OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

OTH Regensburg

Fakultät für Informatik und Mathematik

HSP-Projekt SS17

Implementierung eines SLAM-Algorithmus auf einem autonom-betriebenen Fahrzeug

von

Johannes Lex	3098145
Wolfram Schießl	2903273
Patrick Baumann	3096282
Johannes Schwickerath	3118757

am

31.01.2018

Betreuender Professor: Prof. Dr. Alexander Metzner

Inhaltsverzeichnis

1. Autonomes Fahren

Unter autonomem Fahren versteht man die selbstständige Fortbewegung eines Fahrzeugs, welches ohne menschliches Einwirken auf Lenk- oder Geschwindigkeitsverhalten auskommt. Damit ein Fahrzeug autonom fahren kann, sind verschiedene Sensoren notwendig, die beispielsweise den Raum vor dem Fahrzeug detektieren können. Damit ein Fahrzeug neben der unmittelbaren Hinderniserkennung auch einen Weg in der Umgebung durch die Hindernisse zurücklegen kann, wird eine Karte benötigt. Für das Erstellen einer solchen Karte gibt es verschiedene Möglichkeiten. Eine davon ist der **S**imultanous **L**ocalization **A**nd **M**apping (im Folgenden kurz SLAM) -Algorithmus.

Der SLAM-Algorithmus erstellt eine Karte mit Hilfe der Daten der aktuellen Umgebung während der Fahrt und ermittelt seine eigene Position relativ zum Startpunkt der Messung. Das Fahrzeug ist somit in der Lage selbständig zu fahren ohne eine zu Beginn existierende Umgebungskarte. Falls eine Karte zu einem bereits befahrenen Gebiet existiert, kann diese bei erneutem Befahren, durch eventuell hinzugekommene Hindernisse aktualisiert werden.

2. Vorgängerprojekt

In diesem Kapitel werden die Errungenschaften des vorangegangenen Projekts geschildert.

2.1. Übersicht Vorgängerprojekt

Zu Projektstart war ein batteriebetriebenes Fahrzeug mit verschiedenen Sensoren und Aktoren vorhanden. Als Steuerungseinheiten fundierte ein ***S**ystem **o**n **C**hip* (im Folgenden kurz SoC) für die Sensoren sowie für die Aktoren. Über eine grafische Bedienoberfläche (im Folgenden kurz GUI) kann der Nutzer das Fahrzeug steuern. Diese GUI wird von einem Rechner ausgeführt, welcher per WLAN-Verbindung mit dem SoC-Board verbunden ist. In der GUI werden verschiedene Stati des Fahrzeugs sowie Sensordaten visualisiert. Zum Steuern der Fahrtrichtung, zum Verbinden per WLAN sowie zum Einschalten des Lichts sind Bedienelemente vorhanden.

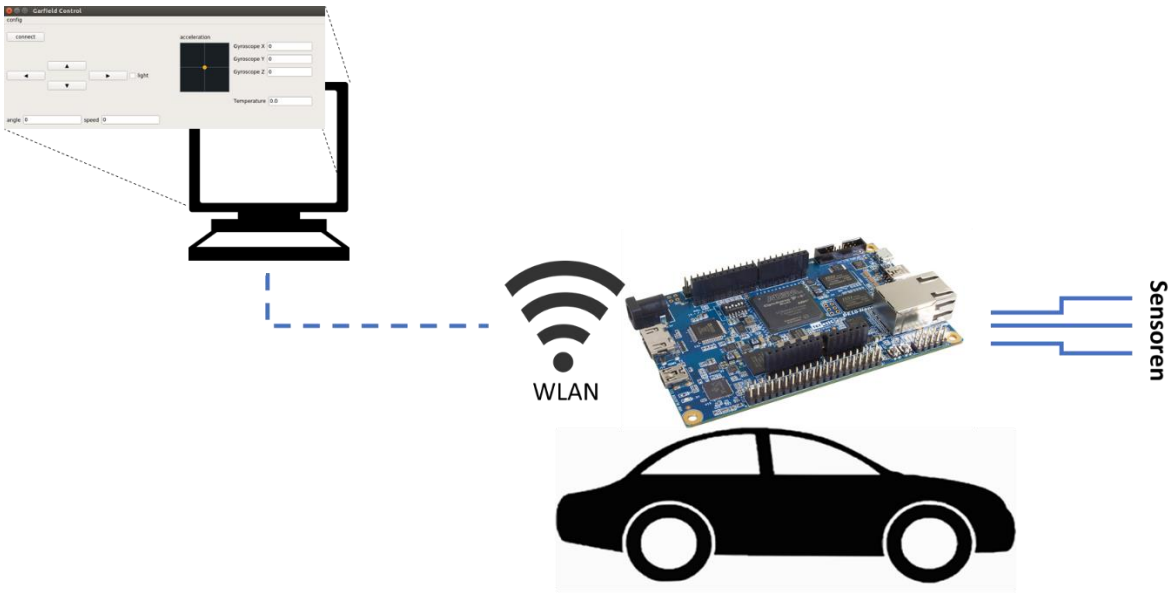

Abbildung 1: Hardwareaufbau - Überblick

2.2. Hardwareaufbau

Das SoC, welches für das Projekt genutzt wird, ist ein **Cyclone V SoC** von **Altera**. Dieses kann man unterteilen in:

- **F**ield **P**rogrammable **G**ate **A**rray (im Folgenden kurz FPGA)
  Für das FPGA kann eine gewünschte Digitalschaltung implementiert werden. Innerhalb des Vorgängerprojekts realisiert das FPGA die Soft-CPU **NIOSII**, welche von Altera zur Verfügung gestellt wird. Diese CPU hat eine 32-Bit RISC Architektur. Das Durchführen der Anwendungen auf dem FPGA wird über **FreeRTOS**, ein Echtzeitbetriessystem für kleine Eingebettete Systeme, umgesetzt.
  Im Ramen des Projektes sind alle Aktoren und Sensoren über verschiedene Kommunikationskanäle (wie SPI, I2C) mit dem FPGA verbunden.
  [1] [2]

- **H**ard **P**rocessor **S**ystem (im Folgenden kurz HPS)
  Der HPS bietet ein komplettes System mit Prozessor, Peripherie und Speicher-Schnittstellen. Beim Prozessor handelt es sich um einen **Cortex A9 ARM**. Auf einer angeschlossenen SD-Karte ist eine **Linux** Distribution installiert die auf dem HPS ausgeführt wird. Alle Programme des Vorgängerprojekts sowie dieses Projekts werden in diesem Linux ausgeführt.
  Über einen Backbone mit einer Bandbreite von 100 Gbps ist die Kommunikation zwischen FPGA und HPS möglich. Die Kommunikation wird ausführlicher in Kapitel 2.3 erläutert. [3]

Abbildung 2 zeigt einen Überblick über den geschilderten Hardwareaufbau.

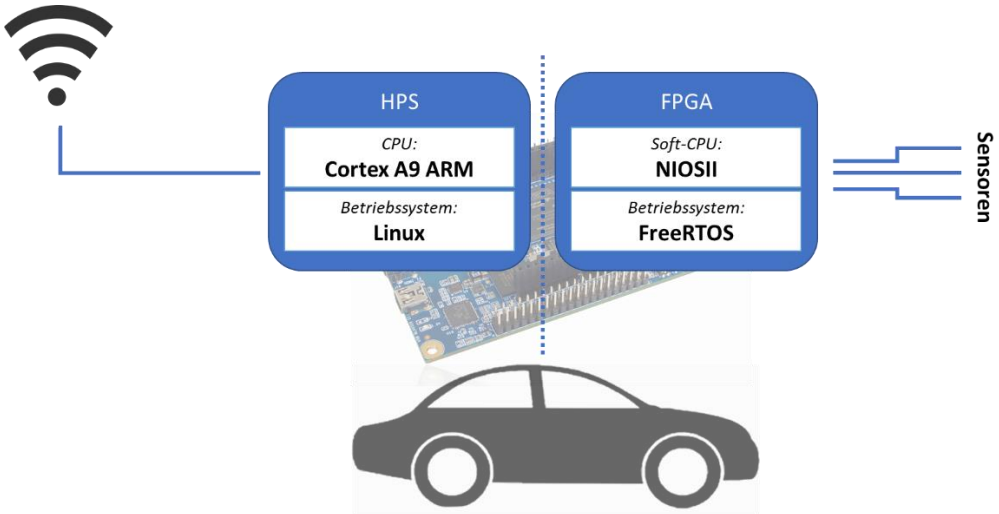

Abbildung 2: Komponenten des SoC

2.3. Kommunikation HPS – FPGA

Auf Abbildung 3 ist der Kontrollfluss der IP-Cores zu sehen, die an der Kommunikation zwischen NIOS2 und HPS beteiligt sind.

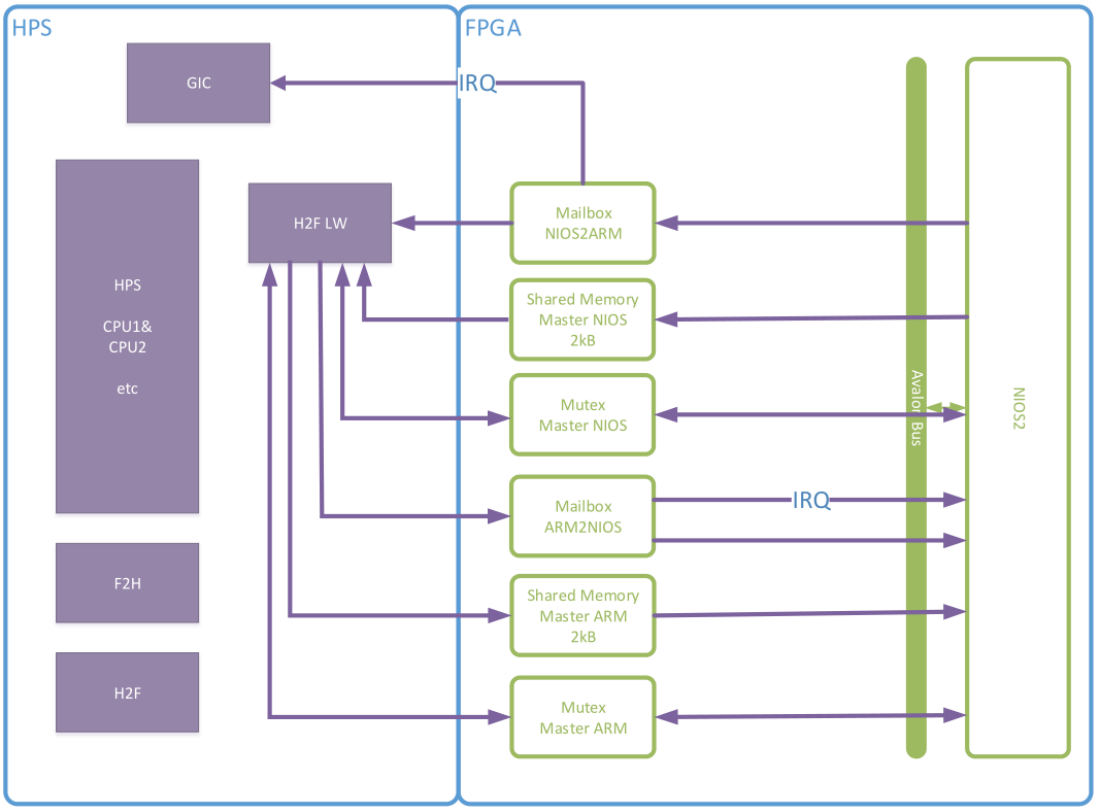

Abbildung 3: Kommunikation zwischen HPS und FPGA [4]

Für die Kommunikation zwischen FPGA und ARM-Prozessor wurden verschiedene IP-Cores erstellt, die als Mailbox bezeichnet werden. Eine Mailbox kann Nachrichten zwischen zwei Buspartnern austauschen. Sie ist zudem unidirektional und fungiert daher entweder als Sender oder Empfänger. Zusätzlich zu den Mailbox- IP-Cores gibt es einen Shared Memory. Während über das Mailbox-System nur die Adresse im Shared Memory und ein Kommando übertragen werden, findet die Nutzdatenübertragung im Shared Memory statt [4]. Die Kommunikation mit Mailbox und Shared Memory erfolgt FPGA-seitig über einen NIOS2-Prozessor. Der Instruktions-und Datenspeicher dieses Prozessors sowie der Shared Memory wurde mit Onchip-Memory des FPGA realisiert.

Mehr Informationen zur Kommunikation zwischen FPGA und ARM-Prozessor können in der Dokumentation des Vorgängerprojekts *„Hsp – Projektbericht“* [4] nachgelesen werden.

2.4. Softwareaufbau

Die Software des Vorgängerprojekts setzt sich aus 3 Teilen zusammen:

1. HQ (**H**ead**Q**uarter) - Software – ausgeführt auf dem PC:
   HQ ist die Graphische Oberfläche(GUI) zum Ansteuern des Fahrzeugs. Zudem kann der Nutzer die gemessenen Sensorwerte darin einsehen. Die GUI läuft auf einem Linux Betriebssystem. Für die Kommunikation wird eine Socketverbindung mit dem Fahrzeug über WLAN erzeugt.

2. ARM Software – ausgeführt auf dem HPS:
   Auf dem ARM-Prozessor wird ein Linux Betriebssystem ausgeführt, welches die Netzwerkaufgaben mit dem HQ übernimmt. Es dient somit als Kommunikations-gateway zwischen HQ und NIOS2.

3. NIOS2 - Software– ausgeführt auf dem FPGA:
   NIOS2 ist für das Ansprechen der Sensoren und Aktoren zuständig. Es liest die Sensorwerte aus und schickt diese an ARM. Zudem werden die empfangenen Nutzereingaben interpretiert und die passenden Aktionen (zB.: Einschalten von Licht, Drehung der Hinterräder) ausgeführt.

2.5. Sensoren

Um das autonome Fahren zu ermöglichen, wurden verschiedene Sensoren auf dem Fahrzeug installiert. Es sind vorhanden:

- SRF08-Ultraschallsensoren
  Diese werden für die Erkennung von Hindernissen unmittelbar vor bzw. hinter dem Fahrzeug verwendet. Beim Erkennen eines Hindernisses soll das Fahrzeug stoppen.
- MPU6050
  Über den MPU6050 können die 3 Beschleunigungsachsen, die 3 Drehachsen und die Temperatur ausgelesen werden. Diese Daten werden an die HQ weitergeleitet, wo der Nutzer diese einsehen kann.
- LIDAR- Sensor
  Der LIDAR- Sensor wird in Kapitel 2.6 beschrieben.

2.6. LIDAR-Sensor

Die Daten, die zur Verarbeitung im SLAM- Algorithmus verwendet wurden, stammen aus einem ***LI**ght **D**etection **A**nd **R**anging*- (im Folgenden kurz LIDAR) Sensor. Verwendet wurde ein **URG04LX** der Firma **HOKUYO**, der speziell für die Anwendung bei mobilen Robotern ausgelegt ist. Ein LIDAR- Sensor sendet Laserstrahlen an die Umgebung und kann durch die Reflektion der Strahlen die Entfernung zu umgebenden Gegenständen ermitteln. Der vorliegende Sensor hat einen Messbereich von 240°, wobei alle 0,36°eine Teilmessung stattfindet. Eine komplette 240°-Messung dauert 100 Millisekunden. Die Messweite liegt bei 4 Metern. Zur Anbindung an das SoC-Board wurde die USB-Schnittstelle des Sensors verwendet. Diese dient auch als Stromzufuhr für den LIDAR- Sensor [5]. In Abbildung 4 ist der verwendete LIDAR-Sensor zu sehen.

Abbildung 4: LIDAR-Sensor Hokuyo URG04LX [6]

3. Projektanforderungen

In diesem Kapitel werden die Errungenschaften des vorangegangenen Projekts sowie die eigenen Ziele grob geschildert.

3.1. Gewünschtes Endprodukt

Ziel ist ein autonomes Fahrzeug, das auf Wunsch und ohne menschliche Hilfe von z.B. einem beliebigen Raum in einen anderen fahren kann. Es ist darauf zu achten, dass Hindernisse, wie Stühle oder Tische, innerhalb der Räume eventuell umgestellt werden. Das Fahrzeug muss diese Änderung während der Fahrt registrieren und diese umfahren. Bestenfalls soll das Fahrzeug sich auch in Umgebungen fortbewegen können, in denen es zuvor nicht gefahren ist und daher auch keine Umgebungskarte von diesen Umgebungen erstellt hat. Dieses Ziel ist in mehrere Teilziele aufgeteilt, sodass es in mehreren HSP-Projekten realisiert werden kann.

3.2. Zielsetzung des Projekts

Als wesentlicher Schritt vom manuellen Fahren zum autonomen Fahren ist es notwendig, dass das Fahrzeug in einer Umgebung eine Strecke ohne Eingreifen eines Benutzers zurücklegen kann. Die Ultraschallsensoren sorgen dafür, dass das Fahrzeug mit seiner unmittelbaren Umgebung nicht kollidiert. Um einen Weg von einem Ort zu einem anderen autonom zu fahren, wird eine Umgebungskarte benötigt. Zum Auslesen der Umgebung wird ein LIDAR-Sensor verwendet.

Ziel des vorliegenden Projekts war es, einen geeigneten SLAM-Algorithmus auf dem HPS des Fahrzeugs auszuführen. Durch diesen soll es möglich sein eine Umgebungskarte zu erzeugen, die für ein autonomes Fahren genutzt werden kann. Die Umsetzung dieses Projekts umfasst die folgenden Punkte:

1. Für das Fahrzeug muss ein passender SLAM-Algorithmus gewählt werden. Das HPS besitzt begrenzten Speicher und auch die Prozessorleistung ist begrenzt. Diese sollen nicht voll ausgelastet werden, sodass das Echtzeitverhalten für das Steuern gefährdet wird. Das Fahrzeug soll zeitgleich zur Ausführung des SLAM-Algorithmus auf die Nutzereingaben u.a. zum Steuern des Fahrzeugs reagieren können.

2. Der SLAM-Algorithmus wird auf dem HPS ausgeführt. Hierbei sollen die benötigten Datenwerte direkt vom LIDAR-Sensor verwendet werden, um eine Umgebungskarte erstellen zu können. Für die weitere Optimierung der Karte soll der SLAM-Algorithmus verschiedene Sensoren einbeziehen.
3. Funktionen, die im ausgewählten SLAM-Algorithmus nicht vorhanden sind, müssen selbstständig implementiert werden. Die Anpassung des SLAM-Algorithmus wird in Kapitel 5.3 geschildert.
4. Dem Benutzer soll eine Echtzeitdarstellung der Umgebungskarte auf der HQ angezeigt werden. Zusätzlich soll die momentane Position des Fahrzeugs auf der Karte angezeigt werden.

4. Einarbeitung

In diesem Kapitel wird auf die Einarbeitung in das Projekt und auf die Probleme bei der Einarbeitung eingegangen.

4.1. Bugfix Lichtansteuerung

Bei der Lichtansteuerung des Vorgängerprojekts funktionierte die Lichtansteuerung nicht, da eine falsche Funktion aufgerufen wurde. Dieses Problem wurde als Teil dieser Projektarbeit gelöst. Hierbei wurde der Aufruf der Funktion *IOWR(…)* durch den Aufruf der Funktion *IOWR_ALTERA_AVALON_PIO_DATA(…)* ersetzt. Diese Änderung erfolgte innerhalb der Datei *task_nios.cpp*.

4.2. Probleme bei der Einarbeitung

Während der Einarbeitung ergaben sich mehrere verschiedene Probleme, die in diesem Kapitel beschrieben werden.

Die erste Hürde des Projekts war die Projektstruktur nachzuvollziehen. Ein Problem war, dass keine Makefiles oder andere Projektdateien vorhanden waren. Die Projekte mussten selbstständig von vorn angelegt werden. Fehlende Tutorials hat die Rekonstruktion der Projektstruktur erschwert.

Für das Kompilieren des Softwareprojekts für den ARM - Controller wurde ein Compiler verwendet, der nicht bekannt war. Leider wurde dieser auch nicht in der Dokumentation erwähnt. Das ARM-Projekt benötigt den ARM-Linux-gnueabihf-Compiler (Version 5.3-2016.02). Dieser kann auf der folgenden Seite heruntergeladen werden:

https://releases.linaro.org/components/toolchain/binaries/5.3-2016.02/arm-linux-gnueabihf/

Um den HQ-Teil der Software zu kompilieren, musste das QT-Framework verwendet werden. Hier war nicht klar welche QT-Version für die Kompilierung der Software notwendig ist. Für dieses Projekt wurde die neuste funktionierende Version verwendet. Es handelt sich um die Qt version 5.5.1. Bei einer 4.x.x – Version des QT- Frameworks traten Probleme auf, so dass diese Version nicht verwendet werden konnte.

Um die beschriebenen Probleme bei nachfolgenden Projektgruppen auf ein Minimum zu reduzieren, wurden verschiedene Dokumente ausgearbeitet, die unterschiedliche Arbeitsvorgänge genau beschreiben. Zusätzlich dazu sind diese Arbeitsvorgänge im Anhang diesem Dokument beigefügt.

5. SLAM

In diesem Kapitel wird der SLAM-Algorithmus genauer erklärt. Dabei wird im nachfolgenden Kapitel zunächst auf die Grundlagen eingegangen. In den darauffolgenden Kapiteln werden der verwendete SLAM-Algorithmus sowie die Anpassungen im Rahmen dieser Projektarbeit erläutert.

5.1. Grundlagen des SLAM-Algorithmus

SLAM ist ein Algorithmus, der die unmittelbare Umgebung und die Position eines Objekts z.B. eines Roboters in einer Umgebungskarte wiederspiegeln kann. SLAM ist besonders nützlich in Situationen, in denen ein Mensch keinen Einfluss auf das Objekt hat. Die Besonderheit eines SLAM-Algorithmus ist, dass unter der Fahrt die Umgebungskarte erstellt wird. Es muss keine Karte zu Beginn vorhanden sein.

5.2. BreezySLAM-Algorithmus

BreezySLAM ist eine simple und effiziente Open-Source Bibliothek für die simultane Lokalisierung und Erstellung der Umgebungskarte.

Ausgehend von der Dokumentation des HSP – Projektes [7], schien der Algorithmus am besten geeignet. Die wichtigsten Vorzüge sind nachfolgend aufgelistet:

- Rechenleistung im Vergleich zu anderen SLAM-Implementierungen gering
- Ausführliche Dokumentation
- Beispielprojekte zum Einarbeiten
- Modifikationen leicht vorzunehmen, da eine Open Source Lizenz zugrunde liegt
- Wenig externe Abhängigkeiten im Gegensatz zu anderen SLAM Algorithmen

Zur Wegfindung beim BreezySlam gibt es zwei algorithmische Ansätze. Einer davon benutzt keine LIDAR – Daten, sondern ausschließlich Odometriedaten und wird auch *Deterministic - SLAM* genannt. Der andere Ansatz heißt ***R**andom **M**utation **H**ill **C**limbing* – SLAM (RMHC) und ist zum Schätzen der neuen Postition gedacht. RMHC ist ein Suchalgorithmus der die Tendenz sich in lokalen Maxima zu verfangen minimiert. Mit Hilfe der Odometriedaten wird zunächst die Startposition ermittelt und zur Verbesserung der Genauigkeit des Ergebnisses werden anschließend Partikelfilter verwendet. [8]

5.3. Eigene Anpassungen an dem BreezySLAM

Der BreezySLAM wird in der ARM-Software aufgerufen. Jeglicher ARM-Code wird im HPS ausgeführt, somit läuft der SLAM-Algorithmus direkt auf dem Fahrzeug. Der Algorithmus kann dadurch schneller auf die Sensorwerte zugreifen, was zu einer echtzeitfähigen Verarbeitung der Daten führt. Wenn der Algorithmus auf dem PC im HQ-Code ablaufen würde, käme es zu einer zeitlichen Verzögerung. Mit der Ausführung des SLAM-Algorithmus auf dem Fahrzeug möchte man zudem, dass das Fahrzeug möglichst autark, ohne Einflüsse von Außerhalb, ist. In Zukunft soll es autonom fahren.

Die Quelldateien werden in Form von Bibliotheken in die ARM-Software eingebunden. Mit deren Hilfe kann der BreezySLAM genutzt werden. Die BreezySLAM-Dateien müssen getrennt vom ARM-Code kompiliert werden, da sie ein spezielles Makefile nutzen. Das Makefile ermöglicht, dass die einzelnen Dateien mit den passenden Includes und in richtiger Reihenfolge kompiliert werden. Der standardmäßige Compileraufruf von Eclipse ist dazu nicht imstande.

Die BreezySLAM-Bibliothek mit dem Namen "**libbreezyslam.so**" ist im Standard - Include Pfad für Bibliotheken **/usr/local/lib** auf dem Linuxsystem des ARM-Controllers hinterlegt. Dieser Pfad kann durch die LibraryDirectory-Variable **LIBDIR** angepasst werden.

Die Main-Datei **Comm_Gateway.cpp** des ARM-Code musste angepasst werden, damit sie die nötigen BreezySLAM Funktionen startet. Sie nutzt nur indirekt die BreezySLAM-Library. Alle Aufrufe an die BreezySLAM-Library erfolgen über einen eigens angelegten Wrapper in **usebreezyslam.hpp**. Der Wrapper soll das Nutzen des Algorithmus erleichtern und dessen Funktionalität übersichtlicher gestalten. Der kompilierte ARM-Code ist unter dem Namen **HSP_ARM_Gateway** im Pfad **<home>/bin**.

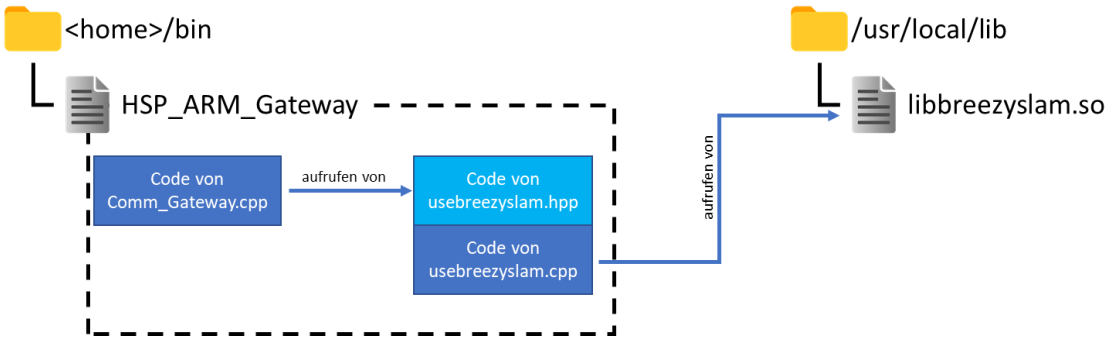

Abbildung 5: Überblick - Einbinden BreezySLAM in ARM-Software

In **Comm_Gateway.cpp** wird zunächst der BreezySLAM gestartet. Die Start-Funktion ist im Wrapper implementiert und enthält die folgenden Schritte:

1. **Aufbauen einer Verbindung zum LIDAR-Sensor**:
   Über diese Verbindung werden dessen Sensorwerte ausgelesen.

2. **Initialisieren des RMHC SLAM**:
   Dieser Algorithmus ist der Hauptbestandteil des BreezySLAM. Er ist für die Erstellung der Umgebungskarte und für das Lokalisieren des Fahrzeugs zuständig. Im Projekt wird der RMHC SLAM bevorzugt, da momentan keine Odometriewerte zur Verfügung stehen. Im Gegensatz zum Deterministic SLAM werden hierbei nur LIDAR-Sensorwerte benötigt. Bei den regelmäßigen Updates der Karte und Fahrzeugposition müssen nicht unbedingt Odometriewerte angegeben werden.

3. **Laden einer vorhandenen Karte**:
   Eine Karte wird aus einem Grauwertbild ausgelesen und in den Algorithmus geladen. Diese Funktion soll ermöglichen, dass eine erzeugte Karte weiter ausgebaut werden kann. Die erneute Ausführung der Anwendung soll somit die zuletzt vorhandene Karte als Basis nutzen. Für diese Funktion muss der Pfad für die einzulesende Karte, sowie die Startposition des Fahrzeugs angegeben werden. Für Vorführungszwecke wird die Karte aus der Datei **<home>/bin/srcSlamMap.pgm** entnommen mit dem Mittelpunkt der PGM-Datei als Startposition.

4. **Erzeuge Thread für Updaten des Algorithmus**:
   Für den Betrieb des Algorithmus muss dieser regelmäßig mit den aktuellsten LIDAR-Sensorwerten aktualisiert werden. Hierfür wird ein eigener Thread erzeugt, welcher das Aktualisieren übernimmt. Falls vorhanden kann man zusätzlich noch Odometriewerten angeben, um die Qualität der Lokalisierung zu verbessern.

Nach dem Starten des BreezySLAMs werden in der Main-Methode alle nötigen Threads erzeugt. Dazu gehören diejenigen, welche die Sensorwerte vom FPGA entnehmen oder die Nachrichten an die HQ senden/empfangen. In unserem Projekt erzeugen wir einen zusätzlichen Thread für das zyklische Speichern der momentanen Karte im BreezySLAM. Die Karte wird im HPS unter dem Pfad **<home>/bin** gespeichert. Die Karte ist eine PGM-Datei mit dem Namen **savingSlamMap.pgm**.

Beim Beenden der Anwendung durch den Benutzer werden alle erzeugten Threads freigegeben. Zudem wird auch der BreezySLAM beendet. Dessen Beenden beinhaltet sowohl das Freigeben des Objekts für den Algorithmus, als auch den Thread für das Aktualisieren. Des Weiteren schließt es die vorhandene Verbindung zum LIDAR-Sensor.

Abbildung 6 veranschaulicht den gesamten Ablauf des BreezySLAMs in der ARM-Applikation.

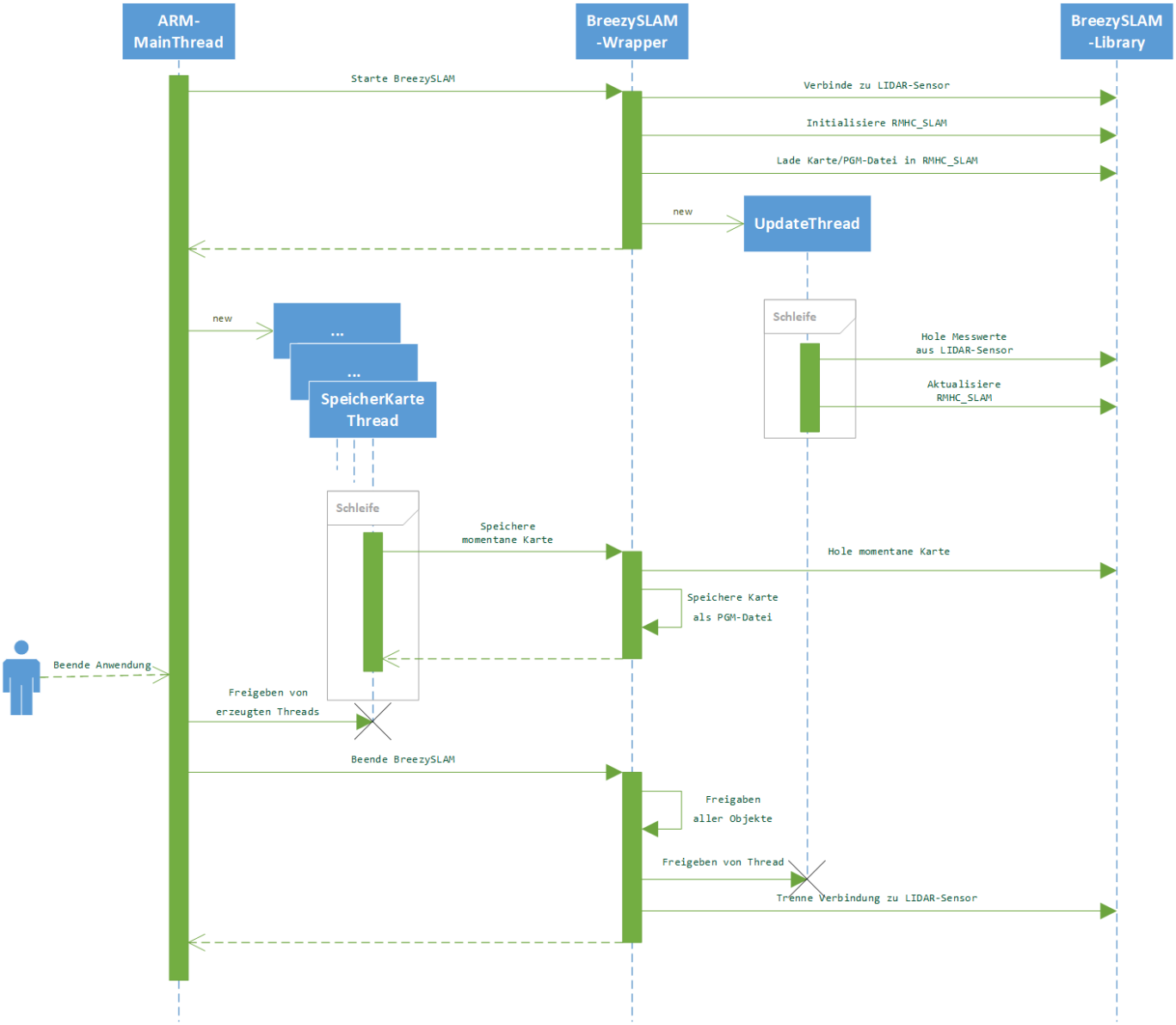

Abbildung 6: Ablauf BreezySLAM in ARM-Applikation

6. Graphical User Interface (GUI)

6.1. Vorhandene GUI

In nachfolgender Abbildung ist die GUI zum Stand der Übernahme abgebildet. Zu sehen sind Buttons mit Richtungspfeilen auf der linken Seite. Mit diesen kann das Fahrzeug gesteuert werden. Oberhalb ist ein Button (*„connect“*), welcher dem Verbinden der GUI mit dem Fahrzeug dient. Unterhalb sind Anzeigen angebracht, um Richtung und Geschwindigkeit auszulesen. Diese werden nicht verwendet. Auf der rechten Seite der GUI sind Anzeigen für Odometrie- Daten und für die Temperatur zu sehen. Auch diese werden aktuell nicht verwendet.

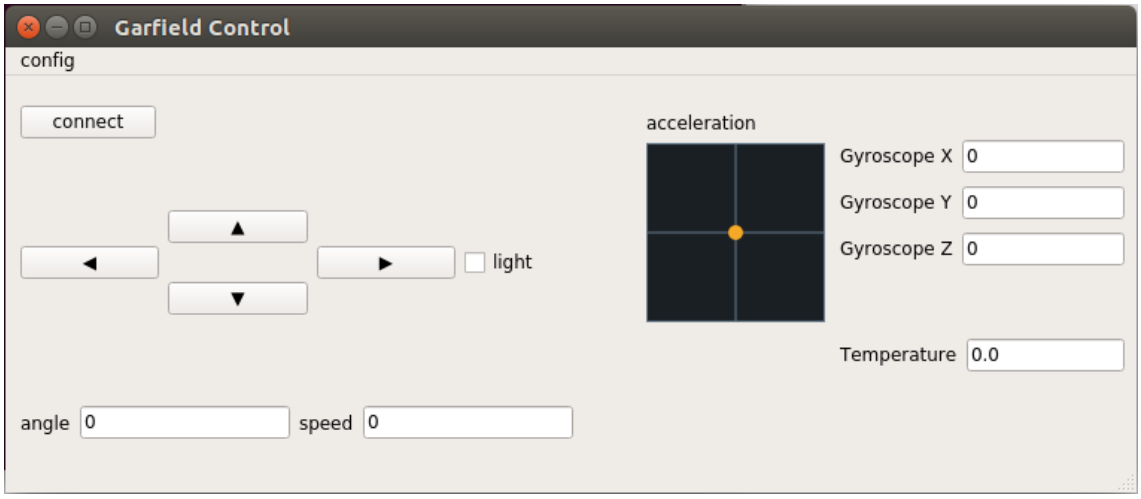

Abbildung 7 GUI des Vorgängerprojekts

6.2. Erweiterungen zur vorhandenen GUI

Damit die mit SLAM-Algorithmus erstellte Karte und die Position des Fahrzeugs vom Benutzer überwacht werden kann, wurde die GUI um eine Anzeige erweitert. Diese ist auf Abbildung 8 zu sehen. Auf dieser Anzeige sind die aufgenommene Karte, sowie die Position des Fahrzeugs zu sehen.

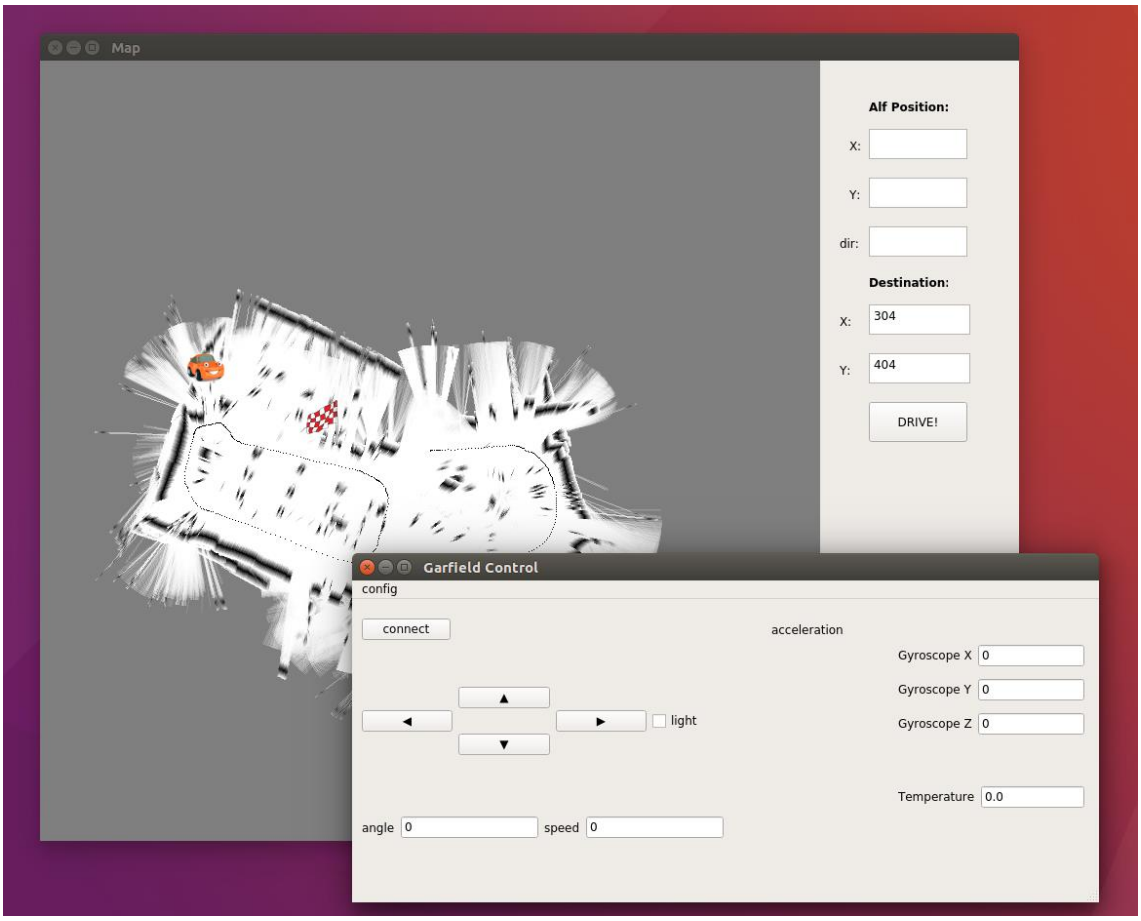

Abbildung 8 Erweiterte GUI

6.3. Erweiterter Kommunikationskanal

In Abbildung 9 ist der Kommunikationskanal zwischen ARM-Prozessor auf dem SoC - System und HQ – Software im Linux-Betriebssystem der Virtuellen Maschine abgebildet. Der Aufbau der Kommunikation wird im Folgenden beschrieben.

Zunächst initialisiert sich der ARM - Server und wartet dann, bis von der GUI der *„connect“*-Button betätigt wird und der HQ-Client eine Verbindungsanfrage an den ARM-Server sendet. Der ARM-Server startet nach Eingang dieses Signals die Threads recThread, sendThread und mapThread. Gleichzeitig startet der HQ-Client seinerseits Threads die namentlich gleichen Threads.

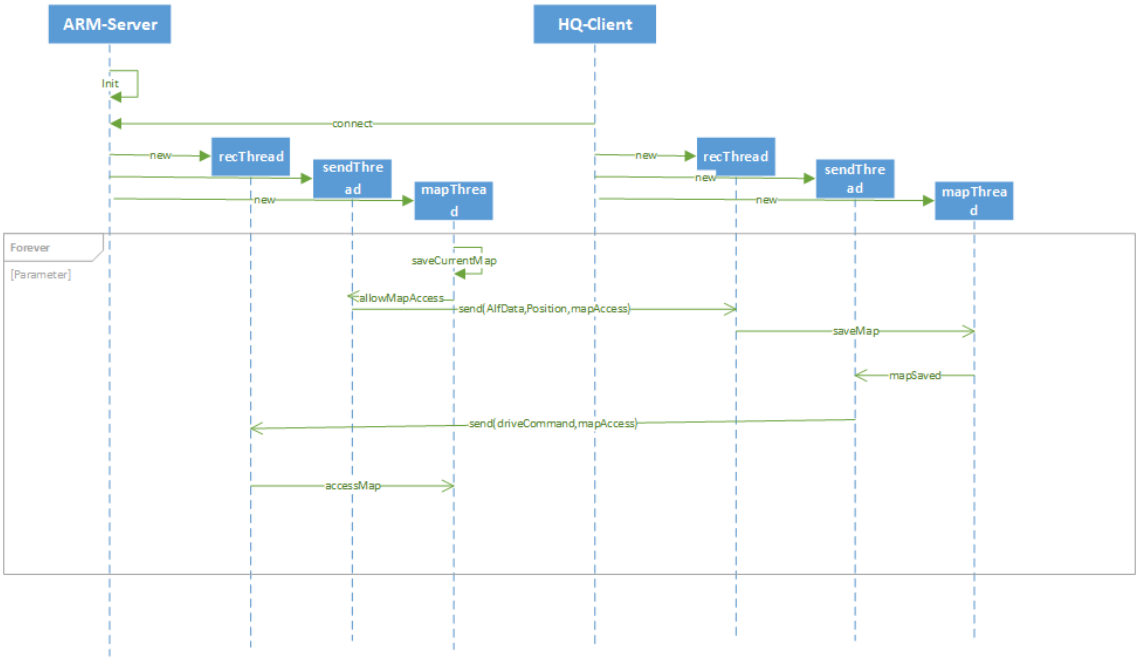

Abbildung 9 Kommunikation zwischen ARM – Prozessor und GUI

In folgender Tabelle sind die Threads mit ihren Funktionen aufgelistet.

Thread-Name	**Funktion**
recThread	Aufnahme der Daten des LIDAR-Sensors
sendThread	Senden der Sensordaten
mapThread	Verarbeitung der Sensordaten

7. Durchgeführte Tests

Innerhalb des Projekts wurden Tests für den BreezySLAM durchgeführt, um dessen Genauigkeit festzustellen. Andere Tests wiederrum waren zum Feststellen und evaluieren von Odometriewerten, welche die Genauigkeit des SLAM-Algorithmus verbessern könnten.

7.1. Qualitätstest BreezySLAM: Positionsbestimmung

7.1.1. Testgegebenheiten

In Anlehnung an verschiedene Beispielanwendungen aus dem BreezySLAM-Projekt, wurde eine Python-Anwendung zur Positionsbestimmung erstellt. Dabei wurde vor allem auf das vorhandene "urgslam" Programm zurückgegriffen, welches Lidar-Messdaten aus einer Datei ausliest und diese zusammen mit der Position des Fahrzeugs graphisch als Karte darstellt. Aufbauend auf dieser Anwendung wurde ein Programm erstellt, welches die Messwerte des Lidarsensors direkt ausließt und die Karte mit Position des Fahrzeugs graphisch darstellt.

Das Programm stellt prinzipiell die Möglichkeit bereit, die erhaltenen Messwerte durch Odometriewerten zu validieren. Da das Fahrzeug allerdings Hardwarebedingt derzeit keine zuverlässige Odometriewerte erhält, arbeitet das Programm nur mit den Werten des Lidarsensors. Eine weitere Testgegebenheit ist, dass der RMHC SLAM anstelle des Deterministic SLAM genutzt wird. Dieser wird verwendet, da er auch ohne Odometriewerte funktioniert.

7.1.2. Testdurchführung

Während der Ausführung der Python-Anwendung wird das Fahrzeug umpositioniert. Man erhofft sich, dass die Änderung der Position vom SLAM-Algorithmus richtig erkannt und dargestellt wird.

Abbildung 11 und Abbildung 10 zeigen die Startposition als auch die erkannte Positionsänderung nach einer Rechtsdrehung. Der Rote Pfeil stellt dabei die Position und Ausrichtung des Fahrzeugs dar. Weiße Flächen auf dem Bild stehen für freie Flächen im Raum, schwarze Flächen für erkannte Objekte (hier: Mauern) und graue Flächen zeigt unbekannte Umgebung.

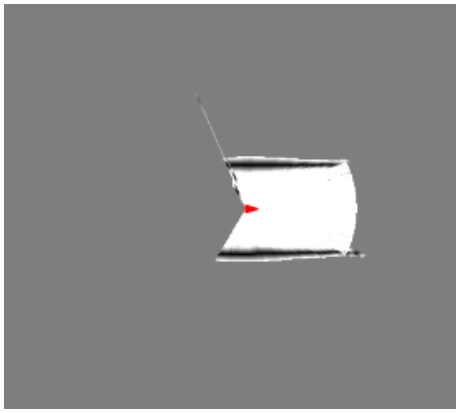

Abbildung 11 Position zu Beginn

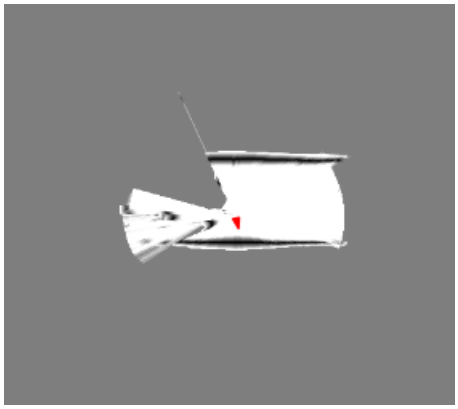

Abbildung 10 Position nach Rechtsdrehung

7.1.3. Testergebnis

Der BreezySLAM ist Imstande nur mithilfe der Lidarsensor-Werte Positionsänderungen zu erkennen. Die Position des Fahrzeugs kann bezüglich der zwei Mauern richtig eingezeichnet werden.

7.2. Qualitätstest BreezySLAM: Kartenerstellung

7.2.1. Testgegebenheiten

Ein weiterer Test stellt die Kartengenerierung mithilfe des SLAM-Algorithmus dar. Auch hierfür wurde eine eigene Anwendung erstellt, welche auf dem Beispielprogramm "log2pgm.cpp" des BreezySLAM Projekts basiert. Beim Erkundschaften der Umgebung wird die interne Karte des Algorithmus erweitert. Nach einem Tastendruck erstellt die Anwendung aus den bisherigen Kartendaten eine PGM-Datei und schließt ab. Genau wie beim vorherigen Test, werden nur die Messdaten vom Lidarsensor genutzt und der RMHC SLAM verwendet.

Abbildung 12 zeigt eine Skizze des Raumes, der für die Kartenerstellung abgelaufen wurde.

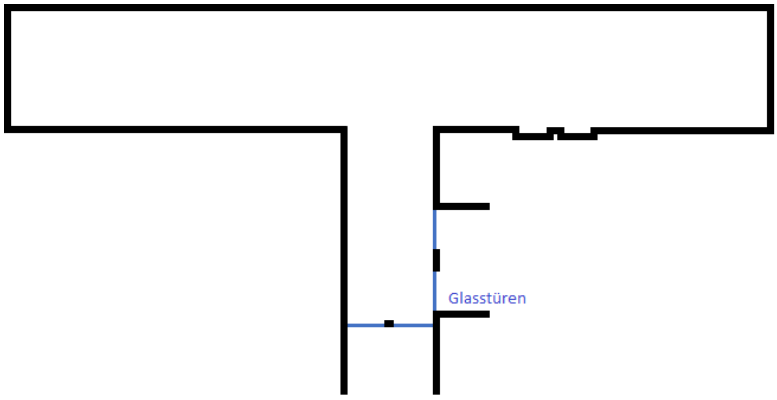

Abbildung 12: Umriss des Testraums

7.2.2. Testdurchführung

Für das Testen werden drei verschiedene Karten (als PGM-Dateien) erstellt. Für jede Karte wurde das Ablaufen der Umgebung anders gestaltet, um die Wahrnehmung des SLAM-Algorithmus in verschiedenen räumlichen Situationen zu validieren.

Abbildung 13, Abbildung 14 und Abbildung 15 zeigen, wie man den Raum durchquert hat.

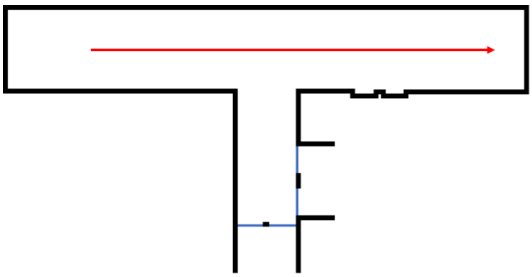

Abbildung 13: Ablaufen - Gerade Strecke

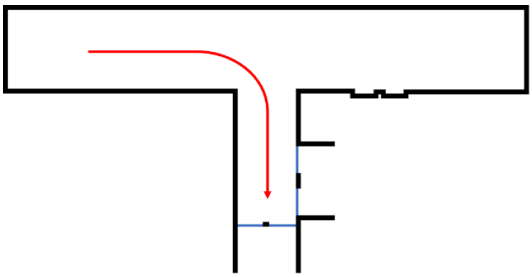

Abbildung 14: Ablaufen - Abbiegung nach rechts

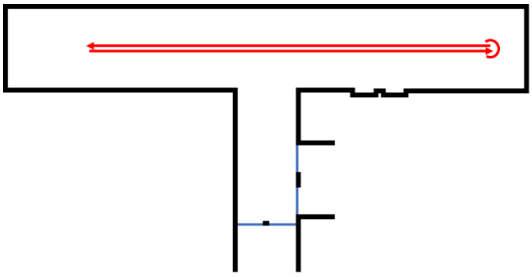

Abbildung 15: Ablaufen - Drehung auf der Stelle

7.3. Testergebnis

Aus dem Test resultierten die folgenden Karten.

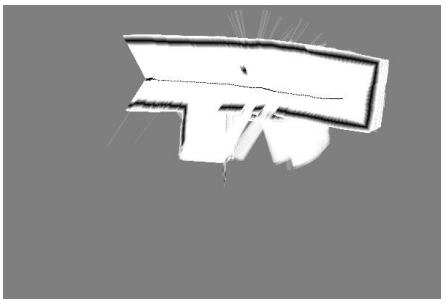

Abbildung 16: Erstellte Karte - Gerade Strecke

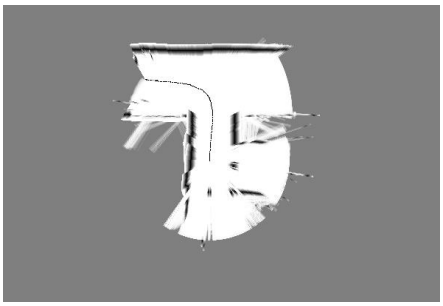

Abbildung 17: Erstellte Karte - Abbiegung nach rechts

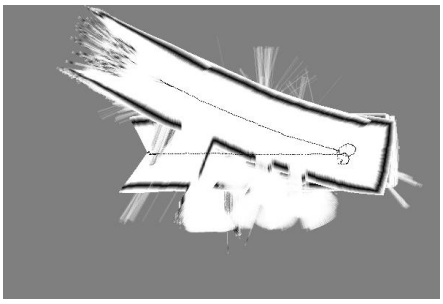

Abbildung 18: Erstellte Karte - Drehung auf der Stelle

In Abbildung 17 ist erkennbar, dass Glastüren nicht als Hindernis erkannt werden. Die Infrarotmessung durch den Lidarsensor wird vom Glas durchgelassen und nicht reflektiert. Dies führt zu einer falschen Wahrnehmung.

Ein anderes Problem erkennt man in Abbildung 18. Bei einer Rotation auf der Stelle kann es zu Orientierungsproblemen kommen. Nach einer fehlerhaften Orientierung wird bei fortgesetzter Kartenerstellung der bisher erkannte Kartenanteil durch einen falsch Gedrehten überschrieben. Für das Orientierungsproblem bei einer Rotation erhofft man sich eine Verbesserung durch das Nutzen von Odometriewerten.

7.4. Wertfeststellung: Schätzung einer mittleren Geschwindigkeit für Odometriewerte

7.4.1. Testgegebenheiten

Mithilfe eines Testaufbaus soll eine Durchschnittsgeschwindigkeit des Fahrzeugs berechnet werden. Die Geschwindigkeit soll anschließend in die zurückgelegte Distanz umgerechnet werden. Die zurückgelegte Distanz ist einer der Odometriewerte, welche das Lokalisieren im SLAM-Algorithmus verbessern kann. Insgesamt gibt es die Werte:

- Neue zurückgelegte Distanz in Millimeter
- Neue Winkelrotation in Grad
- Vergangene Zeit in Sekunden seit der letzten Odometriewerte -Aktualisierung

In diesem Test sollte evaluiert werden, ob theoretisch eine anhand der Steuerbefehle geschätzte Geschwindigkeit prototypisch für die Odometriewerte verwendet werden könnten.

Das Berechnen der durchschnittlichen Geschwindigkeit erfolgt über die Formel:

$$Geschwindigkeit\ v = \frac{zurückgelegter\ Weg\ s}{Zeitspanne\ t}$$

Der zugehörige Testaufbau ist in Abbildung 19 dargestellt. Das Fahrzeug fährt innerhalb einer fixen Zeitspanne t von 3 Sekunden. Anschließend wird der zurückgelegte Weg s gemessen. Aus diesen beiden Werten kann nun die Geschwindigkeit v berechnet werden

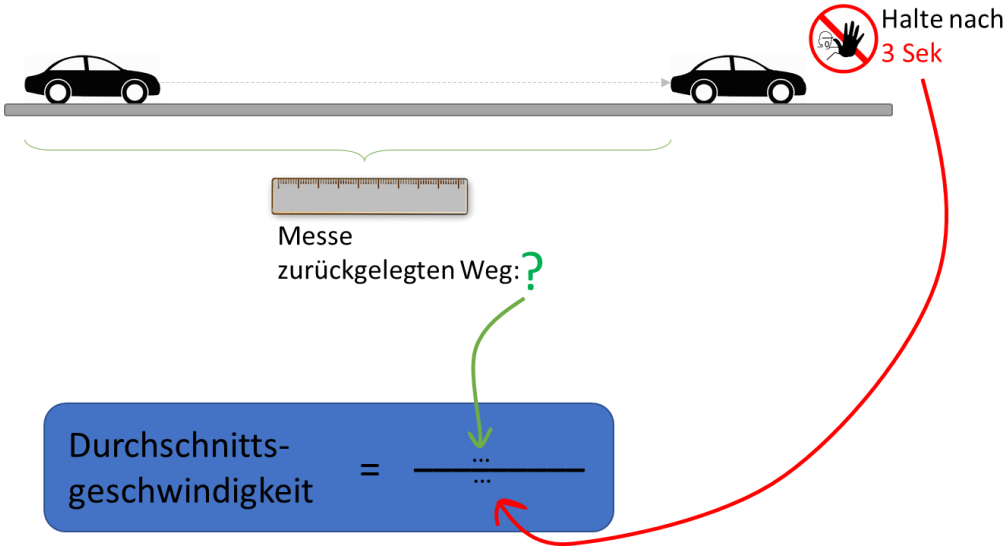

Abbildung 19: Testaufbau für Geschwindigkeitsmessung

7.4.2. Testdurchführung

Die Testdurchführung zeigte, dass innerhalb der Testspanne von 3 Sekunden das Fahrzeug seine maximale Geschwindigkeit nicht erreicht hat. Die Beschleunigungsphase ist zu lang und inkonsistent, als dass eine zuverlässige Geschwindigkeit geschätzt werden könnte.

7.4.3. Testergebnis

Der Testaufbau liefert keine gültige Durchschnittsgeschwindigkeit.

Als Alternative kann man einen Sensor verwenden. Dieser soll die tatsächliche, momentane Geschwindigkeit liefern anstelle eines Durchschnittswertes. Das Fahrzeug besitzt:

- Einen RotaryEncoder, welcher die Anzahl an Reifenumdrehungen der Hinterräder misst.
  Der RotaryEncoder konnte aufgrund eines Hardwaredefekts leider nicht getestet werden. Laut der vorherigen Gruppe sind die Werte des Sensors allerdings wenig aussagekräftig, da die Räder durchdrehen.
- Eine MPU für die Auswertung der momentanen Beschleunigung.
  Auf diese wird im Folgenden näher eingegangen

7.5. Ermitteln der Odometriewerte: Beschleunigungssensor

7.5.1. Testgegebenheiten

Der verbaute MPU Sensor liefert Beschleunigungs- und Rotationswerte in alle drei Achsen und die aktuelle Temperatur. Da man mit korrekten und genauen Beschleunigungswerten theoretisch die genaue Geschwindigkeit und Position des Fahrzeuges berechnen kann, wurde hier versucht die Sensorwerte als Odometriewerte aufzubereiten. Die Beschleunigungssensoren liefern einen 16-Bit Wert, der je nach Skalierung zwischen ±2g und ±16g Beschleunigung abdeckt. Dieser 16-Bit Wert wird im NIOS umgerechnet in g-Werte und über die Mailbox an den Armprozessor geschickt. Für die Umwandlung der Sensorwerte in Odometriewerte für unser Fahrzeug wurde ein neues Modell des Roboters (die Klasse Robot_Alf) erstellt. Darin werden die Beschleunigungswerte von g in die Einheit m/s² umgerechnet und anhand der verstrichenen Zeit seit der letzten Messung die aktuelle Geschwindigkeit und Position berechnet. Durch Messungen ergab sich ein Offset, der von der berechneten Beschleunigung abgezogen werden musste, bevor die Werte weiterverarbeitet werden konnten. Leider zeigte sich, dass sich Beschleunigungswerte in der Praxis nur sehr schlecht zur Bestimmung der Geschwindigkeit geeignet sind. Da die Beschleunigung nur die Änderung der Geschwindigkeit beschreibt, muss für die aktuelle absolute Geschwindigkeit die Geschwindigkeitsänderung und die vorherige Geschwindigkeit (V_{alt}) berücksichtigt werden. Die neue Geschwindigkeit berechnet sich also wie folgt.

$$V_{neu} = V_{alt} + \frac{1}{2} * a * t^2$$

Fehler der Messung der Beschleunigung summieren sich also immer weiter auf und können nicht ausgeglichen werden. Die Werte für die neue Geschwindigkeit beginnen also zu „driften“. Außerdem kann nicht unterschieden werden, ob sich das Fahrzeug mit konstanter Geschwindigkeit bewegt oder steht.

7.5.2. Testergebnis

Die Beschleunigungswerte des MPU Sensors eignen sich also nicht für die Berechnung von sinnvollen Odometriewerten.

8. Ausblick und Verbesserungen

Durch dieses HSP ist es gelungen einen geeigneten SLAM – Algorithmus für das Fahrzeug zu finden und diesen auf der vorhandenen Hardware zu implementieren. Dabei wurde die bestehende Software um die Funktionalität des SLAM – Algorithmus erweitert. Damit diese Erweiterung auch für den Benutzer sichtbar ist, wurde auch die graphische Oberfläche der HQ – Software um eine Anzeige der Kartendaten ergänzt. Damit ist die Grundlage für das autonome Fahren des Fahrzeugs gelegt.

Jedoch gibt es auch noch einige Dinge, die noch optimiert werden können und andere die noch nicht vorhanden sind. Der BreezySlam – Algorithmus funktioniert zwar sehr gut mit der vorhandenen Hardware, jedoch gibt es auch noch Probleme, wenn das Fahrzeug bspw. um 180° gedreht wird. Die Geschwindigkeitsmessung funktioniert nicht wie gewünscht, weshalb eine Korrektur der Karte durch diese aktuell nicht möglich ist.

Damit das Fahrzeug autonom fahren kann, muss noch eine Funktionalität implementiert werden, mit der das Fahrzeug mithilfe der Karte von einem Ort zu einem anderen fahren kann.

9. Literaturverzeichnis

[1] „Altera - NIOSII Processor,“ Intel Corporation, [Online]. Available: https://www.altera.com/products/processors/overview.html. [Zugriff am 01 10 2017].

[2] „DigiKey - Cyclone-V-FPGA-Familie,“ Digikey, [Online]. Available: https://www.digikey.de/de/product-highlight/a/altera/cyclone-v-fpga-family?WT.srch=1&mkwid=s3e2PYj5f&pcrid=77929471460&pkw=&pmt=b&pdv=c&gclid=EAIaIQobChMIocWWmsjF1gIVQr7tCh2rXw0YEAAYASAAEgLkW_D_BwE. [Zugriff am 01 10 2014].

[3] „Digikey - Entwicklungskit DE0-Nano-SoC,“ Digikey, [Online]. Available: https://www.digikey.de/de/product-highlight/t/terasic-tech/de0-nano-soc. [Zugriff am 01 10 2017].

[4] F. L. T. S. Philipp Eidenschink, „HSP Projektbericht,“ Regensburg, 2017.

[5] Hokuyo Automatic Co LTD, „Scanning Laser Range Finder URG-04LX Specifications,“ 2009.

[6] „Robotshop,“ [Online]. Available: http://www.robotshop.com/en/hokuyo-urg-04lx-ug01-scanning-laser-rangefinder.html. [Zugriff am 29. August 2017].

[7] F. L. T. S. Philipp Eidenschink, „HSP Projektbericht,“ 2016.

[8] T. Hughes, „A Framework for Localization and Navigation on a Raspberry Pi,“ 2016.

10. Anhang

Da im vorgehenden HSP keine Angaben zu Entwicklungsumgebungen und deren Einrichten vorhanden war, wurden Anleitungen zum Einrichten der verwendeten Entwicklungsumgebungen erarbeitet. Diese sind in den nachfolgenden Kapiteln enthalten. Der komplette Programmcode befindet sich auf Github unter: *https://github.com/bapoth/Garfield*.

10.1. Einrichten der IDE für die HQ – Software

Als Betriebssystem wurde Ubuntu mit Version 16.04 in einer **V**irtuellen **M**aschine (im folgenden kurz VM) verwendet.

10.1.1. Installation

sudo apt-get install qt5-default

sudo apt-get install qtcreator

oder *https://wiki.ubuntuusers.de/Qt/* aufrufen für manuellen Download.

10.1.2. Projekterstellung

Bei der Projekterstellung ist lediglich zu beachten, dass das Projekt als QT-Widget-Anwendung erstellt wird. Danach sollten alle Dateien die von QT-Creator automatisch erstellt wurden gelöscht werden. Ausgenommen hiervon ist die **.pro-Datei.**

10.1.3. Hinzufügen der Projektdateien

Anschließend können alle existierenden Projektdateien mit

Rechtsklick auf Projekt -> hinzufügen existierender Dateien hinzugefügt werden.

Folgende Dateien sollten hierbei hinzugefügt werden:

- ../HSP/Garfield-master/Software/common/ARM_HQ/<alle Dateien>
- ../HSP/Garfield-master/Software/common/ARM_NIOS_HQ/<alle Dateien>
- ../HSP/Garfield-master/Software/Software_HQ/Garfield_Control/<alle Dateien>

10.1.4. Modifizierung der .pro – Datei

Nun müssen die hinzugefügten Dateien in die .pro – Datei aufgenommen werden.

```
INCLUDEPATH += \
../HSP/Garfield-master/Software/common/ARM_HQ/ \
../HSP/Garfield-master/Software/common/ARM_NIOS_HQ/ \
../HSP/Garfield-master/Software/Software_HQ/Garfield_Control/ \
/usr/include/x86_64-linux-gnu/qt5/QtConcurrent/ \
/usr/include/x86_64-linux-gnu/qt5/QtCore/ \
/usr/include/x86_64-linux-gnu/qt5/QtGui/
CONFIG += c++11
```

10.2. Einrichten der IDE für die ARM - Software

Im Folgenden wird erklärt, wie die IDE für die ARM – Software unter einem Windows – Betriebssystem eingerichtet wurde. Da die IDE während des Projekts zur VM hinzugefügt wurde, ist das Einrichten unter Windows nicht nötig, da die VM bei der Abgabe des Projekts beigefügt ist.

10.2.1. Downloads

Zunächst sind der Download und das Installieren folgender Produkte nötig:

Java Runtime Environment (JRE):

Download – Website:

http://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html

Windows x64 Offline (62.34 MB) auswählen.

Verwendete Version: jre-8u144-windows-x64.exe

IDE: Eclipse Neon:

Download – Website:

http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/neon/3/eclipse-cpp-neon-3-win32-x86_64.zip

Make – Tool:

Download – Website:

http://gnuwin32.sourceforge.net/packages/make.htm

Version laut Website *„Complete package, except sources“* (3384653 Byte).

ARM-Compiler:

Download – Website:

https://releases.linaro.org/components/toolchain/binaries/5.3-2016.02/arm-linux-gnueabihf/

gcc-linaro-5.3-2016.02-i686-mingw32_arm-linux-gnueabihf.tar.xz (173.5 MB) auswählen.

Hierbei ist zu beachten, dass mit der neusten Version des Compilers Fehler auftraten. Daher sollte obige Version verwendet werden.

10.2.2. Installation

1. Entpacken des ARM – Compilers nach *C:\bin.*
2. Ausführen von make-3.8.1.exe

10.2.3. Erweiterung der PATH – Variablen

Für die Verwendung von Eclipse muss die PATH – Variable erweitert werden. Falls alle benötigten Resourcen unter Windows auf Laufwerk C:\ installiert wurden, sollten die Pfade wie folgt aussehen:

- *C:\Program Files (x86)\Java\jre1.8.0_144\bin*
- *C:\Program Files (x86)\GnuWin32\bin*
- *C:\bin\linaro_Older\gcc\gcc\bin*

Diese Pfade werden zu den PATH – Variablen von Windows hinzugefügt.

10.2.4. Einrichten von Eclipse

Im Folgenden wird beschrieben wie die Projektdateien als Links in Eclipse eingebunden werden.

Hinzufügen folgender Dateien als Links:

- <project-path>*Software**Software_ARM**Comm_Gateway*\<*alle Dateien*>
- <project-path>*Software**common**ARM_HQ*\<*alle Dateien*>
- <project-path>*Software**common**ARM_NIOS*\<*alle Dateien außer using_shared_memory_example.cpp* >
- <project-path>*Software**common**ARM_NIOS_HQ*\<*alle Dateien*>

Einbinden des ARM – Compilers:

Project→Properties→C/C++-Build→Settings→Cross GCC Compiler: arm-linux-gnueabihf-gcc

${COMMAND} ${FLAGS} -std=c++11 -lpthread ${OUTPUT_FLAG} ${OUTPUT_PREFIX}${OUTPUT} ${INPUTS}

Project→Properties→C/C++-Build→Settings→Cross G++ Compiler: arm-linux-gnueabihf-g++

${COMMAND} ${FLAGS} -std=c++11 -lpthread ${OUTPUT_FLAG} ${OUTPUT_PREFIX}${OUTPUT} ${INPUTS}

Project→Properties→C/C++-Build→Settings→Cross G++ Linker: arm-linux-gnueabihf-g++

${COMMAND} ${FLAGS} -std=c++11 -lpthread ${OUTPUT_FLAG} ${OUTPUT_PREFIX}${OUTPUT} ${INPUTS}

Hinzufügen von IncludePath für Dateien (durch Angabe von FileSystem):

Project→Properties→C/C++-Build→Settings→Cross G++ Compiler→Includes*:*

- <project-path>*Software**common**ARM_HQ*
- <project-path>*Software**common**ARM_NIOS*
- <project-path>*Software**common**ARM_NIOS_HQ*

10.2.5. Zusätzliches Einrichten von Eclipse zur Verwendung der BreezySLAM - Library

Zunächst muss in Eclipse ein neues C++ - Projekt erzeugt werden:

File→New→C++ - Project

Project type: Makefile Project→Empty Project

Toolchains: Cross Gcc.

Anschließend werden folgende Files als Links hinzugefügt:

- <project-path>*Software**Software_ARM**Comm_Gateway**BreezySLAM*

Danach wird eine Datei names *Makefile* auf oberster Hierarchie in der Eclipse Projektebene erstellt. Eventuell muss hier der Pfad (../BreezySLAM/cpp und ../BreezySLAM/c) angepasst werden.

Nun muss der Befehl *Build Project* ausgeführt werden.

Im eigentlichen Eclipse-Project für die ARM - Software muss der Code für die erstellte Library/.so – Datei erweitert werden.

Include Library unter

Project→Properties→C/C++-Build→Settings→Cross G++ Linker→Libraries

mit:

- Libraries: breezyslam (= Name von erstellter Library, wobei "lib" am Anfang & ".so" am Ende weggelassen werden)

- Library search path: "<project-path>\Software\Software_ARM\eclipse_workbench_arm\BreezySLAMLibOwnMakefile" (=Pfad von erstellter Library)

- Hinzufügen von IncludePath für Dateien (durch Angabe von FileSystem):

Project→Properties→C/C++-Build→Settings→Cross G++ Compiler→Includes:

- *"*<project-path>*Software**Software_ARM**Comm_Gateway**BreezySLAM**c"*
- *"*<project-path>*Software**Software_ARM**Comm_Gateway**BreezySLAM**cpp"*

Hinzufügen von Files als Links:

- <project-path>*Software**Software_ARM**Comm_Gateway**BreezySLAM**cpp**Robot_Alf.hpp*
- <project-path>*Software**Software_ARM**Comm_Gateway**BreezySLAM**cpp**Robot_Alf.cpp*

Hinweis: Kompilierte Projektdatei als auch .so-File(=Library) müssen beide auf das SoC – System auf dem Fahrzeug übertragen werden

Projektdatei in den Ordner */home/ubuntu/bin/*

.so-Datei in den Ordner */usr/local/lib* → dann ausführen "ldconfig".

LIBDIR muss den Pfad */usr/local/lib* beinhalten

10.3. Einrichten des FPGA

Als erstes muss QuartusPrime (LiteEdition) gestartet werden.

Unter *Tools → Qsys* kann Qsys gestartet werden. Nun sollte <*project-path*>\FPGA_Design\Garfield_Design\Garfield_system.qsys geöffnet mit Qsys geöffnet werden.

Im geöffneten Qsys Projekt soll mit *Tools->Options->IP Search Path Options* die Pfade *<project-path>\FPGA_Design/ip_intern* und *<project-path>\FPGA_Design/ip_extern* hinzugefügt werden.

Zum synthetisieren muss als HDL-Design VHDL ausgewählt sein. Synthetisiert wird mit *Generate->Generate HDL* und anschließend *Start Analysis & Synthesis*. Als Ergebnis werden die HDL Design Dateien erzeugt. Das Qsys Projekt kann nun geschlossen werden.

In der IDE QuartusPrime öffnet man nun das Projekt *<project-path>\FPGA_Design\Garfield_Design Garfield.qpf*. Dieses wird mit dem Aufruf von *"Compile Design"* (im Task Fenster) kompiliert.

Starte nun das Tool *Programmer* unter *Tools->Programmer,* welches die Datei zum Programmieren des FPGA erstellt. Im Programmer klicke auf den Button *"Hardware Setup"* und wähle den passenden Wert für *"currently selected hardware"*. Durch den *"Auto Detect"* Button suche automatisch die passende Software. Achte darauf bei der Verwendung des „*Auto Detect*“ Features den zweiten Eintrag auszuwählen. Nun drücke den „*Start*“ Button.

Unter *Tools->NIOS II Software Build Tools for Eclipse* kann das NIOS Eclipse gestartet werden. Innerhalb dieser IDE kann man ein neues Projekt mit *File->New->NIOS II Application and BSP from Template* generieren. Hierbei sollte die Datei *<project-path>\FPGA_Design\Garfield_Design\Garfield_system.sopcinfo* angegeben werden.

Folgende Ordner können im Anschluss eingefügt werden:

- Software/Software_NIOS2/driver
- Software/Software_NIOS2/os
- Software/Software_NIOS2/tasks
- Software/common/ARM_NIOS
- Software/common/ARM_NIOS_HQ

Außerdem sollten im Properties Fenster unter NIOS II Application Properties->NIOS II Application Path der Application Include Directory mit den folgenden Pfaden erweitert werden:

- Software\Software_NIOS2\os

- Software\Software_NIOS2\os/Source
- Software**/**Software_NIOS2/os/Source/include
- Software/Software_NIOS2/os/Source/portable
- Software/Software_NIOS2/driver
- Software/Software_NIOS2/task
- Software/common/ARM_NIOS
- Software/common/ARM_NIOS_HQ
- ../<projectname>_bsp
- ../<projectname>_bsp/drivers/inc
- ../<projectname>_bsp/drivers/src

Die Datei *common/ARM_NIOS/using_shared_memory_example.cpp* wird nicht benötigt und muss gelöscht werden. Ansonsten kommt es während der Kompilierung zu einem Fehler.

Zuletzt müssen noch einige Anpassungen gemacht werden, damit die Kompilierung fehlerfreu erfolgen kann.

In der Datei <projectname>_bsp/system.h muss die Zeile

#define ALT_ENHANCED_INTERRUPT_API_PRESENT

ersetzt werden durch folgende:

#define ALT_LEGACY_INTERRUPT_API_PRESENT

Anschließend mit Rebuild <projectname>_bsp neu erstellen.

Außerdem muss das Makefile unter <projectname>/Makefile für die Unterstützung von c++11 modifiziert werden:

Arguments only for the C++ compiler.

APP_CXXFLAGS := $(ALT_CXXFLAGS) $(CXXFLAGS)

Wird ersetzt mit:

Arguments only for the C++ compiler.

APP_CXXFLAGS := $(ALT_CXXFLAGS) $(CXXFLAGS) -std=c++11

Um den C-Code auf die Hardware zu übertragen müssen folgende Schritte ausgeführt werden:

Auf Unter Run->Run Configurations.. in den Tab Target Connection navigieren

1. Refresh Connection drücken
2. Aktivieren falls nötig:
   "Ignore mismatched systemID" und *"Ignore mismatched system timestamp"*
3. *Apply*
4. *Run*

10.4. Inbetriebnahme des Fahrzeugs

Rechner mit Netzwerk des Fahrzeugs verbinden und folgende Einstellungen innerhalb von Ubuntu 16.04-64 Bit vornehmen:

1. SSH-Verbindung mit Putty starten:

   - IP: 192.168.100.149

   - Port: 22

2. Im Terminal der SSH-Verbindung/Putty:

   Einloggen mit:

   - Login: *ubuntu*

   - Passwort: *temppwd*

3. Ausführen von:

   sudo chmod 777 /dev/ttyACM0 Für Verwendung des Lidarsensors

   sudo -s

   ./startup im Home-Verzeichnis

   bin/HSP_ARM_Gateway im Home-Verzeichnis

4. Starte GUI zur Steuerung des Fahrzeugs

sudo *./Garfield_Control*

(Shell-Skript ist unter Software/Software_HQ/Garfield_Control_Standalone.zip)

5. Einstellungen:

   - IP: *192.168.100.149*

   - Port: *6667 (oder: 6666)*