netzwerk verband, welches jedoch die IP-Adressen automatisch dynamisch vergab. Zu diesem Zeitpunkt war noch unklar, ob am nächsten Tag noch die gleiche IP-Adresse oder andere für den Raspberry vergeben wurde.

Der erste Ansatz war, die aktuelle IP-Adresse auf dem Display anzeigen zu lassen. Jedoch gab es zu diesem Zeitpunkt noch Probleme mit dem Display bzw. es war noch nicht installiert. Außerdem war es nicht möglich, dies kurzfristig umzusetzen. Nach einem Hinweis vom Herrn Altmann, entschieden wir uns die IP-Adresse „fest“ einzustellen, da nur wenige Geräte mit dem Netzwerk verbunden waren und daher eine Kollision eher unwahrscheinlich war. Mit dieser Lösung kam es im Verlauf des gesamten Projektes zu keinen weiteren Problemen.

9.3. Installation des Displays

Ersteller: Mehmet Billor

Für das Display auf dem Raspberry PI wurde ein 3.2 Zoll TFT Display verwendet welcher einfach auf die GPIO Pins aufgesteckt wurde. Dadurch wurden 26 Pins belegt. Die restlichen Pins wurden nicht weiterverwendet. Die Installation des Displays umfasste die Installation eines Treibers und die Konfiguration einiger Dateien (genaueres siehe Installationsanleitung).

Quellen verwendeter Installationsanleitungen:

- Hersteller
http://anleitung.oy-it.net/wp-content/uploads/2017/04/RB-TFT3.2_RB-TFT3.5_Manual_11-12-2017.pdf
- Conrad
http://www.produktinfo.conrad.com/datenblaetter/1300000-1399999/001380381-an-01-de-8_13_CM_3_2_TOUCH_DISPLAY_320X240_PX.pdf

Raspberry Pi Display Installation

Bei der Installation kam es zu zwei Problemen. Zum einen funktionierte der „wget“ Befehl für den direkten Download des Treibers nicht. Nach längerer Recherche wurde der Treiber von einer anderen Seite (Quelle nicht mehr vorhanden) runtergeladen und manuell auf den Raspberry kopiert und installiert. Dadurch konnte die Installation abgeschlossen werden. Jedoch kam es dann zu weiteren Problemen mit der Konfiguration des Displays. Zum einen war das Bild des Displays um 180 Grad verdreht und zum anderen reagierte der Touch nicht wie gewollt.



Die Vorgabe aus der Installationsanleitung für die Standard-Einstellung „passte“ nicht. Da keine weiteren Quellen gefunden worden, wurde die richtige Einstellung durch mehrere verschiedene Einstellungen und rumprobieren letztendlich gefunden. Dieses Problem hätte auch vermieden werden können durch die Wahl eines Images mit vorinstalliertem Display.

9.4. „Sabotage“ / Verschwinden von Teilen

Ersteller: Mehmet Billor

Zu diesem Phänomen kam es vor allem zu Beginn des Projektes, als noch nicht alle Fahrzeuge "fertig" gebaut waren. Es kam immer wieder dazu, dass Teile aus unserer Box verschwunden sind. Beispielsweise hatten wir zu Beginn mehrere Ultraschallsensoren, an manchen Tagen war es dann nur noch einer und an anderen wieder mehr, obwohl niemand aus unserer Gruppe an dem Fahrzeug gearbeitet hatte. Oder es war plötzlich ein "anderer" Ultraschallsensor (anderes Modell) in unserer Kiste. Das Verschwinden bzw. Austauschen von Teilen führte teilweise zu großer Verwirrung innerhalb der Gruppe. Bei jedem Auftreten des Problems sprachen wir uns ab. Wir konnten das Problem zunächst "lösen" indem wir die Teile solange mit nach Hause nahmen, bis unser Auto bereit für den Zusammenbau war. Jedoch kam es dann trotzdem zu ähnlichen Problemen wie zum Beispiel das Verschwinden unserer Akkus, was zur Verzögerungen an Arbeiten am Auto führte.



10. Sensoren / Aktuatoren

10.1. Motor

Ersteller: Dominik Scharnagl, Mehmet Billor

10.1.1. Anforderungen

Die Steuerung des Motors soll mittels Simple High-Power Motor Controller 24v12 umgesetzt werden. Der speziell für Bürsten-DC-Motoren von Pololu Robotics & Electronics entwickelte Motor Controller kann dabei über eine Mini-USB-Schnittstelle mittels kostenloser „Simple Motor Control Center“-Software konfiguriert werden. Der Controller an sich kann mittels USB, analoger Spannung, via TTL seriell als auch per Funk gesteuert werden. Die für diesen Controller nötige Steuerungslogik soll dabei in einem eigenen „StarModule“ namens „StarMotor“ implementiert und gekapselt werden.

10.1.2. Analyse / Design

Die von uns eingesetzte Version des Controllers arbeitet mit einer Spannung von 5,5 V bis 40 V und kann dabei 12 A ohne zusätzlichen Kühlkörper liefern.



Zum Kennenlernen der Steuerungslogik haben wir eine kleine Testanwendung aufgesetzt und die Motorsteuerung am Arduino angeschlossen. Diese Tests zeigten uns, dass die Motorsteuerung über diverse Parameter konfigurierbar ist. Der für uns primäre Parameter zu Steuerung der Geschwindigkeit kann über einen Steuerungswert aus dem Wertebereich [0, 3200] beeinflusst werden. Die dabei entscheidende Drehrichtung wird dem Controller mittels 0x85 für „Vorwärtsfahren“ und 0x86 für „Rückwärtsfahren“ mitgeteilt. Unsere Tests haben ergeben, dass der Controller den maximalen Steuerungswert von 3200 nur sehr schwer erreicht, da er häufig bereits ab Werten über 2800 abstürzt und sich in den „Exit-Safe-Start“-Modus versetzt. In diesem Modus hält der Controller den Motor sofort und direkt an, indem er die Stromversorgung zu diesem unterbricht.

Weiter fanden wir im Onlinehandbuch des Herstellers (Quelle: <https://www.pololu.com/docs/0J44/6.2.1>) heraus, dass man die Motorsteuerung bezüglich des Beschleunigungsintervalls bei der Beschleunigung und Entschleunigung entsprechend den eigenen Wünschen konfigurieren kann.



10.1.3. Implementierung

Nach den in der Analysephase erworbenen Kenntnissen wurde der zur Ansteuerung des Motor Controllers notwendige C/C++ Code in eine gesonderte Klasse namens „StarMotor“ verlagert. Die dabei von der Klasse implementierte API erweiterte die Handhabung der Motorsteuerung um zusätzliche Funktionen, um nicht nur die Soll-Geschwindigkeit des DC-Motors einzustellen. So ermöglicht es die Klasse unter anderem auch, die Ist-Geschwindigkeit der Motorsteuerung oder Fehlercodes auszulesen.

Zur Kommunikation mit dem Motorcontroller entschieden wir uns für das binäre Protokoll von Pololu und integrierten die notwendigen, seriell zu übertragenden Befehle kontextabhängig ohne weitere Abstraktionsschicht in den einzelnen Methoden der „StarMotor“-Klasse. Zur Übertragung der Kommandos verwendeten wir die Arduino SoftwareSerial Library. Ausgangspunkt der Implementierung war das Beispielprojekt von Pololu für den Arduino (Quelle: <https://www.pololu.com/docs/0J44/6.7.1>).

Entgegen des in der Analysephase ermittelten Intervalls des Steuerungswertes für die Geschwindigkeit haben wir uns dazu entschlossen, die Beschleunigung über einen Prozentwert auszudrücken. Hierbei soll 0% keine, -100% eine negative und +100% eine positive Beschleunigung beschreiben. Alle Werte dazwischen steuern entsprechend schwächer, aber in dieselbe Richtung in Abhängigkeit zu ihren Vorzeichen. Bei der Umsetzung des Prozentwertes auf den eigentlichen Steuerungswert haben wir uns auf den maximal möglichen Wert 500 (zum Zeitpunkt der Präsentation war es noch der Wert 1000) geeinigt, da bei größeren Werten das Fahrzeug nur schwer handzuhaben ist. Demnach entspricht der Steuerungswert 500 einem Prozentwert von 100%. Auf diese Weise können wir direkt, schnell und proportional wie auch unabhängig von der Art der Steuerung des Motors die Beschleunigung variieren.

Betrachtet man den Code an dieser Stelle, wird man zudem feststellen, dass wir das Vorzeichen des Steuerwertes umkehren. Das liegt daran, weil der Motor so in das Chassis eingebaut ist, dass dessen Übersetzung ein entgegengesetztes Steuerverhalten der Motorsteuerung verlangt.

Motorsteuerung reagiert nicht mehr

Im Rahmen des weiteren Zusammenbaus des Fahrzeugs führten wir regelmäßige Tests durch. Durch diese Tests stellten wir ab einem gewissen Zeitpunkt fest, dass die Motorsteuerung nicht länger funktionsfähig war. Nachdem wir keine bewussten Änderungen an der Hardware wie auch der Verkabelung vorgenommen hatten, gingen wir davon aus, dass Änderungen an der Software die Ursache sein mussten. Bei der Untersuchung der zuletzt vorgenommenen (kleineren) Änderungen in der Software für ein von der Motorsteuerung völlig unabhängiges Modul, konnten wir keine Einflussfaktoren erkennen. Selbst ein Rückgängigmachen der Änderungen bis zum zuletzt wissentlich



funktionsfähigen Stand führte nicht zur ordnungsgemäßen Funktion der Motorsteuerung. Somit konnten wir einen Fehler in der Software ausschließen. Daher entschieden wir uns zusätzlich dazu, die Steuerungssignale der Motorsteuerung beziehungsweise das PWM-Signal des Controllers an den Pololu mittels Oszilloskop und Multimeter zu prüfen. Nachdem auch diese keine Auffälligkeiten aufwiesen, testeten wir die Funktion des Pololu Motor Controllers mit Hilfe des „Simple Motor Control Centers“ von Pololu. Dabei stellten wir schnell fest, dass auch hier der Controller nicht länger auf die Eingaben reagierte. Diese Tests brachten uns zu dem Schluss, dass womöglich bei der Änderung der allgemeinen Verkabelung am Fahrzeug der Controller durch eine Überspannung zu Schaden kam. Somit wurde ein neuer Motor Controller der gleichen Serie manuell von uns gelötet und der defekte Controller durch diesen ersetzt. Anschließende Tests stellten zusätzlich sicher, dass der neue Controller wie erwartet funktionierte. Selbst die Wiedereinführung der zuvor rückgängig gemachten Software-Änderungen änderte nichts an der vollen Funktion des neuen Controllers.

Motorsteuerung stürzt beim Beschleunigen ab

Während der parallel durchgeföhrten Integration der Steuerung mittels Xbox 360 USB Controller wurde in regelmäßigen Abständen auch das Steuerverhalten in und entgegen der Fahrtrichtung geprüft. Nach bereits mehrfach erfolgreich durchgeföhrten Funktionstests bemerkten wir, dass eine Beschleunigung mittels Xbox Controller an undefinierbaren Punkten zum Absturz des Pololu Controllers führte. Auffällig dabei war, dass insbesondere bei einer „Kick-Down“-Beschleunigung der Pololu direkt und fehlerfrei reagierte. Während wir bei einer sehr langsamen Beschleunigung bis in das obere Mittel des Beschleunigungsgrades ebenso keinerlei Probleme hatten, wurden wir aber bei einer Beschleunigung darüber hinaus gehäuft mit Abstürzen der Motorsteuerung konfrontiert. Nachdem wir jedoch wiederholt keine bewussten Änderungen an Hardware oder Firmware vorgenommen hatten, versuchten wir das Problem weiter einzugrenzen. Hierzu minimierten wir die Firmware auf eine einfache Testanwendung, welche im Sekundentakt den Motor immer weiter beschleunigte. Auch hier konnten wir feststellen, dass eine Beschleunigung über das Mittel des größtmöglichen Beschleunigungsgrads zum Anhalten des Motors sowie zum Absturz der Motorsteuerung führte. Weiter stellten wir fest, dass ein „Exit-Safe-Start“-Kommando an den Controller das Problem behob, jedoch eine wiederholte Ansteuerung über das Mittel hinaus erneut zum gleichen Problem führte. Daraufhin führten wir, ergänzend zu unserer eigenen TTL-basierten Steuerungslogik, weitere Tests mittels „Simple Motor Control Center“ durch. Auch hier konnten wir keine Besserung des Verhaltens feststellen. Weitere Messungen mit dem Voltmeter brachten uns spontan dazu, den selbst gelöteten Controller durch einen bereits vormontierten Controller aus dem Labor-Bestand zu ersetzen. Auch bei diesem Pololu Controller stellten wir selbiges Verhalten fest. Im weiteren Verlauf des Ausschlussverfahrens zur Ermittlung der Ursache des Verhaltens konnten wir schlussendlich nur noch den Akku als Fehlerquelle einordnen. Ein Tausch des Akkus durch einen vollgeladenen Akku führte letztlich zur Lösung des Problems. Interessant hierbei war, dass besagter leerer Akku erst am Vortag von uns



geladen und bis zum Auftreten des Problems nicht genutzt worden war. Demzufolge gehen wir davon aus, dass unser Akku (versehentlich) durch einen leeren Akku durch eines der Mitglieder eines anderen Teams getauscht wurde.

10.1.4. Test

Die für die Motorsteuerung notwendigen Tests wurden mittels einfachen Funktionstests durchgeführt. Hierbei wurde anfangs der Controller über einfache Testroutinen im Sekundentakt beschleunigt und wieder entschleunigt. Im weiteren Verlauf des Projektes wurde die Motorsteuerung dann zum Großteil mittels Xbox 360 USB Controller und gegen Ende des Projektes mittels eZ430-Chronos-Watches gesteuert und zugleich getestet.

Parallele Tests nach jeder Integration eines weiteren Sensors / einer weiteren Komponente oder nach Änderungen an der Verkabelung haben teilweise zu Nebeneffekten geführt, welche aber nicht an der Motorsteuerung – als Quelle des Verhaltens – festgemacht wurden, sondern häufig darin begründet waren, dass die Sensorik das Gesamtsystem ausbremste.

Von besonderer Bedeutung bei allen Tests und vor allem in der Analysephase waren die Beispielprojekte von Pololu sowie das „Simple Motor Control Center“ von Pololu. Mithilfe dieser Quellen und Werkzeuge konnten wir vielfach die korrekte Funktion des Motorcontrollers prüfen und nachweisen.

Ebenso konnten wir während der diversen Tests, insbesondere denen im „nicht aufgebockten Zustand“, das genauere Beschleunigungsverhalten des Controllers einordnen und so die Ansteuerung des Motors für unsere Zwecke optimieren. Im Rahmen dieser so erlangten Erkenntnisse konfigurierten wir die maximale Beschleunigung in und entgegen der Fahrtrichtung mit dem Steuerungswert 1, sowie die maximale Entschleunigung mit dem Wert 10. Diese Werte haben uns in der Praxis gezeigt, dass unser Fahrzeug dadurch ein angenehmeres und flüssigeres Fahrverhalten erhält.



10.2. Servo

Ersteller: Dominik Scharnagl, Mehmet Billor

10.2.1. Anforderungen

Zur Steuerung der Lenkung des Fahrzeugs soll der bereits ab Beginn des Projektes verbaute RC-Car Servo 4519 DBB MG verwendet werden. Die für den Servo nötige Steuerungslogik soll dabei in einem eigenen „StarModule“ namens „StarServo“ implementiert und gekapselt werden.

10.2.2. Analyse / Design

Der doppelt kugelgelagerte Analogservo stellt mit einem Stellmoment von 35 Newton-Zentimeter mit einer Stellzeit von 0,23s bei einer Spannung von 4,8 V in die gewünschte Position.



Zur Erfassung der Steuerungslogik haben wir, wie auch bei der Motorsteuerung, eine kleine Testanwendung aufgesetzt. Aufgabe der Testanwendung war die Steuerung des Servos im Sekundentakt von links nach rechts und umgekehrt. Mithilfe dieser einfachen Arduino Anwendung haben wir weitere Eckdaten über den Servo in Verbindung mit der Ansteuerung über die von uns verwendete Arduino Servo Library festgestellt. Die von der Library bereitgestellte Servoklasse ermöglicht es neben dem Steuern im Gradmaß, die Lenkung im Mikrosekundentakt vorzunehmen. Hierbei loteten wir die untere und die obere Grenze der maximal möglichen Stellungen des Servos nach links und rechts aus. Im Mikrosekundentakt bedeutete das, dass der Stellbereich des Servos über das Intervall [1100, 1600] abgebildet werden kann, wobei es möglich ist, die Neutralstellung mit dem Steuerungswert von 1365 Mikrosekunden zu erreichen. Werte außerhalb des Intervalls führen zu einer Übersteuerung des Servos, die sich entweder in keinerlei Änderung über den Anschlag hinaus oder in ein Zittern des Servos am Anschlag bemerkbar machen. Betrachtet man den Servo und seine möglichen Steuerungswerte über die Library genauer, ist es bestimmt möglich eine noch detailliertere Aussage über das Steuerverhalten zu treffen. Nachdem wir jedoch bei der Auslotung den Servo in Schritten von je 5 Mikrosekunden testeten und die Ergebnisse unseren Bedürfnissen entsprachen, haben wir nicht mehr Zeit als unbedingt nötig in die Analyse investiert.



10.2.3. Implementierung

Die Steuerungslogik des Servos wurde mit Hilfe der Arduino Servo Library implementiert. Da die Betriebsspannung unseres Fahrzeuges bei konstanten 5V liegt, reduzierte sich auch die Stellzeit des Servos entsprechend auf 0,2s. Das zur Ansteuerung des Servos nötige PWM-Signal generiert die Servo Library mithilfe eines 16 Bit-Timers des Arduinos. Das Signal der Pulsweltenmodulation wird dabei über den SIG-Anschluss des Servos übertragen. Nach den in der Analysephase erworbenen Kenntnissen wurde der zur Ansteuerung des Servos notwendige C/C++ Code in eine gesonderte Klasse namens „StarServo“ verlagert. Die dabei von der Klasse implementierte API erweitert die Handhabung der Servosteuerung um zusätzliche Funktionen, um nicht nur die Soll-Stellung des Servomotors einzustellen.

Wie bereits bei der Motorsteuerung haben wir uns auch bei der Servosteuerung dafür entschieden, die Lenkung über einen Prozentwert statt über einen konkreten Wert aus dem ermittelten Steuerungsintervall auszudrücken. So soll ein Prozentwert von -100% einen Links- und +100% einen Rechtsanschlag definieren. Die Neutralstellung des Servos soll über 0% erreicht werden. Entsprechend steuern alle Werte zwischen -100% und 100%, die ungleich 0% sind, die prozentuale Abweichung von der Neutralstellung nach links beziehungsweise nach rechts. Demnach entscheidet das Vorzeichen, in welche Richtung gelenkt werden soll. Ein negativer Prozentwert beeinflusst somit die Steuerung nach links und ein positiver die Steuerung nach rechts.

Die Herausforderung hierbei ist jedoch, dass das Intervall sich im Allgemeinen nicht wirklich „schön“ unterteilen lässt und deswegen der Steuerungswert ausschließlich mit den realen Abweichungen von der Neutralstellung in die eine oder andere Richtung berechnet werden muss. Das folgende Snippet zeigt, wie wir schließlich den Prozentwert auf den entsprechenden Steuerungswert umrechnen:



```
#define SERVO_LEFT 1100
#define SERVO_CENTER 1365
#define SERVO_RIGHT 1600

...
int_t ms = SERVO_CENTER;

sbyte_t dir = car->getDirection();
float_t direction = (float_t)dir / 100;

if (direction < 0) {
    // Results into values between 1100 and 1365 (inclusive).
    ms = SERVO_CENTER - (float_t)
        (SERVO_CENTER - SERVO_LEFT) * ((-1) * direction);
}
else {
    ms = SERVO_CENTER + (float_t)
        (SERVO_RIGHT - SERVO_CENTER) * direction;
}

...
this->servo.writeMicroseconds(ms);
```

Vorbelegung von PINs durch Bibliotheken

Während der ersten Experimente im Rahmen unseres Versuchsaufbaus zum Kennenlernen der notwendigen Steuerungslogik für den Servo wurden wir mit diversen Schwierigkeiten konfrontiert. Gemäß Online-Dokumentation der Servo Library haben wir die Testanwendung zur einfachen Steuerung des Servos implementiert. Zum Anschluss der SIG-Leitung des Servos verwendeten wir den dedizierten PWM-PIN9 des Arduino Uno (laut Arduino Uno Handbuch). Während der Ausführung der Testanwendung konnten wir keine Funktion des Servos feststellen. Selbst nach mehrfachem Untersuchen des Codefragments der Beispielanwendung konnten wir keinen Fehler in der Software finden. Ein anschließend alternativ verwendeter Servo ließ sich ebenso nicht ansteuern. Um letztlich sicherzustellen, dass unser – bereits verbauter – Servo ordnungsgemäß funktionierte, ließen wir ihn von einer anderen Gruppe testen. Bei dieser Gruppe konnten wir uns sicher sein, dass ihre Ansteuerung des Servos (über Python) funktioniert. Nach längerer Suche in Handbüchern, Datenblättern und Foren kamen wir letztlich über die Dokumentation der Servo Library darauf, dass bei Verwendung der Servo-Klasse die dedizierten PWM-PINS PIN9 und PIN10 des Arduino Uno nicht länger funktionieren.

(Quelle: <https://www.arduino.cc/en/Reference/Servo>)



Störungen im PWM-Signal

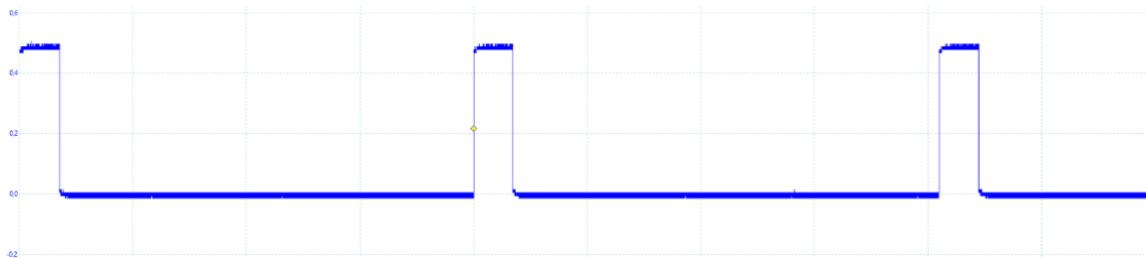
Nach Änderung des gewählten PINs für die SIG-Leitung des Servos von PIN9 auf den ebenfalls dedizierten PWM-PIN5 stellte sich weiterhin keine volle Funktion des Servos ein. Das ab diesem Moment festgestellte Verhalten beschränkte sich ausschließlich auf eine Lenkung nach links bis zum Anschlag. Bei den darauf durchgeföhrten Untersuchungen des PWM-Signals bezüglich dessen Korrektheit und Güte, beratschlagten wir uns mit Herrn Prof. Roth, denn die Analyse des PWM-Signals mittels PicoScope gestaltete sich – mit geringer Übung am Oszilloskop – als nicht sehr aufschlussreich. Mithilfe von Herrn Roth konnten wir letztendlich die zum Oszilloskop zugehörige Software PicoScope so einstellen, dass wir in der Lage waren, das PWM-Signal entsprechend zu analysieren. Selbst hierbei ist uns bezüglich der Korrektheit des Signals nicht aufgefallen, warum die Ansteuerung des Servos nicht funktionierte, denn unserer Ansicht nach war das PWM-Signal fehlerfrei. Bei weiterer Analyse während der stetigen Ausführung der Testanwendung ist Herrn Roth aufgefallen, dass das Signal einen Offset zum Null-Pegel aufweist. Die dabei stetige Differenz von ungefähr 2V zum Null-Pegel stellte sich als die Ursache für das Fehlverhalten des Servos heraus. Zur Lösung des Problems regte Herr Roth mit der Frage an, wo unser GND-Signal denn liege. Wie bereits von Herrn Roth vermutet, lag das GND-Signal des Servos nicht auf dem Arduino Uno, sondern auf dem Steckbrett unseres Versuchsaufbaus. Das Umlegen der GND-Leitung auf einen der GND-PINs des Arduino Uno löste somit auch dieses Problem. Ab diesem Moment ließ sich der Servo problemlos ansteuern.

Servomotor zittert und springt

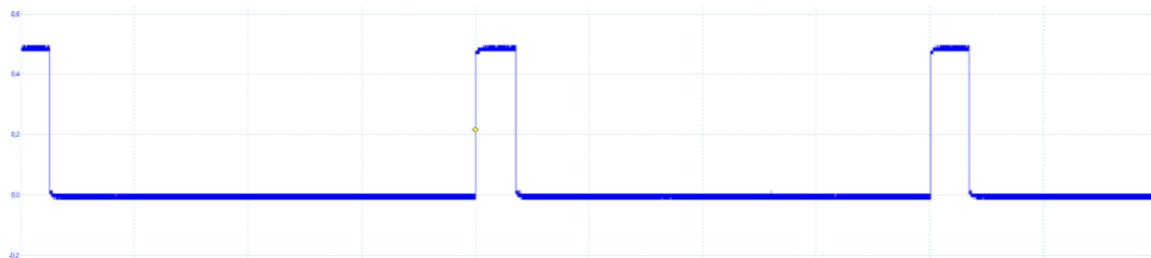
Gegen Ende Dezember bemerkten wir, dass der Servo im aufgebockten Zustand direkt nach der Setup-Sequenz zitterte, was er zuvor nicht getan hatte. Nachdem wir aber schon längere Zeit das Auto nicht im aufgebockten Zustand gehabt hatten, da wir uns viel mehr auf das Fahrverhalten mit direktem Reifen-Boden-Kontakt konzentrierten, fiel es uns schwer, die Ursache zügig zu finden. Codeänderungen wurden zwischenzeitig auch regelmäßig committet und gepusht. Selbst über die Historie der letzten Änderungen an der „StarServo“-Klassen konnten wir keine markanten / möglichen Ursachen dafür feststellen. Wir versuchten also das Problem einzuzgrenzen, indem wir bestimmte – typischerweise auffällige Teile – aus dem Ablauf der Firmware ausschlossen. Auch das Rückbauen der Hardware bis auf den Servo und den Motor brachte keine Besserung. Nach mehreren Stunden probieren, Versuch und Irrtum – was letztlich keinem strategischen Vorgehen mehr ähnelte – haben wir uns mit dem Problem vorerst abgefunden und mit der weiteren Arbeit an noch offenen Punkten befasst. Während des weiteren Verlaufs der Implementierungsphase stellten wir dann fest, dass der bis dato verwendete Arduino Uno nicht länger unseren Ansprüchen gerecht wird. Also tauschen wir den Uno durch einen Arduino Mega 2560. Nach dem „Einbau“ des Arduino Mega bemerkten wir, dass nach Ablauf der Setup-Sequenz der Servo nicht länger zittert, sondern springt und zwar von links nach rechts und umgekehrt. Diese Sprünge fanden dabei in Bruchteilen von wenigen Sekunden statt und waren eindeutig auf das beim Arduino Uno festgestellte Zittern zurückzuföhren, nun jedoch bei der höheren Taktung des Arduino Mega entsprechend



verstärkt. Somit waren wir gezwungen, die Ursache dafür zu finden. Nachdem wir beim ersten Anlauf verhältnismäßig planlos an das Problem herangegangen waren, entschlossen wir uns dazu, das Problem mittels Multimeter und PicoScope zu untersuchen. Die hierbei festgestellten PWM-Signale sind im folgenden Bild zu sehen:



Das hier eingestellte Zeitraster von 5 Millisekunden zeigte uns, dass offenbar jedes zweite Signal einen Versatz von wenigen Mikrosekunden hat. Dieser Versatz im PWM-Signal ließ wiederum auf das Verhalten des Servos schließen. Die Bedienung des Fahrzeugs durch parallele Tests – während weiterhin aktiver Analyse mittels Oszilloskop – behob immer wieder kurz das fehlerhafte PWM-Signal. Durch schrittweises Herantasten und Ausschließen von noch aktiven „StarModules“ stellten wir fest, dass das Problem auftritt, wenn das Modul „StarMotor“ aktiv ist. Ist hingegen jedes andere Modul außer des „StarMotor“ Moduls aktiv, ist das Problem nicht vorhanden. Somit konnten wir die Ursache auf die Motorsteuerung reduzieren. Wir bemerkten hier auch bei expliziten Tests mit der Controller Steuerung, dass bei bestimmten Eingaben am Controller das PWM-Signal wiederholt frei von Störungen war. Mit Hilfe weiter durchgeföhrter Ausschlussverfahren, indem wir den Code der „StarMotor“-Klasse Zeile für Zeile auskommentierten und wieder scharfschalteten, konnten wir die Ursache finden. Das Problem lag darin, dass es beim Senden des „Exit-Safe-Start“-Kommandos an die Motorsteuerung eine kurze Reflexion auf den Servo mit sich brachte. Zusammen mit der damaligen Logik „Beschleunigung ist gleich 0%, versetze die Motorsteuerung in den Exit-Safe-Start Zustand“, welche bei jedem Durchlauf im Main-Loop ausgeführt wurde, erklärte uns das die stetig im gleichen Intervall fehlerhaft produzierten PWM-Signale. Durch eine einfache boolesche Variable konnten wir das Problem schließlich lösen. Abschließend verifizierten wir die Änderung noch einmal mit dem PicoScope und stellten so die Fehlerfreiheit nicht nur im Black Box Test fest.



Wie man der obigen Abbildung entnehmen kann, sieht man jetzt, dass das PWM-Signal konstant an den Kanten der Zeitschlüsse ausgerichtet ist und nicht länger einen Versatz mit sich bringt.



10.2.4. Test

Die für die Servo-Steuerungslogik relevanten Tests wurden zum Großteil während des Projektes mittels Black Box Tests durchgeführt. Dementsprechend wurde nur ab und zu die Lenkung betätigt, während das allgemeine Fahrverhalten des „StarCars“ getestet wurde. Ausführlichere Tests fanden hingegen in Form von kleinen Testanwendungen nur während der Analysephase statt, ebenso wie während des Problems mit dem oben beschriebenen Zittern und Springen des Servos aufgrund des kontinuierlichen „Exit-Safe-Start“-Kommandos an die Motorsteuerung. Hier bewährte sich insbesondere PicoScope als eines der Hilfsmittel der ersten Wahl, wenn Änderungen an der Hardware beziehungsweise Software prinzipiell auszuschließen sind.

10.3. Ultraschallsensor

Ersteller: Simone Huber

Im Projekt wurden Ultraschallsensoren verwendet, um den Abstand zu Hindernissen zu bestimmen und das Auto bei einer berechneten Kollision zum Stoppen zu bringen. Im folgenden Kapitel wird die Umsetzung dieses Sensors detailliert aufgeführt.

Hier ist anzumerken, dass zu Beginn des Projektes geplant war 4 Ultraschallsensoren zu verwenden. Dies war aufgrund des in Kapitel [9.4.](#) beschriebenen Phänomens leider nicht mehr umsetzbar, da zu wenige Ultraschallsensoren des verwendeten Modells mehr vorhanden waren.

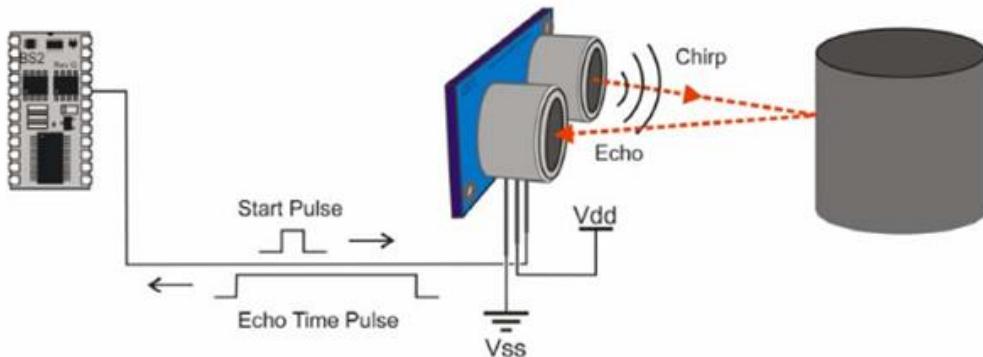
So mit wurden am Ende lediglich zwei Ultraschallsensoren auf dem Auto verbaut. Die ursprüngliche Idee war es, einen Sensor an der Rückseite des Autos mittig und drei Sensoren an der Vorderseite anzubringen, wobei einer dieser drei mittig angebracht worden wäre und die anderen zwei in einem seitlichen Winkel. Bei der Umsetzung wurden dann lediglich ein Ultraschall für beide Fahrtrichtungen verwendet und somit nur Hindernisse erkannt, welche direkt gerade vor oder hinter dem Auto waren.

10.3.1. Funktionalität

Es wurde der Ultraschall Parallax PING Ultraschallsensor verwendet. Dieser ermöglicht es Abstände im Bereich von 3 – 312 cm zu erfassen. Der Ultraschall reagiert nur in einem sehr engen Akzeptanzwinkel und funktioniert nur in einem Temperaturbereich von 0°C bis 70°C nach Angabe des Herstellers. Bezuglich des Akzeptanzwinkels konnte ich während der Test schnell feststellen, dass dies auch der Fall ist.



Die Funktionsweise des Ultraschallsensors ist anhand folgender Grafik gut dargestellt:



Quelle: <https://www.parallax.com/product/28015> unter Downloads & Dokumentationen

Der Ultraschallsensor sendet einen Impuls um die Messung zu starten und wartet dann eine gewisse Zeit auf das Echo. Mit diesem Ergebnis kann dann der Abstand ermittelt werden.

10.3.2. Inbetriebnahme & Programmierung

Die Inbetriebnahme des Ultraschallsensors war, nachdem die Funktionsweise analysiert worden war, sehr einfach für mich. Es musste der PIN, an welchem der Ultraschallsensor angeschlossen wurde, zunächst als Output definiert werden und kurz ein LOW-Signal anliegen und dann auf ein HIGH-Signal geändert werden. Mit dem HIGH-Pulse wird die Abstandsmessung gestartet, da dieses den 40kHz-Ton sendet. Der PIN wird nach diesem HIGH-Signal wieder auf ein LOW-Signal gesetzt und sofort als Input Pin gesetzt, damit das Echo auf diesem empfangen werden kann. Liegt dann auf dem Pin ein HIGH-Signal an. Hier wird dann die verstrichene Zeit zwischen Senden und Empfangen erhalten.

Dieser Wert muss nun noch in den entsprechenden Abstand umgewandelt werden. Hierzu wird die ermittelte Dauer durch die Geschwindigkeit des Ultraschalltons pro Zentimeter geteilt und dieser Wert nochmals durch 2 geteilt, da der Weg ja vom Ultraschallsensor zum Objekt und vom Objekt zum Ultraschallsensor zuerst ermittelt wird.

10.3.3. Stopp-Bedingung

Nachdem der Ultraschall fertig gestellt worden ist. Wurde bei einem Testlauf des Autos von mir die Idee in den Raum gestellt, dass es sinnvoll wäre, das Auto bei einer sehr wahrscheinlichen Kollision, zum Stehen zu bringen. Diese Idee wurde von den Teammitgliedern Dominik Scharnagl und Mehmet Billor als gut empfunden und so gleich mit der Umsetzung dieser Stopp-Bedingung begonnen.



Zuerst wurde eine feste Konstante von 15 cm festgelegt, das bedeutet, wenn der Ultraschall den Wert 15 ermittelt, sollte das Auto stehen bleiben. Schnell wurde diese erste Umsetzung modifiziert und so angepasst, dass das Auto nur stehen bleibt, wenn es sich in die entsprechende Richtung bewegt, in welcher der Ultraschall diesen Sensor auch gemessen hat. Diese Änderung war zwingend erforderlich, damit das Auto nicht bei erstmaliger Erkennung einer Kollision in einem Zustand der Fahruntüchtigkeit stehen bleibt. Somit war es nun für das Auto möglich, nach einem Motorstopp auf Grundlage einer Kollisionserkennung, in die gegengesetzte Richtung weiterzufahren.

Bei den durchgeföhrten Tests mit der zweiten Umsetzungsstrategie, ergab sich schnell ein weiteres Problem. Dies war die Geschwindigkeit des Autos in Kombination mit der Erkennung des Gegenstandes. Bei sehr hoher Geschwindigkeit kollidierte das Auto trotz der entstandenen Stoppbedingung. Hierbei war das Problem, dass auf dem Bodenbelag, auf welchem die Tests durchgeführt wurden, das Auto zulange weiter rutscht. Zu nächste wurde die Konstante zu Erkennung der Kollision höher gesetzt.

Dies empfanden wir, aber nicht als eine schöne Lösung, da dann die Konstante so hoch festgelegt werden müsste, dass bei der höchstmöglichen Geschwindigkeit keine Kollision auftritt. Nach einem Überlegen nach einer Lösung, wurde sich auf die Verwendung der Formel für den Bremsweg bei Gefahrenbremsung entschieden:

$$\text{Bremsweg bei Gefahrenbremsung: } \left(\frac{V}{10}\right)^2 / 10$$

Hiermit wurden die besten Ergebnisse aller versuchten Umsetzungsmöglichkeiten erreicht, wie sich bei mehreren Tests zeigte. So kann bei niedrigerer Geschwindigkeit viel näher an ein Hindernis herangefahren werden, als bei einem höheren Abstand.

10.3.4. Ausblick

Die Verwendung des Ultraschallsensors im Projekt konnte von mir zufriedenstellend abgeschlossen werden. Vor allem die entwickelte Stopp-Bedingung erhöhte das Ergebnis des Projektes in einem kleinen Ausmaß.

Bei einer Weiterführung des Projektes, könnte man weitere Ultraschallsensoren am Auto verbauen, welche seitlich Kollisionen erkennen könnten und somit alle etwaigen Kollisionen ausschließen.

Bezüglich der Stopp-Bedingung könnte man auch einen Schritt weitergehen und das Auto zu einem wirklichen Bremsvorgang auffordern.

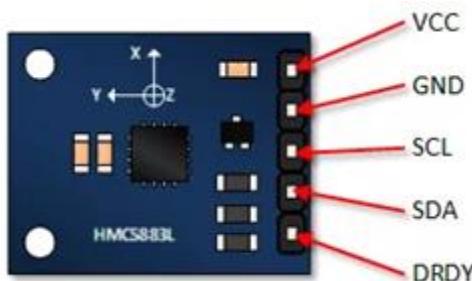


10.4. Kompass Sensor

Ersteller: Annkathrin Bauer

10.4.1. Überblick

Es wurde sich für einen Kompass Sensor entschieden, um die Raum Kartographie zu erleichtern und den UWB-Sensor zu unterstützen.



Der verwendete Sensor ist der Hmc5883l Kompass Sensor, in diesem wurde der Magnetoresitive Sensor von Honeywell verbaut und ist einer der feinfühligsten und genauesten Magnet Sensoren im Nieder-Ampere-Bereich. Dadurch ist der Sensor auf 2° genau und relativ unempfindlich für Störsignale.

Der Sensor wird mit 3,3 Volt betrieben, kann aber auch mit 5 Volt umgehen. Da in diesem Projekt, schlussendlich der Sensor über den Arduino Mega, zuvor über den Arduino Uno und einer externen Batterie betrieben wird, ist die Variante mit 3,3 Volt bevorzugt worden.

Der Hmc5883l besitzt ein I²C digitales Interface, welches die Übertragung der Sensorwerte erleichtert und nur zwei Pins am Board in Anspruch nimmt, ohne Ground und Versorgungsspannung. Diese zwei Pins am Arduino, sind die I²C Schnittstelle des Boards.

10.4.2. Verkabelung

Auf die Verkabelung wird später genauer eingegangen, da sich sowohl der Kompass Sensor als auch der Beschleunigungssensor dieselben Pins am Arduino teilen.

10.4.3. Implementierung und Fehlersuche

Alle grundlegenden Informationen über den Kompass Sensor wurde aus dem Datenblatt geholt. Nachdem mit diesen Werten des Datenblattes₁, eine erste Implementierung des Sensors erfolgte, war das Ergebnis der Sensorwerte relativ ernüchternd. Es kam nur der Wert 0 oder 127 am Arduino Uno an. Da es nah lag das die Implementierung fehlerhaft sein könnte, wurde zum testen eine Bibliothek von Adafruit benutzt und deren Beispielcode auf den Arduino hochgeladen. Dies erbrachte auch nicht den gewünschten Effekt. In dem



Beispielcode kam man nicht einmal soweit, dass man sich die Werte ausgeben lassen konnte, der Code hat sich irgendwo beim holen der Werte aufgehängen und kam nie wieder heraus.

Ab diesen Zeitpunkt wurde es mir bewusst das die Register die im Datenblatt genannt wurden nicht mit den Registern die der Kompass Sensor tatsächlich benutzt übereinstimmt.

Die Tabelle stellt den Unterschied zwischen den Werten des Datenblattes mit den tatsächlichen Werten dar.

Register	Datenblatt Werte	tatsächliche Werte
I ² C Adresse des HMC588I	0x1E	0x0D
Modus Register	0x02	0x09
Control Register	0x01	0x0A
Daten Register Beginn	0x03	0x01

Die tatsächlichen Werte wurden teilweise aus Foren und teilweise durch ausprobieren herausgefunden. Dies war ein langer Prozess aus ausprobieren, kompilieren und schauen ob die Werte Sinn machen. Beim Daten Register Beginn, erwies es sich als besonders schwierig das korrekte Register zu finden, da zuerst ein Arduino Uno benutzt wurde, lag hier der Beginn des Datenregisters bei 0x03. Als auf den Arduino Mega umgestiegen wurde, wurden nur noch Werte im Bereich von 68° bis 79° angezeigt. Das Datenregister musste durch erneutes verwenden der Trail-and-Error-Technique herausgefunden werden, das letztendlich dann bei 0x01 lag. Warum sich der Beginn des Datenregisters durch den Austausch von Uno zu Mega verändert hat, ist bis jetzt noch ein Mysterium.

Mit den richtigen Registern, können nun endlich die Sensorwerte geholt und ausgewertet und der Modus des Sensors auf eine kontinuierliche Messung gesetzt werden. Dies geschieht im „Modus Register“, indem man dort den entsprechenden Wert für die kontinuierliche Messung hineinschreibt. In diesem Fall ist es der Wert „0b01“. Das holen der Werte besteht darin im Datenregister neun Datenbits anzufordern welche die X, Y, Z Werte repräsentieren.

Jedoch kann man die Messwerte des Sensors noch nicht so verwenden, deshalb müssen sie erst skaliert werden. Im Falle der kontinuierlichen Messung wird auf die Rohdaten ein Wert von 0,00024414 multipliziert. Dieser Skalierwert wurde aus dem Forum „forum.arduino.cc“ gefunden, hat funktioniert und wurde nicht hinterfragt.



Um einen Winkel bestimmen zu können, also in welche Richtung der Kompass schaut, wurde der Arcus Tangens aus dem X- und Y-Wert gebildet. Das Ergebnis daraus ist im Bogenmaß, um diese Werte wie bei einem richtigen Kompass ablesen zu können, müssen sie in ein Winkelmaß umgerechnet werden. Bevor das Ergebnis in ein Winkelmaß umgerechnet wird, muss die magnetische Missweisung addiert werden, um ein genaueren Wert zu bekommen.

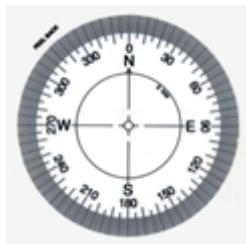
Die magnetische Missweisung ist die Differenz zwischen magnetischen und geographischen Nordpol, diese ist ortsspezifisch und beträgt am Galgenberg in Regensburg $3,16^\circ$ oder $0,05515$ radian.

Mit dem addierten Wert kann man nun das Winkelmaß bestimmen, hierbei genügt die einfache Rechnung:

$$\text{headingDegrees} = \text{heading} * 180/\text{M_PI};$$

Die nun entstandenen Ergebnisse kann man wie einen Kompass ablesen:

- $0/360^\circ$ Norden
- 90° Osten
- 180° Süden
- 270° Westen



Es wurde beschlossen das nur Werte die kleiner als 256 von Arduino zum RaspberryPi verschickt werden sollen, daher gab es kleinere Einschränkungen die das ablesen des Kompasses erschweren. Die Werte werden nun bis 180° verschickt, um trotzdem den Bereich bis 360° abzudecken gibt es negative und positive Werte bis 180° . Vor den eigentlichen Sensorwert wird das Vorzeichen verschickt „0“ steht für positiv und „1“ entspricht einem negativen Vorzeichen. Somit entsprechen die neuen Kompass Werte -180° bis 180° die diesen Himmelsrichtungen zugewiesen werden können:

- $-180/0^\circ$ Norden
- 90° Osten
- $180/-0^\circ$ Süden
- -90° Westen



10.5. Beschleunigungssensor

Ersteller: Annkathrin Bauer

10.5.1. Überblick

Es wurde ein Adxl345 Beschleunigungssensor benutzt um die Geschwindigkeit des Fahrzeugs zu messen, diese Daten werden bei der Raumerkennung mit eingesetzt.



Der Sensor kann mit 3,3 Volt umgehen als auch mit 5 Volt, somit kann man den Beschleunigungs- und den Kompass Sensor mit derselben Stromquelle betreiben.

Der Adxl345 besitzt ein SPI und ein I²C digitales Interface. Aus verschiedenen Gründen wurde sich für den I²C entschieden, welche dabei zwar zu Problemen mit der Kommunikation zwischen Beschleunigungssensor zu Arduino und Kompass Sensor zu Arduino geben könnte. Da der Arduino Uno als auch Mega nur über eine einzige I²C Schnittstelle verfügen, somit müssen sich beide Sensoren eine Schnittstelle am Arduino teilen.

Die Messungen werden in drei Richtungen ausgeführt X,Y und Z, wobei in diesem Projekt die Z Achse außer Acht gelassen wurde. Die Sensoren wurden fest am Auto montiert, die Beschleunigung der Erdanziehung ist verschwindend gering und konstant, um die Auswertung der Daten zu verfälschen.

Die Genauigkeit des Sensors kann man einstellen, die möglichen Werte sind 2g, 4g, 8g und 16g, wobei 2g auf langsamere Geschwindigkeiten ausgelegt sind und 16 g in den High Speed Bereich geht. Alle Messwerte werden in Meter pro Sekunde ausgegeben und können somit sofort benutzt werden.



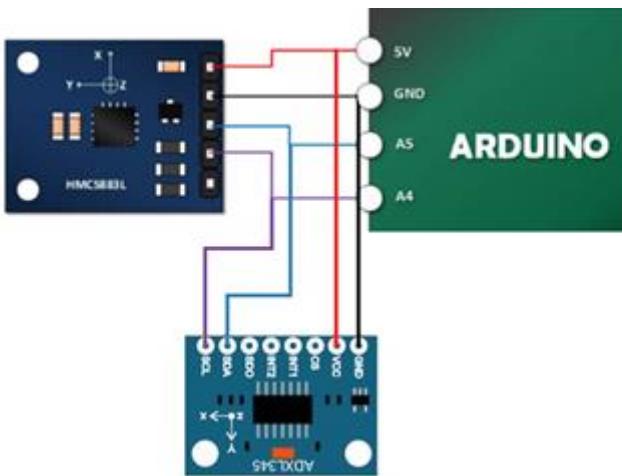
10.5.2. Verkabelung

Wie in der Tabelle zu sehen ist, wurde der Kompass und der Beschleunigungssensor an den selben Pins des Arduino angeschlossen. Dies stellt kein Problem dar, da beide Sensoren eine eigene einzigartige ID haben.

Adxl345 Hmc5883l Arduino Uno Arduino Mega

Vcc	Vcc	3,3V	3,3V
Gnd	Gnd	Gnd	Gnd
Scl	Scl	A5	21
Sda	Sda	A4	20

Durch den Austausch von Uno zu Mega gab es bei dem testen der Sensoren Schwierigkeiten, da angenommen wurde das auch beim Arduino Mega die Pins A4 und A5 I²C fähig sind, kamen dadurch keine Werte von den beiden Sensoren an.



10.5.3. Implementierung und Fehlersuche

Das Vorgehen des Beschleunigungssensors war dasselbe wie beim Kompass Sensor. Die grundlegenden Informationen wurden aus dem Datenblatt₂ übernommen. Die erste Implementierung des Sensors hat nicht funktioniert.

Durch intensive Fehlersuche und stunden langen debuggen ist der Fehler, warum die Sensorwerte nicht übermittelt werden, bzw. nur Null also der Initial Wert angezeigt wird, immer noch unklar. Nachdem schon der Kompass Sensor so viel Zeit gefressen hat und die



erste Vorführung am 15. Dezember immer näher kam, habe ich mich entschlossen die Bibliothek von Adafruit zu benutzen um sicher zu stellen das die Register alle korrekt sind.

Dies führte zu Erfolg und wurde in die Implementierung übernommen, zwecks Zeitmangel.

Nachdem alle zwei Sensoren nun funktionstüchtig sind, wurden beide zusammengeführt um zu testen ob die doppelte Pinbelegung ein Problem darstellt. Es hat sich herausgestellt, dass das kein Problem ist. Jeder der Sensoren hat eine eigene spezielle ID, durch die eine Verwechslung der Daten ausgeschlossen ist. In dem Testprogramm werden nacheinander die Sensorwerte abgerufen mit einem kleinen Delay. Zuerst der Kompass Sensor danach der Beschleunigungssensor, somit ist eine ungewollte zeitgleiche Werteabfrage ausgeschlossen und die Übertragung wird nicht von dem anderen Sensor blockiert.

10.6. Sensoren – Übertragung

Ersteller: Simone Huber, Annkathrin Bauer

10.6.1. Übertragungsansatz

Nachdem alle Sensoren, die mit dem Arduino betrieben wurden, von den jeweiligen Teammitgliedern bearbeitet wurden. War der nächste Schritt diese zusammenzuführen, dies wurde von Annkathrin Bauer und Simone Huber durchgeführt. Wir überlegten uns, wie wir die einzelnen Sensordaten übertragen möchten. Die erste Idee war es, einen Bitframe hierzu zu erstellen. Dieser sollte folgende Struktur aufweisen:

Ultraschallsensor Vorne	Ultraschallsensor Hinten	Magnetsensor	Bewegungssensor
-------------------------	--------------------------	--------------	-----------------

10.6.2. Überprüfung der Sensorfunktionalität

Bevor begonnen wurde einen Bitframe für die Übertragung der Sensorwerte zu erstellen, wurden die einzelnen Sensoren nochmals gemeinsam von uns überprüft und als Vorbereitung für die Konkatenation der Werte umgewandelt.

Die Werte des Ultraschallsensors mussten hierbei nicht bearbeitet werden, da hier durchwegs positive Werte erzielt werden. Wichtig hierbei ist es aber zu berücksichtigen das Werte im Bereich von 3 – 312 auftreten können und somit der Datentyp int verwendet werden muss, um auch größere Werte als 256 darstellen zu können.



Beim Kompasssensor treten wie beim Ultraschallsensor auch größere Werte als 256 auf. Hier wurde aber ein anderes Vorgehen als beim Ultraschallsensor gewählt und der Wertebereich aufgeteilt und von -180 bis +180 festgelegt. Für die Erstellung des Frames wurde hier dann ein Paritybit eingeführt, welches angibt, ob ein positiver Wert oder ein negativer Wert vorliegt.

Beim Beschleunigungssensor war es auch zwingend notwendig für die Erstellung des von uns geplanten Bitframes eine Veränderung der Rückgabewerte durchzuführen. So wurde auch hier wieder ein Bit eingeführt, welches Aussage über positiv oder negativ trifft. Sowohl musste dies vor den X-Wert als auch für den Y-Wert eingeführt werden.

Nach dieser Modifizierung der Sensorwerte wurden wiederholt einige Tests durchgeführt, ob die Sensorwerte korrekt dargestellt werden und mit den alten Programmen überprüft. Diese Tests waren zufriedenstellend und somit waren die Sensorwerte-Darstellung beendet.

10.6.3. Erstellung des Bitframes

Hier wurden mehrere Varianten versucht, nachdem aber keine Variante zu einem schnellen und zufriedenstellenden Ergebnis geführt haben, wurde die Idee der Erstellung eines Bitframes wieder verworfen.

Die erste sehr umständliche Idee war es, die entsprechenden Binärzahlen über eine Funktion zu berechnen. Anschließend die führenden 0en hinzuzufügen und am Ende die einzelnen binären Zahlen aneinander zu reihen und als String abzuspeichern. Die Umwandlung der Dezimalzahlen der Werte in Binärzahlen war leicht über eine Funktion realisierbar. Auch das Auffüllen der Binärzahlen mit 0en war noch leicht umsetzbar. Das eigentliche Problem ergab sich erst bei der Zusammenführung der Binärzahlen in einen String. Da bei einem String die 0 dem Ende entspricht, war es nicht möglich auf diese Weise einen Bitframe zu erstellen.

Einige weitere Versuche führten auch zu keinem Ergebnis, dass dem gewünschten Format entsprach. Da die erste Zwischenpräsentation anstand, entschlossen wir uns, die Idee der Erstellung eines Bitframes zu verwerfen. Nun wurden die einzelnen Sensorwerte in der in Abschnitt b. erstellten Schema in das zu diesem Zeitpunkt verwendete Protokoll von Robert Graf, unter dessen Hilfestellung eingebunden und somit einzeln übertragen.



10.6.4. Bildung einer Struktur

Letzten endlich wurden die Sensorwerte, dann als Struktur angelegt und mit der Ersatzlösung für die Übertragung zwischen Arduino und Raspberry Pi 3 übertragen.

Die Probleme die sich hierbei zeigten, werden im Kapitel [14.5.3.](#) genauer aufgeführt.

Lediglich der Kompasssensor schien bei diesem Test keine sinnvollen Werte ausgeben. Dies lag aber an dem Datenregister des Kompasssensors, welches bereits im Abschnitt [10.4.](#) Kompass genau erklärt wurde.

Bei späteren Tests wurden die Sensorwerte weiterhin wie gewünscht übertragen und auch am Display angezeigt und somit die Thematik der Sensordatenübertragung abgeschlossen.

10.7. UWB

Ersteller: Simone Huber

10.7.1. Anforderungen

Bei der Festlegung des Projektes war geplant einen Ultra-Widebandsensor in das Projekt zu integrieren, welcher den Standpunkt im Raum des Fahrzeugs erfassen soll. Dies wäre für die Umsetzung einer genauen Kartografie notwendig gewesen, aber auch ohne diesen kann eine Raumerkennung erstellt werden, da dennoch einige Sensoren auf dem Fahrzeug verwendet werden können, um einen Überblick über Entfernung zu Gegenständen und der Orientierung im Raum zu liefern.

10.7.2. Berechnung und Funktionalität

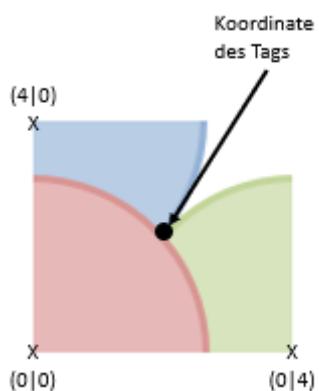
Zuerst habe ich mich der Funktionalität des UWBs befasst und den nötigen Aufbau analysiert. Hierzu habe ich mich zuerst in die Dokumentation eingelesen und eine Recherche im Internet durchgeführt. Hierbei war sofort zu erkennen, dass diese Technologie zu den neueren ihrer Kategorie zu zählen ist. So gibt es im Vergleich zu anderen Sensoren, in diesem Bereich vergleichsweise wenig Informationen.

Der grundsätzliche Aufbau des Ultra-Widebandsystem ist es, mit drei Anchor und einem Tag zu arbeiten. Es werden die drei Anchor in drei Ecken eines Raumes aufgestellt und der Tag auf dem Auto angebracht. Ziel ist es aus den drei ermittelten Entfernungen ein Koordinatensystem zu entwickeln und die Lokation des Autos in diesem festzulegen.

Der Tag kommuniziert mit den einzelnen Anchors und ermittelt so die Entfernung zu diesen. Die Nahbereichsfunkkommunikation ermöglicht dies. Somit werden drei



Entfernung ermittelt, welche bei der Berechnung umgewandelt werden müssen, um ein Koordinatensystem zu bilden.



In der nebenstehenden Grafik ist der grundsätzliche Aufbau erkennbar. Die drei Anchor werden in drei Ecken des Raumes positioniert und ihnen wird bei der Programmierung ein Punkt im Koordinatensystem zugewiesen. Hierzu wird ein Anchor als Ursprung gewählt und die beiden anderen jeweils entweder auf der x-Achse mit der Entfernung zum Ursprung bzw. in der y-Achse festgelegt. Somit wurde ein Koordinatensystem für den Raum festgelegt.

Der Tag ermittelte die einzelnen Entfernungen zu den Anchor.

Diese Entfernung wird zur Berechnung der Lokalisation des Tags so umgewandelt das die ermittelte Entfernung, als Radius interpretiert wird und um den Anchor herum ein Kreis mit dem entsprechenden Radius gebildet wird.

Würde der Tag zu 100% die korrekten Werte des Abstandes zu den einzelnen Anchor ermitteln, würden sich die drei Kreise um die Anchor, mit ihrem jeweils eigenen Radius, in einem einzelnen Punkt schneiden, welcher dann dem Standort des Tags entspricht.

Ohne einen Test durchzuführen, ist hier aber ersichtlich, dass diese 100ige Genauigkeit nicht eintreten wird, daher muss die Berechnung des Standpunktes modifiziert werden. Hierzu muss das Berechnungsmodell so verändert werden, dass dennoch ein ungefährer Schnittraum der Kreise bestimmt wird. Dies wäre möglich, wenn man die Schnittpunkte der einzelnen Kreise mit dem Schnittpunkt der anderen beiden Kreise einzeln ermittelt und dann einen Mittelwert der Ergebnisse bildet und dieses dann als Standort des Tags annimmt.

10.7.3. Grundeinstellungen der DW1000-Module

Nach einiger Zeit habe ich festgestellt, dass die einzelnen Module größtenteils unterschiedlich eingestellt waren. Daher befasste ich mich zunächst genauer mit den einzelnen verbauten Komponenten auf den Modulen.

So stellte ich bei allen Modulen, welche als Anchor fungieren sollte, erst einmal entsprechend der Kurzanleitung, welche TREK1000 beigelegt ist, die Stromversorgung so ein, dass diese über eine externe Stromquelle, läuft.

Des Weiteren stellte ich drei DW1000-Module als Anchor ein und einen als Tag. Auch dies war, für die Verwendung mit dem Standardprogramm von DecaWave, in dieser Kurzanleitung erklärt.



10.7.4. Inbetriebnahme und Probleme

Nachdem sich ausführlich mit der Analyse der Funktionsweise des UWBs beschäftigt worden ist, war der nächste Schritt diesen an den Raspberry Pi 3 anzuschließen. Hierzu wurde sich zunächst mit dem Raspberry Pi 3 vertraut gemacht und dann festgelegt, dass die bestmögliche Anschlussmöglichkeit über die USB-Schnittstelle des Raspberry Pi 3 ist. Hierzu würde ein USB-Kabel am Raspberry Pi 3 angeschlossen und mit dem UWB verbunden.

Bevor aber das DW1000-Modul an den Raspberry Pi 3 angeschlossen werden sollte. War es erst mal mein Ziel diesen per USB-Kabel an einen gewöhnlichen Laptop anzuschließen. Hier ergab sich schnell das erste Problem, dass es nicht möglich war das DW1000-Modul auf einem Apple-Laptop anzusprechen. Hier wurde eine genaue Fehlersuche durchgeführt, bei welcher mit der Überprüfung der Einstellung des Moduls gestartet wurde. Hier war schnell erkennbar, dass die einzelnen DW1000-Module alle unterschiedlich eingestellt waren, unter Vernachlässigung der Einstellung, ob das Modul ein Tag bzw. ein Anchor ist und ob dieses über das USB-Kabel mit Strom versorgt wird oder über eine externe Stromquelle.

Nach einigen Recherchen, warum der USB-Port für das Modul nicht auf dem Mac erkennbar ist, kam ich zu dem Entschluss auf einen Laptop mit Windows-Betriebssystem zu wechseln. Da auch in einer auf dem Mac laufenden VM, der entsprechende USB-Port nicht erkannt wurde.

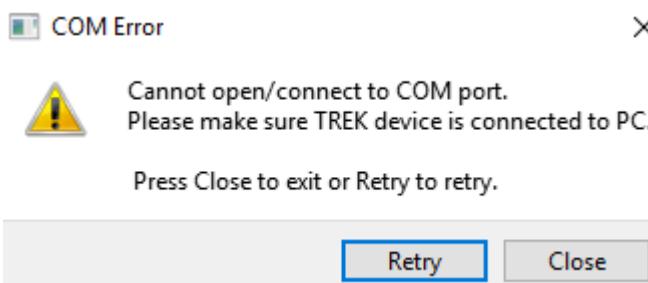
Auf dem Laptop mit Windows-Betriebssystem wurde der USB-Port in den Systemeinstellungen zu mindestens angezeigt.

Somit wurde als nächstes versucht das DW1000-Modul auf dem Raspberry Pi 3 anzuschließen, auch hier wurde dann der USB-Port sofort erkannt. Somit war das erste Problem gelöst. Mittels CuteCom wurde versucht, ob bei der entsprechenden Datenrate und allen nötigen Eingabewerten in diesem Programm, überhaupt Daten auf dem Raspberry Pi 3 empfangen werden.

Man konnte sehen, dass nichts angezeigt wird auf dem Raspberry Pi 3. Nach einiger Zeit an Recherche im Internet, wurde ich aber nicht fündig, wie ich mein Problem lösen könnte. Dies ist sicherlich auch der Fall, da die im Internet gefunden Beispiele, welche die gleiche Technologie verwenden wie die DW1000-Module anders aufgebaut waren.

Ich versuchte mir anhand dieser aber einen weiteren Bearbeitungsvorgang zu erarbeiten. Leider schienen alle meine Versuche, doch noch Informationen vom DW1000-Modul zu bekommen, ins Leere zu verweisen.

Somit befragte ich andere Studenten aus höheren Semestern, ob mein Vorgehen, das DW1000-Modul an den Raspberry Pi 3 mittels USB-Kabel anzuschließen, überhaupt der richtige Ansatzpunkt ist. Mir wurde aber bestätigt, dass dies die in den letzten Semestern verwendete Variante ist.



Somit war ich mir zumindest sicher, dass die Grundidee meines Vorgehens richtig ist. Um nicht weiter irreführende Optionen durchzuführen, holte ich mir Verstärkung von einem befreundeten Informatiker. Dieser

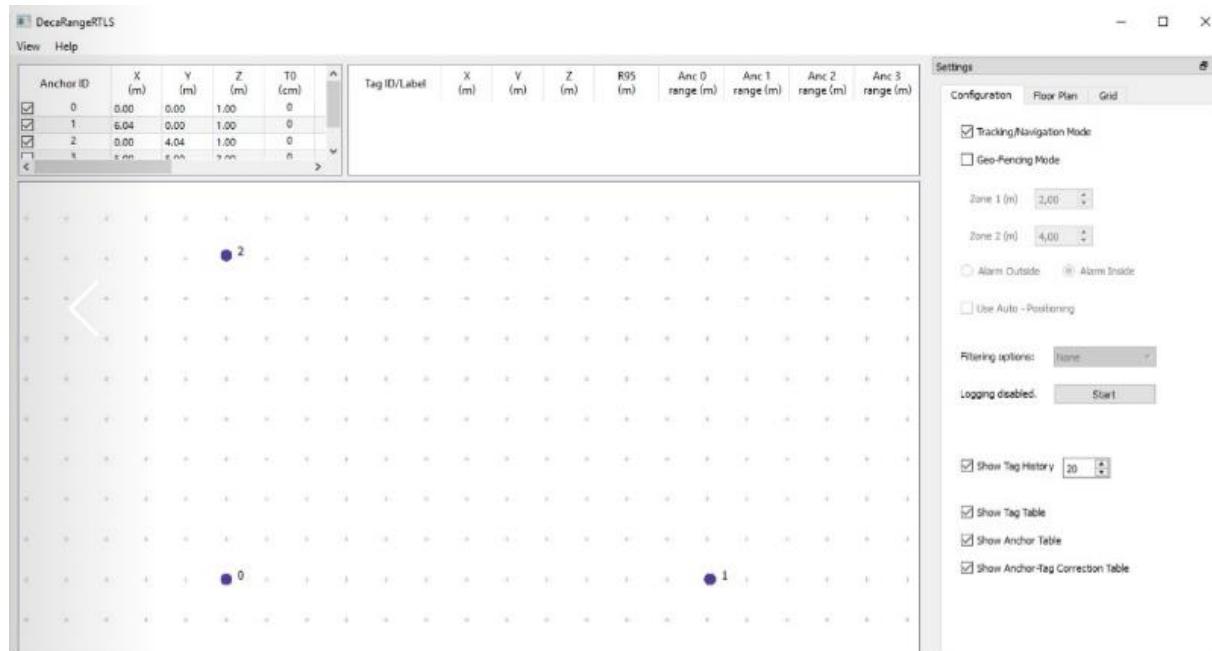
arbeitete bereits in vielen Projekten mit dem Raspberry Pi und ist sehr an ihm neue Thematik interessiert. Gemeinsam versuchten wir, dann das DW1000-Modul mittels eines im Internet gefunden, von DecaWave bereitgestellten, Programm anzusprechen.

Hierbei trat erneut der Fehler auf, dass es Probleme mit dem USB-Port gibt, mit welchem das Modul angesprochen werden sollte.

Dieses Problem konnte nach einiger Recherche mit Hilfe folgender Seite gelöst werden:

<https://sourceforge.net/projects/realterm/>

Somit konnte das Programm nun gestartet werden, insofern ein DW1000-Modul mittels USB-Kabel mit dem Laptop angeschlossen ist und die Grundeinstellungen des Herstellers auf dem Modul eingestellt waren.



Trotz nun exakt eingestellter DW1000-Module, wie vom Hersteller in der Kurzanleitung angegeben, und aller richtig positionierten Anchor, wurde auch mittels des Programms keine Daten, wie vom Hersteller in einem Einführungsvideo gezeigt, dargestellt.

(<https://www.decawave.com/video/trek1000-quick-start-video>)



Nach einigen weiteren Stunden des Testens, kam ich zu dem Entschluss, dass es für mich nicht möglich ist, die DW1000-Module so anzusprechen, dass ich die gewünschten Informationen erlange.

Kurzzeitig versuchte auch noch das DW1000-Moduls mit dem Arduino zu verbinden, aber auch hier erhielt ich keine Informationen und verwarf diese Idee relativ zügig wieder.

Somit teilte ich dem Team mit, dass die Realisierung des UWBs im Rahmen des Projektes nicht stattfinden wird. Da hier keine weiteren Beschwerden aufkamen, teilte ich mit, dafür die restliche Zeit beim Testen zu helfen, sowie die Zusammenführung der Präsentation und Dokumentation durchzuführen.

10.7.4. Fazit

An und für sich finde ich persönlich die Technologie, mit welcher ich versucht habe zu arbeiten, als hochspannend, aber dennoch auch sehr komplex. Abschließend ist damit zusagen, dass ich bei der Entscheidung mit den DW1000-Modulen zu arbeiten meine eigenen fachlichen Kompetenzen überschätzt habe.

Dennoch kann ich nur empfehlen, sich mit dieser Thematik zu beschäftigen. Für andere Gruppen empfehle ich, die Arbeit an diesem Sensor in mindestens einem Zweier-Team zu bearbeiten und fundierte Grundlagen bezüglich SPI und deren Anpassungsmöglichkeiten auf dem Raspberry Pi 3 zu haben.

10.8. LIDAR

Ersteller: Anja Strobel

10.8.1. Spezifikationen

In diesem Projekt kam ein URG-04LX-UG01 LIDAR der Firma Hokuyo Automatic zum Einsatz. Dieser Sensor verfügt über ein Messgebiet von 20 bis 5600 mm bei einem Sichtwinkel von 240°. Die Messabweichung liegt bei einer Distanz von unter 1m bei ±30mm und bei einer Distanz bis 4,095 m bei ±3%. Die Messzeit beträgt 100ms/Messung, somit können bis zu 10 Aufnahmen pro Sekunde gemacht werden. Die Datenübertragung und Stromversorgung erfolgt über USB (5V).

(<https://www.hokuyo-aut.jp/search/single.php?serial=166>)



10.8.2. Inbetriebnahme

Erste Tests dieses Sensors wurden mit einer vom Hersteller zur Verfügung gestellten Software (UrgBenri) unter Windows durchgeführt. Bei diesen Tests wurde die Funktionstüchtigkeit und Genauigkeit überprüft. Um den Sensor unter Windows zu verwenden muss ein vom Hersteller gestellter Treiber installiert werden.

Des Weiteren konnte in Erfahrung gebracht werden, dass der Sensor einen einstellbaren Messwinkel besitzt und bei voller Ausnutzung dieses Winkels 682 x,y-Wertepaare liefert. Der Nullpunkt liegt in der Mitte des Sensors.

Anschließend wurde ein C++ Programm zur Ausführung unter Windows mit Hilfe von bereitgestellten Beispielen erstellt.

10.8.3. Verwendung

Der LIDAR wird direkt an den Raspberry Pi angeschlossen und war dort /dev/ttyACM0 zu finden. Der Sensor wird über dieses Device identifiziert. Falls der Sensor als ein anders Device registriert ist, muss dieser Wert im Sourcecode angepasst werden.

Um den LIDAR unter Linux zu verwenden muss die Herstellerlibrary mit "make" und "make install" installiert werden. Diese erzeugt Library files im Ordner "/usr/local/include/lib" ab. Dieser Pfad muss in der Konfiguration des Linkers hinzugefügt werden. Dies geschieht indem der Pfad in der Datei "/etc/ld.so.conf" angefügt wird. Danach muss die List der Libraries mit dem Kommando "ldconfig" neu geladen werden. Hier ist zu überprüfen, dass ein Eintrag mit "liburg" vorhanden ist.

Nun kann der Sensor, unter Verwendung der zur Verfügung stehenden Libraries, durch ein Programm angesprochen werden.

10.8.4. Sourcecode

Das Programm zur Steuerung des Sensors ist in C++ geschrieben. Es stellt eine Methode bereit um eine Messung am LIDAR Sensor auszulösen und die Werte dieser Messung in einer Datei abzuspeichern. Es ist möglich Debug Messages über #define Direktiven zu de-/aktivieren.

Zu Beginn muss das Device identifiziert werden und die Baudrate eingestellt werden.

```
const char connect_device[] = "/dev/ttyACM0";
const long connect_baudrate = 115200;
```

Anschließend werden die benötigten Variablen initialisiert. Der Sensor kann danach mit dem Befehl "urg_open" aktiviert werden. Diesem Befehl muss ein in der Library zur Verfügung gestellter Enumerationstype übergeben werden, der den Verbindungstyp beschreibt (hier serielle Kommunikation). Um das Messergebnis abspeichern zu können,



muss Speicher in der benötigten Größe bereitgestellt werden. Die Speichergröße errechnet sich aus der Länge des Datentyps (hier long) und der maximalen Anzahl von Datenpunkten. Mit dem Befehl "urg_start_measurement" wird die Messung ausgeführt, "URG_DISTANCE" gibt an, dass nur die Entfernungsdaten zurückgegeben werden sollen. Mögliche Messfehler werden in der Variable "ret" abgespeichert und können im Bedarfsfall ausgegeben werden.

```
urg_t urg;
int ret;
long *length_data;
int length_data_size;

ret = urg_open(&urg, URG_SERIAL, connect_device, connect_baudrate);

length_data = (long *)malloc(sizeof(long)* urg_max_data_size(&urg));

ret += urg_start_measurement(&urg, URG_DISTANCE, 1, 0);
```

Die erfassten Daten werden mit der folgenden Funktion angefordert, die Daten werden in der übergebenen Variable "length_data" gespeichert, während die Anzahl an Datenpunkten in als return - Wert zur Verfügung stehen.

```
length_data_size = urg_get_distance(&urg, length_data, NULL);
```

Im Folgenden werden die Daten weiterverarbeitet. Es werden aus den gemessenen Distanzen x - und y- Koordinaten berechnet, die in Strings zusammengefügt werden. Um diese Korrdinaten zu berechnen, wird der Messwinkel der Messung benötigt. Dieser kann über den Index des Messpunkts mit "urg_index2rad" bestimmt werden.

```
std::string xValues = "";
std::string yValues = "";

for(int i=0; i < length_data_size; ++i)
{
    double radian;
    long length, x, y;

    radian = urg_index2rad(&urg, i);
    length = length_data[i];
    *(xVal +i) = (long)(length * sin(radian));
    *(yVal +i) = (long)(length * cos(radian));

    xValues += std::to_string(*(xVal+i)) + ", ";
    yValues += std::to_string(*(yVal+i)) + ", ";
}
```

Die in Strings abgespeicherten Daten werden in eine Datei geschrieben. Zusätzlich wird in die erste Zeile des Files ein numerische Wert geschrieben, der indiziert, ob eine Messung fehlerfrei war. Werte ungleich 0 geben an, dass es ein Problem bei der Messung gab. Dies ermöglicht es diese Datensätze bei der Auswertung zu ignorieren.



```
std::ofstream file;
file.open(file_name);
file << std::to_string(ret) + "\n";
file << xValues + "\n";
file << yValues + "\n";
file.close();
```

Am Ende der Methode wird der benötigte Speicher freigegeben und die Verbindung zum Sensor geschlossen.

```
urg_close(&urg);
free(xVal);
free(yVal);
```

Diese Methode wird als Library für das Hauptprogramm zur Verfügung gestellt. Bei der Kompilierung dieses Programms ist zu beachten, dass C++ Version 11 benötigt wird.

10.8.5. Beobachtungen

Bei Testmessungen in kontrollierter Umgebung wurde festgestellt, dass vom Sensor teils fehlerhafte Daten geliefert werden. So werden teilweise Messpunkte mit Koordinaten (0,0) zurückgegeben, die indizieren, dass sich ein Gegenstand direkt am Sensor befindet, obwohl nur ein Gegenstand in weiterer Entfernung zu finden ist. Hierbei wurde darauf geachtet, dass sich der Gegenstand im definierten Messbereich befindet. Dies trifft allerdings nur auf einige Messpunkte zu, während angrenzende Messpunkte den Gegenstand richtig verzeichnen. Dieses Verhalten tritt allerdings zufällig und bei mehreren Messungen an verschiedenen Messpunkten auf. Weiter konnte beobachtet werden, dass die Oberfläche des Gegenstands etwas Einfluss auf dieses Verhalten hat. Glatte aber nicht spiegelnde Oberflächen führen zu weniger Messfehlern als spiegelnde (Spiegel, glanzlackierte Oberflächen) oder unregelmäßige (gewellte Pappe in kurzer Entfernung) Oberflächen. Es konnte auch beobachtet werden, dass die gemessenen Werte einem Rauschen unterliegen. So schwanken die Messwerte teils um einige Millimeter.



11. Steuerung

Ersteller: Dominik Scharnagl

Im Rahmen des Projektes soll sich das Fahrzeug manuell steuern lassen. Dafür vorgesehen waren zu Beginn des Projektes ausschließlich die eZ430-Chronos Watches von Texas Instruments. Nachdem es aber zu Verzögerungen im Entwicklungsfortschritt bei der Implementierung der Firmware der Uhren wie auch der Integration dieser in das System des Fahrzeugs kam, entschlossen wir uns dazu, eine weitere – alternative – Steuerung vorzusehen. Diese Entscheidung führte dazu, dass wir, auch aus Gründen anstehender Funktionstests, eine weitere Fahrzeugsteuerung mittels Xbox 360 Controller festlegten. Diese wenn auch kabelgebundene Variante der Steuerung ermöglichte es uns, unabhängig vom Fortschritt der Uhrensteuerung mit der Entwicklung am Projekt fortzufahren und auch beim Termin der ersten Vorführung erste Ergebnisse nachzuweisen.

11.1. Steuerung mittels Xbox 360 USB Controller

Der für die alternative Steuerung vorgesehene Xbox 360 USB Controller war zu Beginn und zum Bestellzeitraum (für etwaige fehlende Komponenten) nicht als Teil des Projektes geplant und wurde dem Projekt deshalb aus privaten Mitteln von Dominik Scharnagl zur Verfügung gestellt.

Zum Wechsel in die Steuerung mit dem Controller muss dieser am USB Host Shield des Arduino angeschlossen, der Arduino (zur Erkennung des neuen USB Gerätes) neugestartet und auf dem Raspberry Pi der Steuerungsmodus auf „Controllersteuerung“ geändert werden.

11.1.1. Anforderungen

Der Controller soll es ermöglichen, das Fahrzeug mittels linkem Stick zu lenken und wahlweise entweder mit dem rechten Stick oder mit den Triggern RB / LB in / gegen die Fahrtrichtung zu beschleunigen. Weiter soll es möglich sein, das Fahrzeug in einen „fahrbereiten“ Zustand zu versetzen und auch wieder aus diesem „herauszuholen“, indem man den „Motor“ ein- und wieder ausschalten kann. Erst wenn der Motor eingeschaltet, also das Fahrzeug im „fahrbereiten“ Zustand ist, soll mit dem Controller das Fahrzeug gesteuert werden können. Die für den Controller nötige Steuerungslogik soll dabei in einem eigenen „StarModule“ namens „StarController“ implementiert und gekapselt werden.



11.1.2. Analyse / Design



Während der Analyse der Funktionen des Controllers, welche bereits durch den privaten Einsatz im Groben bekannt waren, stellten wir fest, dass der „Ring“ um den „X“-Button in der Mitte des Controllers, der in vier viertelkreisförmige Segmente geteilt ist, diverse Beleuchtungsmodi bereitstellt. Wir entschieden uns dafür, dass wir den Modus „Beschleunigung mittels rechtem Stick“ mit dem oberen linken und den Modus „Beschleunigung mittels LB / RB Triggern“ mit dem oberen rechten Segment signalisieren wollen. Je nachdem, welcher Modus gerade aktiv ist, soll das zugehörige Segment leuchten und das entsprechende andere Segment nicht leuchten. Standardmäßig ist der Modus „Beschleunigung mittels rechtem Stick“ aktiv. Ein Wechseln zwischen den Modi soll über einen einmaligen Druck des „X“-Buttons durchgeführt werden (siehe Abbildung unten).



Zusätzlich zu diversen weiteren Eingabeelementen besitzt der Xbox Controller farbige Aktionstasten, die mit den Buchstaben X, Y, A und B beschriftet sind. Nachdem unser Motor über den Controller ein- und wieder ausschaltbar sein soll, legten wir uns darauf fest, dass mittels Druck des X-Taste der Motor ein- und durch einen wiederholten Druck wieder ausgeschaltet werden können soll (siehe Abbildung oben).

Bei der weiteren Untersuchung der Eigenschaften des Controllers stellten wir fest, dass der Controller zwei kleine Vibrationsmotoren enthält, von denen einer ein größeres „Unwucht“-Gewicht als der andere besitzt. Nachdem wir es uns zum Ziel gesetzt hatten, dem Benutzer so viel Feedback wie möglich zu geben, haben wir die Vibrationsmotoren als Feedback beim Ein- und Ausschalten des Motors vorgesehen. Während das Einschalten des Motors durch Vibrationen mit beiden Motoren für eine Sekunde repräsentiert werden soll, soll das





Ausschalten des Motors über eine halb so lange und halb so starke Vibration mit beiden Vibrationsmotoren signalisiert werden.

Als Treiber zur Ansteuerung des Controllers orteten wir schnell den „XBOXUSB“ Treiber der „USB Host Shield Library 2.0“. Über das im Open Source Projekt (https://github.com/felis/USB_Host_Shield_2.0) befindliche Beispielprojekt (https://github.com/felis/USB_Host_Shield_2.0/blob/master/examples/Xbox/XBOXUSB/XBOXUSB.ino) zur Xbox Controller Steuerung konnten wir weitere Randdaten des Controllers feststellen und einordnen. Dazu gehörte, dass die Sticks einen Steuerungswert aus dem Intervall [-32768, 32768] jeweils in vertikale sowie in horizontale Richtung liefern. Ein negativer Wert beschreibt dabei eine Stickbewegung zum Fahrzeugführer hin oder nach links. Ein positiver Wert definiert wiederum eine Stickbewegung vom Fahrer weg oder nach rechts. In Neutralstellung liefert der Treiber den Wert 0.

Die weiteren Experimente und Code-Reviews am Beispielprojekt brachten uns zur Erkenntnis, dass die Trigger LB und RB jeweils einen Steuerungswert aus dem Intervall [0, 255] liefern. Während die 0 eine Neutralstellung des Triggers beschreibt, definiert ein Wert größer 0, wie weit der Trigger in den Controller gedrückt wird. Beim Drücken bis „zum Anschlag“ des Triggers erhält man den Steuerungswert 255.

11.1.3. Implementierung

Zum Anschluss des Xbox USB Controllers wurde das USB Host Shield des Arduino verwendet. Als Treiber zur Ansteuerung des Controllers wurde die „XBOXUSB“-Klasse der „USB Host Shield Library 2.0“ verwendet. Die Implementierung der nötigen Logik zur Interaktion mit dem Controller begann erst, nachdem die volle Funktion der Servo- wie auch Motorsteuerung des Fahrzeugs gegeben war.

Erste Funktionstests mittels vom Entwickler der Library verfügbaren Beispielanwendung konnten wir auch problemlos durchführen und begannen somit, die Hülle der bereits vorbereiteten „StarController“-Klasse Schritt für Schritt mit der notwendigen Logik auszufüllen. Die ersten in den Anforderungen gewünschten (einfachen) Funktionen wie die Lenkung waren dank der Beispielanwendung schnell in das Projekt integriert.

Xbox Controller lässt sich nicht mehr ansprechen

Nachdem wir die ersten Schritte der Implementierung mittels Funktionstests validieren wollten, stellten wir fest, dass die Ansteuerung nicht wie in der Beispielanwendung verifiziert funktioniert. Während wiederholte Tests mit der Beispielanwendung weiterhin funktionierten, funktionierte selbst 1:1 kopierter Code in unserem Projekt nicht. Dies brachte uns dazu, dass wir unbewusst veränderte Projekteinstellungen oder andere Änderungen am Code für dieses Verhalten verantwortlich machten und versuchten, die vorgenommenen Änderungen nach bestem Wissen rückgängig zu machen. Unser



anfänglicher Leichtsinn - viele Änderungen ohne regelmäßige Commits in unser Git - führte schließlich dazu, dass wir unser Projekt komplett entleerten und erst einmal den Beispielcode 1:1 übernahmen. Nachdem wir sichergestellt hatten, dass der Beispielcode in unserem Projekt funktionierte überführten wir Schritt für Schritt den Code in die von uns gewünschte Struktur der „StarModules“, bis wir die Ursache mittels paralleler Funktionstests ausfindig machen konnten. So stellten wir fest, dass die vom Entwickler der „USB Host Shield Library 2.0“ gewählte Klassenarchitektur es uns nicht erlaubte, eine Heap-Referenz auf den generischen USB Treiber für den Co-Prozessor MAX3421E des USB Host Shields zu verwenden. Der vom „XBOXUSB“-Treiber benötigte „Kernel“-Treiber muss somit immer auf dem Stack allokiert werden. Bis diese Ursache aber gefunden war, mussten wir einige Male immer wieder in kleinen Schritten unsere Codestruktur auf- und wieder abbauen. Nachdem die Ursache gefunden und behoben war, konnten wir ohne weitere Probleme – dieser Größenordnung – mit der Implementierung fortfahren.

Das Ein- und Ausschalten des Motors mittels „X“-Button-Druck und die geforderten Vibrations-Effekte als Feedback konnten schnell und ohne Probleme realisiert werden. Eine der letzten Funktionen war die Steuerung mittels Trigger-Buttons, da wir einige Zeit in den Standardmodus der Steuerung mittels Sticks investiert hatten. Hier wollten wir sicherstellen, dass die Steuerung mit den Sticks zum einen so direkt wie möglich und zum anderen so fehlerfrei wie möglich abläuft. Obwohl wir während dieser Phase ein leichtes Fehlverhalten bei der Steuerung des Fahrzeugs aufgrund von Nachschwingungen der Sticks feststellten, haben wir uns dafür entschieden, dieses nicht als Störfaktor bei der weiteren Realisierung des Projektes einzustufen. Deshalb verwendeten wir keine weitere Zeit bei der Optimierung der Steuerung mittels Sticks.

Während des Übergangs der Implementierung des einen Modus zur Implementierung des anderen Modus wurde der dafür notwendige Moduswechsel mittels „Xbox“-Button realisiert. Stets wiederholte Funktionstests stellten dabei durchgehend die gewünschte Funktion der Steuerung sicher.

Da bereits bei der Implementierung der Servo- beziehungsweise Motorsteuerung bekannt war, dass die Steuerung des jeweiligen Aktuators in Prozentwerten im Intervall [-100, 100] ausgedrückt und durchgeführt wird, musste die Vorbedingung entsprechend berücksichtigt werden. Daher haben wir bei der Steuerung mittels Sticks den Wertebereich [-32768, 32768] sowie Wertebereich [0, 255] der Trigger auf das Intervall [-100, 100] umgelegt. Eine Besonderheit haben wir dabei bei der Steuerung mittels LB und RB implementiert.



Da LB für die Beschleunigung entgegen und RB für die Beschleunigung in Fahrtrichtung zuständig sind, wird der Steuerungswert der Beschleunigung im Modus „Steuerung mittels Trigger“ wie folgt ermittelt:

```
// Determine speed using positive and negative press intensity.  
int_t speed = this->xboxController->getButtonPress(R2); // RB  
speed += -this->xboxController->getButtonPress(L2); // LB  
  
car->setSpeed(((float_t)speed / 255.0) * 100);
```

Dadurch, dass die Beschleunigung in Fahrtrichtung positiv und die Beschleunigung entgegen der Fahrtrichtung negativ in die Berechnung des Steuerungswertes zur Beschleunigung eingeht, ist es möglich, das Fahrzeug mit einer konstanten Vorfahrtsbeschleunigung zu fahren, zugleich jedoch die Beschleunigung durch Drücken des „Rückwärtsfahrt“-Triggers LB abzubremsen, aufzuheben oder gar in eine Beschleunigung entgegen der Fahrtrichtung umzukehren.

11.1.4. Test

Tests am Fahrzeug wurden bereits von Anfang an und während des gesamten Projektverlaufs durchgeführt. Diese beschränkten sich stets auf Funktionstests. Der Umfang dieser Tests beinhaltete mindestens das Lenken des Fahrzeugs. Ausführlichere Tests wie der Moduswechsel, das Ein- und Ausschalten des Motors, die damit verbundene Vibration als Feedback sowie die Beschleunigung wurden immer wieder zusammen mit anderen Komponenten des Fahrzeugs getestet. Ein besonderer Vorteil dieser parallelen Tests, auch wenn es nicht direkt um den Controller ging, war, dass man schnell einen Absturz des Arduinos nachweisen konnte, denn genau in diesem Moment gingen alle Segmente des „Xbox“-Button Rings aus beziehungsweise beim Hochfahren alle Segmente kurz an. Hinzu kam, dass durch den Standardmodus nach erfolgreicher Setup-Sequenz des Arduinos das obere linke Segment des „Ringes“ leuchtete, was ausschlaggebend für den Verlauf der Tests war.

11.2. Steuerung mittels eZ430-Chronos-Watch

Die Anfangs mittels eZ430-Chronos-Watch vorgesehene primäre Steuerung wurde aufgrund der Startschwierigkeiten bei der Einarbeitung in die Plattform zur sekundären Steuerung. Zur Steuerung des Fahrzeugs werden zwei eZ430-Chronos-Watches *Black* und ein eZ430-Chronos-AccessPoint *White* benötigt. Die Watches müssen dabei mit der von Dominik Scharnagl entwickelten Firmware „Carmotion“ geflasht sein. Zugleich muss das Projekt „ez430_chronos_access_point“ – hier „Carcontrol“ genannt – auf dem AccessPoint geschrieben sein. Die dafür notwendigen Hardwarekomponenten wurden im Rahmen der



Veranstaltung „Algorithmen für Sensornetze“ bereitgestellt und wieder an den Dozenten Herrn Volbert zurückgegeben. Zur einfacheren Handhabung (dem An- und Abstecken des Xbox 360 USB Controllers wie auch des AccessPoints) wurde dem Projekt aus privaten Mitteln von Dominik Scharnagl ein USB3.0-L-Stück für den AccessPoint zur Verfügung gestellt.

Zum Wechsel in die Steuerung mit den Uhren muss der AccessPoint am USB Host Shield des Arduino angeschlossen, der Arduino (zur Erkennung des neuen USB Gerätes) neugestartet und auf dem Raspberry Pi der Steuerungsmodus auf „Uhrensteuerung“ geändert werden. Ist die Aktivitäts-LED des AccessPoints an, wartet dieser auf eingehende Verbindungen. Wird dann eine Uhr in den Sendemodus versetzt, fängt die Aktivitäts-LED des AccessPoints an zu blinken, wenn dieser Daten von einer Uhr erhält. Aus Gründen der Fähigkeiten des verwendeten Protokolls SimpliciT™ kommt es zeitweise zu Verbindungsproblemen mit der zweiten Uhr. Die Ursachen und Lösungen werden im Abschnitt „Test“ beschrieben. Eine gute Lösung ist dabei aber das Einschalten des Sendemodus der Uhren, bevor der Arduino neugestartet und in den Uhrensteuerungs-Modus gewechselt wird.

Die zur Ansteuerung und Interaktion mit den AccessPoint und dadurch die Kommunikation mit den Uhren nötige Steuerungslogik soll dabei in einem eigenen „StarModule“ namens „StarWatch“ implementiert und gekapselt werden.

Die folgenden Abschnitte dieses Kapitels sind direkt aus der Projektdokumentation des Projektes „CARMOTION“ – *Bewegungsbasierte Fahrzeugsteuerung mittels ez430-Chronos (von Dominik Scharnagl)* der Veranstaltung „Algorithmen für Sensornetze“ im Wintersemester 2017 / 2018 entnommen worden.

11.2.1. Anforderungen

Allgemein

Im Rahmen des Veranstaltung Algorithmen für Sensornetze sollen innerhalb des studentischen Projektes **Carmotion** Firmwares für das Development Tool ez430-Chronos von Texas Instruments entwickelt werden. Das dabei zum Einsatz kommende Development Tool enthält eine ez430-Chronos Watch sowie einen ez430-Chronos AccessPoint. Aufgrund der von der Veranstaltung gestellten Anforderung an die Realisierung eines Sensornetzes sollen für das Projekt zwei Development Tools verwendet werden. Ausgehend von der Situation, dass der standardmäßig im Development Tool enthaltene AccessPoint vom Typen Black nicht ohne weiteres neu programmiert werden kann, soll ein anderes Modell des AccessPoints verwendet werden. Anstelle des AccessPoints vom Typen Black soll somit das Modell vom Typen White verwendet werden, welches neu programmiert werden kann.



Die hierbei zu entwickelnden Firmwares zielen zum einen auf den Einsatz auf der ez430-Chronos Watch und zum anderen auf den Einsatz auf dem ez430-Chronos AccessPoint des Typs White ab. Mit ihnen soll es letztendlich möglich sein, die Steuerung eines Fahrzeugs durch die Bewegung der Handgelenke des Fahrzeugführers durchzuführen.

Fahrzeug

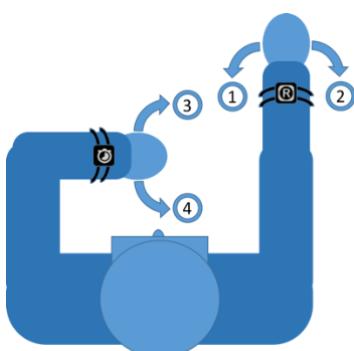
Im Rahmen der Veranstaltung Datenverarbeitung in der Technik soll ein teilautonomes Modellauto entstehen, welches über einen Elektromotor vor- und zurückfahren kann. Die Lenkung des Gefährtes soll über die Vorderachse mittels elektrischem Servomotor durchgeführt werden. Diese Motoren sollen dabei über einen Arduino Mega 2560 gesteuert werden.

Über ein auf dem Arduino angebrachtes USB Host Shield soll es möglich sein, die nötigen Steuerungsdaten über den ez430-Chronos AccessPoint (Typ White) von zwei ez430-Chronos Watches (Typ Black) abzurufen.

Fahrzeugsteuerung

Zur Steuerung des Fahrzeugs sollen zwei Steuerungswerte ermittelt werden. Ein Steuerungswert zur Steuerung der Beschleunigung und ein Steuerungswert zur Lenkung des Gefährtes. Die dabei zu generierenden Steuerungswerte sollen im Intervall $[-100,100]$ liegen. Während der Mittelwert 0 im Falle der Richtungssteuerung eine Neutralstellung, also eine Lenkung gerade aus ausdrücken soll, soll er für die Steuerung der Beschleunigung des Fahrzeugs ein Anhalten bedeuten.

Negative Werte, also alle Werte im Intervall $[-100, 0]$, sollen den prozentualen Anteil der Abweichung von der Neutralstellung nach links für die Servosteuerung festlegen. Für die Motorsteuerung sollen diese den Anteil an der maximal möglichen Beschleunigung entgegen der Fahrtrichtung des Fahrzeugs, also nach hinten bedeuten.



Positive Werte, also alle Werte im Intervall $[0,100]$, sollen den prozentualen Anteil der Abweichung von der Neutralstellung nach rechts für die Servosteuerung festlegen. Für die Motorsteuerung sollen diese den Anteil an der maximal möglichen Beschleunigung in Fahrtrichtung des Fahrzeugs, also nach vorne bedeuten.

Die Steuerung an sich (siehe Abbildung links) soll über eine Uhr am rechten Handgelenk für die Steuerung der Lenkung und über eine Uhr am linken Handgelenk für die Steuerung der Beschleunigung funktionieren.



Hierbei soll im Falle des rechten Handgelenks die Uhr waagrecht am ausgestreckten Arm getragen werden. Eine in der Position getragene und auf „Lenkung“ eingestellte Uhr soll durch eine Drehung des Handgelenks nach links (1) die Lenkung des Fahrzeugs im Intervall $[-100, 0]$ sowie mit einer Drehung nach rechts (2) die Lenkung des Fahrzeugs im Intervall $[0, 100]$ steuern.

Im Falle der Uhr am linken Handgelenk soll diese waagrecht und parallel zur Brust des Fahrers gehalten werden. Eine in der Position getragene und auf „Beschleunigung“ eingestellte Uhr soll durch eine Drehung des Handgelenks nach vorne (3) die Beschleunigung des Fahrzeugs im Intervall $[0, 100]$ sowie mit einer Drehung nach hinten (4) die Beschleunigung des Fahrzeugs im Intervall $[-100, 0]$ steuern.

Wird die Uhr waagrecht und mit dem Display nach oben gehalten, soll der Steuerungswert 0 generiert werden.

Besonders wichtig ist die Erkennung einer bewussten Steuerung des Fahrzeugs. So soll es nur möglich sein das Fahrzeug zu steuern, wenn der Fahrzeugführer beide Motionwristlets in horizontaler Lage hält. Eine entsprechende Toleranz soll dazu beitragen, dass ein Fahrzeugführer auch ohne eine hundertprozentige Ausrichtung seiner Wristlets (engl. Armbänder) das Fahrzeug steuern kann. Befindet sich ein Wristlet des Fahrers nicht in horizontaler Lage und außerhalb des Toleranzbereichs, sollen keine Steuerungsdaten produziert werden, um so den fahrzeugführerlosen Zustand gegenüber dem Gefährt zu repräsentieren. In diesem Zustand muss das Fahrzeug selbstständig sicherstellen, dass es nach Steuerungsausstieg des Fahrers in keine Gefahrensituation gerät.

eZ430-Chronos AccessPoint



Der zum Einsatz kommende eZ430-Chronos AccessPoint (siehe Abbildung links) soll mittels Breitbandnetz über das SRD-Frequenzband 868 MHz eine drahtlose Verbindung mit den beiden eZ430-Chronos Watches (=Wristlets) aufbauen. Über diese mittels asynchronem CDMA (= Code Division Multiple Access) hergestellte Drahtlosverbindung sollen in Kombination mit änderungsbasiertem TDMA (= Time Division Multiple Access) die Steuerungsdaten von den Wristlets empfangen werden.

Über CDMA soll es dann möglich sein, die beiden Teilnehmer (= Wristlets) bereits beim Verbindungsaufbau zu differenzieren. Diese Differenzierung erlaubt es dem AccessPoint, die Art des Steuerungswertes ohne weitere Angabe der Art während der Kommunikation zuzuordnen. Dies soll zur Reduktion der zu übertragenden Datenmenge beitragen.



Zusammen mit TDMA, welches änderungsbasiert umgesetzt werden soll, soll die Lebensdauer der Stromversorgung der Wristlets zusätzlich erhöht werden. So soll jedes Wristlet nicht nur innerhalb eines bestimmten Zeitfensters senden, sondern selbst innerhalb eines aktiven Zeitschlitzes nur dann Daten übertragen, wenn eine Änderung des Steuerungswertes im Vergleich zum zuvor gesendeten Steuerungswert (im letzten Zeitfenster) eingetreten ist. Ebenso soll nur dann ein Steuerungswert an den AccessPoint übermittelt werden, wenn sich das Wristlet nicht im fahrzeugführerlosen Zustand befindet.

Darüber hinaus soll der AccessPoint so angepasst werden, dass über seine USB-Schnittstelle immer die zuletzt empfangenen / gespeicherten Steuerungswerte bereitgestellt werden können. Befinden sich die Wristlets im fahrzeugführerlosen Zustand, so soll der AccessPoint beim Abrufen der Steuerungswerte keine Steuerungswerte bereitstellen, dafür aber über einen Statuscode den fahrzeugführerlosen Zustand mitteilen.

Zur Umsetzung der nötigen Funktionalität soll eine entsprechende Firmware für einen ez430-Chronos AccessPoint vom Typen White entwickelt werden. Diese Firmware soll die zuvor beschriebenen Eigenschaften realisieren und die genannten Protokolle implementieren. Hierzu soll das MSP430 USB-Debug-Interface MSP-FET430UIF von Texas Instruments zum Einsatz kommen, um den AccessPoint mit der neuen Firmware zu beschreiben sowie zu debuggen.

eZ430-Chronos Watch

Die Steuerung des Fahrzeugs soll mittels zweier eZ430-Chronos Watches (siehe Abbildung links), im weiteren Verlauf Motionwristlets genannt, umgesetzt werden. Hierbei soll ein Wristlet zur Steuerung der Richtung, also zum Lenken, des Fahrzeugs verwendet werden. Ein weiteres Wristlet soll zur Steuerung der Beschleunigung des Fahrzeugs zum Einsatz kommen. Zur Ermittlung des Steuerungswertes sollen die Sensorwerte des im Wristlet ab Werk verbauten Beschleunigungssensors CMA3000-D01 verwendet werden. Zudem soll über diese Werte auch der fahrzeugführerlose Zustand festgestellt werden.

Nachdem die Wristlets entgegen ihrer standardmäßig vorgeflaschten Firmware „Sports Watch“ in einem völlig anderen Szenario zum Einsatz kommen sollen, muss auch entsprechend die Benutzerführung der Uhr neu definiert werden. Ab Werk versteht das Gerät beziehungsweise die Firmware darunter eine Navigation durch „Menüs“ und zugehörigen Optionen über die links vom Display des Wristlets angebrachten Buttons. Zur Bestätigung / Änderung der gewählten Option / des gewählten Menüeintrags werden die Buttons rechts vom Display verwendet.





Das neue Bedienkonzept soll dabei eine Auswahl des Steuermodus ermöglichen, um so ein Wristlet für die Lenkung und ein Wristlet für die Beschleunigung einzustellen. Je nach eingestelltem Steuermodus soll dann das Gerät bei der Kommunikation über CDMA die Codierung der Kommunikation in Abhängigkeit des Modus ändern. Zudem soll die Uhr erst dann mit der Übertragung beginnen, wenn über einen weiteren Buttondruck die Verbindung gestartet wird. Mit selbigem Button soll diese auch wieder beendet werden. Weiter soll das Wristlet den aktuell durch die Bewegung produzierten Steuerungswert auf dem Display anzeigen, damit der Benutzer ein besseres Gefühl für die Handhabung der Steuerung bekommen kann. Nach dem Einlegen einer Batterie, also dem Einschalten der Uhr soll ein „Car Hello“ - als ein kleines „Hello World“ - am Display angezeigt werden, bis der Benutzer einen Button drückt.

11.2.2. Analyse / Design

ez430-Chronos Development Tool

Zu Beginn des Projektes musste ich mich zuerst einmal mit dem ez430-Chronos Development Tool (siehe Abbildung links) auseinandersetzen. Das aus einer ez430-Chronos Watch Typ Black, einem ez430-Chronos AccessPoint Typ Black, ez430-Chronos Programmer, einem Uhrenschraubendreher, einer CR2032 3V Lithium Knopfbatterie sowie ein paar Ersatzschrauben bestehende Development Tool bietet ein rundum vollständiges Paket, um ohne weitere Hardware ein ULP (= Ultra Low Power) Wearable auf Basis einer MCU (= Micro Controller Unit) an neue Anforderungen anzupassen. Das hierbei von Texas Instruments verbaute CC430F6137 SoC (= System-on-a-Chip) enthält neben dem MSP (= Mixed Signal Processor) MSP430 auch einen Wireless Transceiver, welcher zur drahtlosen Kommunikation mit dem beiliegenden AccessPoint verwendet werden kann. Zur Realisierung des Projektes erhielt ich die notwendigen zwei Uhren in Form zweier ez430-Chronos Development Tools, sowie einen ez430-Chronos AccessPoint vom Typen White und das zum Flashen / Debuggen notwendige USB-Debug-Interface MSP-FET430UIF von Herrn Volbert.





CC1101 Radio Controller

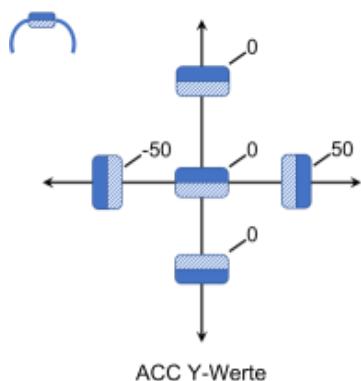
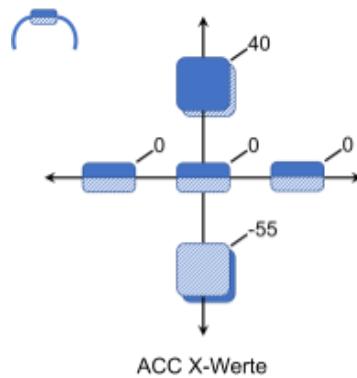
Der im SoC integrierte



Radio Controller CC1101 unterstützt neben diversen SRD-Frequenzbändern (< 1 GHz) auch das für den europäischen Breitengrad gültige SRD-Band mit 868 MHz. Als Low-Power RF (= Radio Frequency) Protokoll stellt Texas Instruments das eigens für batteriebetriebene Geräte entwickelte SimplexITI™ bereit. Das wenn auch nicht gerade optimal implementierte Protokoll bietet eine einfache Möglichkeit, die erfassten Steuerungswerte per Drahtlosverbindung weiterzureichen. Aus diesem Grund soll zu Beginn des Projektes SimplexITI™ anstelle von CDMA mit TDMA zum Einsatz kommen, um so erst einmal die Machbarkeit und Grundfunktionalität des Projektes sicherzustellen. Ein weiterer Grund hierfür ist auch der erhöhte Komplexitätsgrad bei der Implementierung der Firmware des AccessPoints, den der Einsatz von CDMA mit sich bringen würde. Wird im Verlauf des Projektes die Machbarkeit bestätigt und besteht noch ausreichend Zeit, dann sollen die geforderten Protokolle entsprechend implementiert werden.

CMA3000-D01 Beschleunigungssensor

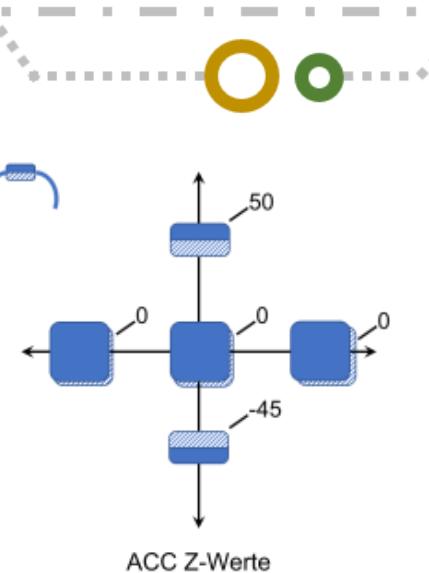
Neben den zudem verbauten Sensoren zur Messung der Restspannung der Batterie, der Temperatur und des Luftdrucks ist der Dreiachsen-Beschleunigungssensor CMA3000-D01 essentiell für die Realisierung des Projektes. Der Sensor erfasst im aktiven Zustand die Bewegung / Lage der Uhr über drei Achsen, die mit X, Y und Z beschrieben werden. Die hierbei gemessenen Werte werden in G (= $9,81\text{m}\cdot\text{s}^{-2}$) vom Sensor über drei Register bereitgestellt, je ein Register für den X-, Y- und Z-Wert. Um ein Gefühl und die Grenzen für die Wertebereiche der drei Achsen zu bekommen (welche nicht von Texas Instruments dokumentiert sind), wurde auf einem in Waage stehenden Tisch eine eZ430-Chronos Watch am Hals einer vollen 1,5 Liter PET-Flasche befestigt und ebenfalls in Waage liegend positioniert. Die in dieser Position von der Uhr indizierten X-, Y- und Z-Achsenwerte wurden entsprechend aufgezeichnet.



Anschließend wurde die Flasche um 90° nach vorne gedreht und wiederholt wurden die Messwerte protokolliert. Nach einer weiteren Drehung um 90° war das Display der Uhr parallel zur Tischplatte ausgerichtet. Auch in dieser Position wurden die Sensorwerte notiert. Daraufhin wurde die Konstruktion um weitere 90° und somit, ausgehend von der ersten Messung, um insgesamt 270° weitergedreht und auch hier wurden die Messdaten dokumentiert.

Auf diese Untersuchungen hin wurde die Flasche auf den Tisch gestellt, sodass die Uhr im 90° Winkel zum Tisch positioniert war. Auch hier wurden die Sensordaten protokolliert. Nach Abschluss der Messvorgänge wurden die notierten Sensorwerte in ein 2D-Koordinaten System übertragen (siehe Abbildungen).

Die anschließende Auswertung der verschiedenen gemessenen Sensordaten haben zu der Entscheidung geführt, dass der eigentliche Steuerungswert basierend auf dem Wert der X-Achse ermittelt wird. Der Grund hierfür liegt darin, dass der Sensor, wie in Abbildung rechts zu sehen ist, stets in Abhängigkeit der Drehung des Displays (ausgehend davon, dass die Uhr am Handgelenk getragen parallel zur Brust mit waagrechter Haltung des Displays nach oben zeigt) nach vorne, also vom Träger weg, und nach hinten gedreht immer einen entsprechenden Wert liefert. Zugleich ändert sich der Wert nicht, wenn die Uhr in selbiger Ausgangslage bewegt wird.



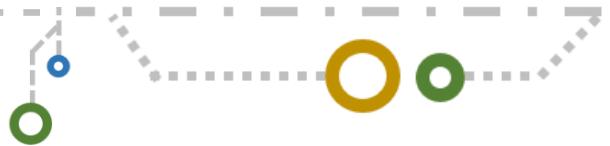
Die Untersuchung der Ergebnisse der Y-Achsendaten (siehe Abbildung „ACC Y-Werte“) führte hingegen zur Entscheidung, dass dieser Wert zur Ermittlung der bewussten Steuerung als Referenzwert verwendet werden kann, denn er ändert sich nur in Abhängigkeit von der Neigung des Unterarmes des Fahrers. Ist zum Beispiel die Uhr am linken Handgelenk befestigt und legt der Benutzer den Arm parallel zur Hüfte an seiner Seite an, also „bequem stehend“, liefert der Sensor einen Y-Wert von -50. Ist die Uhr hingegen am rechten Handgelenk in derselben Haltung angebracht, liegt der Y-Wert bei 50. Für die abschließende Bewertung des Z-Wertes (siehe Abbildung „ACC Z-Werte“) kam ich zu dem Schluss, dass sich dieser in etwa mit den X-Wert vergleichen lässt, jedoch in seinem Verhalten „gespiegelt“. Der Z-Wert wäre somit ein alternativer Wert für den X-Wert gewesen. Allerdings liegt sein Wert bei 50, wenn sich die Uhr in der angedachten „Neutralstellung“ befindet, wohingegen der X-Wert bereits von sich aus die „Neutralstellung“ mit dem Wert 0 ausdrückt.

Entwicklungswerkzeuge für die Plattform



Im Rahmen der Analyse des Development Tools testete ich die von Texas Instruments bereitgestellten Tools wie den eZ430-Chronos Datalogger, das eZ430-Chronos Control Center und die IDEs Code Composer Studio v7, sowie die IAR Embedded Workbench. Beide integrierten Entwicklungsumgebungen werden von

Texas Instruments bereitgestellt, wobei das Code Composer Studio auf dem Eclipse Projekt basiert und somit kostenlos zur Verfügung steht. Angesichts einer gegebenenfalls über diese Veranstaltung hinausgehende Beschäftigung mit dieser Plattform oder Ähnlichem wollte ich das Code Composer Studio zur Entwicklung verwenden.



Nach einem wirklich sehr langen Installationsprozess und anschließendem sehr langwierigen Updateprozess (trotz zuvor aktuellem Download des Setups) konnte ich mit der IDE arbeiten. Jedoch musste ich beim einfachen Versuch das Beispielprojekt „Sports Watch“ von Texas Instruments zu kompilieren feststellen, dass diverse notwendige Erweiterungen fehlten.

Durch zeitaufwändiges Suchen im Internet konnte ich diverse Erweiterungen ausfindig machen, die ich nachträglich einrichten musste, wobei sich die Community bezüglich der Erweiterungen nicht ganz einig ist. Nachdem ich manche Erweiterungen erst gar nicht im Komponentenmanager finden konnte, andere sich nicht installieren ließen oder trotz Installation nicht aktiv waren und darüber hinaus teilweise nach einem Neustart des Code Composer Studios wiederholt sehr lange auf Updates geprüft wurde, habe ich nach Rücksprache und auf Empfehlung von Herrn Volbert zur IAR Embedded Workbench gewechselt.

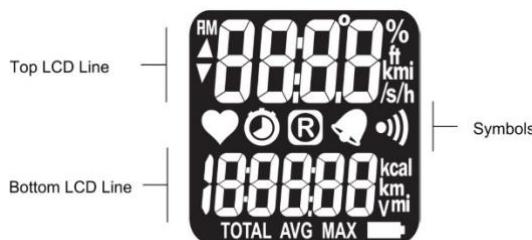


www.iar.com

Es stellte sich bald heraus, dass die aktuelle Demoversion der IAR Embedded Workbench - entgegen den Erfahrungen von Herrn Volbert - nicht länger für den Einsatz mit dem „Sports Watch“ Projekt geeignet ist. Deshalb riet Herr Volbert mir, eine Lizenz der OTH Regensburg für die IDE zu verwenden. Leider funktionierte das nicht, da die Lizenz der OTH nur für eine ältere Version der IAR Embedded Workbench gültig ist. Nach längerer Suche konnte ich keine entsprechende Version der IAR Embedded Workbench im Internet finden, welche mit der Lizenz der OTH Regensburg lauffähig gewesen wäre. Am 08. November (die Veranstaltung startete am 04. Oktober) erhielten alle Teams von Herrn Volbert eine Version der IAR Embedded Workbench per USB Stick zu Beginn der Veranstaltung. Mit dieser war es mir ab dann möglich, mich genauer und tiefergehend als zuvor mit der Plattform auseinanderzusetzen.

96-Segment LCD Anzeige

Die direkt am CC430F6137 angebrachte und somit auch über diesen ansteuerbare 96-



Segment LCD Anzeige (siehe Abbildung links) dient im Beispielprojekt „Sports Watch“ der Navigation in den diversen Menüs, der Konfiguration der Einstellungen und der Anzeige erfasster Werte. Im Rahmen des Projektes soll der Benutzer nach Einlegen der

Batterie mit einem „Car Hello“ begrüßt werden. Hierzu wurde die Ansteuerung des Displays untersucht und festgestellt, dass der im „Sports Watch“ Projekt implementierte LCD-Treiber eine unvollständige Umsetzung der Zeichen A-Z und a-z auf die 7-Segment Felder verwendet. Dementsprechend muss im Rahmen der Implementierung die Übersetzungsmatrix von ASCII-Zeichen auf 7-Segment Felder erweitert werden, sodass die

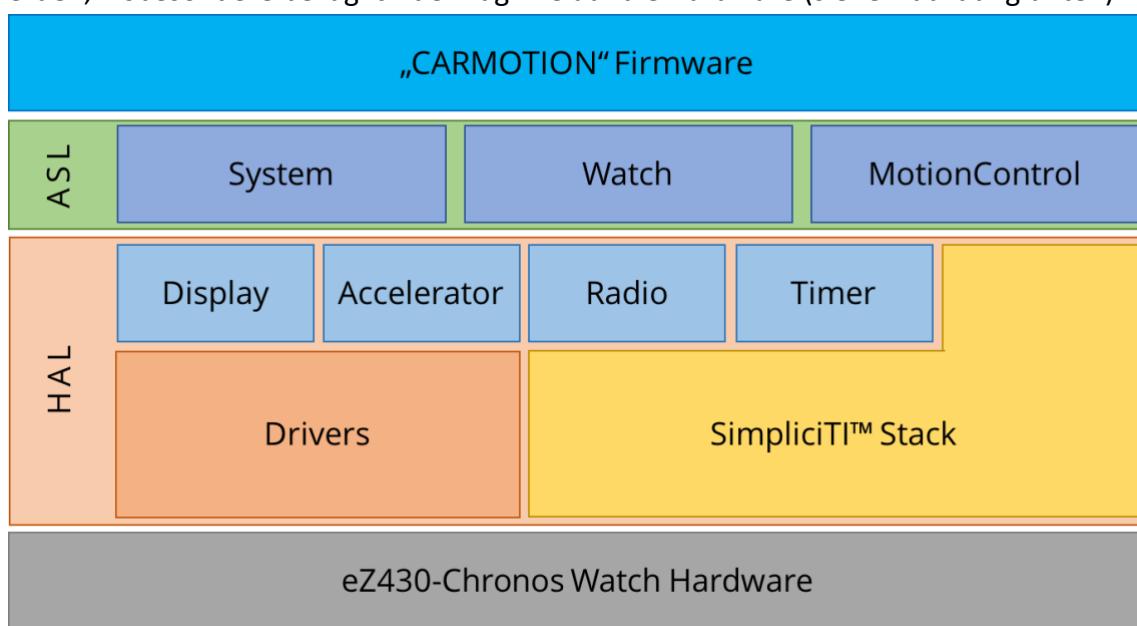




für das Projekt geforderte Anzeige realisiert werden kann. Ebenso soll zur Anzeige des aktuellen Steuermodus entweder das Stopwatch Symbol zur Steuerung der Beschleunigung oder das Record Symbol zur Steuerung der Lenkung angezeigt werden.

Projektumgebung

Das während der Implementierung teilweise leicht angepasste, aber von Anfang an zum Großteil vorgesehene Design des Aufbaus der Firmwares sieht eine typische Trennung nach dem Vorbild bekannter HALs (= Hardware Abstraction Layers) vor. Aufgrund der Beschränkung der Plattform auf natives C, das einen objektorientierten Ansatz - welcher hier nicht gegangen werden sollen - nur über Umwege zulässt, soll ein Zwischenweg gegangen werden. Der Aufbau der Firmware soll von oben nach unten immer spezifischer werden, insbesondere bezüglich der Zugriffe auf die Hardware (siehe Abbildung unten).



Schichtenübergreifende Operationen sollen durch das „System“ geregelt werden, während zugleich im „Application System Layer“ (= ASL) höhere Komponenten der Firmware angesiedelt sind. Insbesondere „Watch“ und „MotionControl“ besitzen ausschließlich Referenzen in die HAL. Die Hardware Abstraction Layer hat hingegen nur minimale Verweise auf das „System“ der ASL. Alle tiefergehenden Zugriffe auf die Hardware finden ausschließlich über die HAL statt. Was SimpliciT™ an sich angeht, so erweitert der Stack den Treiberanteil wie auch den Komponentenanteil um die für SimpliciT™ relevante API.



11.2.3. Implementierung

ez430-Chronos Watch - Firmware: "Carmotion"

Schon zu Beginn, als ich das erste Mal das „Sports Watch“ Projekt kompilieren konnte, stellte ich schnell fest, dass das als „Beispielprojekt“ von Texas Instruments betitelte Projekt nicht nur ein Beispielprojekt ist, sondern eine wirklich sehr umfangreiche und mit jedem Sensor verknüpfte Firmware darstellt. Das hat auch zur Folge, dass die Kompilierung und insbesondere der Upload der Firmware auf die ez430-Chronos Watch verhältnismäßig viel Zeit in Anspruch nimmt. Dieser Umstand brachte mich zu dem Entschluss, dass ich im Rahmen meines Projektes eine völlig neue Projektumgebung aufsetzen würde. Dieses Vorgehen hatte sich auch in der Vergangenheit schon des Öfteren für mich zum einen als zielführend und zum anderen als sehr lehrreich herausgestellt.

Ein leeres Projekt war schnell aufgesetzt und nach dem groben Abgleich der Projekteinstellungen, insbesondere derer für die Zielplattform, mit meinem neuen Carmotion Projekt und dem „Sports Watch“ Projekt war ich schnell im Stande ein kleines „Hello World“ auf die Uhr zu flashen (dieses ließ ein einzelnes Segment der Segmentanzeige aufleuchten). Nachdem ich dann versuchte, den (für mich auf dem ersten Blick sauber strukturierten und gekapselten) Code des „Sports Watch“ Projektes schrittweise on-demand zu portieren, stellte ich schnell fest, dass das Beispielprojekt zum großen Teil aus Spaghetti Code besteht. Dieser führte dazu, dass ich wirklich viel Zeit damit verbracht habe, mein entworfenes Design mit den nötigen Funktionen aus dem Beispielprojekt aufzubauen. Immer wieder musste ich Querverweise auf anderen Code auflösen oder bestimmte Symbole neu definieren, suchen und aus dem Beispielprojekt in mein Projekt überführen.

Ich begann immer mehr, das „Sports Watch“ Projekt im Rahmen meines Reverse Engineerings zu verstehen, insbesondere auf welche Weise die Firmware auf der Uhr mit den Hardwarekomponenten interagiert. Durch zusätzliches Debugging konnte ich noch viele weitere Details über die Plattform und deren Funktionsweise erlernen, welche ich dann auch parallel und Stück für Stück in mein Carmotion Projekt überführte. Dieses Vorgehen verfolgte ich solange, bis ich auf einen Linker-Fehler stieß, über den ich mit der Kompilierung nicht hinauskam. Alle zuvor aufgetretenen Fehler konnte ich verhältnismäßig schnell lösen, an diesem Punkt kam ich jedoch nicht mehr mit meinem neuen Projekt weiter.

Aufgrund der Probleme mit dem neu aufgesetzten Projekt entschied ich mich dazu, das „Sports Watch“ Projekt so lange schrittweise auszuschalten, bis das Projekt nur noch die von mir benötigten Treiber implementiert. Hierbei war es besonders aufwendig diverse Querverweise zu anderen Treibern aus den notwendigen Treibern sowie der Logikschicht herauszulösen. Deswegen habe ich parallel immer wieder das Projekt von Neuem kompiliert, auf eine Uhr hochgeladen und Funktionstests durchgeführt. Vor der



eigentlichen Kompilierung habe ich zusätzlich die Ausgabeverzeichnisse manuell bereinigt, um sicherzustellen, dass es nicht zu missverständlichen Linker-Fehlern kommt oder gar zu einer augenscheinlich funktionierenden Firmware.

Sobald ich einen gewissen Grad der Bereinigung des Projektes erreicht hatte, begann ich damit, die im ersten Vorgehen geschaffene Infrastruktur schrittweise in die des ehemaligen „Sports Watch“ Projekts zu überführen. So wurde nach und nach aus dem Beispielprojekt die von mir benötigte Firmware.



Die auf diese Weise geschaffene Projektumgebung konnte ich dann gemäß den Anforderungen weiterentwickeln und implementierte im ersten Schritt das beschriebene Car Hello. Daraufhin folgte die Implementierung der spezifizierten GUI zur Interaktion mit dem Benutzer (siehe Abbildung oben). Dabei wurde die mangelhafte Übersetzungsmatrix des LCD-Treibers um das Mapping weiterer ASCII-Zeichen auf entsprechende Segment-Anzeige-Kombinationen erweitert.

Fortgesetzt wurde die Implementierung der „Carmotion“ Firmware, indem die in der Analysephase erlangten Erkenntnisse über die Wertebereiche der drei Sensorwerte (X, Y und Z) und deren Nutzen für das Projekt entsprechend in die Software integriert wurden. Hierzu mussten die Werte erst vorverarbeitet werden, bevor diese dem AccessPoint mitgeteilt werden.

Generell liegen die Sensorwerte als unsigned 8 Bit Wert vor. Deshalb war es notwendig, die Sensorwerte auf einen Wertebereich um den Nullpunkt zu arrangieren (siehe Snippet unten).

```
// Align values around zero and determine -/+ range.  
if (x > 127)  
    x = -(255 - x);  
  
if (y > 127)  
    y = -(255 - y);
```



Nach der Arrangierung der Sensorwerte wird geprüft, ob im Rahmen des aktuellen Steuermodus der ermittelte X- Wert überhaupt als gültiger Sensorwert weiterverarbeitet werden soll. Somit muss der Y-Wert im Intervall $] -5, 5[$ liegen, wenn das Wristlet auf den Modus Beschleunigung eingestellt wurde, damit eine weitere Berechnung des Steuerungswertes erfolgt. Im Falle, dass die Uhr auf den Modus Lenkung eingestellt ist, wird nur dann der Steuerungswert berechnet, wenn der Y-Sensorwert im Intervall $] -15, 0[$ liegt. Liegt der Y-Wert außerhalb des für den Modus gültigen Intervalls, befindet sich die Uhr im fahrzeugführerlosen Zustand.

Befindet sich die Uhr nicht im fahrzeugführerlosen Zustand, muss anschließend der Wert in seinen entsprechenden Prozentwert umgerechnet werden, welcher abhängig von der Lage des Wertes nach der Arrangierung ist. So muss ein Wert, der < 0 ist, mit einem anderen Nenner als ein Wert, der > 0 ist, dividiert werden, um den Prozentwert zu errechnen (siehe Snippet unten).

```
// Calculate percentage portion of the value relative to its range.  
if (x < 0)  
    x = (-x / 55.0) * 100;  
else  
    x = (-x / 40.0) * 100;
```

Anschließend muss der aus der vorangegangen Berechnung resultierende Wert eine Toleranzprüfung durchlaufen. Diese Prüfung ist notwendig, um ein Zittern der Steuerungswerte bei minimaler (unbewusster) Bewegung der Uhr zu vermeiden. Somit sollen keine geänderten Steuerungswerte produziert werden, wenn der Benutzer gerade versucht, die Uhr in Waage zu halten (siehe Snippet unten).

```
// Skip values lower than 5% (in modulus).  
if (x > -5 && x < 5)  
    x = 0;
```

Erfüllt er auch diese Toleranzprüfung, wird noch ein mögliches Überschwingen der Steuerungswerte ausgeschlossen, indem die Werte an ihre Grenzen im Intervall $[-100, 100]$ ausgerichtet werden (siehe Snippet unten). Dieses Vorgehen soll selbst im Falle dessen, dass der Beschleunigungssensor aus diversen Gründen einen Sensorwert außerhalb des bekannten Wertebereiches liefert, verhindern, dass dann immer noch gültige Steuerungswerte im Intervall $[-100, 100]$ von der Firmware an den AccessPoint gesendet werden.



```
// Align values lower than -100% to exact -100%.
if (x < -100)
    x = -100;

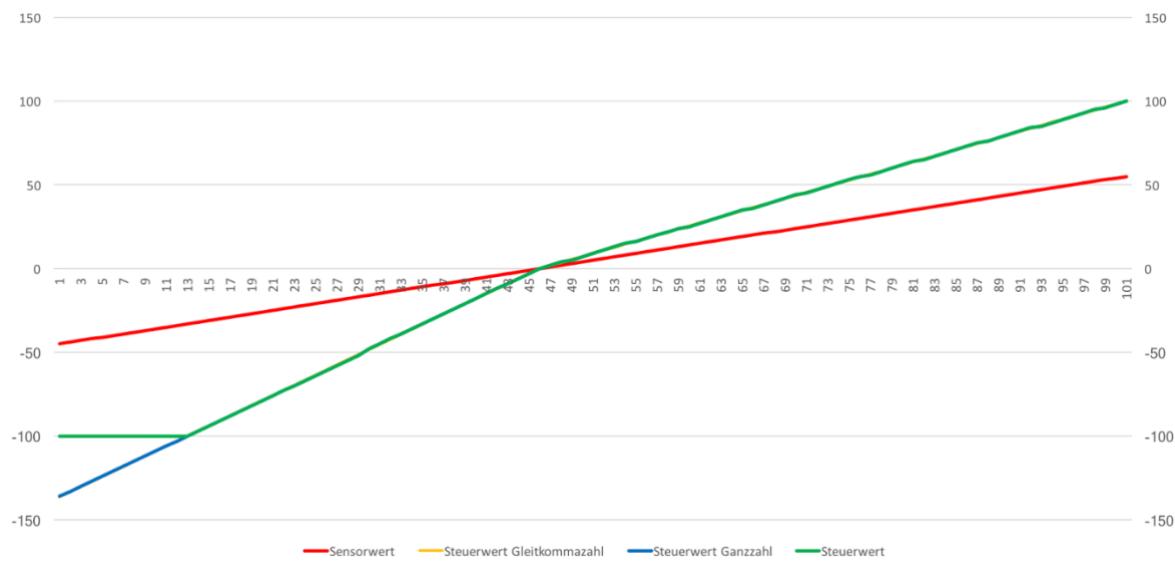
// Align values greater than 100% to exact 100%.
else if (x > 100)
    x = 100;
```

Der daraus resultierende Wert wird daraufhin entsprechend des physikalisch möglichen (= schmerzfreien) Bewegungsradius des Handgelenks angepasst (siehe Snippet unten). So ist eine Bewegung des Handgelenkes, bei parallel zur Brust gehaltenem Unterarm, zur Brust hin nur zu knapp 60% einer 90°-Drehung möglich. Da aber der Grenzwert zur Berechnung des Prozentwertes zu 100% eingeht, muss dieser Prozentwert nachträglich, basierend auf den möglichen 60%, neu berechnet werden. Hinzu kommt, dass auch hier wieder die Intervallgrenzen eingehalten werden müssen, falls ein Fahrer über den 60%-Bewegungsradius kommen sollte. Zu beachten ist, dass dieses Vorgehen nicht nur auf den Bewegungsradius der Beschleunigung - also auf das linke Handgelenk - zutrifft, sondern auch auf den Bewegungsradius der Lenkung - also auf das rechte Handgelenk. So kann auch bei ausgestrecktem Arm das Handgelenk nach links nur zu knapp 60% einer 90°-Drehung gedreht werden.

```
if (x < 0) {
    // Align value to maximum possible motion range.
    // The motion range -60% to 100% (without alignment).
    x = (x / 60.0) * 100;

    // Align values lower than -100% to exact -100%.
    if (x < -100)
        x = -100;
}
```

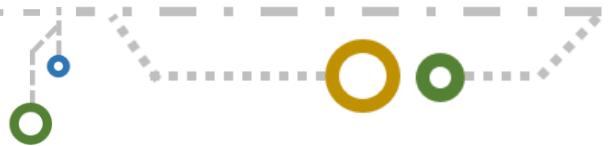
Hat der Sensorwert alle vorangegangenen Bedingungen erfüllt, wird er von der Firmware als gültiger Steuerungswert erfasst, gespeichert und an den AccessPoint weitergeleitet. Bei der Weiterleitung wurde aufgrund der binären Datenübertragung entschieden, dass die Steuerungswerte vom Intervall $[-100, 100]$ auf das Intervall $[0, 200]$ abgebildet werden, damit der Wert zum Wertebereich eines unsigned 8 Bit Wertes kompatibel ist.



Während der Testphase wurde allerdings festgestellt, dass die zuvor auf dem Arduino implementierte Logik zur Berechnung des Steuerungswertes auf der eZ430-Chronos Watch die Steuerungswerte zu ungenau berechnet. Die fehlende Genauigkeit in der im MSP430 integrierten ALU führte somit dazu, dass die Motoren bei der Steuerung nur Sprünge anstelle von Bewegungen durchführten. Aufgrund dieser Erkenntnis versuchte ich, das Problem mit einer Zuordnungsmatrix zu beheben (siehe Abbildung oben). Diese sollte einen auf 0 ausgerichteten Sensorwert einem bestimmten Steuerungswert 1:1 zuordnen. Dieses Vorgehen wäre dabei nicht nur genauer, sondern auch performanter für die Armbanduhren. Leider zeigten weitere Tests, dass selbst die mit Microsoft Excel generierten Werte der Matrix kaum eine Besserung mit sich brachten. Aus diesem Grund wurde die Berechnung des Steuerungswertes für die Richtung und die Beschleunigung auf dem Arduino Mega 2560 implementiert, wohingegen die Bewertung der Lage der Uhren weiterhin in der Firmware der Wristlets durchgeführt wird. Hierfür wird keine entsprechende Fließkommaarithmetik benötigt. Zusätzlich wird dadurch der Datenverkehr zwischen den Uhren und dem AccessPoint verringert, wenn der Fahrer seinen Arme nicht in „Steuerungslage“ hält.

eZ430-Chronos AccessPoint - Ansteuerung

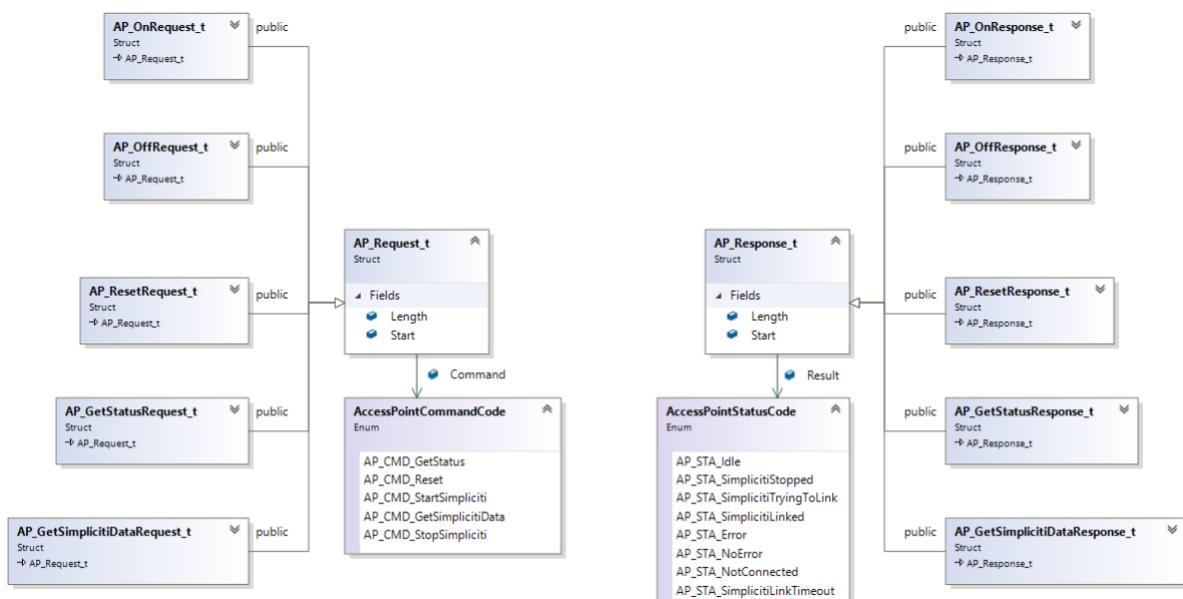
Am 05. Dezember begann ich mit der Untersuchung der Ansteuerung des eZ430-Chronos AccessPoints. Hierbei versuchte ich zu allererst Requests, welche ich in der Firmware und in Beispielen aus dem Internet gefunden habe, vom Arduino Mega über das USB Host Shield an den AccessPoint zu senden. Jeglicher Schreib- wie auch Lesezugriff über den generischen USB Treiber der Arduino USB Host Library 2.0 zur seriellen Kommunikation mit dem AccessPoint scheiterte. Nach zahlreichen Versuchen stellte ich wenige Tage später fest, dass die Treiberklasse des USB Treibers nicht auf dem Heap allokiert werden darf, damit diese sich bedienen lässt. Zu dieser Erkenntnis kam ich im Rahmen der Arbeit für die Veranstaltung Datenverarbeitung in der Technik, als ich als alternative Steuerung des



Fahrzeuges einen Xbox 360 USB Controller am USB Host Shield des Arduinos ansteuern wollte und auch dies nicht funktionierte.

Trotz dieser Erkenntnis konnte ich weiterhin keine Requests erfolgreich an den AccessPoint senden. Also entschied ich mich - nicht zuletzt wegen der verhältnismäßig längeren Kompilier- und Uploadzeiten bei der Entwicklung der Spikes für die AccessPoint-Ansteuerung am Arduino - dazu, einen Spike mittels einfacher C# Konsolenanwendung zu implementieren. Mit Hilfe einer kleinen Beispielanwendung, welche sich auch schnell im Internet bei der Community der Plattform finden ließ, konnte ich erfolgreich mit dem AccessPoint kommunizieren. Das funktionierte mit dem Beispiel aber nur einmalig. Auch wenn ich mit dem Beispielprojekt sogar in der Lage war, nicht nur den AccessPoint ein- und auszuschalten, sondern auch Daten des Beschleunigungssensors einer mit dem AccessPoint verbundenen Uhr abzurufen, funktionierte die Kommunikation mit dem AccessPoint nur ein einziges Mal. Nach einigen Versuchen stellte ich fest, dass das Problem behoben ist, wenn ich vor dem Start der Konsolenanwendung den AccessPoint vom PC ab- und wieder anstecke. Dann funktionierte die Anwendung auch wieder nur für eine Ausführung. Somit musste ich vor jeder weiteren Ausführung immer den AccessPoint ab- und wieder anschließen, um meine Tests fortzusetzen. Den Grund hierfür habe ich selbst nicht bis zum Abschluss des Projektes ausfindig machen können.

Mit der nun (grundsätzlich) funktionsfähigen Konsolenanwendung hatte ich eine Vorlage, mit der ich auf dem Arduino Mega die Implementierung des seriellen Protokolls zur Ansteuerung des eZ430-Chronos AccessPoints implementieren konnte. Hierzu entschied ich mich aber, die Implementierung gemäß meinen Vorstellungen nicht mit fest kodierten Bytes (wie es die Beispielanwendung tat), sondern mit Enumerationen und Strukturen im Rahmen eines C/C++ Projektes durchzuführen. Das daraus entstandene Klassendiagramm für die Request- und Response-Strukturen ist in der Abbildung unten zu sehen.





Nach der Implementierung des entsprechenden Objektmodells für Request- und Response-Daten zur Interaktion mit dem AccessPoint analysierte ich die Funktion des nun strukturbasierten Stacks. Hierbei stellte ich immer zuerst einen On-Request und anschließend fragte ich zyklisch mittels GetStatus-Request den Status des AccessPoints ab. Erst wenn der Status mir anzeigen, dass der AccessPoint mit einer Uhr verbunden ist, dann hätte ich mit einem GetSimplicitiData-Request die Daten abgerufen. Jedoch erhielt ich mit diesem Stack stets als Ergebnis der GetStatus Operation den Code 0x06, welcher für NoError steht, anstelle des Codes 0x00 für Idle oder 0x03 für SimplicityLinked. Hier habe ich verhältnismäßig lange danach gesucht, bis ich feststellte, dass ich den Wert 0x06 als Initialwert im Konstruktor der Struktur gesetzt hatte. Aber auch nachdem ich diesen Fehler gelöst hatte, erhielt ich keinen anderen Wert für den AccessPoint Status als den, den ich selbst in der Struktur vor der Anfrage setze.

Nach Rücksprache mit Herrn Volbert blieb mir keine andere Möglichkeit, als die Kommunikation zwischen einer funktionierenden Software und dem AccessPoint zu überwachen und jede Read- wie auch Write-Operation auf serieller Ebene zu belauschen. Dazu verwendete ich das Device Monitoring Studio 7.81 und protokollierte die Daten, die bei jedem Request und Response zwischen dem eZ430-Control Center und dem eZ430-Chronos AccessPoint ausgetauscht werden. Die so erfassten Rohdaten betrachtete ich in hexadezimaler Schreibweise. So konnte ich, basierend auf dem Beispielcode des AccessPoint-Projektes schnell die einzelnen Byte Sequenzen den entsprechenden Übertragungen zuordnen. Das sehr zeitaufwändige Verfahren machte mich aber darauf aufmerksam, dass das Längen-Byte im GetStatus-Request um 1 größer ist als das, welches ich bei meinen Requests definierte. Nach genauerer Untersuchung stellte sich heraus, dass Texas Instruments zwar beim GetStatus-Request ein weiteres Byte im Request vorsieht, dieses aber nie mit einen konkreten Wert belegt. Nachdem ich dann meine Struktur um das fehlende Byte erweitert hatte, funktionierte bei mir auch die gesamte Kommunikation mit dem AccessPoint, wie bereits in der C# Konsolenanwendung sichergestellt. Die anschließende Integration des C/C++ Codes in die Firmware des Arduino Mega 2560 gestaltete sich daraufhin als entsprechend leicht. Interessant wie auch positiv stellte sich heraus, dass das Problem der einmaligen Nutzbarkeit des AccessPoints - welches ich in der C# Konsolenanwendung hatte - auf dem Arduino nicht existiert.

eZ430-Chronos AccessPoint - Firmware: "Carcontrol"

Entsprechend den Anforderungen sowie mangels Zeit (es war bereits der 04. Januar) und des seitens Texas Instruments leichtgewichtigen eZ430-Chronos AccessPoint Projektes habe ich dieses 1:1 kopiert und die notwendigen Anpassungen an der Firmware vorgenommen, ohne erneut mit einem leeren Projekt zu starten. Nachdem aber die meiste Logik in der Firmware der Watches zu implementieren ist, wurde hier lediglich die Speicherung der letzten Steuerungswerte für die Lenkung und Beschleunigung eingebaut. Abschließend wurde der bereits vorhandene GetSimplicitiData USB Handler so modifiziert, dass er nur noch die beiden Steuerungswerte liefert.



11.2.4. Test

Anwendung versus Theorie

Im Rahmen der Tests wurde festgestellt, dass die X-Werte negiert in die Berechnung des Steuerungswertes eingehen müssen, da die Steuerung ansonsten invers funktionieren würde. Somit hätte eine Bewegung des Beschleunigungs-Handgelenkes nach vorne zu einem Zurückfahren und eine Bewegung nach hinten zu einem Vorwärtsfahren geführt. Ebenso wäre auch der Lenkarm davon betroffen gewesen, sodass eine Lenkung nach links eine Lenkung nach rechts und umgekehrt hervorgerufen hätte. Somit sah das Snippet vor dieser Erkenntnis wie im Snippet unten aus.

```
// Calculate percentage portion of the value relative to its range.  
if (x < 0)  
    x = (x / 55.0) * 100;  
else  
    x = (x / 40.0) * 100;
```

Kalibrierung zur schmerzfreien Steuerung

Ebenso kam ich während der Testphase zur Erkenntnis, dass der Bewegungsradius der Handgelenke nach innen und zur Brust hin eingeschränkt ist. Das brachte mich zur Implementierung der zusätzlichen Kalibrierung der Steuerungswerte im Falle einer Linkslenkung beziehungsweise im Falle einer negativen Beschleunigung des Fahrzeugs. Durch weitere Tests stellte sich heraus, dass eine schmerzfreie sowie einfache Lenkung nur dann möglich ist, wenn die Wristlets nach innen wie auch zur Brust hin nur zu 60% der ursprünglich angedachten 90°-Drehung bewegt werden. Dementsprechend sieht die zusätzliche Kalibrierung vor, dass eine 60%- einer 100%-Drehung entspricht.

Seiteneffekte der Kalibrierung

Nach der Implementierung der 60%-Kalibrierung stellte sich wiederum heraus, dass durch den eingeschränkten Bewegungsradius bei der Lenkung nach links oder bei negativer Beschleunigung die Steuerung spürbar empfindlicher reagiert als wenn man eine Lenkung nach rechts beziehungsweise eine Beschleunigung in Fahrtrichtung vornimmt. Somit hatte eine geringere Bewegung der Handgelenke eine größere Auswirkung auf das Fahrverhalten nach hinten und nach links als nach vorne und nach rechts.

Die einzige mögliche Lösung des Seiteneffektes wäre es gewesen, dass man den Nullpunkt, also die „Neutralstellung“ von Servo und Motor, nicht auf einen Sensorwert gleich 0 legt, sondern einen Mittelwert aus den schmerzfrei möglichen Grenzwerten bildet und diesen als Referenz für die „Neutralstellung“ verwendet.



Letztendlich habe ich mich jedoch dafür entschieden, dass weiterhin die „Neutralstellung“ beim Sensorbasierten Mittelwert 0 bleibt, da mit etwas Übung auch dieses Verhalten beherrschbar wird. Darüber hinaus ist es für den Fahrer erheblich schwieriger, sich an ein nach leicht vorne zeigendes beziehungsweise leicht nach rechts geneigtes Handgelenk für die Neutralstellung zu gewöhnen, als das die Lenkung nach links wie auch die Beschleunigung entgegen der Fahrtrichtung empfindlicher als ihr Gegenpart reagieren.

Kalibrierung der Toleranz

Da bereits eine gewisse Toleranz bereits zur Spezifikationsphase der Anforderungen bekannt und gefordert war, wurden die Intervallgrenzen der Toleranz erst während der Testphase ausgelotet. So haben die Tests ergeben, dass ein betragsmäßig betrachteter Steuerungswert größer als 5% sein muss, damit von einer bewussten Bewegung der Uhr und somit von einer aktiven Steuerung des Fahrzeugs durch den Fahrzeugführer ausgegangen werden kann.

Steuerungslogik

Während der Implementierung durchlief der Code beziehungsweise der Algorithmus zur Berechnung der Steuerungswerte diverse Projekte und Plattformen. Zu Beginn wurde dieser in C# implementiert, anschließend auf C/C++ portiert und danach vom PC auf den Arduino des Fahrzeugs umgezogen. Ziel meines Projektes war es jedoch, den größtmöglichen Teil der logischen Datenverarbeitung in die Firmware der Uhr zu packen. Während der Tests stellte ich aber fest, dass der Algorithmus zur Berechnung der Steuerungswerte in Prozent stark von der Genauigkeit der auf der Plattform implementierten Fließkommaarithmetik abhängig ist. Während der Algorithmus auf dem PC und auf dem Arduino stets Werte lieferte, mit denen die Motoren immer flüssig und nahezu synchron zu meinen Bewegungen verliefen, führte die Berechnung der Steuerungswerte auf der Watch zu einem Springen der Motoren. Dieses konnte selbst durch eine über Microsoft Excel mit hoher Genauigkeit berechnete Zuordnungsmatrix (Sensorwertindex → Steuerungswert) nicht viel mehr verbessert werden. Diese Testergebnisse führten auch dazu, dass die Berechnung des Steuerungswertes auf dem Arduino implementiert wurde.

Abhängen durch „Davonfahren“

Teil meiner Tests war auch das Verhalten des Fahrzeugs zu untersuchen, wenn es keine Verbindung mehr zu den Wristlets hat. Hierzu ließ ich im Flur des Sammelgebäudes (3. Stock) das Fahrzeug bis ans Ende des Flures fahren, während ich am anderen Ende des Flures stand. Das Fahrzeug konnte problemlos die Verbindung zu den Uhren aufrechterhalten und somit den gesamten Flur abfahren. Daher war der im Rahmen der Tests zur Verfügung stehende Raum nicht ausreichend, um das Fahrzeugverhalten zu beobachten, wenn es zu einem Verbindungsabriß kommt.

Verbindungstests: Uhr → Auto / Auto → Uhr



Sobald das Projekt an dem Punkt angelangt war, dass beide Uhren das Fahrzeug steuern konnten, habe ich vermehrt geprüft, wie sich der AccessPoint beziehungsweise dessen Firmware verhält, insbesondere was das Aufbauen und Halten der Verbindung zu den Uhren anbelangt. Hier habe ich festgestellt, dass es aufgrund des schwachen SimpliciTI™ Stacks zu Verbindungsproblemen kommen kann, wenn bereits eine Uhr Daten sendet, während der AccessPoint bereits aktiv ist und ebenfalls stetig über USB die Daten weiterreichen muss. So kommt es gehäuft vor, dass in solch einem Szenario keine Verbindung zur zweiten Uhr aufgebaut werden kann. Die Lösung hierfür ist, dass man erst die Uhren in den „Sendemodus“ versetzt und dann den AccessPoint per USB nach Daten abfragt. Im Falle des Projektes StarCar war die Lösung, dass man den Arduino neustartet.

Eine alternative Lösung des Problems war (mit der alten Firmware „Sports Watch“), dass man immer wieder die Uhr, die sich nicht verbinden kann, in den Sendemodus versetzt und wieder herausnimmt, wahlweise auch im Wechsel mit der Uhr, mit der sich eine Verbindung auf- und wieder abbauen lässt. Dann erkannte der AccessPoint beide Teilnehmer. Dieses Verfahren funktionierte aber nicht länger mit der eigenen Firmware.

12. Serieller Port – Raspberry Pi 3 Serieller Port

Ersteller: Florian Boemmel

12.1. 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.

12.2. Grundlagen

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



12.2.1. 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 unserem 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 können.

12.3. 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:

- **Canonical 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.
- **NonCanonical Mode:** Dieser Modus ist im Gegensatz zum Canonical 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.



12.3.1. Öffnen und Schließen

Zum Öffnen eines seriellen Ports unter Linux wird der Systemaufruf [open\(2\)](#) verwendet:

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

fd: File-Deskriptor

/dev/ttyUSBO: 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 serielle Port erfolgreich geöffnet, erhält fd einen positiven Wert. Im Fehlerfall liefert open -1 zurück.

Zum schließen wird [close\(2\)](#) verwendet:

```
close(fd);
```

12.3.2. 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);  
  
cfsetispeed(&SerialPortSettings,B115200);  
cfsetspeed(&SerialPortSettings,B115200);  
  
SerialPortSettings.c_cflag &= ~PARENB;  
SerialPortSettings.c_cflag &= ~CSTOPB;  
SerialPortSettings.c_cflag &= ~CSIZE;  
SerialPortSettings.c_cflag |= CS8;  
SerialPortSettings.c_cflag |= (CREAD | CLOCAL);  
  
cfmakeraw(&SerialPortSettings);
```

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.

12.3.3. 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.



12.3.4. 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.

12.4. Probleme

12.4.1. 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.

12.4.2. 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 weiter arbeitet und z.B. ein [write\(2\)](#) zu früh ausführt. Ein [usleep\(3\)](#) von 200 Millisekunden direkt nach dem Öffnen löste das Problem dauerhaft.



12.4.3. Neustart von Arduino

Während der ersten Tests fiel 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\(3\)](#).

12.4.4. 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.

12.5. Ausblick

Die geforderten Anforderungen wurden umgesetzt und der serielle Port erwies 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 auf dem Gebiet der Performance erreicht werden.

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



13. Kommunikationsprotokoll

Ersteller: Robert Graf

Das *Inter Board Protocol (IBP)* dient als Vereinbarung von Kommunikationsregeln über eine serielle Kommunikationsschnittstelle.

Innerhalb dieses Projektes soll es die Kommunikation über eine serielle USB Schnittstelle erleichtern.

Hauptsächlich benutzte Programmiersprachen war C++. C-Code wurde, wenn vorhanden auf C++ geportet, in dem Funktionen und Daten in Klassen gekapselt wurden. Für zusätzliche Tools wurde Python verwendet. (funktionieren mit Python2 und Python3)

Entwickelt wurden Komponenten auf beiden Benutzten Plattformen, Raspberry Pi 3 und Arduino Uno.

13.1. Versionsübersicht

Der Entwicklungsverlauf wird hier durch Versionierung dargestellt. In den einzelnen Versionen werden dann die neu hinzugefügten Features beschrieben und ihre Umsetzung dargelegt.

Dabei wird zur zeitlichen Einschätzung Entwicklungsbeginn und -ende angegeben, welche mit der Stundenliste abgeglichen werden können.

Versionsnummer	Beschreibung	Entwicklungsbeginn	Entwicklungsende
0.0	Erste Versuche	13.11.2017	13.11.2017
0.1	Kontrollstrukturen Pi	14.11.2017	17.11.2017
0.2	Protokollablauf	18.11.2017	18.11.2017
0.3	Konzeptionelle Fehlerbehandlung im Protokollablauf	19.11.2017	10.12.2017
0.4	Kontrollstrukturen Arduino	6.12.2017	9.12.2017
1.0	Integrierung und Fehleranalyse	11.12.2017	11.12.2017
1.1	Fehlerbehebung	12.12.2017	19.12.2017 - 1.1.2018



1.2	Richtigstellung Protokollablauf	28.12.2017	28.12.2017
1.3 final	Finales Debugging	9.1.2018	15.1.2018

13.2. Anforderungs- und Konzeptliste

Im Folgenden werden Anforderungen im Rahmen dieses Teilprojektes aufgelistet.

Manche Anforderungen wurden erst später im Entwicklungsprozess entdeckt oder erdacht, weshalb zu jeder Anforderung auch eine Versionsnummer des ersten Auftretens und eine Versionsnummer der ersten Umsetzung gegeben ist. Zusammen mit der Versionsübersicht kann das eine zeitliche Einschätzung des Auftretens der Anforderung ermöglichen. Zu den einzelnen Versionen steht jeweils eine Liste an begnügen Anforderungen.

Dadurch das die tatsächlichen Anforderungen an das Protokoll zu Beginn der Entwicklung noch nicht genau festgelegt werden konnten, wurde eine Reihe von praktischen Anforderungen erdacht.

- [A01] Frage-Antwort-Schema

Auftritt: [V0.0]

Erste Umsetzung: [V0.0]

Auf Grund der Art einer seriellen Übertragung wird es als vorteilhaft angesehen, wenn das Protokoll ein Frage-Antwort Konzept mit einem dominanten Kommunikationspartner abbildet.

Das heißt es existieren ein Master und ein Slave. Der Slave wird über Befehle vom Master zum Handeln aufgefordert und kann dann eine Antwort im Rahmen von vorher getroffener Vereinbarungen senden.

- [A02] Identifikationsnummer → Anzahl verschiedener Befehle

Auftritt: [V0.0]

Erste Umsetzung: [V0.0]

Eine Identifikationsnummer (ID oder MID) pro Nachrichtenart hilft dem Empfänger den Zweck der Kommunikation zu erkennen. Die Integer-Größe der ID kann effizient gewählt werden und hat Einfluss auf die Menge der verschiedenen möglichen Nachrichten und muss dem Kommunikationspartner pro Ablauf einmal propagiert werden. 8 Bit sei hier die gewählte Größe → 256 verschiedene Nachrichten möglich.



- [A03] Priorisierung
Auftritt: [V0.1]
Erste Umsetzung: [V0.1]
Funktionalitäten wie Echtzeitfähigkeit können durch Priorisierung der Nachrichten ermöglicht oder erleichtert werden. Hier sollen Nachrichten mit kleinerer ID priorisiert werden.
- [A04] Maximalgröße der Payload
Auftritt: [V0.1]
Erste Umsetzung: [V0.1]
Eine Maximalgröße der Payload eines Befehls/einer Antwort wird auf 255 Bytes auf Grund von maschineller Repräsentierbarkeit Festgelegt. (8bit) Diese Größenvereinbarung wäre hinsichtlich einer geplanten Speicherung oder Übertragung der Größe interessant.
- [A05] Vereinbarung der Übertragungslängen der Payloads\\
Auftritt: [V0.1]
Erste Umsetzung: [V0.1]
Die Kommunikationspartner müssen Informationen über die Länge der verschiedenen Übertragungen besitzen, um empfangene Daten ihrem Zweck zuweisen zu können und den Start der nächsten Sendung zu ermitteln.
 - [A05.1] statisch
Auftritt: [V0.1]
Erste Umsetzung: [V0.1]
Eine statische Größenvereinbarung ist zu Beginn der Laufzeit bei allen Kommunikationspartnern bekannt.
 - [A05.2] dynamisch
Auftritt: [V0.1]
Erste Umsetzung: [V0.2]
Eine dynamische Größenvereinbarung wird bei laufender Kommunikation jedes Mal neu vereinbart. Die dynamische Art einer Nachricht muss jedoch statisch bekannt sein, sodass die Kommunikationspartner auf einen dynamischen Austausch einstellen können.

Hinsichtlich des Frage-Antwort-Schemas sollte also für jede Frage und Antwort jeweils eine Größe bekannt gemacht werden (Requestsize und Response-/Answersize). Alternativ zu einer Größe soll angegeben werden können, dass die Art einer Nachricht dynamisch ist. Die Größe muss dann während der Kommunikation ausgehandelt werden.



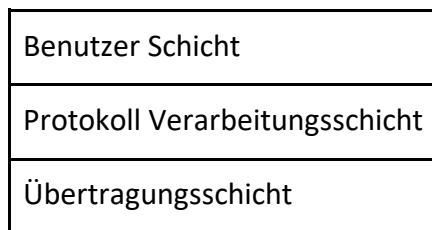
- [A06] Fehlererkennung
Auftritt: [V0.1]
Erste Umsetzung: [V0.2]
Checksummen/Hashes helfen Fehlerhafte Übertragung durch redundante Zusatzinformation zu erkennen.

- [A07] Fehlerbehandlung
Auftritt: [V0.1]
Erste Umsetzung: [V0.2]
 - [A07.1] Fehlerbekanntmachung
Auftritt: [V0.2]
Erste Umsetzung: [V0.3]
Fehler werden, wenn nötig, dem Kommunikationspartner propagiert. Um den übrigen Kommunikationsablauf dabei nicht bis wenig zu belasten, kann hierfür ein Statusfeld verwendet und mitgeschickt werden.
 - [A07.2] Negative Antwort
Auftritt: [V0.2]
Erste Umsetzung: [V0.3]
Fehler auf Slave-Seite führen zu Fehlerhaften Antwortdaten. Deshalb hat der Slave die Möglichkeit eine Negative Antwort (negative response) zu senden, die keine Antwortdaten mehr mitführt. Die Negative Antwort bietet jedoch wiederum Möglichkeiten verschiedene Fehler anzugeben.
 - [A07.3] Reaktion
Masterseitig können Fehler pragmatisch durch wiederholtes senden behandelt werden. Durch den Status kann die Kommunikationsschnittstelle hier hauptsächlich nebenbei bestimmte Konfigurierungen der Schnittstelle auf den Slaves zur Laufzeit anregen. Szenarien wie ein kontrollierter Verbindungsabbruch sind denkbar.

- [A08] plattformspezifische Umsetzung
Auftritt: [V0.1]
Erste Umsetzung: [V0.1]
Auf Grund der Unterschiedlichen Beschaffenheit der möglichen Zielplattformen können Implementierungen variieren. Die Möglichkeiten der Plattformen sollten dabei jeweils optimal ausgenutzt werden.



- [A08.1] auf Raspberry Pi 3
Auftritt: [V0.1]
Erste Umsetzung: [V0.1]
 - [A08.1.1] Der Protokollablauf findet in seinem eigenen Thread statt. Alle anderen Threads haben die Möglichkeit das Protokoll zu benutzen → Anforderung an thread-safety.
 - [A08.2] auf Arduino Uno
Auftritt: [V0.1]
Erste Umsetzung: [V0.4]
 - [A08.2.1] Der Protokollablauf muss Speichersparend ablaufen.
- [A09] Benutzerfreundlichkeit
Der Prozess der Benutzung des Protokolls soll effizient gekapselt werden. Für den tatsächlichen Protokollablauf wird so eine Schicht erstellt, in dem störungsfrei gearbeitet werden kann, ohne dass sich für die Benutzung benötigte Schnittstellen zu oft ändern würden.



- [A09.1] auf Raspberry Pi 3
Auftritt: [V0.1]
Erste Umsetzung: [V0.1]
- [A09.2] auf Arduino Uno
Auftritt: [V0.1]
Erste Umsetzung: [V0.4]

13.3. Entwicklungsvorgang

13.3.1. [V0.0] Erste Versuche

Minimalistischer, erster Versuch zum Testen von Anforderungen und Findung eventuell übersehener Basisanforderungen.

Die Funktionalität des Slaves wurde zunächst nur simuliert.

Es konnte so zunächst ohne Integrationstests entwickelt werden, da die Funktionsfähigkeit der darunterliegenden Softwareschicht (Serieller Port) bereits festgestellt wurde.



Protokollablauf trivial:

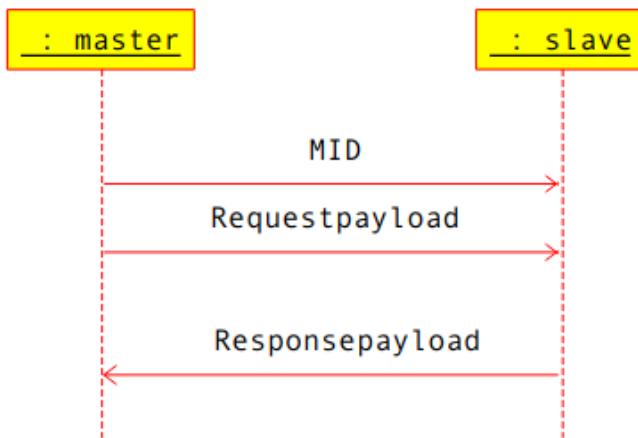


Abbildung 13.1: [V0.0] trivialer Protokollablauf

13.3.2. [V0.1] Kontrollstrukturen Pi

Mehr Überlegungen, kontrollierende Strukturen, Kapselung interner Protokollprozesse.
Die Entwicklung auf der Slave-Plattform Arduino Uno wurde zunächst vernachlässigt.

Gemäß [A08.1] wurden zunächst die Eigenschaften des Raspberry Pi eingeschätzt. Der Raspberry Pi nimmt dabei die Rolle des Masters bei der Kommunikation ein.
Folgende Eigenschaften waren dabei besonders interessant:

- Raspberry Pi 3 wurde mit Betriebssystem ausgestattet (Raspbian) → Raspberry ermöglicht Multitasking und -threading
- Für einen Controller besitzt der Raspberry vergebend viel Arbeitsspeicher (1GB) → Das Anlegen eines Buffers für Nachrichten ist nicht kritisch.
- Das Raspbian Betriebssystem hat ein Filesystem → Konfigurationsdateien direkt auf der Plattform sind möglich.

Daher Modellidee:

- Ein dedizierter Thread ist für das Senden und Empfangen, bzw. den Ablauf des Protokolls, zuständig.
- Benutzer geben ihre Befehle an den Slave als Pakete dem Thread. Die zugehörige Softwarekomponente wird **Packet** genannt.
- Eine Softwarekomponente kapselt die Funktionalität des Threads, diese wird **Transceiver** genannt.
- Ein Benutzer kann mittels einer weiteren Softwarekomponente, **Inbox**, auf Antworten warten, bzw. Antworten werden dem Benutzer überbracht.



- Informationen über die Länge der Payload eines Request oder einer Response kann über eine Konfigurationsdatei bewerkstelligt werden. Das einlesen der Datei und die Bereitstellung der Information wird durch die Komponente **Rule** verwaltet.

Dadurch können Anforderungen bedient werden:

- [A01] Die Frage wird dem Transceiver übergeben, die Antwort landet in der *Inbox*.
- [A02] **Packeten** wird eine Nachrichtenart, ID, zugeschrieben.
- [A03] Der Thread kann selbst entscheiden in welcher Reihenfolge angekommene **Packete** verschickt werden. Damit ist Priorisierung umsetzbar.
- [A04] **Packete** fassen ein Maximum an Übertragungsdaten ein.
- [A05.1] Eine über **Rule** eingelesene Konfigurationsdatei enthält Informationen über die Größe der Payload für eine Nachrichtenart (ID).
- [A08.1] Raspberry Eigenschaften wurden ausgenutzt (Threading, etc.)

und erzeugt weitere Annehmlichkeiten:

- Benutzer können in ihren eigenen Threads arbeiten, d.h. verschiedenen Komponenten können potentiell zeitgleich Sendeaufrufe tätigen und Antworten abrufen.
- Tatsächliche Kommunikation läuft zentral (nicht jeder Benutzer baut eine eigene Lösung) → Kommunikation leichter zu verwalten.

Es entsteht dadurch auch die neue Anforderung [A8.1.1] zur thread-safety, die hauptsächlich zur Absicherung gegen Race-Conditions dient.

13.3.2.1. Implementierung

Die Implementierung sieht für jede der besprochenen Komponenten eine Klasse vor:

13.3.2.1.1. Packet

Zur Klärung: **Packet** steht im Englischen natürlich für Paket. In diesem Dokument wird jedoch hauptsächlich die Komponentenbezeichnung benutzt und daher mit 'ck' geschrieben.

Das **Packet** dient als Speicherort für zu sendende und empfangene Daten und als Paket/Bündel des Inhalts für die Logik.



Packet
- id : int
- size : int
- data : char*
+ Packet(id : int, size : int, data : char*)
+ Packet(id : int, size : int)

Abbildung 13.2 : [V0.1] Packet

Tatsächlich wurden intern die meisten **Packete** mit sog. Smart-Pointern speichermäßig verwaltet, was das mehrfache kopieren eines **Packetes** meistens verhindert → Performance. Die Klasse bedient [A02] [A04].

13.3.2.1.2. Transceiver

Der **Transceiver** dient als Sender und Empfänger, als aufrufende Instanz für den Seriellen Port und als Organisator der Übertragung.

Transceiver
- worker : Thread
- sendbuff_lock : mutex
- rule : Rule
- sendbuffer : priority_queue<Packet>
- s : Serial
+ send(p : Packet)
+ addreceiver(i : undef, id : int)
+ removereceiver(i : undef, id : int)
+ runworker(run : bool)
- store(p : Packet)
- transfer_sendbuffer()

Abbildung 13.3: [V0.1] Transceiver

Beschreibung:

- worker : kümmert sich um den Protokollablauf [A08.1.1] und arbeitet dabei den sendbuffer ab.
- sendbuff_lock : schützt den sendbuffer vor Fremdzugriff [A08.1.1]
- rule : enthält Konfigurationsinformationen [A05]
- sendbuffer : speichert zu sendende **Packete** aufsteigend sortiert nach Packet-ID [A08.1.1]. Die Sortierung hat zur Folge, dass die Bearbeitungsreihenfolge der Packete organisiert wird → Priorisierung [A03]
- s : kann senden und empfangen
- send () : ein Requestpacket gelangt so thread-safe in den sendbuffer [A08.1.1]



- add/removereceiver () : meldet eine Inbox für das Empfangen von Nachrichten einer bestimmten Nummer an/ab. [A08.1]
- runworker() : startet/stoppt den worker Thread [A08.1.1]
- store() und transfer_sendbuffer() sind interne Funktionen, die thread-safety bei der Übertragung von **Packeten** von und zum worker-Thread ermöglichen. Dabei verwendet transfer_sendbuffer() z.B. die sendbuff_lock

13.3.2.1.3. Inbox

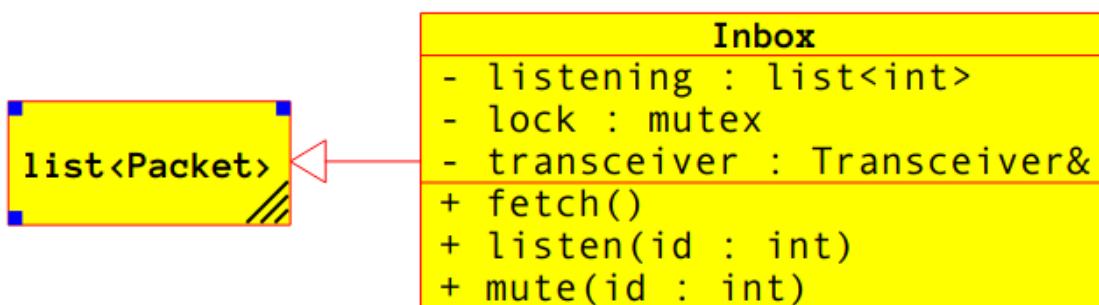


Abbildung 13.4 : [V0.1] Inbox

Beschreibung:

- Die **Inbox** ist eine Liste von Antwort**packeten** und bietet damit alle Funktionalitäten einer C++ std::list
- Die **Inbox** hat eine Liste an ID's, denen sie "zuhört", d.h. wenn Antworten für diese ID am **Transceiver** ankommen, bekommt diese Inbox eine Version dieses Antwort**packets**. [A08.1]
- Besitzt eine lock, da sie als Interface zwischen **Transceiver** und User Race-Conditions ausgesetzt ist. [A08.1.1]
- Besitzt eine Referenz auf den Transceiver um sich nach dem RAI Prinzip selbstständig dort für seine ID's an und abmelden kann. Der Benutzer hat dadurch diesen Aufwand nicht. [A08.1]
- fetch() holt alle angekommenen **Packete** der ID's, denen die sie zuhört, in die **Inbox**. Dieser Aufruf wird gebraucht, da üblicherweise der **Transceiver** die lock beansprucht. fetch() sperrt die lock, holt alle **Packete** und gibt die lock wieder frei. [A08.1.1] So kann auch die komplette Funktionalität der std::list behalten werden, da diese nun nicht mehr extra abgesichert werden muss, um Race-Conditions vorzubeugen.
- mit listen() und mute() kann sich potentiell für eine oder mehrere IDs beim Transceiver an/abgemeldet werden.



13.3.2.1.4. Rule & Konfigurationsdatei

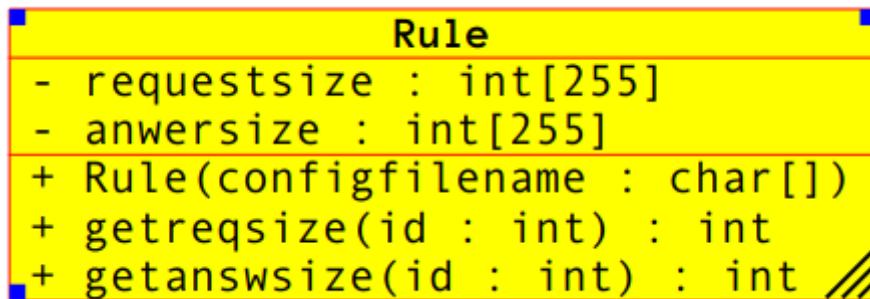


Abbildung 13.5 : [V0.1] Rule

Rule liest die Konfigurationsdatei aus und macht die Werte zugänglich.

```

24 101 0 0
25 180 0 2
26 181 0 2
27 182 0 3
28 183 0 6
29
30 #example
31 254 4 8
32 253 16 4
33
34 #general purpose message
mand is issued with lowes
35 #this can be used to pass
36 #255 0 255
37

```

Abbildung 13.6 IBC_Config

Die Konfigurationsdatei wird in dem Muster [ID] [Requestsize] [Anwersize] beschrieben. Kommentare in der Datei sind mit einem '#' am Anfang der Zeile realisiert.

Nach der Implementierung wurde der triviale Protokollablauf aus V0.0 verwendet, um zu testen.

13.3.3. [V0.2] Protokollablauf

Anforderungen erfüllt : [A01] [A02] [A03] [A04] [A05.1] [A05.2] [A06] [A08.1] [A09.1]

Legende:

MID	Identifikationsnummer
SIZE	optionales Feld für die Größe bei dynamisch vereinbarter Übertragung
H	Hash/Checksumme der Übertragungsdaten
Payload	Nutzdaten



Request:

MID	SIZE	Payload	H
-----	------	---------	---

Response:

SIZE	Payload	H
------	---------	---

Bezug zu Anforderungen:

[A01] Ein optimaler Ablauf wird als recht unkompliziert Frage-Antwort-Schematisch angenommen, in dem in einem Stück Daten hin und darauf Daten zurückgesendet werden. In anderen Worten ein Request geht in einem Stück zum Slave, der sendet in einem Stück seine Response.

[A05.2] Optional kann ein Feld für die Länge der Payload mitgesendet werden, um dynamische Frames zu ermöglichen. Die optionale Routine würde ausgelöst werden, wenn in der Konfigurationsdatei (über >Rule< eingelesen) ein bestimmter, zu großer Wert (255) als statische Größe stünde → size = 255 → dynamisch . Ein Benutzer kann also 255 als Größe definieren und ein >Packet< mit beliebiger Größe im Rahmen der Maximalgröße [A04] versenden. Das ist sowohl bei Requests, als auch bei Responses oder bei beiden gleichzeitig möglich.

```

37 #dynamisch hin und danamisch zurück
38 243 255 255
39 #dynamisch hin und statisch zurück
40 254 255 3
41 #statisch hin und dynamisch zurück
42 255 0 255

```

[A06] Eine Checksumme macht es dem Kommunikationspartner möglich die Korrektheit der übertragenen Daten festzustellen. Als Checksumme/Hash wurden dabei meistens alle gesendeten Bits effizient in einer bestimmten Schrittweite exklusiv verodert.

Hashing / Bildung von Checksummen

Verwendete Hashfunktionen wurden selbst erstellt, jedoch nach einem Schema:

- Die Bitlänge des Hashes Lh ist praktisch immer als Potenz von 2 gewählt (meist 2 , 4 oder 8)

$$L_h \in \{i : \mathbb{N} | 2^i\}$$

- Die zu hashenden Daten der Bitlänge Ldsind immer ein Vielfaches von 2 (tatsächlich meist von 8 da Daten meist byteweise behandelt werden)

$$L_d \in \{i : \mathbb{N} | 2i\}$$

Nun kann die Länge Ldin Schritten der Länge Lhdurchlaufen und exklusiv verodert werden.



Dabei kann ein Rest Ld modLh übrigbleiben. Dieser ist ein Vielfaches von 2 aber kleiner Lh und kann deshalb mehrmals nebeneinander geschrieben werden bis er die Bitlänge Lh wieder erreicht, um im Letzten Schritt exklusiv verodert zu werden.

Beispiel 1:

Wir hashen 20 bit in einen 8-Bit hash. Zunächst werden deshalb in 8-Bit-Schritten die Daten exklusiv verodert, bis der Rest nicht mehr reicht.

$$Ld=20 \text{ Bit}; Lh=8\text{bit} ; d = 0111\ 0011\ 0000\ 1111\ 1010$$

Schritt 1:

$$\text{XOR}(0111\ 0011, 0000\ 1111) = 0111\ 1100$$

Schritt 2: Der Rest (1010) reicht nicht aus, deshalb schreiben wir ihn mehrmals hintereinander auf und exclusiv verodern ihn daraufhin:

$$\text{XOR}(0111\ 1100 ,1010\ 1010) = 1101\ 0110$$

Beispiel 2:

$$Ld=26 \text{ Bit}; Lh=4 \text{ Bit} ; d = 1110\ 1001\ 1111\ 0000\ 1010\ 1111\ 10$$

- Schritt 1:

$$htemp = \text{XOR}(\text{XOR}(\text{XOR}(\text{XOR}(1110, 1001), 1111), 0000), 1010), 1111) = 1101$$

- Schritt 2 :

$$\text{Rest} = 10 \rightarrow \text{nebeneinander bis Länge} = 4: 1010$$

$$h = \text{XOR}(htemp, 1010) = 0111$$

13.3.4. [V0.3] Konzeptionelle Fehler im Protokollablauf

Probleme: Der in [V0.2] entwickelte Ablauf weist konzeptionelle Fehler auf.

Request:

MID	SIZE	Payload	H

Response:

SIZE	Payload	H

Bei einer fehlerhaften Übertragung von **MID** oder **SIZE** kann die Länge der **Payload** nicht ermittelt werden. Das bedeutet, dass der Empfänger keine Möglichkeit hat zu erfahren **H** in der Übertragung überhaupt befindet. Schlimmer: Der Empfänger kann den Beginn eines neuen Requests nicht mehr ermitteln und die Kommunikation wird bei darauffolgenden Übertragungen immer fehlerhaft. Potentiell könnte über Hash **H** die fehlerhafte



Übertragung erkannt werden, aber da der Empfänger nicht ermitteln kann, wo diese ist, ist dies nutzlos.

Lösung: Ein dedizierter Header mit einem eigenen Hash und statischer Größe kann dieses Problem lösen. Idealerweise Enthält der Header die Felder **MID**, **SIZE** und **H**. Da das **SIZE** Feld jedoch optional ist und bei statischen Nachrichten nicht mitgeschickt werden soll würde der Header nicht mehr statische, sondern dynamische Größe haben. Deshalb wird der Header an dieser Stelle getrennt und jeder Teil selbst gehasht.

[A07.1]

Zusätzlich wurde in dieser Version noch Statusübertragung entwickelt.

Das Statusfeld wird verwendet um den Status der Kommunikationspartner zu kommunizieren.

Die übliche Datenübertragung kann dabei nebenher unbeeinträchtigt weiterlaufen.

Das Statusfeld ist 4-Bit lang => 16 verschiedene Stati möglich.

Die Bedeutung des Status kann je nach Art des Kommunikationspartners (Master oder Slave) variieren, d.h. der Status 8 bedeutet beispielsweise auf dem Master etwas anderes als auf dem Slave.

Masterseitig:

- Bit 0 wird als STOP Befehl benutzt. Sein Versand bedeutet das Ende der Kommunikation.
- Bit 1 wird als REINIT Befehl benutzt. Sein Versand fordert eine Reinitialisierung der Kommunikation auf Slave-Seite.
- Bit 2 und 3 wird verwendet um eine erneute Sendung mit einer Nummer 0-3 zu kennzeichnen. Dies geschieht, wenn eine Sendung im Vorhinein fehlgeschlagen hat. Eine Sendung kann bis zu 3 Mal wiederholt werden.

Slaveseitig:

- Bit 0 wird verwendet um einen internen Fehler am Slave zu propagieren, der zur Folge hat dass dieser nicht korrekt oder gar nicht auf Befehle reagiert. Der Master kann dadurch auf den Zustand reagieren um zum Beispiel den Fehler zu loggen, den Sendevorgang für eine bestimmte Zeit einzustellen, Systeme kontrolliert herunterfahren zu lassen, etc.
- Bit 1, 2 und 3 werden zum Kommunizieren von Fehlern der 3 Checksummen des Protokolls wie folgt verwendet.

[A07.2] Negative Antwort

Das Statusfeld hat im Kontrollfluss die wichtige Aufgabe, dem Master eine Möglichkeit zu geben, zwischen positiver oder negativer Antwort zu unterscheiden. Slaveseitig kann dafür das Statusbit 0 verwendet werden.

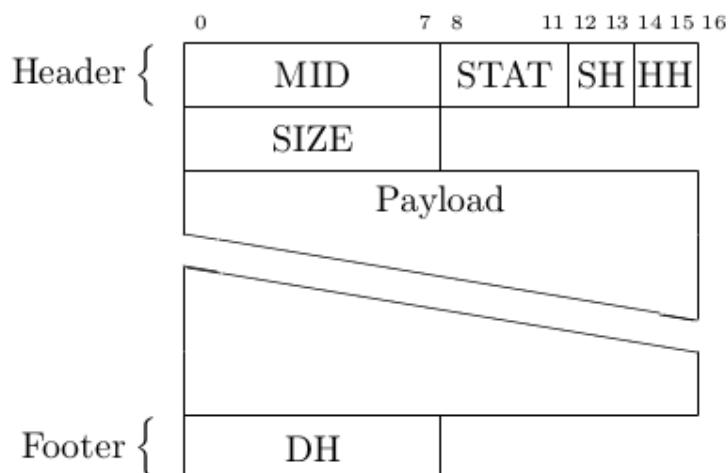


Legende:

MID	Identifikationsnummer
EID	Nummer des Fehlers (Benutzerdefiniert)
SIZE	optionales Feld für die Größe bei dynamisch vereinbarter Übertragung
HH	Hash/Checksumme des Headers
SH	Hash/Checksumme des dynamischen Größe (0 Wenn statisch)
DH	Hash/Checksumme der Nutzdaten
STAT	Status
Payload	Nutzdaten



Request



Response

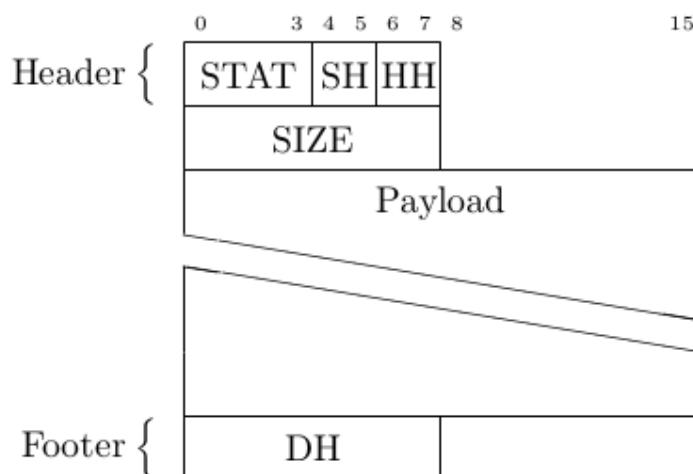


Abbildung 13.7 [V0.3] Request und Response

Negative Response

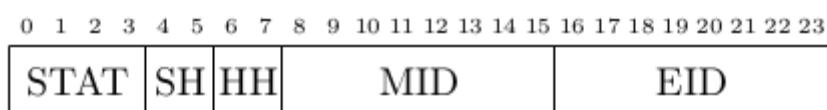


Abbildung 13.8 [V0.3] Negative Response

[A9.1]

Auf dem Raspberry Pi wurde eine zusammenfassende Komponente implementiert, welche den Benutzern eine klare Benutzungsschnittstelle liefert und dabei Funktionalität, welche nicht angedacht ist vom Benutzer verwendet zu werden, versteckt. Die Komponente wurde >IBC< genannt.



```

IBC
- rule : Rule
- transceiver : Transceiver
+ send(p : Packet)
+ getInbox(id : int) : Inbox
+ IBC(device_name : string, configfile : string)
```

Abbildung 13.9 [V0.3] IBC

- wird erstellt mit einem Bezug zum Device-File des Seriellen Ports auf dem Raspberry, meist “/dev/ttyUSB0” und einem Bezug zur Konfigurationsdatei
- send() reicht Packete an den Transceiver weiter.
- getInbox erzeugt eine bereits mit dem Transceiver verknüpfte Instanz einer Inbox, die auf eine bestimmte ID hört.

Das führt zu folgendem Benutzungsbild:

Hier wird eine Inbox auf dem Heap angelegt, Stack ist natürlich auch möglich!

In der Konfiguration wurde 254 4 16 eingestellt.

```

29 int run (IBC* ibc)
30 {
31     char buff [4] = "Hi!";
32
33     Inbox *i = new Inbox(ibc->getInbox(180));
34
35     Packet p (254, 4,(uint8_t*) buff);
36     ibc->send(p);
37
38     //wait for an answer to arrive
39     std::this_thread::sleep_for(std::chrono::seconds(1));
40
41     i->fetch();
42
43     if(i->size())
44     {
45         std::cout << i->front() << '\n';
46     }
47
48     delete i;
49
50     return 0;
51 }
```

Abbildung 13.10 [V0.3] Pi-Codebeispiel

Der komplette Aufbau lässt sich zusammengefasst mit einem Klassendiagramm darstellen:

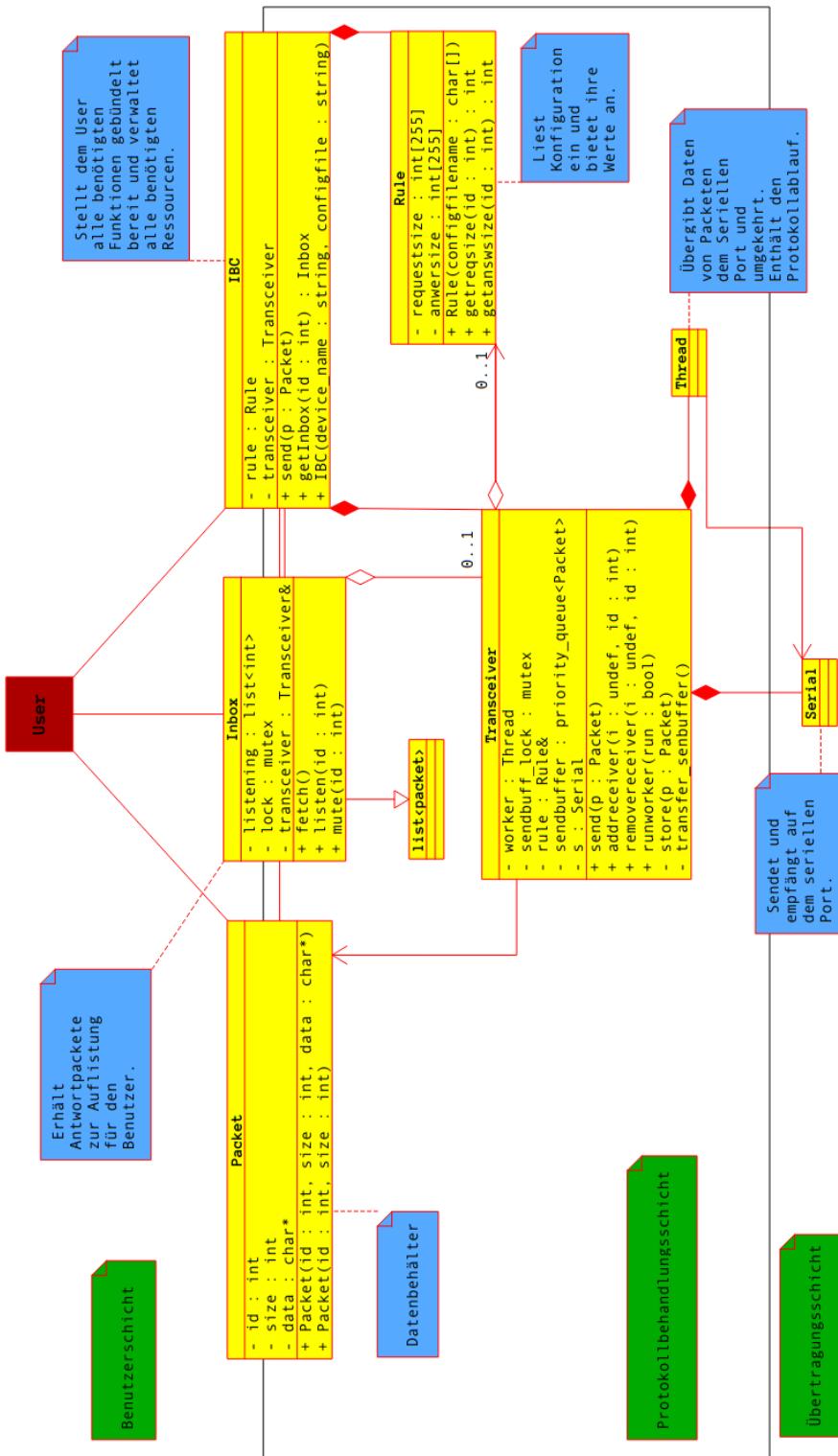


Abbildung 13.11 [V0.3] Klassendiagramm



13.3.5. [V0.4] Kontrollstrukturen Arduino

In dieser Version wurde sich der Entwicklung von Komponenten auf der Slave-Plattform Arduino Uno angenommen.

[A08.2] Dabei waren folgende Eigenschaften der Plattform interessant:

Arduino Uno führt nur sequenziell Befehle aus, kein paralleler Ablauf von Code ist möglich. Bestimmte Aufgaben können der Hardware übergeben werden, was jedoch für das Protokoll relativ uninteressant ist.

Arduino Uno besitzt verhältnismäßig wenig Arbeitsspeicher (2Mb). Wenn man nun die unter [A04] angesprochene Maximalgröße einer Datensendung bedenkt (255 Bytes), würde das bereits 1/8 des Speichers einnehmen. Das ist inakzeptabel.

Deshalb Modellidee:

Eine Komponente/Klasse IBC auf dem Arduino Uno stellt alle benötigten Funktionalitäten bereit.

Benutzer machen alle Aufrufe zum Senden und Empfangen des variablen Teils der Daten, welche gesendet werden, selbst. Dabei fügen sie berechnenden Code direkt hinzu.

Der übrige Teil des Protokollablaufs wird den Benutzern durch Codegeneration abgenommen.

Die Codegeneration hilft den Benutzern durch Kommentierung und Beispielcode.

Dadurch können alle nötigen Befehle sequentiell in den Programmablauf direkt eingebettet werden. Das Speicherproblem wird auf den Stack verlegt, d.h. es findet kein Buffering von Nachrichten statt. Stattdessen können Daten direkt aus dem Bereits benutzten Speicher des Benutzers gesendet oder in diesen geschrieben werden.

Eine effiziente Kapselung wird dadurch jedoch leider in den meisten Teilen verhindert und [A09.2] ist leider nur bedingt umsetzbar.

Codegenerator:

Anforderungen an den Codegenerator:

Code soll nur in einer Datei generiert werden müssen.

Generierungsparameter sind die Daten aus derselben Konfigurationsdatei wie für die masterseitige Komponente.

Generator kann wiederholt auf derselben Datei ausgeführt werden.

Generator erhält von Benutzern geschriebenen Programmcode.

Der Codegenerator wurde in Python erstellt.

1 Mittels durch Code-Kommentare realisierter Tags sucht der Generator die Stelle zum generieren.



```
/* IBC_FRAME_GENERATION_TAG_BEGIN */  
/* IBC_FRAME_GENERATION_TAG_END */
```

- 2 Er sucht daraufhin alle zu erhaltenen Stellen im Ursprungsdokument.

```
/* IBC_PRESERVE_RECV_BEGIN 252 vvvvvvvvvvvvvvvvvvvvvvvvvvvvvv */  
/* IBC_PRESERVE_RECV_END 252 ^^^^^^^^^^^^^^^^^^ */  
/* IBC_PRESERVE_SEND_BEGIN 252 vvvvvvvvvvvvvvvvvvvvvvvvvvvv */  
/* IBC_PRESERVE_SEND_END 252 ^^^^^^^^^^^^^^ */
```

- 3 Über die Konfigurationsdatei werden nun für jede konfigurierte Nachrichtenart entsprechende Code-Fragmente erstellt. Dabei werden Eingabeparameter aus der Konfigurationsdatei berücksichtigt.

```
/* IBC_MESSAGE_BEGIN 253 255 2 */  
/* IBC_MESSAGE_END 253 255 2 */
```

- 3.1 Dabei wird der zu kapselnde Ablaufcode außerhalb der Preservierung auf die neueste Version gebracht.
- 3.2 Falls unter 2tens zu erhaltener Code gefunden wurde wird dieser wieder eingefügt und dadurch erhalten.
- 3.3 Falls unter 2tens kein Code gefunden wurde wird Beispielcode generiert unter einbezug der Konfiguration generiert und eingefügt.

```
/* IBC_PRESERVE_RECV_BEGIN 252 vvvvvvvvvvvvvvvvvvvvvvvvvvvv */  
    byte buffr252[4];  
    recv(buffr252,4);  
    //DONT FORGET TO HASH  
    setDH(createDH(buffr252,4));  
/* IBC_PRESERVE_RECV_END 252 ^^^^^^^^^^ */  
/* IBC_PRESERVE_SEND_BEGIN 252 vvvvvvvvvvvvvvvvvvvvvvvvvv */  
    byte buffs252 [8] = {1,2,3,4,5,6,7,8};  
    for(int i = 0 ; i<8;i++) {send(0);}  
    //DONT FORGET TO HASH  
    setDH(createDH(buffs252, 8));  
/* IBC_PRESERVE_SEND_END 252 ^^^^^^ */
```

- 3.4 Unter Beachtung der Konfiguration werden erklärende und helfende Kommentare generiert.

```
/*Send exactly 8 bytes in the following */  
/*If you have a dynamic size you have to send this size first! */
```



```
/*Also calculate their data hash along the way by */
/* xorring all bytes together once */
/* or use the provided function createDH(..) */
/* Make the hash public to the IBC by setDH(Your DATAHASH HERE) */
/* IBC_PRESERVE_SEND_BEGIN 252 vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv*/*
```

- 4 Der Generator schlägt fehl, wenn die Ursprungsdatei und die Konfiguration divergieren. Das ist gewünschtes Verhalten. Nicht konfigurierte Nachrichten sollten daraufhin aus dem Code entfernt und der Generator erneut benutzt werden. Der Benutzer muss sich beim Entwickeln seiner Nachricht nun nur an folgende Regeln halten:
 - Benutzercode wird nur in den angegebenen PRESERVE Tags eingefügt
 - Die in der Konfigurationsdatei angegebene Anzahl an Bytes muss gesendet, bzw. empfangen werden. Generierte Kommentierung weist ausdrücklich darauf hin. Auch müssen die Daten gehasht werden und ein Hashwert gesetzt. Dieser wird dann benutzt um den letztendlichen Hash zu erstellen, der über die Leitung geht.

Output Hilfe:

```
pi@raspberry ~/DT_WS1718_02_StarCar/uno $ ./Uno_ibcgeneration.py
Help for this code generator.

This generator generates code in a correctly tagged code file in c++ style.

Arguments : [Rule] [Output]
  Rule  Is a file which specifies an IBC ruleset in form of numbers in a pattern [MID]
  [REQSIZE] [RESSIZE]
  Output Is a file where the output will go. It will be searched for a tags which look
like this.

  /* IBC_FRAME_GENERATION_TAG_BEGIN */
  /* IBC_FRAME_GENERATION_TAG_END */

  this script will not touch any lines outside these tags. This script will terminate when
no tags are found in [Output]

  This script will also preserve custom code inside of tags

  /* IBC BEGIN [MID] [RECV/SEND]*/
  /* IBC END [MID] [RECV/SEND]*/
```

Benutzung:

- *Rule* zielt auf die Konfigurationsdatei ab.
- *Output* auf das File in dem generiert wird.

```
pi@raspberry      ~/DT_WS1718_02_StarCar/uno      $      ./Uno_ibcgeneration.py
..../pi/IBC_config.cfg IBC.cpp
```



Wichtig: Die Konfigurationsdatei liegt im Projektverzeichnis beim pi-seitigen Code, da dieser diese direkt einliest. Achten Sie bei der Benutzung auf korrekte Pfade.

Nachdem diese Version erstellt wurde, wurde auf den Zielplattformen standalone integriert und getestet, d.h. keine anderen Komponenten außer die Kommunikation liefen zu der Zeit auf den Zielplattformen simultan mit.

Einschub: Der Codegenerator sollte nicht auf die finalen Versionen nicht angewendet werden, da Kollegen doch Code außerhalb der erlaubten Tags verändert haben, nachdem alle nötigen Nachrichtentypen generiert waren. Zum Testen empfehle ich deshalb eine Textdatei mit dem unter 1. angegebenen Inhalt anzulegen und die Codegeneration darauf auszuführen.

13.3.6. [V1.0] Integrierung und Fehleranalyse

Diese erste volle Version wurde in die übrige Projektumgebung integriert. Tests positiv, Kollegen wurden in der Benutzung geschult.

Problem: Nach einiger Zeit traten Speicherzugriffsfehler (Segmentation Faults) auf. Diese Fehler sind meist durch undefiniertes Verhalten verschuldet, welches letztendlich zu einem falschen Speicherzugriff führt.

Debug: Mittels GNU-Debugger gdb wurde ein Backtrace des Programmablaufes angelegt. Der Ablauf wanderte dabei in protokollspezifischen Code (Inbox-Pointer-sets in Transceiver, welche die Adressen zu Inboxen speichert), daraufhin in die Standardbibliothek und dort passierte dann der Fehler. Aufgrund dieses Bildes wurde die Schuld dem Protokollcode gegeben.

Auf Grund der folgenden Fehler wurde zu Dokumentationszwecken ein Snapshot dieser Version erstellt. Dieser kann im Projektverzeichnis unter "./docs/team/grr37213/versions/1_0/" gefunden werden. Der pi-seitige Code kann simuliert getestet werden: 1. Wandern Sie in das unterverzeichnis "pi/" 2. Überprüfen sie die Existenz eines "makefile"s in diesem Verzeichnis und führen Sie "make" aus. 3. Führen Sie den nun kompilierten UnitTest "./IBCEmu" aus. Es wird davon ausgegangen, dass eine Kompilierumgebung (g++ -std=c++14 -lpthread) auf Ihrer Plattform existiert. Der Test zeigt daraufhin den Inhalt von 3 Inbox-Objekten nach gesendeten Nachrichten an. Der Code zum Test liegt in "IBC.test.cpp"

Auch die Codegenerierung kann in diesem Snapshot getestet werden. Dazu wandern Sie in das Unterverzeichnis "uno/" und führen Sie die unter [V0.4] beschriebenen Schritte zur Codegenerierung aus. Achten Sie bei der Benutzung des Generierungsscripts auf korrekte Pfade.



13.3.7. [V1.1] Fehlerbehebung

Der in Version [V1.0] aufgetretene Fehler sollte hier behoben werden.

Zunächst wurde manuell gesucht. Nach erheblichem Zeitverlust durch die Suche wurde zu versuchen, den Fehler zu vermeiden. So wurde ein Großteil des Sourcecodes refactored um Aufrufe der Standardbibliothek zu vermeiden. Leider ohne Erfolg. Es traten Seg-Faults an anderer Stelle auf. Bei dem Refactoring wurden Features teils nicht reimplementiert:

- [A05.2] Es zeichnete sich ab, dass dynamische Größen nicht gebraucht werden würden.
- [A07.2] Ebenso wie würde negative Antworten wohl nicht benutzt werden.
- [A07.3] Solcherlei Fehler traten viel zu wenig auf, als dass sich die Reimplementierung unter Zeitdruck gelohnt hätte.

Es wurde danach vermutet, dass die neu eingeführte Entwicklungsplattform des Kollegen für die GUI (Qt via QtCreator) nicht kompatibel war. Es wurden deshalb alle Aufrufe der Standardbibliothek auf die umfangreiche Qt-Bibliothek umgeschrieben.

Mit zeitlichen Pausen in der Entwicklungszeit von ca. einer Woche, wurden trotzdem wieder Seg-Faults festgestellt.

Nach ca. einer weiteren Woche entdeckte der Kollege die falsche Benutzung eines Pointers. Der Kollege hatte ein Objekt der Klasse **IBC** auf dem Heap angelegt und den Pointer dazu überschrieben. Das hat zur Folge, dass alle Ressourcen des Protokollcodes von Benutzercode falsch adressiert werden. Das führt dazu, dass aufrufender Code sogar noch in logische Anweisungen des Protokollcodes wandert, jedoch bei einem Speicherzugriff an die falsche Stelle greift.

Ergebnis: Der Fehler war nicht durch das Protokoll verschuldet gewesen. Die komplette Arbeit unter dieser Version war also effektiv nichts Wert. Jedoch wurden Teile des Refactoring übernommen, um wegen nun nicht mehr gebrauchter, nicht vorhandener Features die Vorteile einer schlankeren Codebasis ausnutzen zu können. (leichter zu lesen, zu verwalten, Protokollablauf ohne dynamische optionale Größe z.B. leichter)

Persönliche Einschätzung:

Der Zeitverlust und der Verlust bereits ausgearbeiteter Features war für meinen eigenen geplanten Projektablauf ein Desaster. Tatsächlich wollte ich noch an etwas anderes als dem Protokoll arbeiten. Meine ursprünglich angedachte Hauptaufgabe war die Kartographierung.

In der Situation waren folgende Fragen für die Entscheidung meines weiteren Vorgehens von Wichtigkeit:

- Ist das Protokoll bei Nichterfüllung für den Projekterfolg von Kollegen potentiell gefährlich? Antwort: Ja, da potentiell Sensor und Motordaten nicht übertragen werden können.



- Wie viel Zeit bleibt noch im Projekt und welches Teilprojekt ist dabei für das Gesamtprojekt von größerem Nutzen? Antwort: Das Protokoll gliedert sich mehr in die Arbeit der übrigen Kollegen mit ein. Kartographierung ist ein mehr separates Projekt.
- Sind die zur Wahl stehenden Teilprojekte überhaupt im derzeitigen Stand umsetzbar? Antwort: Die Kartographierung ist abhängig von vororganisierten Sensorwerten von der Raumerkennung. Zu dieser Zeit war noch nicht vollends klar, welche Sensoren funktionieren und noch nicht alle waren fertig integriert.

An dieser Stelle wurde deshalb beschlossen, Kartographierung bis auf Weiteres auf Eis zu legen und sich weiter um das Protokoll zu kümmern.

Nun war der Großteil der angedachten Arbeitszeit schon dafür verbraucht und das Endprodukt genügte nicht mehr meinen eigenen Vorstellungen, da nun Features fehlten, deren Reimplementierung oder Reintegration als zu Aufwändig, bzw. nicht mehr zweckmäßig angesehen wurden (einige Features würden offenbar von den letztendlichen Benutzern gar nicht gebraucht werden, wie zum Beispiel [A05.2]). Dadurch, dass das Protokoll so von zentraler Wichtigkeit für einige Kollegen war, die Daten zwischen den Controllern versenden wollten, musste das Protokoll weiter Priorisiert entwickelt werden. Im Nachhinein betrachtet war auch das weiterarbeiten auf Teilen von [V1.1] hinsichtlich des Gesamtergebnisses des Teilprojektes ein Fehler. Stattdessen hätte komplett auf [V0.4], bzw. [V1.0], zurückgesetzt werden müssen, um auch die nicht gebrauchten Features zu behalten. Das wurde jedoch unterlassen um die Übersicht nach dem Fehlerchaos zu behalten.

13.3.8. [V1.2] Richtigstellung Protokollablauf

Der Protokollablauf aus [V0.4] ist immer noch ungenügend. Einfaches Beispiel:

Wir nehmen an das **MID** falsch übertragen wird. Der Empfänger erkennt die falsche Übertragung anhand des Headerhashes. Der Protokollablauf sieht jedoch an dieser Stelle keinen Abbruch vor. Schlimmer: Der Master wird den Rest seiner Übertragung senden, obwohl der Slave auf Grund der fehlerhaften Übertragung wieder nicht weiß wo die Übertragung endet. Bis dato wurde in diesem Fall einfach eine bestimmte Zeit gewartet, alle bis dahin empfangenen Daten verworfen und eine negative Nachricht zurückgesendet. Da dies jetzt keine Option mehr darstellt (und Wartezeit niemals eine optimale Lösung darstellt) muss der Protokollablauf wieder geändert werden.

Erdacht wurde die im folgenden Sequenzdiagramm dargestellte Lösung :

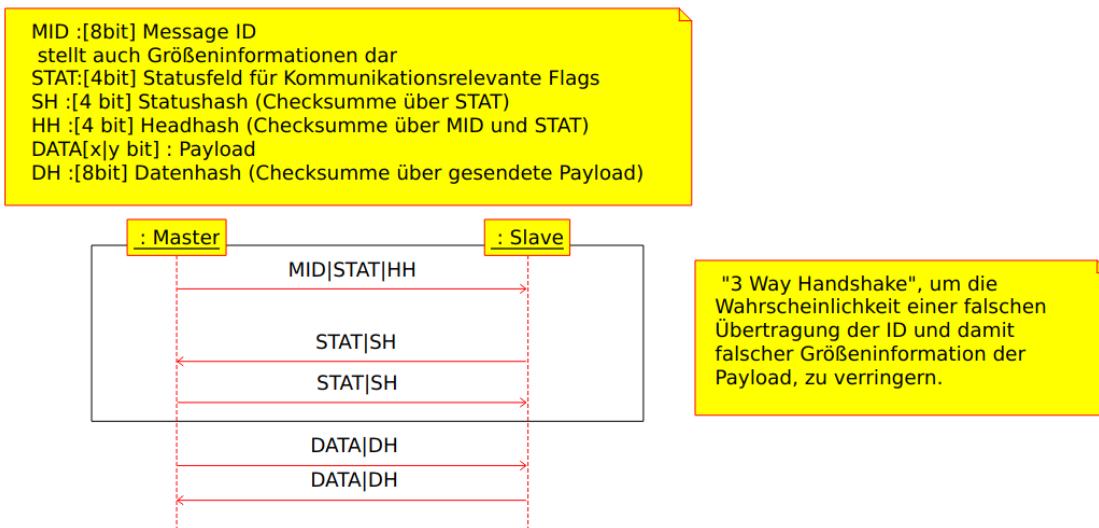


Abbildung 13.11 [V1.2] Ablaufsequenzdiagramm

Durch den Status-Handshake kann dieser Fehler direkt behandelt werden.

Ein Problem stellte sich bei der genaueren Entwicklung des Status-Handshakes dar. Es ist ersichtlich, dass das letzte übertragene Byte des Handshakes immer kritisch bleiben wird, im Falle, dass es falsch übertragen wird. Ein Beispiel :

Handshake :

1. Master sendet ID 5.
2. Slave sendet das die ID korrekt Empfangen worden ist. OK.
3. Master schließt den Handshake ab.

Nun ist es für den Master unwichtig, ob Schritt 3 denn tatsächlich richtig empfangen worden ist. Für den Fall der korrekten Übertragung ist alles im Rahmen, jedoch im Fall der inkorrekten Übertragung muss der Slave sich trotzdem auf die daraufkommende Flut an Daten einstellen.

Hat der Slave in Schritt 2 nicht OK, sondern einen Fehler gemeldet, soll der Master den Ablauf neu beginnen. Dazu signalisiert er dem Slave in Schritt 3 einen Ablaufneustart. Wieder kann der Slave in diesem Fall dieses Signal falsch erhalten und unter Umständen nicht interpretieren.

Die Krux der Überlegungen an dieser Stelle ist vor allem, dass sie dazu verleiten im Kreis zu denken. Es gäbe

nahe an 100% sichere Methoden an dieser Stelle, die hauptsächlich auf rekursiven Sendaufufen beruhen, diese sind jedoch für den Rahmen des Projektes nicht angemessen (Auto ist kein Mars-Rover).

Der Handshake wie beschrieben bietet eine Verbesserung der Situation, jedoch keine 100% Lösung. Das soll als angemessen angesehen werden.

Der Handshake und der neue Protokollablauf wurden implementiert, die Behandlung von propagierten Fehlern jedoch nicht.



13.3.9. [V1.3] Finales Debugging

Problem: Bei der erneuten Integrierung des Teilprojektes wurden Sensordaten (Slave zu Master) nicht richtig übertragen. Langwierige Untersuchungen des Problems zeigten beim Mithören auf dem Port unerklärliche Sendeaufrufe, welche nicht im Protokollablauf codiert waren. Nach dem Fehler wurde gesucht, er wurde jedoch im noch zur Verfügung stehenden Zeitrahmen nicht mehr rechtzeitig gefunden. So wurde der Handshake und [V1.2] weitestgehend zurückgebaut. Integrationstests konnten dabei nicht selbstständig ausgeführt werden, da zu den benötigten eigens konfigurierten Technologien zum bespielen/flashen des Arduino gegen Ende der Entwicklungszeit nicht jeder direkt Zugang hatte. So wurde in einem Unit Test getestet. Dieser zeigte nach dem Rückbau wieder ordentliche Übertragung.

In der letzten Woche wurde Protokollcode erneut integriert. Dabei wurden unerklärlicherweise wieder Übertragungsfehler erzeugt. Der Protokollablauf an dieser Stelle war quasi der von [V0.1] mit der zusätzlichen Infrastruktur der späteren Versionen, was die Sache noch unerklärlicher machte.

Aus Zeitmangel wurde auf eine vom Kollegen Bömmel und Scharnagel erstellte Backup-Lösung zurückgegriffen. Gegen Ende der Entwicklungszeit mit allen tatsächlich gebrauchten Anforderungen ausgestattet, konnte in dieser eine statische Anzahl an Bytes für alle Sensordaten gemeinsam verschickt werden. Der komplexe Overhead der IBC-Komponenten wurde dafür nicht gebraucht.

13.4. Fazit

Die Erstellung eines Protokolls im Embedded Bereich ist ein interessantes Thema. Innerhalb dieses Teilprojektes wurde versucht, viele bekannte Problemstellungen abzudecken, viele Funktionalitäten sollten angeboten werden. Sich über diese zu informieren und diese selbst zu meistern brachte viel Spaß und vor allem Lernerfolg. Dabei traten jedoch einige Probleme auf:

- Der durch Fehler verursachte Zeitaufwand lief aus dem Ruder
- Es wurde sich am Projektanfang zu viel vorgenommen
- Ein Top-Down-Ansatz war für dieses Teilprojekt nicht angemessen → Eine kurze Backuplösung, Bottom-Up erstellt, hat am Ende den Zweck des kompletten Teilprojektes seiner statt erfüllt. Viel Zeit und Mühe hätte sich also gespart werden können.
- Die tatsächlichen Anforderungen waren erst gegen Ende der Entwicklungszeit bekannt. Wer will/muss tatsächlich senden? Mit welchem Umfang? → Frühere Versionen des Teilprojektes sind stark überdimensioniert. Es wurde dabei Versucht Komponenten zu entwickeln, welche viel Funktionalität beherrschen, um später viel abdecken zu können. Das ist zwar "Nummer sicher" und erzeugt Arbeitszeit, jedoch wie sich am Ende zeigt kaum Mehrwert für das Projekt.



- Die “zentrale Wichtigkeit” (angesprochen unter [V1.1]) des Protokolls wurde falsch eingeschätzt, wie die ausreichende Backup-Implementierung zeigt.

Trotz aller Probleme können aus diesen viele Lehren hinsichtlich Projektvorgängen und eigenen Herangehensweisen gezogen werden. Des Weiteren war die Arbeit theoretischem Lernerfolg gegenüber Protokollen im Embedded-Umfeld bestimmt nicht umsonst. Es durfte viel selbstständig entworfen werden, was Spaß gemacht hat und viel Überlegung und Arbeit floss auch in dieses Teilprojekt, obwohl es vom erwünschten Erfolg nicht gekrönt wurde. Auch Programmier-Fertigkeiten wurden verbessert und der Horizont für verschiedenen Arbeitsweisen und Entwicklungsumgebungen durch Kollegen erweitert.



14. Grafische Benutzeroberfläche

Ersteller: Florian Boemmel

14.1. Generelles

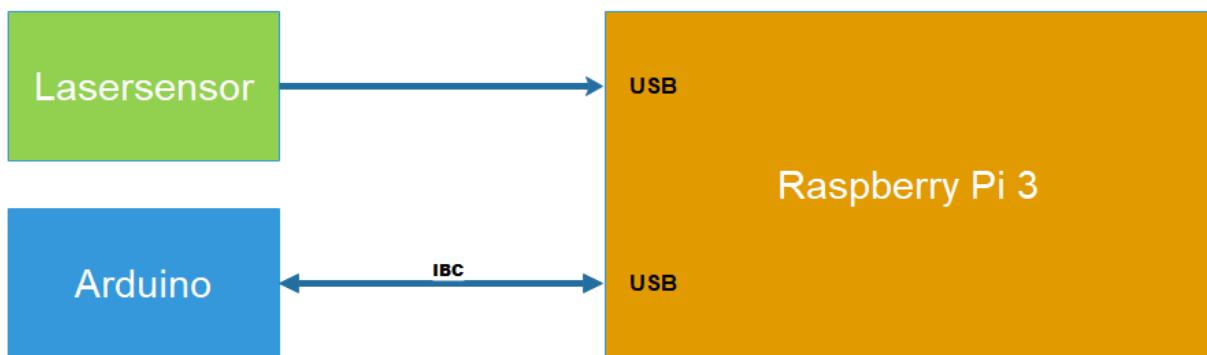
Die grafische Benutzeroberfläche (im Folgenden als "GUI" bezeichnet) stellt, abstrakt dargestellt, das Bindungsglied zwischen Benutzer und dem Fahrzeug dar. Über diese soll die Steuerung des Fahrzeugs erfolgen.

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 als "Pi" bezeichnet) in C++ entwickelt werden.

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

Folgende Module existieren:

- Lasersensor
- IBC



14.2. 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 den Pi, oder einer App schwankte. Jedoch ist gerade die Zielplattform ein ausschlaggebender Faktor, um das passende GUI-Toolkit auszuwählen.



Aufgrund der unbekannten Zielplattform, recherchierte ich über mögliche GUI-Toolkits, die sowohl eine Desktop-Anwendung unter Linux / Windows / OSX oder eine mobile Anwendung unterstützen. 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 können. 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 hinzugefügt werden und erfüllen 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 integrierten 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.

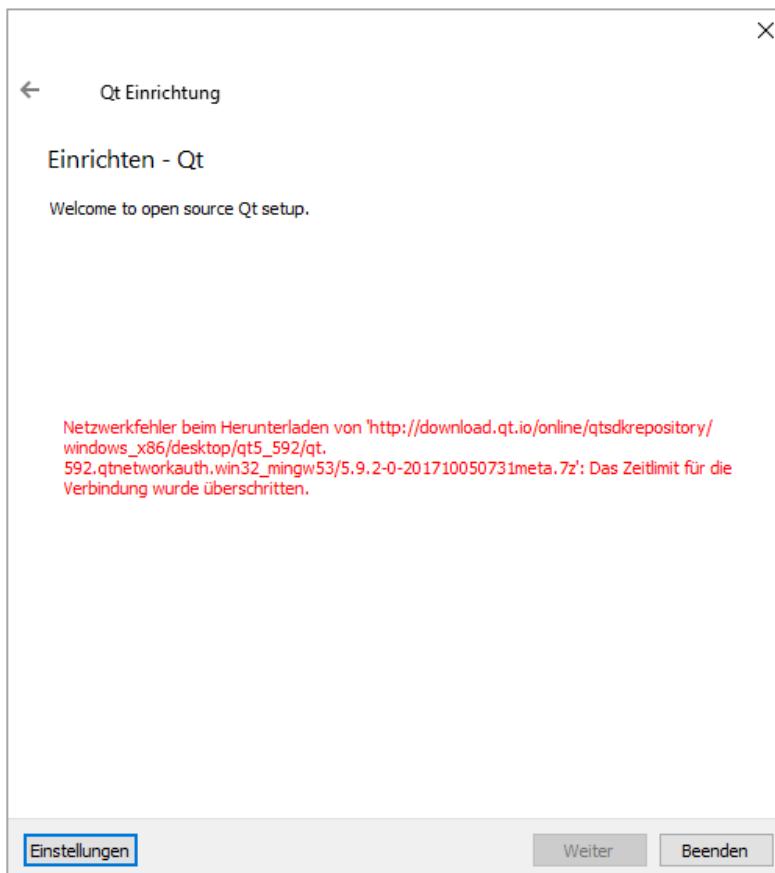
Aus den bereits aufgeführten Gründen wird die GUI in der Sprache C++ mit dem GUI-Toolkit Qt5 realisiert. Dabei möchte ich noch kurz anfügen, dass ich bis dato noch nichts mit Qt gemacht habe und generell noch keine GUI geschrieben habe.

14.3. Installation & Einrichtung von QtCreator

Qt ist in zwei Versionen verfügbar. Zum einen Open Source oder Commercial. Die Unterschiede können unter der [Download-Seite](#) eingesehen werden. Ich verwende die kostenlose Open Source Version.

14.3.1. Installation unter Windows

Ich möchte hier ausführlicher auf die Installation unter Windows eingehen, da es dabei ein entscheidendes Problem gab, dies ist auch unter OSX zu beobachten. Downloadet man die Installationsdatei 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 und Plattform gewählt werden. Danach sollte die Installation reibungslos funktionieren.

Ein letzter wichtiger Punkt ist das Auswählen der zu installierenden Pakete. Unter Windows reichen die von Qt standardmäßig 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.



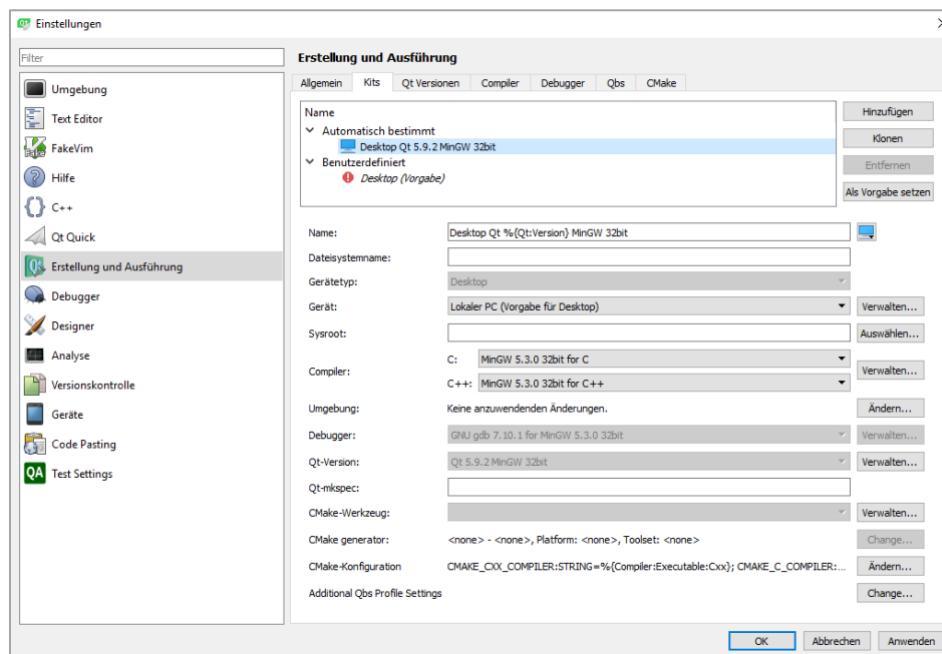
14.3.2. Installation unter Linux (Raspberry Pi)

Unter Linux gestaltet sich die Installation etwas einfacher. Dazu müssen lediglich folgende 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 libqt5serialports5
sudo apt-get install libqt5serialport5-dev
```

14.3.3. Einrichten

Einige erstmalige Einstellungen sind für einen korrekten Kompiliervorgang nötig. Dazu muss ein Kit (Bezeichnung von Qt) eingerichtet werden, falls dies nicht automatisch geschehen ist. 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, dann ist es erfolgreich eingestellt worden und eine Kompilierung ist nun möglich.

Die Einrichtung unter Linux ist äquivalent dazu, lediglich die Compiler sind verschieden.



14.3.4. 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 ist. Weiterhin konnte mir hier die Dokumentation von Qt 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, pushte dieses auf GIT und pullte dieses anschließend auf den Pi. Unerwartet konnte das Projekt ohne Probleme auf dem Pi geöffnet und übersetzt werden.

Ich entschied mich von nun an dafür, 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 Graf, war das Problem der plattformspezifischen Funktionalitäten stets gegenwärtig, da in diesem Linux Systemaufrufe getätigt wurden. Daraufhin setzte ich ein virtuelles Ubuntu auf, um ohne weitere Compiler Schalter kompilieren zu können und somit die Entwicklung etwas angenehmer zu gestalten.

14.4. 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 und das Inter Board Protocol (IBC) gestartet wird. Weiterführende Steuerungsmöglichkeiten dürfen dem Benutzer zu diesem Zeitpunkt nicht zur Verfügung stehen.



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

- 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 dem Benutzer, über einen Button, die Möglichkeit gegeben werden, die Raumkartographie zu starten (/G0111/).

/G0107/ **Controllersteuerung:** Wählt der Benutzer den Modus "Controllersteuerung", muss diesem zunächst eine kurze Anleitung dargestellt werden, dass er den Controller bereitzuhalten 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 dem Benutzer, über einen Button, die Möglichkeit gegeben werden, die Raumkartographie zu starten (/G0111/).

/G0108/ **Navigation:** Der Benutzer muss jederzeit die Möglichkeit erhalten, zur Moduswahl (/G0103/) zurückzukehren um 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 aktueller 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 einen 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).

14.5. Umsetzung

14.5.1. Implementierung der GUI

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

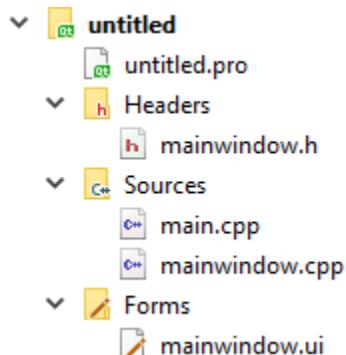
14.5.1.1. Projekterstellung

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 Klassename 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 Versionskontrolle 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:

```
1 #include "mainwindow.h"
2 #include <QApplication>
3
4 int main(int argc, char *argv[])
5 {
6     QApplication a(argc, argv);
7     MainWindow w;
8     w.show();
9
10    return a.exec();
11 }
```

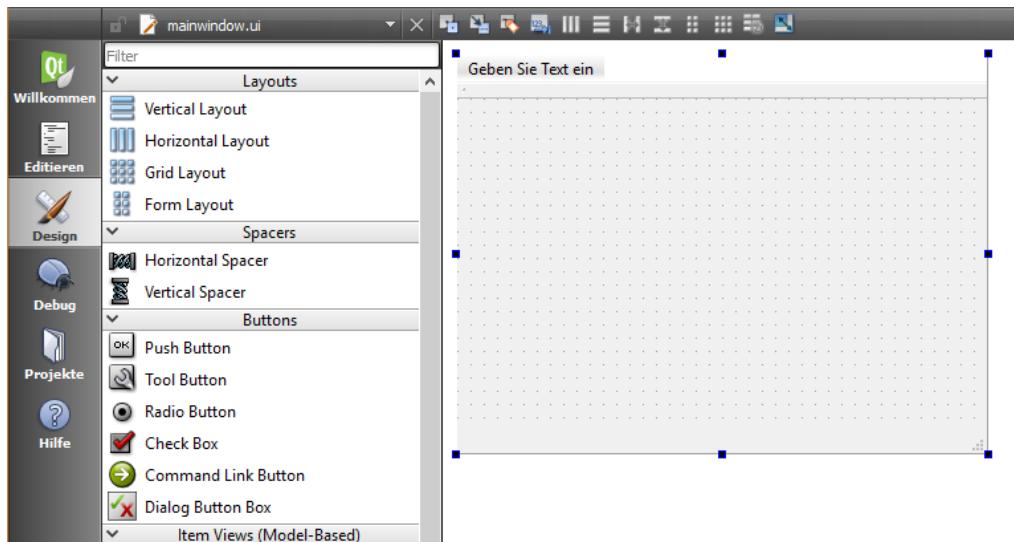
Der Ablauf einer Qt-Widgets-Anwendung ist denkbar simpel. Zunächst wird eine Instanz von QApplication erzeugt.

Anschließend wird eine Instanz des Hauptfensters erstellt und mit der Methode “show()” angezeigt. Wenn man “show()” nicht aufrufen würde, wird die Anwendung dennoch ausgeführt, jedoch 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 und Kontroll- sowie Layoutelemente hinzugefügt werden. Formulardaten sind XML basiert und können im Qt-Creator nur über den Designer geändert werden.

14.5.1.2. Designer vs. Code

Zu Beginn eines jeden 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 unterschiedliche Meinungen.

Auch ich überlegte lang, welche Herangehensweise ich für unser Projekt nutzen möchte. Es folgten einige Versuche mit beiden Möglichkeiten. Ich entschied mich schlussendlich für eine reine Implementierung des Designs und somit gegen die Verwendung des Designers.

14.5.1.3. 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 Eingaben werden nicht immer gleich erkannt und sind teilweise ungenau
- Die Größe des Displays ist für die definierten 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, dass die dargestellten Elemente ein sogenanntes Masterlayout bilden, dieses ändert sich nicht und nur die wichtigsten Elemente sind jederzeit erreichbar. Wie der Zugang zur Fehleranzeige oder das Menü.

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

14.5.1.4. 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 eine Box dar, in die Elemente vertikal oder horizontal angeordnet werden können.

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, welche sowohl einen Button für die Fehleranzeige als auch für das Menü enthält.

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 gestyliert:

```
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 wird der gesamte Platz ausgenutzt. Standardmäßig werden 8 Pixel Randbreite automatisch festgesetzt.

Ein ganz wichtiger Punkt ist, 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 dieses später im Vollbildmodus laufen soll und der Rand nicht benötigt wird.

Arbeitet man mit oder ohne dem 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(removeWindowfromStack()), 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 removeWindowfromStack();
```

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

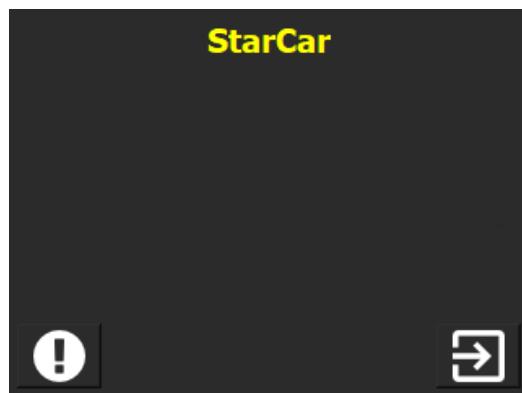
Schließlich wird das zuvor erzeugte ExitWidget in das QStackWidget eingefügt und angezeigt.

Die Vorgehensweise, um die Verbindung des Buttons für die Fehleranzeige zu erstellen, ist äquivalent dazu.



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

Das Masterlayout nach der Implementierung:

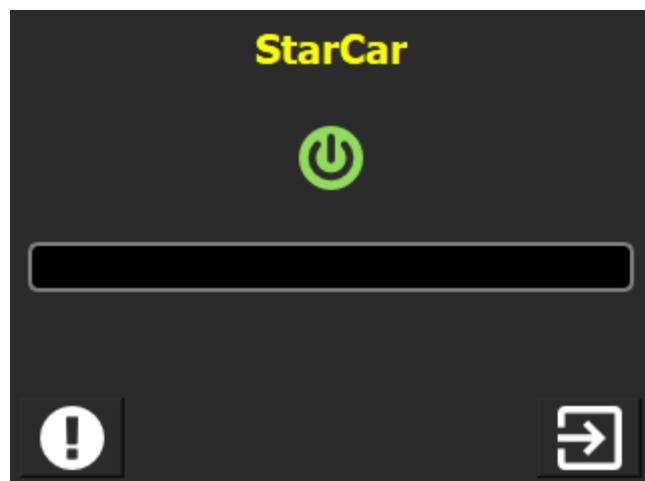


Zu sehen sind der Titel und danach ein freier Platz für die austauschbaren Widgets. Anschließend folgt in der linken unteren Ecke der Button für die Fehleranzeige. Gegenüber befindet sich der Button für das Menü. Auf die dazugehörigen Widgets wird später detaillierter eingegangen.

14.5.1.5. Startfenster

Wird die GUI gestartet wird zunächst wie eben beschrieben 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, sowie eine Fortschrittsanzeige.



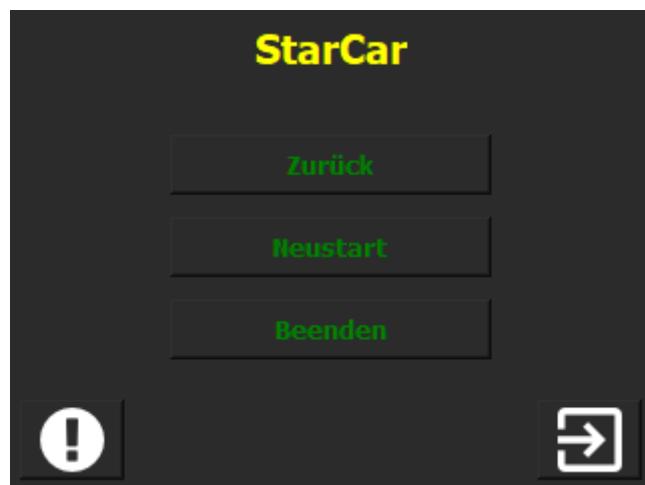
Drückt der Benutzer auf den grünen Button wird ein neuer Thread gestartet. Dieser übernimmt, im späteren Verlauf des Projekts, nach der Integrierung des IBCs in die GUI dessen Initialisierung und öffnet den seriellen Port zum Arduino. Dieser Prozess muss in einem Thread ausgelagert werden, da sonst während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

14.5.1.6. Menü

Drückt der Benutzer auf den Button "Menü", wird das aktuelle Widget nicht aus dem QStackedWidget entfernt, sondern lediglich das ExitWidget auf den 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, das 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:

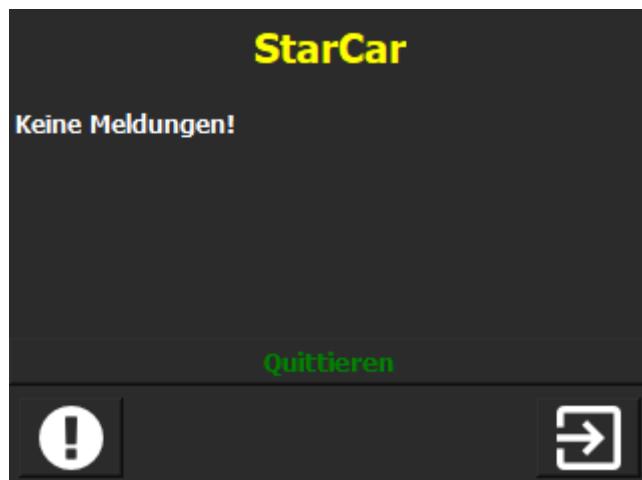
```
qApp->quit();  
QProcess::startDetached(qApp->arguments() [0], qApp->arguments());
```

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 Kommandos realisiert werden:

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

14.5.1.7. Fehleranzeige

Drückt der Benutzer auf den Button “Fehleranzeige”, wird das aktuelle Widget nicht aus dem QStackedWidget entfernt, sondern lediglich das AlertWidget auf den 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 das ein Fehler oder eine Warnung aufgetreten ist, folgendermaßen aus:



Dabei ist der Aufbau recht schlicht gehalten. Dieser enthält eine QListWidget, 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 werden aber nicht aus der QListview gelöscht.

Die QListView ist dabei auf das minimalste reduziert worden. Dies bedeutet, dass weder Ränder noch ein Scroll-Balken zu sehen ist. Erreichen die Meldungen aber eine gewisse Anzahl, so erscheint der Scroll-Balken 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 im Code funktioniert folgendermaßen:

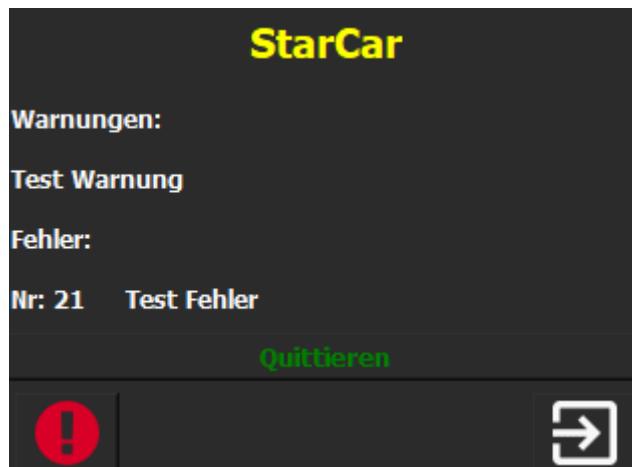
```
this->alertThread->fireWarning();
this->alertThread->fireWarning("Test Warnung");
this->alertThread->fireError("Test Fehler", 21);
```

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, dass der Lasersensor vergessen wurde anzustecken. Sobald die GUI den Lasersensor anspricht, würde der Button rot blinken und nach dem Betätigen des Buttons „Fehleranzeige“ würde die Fehlermeldung „Lidar not working“ ausgegeben werden. Gerade während der Entwicklung erwies sich die Fehleranzeige als sehr nützlich.

14.5.1.8. 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 der Raumkartographie um.



14.5.1.9. 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. Dem Benutzer soll mitgeteilt werden, dass er die Uhren anlegen 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 Moduswechsel stattfinden muss. Unabhängig von der Mitteilung an den Arduino wird das Widget diesmal nicht entfernt, sondern 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 präzisere Darstellung der Anleitung. Jedoch sind die Pfeile während



der Ausführung animiert. Der Benutzer kann entweder zur Moduswahl zurückkehren oder die Raumkartographie starten. Eine Steuerung des Autos mit den Uhren ist nun möglich.

14.5.1.10. 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. Dem Benutzer soll mitgeteilt werden, dass er den Controller bereithalten 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 Moduswechsel stattfinden muss. Unabhängig von der Mitteilung an den Arduino wird das Widget diesmal nicht entfernt, sondern folgendermaßen umgebaut:

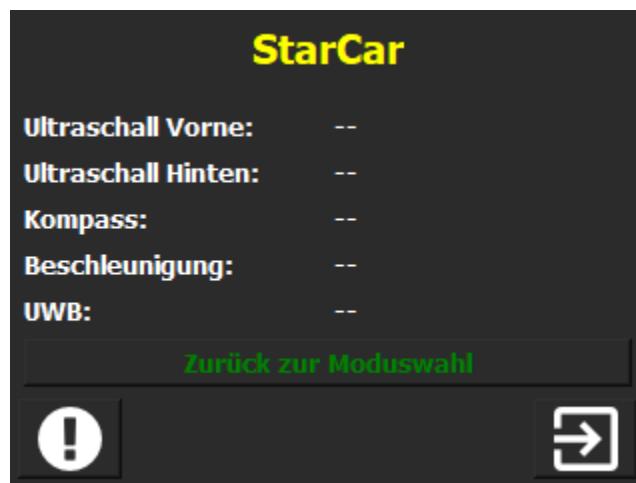




Zunächst wird der Benutzer über einen in der Farbe Grün gewählten Text darüber informiert, dass das Auto nun mit dem Controller steuerbar ist. Daraufhin wird dem Benutzer eine minimale Anleitung dargestellt, wie er mit den Uhren das Auto steuern kann. Leider fehlte der Platz für eine präzisere Darstellung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

14.5.1.11. Sensorwerte / Raumkartographie

Hat der Benutzer entweder in der [Operationsauswahl](#) den Button „Sensorwerte“ oder in der [Uhrsteuerung / Controllersteuerung](#) den Button „Starte Raumsan“ 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



14.5.2. Integration Lasersensor und IBC Protokoll

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

14.5.2.1. 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, entwickelte 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.

14.5.2.2. Test Lasersensor

Für den 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 Tests lieferten gleiche Ergebnisse. Der Lasersensor galt somit als erfolgreich integriert und getestet. Auch im späteren Verlauf zeigte dieser keine Auffälligkeiten und arbeitete zuverlässig.



14.5.2.3. Integration IBC Protokoll

Das Protokoll wurde in C++ 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 ließ sich aufgrund einer sehr guten Kapselung sehr einfach in die GUI integrieren und bereitete keine Integrationsprobleme.

14.5.2.4. Test IBC Protokoll

Das Protokoll bestand aus dem Protokoll selbst und einer minimalen Konsolenanwendung zur Veranschaulichung der Funktionsweise, sowie der Benutzung auf der Seite 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 Beispielanwendung 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 der Seiten des Arduinos wurde das Protokoll ebenfalls integriert und die beiden Geräte mit einem 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 in der 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 erst recht spät 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.

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 und anschließend in die Textdateien geschrieben.

Zunächst kamen allerdings 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.

14.5.3. Entwicklung Backup-Protokoll

Nach der Abschlusspräsentation am 12.01.2018 fasste Dominik Scharnagl, Simone Huber und ich den Entschluss eine mögliche Backupslösung 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 weitere 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 dort keine Schattenfelder vorhanden waren.

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 zum einen dargestellt und zum anderen in die Textdateien für die Raumkartographie geschrieben werden.

In weiteren Tests zeigte sich allerdings, dass die GUI einfrierte, 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 dies bis zur Abschlussvorführung nicht mehr umgesetzt werden.

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.

14.6. 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 die Raumkartographie 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 neu starten. 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 das 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 und/oder 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.



14.7. Ausblick

Die Entwicklung der GUI sowie deren Implementierung konnte ich für mich zufriedenstellend 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. Zum Beispiel müsste dieses in einen eigenen Thread ausgelagert werden. Gerade bei einem Verbindungsabbruch 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 dieses Projektes meine eigenen Ideen und Vorstellungen frei umsetzen.



15. Raumdarstellung

Ersteller: Anja Strobel

Die gesammelten Daten werden während des Fahrens an einem externen PC ausgewertet. Dies geschieht im zeitlichen Abstand von einer Sekunde. Die Auswertung erfolgt mit Hilfe von MATLAB.

15.1. Voraussetzungen

Die gesammelten Daten werden über WLAN an einen externen PC übertragen. Es ist zu beachten, dass sich das Fahrzeug und dieser PC im selben Netzwerk befinden müssen. Im Fall dieses Projektes war dieser PC mit einem Windows Betriebssystem ausgestattet. Um die Daten zu übertragen und auszuwerten werden die folgenden Programme auf dem verwendeten Rechner benötigt.

- PuTTY mit PSFTP
- Cygwin Terminal (Installation mit den zusätzlichen Paketen "gcc-g++", "dos2unix")
- MATLAB

15.2. Datenübertragung

Sämtliche Daten werden unter Verwendung von PSFTP (PuTTY SFTP) übertragen. Die zu transferierenden Daten sind mehrere Textdateien, die Messwerte enthalten. Sie werden im folgenden aufgezählt:

- BeschleunigungX.txt (Fahrzeugbeschleunigung in Richtung X)
- BeschleunigungY.txt (Fahrzeugbeschleunigung in Richtung Z)
- Compass.txt (Ausrichtung des Fahrzeugs)
- Lidar.txt (Daten des LIDAR Sensors)
- UltraHinten.txt (Entfernung des hinteren Ultraschallsensors)
- UltraVorne.txt (Entfernung des vorderen Ultraschallsensors)

Um die Datenübertragung zu starten wird das Programm "starCarTransfer.cpp" auf dem externen PC über Cygwin Terminal PC ausgeführt. Cygwin Terminal simuliert eine Linux Umgebung auf dem Windows PC. Falls der Sourcecode unter Windows verändert wurde sollte vor dessen Kompilierung in Cygwin Terminal der Befehl "dos2unix starCarTransfer.cpp" ausgeführt werden. Dieser führt dazu dass Zeilenumbrüche von "\r\n" in "\n" geändert wird. Anschließend kann das Programm kompiliert werden.

Innerhalb einer Schleife wird zuerst das Konsolenprogramm PSFTP aufgerufen, dem die NetzwerkkAdresse des Raspberry Pi, den Hostname, das Passwort und ein Skript mit Befehlen für PSFTP übergeben werden.



Nachdem die Daten übertragen wurden wird für jede Datei umbenannt. Dies geschieht damit es keine Probleme bei einem möglichen zeitgleichen Zugriff auf die Dateien durch PSFTP und MATLAB gibt, da MATLAB immer die umbenannte Datei verwendet. Die Pfade müssen bei Verwendung natürlich angepasst werden.

```
system("C:/Program Files/PuTTY/psftp.exe' pi@172.16.24.254 -pw starcar -b 'C:/Program Files/PuTTY/test_commands.sh'");
std::string str = "BeschleunigungX.txt";
str.insert(15, "Cp");
rename("D:\\DT\\FinalRun\\BeschleunigungX.txt", ("D:\\DT\\FinalRun\\" + str).c_str());
```

Die folgenden Befehle werden von PSFTP ausgeführt. Die Messdaten werden auf dem Raspberry Pi im Ordner "/home/pi/SensorOutput" abgespeichert. Der lokale Pfad muss bei Bedarf geändert werden (lcd). "mget" holt die angegebenen Dateien vom Raspberry Pi und speichert sie mit gleichem Namen auf dem lokalen Pfad ab. Mit "quit" wird PSFTP beendet.

```
cd /home/pi/SensorOutput
lcd D:\\DT\\FinalRun

mget "BeschleunigungX.txt" "BeschleunigungY.txt" "Compass.txt" "Lidar.txt" "UltraHinten.txt" "UltraVorne.txt"
quit
```

15.3. Verarbeitung der Daten in Matlab

Sämtliche Daten werden in Matlab verarbeitet und in einer Figure angezeigt. Zuerst wird eine Figure mit Größe von -3000 bis 3000 erzeugt. Dies entspricht einer maximalen Entfernung zum Mittelpunkt von 3m. Der Nullpunkt entspricht dem Standpunkt des Fahrzeugs. Das Programm läuft in einer Endlosschleife.

```
close all
str = sprintf('StarCar');
figure('Name', str, 'NumberTitle', 'off', 'position', [0, 0, 800, 800])
axis([-3000 3000 -3000 3000])
```

Anschließend werden die Dateien mit den Messdaten ausgelesen. Die von MATLAB gelesenen Zeilen werden in einer Zelle abgespeichert. Da die Daten von Beschleunigungssensoren, Ultraschallsensoren und Kompasssensor immer an eine bereits existierende Datei angehängt werden, wird in diesem Fall die letzte Zeile der Datei in eine Variable abgespeichert.

Die Daten des LIDAR Sensors liegen mit einer Datei pro Messung vor. Hier werden alle Zeilen (Fehlerindikator, X-Koordinaten, Y-Koordinaten) in einer Zelle abgespeichert.



```
fid = fopen('BeschleunigungXCp.txt');
C = textscan(fid, '%s', 'Delimiter', '\n', 'CommentStyle', '%');
fclose(fid);
cell = C{1};
[lines,a] = size(cell);
accelerateX = str2num(cell{lines,1});
```

15.3.1. Kompass

Die Daten des Kompass werden von einem Winkel im Gradmaß in einen Vektor mit Startpunkt (0,0) und Länge 200, der in die gemessene Richtung zeigt, umgewandelt. Dieser wird als Pfeil dargestellt.

```
direction_x = 0 * cosd(-compass) - 200 * sind(-compass);
direction_y = 0 * sind(-compass) + 200 * cosd(-compass);

drawArrow = quiver( 0,0,direction_x,direction_y, 0, 'LineWidth',1,'MaxHeadSize',2);
```

15.3.2. Ultraschall

Die beiden Ultraschallsensoren liefern Entferungen zum ersten Gegenstand direkt vor/hinter dem Fahrzeug. Zusammen mit den Daten des Kompass werden diese Entfernungen in Koordinaten umgewandelt, die mit der Ausrichtung des Fahrzeugs übereinstimmen. Diese Punkte werden mit einem X gekennzeichnet.

```
usf_x = 0 * cosd(-compass) - (usFront*10) * sind(-compass);
usf_y = 0 * sind(-compass) + (usFront*10) * cosd(-compass);

usb_x = 0 * cosd(-compass) - (usBack*10) * sind(-compass);
usb_y = 0 * sind(-compass) + (usBack*10) * cosd(-compass);

scatter(usf_x, usf_y, 100, 'x', 'LineWidth',1.5)
scatter(-usb_x, -usb_y, 100, 'x', 'LineWidth',1.5)
```

15.3.3. LIDAR

Die LIDAR Daten wurden in einer Zelle gespeichert aus der nun Arrays mit x- und y-Koordinaten erzeugt werden. Diese Koordinaten werden nun unter Verwendung des Winkels aus den Kompassdaten gedreht, damit deren Ausrichtung mit der Ausrichtung der restlichen Daten entspricht. Da, wie bereits im zugehörigen Kapitel beschrieben, Messfehler und Rauschen vorhanden sind wird mittels Bildung des gleitenden Mittelwerts versucht das Ergebniss zu glätten. Zuerst wird ein Mittelwert mit dem direkten



Nachbarwert gebildet, danach wird der Mittelwert über eine Gruppe von 5 Werten gebildet. Dieses Verfahren hilft etwas, die Werte zu glätten führt aber zu einer geringeren Anzahl von Messpunkten.

Die Daten des LIDAR Sensors werden, sowohl in roher Form als auch in geglätteter Form, als Linie in der Figure dargestellt.

```
lidarData = horzcat(lidarX', lidarY');
compass_rad=((-compass*pi)./180);
[THETA,R] = cart2pol(lidarX,lidarY); %Convert to polar coordinates
THETA=THETA+compass_rad; %Add a _rad to theta

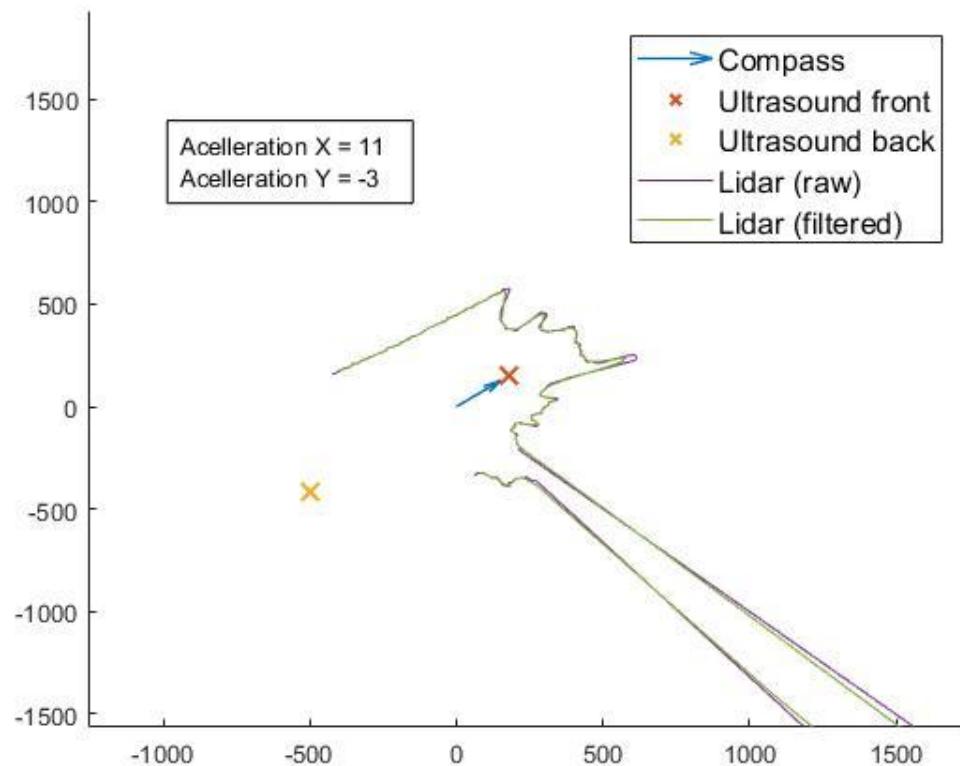
[lidarXrotated,lidarYrotated] = pol2cart(THETA,R);
plot(lidarXrotated,lidarYrotated)
```

15.3.4. Beschleunigungssensoren

Die Daten der Beschleunigungssensoren werden als Annotation in schriftlicher Form angezeigt.

15.3.5. Ergebnis

Die folgende Grafik zeigt die verarbeiteten Daten einer Messung.





15.4. Probleme und Ausblick

Zu Beginn des Projektes war angedacht auch die Positionsdaten des UWB Senors mit zu verwerten. Diese Daten sollten mit den Daten der anderen Sensoren verechnet werden. Mit Hilfe der Fahrzeugposition wäre es möglich Hindernissen absoluten Positionen zuzuweisen und so eine Karte der Umgebung zu erstellen. Es war geplant hierzu ein Modell zu erstellen, dass erlaubt die Daten mehrerer Messungen zu fusionieren und zu einem vollständigen Abbild der Umgebung zu verrechnen. Ohne Positionsdaten war dies leider nicht möglich.

Bei einer Weiterführung des Projektes könnte hierauf eingegangen werden.



16. Zusammenbau

Ersteller: Mehmet Billor

16.1. Montage Fahrzeug - Grundgerüst

Das Auto befand sich zu Anfang des Projektes bei der Übernahme in einem relativen "Rohzustand". Es war lediglich der Antriebs- und Servomotor verbaut. Um genügend viel Platz für die Montage aller Teile zu haben viel die Entscheidung auf eine Spanholzplatte, da diese nicht elektrisch leitend ist und ein geringes Gewicht hat. Die Platten wurden eigenständig gekauft und auf das entsprechende Maß geschnitten. In die Hauptplatte wurden 4 Löcher gebohrt, um diese mit dem Fahrzeug über 4 Gewindestangen zu verbinden und jeweils 4 Sechskantmuttern befestigt um die nötige Stabilität zu gewährleisten.

Für die Befestigung der Ultraschallsensoren am vorderen und hinteren Ende des Fahrzeugs wurden zusätzlich 2 kleinere Spanholzplatten angefertigt und mit einem 90 Grad Winkel montiert. Die zwei Spanholzplatten dienen gleichzeitig auch als „Stützbeine“ für die gesamte Hauptplatte, welches arbeiten am Fahrzeug erleichtert hat. Ebenso wurde eine kleinere Platte mit einem 90 Grad Winkel in der Mitte der Hauptplatte montiert, welche als Halterung für den UWB-Sensor dient. Die LEDs vorne und hinten am Fahrzeug waren bereits vormontiert. Es musste lediglich nur noch die Verkabelung erneuert werden, da nicht alle LEDs funktionierten.

16.2. Montage der einzelnen Sensoren / Steuerungen / Boards

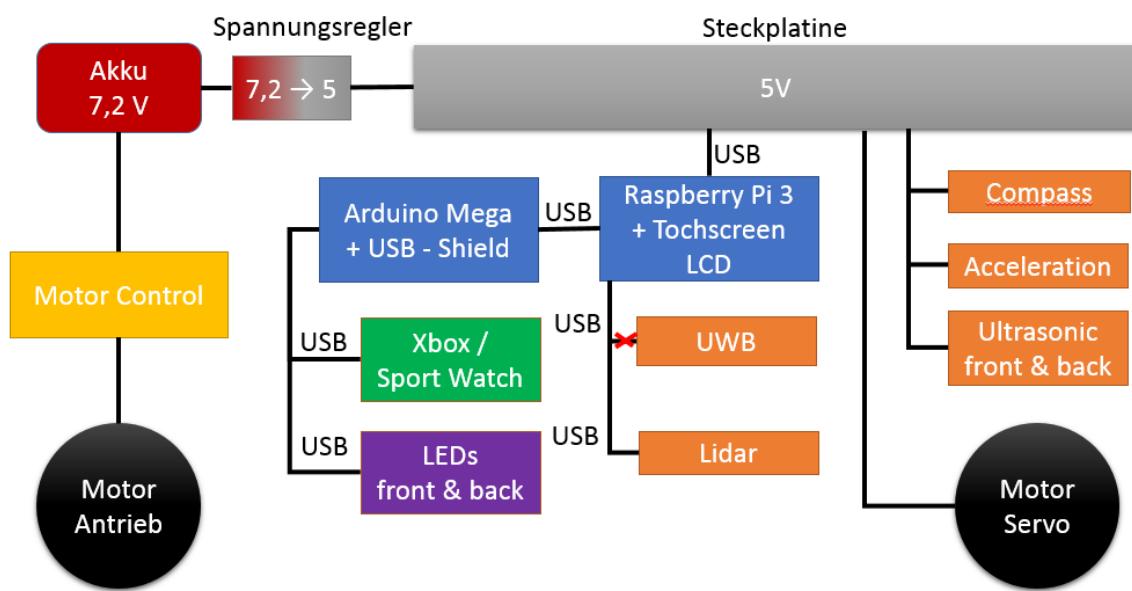
Die zwei Ultraschallsensoren wurden jeweils am vorderen und hinteren Ende des Fahrzeugs auf den 2 kleineren Spanholzplatten montiert. Der Beschleunigungssensor sowie der Kompasssensor wurden auf die Steckplatine aufgesteckt.

Der Lidar wurde vorne am Fahrzeug befestigt, da es freie Sicht auf die Fahrbahn für fehlerfreie Auswertung benötigte. Um dies zu gewährleisten, wurde der Lidar durch Abstandsringe höher gesetzt. Der UWB-Sensor wurde mittig auf der mittleren Spanholzplatte befestigt. Für den Arduino Mega sowie den Raspberry Pi haben wir jeweils Gehäuse organisiert und auf der Hauptplatte befestigt. Die Motorsteuerung wurde am hinteren Ende des Fahrzeugs, welcher auch mit Abstandsringen an der Hauptplatte befestigt ist, montiert. Da der Arduino Mega keinen USB-Anschluss besitzt wurde ein USB-Shield aufgesteckt. Das Display für den Raspberry Pi wurde ebenso aufgesteckt. Der Antriebsmotor sowie der Servomotor waren bereits im Fahrzeug eingebaut.



16.3. Spannungsversorgung

Das Fahrzeug wird von einem 6-Zellen 7.2 V mAh Akku mit Spannung versorgt. Die Motorsteuerung sowie der Motor sind an 7.2 V angeschlossen. Um die einzelnen Teile mit Spannung zu versorgen bzw. die Spannung zu verteilen haben wir eine Steckplatine verwendet. Da die restliche Hardware nur eine Betriebsspannung von 5 V benötigten, wurde ein Spannungsregler auf die Steckplatine gesteckt. Für die Verkabelung wurden 1-Polige Steckverbinder oder USB-Kabel verwendet. Eine Übersicht der Spannungsversorgung kann man in der unteren Abbildung entnehmen. (Anmerkung: Der Anschluss des UWB ist durchgestrichen, da er für den finalen Aufbau zwar montiert, aber nicht angeschlossen wurde)



Für die Spannungsversorgung des Raspberry wurde ein Micro-USB-Kabel modifiziert, welches mit einer Lüsterklemme verbunden wurde und die Lüsterklemme mit der Steckplatine. Die Verbindungen auf der Steckplatine wurden mit starren Drähten verbunden.



16.4. Übersicht verwendeter Teile

- Kleinmaterial
 - Schrauben
 - Sechskantmuttern
 - Kabelbinder
 - Lüsterklemmen
 - doppelseitiges Klebeband
 - Abstandshalter aus Plastik
 - Adernendhülsen
 - Isolierband
- Befestigungsplatten
 - Grundbefestigungsplatte: Spanholzlatte 40x20x1 cm / 240g
 - Sensorbefestigungsplatte: 2 Spanholzplatten 10x10x1 cm / jeweils 30g
- Steckplatine rutschfest Polzahl Gesamt 730
 - Mikrocontroller-Board
 - Arduino Mega 2560 inklusive USB-Host-Shield (zuvor Arduino Uno R3)
 - Raspberry Pi 3
- Sensoren
 - 2 Ultraschall – Sensoren
 - Beschleunigungssensor
 - Kompasssensor
 - Lidar
 - UWB

Zu Beginn und bis Ende des Bestellzeitraums (für etwaige fehlende Komponenten) war nicht bekannt, dass der geplante Arduino Uno R3 (aus dem Laborbestand) nicht genügend Leistung, insbesondere Speicherkapazitäten für das Projekt hat. Erst gegen Ende des Projektes stellten sich Probleme bei einer 80%igen Ausnutzung der Ressource des Arduino Uno R3 ein. Dominik Scharnagl hat deshalb aus privaten Mitteln einen Arduino Mega 2560 dem Projekt zur Verfügung gestellt.

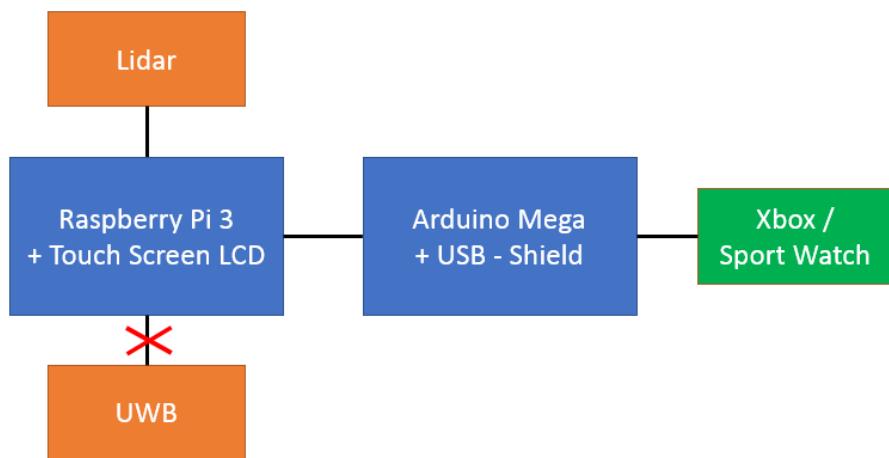


17. Verkabelung

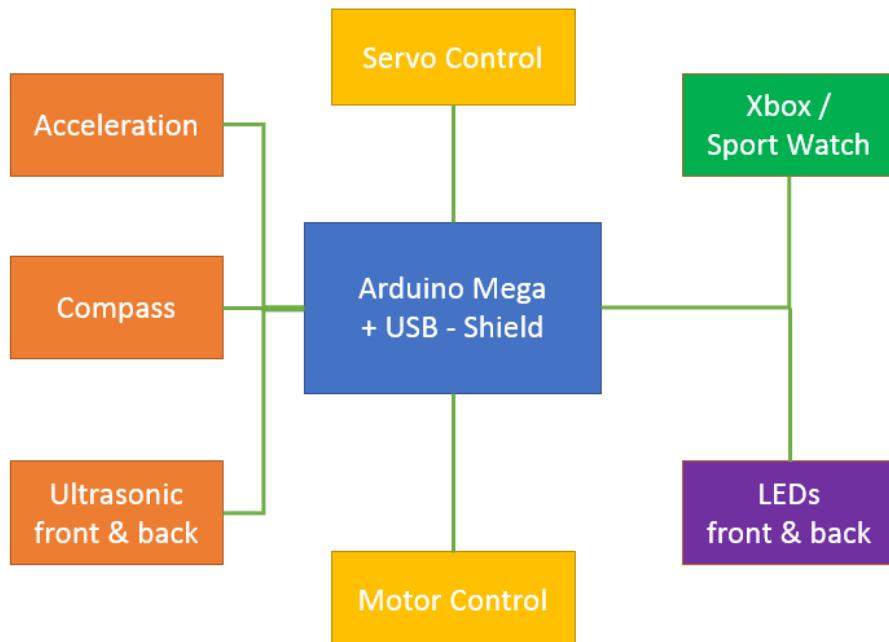
Ersteller: Mehmet Billor

17.1. Verkabelung Daten

Für die Verkabelung der Daten wurde ebenfalls 1-polige Stecker oder USB-Kabel benutzt.
Die Verkabelung der Daten per USB-Kabel ist folgender Abbildung zu entnehmen:



Die Verkabelung der Daten ist folgender Abbildung zu entnehmen.





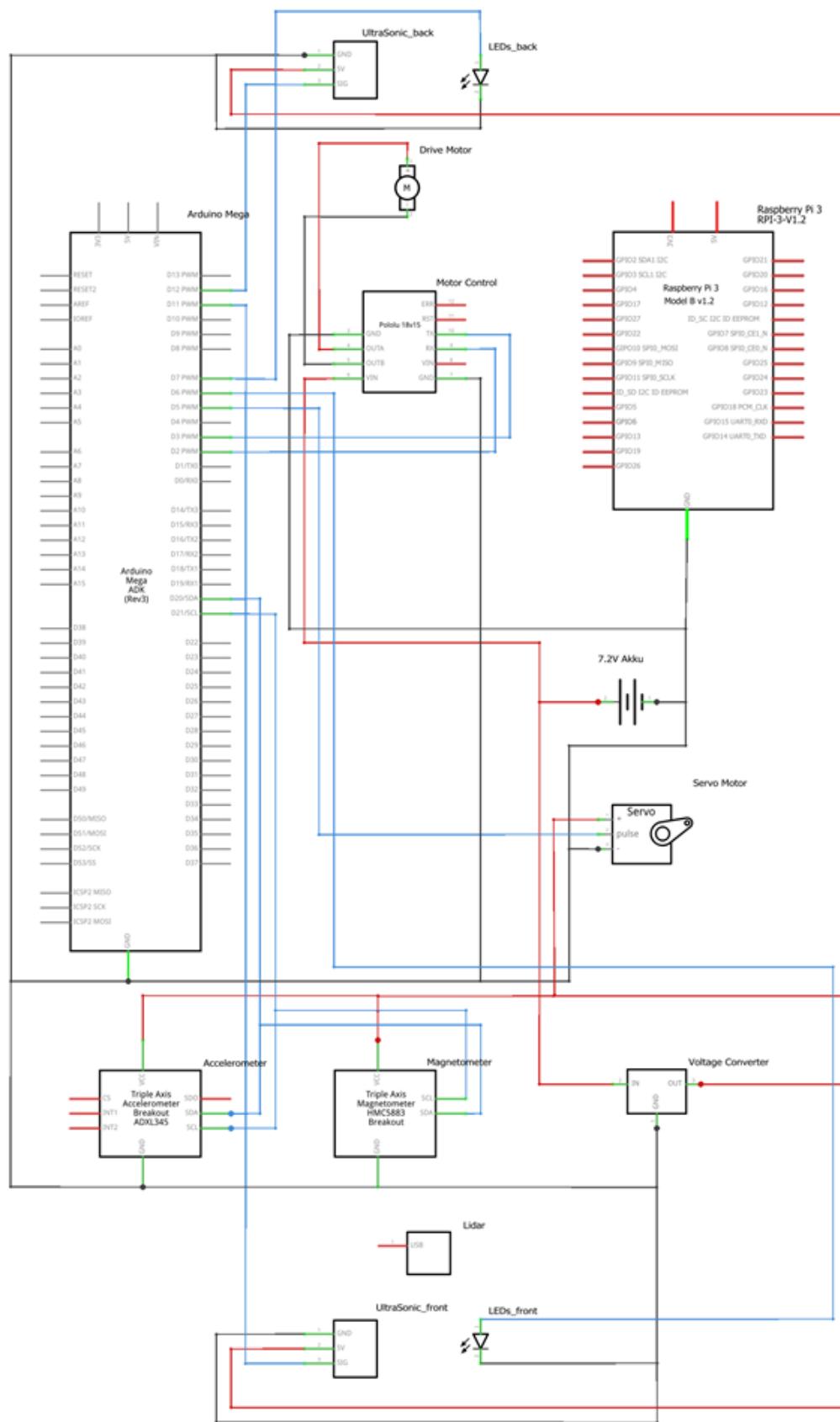
17.2. Verkabelung Gesamtübersicht

Im folgenden Schaltplan sind alle Verbindungen des Fahrzeugs eingezeichnet (bis auf die USB-Kabel Verbindungen).

Es wurden 3 verschiedene Farben verwendet um den Plan darzustellen:

- Schwarz – GND / Ground
- Rot – V / Voltage
- Blau - Data

Für die Endmontage wurden nochmal sämtliche Verbindungen überprüft und durchgemessen. Sämtliche Kabel oberhalb des Autos wurden mit Kabelbindern befestigt und fixiert (unterhalb des Fahrzeugs soweit wie möglich). Zusätzlich wurden nochmal alle Schrauben überprüft und fest angezogen.



fritzing



18. Ausblick

Das Projekt kann in vielen weiteren Aspekten noch verbessert und erweitert werden. Auf diese ist bereits in den einzelnen Kapiteln teilweise eingegangen worden.

Im Allgemeinen ist zu sagen, dass das Projekt um den UWB-Sensor zu erweitern ist, wie die Raumerkennung weiter mit diesem Sensor verbessert werden kann. Es sind auch kleine Verbesserungen möglich, wie Optimierungen an der Benutzeroberfläche, an der Stopp-Bedingung mit dem Ultraschallsensor. Aber auch Erweiterungen mit einem Farbsensor oder dem anfangs geplanten autonomen Fahren, wären denkbar.

Abschließend ist somit zusagen, dass das Projekt eine sehr gute Leistung ist und sehr gut mit diesem weitergearbeitet werden könnte und dann ein hochkomplexes Auto entstehen würde. Da die Grundlagen bereits einige Features aufweisen, welche sehr ansprechend sind.

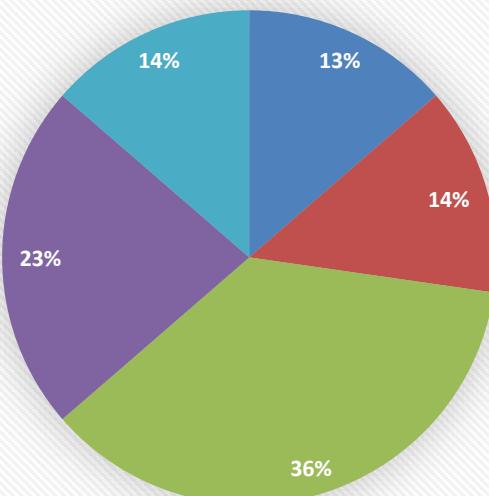


19. Stundenzettel

19.1. Annkathrin Bauer

	Stunden	Kurzbeschreibung
KW 40	2,25	Rahmenfindung des Projekts
KW 41	5,25	Festlegung des Projektes
KW 42	4,25	Projektbeschreibung, Terminplanung festlegen
KW 43	4,00	Recherche Kompass, Teamtreffen
KW 44	8,90	Kompass Implementierung, Kompass Debugging
KW 45	4,57	Beschleunigungssensor Implementierung
KW 46	5,77	Recherche, Debugging
KW 47	8,02	Recherche, Debugging
KW 48	12,17	Testen, Debugging
KW 49	15,33	Frame Erstellung, Einbindung in das Protokoll
KW 50	7,08	Sensorübertragung einbinden, Vorstellung Projekt
KW 51	0,00	
KW 52	0,00	
KW 1	4,67	Kompass Überarbeitung
KW 2	9,62	Präsentationsfolien bearbeiten
KW 3	15,18	Testen, Dokumentation bearbeiten
KW 4	3,25	Dokumentation bearbeiten
KW 5	0,00	
Gesamt	110,30	

Zeiteinteilung Annkathrin Bauer

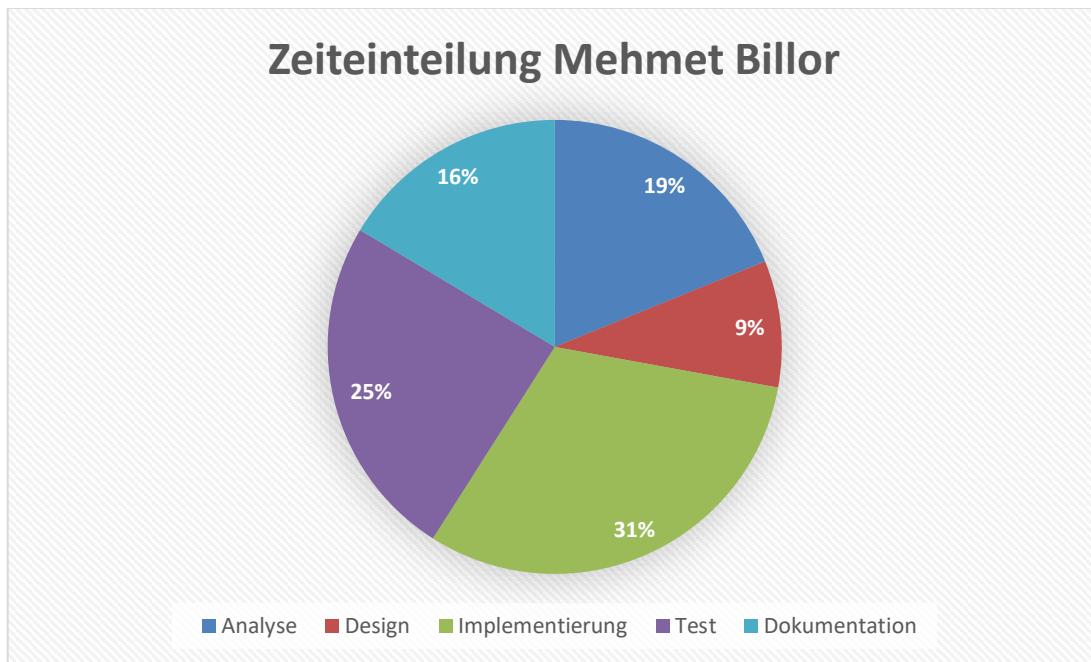


■ Analyse ■ Design ■ Implementierung ■ Test ■ Dokumentation



19.2. Mehmet Billor

	Stunden	Kurzbeschreibung
KW 40	4,25	Rahmenfindung des Projekts
KW 41	12,25	Festlegung des Projektes, Aufsetzen Raspberry Pi 3, Aufsetzen Display
KW 42	6,25	Projektbeschreibung, Terminplanung festlegen
KW 43	0,00	
KW 44	4,25	Teamtreffen
KW 45	9,25	Motorsteuerung Implementierung
KW 46	14,25	Servosteuerung Implementierung
KW 47	13,25	USB Shield Integration
KW 48	6,25	Sensoren Einbindung
KW 49	11,25	Umsetzung XBOX-Controller-Steuerung
KW 50	20,25	Testen, Projekt Zusammenführung, Projektvorführung
KW 51	18,25	Testen, Erstellung der Komponentendarstellung
KW 52	2,00	Komponentendarstellung zeichnen
KW 1	18,00	Implementierung erste Uhr
KW 2	14,25	Implementierung zweite Uhr, Präsentation bearbeiten
KW 3	23,25	Finale Tests durchführen, Vorstellung Projekt, Dokumentation bearbeiten
KW 4	0,00	
KW 5	0,00	
Gesamt	177,25	

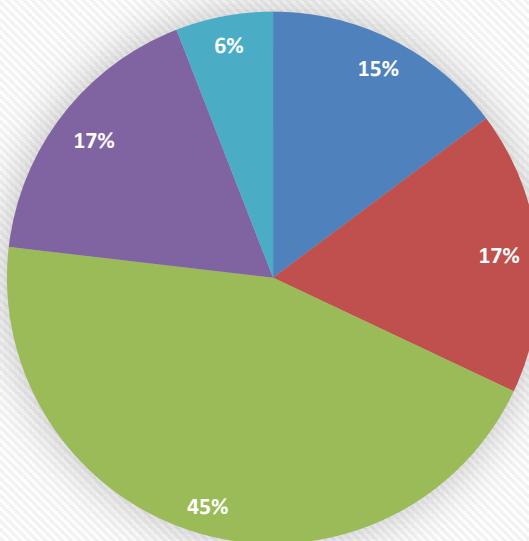




19.3. Florian Boemmel

	Stunden	Kurzbeschreibung
KW 40	1,50	Rahmenfindung des Projekts
KW 41	3,00	Festlegung des Projekts
KW 42	3,25	Projektbeschreibung, Terminplanung festlegen
KW 43	3,25	Recherche Serielle Kommunikation
KW 44	6,02	Recherche Serielle Kommunikation, Teamtreffen
KW 45	21,13	Entwicklung Serielle Kommunikation, Rücksprache Protokoll
KW 46	13,97	Entwicklung Serielle Kommunikation, Teamtreffen
KW 47	12,22	Erstellung GUI, Teamtreffen
KW 48	26,23	Entwicklung GUI, Teamtreffen
KW 49	12,23	Entwicklung GUI
KW 50	41,43	Entwicklung GUI, Teamtreffen, Vorstellung Projekt
KW 51	18,12	Entwicklung GUI, Rücksprache Protokoll
KW 52	0,00	
KW 1	1,75	Präsentationsfolien bearbeiten
KW 2	21,93	Testen, Präsentation, Backup Protokoll entwickeln
KW 3	17,78	Testen, Backup Protokoll entwickeln, Dokumentation bearbeiten, Vorführung Projekt
KW 4	0,00	
KW 5	0,00	
Gesamt	203,82	

Zeiteinteilung Florian Boemmel

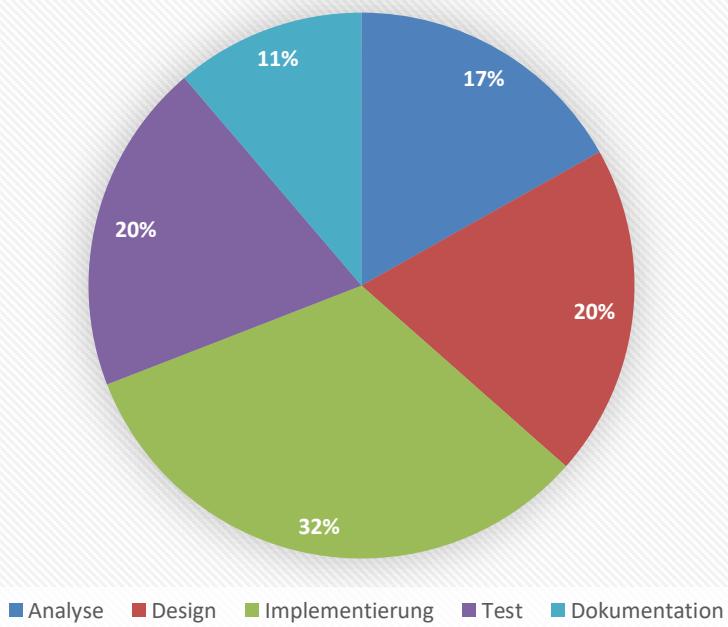




19.4. Robert Graf

	Stunden	Kurzbeschreibung
KW 40	1,50	Rahmenfindung des Projekts
KW 41	2,00	Festlegung des Projekts
KW 42	8,57	Projektbeschreibung schreiben, Terminplanung festlegen
KW 43	4,18	Allgemeine Montageaufgaben, Softwareinstallationen, Recherche
KW 44	9,43	Recherche, Festlegung von Spezifikationen
KW 45	6,97	Versuche zur Seriellen Kommunikation
KW 46	16,93	Erstellung des Protokolls
KW 47	8,57	Überarbeitung Protokoll
KW 48	18,25	Überarbeitung Protokoll, Teamtreffen
KW 49	22,87	Überarbeitung Protokoll, Teamtreffen
KW 50	31,32	Überarbeitung Protokoll, Vorführung Projekt
KW 51	1,00	Überarbeitung Protokoll
KW 52	2,00	Umstrukturierung Protokoll
KW 1	2,00	Präsentation bearbeiten
KW 2	28,68	Fehlersuche Protokoll, Präsentation, Dokumentation bearbeiten
KW 3	11,80	Testen, Vorführung Projekt, Dokumentation bearbeiten
KW 4	2,25	Dokumentation bearbeiten
KW 5	0,00	
Gesamt	178,32	

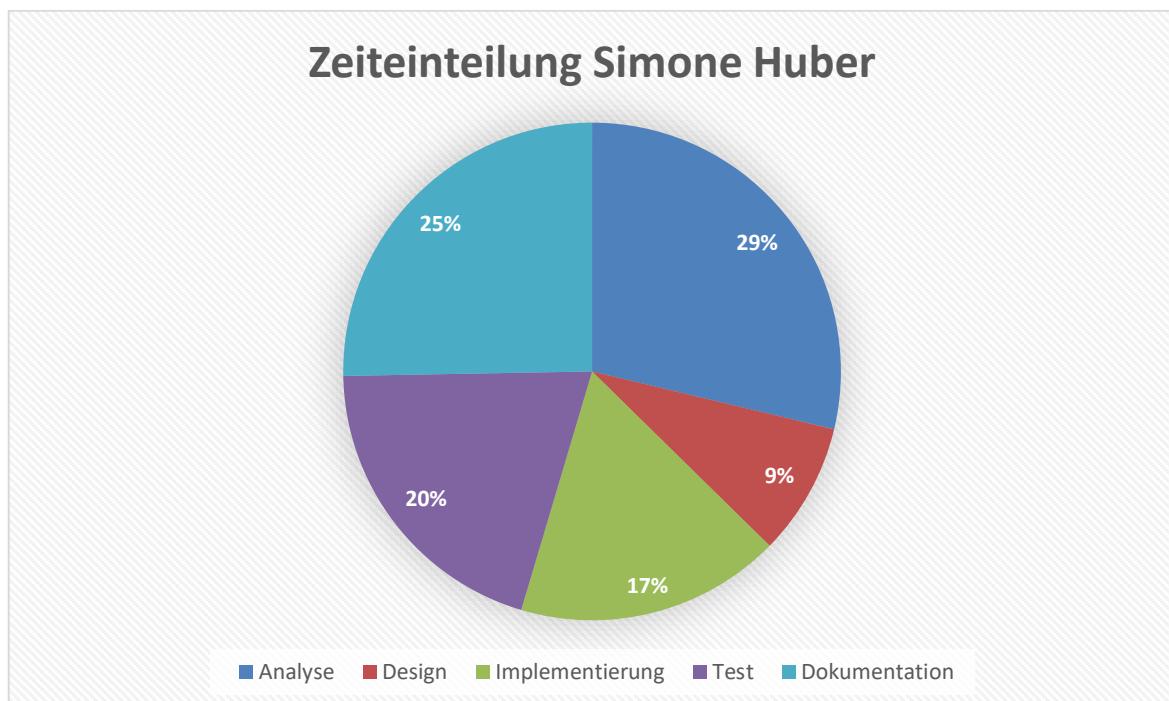
Zeiteinteilung Robert Graf





19.5. Simone Huber

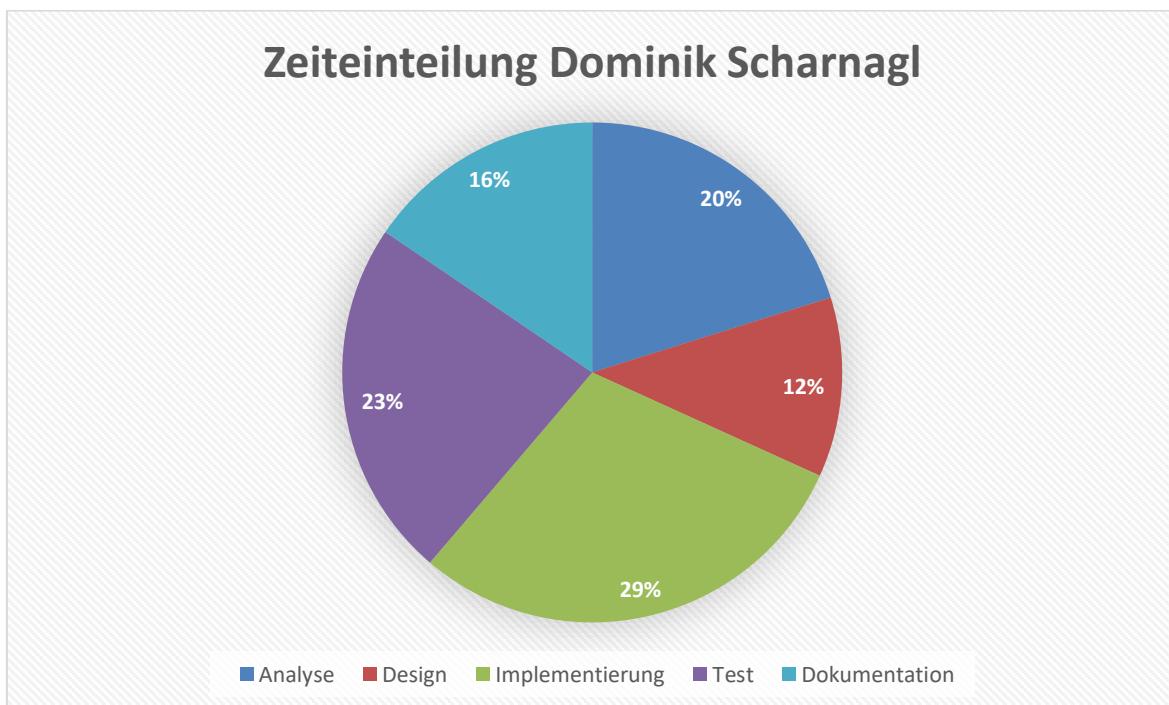
	Stunden	Kurzbeschreibung
KW 40	1,50	Rahmenfindung des Projekts
KW 41	4,00	Festlegung des Projektes
KW 42	7,00	Erstellung des Zeitplanes
KW 43	3,25	Recherche Ultraschall
KW 44	3,25	Teamtreffen
KW 45	10,25	Ultraschall Recherche, Ultraschall Implementierung
KW 46	7,75	Ultraschall Fertigstellung
KW 47	6,00	UWB Recherche
KW 48	10,00	UWB Recherche / Einarbeitung
KW 49	11,00	UWB Einarbeitung, Datenframe Erstellung
KW 50	17,75	UWB Bearbeitung, Datenframe Bearbeitung, Projektzusammenführung, Projektvorführung
KW 51	4,25	UWB Funktionsweise
KW 52	2,25	UWB Funktionsweise
KW 1	31,42	UWB Bearbeitung, Testen, Optimierungen durchführen, Ultraschall-Stopp-Bedingung implementieren
KW 2	20,42	Testen, Erstellung der Präsentation
KW 3	14,00	Finale Tests durchführen, Vorstellung Projekt, Dokumentation erstellen
KW 4	2,00	Dokumentation bearbeiten
KW 5	18,50	Dokumentation bearbeiten, Dokumentation fertigstellen
Gesamt	174,58	





19.6. Dominik Scharnagl

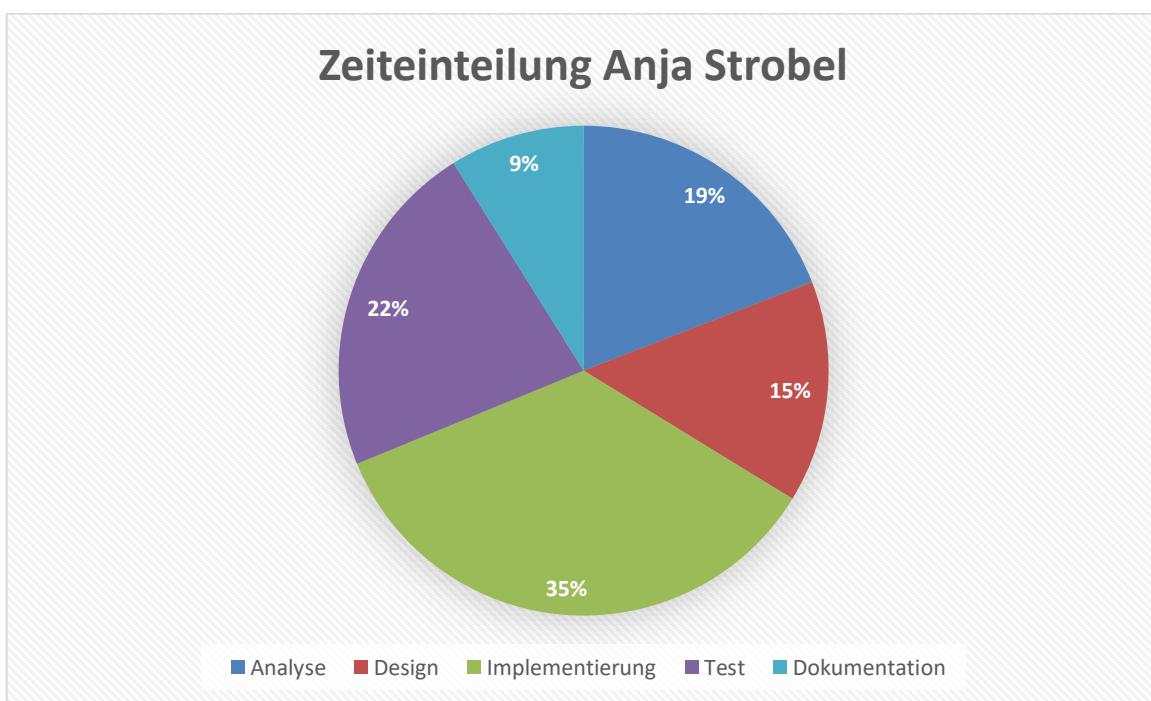
	Stunden	Kurzbeschreibung
KW 40	4,25	Rahmenfindung des Projekts
KW 41	12,25	Festlegung des Projektes, Erstellung GitHub-Projekt
KW 42	9,25	Plakaterstellung, Dokumentation
KW 43	0,00	
KW 44	4,25	Teamtreffen
KW 45	9,25	Motorsteuerung Implementierung
KW 46	14,25	Servosteuerung Implementierung
KW 47	13,25	USB Shield Integration
KW 48	6,25	Sensoren Einbindung
KW 49	11,25	Umsetzung XBOX-Controller-Steuerung
KW 50	20,25	Testen, Projekt Zusammenführung, Projektvorführung
KW 51	18,25	Testen, Zusammenführung Code
KW 52	2,00	Bearbeitung Uhrensteuerung
KW 1	22,00	Implementierung erste Uhr
KW 2	14,25	Implementierung zweite Uhr, Präsentation bearbeiten
KW 3	23,25	Finale Tests durchführen, Vorstellung Projekt, Dokumentation bearbeiten
KW 4	0,00	
KW 5	0,00	
Gesamt	184,25	





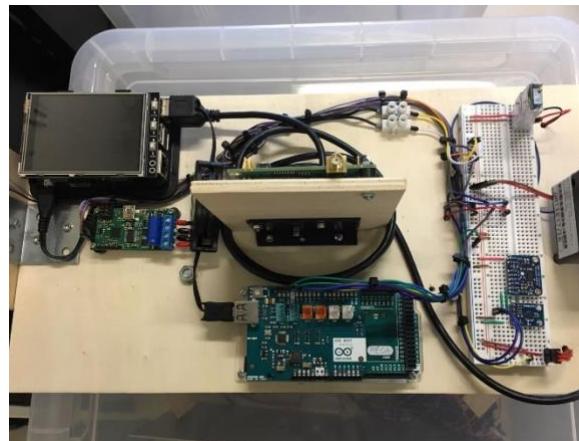
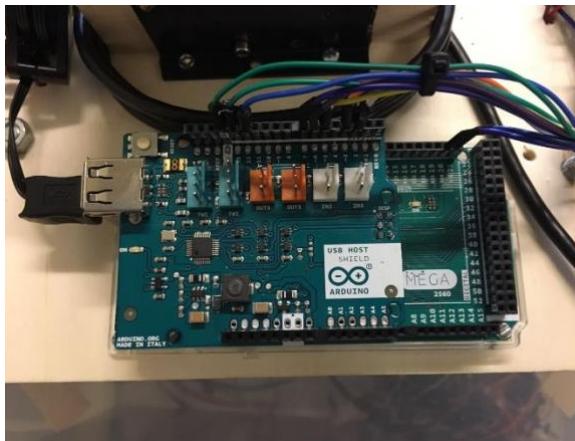
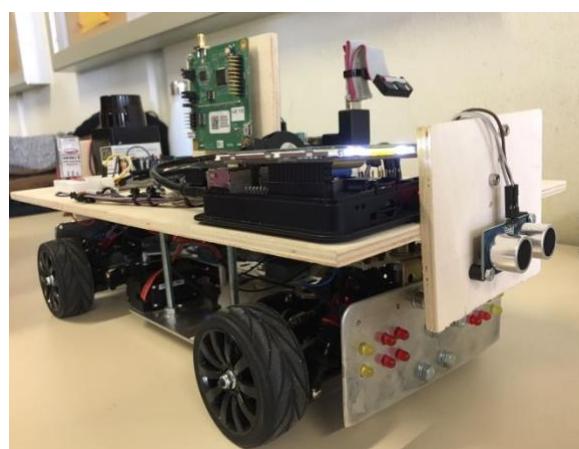
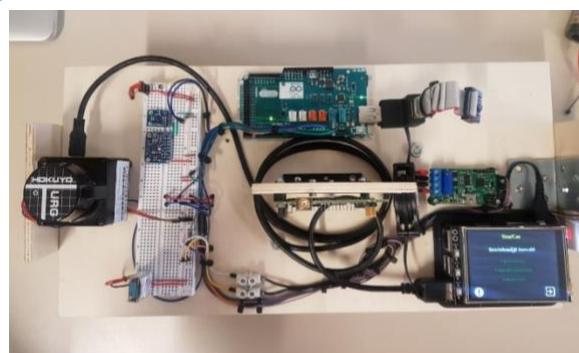
19.7. Anja Strobel

	Stunden	Kurzbeschreibung
KW 40	1,50	Rahmenfindung des Projekts
KW 41	3,00	Festlegung des Projektes
KW 42	3,25	Erstellung des Zeitplanes
KW 43	9,25	Recherche zu URG04_LX-UG01 Lidar, Planerstellung
KW 44	6,50	Teamtreffen, Lidar Testmessung
KW 45	8,75	Teamtreffen, Erstellung Programm für Lidar
KW 46	13,50	Bearbeitung Lidar-Auswertung, Fehlersuche
KW 47	10,50	Teamtreffen, Lidar Fehlerbehebung
KW 48	11,25	Matlab-Auswertung erstellen
KW 49	16,25	Teamtreffen, Recherche Übertragung
KW 50	28,00	Testen, Debuggen, Verbesserung des Codes
KW 51	8,25	Testdaten für Präsentation aufbereiten
KW 52	0,00	
KW 1	0,00	
KW 2	13,75	Präsentation bearbeiten
KW 3	14,75	Testen, Anpassungen Matlab
KW 4	8,50	Dokumentation bearbeiten
KW 5	0,00	
Gesamt	157,00	





20. Abschließende Eindrücke





21. Abbildungsverzeichnis

Alle Abbildungen zu 13.Kommunikationsprotokoll befinden sich auch im Projektverzeichnis unter ./docs/team/grr37213/

- 13.1 : [V0.0] trivialer Protokollablauf
- 13.2 : [V0.1] Packet
- 13.3 : [V0.1] Transceiver
- 13.4 : [V0.1] Inbox
- 13.5 : [V0.1] Rule
- 13.6 IBC_Config
- 13.7 [V0.3] Request und Response
- 13.8 [V0.3] Negative Response
- 13.9 [V0.3] IBC
- 13.10 [V0.3] Pi-Codebeispiel
- 13.11 [V0.3] Klassendiagramm
- 13.11 [V1.2] Ablaufsequenzdiagramm

Arduino

<https://www.arduino.cc/en/Trademark/CommunityLogo>

<http://cdn.instructables.com/FSS/Q6FC/I0NOI8UM/FSSQ6FCI0NOI8UM.MEDIUM.jpg>

Pololu Motor Controller

<https://a.pololu-files.com/picture/0J2860.250.jpg>

RC-Car Servo

https://asset.conrad.com/media10/isa/160267/c1/-/de/206461_BB_00_FB/rc-car-servo-4519-dbb-mg.jpg

Xbox Controller

<https://www.idea-booth.com/img/cases/controller.png>

eZ430-Chronos-Watch + AccessPoint

<https://hackadaycom.files.wordpress.com/2009/11/rf-development-platform.jpg>

<http://www.ti.com/lit/ug/slau292g/slau292g.pdf>

Ultraschallsensor

<https://www.parallax.com/product/28015>



22. Literaturverzeichnis

Datenblatt₁ Kompass Sensor

https://cdn-shop.adafruit.com/datasheets/HMC5883L_3-Axis_Digital_Compass_IC.pdf

<https://www.arduino.cc/en/Reference/Wire>

Datenblatt₂ Beschleunigungssensor

<https://www.adafruit.com/product/1231>

<https://www.sparkfun.com/datasheets/Sensors/Accelerometer/ADXL345.pdf>

https://github.com/adafruit/Adafruit_ADXL345

Arduino Uno R3

<https://store.arduino.cc/usa/arduino-uno-rev3>

Arduino Mega 2560

<https://store.arduino.cc/arduino-mega-2560-rev3>

Arduino Servo Library für Servosteuerung

<https://www.arduino.cc/en/Reference/Servo>

Arduino SoftwareSerial Library für Motorsteuerung

<https://www.arduino.cc/en/Reference/SoftwareSerial>

Pololu Motor Controller

<https://www.pololu.com/product/1379>

<https://www.pololu.com/docs/0J44/6.2.1>

RC-Car Servo

<https://www.conrad.de/de/rc-car-servo-4519-dbb-mg-206461.html>

USB Host Shield Library 2.0 für Controller und Watch Steuerung

http://felis.github.io/USB_Host_Shield_2.0/

Tutorial zur Watch Integration (als Referenz)

<http://www.instructables.com/id/Control-an-Arduino-With-a-Wristwatch-TI-eZ430-Chr/>



eZ430-Chronos Watch + AccessPoint

- Datenblatt: <http://www.ti.com/lit/ds/symlink/cc430f6137.pdf>
- CC430 User Guide: <http://www.ti.com/lit/ug/slau259e/slau259e.pdf>
- eZ430-Chronos Dev. Tool: <http://www.ti.com/lit/ug/slau292g/slau292g.pdf>
- eZ430-Chronos Wiki
 - <http://processors.wiki.ti.com/index.php/EZ430-Chronos?DCMP=Chronos&HQS=Other+OT+chronoswiki>
 - http://processors.wiki.ti.com/index.php/CC430_General_Overview
- Implementierungsbeispiele
 - <http://www.ti.com/lit/zip/slac279>
 - <http://www.ti.com/lit/zip/slac525>

Parallax Ultrasonic Distance Sensor

<https://www.parallax.com/product/28015>

UWB Sensor

<https://sourceforge.net/projects/realterm/>
[\(https://www.decawave.com/video/trek1000-quick-start-video\)](https://www.decawave.com/video/trek1000-quick-start-video)