Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterthesis

Mario Wegner

Konzeption und Entwicklung eines hochpräzisen Systems zur Positionsbestimmung von Fahrzeugen in urbanen Gebieten

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Mario Wegner

Konzeption und Entwicklung eines hochpräzisen Systems zur Positionsbestimmung von Fahrzeugen in urbanen Gebieten

Masterthesis eingereicht im Rahmen der Masterprüfung
im Masterstudiengang Informations- und Kommunikationstechnik
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Rasmus Rettig
Zweitgutachter : Prof. Dr. -Ing. Karin Landenfeld

Abgegeben am 10. April 2017

Mario Wegner

Thema der Masterthesis

Konzeption und Entwicklung eines hochpräzisen Systems zur Positionsbestimmung von Fahrzeugen in urbanen Gebieten

Stichworte

Positionsbestimmung, Laser, Ultraschall, autonomes Fahren

Kurzzusammenfassung

Diese Arbeit umfasst die Konzeption und Entwicklung eines Sensorsystems für die Positionsbestimmung in urbanen Gebieten. Dies wird durch die Vermessung von Gebäudefassaden realisiert. Die Arbeit beschreibt die Entwicklung eines Konzepts bis hin zum Aufbau eines Prototypen und eine Applikation an eine Testfahrzeug.

Mario Wegner

Title of the paper

Conception and Development of a high-precision system for vehicle positioning in urban areas

Keywords

positioning, laser, ultrasonic, autonomous driving

Abstract

This report decribes the development and construction of a sensor system for positioning in urban areas. This is done by measureing of urban canyons. Part of this work is the creation of a concept and the set-up of a prototype and the application to a test car-

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich bei der Erstellung der Abschlussarbeit unterstützt und motiviert haben. Besonderer Dank gilt hierbei Prof. Dr. rer. nat. Rasmus Rettig für die intensive Betreuung. Außerdem möchte ich mich bei Prof. Dr. Ing. Karin Landenfeld für die Übernahme des Zweitgutachtens bedanken.

Ebenfalls möchte ich mich bei alle bedanken, die diese Arbeit Korrektur gelesen haben und mir somit eine große Hilfe bei der Fertigstellung waren.

Inhaltsverzeichnis

Abkürzungsverzeichnis

ASCII American Standard Code for Information Interchange

BASt Bundesanstalt für Straßenwesen

CAN Controller Area Network

CEP Circular Error Propble

CSI Camera Serial Interface

DIN Deusche Industrie Norm

DHCP Dynamic Host Configuration Protocol

DSI Display Serial Interface

FMCW Frequency Modulatd Coninous Wave

FSK Frequency-Shift-Keying

FTP File Transfer Protocol

GNSS Global Navigation Satellite System

GPIO General Purpose Input/Output

GPS Global Positioning System

GSM Global System for Mobile Communications

HAW Hochschule für Angewandte Wissenschaften

HDMI High Definition Multimedia Interface

I^2C Inter-Integrated Circuit

ISR Interrupt Service Routine

IP Internet Protocol

LAN Local Area Network

LIDAR Light Detection And Ranging

OBDII On-Board-Diagnose II

PKW Personenkraftwagen

LRR Long Range Radar

PoE Power over Ethernet

PTP Precision Time Protocol

QR Quick Response

RADAR Radio Detection And Ranging

SLAM Simultaneous Localization and Mapping

SSH Secure Shell

SSID Service Set Identifier

SPI Serial Peripheral Interface

SRR Short Range Radar

TCP Transmission Control

UART Universal Asynchronous Receiver Transmitter

USB Universal Serial Bus

1. Einführung

Der Bereich des autonomen Fahrens im öffentlichen Straßenverkehr hat in den letzten Jahren große Fortschritte vorweisen können. Automobilhersteller bieten bereits heute erste autonome Lösungen für Fahrzeuge an [19]. Doch autonomes Fahren ist nicht gleich autonomes Fahren, es kann auf viele verschiedene Weisen realisiert werden. Zur Differenzierung hat die Bundesanstalt für Straßenwesen (BASt) den Automatisierungsgrad in fünf Stufen unterteilt und definiert [10]. Die fünf Grade der Automatisierung sind in Abbildung 1.1 dargestellt.

Nomenklatur	Fahraufgaben des Fahrers nach Automatisierungsgrad
Vollautomatisiert	Das System übernimmt Quer- und Längsführung vollständig in einem definierten Anwendungsfall • Der Fahrer muss das System dabei nicht überwachen • Vor dem Verlassen des Anwendungsfalles fordert das System den Fahrer mit ausreichender Zeitreserve zur Übernahme der Fahraufgabe auf • Erfolg dies nicht, wird in den risikominimalen Systemzustand zurückgeführt • Systemgrenzen werden alle vom System erkannt, das System ist in allen Situationen in der Lage, in den risikominimalen Systemzustand zurückzuführen
Hochautomatisiert	Das System übernimmt Quer- und Längsführung für einen gewissen Zeitraum in spezifischen Situationen • Der Fahrer muss das System dabei nicht überwachen • Bei Bedarf wir der Fahrer zur Übernahme des Fahraufgabe mit ausreichender Zeitreserve aufgefordert • Systemgrenzen werden alle vom System erkannt. Das System ist nicht in der Lage, aus jeder Ausgangssituation den risikominimalen Zustand herbeizuführen
Teilautomatisiert	Das System übernimmt Quer- und Längsführung (für einen gewissen Zeitraum oder/und in spezifischen Situationen) • Der Fahrer muss das System dauerhaft überwachen • Der Fahrer muss jederzeit zur vollständigen Übernahme der Fahrzeugführung bereit sein
Assistiert	Fahrer führt dauerhaft entweder die Quer- oder die Längsführung aus. Die jeweils andere Fahraufgabe wird in gewissen Grenzen vom System ausgeführt • Der Fahrer muss das System dauerhaft überwachen • Der Fahrer muss jederzeit zur vollständigen Übernahme der Fahrzeugführung bereit sein
Driver only	Fahrer führt dauerhaft (während der gesamten Fahrt) die Längsführung (Beschleunigen/Verzögern) und die Querführung (lenken) aus

Automatisierungsgrad

Abbildung 1.1.: Grade der Automatisieung und ihre Definition (BASt) (Auszug: [10])

So findet in der niedrigsten Stufe *Driver only* keine autonome Fahrerunterstützung statt. Es folgen erste einfache Unterstützungsmaßnahmen in der Stufe *Assistiert*, hierzu zählt beispielsweise Abstandsregeltempomaten, ein Eingriff in die Lenkung findet nicht statt. Der Fahrer besitzt zu jeder Zeit die volle Kontrolle über die Fahrtrichtung des Fahrzeuges. Ab der Stufe *Teilautomatisiert* übernimmt das Fahrzeug kurzzeitig auch die Lenkung. Spurhaltefunktionen oder automatisches Einparken können dieser Stufe zugeordnet werden. Somit sind die Funktionen dieser Stufen bereits serienreif und werden von nahezu jedem Automobilhersteller angeboten. Bis hierhin hat der Führer des Fahrzeuges zu jeder Zeit die volle Kontrolle und damit auch die Verantwortung für das Fahrzeug.

Ab den folgenden Graden *Hochautomatisiert* und *Vollautomatisiert* bewegt sich die Verantwortung in Richtung des Fahrzeuges, der Fahrer ist nicht mehr verpflichtet, dass System zu überwachen. Dies markiert eine wichtige Grenze, da ab hier auch die Navigation, also Wegfindung, vom Fahrzeug übernommen wird. Es muss also die eigene Position kennen. Während sich ein menschlicher Fahrer anhand von Straßenschildern oder anderen markanten Punkten orientiert, ist die autonome Positionsbestimmung nicht trivial. Hierfür wird in der Regel ein Global Navigation Satellite System (GNSS), wie zum Beispiel GPS, verwendet. Diese Systeme führen eine Positionsbestimmung durch Messung von Signallaufzeiten zwischen einem Empfänger und Satelliten durch. Grundsätzlich sind diese Systeme sehr robust und in der Lage, an jedem Punkt der Erde eine Positionsbestimmung zu ermöglichen. Es gibt allerdings einen großen Nachteil, gerade in dicht besiedelten, urbanen Umgebungen, also dort, wo viele Fahrzeuge unterwegs sind, treten große systembedingte Fehler auf. In Innenstädten bilden hohe Gebäude sogenannte *Urban Canyons*, die einen direkten Sichtkontakt zu den Satelliten verhindern. Die Signale gelangen so durch Reflexionen an den Gebäuden nur noch über Umwege zum Empfänger. Da die Positionsbestimmung aus den Laufzeiten der Signale berechnet wird, führt dies zu Fehlpositionierungen [7]. Abbildung 1.2 zeigt hierfür ein Beispiel.

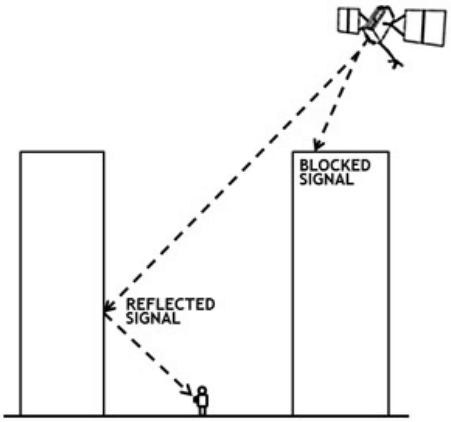

Abbildung 1.2.: GNSS *Multipath*-Fehler in *Urban Canyons* (Auszug: [7])

Damit ist eine für das autonome Fahren benötigte Genauigkeit durch Verwendung von GNSS in Innenstädten nicht gegeben. Auf offenem Gelände hingegen reicht die Genauigkeit aus, erste Automobilhersteller bieten bereits ersten Fahrfunktionen des Grades *Hochautomatisiert* an [19]. Die autonome Fahrfunktion ist hier allerdings nur auf Autobahnen verfügbar. Damit ein autonomes Fahren auch in urbanen Gebieten möglich ist, wird für diesen Bereich eine alternative Positionsbestimmung benötigt.

Im Rahmen dieser Arbeit soll ein System zur Positionsbestimmung in urbanen Gebieten konzeptioniert und entwickelt werden, welches in urbanen Gebieten als Alternative für ein GNSS dienen soll. Die Arbeit wird im *Urban Mobility Lab*, gefördert von der Fakultät Technik und Informatik, durchgeführt und umfasst den Entwicklungsprozess eines Systems zur Positionsbestimmung. Zunächst wird im Kapitel *Stand der Technik* ein Überblick über kartenbasierte Positionsbestimmungsverfahren und Sensoren für eine Umgebungsvermessung gegeben. Danach wird, orientiert am V-Modell zunächst die Anforderungsanalyse durchgeführt. Mit diesen Ergebnissen wird dann im Systementwurf ein Konzept für die Positionsbestimmung entwickelt und die Systemarchitektur definiert. In der Realisierung erfolgt dann der Aufbau und die Programmierung eines ersten Prototypen, welcher im Kapitel *Test und Bewertung* beginnend bei den Systemkomponenten bis hin zu einer Testfahrt unter realen Bedingungen getestet und bewertet wird. Die Arbeit endet dann mit einem Ausblick und einem Fazit.

2. Stand der Technik

2.1. Kartenbasierte Positionsbestimmung

Unter Positionsbestimmung versteht man die Ermittlung der Position eines Objektes in einem definierten, beliebigen Koordinatensystem. Die Position ist ein Teil des Objektstandortes, welcher noch weitere Informationen, wie Ausrichtung oder Geschwindigkeit enthalten kann. Ein Standort kann in einem zweidimensionalen, kartesischen System beispielsweise als

$$\vec{x_t} = (x, y, \theta)^T \tag{2.1}$$

dargestellt werden. Wobei $\vec{x_t}$ für den Standort zum Zeitpunkt t, x und y für die kartesischen Koordinaten und θ für die Ausrichtung des Objektes steht. Ein autonomes Objekt hat in der Regel keine Möglichkeit den eigenen Standort auf direkte Weise zu bestimmen. Dies geschieht durch Messungen in der Umgebung, aus denen dann eine Position geschätzt wird. Es gilt also eine Messung $\vec{z_t}$ in eine Position $\vec{x_t}$ zu überführen. Bei den Messungen handelt es sich um einen mehrdimensionalen Vektor, der sämtliche Messdaten aller Sensoren des Objektes enthält. Oft handelt es sich hierbei um Distanzmessungen zu Objekten in der Umgebung. Eine Positionsbestimmung erfordert nun zusätzliche Informationen über die Umgebung, welche in einer Karte m enthalten sind. Weitere Methoden, zum Beispiel durch Verwendung von Bewegungsdaten (z. B. Beschleunigungen), sind ebenfalls möglich. Aber wegen des im Laufe der Zeit immer größer werdenden Integrationsfehlers sind diese Methoden keine Alternative zu einer Positionsbestimmung anhand der Umgebung. Die Beschreibung der Umgebung kann durch verschiedene Methoden erfolgen, welche dann auch gleichzeitig die Methode zur Lokalisierung definieren. Die Genauigkeit der Karte bestimmt somit auch die Genauigkeit der Positionsbestimmung. Ein Nachteil hierbei ist, dass das Erstellen einer Karte mit einem hohen Aufwand verbunden ist. Dies kann durch ein Simultaneous Localization and Mapping (SLAM) Verfahren gelöst werden. Hier wird während der Fahrt eine Karte erzeugt, die dann gleichzeitig zur eigenen Lokalisierung verwendet wird. Dies hat den Vorteil, dass keine vorausgehende Kartenerstellung notwendig ist und dynamische Elemente auch während des Betriebs in der Karte gespeichert werden können. SLAM kann auf beliebige Arten implementiert werden. Es gibt Lösungen für alle kartenbasierten Positionsbestimmungsverfahren [11] [5] [3]. Für die Realisierung wird das System, bestehend aus der Karte m, den Messungen $\vec{z_t}$ und der eigenen Position $\vec{x_t}$ um die relative Bewegung des

autonomen Objekts $\vec{u}_t$ erweitert. Hierbei handelt es sich in der Regel um Odometriedaten, die während der Bewegung aufgenommen werden. Es kann aber andere interne Sensorik verwendet werden.

In den folgenden Abschnitten sollen die aktuell verwendeten Methoden für die Umgebungsbeschreibung und die dazugehörige Methode zur Positionsbestimmung beschrieben werden. Als Grundlage hierfür dient das *Springer Handbook of Robotics* [30] und die Literatur von Sebastian Thurn [32].

2.1.1. Belegungs-Rasterkarten (Occupancy Grid Maps)

Das Verfahren des Kartenabgleichs oder auch *Map Matching* genannt nutzt eine positionsbasierte Karte (*grid map*) zur Darstellung der Umgebung.

Eine *Occupancy Grid Map* m besteht aus einer endlichen Menge von Kartenobjekten m_n mit bestimmten Eigenschaften. Bei einer positionsbasierten Karte steht jedes Kartenobjekt für eine Position in einem Koordinatensystem. Handelt es sich hierbei zum Beispiel um ein zweidimensionales kartesisches System, werden die Kartenobjekte statt m_n mit $m_{x,y}$ gekennzeichnet, um die Darstellung intuitiver zu gestalten. Jedes Kartenobjekt kann nun beliebig viele Informationen enthalten, welche die Umgebung beschreiben. Diese sind abhängig von den durchgeführten Messungen $\vec{z}_t$. Bei Distanzmessungen handelt es sich oft um eine Wahrscheinlichkeit, die angibt, ob diese Position befahrbar ist oder nicht. Bei visuellen Messverfahren kann ein Kartenobjekt außerdem einen Farbwert enthalten. Der Vorteil dieser Karten liegt darin, dass sie in jeder Umgebung eingesetzt werden können, leicht zu interpretieren und zu verarbeiten sind. Ein Nachteil ist der bei der Kartierung entstehende Diskretiesierungsfehler. Durch die Wahl der Rastergröße kann eine Umgebung zwar beliebig genau dargestellt werden, dies führt allerdings einem sehr hohen Speicherbedarf.

Die Positionsbestimmung auf diesen Karten wird durch *Map Matching* realisiert und beinhaltet grundsätzlich zwei Karten. Die globale Karte m enthält alle Informationen der gesamten Umgebung, in der die Positionsbestimmung durchgeführt werden soll. Die zweite Karte m_{local} enthält die direkte Umgebung des Objektes, welche durch eine aktuelle Messung $\vec{z}_t$ erstellt wurde und ist damit kleiner als die Karte m. Es gilt also

$$f(\vec{z}_t) = m_{local} \tag{2.2}$$

Beide Karten werden nun miteinander verglichen und der Ort der maximalen Ähnlichkeit gesucht. Dies kann durch Korrelation oder andere Verfahren durchgeführt werden. Da die lokale Karte im Koordinatensystem des Objektes aufgenommen wurde, ist es zuvor notwendig sie ins Koordinatensystem der globalen Karte zu transformieren. Die wahrscheinlichste Position ist dann der Punkt, an dem lokale und globale Karte die höchste Ähnlichkeit aufweisen.

2.1.2. Topologische Karten (Topoligical Maps)

Während die zuvor beschriebene Umgebungsdarstellung hauptsächlich die Struktur der Umgebung beachtet, basieren topologische Karten auf einem anderen Prinzip. Eine Karte besteht hier aus einer Graphenstruktur, bestehend aus Knoten (*Notes*) verbunden durch Bewegungslinien (*travel edges*) auf denen sich ein mobiles Objekt bewegen kann. Knoten sind durch die Umgebungssensorik wiedererkennbare Positionen. Diese sind definiert durch die Distanz zu Objekten oder Signaturen in der Umgebung. Während der Bewegung von einem Knoten zum Anderen findet hier keine Lokalisierung statt. Daher werden topologische Karten größtenteils in sehr strukturierten Umgebungen, wie zum Beispiel innerhalb von Gebäuden oder urbanen Gebieten verwendet. Abbildung 2.1 zeigt ein Beispiel für eine topologische Karte in einer strukturierten Umgebung. Es handelt sich hierbei um das Innere eines Gebäudes. Die Knoten u_1 bis u_9 stehen für wiedererkennbare Positionen und die Linien c_{nm} entsprechen den Bewegungslinien.

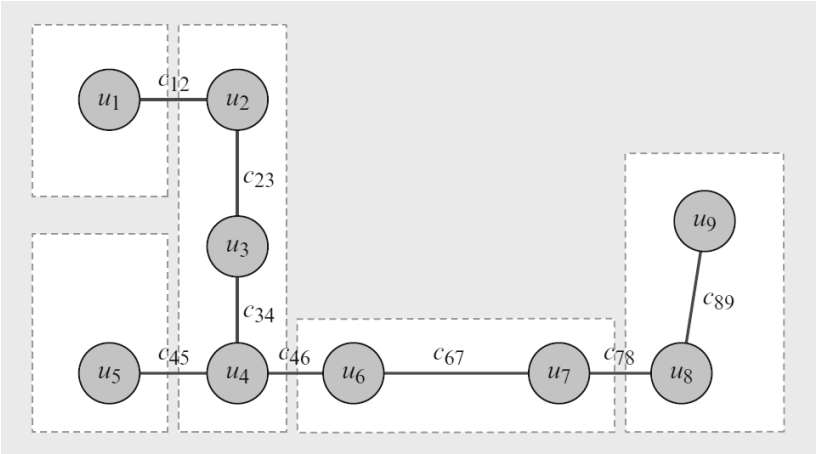

Abbildung 2.1.: Topologische Karte einer strukturierten Umgebung (Auszug: 'Springer Handbook of Robotics' Abb. 5.19)

Die Bewegung eines Objektes muss für die Nutzung dieser Karten nach einem festen Algorithmus erfolgen, in Gebäuden kann dies zum Beispiel eine Wandverfolgung und in urbanen Gebieten die Verfolgung der Straße sein. Die Karte muss sicherstellen, dass bei Verwendung des Bewegungsalgorithmus immer ein Knoten erreicht werden kann.

Der Vorteil dieser Karte ist der wesentlich geringere Speicherbedarf im Vergleich zur *Occupancy Grid Map* und auch der Diskretisierungsfehler tritt nur noch bedingt auf, da die Position der Knoten beliebig genau angegeben werden kann. Allerdings sind diese Vorteile durch eine geringere Flexibilität für Umgebungen aufgewogen, denn diese Karte kann nur

in sehr strukturierten Umgebungen eingesetzt werden, in denen die Bewegungsfreiheit eingeschränkt ist. Außerdem muss die Umgebung gleichzeitig über ausreichend viel markante Orte verfügen, um dort Knoten zu definieren.

2.1.3. Merkmalsbasierte Karten (Fature-Based Maps)

Verfügt eine Umgebung wiedererkennbare Objekte, so können merkmalsbasierte Karten verwendet werden. Hierbei besteht die Karte m, wie bei der *Occupancy Grid Map*, ebenfalls aus einer Menge von Kartenobjekten. Ein Kartenobjekt steht allerdings nicht für eine Position in einem Koordinatensystem, sondern für ein vom autonomen Objekt identifizierbares Merkmal in der Umgebung. Während das *Map Matching* die Rohdaten von Sensoren verwendet, erfolgt bei merkmalsbasierter Positionsbestimmung ein höher integrierter Ansatz. Hierbei besteht die Karte aus einer Reihe von physikalischen Merkmalen m_i (*landmarks*), die aus einer Signatur s_i und der Position x_i im Koordinatensystem bestehen. Die Positionserkennung erfolgt dann über die Identifizierung der Signatur und der relativen Positionsbestimmung zum Merkmal. Da die Position des Merkmals bekannt ist, kann auf die eigene Position geschlossen werden. Hierbei können zwei Fälle unterschieden werden.

Mehrdeutige Signaturen

Bei mehrdeutigen Signaturen besitzt die Karte mehrere Merkmale mit gleicher Signatur, die an verschiedenen Punkten in der Umgebung vorkommen. Dies kommt bei Signaturen vor, die Umrisse und das grobe Aussehen vom Objekten beschreiben. Hierzu zählen beispielsweise gerade Wände, Ecken, runde Objekte oder andere geometrische Formen, welche in der Umgebung mehrfach vorkommen können. Die Signatur kann hierbei beliebig genau beschrieben werden und zusätzliche Informationen wie Höhe oder Breite beinhalten. Ein autonomes Objekt muss in der Lage sein, diese Signaturen zu erkennen und sich relativ dazu zu lokalisieren. Dies erfolgt über die Bestimmung der Distanz und idealerweise auch der Orientierung zum Merkmal. Es gilt die Messung $\hat{z}_t$ in Merkmale f_t^i zu überführen.

$$f(\vec{z}_t) = \{ \begin{pmatrix} r_t^1 \\ \Phi_t^1 \\ s_t^1 \end{pmatrix}, \begin{pmatrix} r_t^2 \\ \Phi_t^2 \\ s_t^2 \end{pmatrix}, \ldots \} \tag{2.3}$$

Wobei r_t^i für die Distanz, Φ_t^i für den Winkel des Objekts steht, sowie s_t^i das erkannte Merkmal beschreibt. Da die Karte des autonomen Objekts mehrere Merkmale der gleichen Signatur an unterschiedlichen Positionen enthält, müssen nun unter der Rücksichtnahme aller

sichtbaren Merkmale die Kartenobjekte identifiziert werden. Hierbei wird davon ausgegangen, dass die Konstellation der sichtbaren Merkmale in der Karte einzigartig ist. Die eigene Position kann durch geometrische Berechnung erfolgen.

Eindeutige Signaturen

Bei Kartensystemen mit eindeutigen Signaturen enthält die Karte jede Signatur nur einmal. Daher entfällt hier die Identifizierung des Merkmals und die geometrische Positionsbestimmung kann direkt erfolgen. Abbildung 2.2 zeigt hierfür ein Beispiel, das autonome Objekt erkennt bei einer Messung drei Merkmale mit den Signaturen s_t^1, s_t^2 und s_t^3 und bestimmt die Distanzen r_t^i. Da die Positionen der Merkmale in der Karte enthalten sind, kann die eigene Position x_t bestimmt werden.

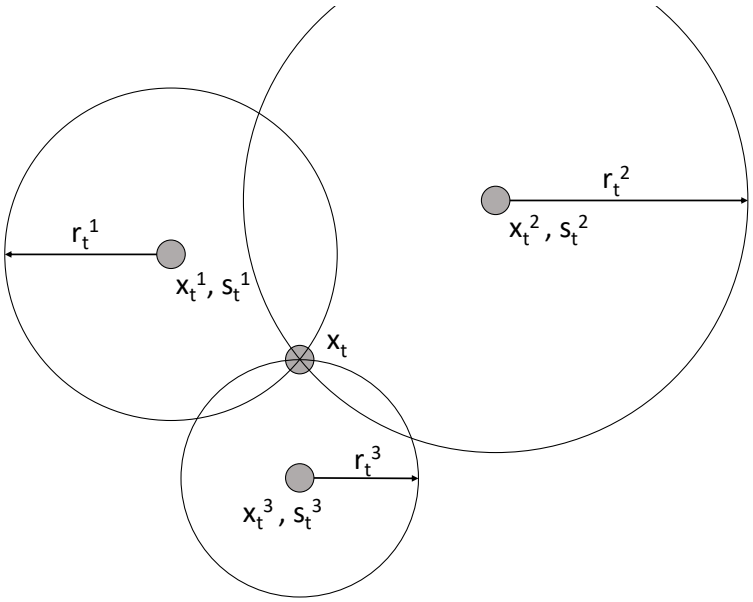

Abbildung 2.2.: Merkmalsbasierte Positionsbestimmung

Bei den Signaturen handelt es sich in der Regel um technische Identifizierungsverfahren. Dies können bei kamerabasierten Lösungen beispielsweise QR-Codes sein. Das autonome Objekt kann die Signatur und somit das Merkmal erkennen. Ein Global Navigation Satellite System (GNSS) nutzt merkmalsbasierte Karten mit eindeutigen Signaturen für die Positionsbestimmung. Ein GNSS-Empfänger kann sich aus den Umlaufdaten von Satelliten eine aktuelle Karte erzeugen, diese enthält die Position und Identifikationsnummer des Satelliten. Aus den von den Satelliten gesendeten Signalen kann der Empfänger die Identifikationsnummer

und die Signallaufzeit, also dem Abstand zum Satelliten, bestimmen. Eine Winkelbestimmung ist somit nicht möglich und die Position kann bestimmt werden, wenn der Empfänger die Signale von mehr als zwei Satelliten empfängt.

2.2. Sensoren

Da die beschriebenen Karten und die dazugehörigen Positionsbestimmungsverfahren auf Distanzmessungen basieren, soll in diesem Abschnitt eine Übersicht der einsetzbaren Sensoren gegeben werden. Hierbei soll zunächst auf die grundsätzliche Funktionsweise und mögliche Ausführungen der Sensoren eingegangen werden.

2.2.1. Ultraschallsensoren

Ultraschallsensoren nutzen die Luftschallgeschwindigkeit zur Messung von Distanzen. Es gibt weitere Anwendungsbereiche, wie die medizinische Diagnose oder Werkstoffprüfung, welche hier nicht weiter beschrieben werden sollen. Die Distanzmessung wird nach dem Puls/ Laufzeitprinzip durchgeführt, hierbei sendet der Sensor einen Ultraschallpuls aus und misst die Zeit bis zum Empfangen des Pulsechos. Mit dem Wissen über die Schallgeschwindigkeit kann dann die Distanz zur echogebenden Struktur berechnet werden. Ein Ultraschallsensor besteht grundsätzlich aus zwei Hautkomponenten, dem akustischen Wandlerelement und der Elektronik. Das Wandlerelement hat die Aufgabe, den Schall in die Luft abzustrahlen bzw. aus der Luft zu empfangen und in elektrische Signale umzuwandeln. Diese werden von der Elektronik verarbeitet, welche die Ergebnisse dann über eine Nutzerschnittstelle zur Verfügung stellt. Die Komplexität dieser Elektronik variiert stark in Abhängigkeit zur bereitgestellten Schnittstelle und lässt sich in folgende Gruppen unterteilen:

- Sensoren mit analoger Schnittstelle
- Sensoren mit digitaler Schnittstelle
- Sensoren mit zeitanaloger Schnittstelle
- Sensoren mit amplitudenanloger Schnittstelle

Die Elektronik der Sensoren mit analoger Schnittstelle führt keine Auswertung des Signals durch. Die Ansteuerung erfolgt durch eine Wechselspannung, welche direkt an den Wandler geleitet wird. Der Sensor gibt dann das rohe oder verstärkte Echosignal zurück. Die Elektronik besteht hier meistens aus wenigen passiven und diskreten Komponenten. Als Beispiel kann der Ultraschallsensor MA40S4R von *muRata* genannt werden [23]. Sensoren mit digitaler Schnittstelle verfügen zusätzlich über eine Einheit zur Auswertung des Echos. Somit

wird hier die Distanz zum echogebenden Objekt im Sensor berechnet und mittels digitaler Schnittstelle wie UART, I^2C oder CAN zur Verfügung gestellt. Der I2CXL-MaxSonar-Sensor von *MaxBotix* verfügt beispielsweise über eine I^2C Schnittstelle [17]. Sensoren mit zeitanaloger Schnittstelle verfügen in der Regel über eine bidirektionale, digitale Signalleitung. Durch Senden eines Pulses an den Sensor wird eine Messung ausgelöst, auf der selben Leitung wird dann bei Ankunft des Echos ein Schaltpuls zurückgesendet. Das ansteuernde Gerät kann so durch Zeitmessung die Distanz zum echogebenden Objekt berechnen. Der Ultraschall Distanzmesser von *Parallax* nutzt diese Funktionsweise [18]. Sensoren mit digitaler oder zeitanaloger Schnittstelle verbindet eine wichtige Gemeinsamkeit, bei Durchführung einer Messung wird nur die Distanz zu einem Objekt bestimmt. In komplexen Umgebungen bilden sich allerdings oft mehrere Echos, die zu unterschiedlichen Zeiten den Sensor erreichen. Sensoren mit amplitudenanaloger Schnittstelle geben das demodulierte Echosignal zurück. Abbildung 2.3 zeigt das amplitudenanloge Ausgangssignal des XL-MaxSonar-AE von *MaxBotix* [16], aus diesem Signal lassen sich mit einer Messung mehrere Objekte identifizieren.

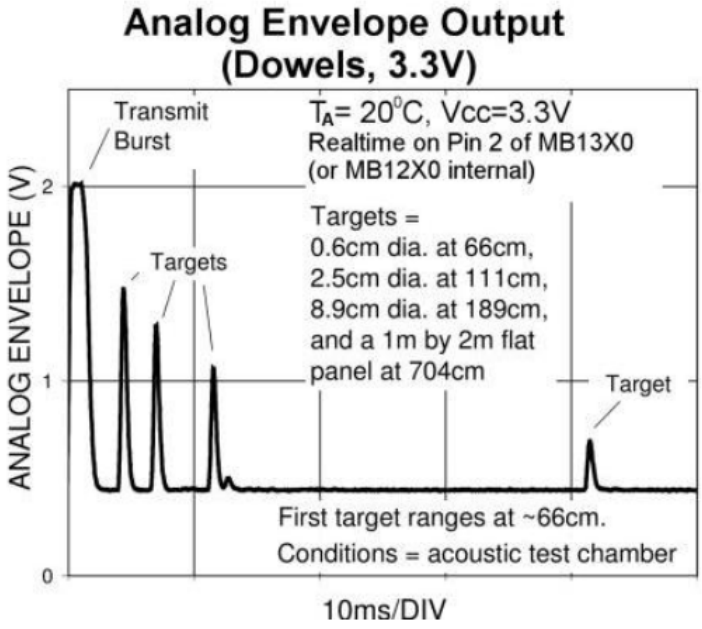

Abbildung 2.3.: Amplitudenanaloges Signal des XL-MaxSonar-AE Ultraschallsensors (Auszug: Datenblatt XL-MaxSonar-AE [16])

Da Ultraschallsensoren eine Distanz aus der Signallaufzeit bestimmen, ist dieses Verfahren abhängig von der Luftschallgeschwindigkeit, welche nicht als konstant angenommen werden kann. Die Schallgeschwindigkeit ist abhängig von der Temperatur, dem Luftdruck und der

Luftfeuchtigkeit. Sie kann in Abhängigkeit von der Temperatur approximiert werden, da dieser der größte Einflussfaktor ist.

$$c_T = 20.05\sqrt{T_C + 273.16}\frac{m}{s} \tag{2.4}$$

Unter den meisten Bedingungen (atmosphärischer Druck auf Meereshöhe) kann hierbei eine Genauigkeit besser als 1% erreicht werden [30].

Das Abstrahlverhalten von Ultraschallsensoren ist abhängig von der Bauform des Wandlerelements und dem dazugehörigen Gehäuse. Der effektive Öffnungswinkel kann hierbei bis zu 140° betragen [34] und ist somit in der Lage, einen großen Bereich auszumessen, eine Winkelinformation kann so nicht bestimmt werden. Auch ist die Messrate durch die vergleichsweise niedrige Schallgeschwindigkeit begrenzt und abhängig von der maximal messbaren Distanz. Die maximale theoretische Messrate f_m beträgt bei einer gewünschten Messdistanz von $s = 10m$ beispielsweise

$$f_m = \frac{1}{T_m} = \frac{1}{\frac{2s}{c_T}} = \frac{343\frac{m}{s}}{2 \cdot 10m} = 17,15Hz. \tag{2.5}$$

Hierbei wird von einer Temperatur von $20°C$ ausgegangen. [34] [26]

2.2.2. Radar-Sensoren

Ähnlich wie bei Ultraschallsensoren nutzt die RADAR-Technologie das Puls/ Laufzeitprinzip zur Distanzmessung. Der wesentliche Unterschied ist das genutzte Übertragungsmedium. Bei RADAR-Technologie sendet eine Antenne elektromagnetische Wellen aus, deren Reflexionen dann von einer Empfangsantenne aufgenommen werden. Die Bestimmung der Distanz zu einem reflektierenden Objekt kann hier auf verschiedene Weisen umgesetzt werden. Wie jedes System, das elektromagnetische Wellen aussendet, wird eine Funkzulassung der Sensoren benötigt. Hierbei wird zwischen Long Range Radar (LRR) und Short Range Radar (SRR) unterschieden, LRR-Systeme können einen Bereich bis zu 200 m erfassen und verwenden ein Frequenzband zwischen 77 und 81 GHz [2]. Im Vergleich dazu senden SRR-Systeme im Frequenzbereich 24,25 - 26,65 GHz [1] die Reichweite dieser Sensoren beträgt maximal 50 m mit einer Mindestreichweite von ca. 25 cm.

Puls-Demodulation

Das Puls-Demodulationsverfahren misst die Distanz zum reflektierenden Objekt ähnlich wie ein Ultraschallsensor nach dem Puls / Laufzeitprinzip. Hierbei erzeugt der Sender einen

bandbegrenzten Puls und misst die Zeit bei zur Ankunft der reflektierten Welle. Da die Ausbreitungsgeschwindigkeit nahezu der Lichtgeschwindigkeit entspricht ist, hier eine sehr aufwändige Elektronik notwendig. Bei einer Distanz von beispielsweise d = 150 m beträgt die Zeitdauer τ für einen Messpfad (Sende-/ und Reflexionsweg)

$$\tau = \frac{2 \cdot d}{c_0} \approx \frac{2 \cdot 150m}{300000\frac{km}{s}} \approx 1\mu s. \tag{2.6}$$

Aus diesem Grund wird dieses Verfahren für den Fahrzeugbereich nicht verwendet.

Frequency Modulated Continous Wave

Da das oben beschriebene Puls-Demodulationsverfahren sehr hohe Anforderungen an die Messtechnik stellt, wird die Laufzeitmessung beim Frequency Modulatd Coninous Wave (FMCW)-Verfahren indirekt durchgeführt. Anstatt die Zeiten zwischen Sendepuls und der Ankunft der Reflexion direkt zu messen, werden bei diesem Verfahren die Frequenzen des Sende / und Empfangssignals miteinander verglichen. Der Vergleich zweier Frequenzen ist in der Hochfrequenztechnik deutlich einfacher durch Mischen der Signale umzusetzen. Es gilt

$$a_0 cos(2\pi f_s t) \cdot a_1 cos(2\pi f_e t) = \frac{a_0 a_1 (cos(2\pi(f_s - f_e)t) + cos(2\pi(f_s + f_e)t))}{2}. \tag{2.7}$$

Durch das Mischen werden die Frequenzen f_s und f_e unterdrückt und es bilden sich neue Frequenzanteile, die der Differenz und der Summe beider Frequenzen entsprechen. Die Summe enthält keine hier relevanten Informationen und kann durch einen Tiefpassfilter unterdrückt werden. Um auf diese Weise eine Distanzmessung durchzuführen muss die Sendefrequenz zeitabhängig sein. Beim FMCW-Verfahren ändert der Sender seine Frequenz in einer Rampenfunktion, die Frequenz steigt also linear an. Der Sensor mischt durchgehend die ankommenden Signale mit seinem aktuellem Sendesignal. Aus dem Frequenzunterschied lässt sich dann auf die Distanz schließen. Ein Problem stellt bei dieser Methode der Doppler-Effekt dar. Sollte sich das reflektierende Objekt auf den Sender zu bewegen, oder vom Sender entfernen, so hat dies eine weitere Frequenzverschiebung zur Folge. Dieser kann durch eine zusätzliche abfallende Frequenzrampe herausgerechnet werden. Die Sendefrequenz verändert sich somit wie eine Dreiecksfunktion.

Abbildung 2.4 zeigt einen exemplarischen zeitlichen Ablauf des FMCW-Verfahrens, bei stillstehendem Sender und Empfänger lässt sich zu jeder Zeit der Frequenzunterschied *f_delta* bestimmen, welcher sich linear zur Distanz verhält. Bei relativer Geschwindigkeit zwischen Sensor und dem reflektierenden Objekt ändert sich die empfangene Frequenz entsprechend. Die nun gemessene Frequenzdifferenz *f_0* ist kleiner, da die Doppler-Frequenz

hinzukommt. Bei fallender Sendefrequenz ist die gemessene Empfangsfrequenz um die Doppler-Frequenz größer. Sind die beiden Frequenzdifferenzen *f_0* und *f_1* bekannt, kann die Doppler-Verschiebung herausgerechnet werden. Zusätzlich kann eine relative Geschwindigkeitsinformation zur Verfügung gestellt werden.

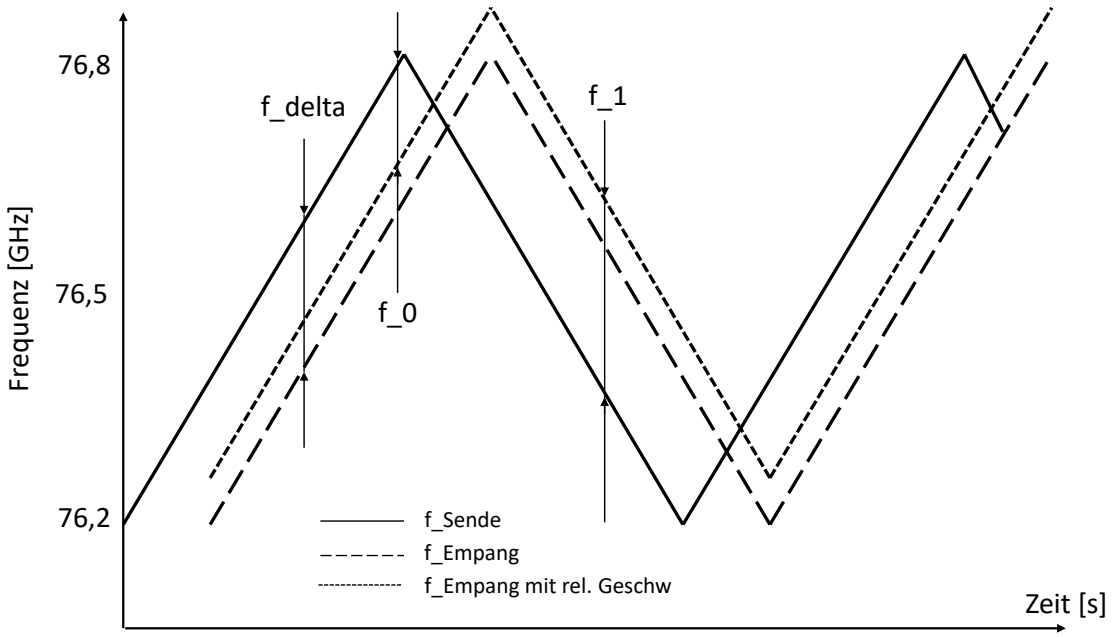

Abbildung 2.4.: Frequenzverlauf beim FMCW-Verfahren

Dieses Verfahren wird heute in vielen gängigen RADAR-Sensoren verwendet. Der Long-Range-Radar LRR3 von der *Robert Bosch GmbH* nutzt das FMCW-Verfahren zur Distanz/ und Geschwindigkeitsmessung [13]. Es gibt noch weitere Verfahren, die aktuell verwendet werden. Diese basieren allerdings auf dem FMCW-Verfahren und variieren die Form der Sendefrequenz. So werden beim *Chirp Sequense Modulation*-Verfahren mehrere hintereinander folgende Rampen gesendet. Die Dauer dieser Rampen ist so kurz, dass die Dopplerverschiebung nicht mehr relevant ist. Die SRR 200-Serie von *Continental* nutzt dieses Verfahren [4]. [34] [26]

2.2.3. LIDAR-Sensoren

Das LIDAR-Messverfahren ist ein optisches Verfahren zur Distanzmessung, hierbei können Ultraviolett-, Infrarot- oder für den Menschen sichtbare elektromagnetische Strahlen verwendet werden. Die Bestimmung erfolgt durch eine *Time of Flight*-Messung, hierbei wird, wie

bei der Ultraschallmessung, die Zeitdauer von der Aussendung eines Lichtpulses bis zum Empfang der reflektierten Strahlen bestimmt. Die Distanz zum reflektierenden Objekt kann dann mit Hilfe der Lichtgeschwindigkeit berechnet werden.

$$d = \frac{c_0 \cdot t}{2} \tag{2.8}$$

Wobei d für die Distanz zum Objekt, t für die Zeitdifferenz zwischen Sende- und Empfangszeitpunkt und c_0 für die Lichtgeschwindigkeit steht. Da die Zeitmessung den Sende- und Reflexionsweg beinhaltet, muss die berechnete Distanz halbiert werden. Im Vergleich zur RADAR-Messung gibt es beim LIDAR-Verfahren keine günstigere Methode, wie zum Beispiel FMCW, da das Mischen optischer Signale deutlich aufwendiger ist. Bei der *Time of Flight*-Messung sendet der Sensor ein moduliertes Lichtsignal aus, dieses ist nicht wie bei RADAR in der Frequenz, sondern in der Intensität des Lichtes moduliert. Hierbei werden Rechteckschwingungen, Sinusschwingungen oder Pulse erzeugt. Aus der Laufzeit oder Phasenverschiebung kann dann die Distanz berechnet werden. Die am meisten verwendete Methode ist die Puls-Modulation. LIDAR-Sensoren für Automobile erreichen ähnliche Genauigkeiten und Reichweiten wie RADAR-Sensoren, also ungefähr 200 m. Begrenzt ist die Reichweite durch die maximal erlaubt Intensität des ausgestrahlten Lichts, um Augenschäden bei Blendung zu vermeiden. Die Grenzwerte sind in der DIN-EN-60825-1 festgelegt.

Um eine Distanzauflösung im Zentimeterbereich zu erhalten, wird das 'Parallel-Gating'-Verfahren angewendet. Das durch die Lichtdiode empfangende reflektierte Signal wird hierbei zunächst vorverstärkt und dann an einen zeitlich gesteuerten Multiplexer geleitet. Dahinter wird das Signal digitalisiert und in einem Range-Gate gespeichert. Jeder Wert steht damit für eine Intensität und einem relativen Zeitpunkt nach dem Senden eines Pulses. Durch mehrmaliges, aufeinanderfolgendes Durchführen des Messvorgangs kann der Signal- / Rauschabstand noch weiter erhöht werden. Die Auflösung ist in diesem Fall bestimmt durch die Schaltrate des Multiplexers und kann mit der Kenntnis über die Form des ausgesendeten Pulses noch weiter erhöht werden. Hierbei wird die Form des ausgesendeten Pulses aus den empfangenen rekonstruiert. Mit dieser Methode lassen sich Genauigkeiten im Zentimeterbereich erreichen. Abbildung 2.5 zeigt die entstehenden Range-Gates bei einem reflektierenden Objekt und die Rekonstruktion des Sendepulses.

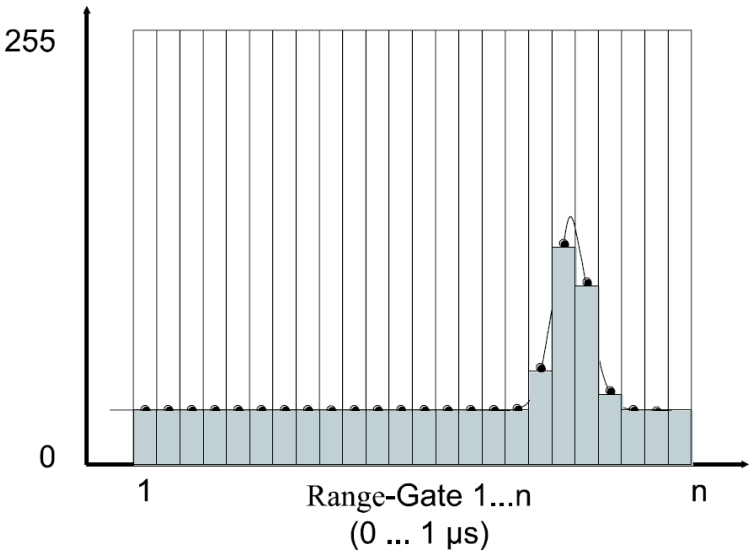

Abbildung 2.5.: Digitalisieren mittels Parallel-Gating (Auszug: 'Handbuch Fahrerassistenzsysteme' Abb.18.9 [34])

Das Pulsverfahren wird beispielsweise im SF02/F Laser Rangefinder von *LightWare Optoelectronics Ltd.* verwendet [22].

Da LIDAR-Sensoren Licht zur Distanzmessung verwenden, kann dieses Verfahren durch die Umgebung beeinflusst werden. So führt beispielsweise Nebel zu einer Dämpfung des ausgesendeten Lichts, was zu geringerer Genauigkeit und Reichweite führt. Auch spiegelnde Oberflächen, die den Lichtstrahl ablenken, beeinflussen das Messverfahren. Aus diesen Gründen werden LIDAR-Sensoren selten für sicherheitsrelevante Lösungen verwendet, obwohl sie unter guten Bedingungen mit sehr hoher Genauigkeit messen können. Durch die Bündelung des ausgesendeten Lichtpulses können Öffnungswinkel kleiner als 0.1° erreicht werden [34]. [27] [26]

2.2.4. Stereo-Kameras

Stereo-Kameras nutzen passive Stereoskopie um eine 3d Rekonstruktion der Umgebung durchzuführen. Das Prinzip basiert darauf, die gleiche Szene aus zwei verschiedenen Kamerapositionen aufzunehmen. Hierbei wird in allen aufgenommenen Bildern ein bestimmtes Korrespondenzmerkmal ermittelt. Da die Parameter (Position und Kalibrierung) beider verwendeten Kameras bekannt sind, lässt sich die räumliche Position des Szenenpunktes bestimmen. Die Erzeugung eines Stereo-Bildes erfolgt in zwei Schritten:

- Bestimmung der Korrespondenzmerkmale
- Berechnung der Stereogeometrie

Die Bestimmung von Korrespondenzmerkmalen kann durch unterschiedliche Verfahren durchgeführt werden. Am häufigsten wird hierfür das *Matching*-Verfahren verwendet. Hierbei wird aus einem der Bilder eine Menge von Korrespondenzkandidaten erzeugt, diese bestehen aus einer den Referenzpunkt umgebenen Region aus Bildpunkten. Diese besteht oft aus einem symmetrischen Fenster, zum Beispiel ein Rechteck, welches um den Referenzpunkt gelegt wird. Daher wird das Verfahren auch *Block-Matching* genannt. Die Region wird dann im zweiten Bild gesucht und der Punkt mit der höchsten Übereinstimmung ist dann der korrespondierende Punkt. Die Detektion kann besonders bei homogenen oder periodisch texturierten Bildbereichen zu Problemen führen. Daher wird der Suchraum häufig auf markante Bildbereiche eingeschränkt, um den Rechenaufwand zu verringern.

Als zweiter Schritt erfolgt die Berechnung der Stereogeometrie, hierbei wird die räumliche Position des Szenenpunktes relativ zur Kameraposition bestimmt. Der Aufbau eines Systems ist in Abbildung 2.6 dargestellt. Er besteht aus zwei Kameras, deren optische Zentren C_l und C_r parallel zueinander ausgerichtet sind. Der Szenenpunkt X_W ist in beiden Bildern als x_l und x_r enthalten. Ein solches rektifiziertes oder achsparalleles System stellt einen Sonderfall der allgemeinen Beschreibung dar, in dem die Kameras beliebig Ausgerichtet sind. Der Vorteil des achparallelen System ist, dass der Szenenpunkt ausschließlich eine horizontale Verschiebung aufweisen kann, was den Rechenaufwand deutlich verringert.

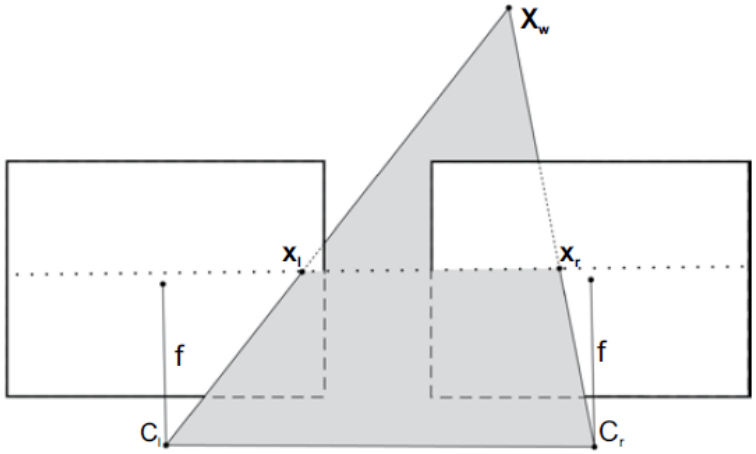

Abbildung 2.6.: Rektifiziertes Stereosystem (Auszug: 'Handbuch Fahrerassistenzsysteme' Abb.21.8 [34])

Zusätzlich zur reinen 3d Rekonstruktion der Umgebung ist es hierbei möglich, auch eine Detektion bewegter Objekte durchzuführen. Das 6D-Vision Verfahren stellt zusätzliche eine Geschwindigkeitsinformation für bestimmte Bildpunkte zur Verfügung. Verfahren wie *Sixtel-Welt* oder *Scene-Labeling* führen Objektklassifizierungen durch und können im Straßenverkehr so beispielsweise Fahrzeug, Gebäude und Fußgänger unterscheiden. Die *Stereo Video Camera 1* von Bosch stellt beispielsweise eine Tiefeninformation für Distanzen über 50 m zur Verfügung [12]. [34] [30]

2.3. Vorarbeiten im Urban Mobility Lab

2.3.1. Datenlogger

Im Rahmen eines Projektes im *Urban Mobility Lab* wurde ein autonom arbeitender GPS-Datenlogger für den Einsatz in Nahverkehrsbussen entwickelt [33]. Neben der GPS-Position werden weitere Messungen durchgeführt, die Datenrate beträgt für alle Parameter 10 Hz. Diese sind in Tabelle 2.1 dargestellt.

Tabelle 2.1.: Übersicht HAW-Datenlogger (Übernommen aus [33] und [14])

Parameter	Beschreibung	Einheit	Auflösung	Genauigkeit
Position	aus GPS	° dec.	0.0001	2.5 m CEP
Odometriedaten	aus Tachopulsen	4 Pulse/ m	Puls	0.25 m/ Puls
Beschleunigung	Drei Achsen	g	0.001	-
Drehrate	Drei Achsen	°/s	0.001	-
Magnetfeld	Drei Achsen	gauss	0.0002	-
Temperatur	Innerhalb des Datenloggers	°C	0.1	$\pm$ 1 °C
Luftdruck	Innerhalb des Datenloggers	hPa	0.01	$\pm$ 0.12 hPa
Türsignal	Tür offen/ geschlossen	logik	wahr/ falsch	-

Ein Blockschaltbild des HAW-Datenloggers ist in Abbildung 2.7 dargestellt. Der Datenlogger wurde für einen dauerhaften Einsatz für einen Zeitraum von bis zu einem Jahr entwickelt. Daher erfolgt die Energieversorgung durch das Stromnetz des Busses, zusätzlich wird die Energie durch einen Akkumulator zwischengespeichert. Außerdem erfolgt ein automatisches Hochladen der Messdaten auf einen hochschuleigenen Server. Dies ist durch ein GSM-Modem realisiert. Die Messdaten werden ebenfalls dauerhaft auf einem Wechselspeicher gespeichert. Ein Bild des HAW-Datenloggers ist in Abbildung 2.8 dargestellt.

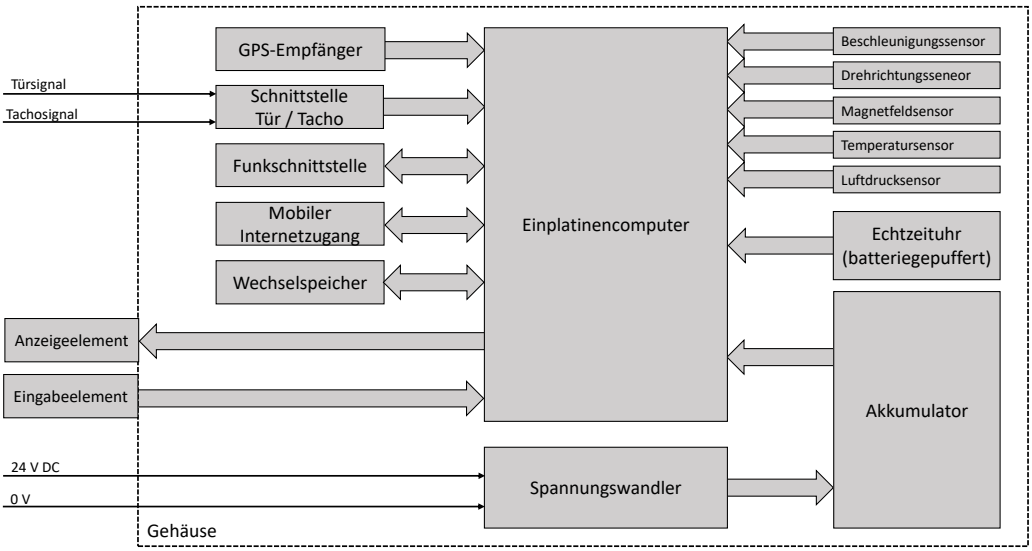

Abbildung 2.7.: Blockschaltbild des HAW-Datenloggers (Auszug: 'Bachelorthesis: Entwicklung eines autonom arbeitenden GPS-Datenloggers mit hoher Updatefrequenz für die Anwendung in Nahverkehrsbussen' [33] Abbildung 4.1)

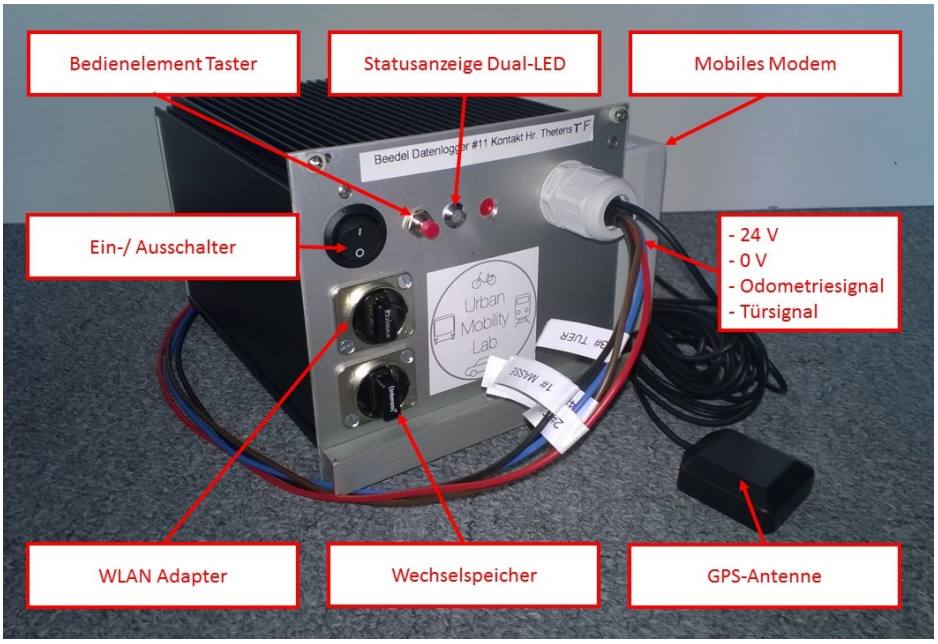

Abbildung 2.8.: HAW-Datenlogger (Auszug: 'Bachelorthesis: Entwicklung eines autonom arbeitenden GPS-Datenloggers mit hoher Updatefrequenz für die Anwendung in Nahverkehrsbussen' [33] Abbildung 5.11)

3. Anforderungsanalyse

Ziel der Arbeit ist es, ein System zu entwickeln, welches die Position eines Fahrzeuges in dicht besiedelten, urbanen Gebieten bestimmen kann. Das System soll somit dort zum Einsatz kommen, wo die Positionsbestimmung durch GNSS nur eingeschränkt funktioniert.

Die aus diesem Grund benötigten Messdaten sind in Tabelle 3.1 dargestellt. Ziel hierbei ist, eine Positionsinformation im gleichen Format, wie die von einem GNSS, bereitgestellt wird, zu erzeugen. Diese werden in geografischen Koordinaten angegeben, welche sich aus einem Längen- und Breitengrad zusammensetzen. Als Darstellungsart wird die Dezimalschreibweise gewählt, da sich hiermit eine beliebige Auflösung erreichen lässt. Mit $\pm$ 10 cm wird die zu erreichende Genauigkeit deutlich höher gewählt, als die durch ein GNSS-Sensor zur Verfügung gestellt wird. Diese liegt im zivilen Bereich bei ca. 2.5 m CEP. Zusätzlich soll auch eine Geschwindigkeitsinformation mit einer Auflösung von 0.1 km/h zur Verfügung gestellt werden, welche relativ zur Umgebung des Fahrzeuges gemessen wird.

Tabelle 3.1.: Messdatenanforderungen an das Gesamtsystem

Messdaten	**Anforderung**	**Auflösung**	**Genauigkeit**
Position	Wie GNSS (Geografische Koordinaten)	Dezimalgrad, 6 Nachkommastellen Entspricht $\approx 10cm$	$\pm$ 10 cm
Geschwindigkeit	Relativ zu Umgebung	0.1 km/h	$\pm$ 1 km/h

Neben den Messdatenanforderungen muss das System die in Tabelle 3.2 aufgeführten technischen Spezifikationen erfüllen. Hierzu zählt eine drahtlose Benutzerschnittstelle, über die sich das System starten und stoppen lässt. Außerdem soll es möglich sein, die aufgenommenen Daten hierüber übertragen zu können. Neben der reinen Übertragung von Messdaten erfolgt zusätzlich eine Speicherung aller Rohdaten über einen Zeitraum von mindestens einer Stunde, um ein zusätzliches nachträgliches Auswerten der Rohdaten durchführen zu können. Aus diesem Grund soll das System mehr Rohdaten aufnehmen, als zur Bestimmung von Position und Geschwindigkeit benötigt werden. So können weitere Funktionen implementiert oder die Positionsbestimmung verbessert werden. Um Echtzeitfähigkeit gewährleisten zu können, darf die Ausgabenverzögerung nicht mehr als 50 ms betragen. Bei

Schrittgeschwindigkeit als gewählte Operationsgeschwindigkeit würde dies zu einem Ortsverzögerung von 5 cm führen. Die Datenrate soll passend zum HAW-Datenlogger 10 Hz betragen.

Zusätzlich soll das System über Referenzsensoren, für die zu bestimmende Position und Geschwindigkeit, verfügen. Als Referenzposition soll ein GNSS-Sensor dienen. Die Geschwindigkeit wird anhand der Fahrzeugodometrie bestimmt.

Tabelle 3.2.: Technische Anforderungen

Funktion	**Anforderung**
Nutzer-Schnittstelle	Drahtlos
Datenaufzeichnung	Rohdatensicherung für 1 Stunde
Erweiterbarkeit	Möglichkeit zur Funktionserweiterung und weiteren Auswertung
Ausgabeverzögerung	Maximal 50 ms
Fahrzeuggeschwindigkeit	0,1 m/s - 1 m/s
Ausgaberate	Mindestens 10 Hz
Referenzsystem Position	GNSS
Referenzsystem Geschwindigkeit	Odometrie

Zuletzt folgen die in Tabelle 3.3 aufgeführten Umweltanforderungen. Als Applikationsfahrzeug steht ein Mazda 5 für Testfahrten zur Verfügung. Die Montage des Systems muss erfolgen, ohne dauerhafte Änderungen am Fahrzeug durchführen zu müssen, so dass ein rückstandsfreie Demontage möglich ist. Auch darf während des Betriebes die Fahrzeugzulassung nicht erlöschen, damit Testfahrten im öffentlichen Straßenverkehr durchgeführt werden können. Die Energieversorgung soll aus dem Fahrzeug erfolgen oder alternativ aus Akkumulatoren. Die Arbeitstemperatur soll in einem Temperaturbereich von 0°C bis 40°C liegen.

Tabelle 3.3.: Umweltanforderungen

Funktion	**Anforderung**
Applikationsfahrzeug	Mazda 5 (2006 USA)
Einbauart	Rückstandsfreie Demontage
Fahrzeugzulassung	Darf nicht erlöschen
Energieversorgung	Energieversorgung aus Fahrzeug oder Akkubetrieb
Temperaturbereich	0°C - 40°C

4. Systementwurf

4.1. Entwicklung der Messmethode

In diesem Abschnitt wird die Entwicklung der Messmethode für die Bestimmung von Position und Geschwindigkeit beschrieben. Beide Methoden bedienen sich grundsätzlich dem gleichen Messprinzip.

4.1.1. Bestimmung der Position

Die Positionsbestimmung basiert auf der Annahme, dass in urbanen Gebieten viele wiedererkennbare Strukturen auftreten, an denen sich orientiert werden kann. Hierzu zählen vor allem Gebäudefassaden, die eindeutige Merkmale aufweisen. Die sich so bildenden Häuserschluchten werden *Urban Canyons* genannt. Das System soll diese Canyons und weitere Merkmale wie zum Beispiel Bäume oder Straßenlaternen vermessen, um eine Positionsbestimmung anhand einer Karte durchzuführen. Abbildung 4.1 zeigt den grundsätzliche Messvorgang. Ein Fahrzeug, ausgestattet mit dem Messsystem, durchfährt einen *Urban Canyon*, welcher eine wiedererkennbare Struktur ausweist. Die Vermessung erfolgt durch Distanzmessungen zu beiden Seiten, es ergibt sich der Messvektor zum Zeitpunkt t

$$\vec{z}_t = \begin{pmatrix} r_L \\ r_R \end{pmatrix}. \tag{4.1}$$

Wobei r_L und r_R für die Distanz zum parallel verlaufenden *Canyon* stehen. Durch diese und die vergangenen Messungen ($\hat{z}_{t-1} \ldots \hat{z}_{t-n}$) soll dann eine lokale Karte m_{local} erstellt werden, die dann mit einer globalen Karte abgeglichen wird. Mit dem Wissen über die gefahrene Geschwindigkeit v kann die Zeitfunktion $\vec{z}(t)$ in eine Ortsfunktion $\vec{z}(s)$ umgerechnet werden. Unter der Annahme, dass sich das Fahrzeug immer auf der Straße befindet, ist eine eindimensionale Geschwindigkeitsinformation ausreichend. Mit diesem System ist es möglich die *Urban Canyons* mit nur zwei fest installierten Distanzsensoren zu vermessen.

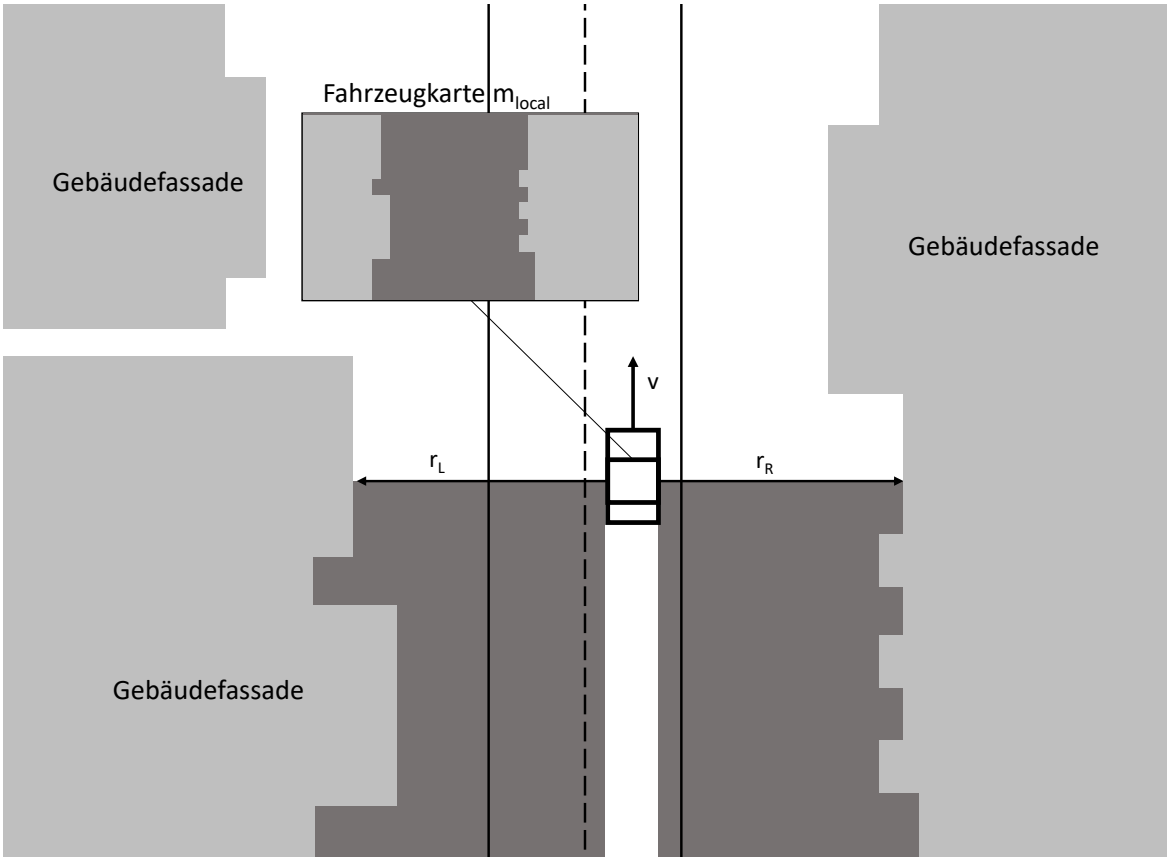

Abbildung 4.1.: Skizze der Messmethode zur Positionsbestimmung

Um sich anhand der aufgenommenen Daten orientieren zu können, muss eine globale Karte der Region vorhanden sein. Für die Wahl des Kartenprinzips wird die Realisierbarkeit der in Abschnitt 2.1 beschriebenen Karten geprüft.

- **Rasterkarte:** Ein urbanes Gebiet mit einer Rasterkarte darzustellen ist grundsätzlich möglich, aufgrund der hohen benötigten Auflösung führt dies allerdings zu einem sehr hohen Speicherbedarf. Auch ist es für die Durchführung eines *Map Matching* notwendig die, lokale Karte relativ zur globalen Karte auszurichten. Hierfür ist die Kenntnis über die Ausrichtung des Fahrzeuges notwendig. Ist dies gegeben, so ist eine Positionsbestimmung direkt aus den Messungen möglich.
- **Topologische Karte:** Diese Karte hat den Vorteil, dass der Grundaufbau sehr stark dem einer urbanen Umgebung entspricht. Da davon ausgegangen werden kann, dass sich ein Fahrzeug ausschließlich auf Straßen befindet, können Kreuzungen in Knoten (*Notes*) und Straßen in Linien (Traveling Edges) substituiert werden. Auf diese Weise enthält eine Karte nur für die Positionsbestimmung relevante Informationen. Auch die

Kenntnis über die Ausrichtung des Fahrzeuges ist dann nicht notwendig, da Fahrzeug eine Straße nur in zwei Richtungen befahren kann. Da die Positionsbestimmung bei diesen Karten an den Knoten erfolgt, müssten in diesem Fall Kreuzungen und keine *Urban Canyons* erkannt werden.

- **Merkmalsbasierte Karte:** Auch die Erzeugung merkmalsbasierter Karten ist möglich. Es würde sich hierbei um eine Karte mit mehrdeutigen Signaturen handeln, welche Fassadenstrukturen beschreiben. Mit dieser Methoden könnte ein urbanes Gebiet mit relativ geringem Speicheraufwand beschrieben werden. Das Verfahren setzt eine sehr robuste Merkmalserkennung voraus.

Das System soll die Positionsbestimmung in einer topologischen Karte durchführen, da der Aufbau stark dem eines Straßennetzes entspricht. Die Karte muss allerdings dahingehend verändert werden, dass die Karteninformationen in den *Traveling Edges* $c_{i,j}$ und nicht in den *Notes* enthalten sind. Die *Traveling Edge* sind dann wie folgt definiert:

$$c_{i,j} = \{r_L(s), r_R(s), x_{Lat}(s), x_{Lon}(s)\} \tag{4.2}$$

Eine *Traveling Edge* beschreibt den *Urban Canyon* einer Straße zwischen zwei Kreuzungen, die Funktionen r_L un r_R beschreiben den Verlauf der Distanzen nach links und rechts zu Häuserfassaden oder anderen Objekten. Die Funktionen $x_{Lat}(s)$ und $x_{Lon}(s)$ beschreiben den Verlauf der geografischen Koordinaten, den die Straße durchfährt. Die Strecke s steht für die zurückgelegte Distanz von einem Knoten zum nächsten, das Maximum gibt somit die Länge der Straße an. Zur Speicherung erfolgt eine Diskretisierung der Funktionen auf die benötigte räumliche Auflösung.

Beim entwickelten Messverfahren muss zur Positionierung zwischen der Position in Fahrtrichtung (x_y) und der seitlichen ausgerichteten Position (x_x) unterschieden werden. Die Position x_x ist hierbei direkt durch die Genauigkeit der Distanzmessung gegeben. Die Auflösung der Position in Fahrtrichtung x_y ist in diesem Fall nicht von Genauigkeit der Messung, sondern von der Messrate und Geschwindigkeit des Fahrzeuges abhängig. Je schneller sich das Fahrzeug bewegt, desto größer ist die räumliche Distanz der Distanzmessungen. Es gilt

$$\Delta x_y(t) = \frac{v(t)}{f_{mess}} \tag{4.3}$$

Wobei Δx_y für die räumliche Auflösung der Distanzmessungen zum Zeitpunkt t, $v(t)$ für die aktuelle Geschwindigkeit des Fahrzeuges und f_{mess} für die Messrate der Distanz steht. Die zu erwartende Positionsauflösung ist also antiproportional zur Geschwindigkeit und die Messrate der Distanzmessung muss ausreichend hoch sein, damit die geforderte Messauflösung erreicht wird.

4.1.2. Bestimmung der Geschwindigkeit

Die aktuelle Geschwindigkeit des Fahrzeuges soll ebenfalls durch Vermessung der Gebäudefassaden bestimmt werden, dies wird durch eine parallele Distanzmessung zweier Sensoren durchgeführt. Das Messprinzip ähnelt dem Verfahren zur Positionsbestimmung, nur dass in diesem Fall kein Vergleich mit einer globalen Karte, sondern mit den Distanzmessungen des parallel installierten Sensors stattfindet. Beide Sensoren werden die gleichen Distanzen messen, auf Grund des räumliche Abstandes r_A allerdings mit einer Zeitdifferenz Δt. Aus dieser Zeitdifferenz kann dann mit der Kenntnis des Abstandes r_A eine Geschwindigkeit berechnet werden.

$$v = \frac{r_A}{\Delta t} \tag{4.4}$$

Da für die Bestimmung der Zeitdifferenz eine einzelne Distanzmessung nicht ausreicht, werden auch die vergangenen Messungen mit einbezogen und ein Vektor

$$\vec{z_L} = \{r_L(t), r_L(t - T_{mess}), r_L(t - 2T_{mess}), ..., r_L(t - nT_{mess})\} \tag{4.5}$$

gebildet. Aufgrund der zeitdiskreten Abtastrate kann die Zeitverschiebung nur als ein vielfaches der Abtastperiode bestimmt werden und ist durch die Länge des Messvektors n begrenzt.

$$max(\Delta t) = n \cdot T_{mess} \tag{4.6}$$

Die Zeitverschiebung kann somit durch einen Verschiebungsfaktor $m \in \{0, 1, 2, ...n\}$ beschrieben werden, welcher durch

$$m = round\left(\frac{\Delta t}{T_{mess}}\right) \tag{4.7}$$

bestimmt ist. Durch Verwendung der Gleichungen 4.4 und 4.7 kann dann die geschätzte Geschwindigkeit v' bestimmt werden.

$$v' = \frac{r_A}{m \cdot T_{mess}} = \frac{r_A}{T_{mess}} \cdot \frac{1}{m} \text{ mit } m \neq 0 \tag{4.8}$$

Bei $m = 0$ handelt es sich um einen Sonderfall, da dies bedeutet, dass innerhalb einer Messperiode keine Zeitverschiebung aufgetreten ist. Somit kann nicht zwischen Stillstand und der maximal ermittelbaren Geschwindigkeit unterschieden werden. Die maximal messbare Geschwindigkeit bei $m = 1$ ist durch die Messrate und den Abstand der Sensoren gegeben. Die minimal messbare Geschwindigkeit wird zusätzlich durch die Länge der Vek-

toren n festgelegt.

$$v'_{max} = \frac{r_A}{T_{mess}} \tag{4.9}$$

$$v'_{min} = \frac{r_A}{n \cdot T_{mess}} \tag{4.10}$$

4.2. Architektur des Sensorsystems

Um die in Abschnitt 4.1 entwickelten Methoden realisieren zu können, ist es notwendig an mehreren Positionen des Fahrzeuges Distanzmessungen durchzuführen. Der grundsätzliche Aufbau des Messsystems ist in Abbildung 4.2 dargestellt, dieser sieht je zwei parallele Distanzmessungen jeweils orthogonal zur Fahrtrichtung vor. Zur Wahrung der Übersichtlichkeit wird das Gesamtsystem in fünf Funktionseinheiten aufgeteilt. Die vier Sensoreinheiten führen Distanzmessungen durch und sind in Abschnitt 4.4 genauer beschrieben. Die parallel liegenden Einheiten werden mit dem, für die Geschwindigkeitsbestimmung, wichtigen Abstand r_A am Fahrzeug befestigt. Alle weiteren Funktionen werden von einer zentralen Einheit umgesetzt, diese ist mit allen Sensoreinheiten verbunden. Um die Integrität der Messdaten zu erhalten, ist die Zeitsynchronität der Messungen von besonderer Bedeutung.

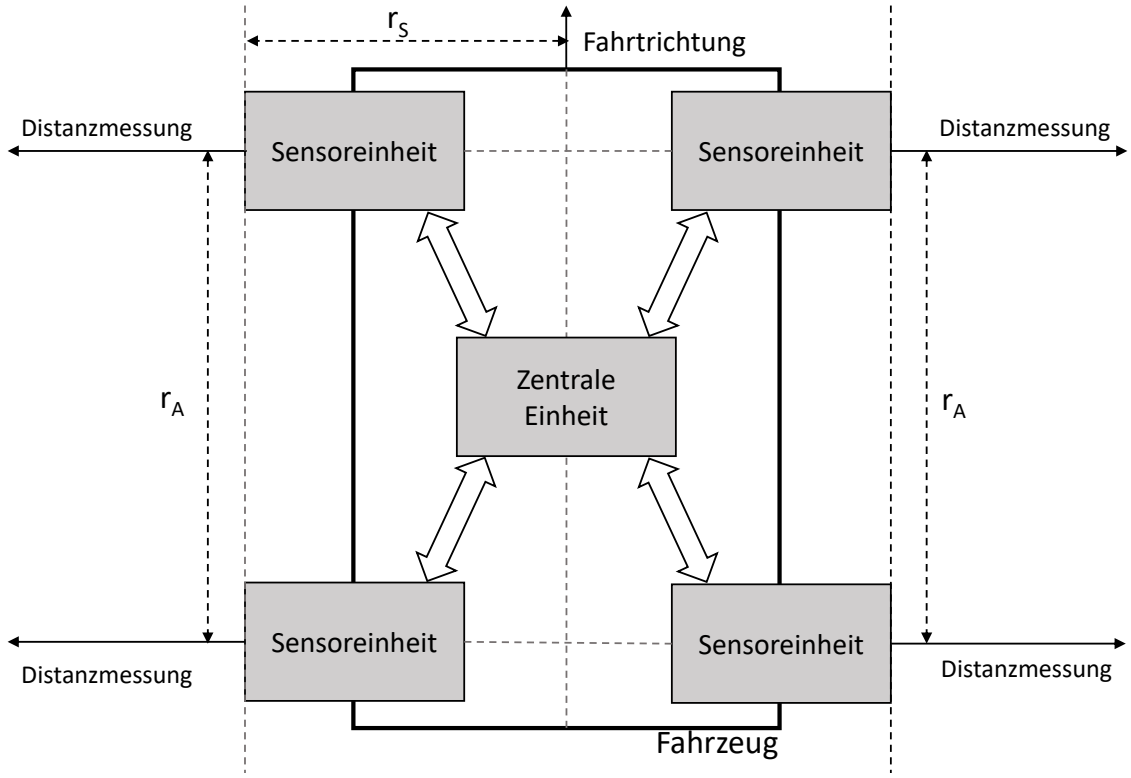

Abbildung 4.2.: Aufbau des Messsystems

4.3. Zentrale Einheit

Das Grundkonzept der zentralen Einheit ist in Abbildung 4.3 dargestellt. Sie hat die Aufgabe, die Distanzmessungen der Sensoreinheiten zu empfangen. Gleichzeitig findet hier die Synchronisierung und Verarbeitung der Messungen statt. Das System sieht als Hauptkomponente einen Einplatinencomputer vor, der die Ansteuerung und Verarbeitung aller weiteren Komponenten übernimmt. So soll die zentrale Einheit die Nutzer-Schnittstelle zur Verfügung stellen. Außerdem sind hier die Referenzsensoren für Position und Geschwindigkeit enthalten. Die Rohdaten werden in einer Speichereinheit gespeichert. Die für die Positionsbestimmung wichtigste Komponente ist der Einplatinencomputer, hier werden die Sensoreinheiten angesteuert, die empfangenen Messdaten aufgenommen und ausgewertet, so dass eine Positions- und Geschwindigkeitsinformation zur Verfügung steht. Außerdem muss die zentrale Einheit sicherstellen, dass alle Funktionseinheiten mit der gleichen Zeitbasis arbeiten.

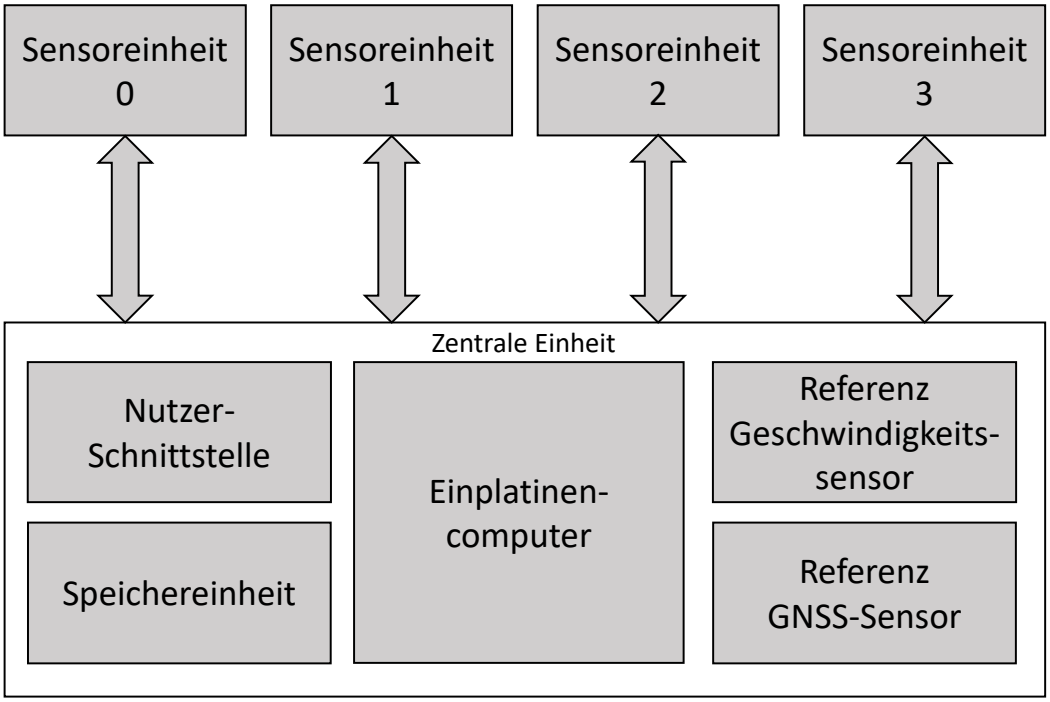

Abbildung 4.3.: Grundkonzept der zentralen Einheit

4.3.1. Einplatinencomputer

Der Einplatinencomputer soll die Ansteuerung und Verarbeitung der Komponenten übernehmen. Hierfür müssen die folgenden Mindestanforderungen erfüllt werden:

- Betriebssystem mit den benötigten Grundfunktionen
- Ausreichende Rechenleistung zur Ansteuerung der Komponenten und Ausführung der Datenverarbeitung
- Schnittstellen für alle Komponenten
- Einsetzbar im geforderten Temperaturbereich

4.3.2. Nutzer-Schnittstelle

Die Nutzer-Schnittstelle stellt die Verbindung zwischen dem Messsystem und einem Nutzer her, damit dieser eine Messung starten und stoppen kann. Auch die Extraktion der Roh-

daten erfolgt hierüber. Außerdem ist eine drahtlose Verbindung gefordert, daher muss die Schnittstelle gegen unbefugte Zugriffe geschützt werden können und über eine Reichweite von mindestens 5 m verfügen, damit eine Verwendung innerhalb eines Fahrzeuges gegeben ist. Zuletzt ist es notwendig, dass die Schnittstelle eine hohe Kompatibilität zu anderen Computersystemen aufweist. Zusammengefasst ergibt dies:

- Schnittstelle zum Einplatinencomputer
- Hohe Kompatibilität zu anderen Computersystemen
- Schutz vor Fremdzugriff
- Reichweite mindestens 5 m
- Möglichkeit für Rohdatenextraktion
- Einsetzbar im geforderten Temperaturbereich

4.3.3. Speichereinheit

Die Speichereinheit soll die Rohdaten der Distanzmessungen speichern, damit diese für nachträgliche Auswertungen zur Verfügung stehen. Laut den Anforderungen aus Tabelle 3.2 muss dies für einen Messzeitraum von mindestens einer Stunde gegeben sein. Auch die Geschwindigkeit der Daten-Schnittstelle muss ausreichend hoch sein. Sie muss also folgende Eigenschaften vorweisen:

- Schnittstelle zum Einplatinencomputer
- Speicherkapazität für eine Stunde Rohdaten
- Schnittstellengeschwindigkeit ausreichend hoch
- Einsetzbar im geforderten Temperaturbereich

4.3.4. Referenz GNSS-Sensor

Der GNSS-Sensor soll eine Referenzposition zur Verfügung stellen, mit dem das System validiert werden kann. Der Sensor muss somit in der Lage sein, die Position des Fahrzeuges in gleicher Rate, wie der des Sensorsystems zu bestimmen. Die Rohdaten sollen ebenfalls über einen Zeitraum von mindestens einer Stunde gespeichert werden können. Es ergeben sich folgende Mindestanforderungen:

- Schnittstelle zum Einplatinencomputer

- Speicherung der Rohdaten für mindestens eine Stunde
- Datenrate mindestens 10 Hz
- Einsetzbar im geforderten Temperaturbereich

4.3.5. Referenz Geschwindigkeitssensor

Zusätzlich zur Positionsmessung verfügt die zentrale Einheit über einen Geschwindigkeitssensor, der auf Basis von Odometrieinformationen eine Geschwindigkeitsinformation mit der geforderten Auflösung und Genauigkeit zur Verfügung stellt. Diese soll zur Validierung der durch das Sensorsystem erfassten Geschwindigkeit verwendet werden. Die Messdaten müssen ebenfalls über einen Zeitraum von mindestens einer Stunde gespeichert werden können und mit der gleichen Messrate von 10 Hz zur Verfügung stehen. Es ergeben sich folgende Mindestanforderungen:

- Schnittstelle zum Einplatinencomputer
- Speicherung der Rohdaten für mindestens eine Stunde
- Datenrate mindestens 10 Hz
- Auflösung 0.1 km/h
- Genauigkeit $\pm$ 1 km/h
- Einsetzbar im geforderten Temperaturbereich

4.4. Sensoreinheit

Die Sensoreinheit hat die Aufgabe, Distanzmessungen durchzuführen und an die zentrale Einheit zu senden. Hierfür soll eine eigenständige Einheit entwickelt werden, welche die Ansteuerung und das Auslesen der Sensoren übernimmt. Für die Vermessung von Gebäudefassaden wird ein Distanzsensor mit einem sehr konzentrierten Abstrahlverhalten verwendet werden. Um zusätzlich, wie in den Anforderungen in Tabelle 3.2 angegeben, weitere Möglichkeiten zur Auswertung bieten zu können, soll eine Vermessung des Nahbereichs erfolgen. Dieser soll es ermöglichen, Objekte zur Erstellung von beispielsweise merkmalsbasierten Karten zu erkennen.

Das Grundkonzept der Sensoreinheit ist in Abbildung 4.4 darstellt, es sieht die Verwendung von drei unterschiedlichen Sensoren vor. Für die Vermessung des *Urban-Canyons*

wird ein LIDAR-Sensor verwendet, außerdem soll das System über einen Ultraschallsensor für zeitanaloge Distanzmessung und eine Stereo Kamera verfügen. Die Ansteuerung der Sensoren soll über einen Einplatinencomputer erfolgen, welcher auch die Kommunikation mit der zentralen Einheit übernimmt. Außerdem sieht es eine Debug-Schnittstelle vor, welche die Fehlersuche und Behebung, sowie eine Möglichkeit zur Überprüfung des Status ermöglicht.

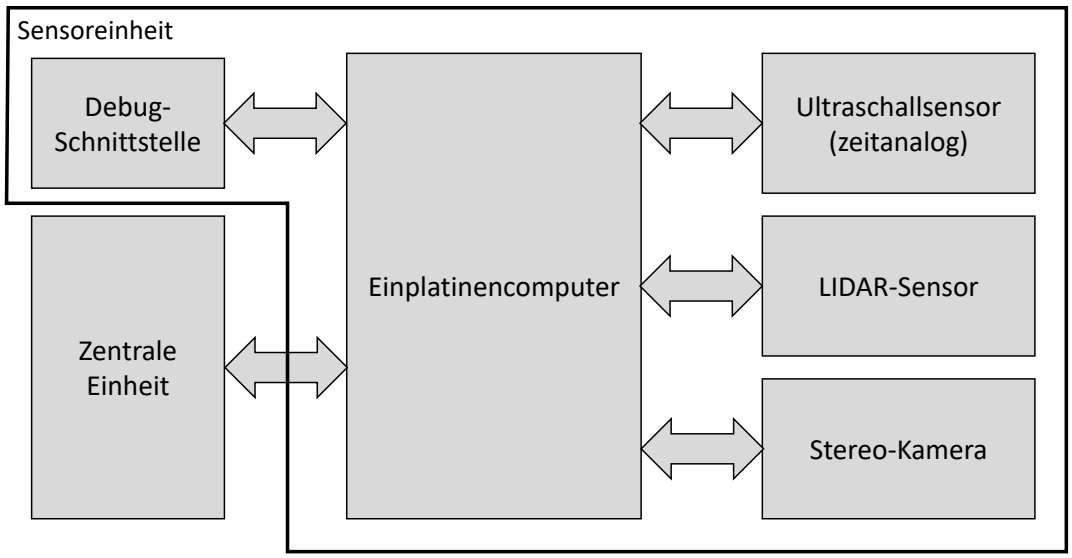

Abbildung 4.4.: Grundkonzept der Sensoreinheit

4.4.1. Einplatinencomputer

Als zentrale Einheit hat der Einplatinencomputer die Aufgabe, die Sensoren anzusteuern und deren Messdaten aufzunehmen. Hierfür muss dieser die folgenden Mindestanforderungen erfüllen:

- Betriebssystem mit den benötigten Grundfunktionen
- Ausreichende Rechenleistung zur Durchführung der Messungen und dem Senden der Daten
- Schnittstellen für alle Sensoren
- Schnittstelle zur zentralen Einheit
- Einsetzbar im geforderten Temperaturbereich

4.4.2. Debug-Schnittstelle

Die Debug-Schnittstelle soll, ähnlich wie die Nutzer-Schnittstelle der zentralen Einheit, einen Zugriff auf die Einheit ermöglichen. Dieser erfolgt ebenfalls drahtlos für gleiche Reichweiten und bietet eine hohe Kompatibilität zu anderen Computersystemen. Zusammengefasst ergibt dies:

- Schnittstelle zum Einplatinencomputer
- Hohe Kompatibilität zu anderen Computersystemen
- Schutz vor Fremdzugriff
- Reichweite mindestens 5 m
- Einsetzbar im geforderten Temperaturbereich

4.4.3. LIDAR-Sensor

Der LIDAR-Sensor soll die *Urban-Canyons* mit höchstmöglicher Genauigkeit vermessen. Da aus diesem Messungen die Position und die Geschwindigkeit ermittelt wird, muss die Messrate ausreichend hoch sein, damit die in Tabelle 3.1 geforderten Auflösungen und Genauigkeiten erreicht werden können. Für die Positionierung seitlich zur Fahrtrichtung gilt dies ebenfalls für Messgenauigkeit des Sensors.

Somit muss der Sensor die folgenden Eigenschaften vorweisen:

- Schnittstelle zum Einplatinencomputer
- Messbereich 1 m bis 30 m
- Lichtstrahl nicht gefährlich für Augen
- Lichtstrahl nicht sichtbar für Augen
- Öffnungswinkel kleiner als 0.5°
- Messrate ausreichend für Positions- und Geschwindigkeitsbestimmung
- Messgenauigkeit ausreichend für Positionsbestimmung
- Möglichkeit zur Synchronisierung der Messungen
- Einsetzbar im geforderten Temperaturbereich

4.4.4. Ultraschallsensor

Der Ultraschallsensor soll das Umfeld mit einem größeren Erfassungswinkel vermessen. Hierzu zählen nicht nur die Gebäudefassaden, sondern auch Objekte in direkter Umgebung. Der Sensor muss über die folgenden Eigenschaften verfügen:

- Schnittstelle zum Einplatinencomputer
- Messbereich 0 m - 5 m
- Amplitudenanloge Messung möglich
- Möglichkeit zur Synchronisierung der Messungen
- Einsetzbar im geforderten Temperaturbereich

4.4.5. Stereo-Kamera

Die Stereo-Kamera soll, wie der Ultraschallsensor, die Umgebung in einem größeren Winkel erfassen. Neben der Erstellung eines Tiefenbildes erfolgt zusätzlich eine Speicherung der aufgenommenen Bilder, um nachträgliches Abgleichen der Sensorwerte zu ermöglichen. Außerdem soll es möglich sein, das ausgesendete Licht des LIDAR-Sensors erfassen zu können. Die Kamera soll Distanzen von mindestens 10 m erfassen und muss über die folgenden Eigenschaften verfügen:

- Schnittstelle zum Einplatinencomputer
- Messbereich 0 m - 10 m
- Speichern der Rohdaten möglich
- LIDAR-Lichtpunkt in Rohdaten sichtbar
- Einsetzbar im geforderten Temperaturbereich

5. Realisierung

Da einige, der im Systementwurf entwickelten Funktionen bereits vom HAW-Datenlogger [33] realisiert sind, soll das Sensorsystem darauf aufbauen. Die folgenden Abschnitte beschreiben zunächst den Aufbau des Sensorsystems. Eine detaillierte Beschreibung von zentraler Einheit und Sensoreinheit erfolgt in den folgenden Abschnitten.

5.1. Sensorsystem

5.1.1. Blockschaltbild

Das Blockschaltbild des Sensorsystems ist in Abbildung 5.1 dargestellt. Als Schnittstelle zwischen der zentralen Einheit und den Sensoreinheiten wird die Ethernet-Technologie verwendet, so dass sich alle Einheiten in einem Local Area Network (LAN) befinden. Hierbei soll die zentrale Einheit, welche die Messdaten empfängt und verarbeitet, als Server und die Sensoreinheiten als Klienten agieren. Hierfür muss für die jeweiligen Einheiten eine entsprechenden Software erstellt werden. Die Energieversorgung der Sensoreinheiten erfolgt über die *Power over Ethernet*-Technologie. Hierbei wird die Energie über das Ethernet-Kabel übertragen. Die PoE-Schnittstelle für die Sensoreinheiten werden durch einen *DLOCK 87682 4 + 1 Port PoE Switch* zur Verfügung gestellt, welcher alle Komponenten miteinander verbindet. Ein Wechselrichter vom Typ *HQ-INV150WU-12* wandelt die im Applikationsfahrzeug zur Verfügung stehende 12 V Bordspannung in ein 230 V Wechselspannungssignal um und versorgt so das Netzteil des *Switches* mit Energie. Zusätzlich verfügt der Wandler über einen 5 V Gleichspannungsausgang, welcher zur Versorgung der zentralen Einheit verwendet wird.

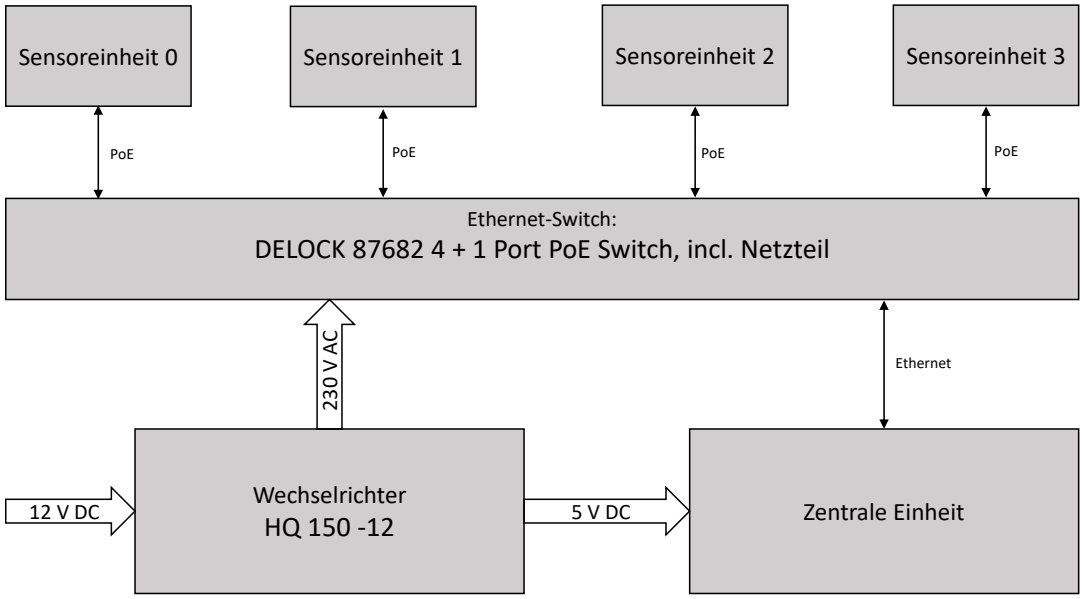

Abbildung 5.1.: Blockschaltbild des Sensorsystems

5.1.2. Gehäuse

Als Gehäuse wird eine Fahrzeug-Dachbox verwendet, welche auf dem Dachgepäckträger eines Fahrzeuges angebracht werden kann. Im Inneren werden die Sensoreinheiten an einem Metallrahmen befestigt, der mit der Dachbox und dem Dachgepäckträger verbunden ist. Auch der Ethernet-Switch ist hier fest eingebaut (Abbildung 5.2). Der Wechselrichter und die zentrale Einheit befinden sich während der Messung im Fahrzeug, somit werden zwei Leitungen zwischen Fahrzeuginnenraum und der Dachbox benötigt:

- 230 V Leitung für die Energieversorgung
- Ethernet-Kabel für die Datenübertragung an den Datenlogger

Durch den Aufbau mit einem Metallrahmen wird ein paralleler Abstand zwischen den Sensorboxen von 0.51 m erreicht. Abbildung 5.3 zeigt die Sensoreinheit auf dem Applikationsfahrzeug befestigt. Durch diese Methode ist eine rückstandsfreie Demontage möglich.

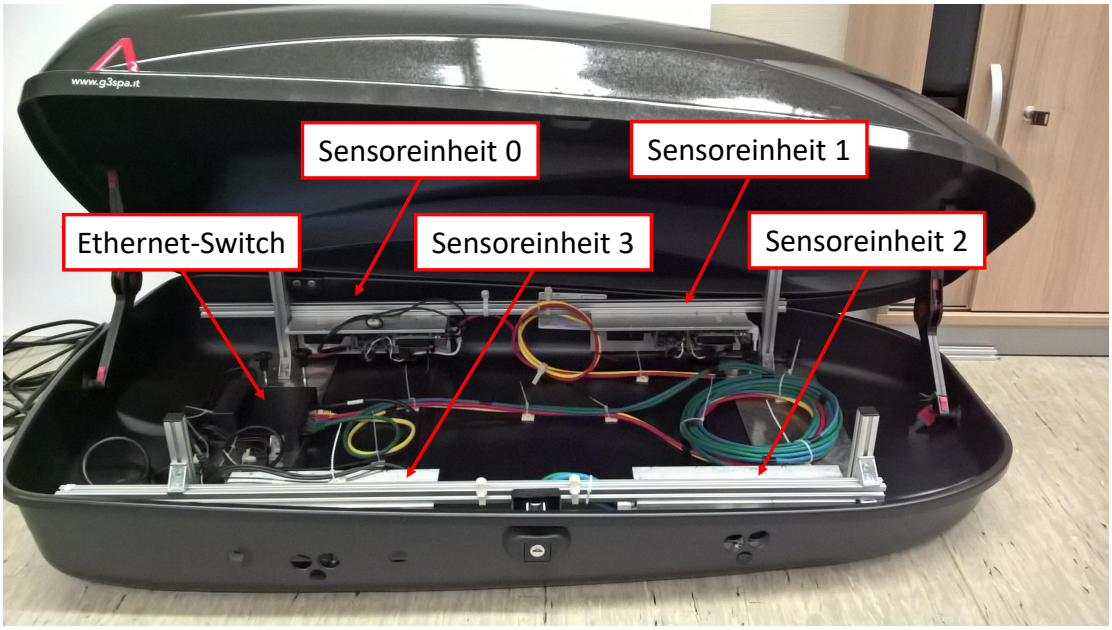

Abbildung 5.2.: Gehäuse des Sensorsystems

Abbildung 5.3.: Sensorsystem auf Autodach montiert

5.2. Zentrale Einheit

5.2.1. Einplatinencomputer

Der HAW-Datenlogger enthält als Einplatinencomputer einen *Raspberry Pi B+*, dieser soll durch die aktuelle Version *Raspberry Pi 3 Model B* ersetzt werden. Da beide Versionen baugleich und voll Pin-kompatibel zueinander sind, kann ein Austausch durchgeführt werden, ohne dass Hardwareänderungen an der Hauptplatine das HAW-Loggers notwendig sind. Der *Raspberry Pi 3 Model B* enthält eine 1.2GHz 64-bit quad-core ARMv8 CPU [8], damit erhöht sich die Zahl der CPU-Kerne auf 4 und die Taktrate von 700 MHz auf 1,2 GHz. Der Einplatinencomputer verfügt über die in Tabelle 5.1 enthaltenen Schnittstellen, die Betriebsspannung beträgt 5 V und es werden 5 V und 3,3 V über eine Stiftleiste bereitgestellt. Als Betriebssystem wird die Linux Distribution *Raspian Jessie Lite* verwendet, welche auf der *Debian Jessie* Distribution basiert [9].

Tabelle 5.1.: Schnittstellen des Einplatinencomputers

Schnittstelle	**Ausführung**	**Anzahl**	**Bemerkung**
USB 2.0	A-Buchse	4	
10/100 MBit-Ethernet	RJ45 Buchse	1	
2,4GHz WLAN b/g/n	-	1	
Bluetooth 4.1	-	1	
GPIO 3,3 V	Stiftleiste 2,54, 2x20	26	
I^2C	Stiftleiste 2,54	1	Bei Verwendung verringert sich die Anzahl der GPIO-Pins um 2
SPI	Stiftleiste 2,54	1	Bei Verwendung verringert sich die Anzahl der GPIO-Pins um 3
UART	Stiftleiste 2,54	1	Bei Verwendung verringert sich die Anzahl der GPIO-Pins um 2
HDMI	HDMI-Buchse	1	
DSI	ZIF 15	1	
CSI-2	ZIF 15	1	

5.2.2. Nutzer-Schnittstelle

Als Nutzer-Schnittstelle wird die Lösung des HAW-Datenloggers [33] übernommen. Hierbei wird mit einem USB-Wifi-Modul ein Drahtlosnetzwerk (WLAN) nach dem Standard *IEEE802.11 b/g/n* aufgebaut. Die Funktionsübersicht des verwendeten Moduls ist in Tabelle 5.2 darstellt. Das Modul erfüllt auch hier die Mindestanforderungen aus dem Systementwurf. Der verwendete *Raspberry Pi 3 Model B* verfügt bereits über ein integriertes WLAN-Modul,

welches nicht verwendet wird, da sich der Einplatinencomputer innerhalb des Aluminiumgehäuses befindet. Dies würde zu einer Reduzierung der Reichweite führen. Das USB-Wifi-Modul ist außerhalb des Gehäuses angebracht.

Tabelle 5.2.: Funktionsübersicht EDIMAX EW-7811UN (Übernommen aus [33], Tabelle 5.3)

Anforderung	**EDIMAX EW-7811UN**
Reichweite > 5m	innen 20 m
Schutz von Fremdzugriff	WEP 64/128, WPA, WPA2 und IEEE2.1x
Schnittstelle zum Einplatinencomputer	USB2.0
Temperaturbereich	0 -40 °C

Zur Realisierung der Schnittstelle wird mit dem verwendeten Modul durch Verwendung des Paketes *hostapd* ein Lokales Netzwerk aufgebaut. Hierdurch ist es möglich, sich mit einem beliebigen Computersystem, welches den Drahtlosstandard *IEEE 802.11 b/g/n* unterstützt, im Netzwerk anzumelden. Als Nutzer-Schnittstelle wird das Secure Shell (SSH)-Protokoll verwendet, welches einen entfernten Zugriff auf das System ermöglicht. Die Extraktion von Messdaten kann außerdem mit dem File Transfer Protocol (FTP) erfolgen. Die Eigenschaften des aufgebauten Netzwerkes sind in Tabelle 5.3 dargestellt.

Tabelle 5.3.: Eigenschaften des Drahtlosnetzwerkes

Eigenschaft	**Wert**
SSID	SensorboxServer
IPv4 Netzwerk	192.168.2.0
Netzwerkmaske	255.255.255.0
Host IPv4 Adresse	192.168.2.1
Verschlüsselungsart	WPA2
DHCP	nicht implementiert

Da für das Netzwerk kein Dynamic Host Configuration Protocol (DHCP)-Server implementiert ist, ist es notwendig, dass Klienten über eine statische IP-Adresse verfügen. Diese muss innerhalb des Netzwerks liegen. Die IP-Adressen 192.168.2.10/ .11/ .12 und .13 sind bereits reserviert und können nicht von anderen Klienten verwendet werden.

5.2.3. GNSS-Sensor

Als GNSS-Sensor wird ebenfalls die Lösung HAW-Danteloggers [33] verwendet, diese nutzt ein *SparkFun Venus GPS Modul* für die Positionsbestimmung. Als Schnittstelle zum Einplatinencomputer wird eine UART-Verbindung verwendet. Zur Ansteuerung und zum Auslesen

des Sensors kann das zur Verfügung stehende Programm *GPS_Logger* [33] verwendet werden. Die Messrate beträgt 10 Hz.

5.2.4. Geschwindigkeitssensor

Der HAW-Datenlogger verfügt über eine Schnittstelle für das Einlesen und Zählen von Drehzahl- oder Odometriepulsen. Aus diesen kann durch das Ableiten eine Geschwindigkeitsinformation berechnet werden. Da diese Schnittstelle für den Einsatz in Nahverkehrsbussen entwickelt wurde, kann sie nicht ohne Weiteres im Applikationsfahrzeug verwendet werden. Hier sind die Drehzahlpulse nicht direkt messbar. Daher wird die vorhandene Schnittstelle nicht verwendet.

Als Alternative zu dieser Methode soll das Geschwindigkeitssignal über den CAN-Bus gemessen werden. Damit die Aufzeichnung der CAN-Busdaten während der Fahrt möglich ist, ohne dass das Fahrzeug die Straßenzulassung verliert, muss das Bussignal zuvor entkoppelt werden. Hierfür wird ein *Sensor-Anpassungsmodul zur CAN Bus Isolation* (SAM-ISO011-23A0) von IPETRONIK verwendet. Dieser Koppler ermöglicht ein ausschließliches Lesen des Controller Area Network (CAN)-Busses und besitzt eine allgemeine Betriebserlaubnis für PKWs. Der Bus wird über die OBDII-Schnittstelle im Fahrzeug abgegriffen. Das Signal wird dann von einer *IXXAT USB-to-CAN V2*-Schnittstelle in ein USB-Signal umgewandelt.

5.2.5. Speichereinheit

Als Speichereinheit wird die Speicherkarte verwendet, die auch das Betriebssystem des Einplatinencomputers enthält. Es handelt sich hierbei um eine *micro SDHC*-Karte der Klasse 10 mit einer Speichergröße von 16 GB. Abzüglich des Speichers, der für das Betriebssystem verwendet wird, stehen ca. 13 GB zur Verfügung. Die Schreib- und Lesegeschwindigkeit beträgt mindestens 10 MB/s [28]. Das Datenaufkommen ist in Tabelle 5.17 aufgeführt und beträgt 28,4 kB/s pro Sensoreinheit. Außerdem werden vom verwendeten GNSS-Sensor 14,45 kb/s erzeugt [33]. Damit ist die Schreibgeschwindigkeit ausreichend. Um die Rohdaten über einen Zeitraum von einer Stunde speichern zu können wird ein Speicherplatz von

$$D = \left(4 \cdot 28,4\frac{kB}{s} + 14,45\frac{kB}{s}\right) \cdot 3600s = 460,9MB \tag{5.1}$$

benötigt. Die Speicherkarte besitzt somit genug Speicherplatz, um die aufkommenden Daten speichern zu können.

5.2.6. Gesamtsystem

Das Blockschaltbild der zentralen Einheit ist in Abbildung 5.4 dargestellt. Als Hauptkomponente dient der Einplatinencomputer, welcher die Kommunikation mit allen Komponenten übernimmt. Die Nutzerschnittstelle, realisiert durch den *EDIMAX EW-7811UN*, wird über eine USB2.0-Schnittstelle mit dem Einplatinencomputer verbunden und generiert das Drahtlos-Netzwerk. Ebenfalls über diese Schnittstelle ist der CAN-USB-Umsetzer verbunden, dieser ist zusätzlich über einen CAN-Koppler vom PKW entkoppelt, so dass Störungen auf dem CAN-Bus des PKWs ausgeschlossen werden. Außerdem ist eine GNSS-Antenne nach außen geführt, die mit dem GNSS-Sensor verbunden ist. Zur Wahrung der Übersicht enthält das Blockschaltbild nur die für das Sensorsystem relevanten Komponenten.

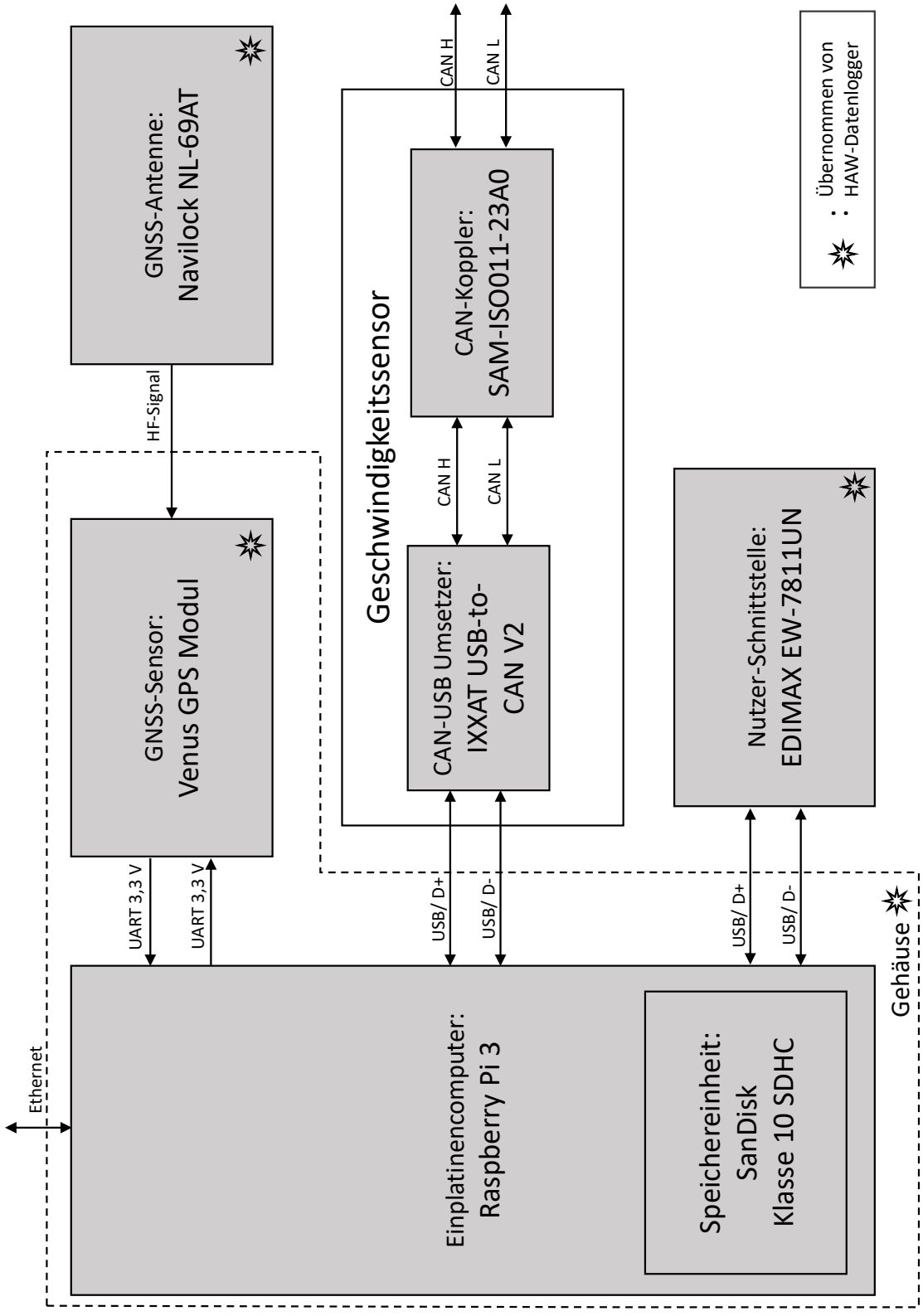

Abbildung 5.4.: Blockschaltbild der zentralen Einheit

5.2.7. Prototyp 1

Im Rahmen dieser Thesis wird ein erster Prototyp entwickelt, mit dem die grundsätzliche Funktionalität des Gesamtsystems überprüft werden kann. Es sieht zunächst keine Echtzeitberechnung von Position und Geschwindigkeit vor, diese soll erst in einer Nachbearbeitung erfolgen, um so Abschätzungen zur Realisierbarkeit und Genauigkeit treffen zu können. Der erste Prototyp unterscheidet sich zum entwickelten Gesamtsystem in den folgenden Punkten:

- Berechnung von Geschwindigkeits- und Positionsinformationen erfolgen in einer Nachbearbeitung.
- Aufzeichnung der CAN-Daten erfolgt auf einem externen Computersystem.

5.2.8. Software

Für die Realisierung der Software werden verschiedene Softwarekomponenten benötigt, diese sind in Tabelle 5.4 zusammen mit den relevanten Anforderungen aus den Tabellen 3.1 und 3.2, sowie weiteren Anforderungen aus dem Systementwurf, aufgelistet. Außerdem ist hier die Realisierung für den ersten Prototypen enthalten.

Tabelle 5.4.: Software in der zentralen Einheit

Softwarekomponente	**relevante Anforderungen**	**Realisierung für Prototyp 1**
Datenaufnahme	Ausgabeverzögerung, Datenaufzeichnung	C++ Programm: *Server*
Datenverarbeitung	Ausgabeverzögerung, Position, Geschwindigkeit, Ausgaberate	Nachbearbeitung (MATLAB)
Referenzposition	Referenzsystem Position	C Programm: *GPS_Logger* aus [33]
Referenzgeschwindigkeit	Referenzsystem Geschwindigkeit	externe Datenaufnahme
Zeitsynchronität	Zeitsynchronität der Funktionskomponenten (Abschnitt 4.2)	Linux Daemon: PTPd

Datenaufnahme

Das Programm zur Datenaufnahme trägt den Namen *Server* und ist in der Programmiersprache C++ geschrieben. Dieses hat die Aufgabe, einen Server im LAN bereitzustellen, an den die Sensoreinheiten die aufgenommenen Messwerte senden können. Diese sollen dann, in Voraussicht auf folgende Prototypen, für andere Programme zu Verfügung stehen. Außerdem findet hier eine Synchronitätsprüfung der empfangenen Messwerte statt. Um die

Zeitsynchronität zu überprüfen ist es notwendig, dass die übertragenen Messwerte einen Zeitstempel enthalten.

Das Programm besteht aus mehreren Klassen, die in Tabelle 5.5 aufgelistet sind und kurz beschrieben werden. Eine ausführliche Beschreibung erfolgt in den folgenden Abschnitten. Der Quellcode ist im Anhang B.2.1 enthalten.

Tabelle 5.5.: Übersicht der Klassen des Programms zur Datenaufnahme

Klassenname	**Beschreibung**
Server	Klasse zur Erzeugung von Serverobjekten für den Aufbau von TCP/ IP Verbindungen. Die empfangenden Daten werden in einer PIPE bereitgestellt.
SensorboxServer	Klasse zur Verarbeitung von 4 Sensoreinheiten. Hierzu zählt die Ansteuerung der Sensoreinheiten, das Speichern der Rohdaten, Prüfen der Synchronität und Bereitstellen der Messdaten in Ring-Puffern.
MeasuringPoint	Abstrakte *Interface*-Klasse, für Messpunkte. Definiert die Nutzung des Zeitstempels.
MeasurePoint_Laser	Diese Klasse erbt die Eigenschaften von *MeasuringPoint* und fügt einen LIDAR-Messwert hinzu.
MeasurePoint_Sonar	Diese Klasse erbt die Eigenschaften von *MeasuringPoint* und fügt eine Ultraschallmessung hinzu.

Klasse: Server: Diese Klasse erzeugt Objekte vom Typ *Server*, das Klassendiagramm ist in Abbildung B.1 dargestellt. Als Vorlage hierfür wird die Literatur von Jürgen Wolf [35] verwendet. Der Server wird mit Internet-Sockets realisiert, welcher dann über eine beliebige (freie) Portadresse angesprochen werden kann. Durch die Methode *Server::initialise(port)* wird ein Internet-Sockel erzeugt, an die übergebene Port-Nummer gebunden und eine namenlose Pipe (FIFO) zur Bereitstellung der empfangenen Daten erstellt. Außerdem wird der Server so konfiguriert, dass dieser TCP-Verbindungen aufbaut. Die Kommunikation erfolgt durch ASCII-Zeichen. Die Bearbeitung von Anfragen wird in der Methode *Server::start()* durchgeführt, der Programmablaufplan ist in Abbildung 5.5 dargestellt. Diese übernimmt sowohl dem Aufbau von Klienten-Verbindungen, als auch den Empfang von Daten, die von den verbundenen Klienten an den Server gesendet werden. Aktives Warten (polling) wird durch die Verwendung von synchronen IO-Multiplexen verhindert. Durch die Funktion *select()* wird das Programm solange blockiert, bis ein Sockel-Deskriptor bereit zum Auslesen ist. Die zu betrachtenden Deskriptoren müssen hierfür einer Instanz *FD_SET* des Datentyps *fd_set* hinzugefügt werden, die dann bei Aufruf der *select*-Funktion übergeben wird. Die Menge *FD_SET* beinhaltet zunächst nur den Deskriptor des Server-Sockels. Dieser gibt mit der Bereitschaft zum Auslesen die Verbindungsanfrage eines Klienten an. Der Server reagiert mit dem Verbindungsaufbau, erzeugt einen individuellen Klienten-Sockel und speichert IP-Adresse und Deskriptor des Klienten. Der neu erstellte Deskriptor wird außerdem der Menge *FD_SET* hinzugefügt, so dass das Empfangen von Daten dieses Klienten ebenfalls zum Verlassen

der *select()*-Funktion führt. Ist dies der Fall erfolgt die Klienten-Bearbeitung, die in Abbildung 5.6 dargestellt ist.

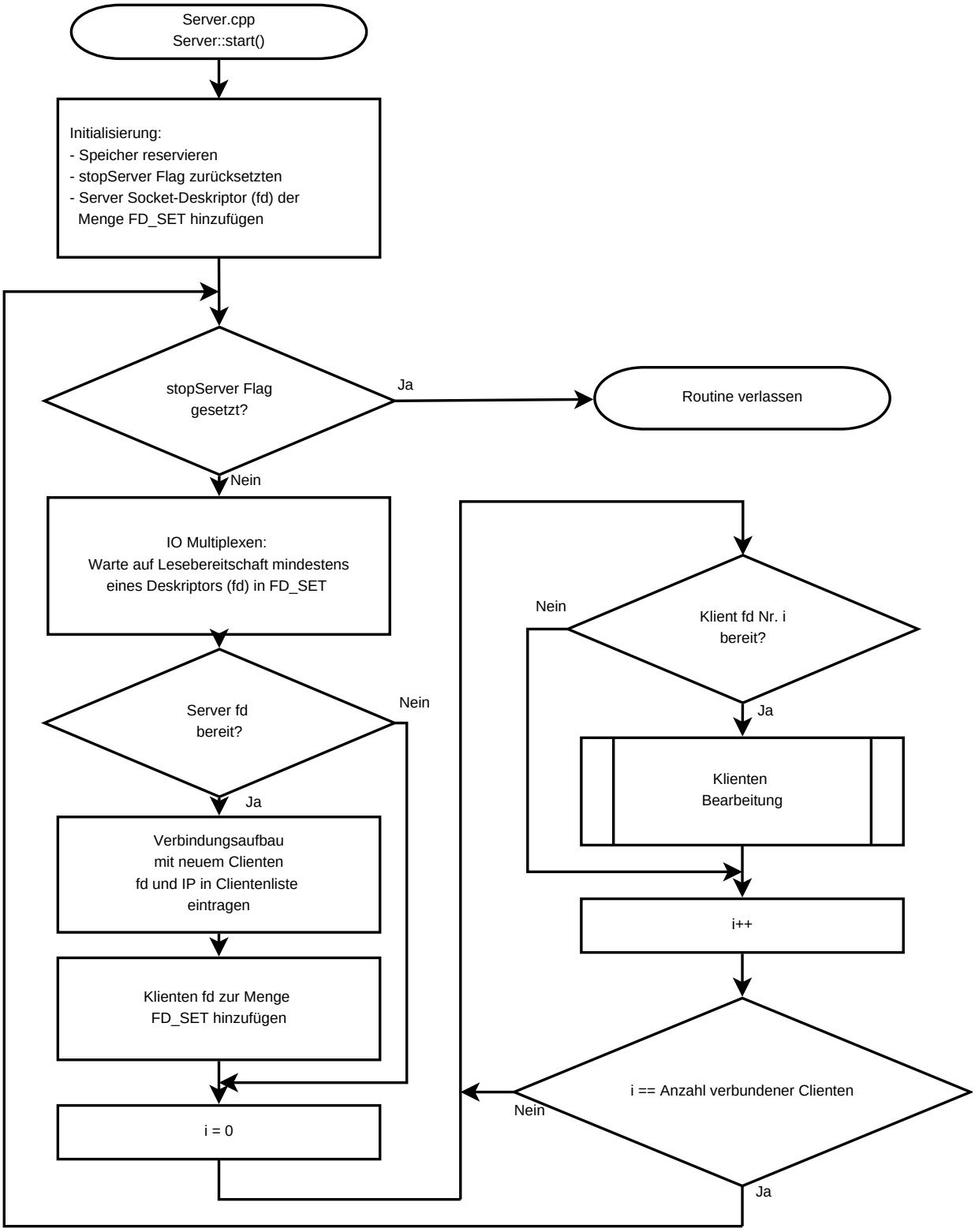

Abbildung 5.5.: Programmablaufplan der Methode Server::start()

Die Kommunikation zwischen Server und Klient basiert auf Nachrichten. Eine Nachricht soll beispielsweise einem Messpunkt der Sensoreinheit entsprechen. Da die Länge einer TCP-Nachricht auf maximal 1500 Byte begrenzt ist, können die folgenden Fälle auftreten:

- Fall 1: Eine Nutzernachricht in einer TCP-Nachricht (Nutzernachricht $\leq$ 1500 Byte)
- Fall 2: Mehrere Nutzernachrichten in einer TCP-Nachricht (Nutzernachricht $\leq$ 1500 Byte)
- Fall 3: Eine Nutzernachricht in mehreren TCP-Nachrichten (Nutzernachricht $>$ 1500 Byte)

Als Trennungszeichen für Nutzernachrichten wird ein Zeilenumbruch (\r\n) verwendet. Die Klienten-Bearbeitung hat somit die Aufgabe, Nutzernachrichten von Klienten aufzulösen. Zunächst aber erfolgt eine Prüfung, ob die empfangene Nachricht Daten enthält. Ist dies nicht der Fall, symbolisiert dies, dass der Klient die Verbindung abgebrochen hat. Daraufhin wird der Name entfernt und der Deskriptor aus der Menge *FD_SET* gelöscht. Empfangene Daten werden zunächst auf einen Zeilenumbruch geprüft. Ist dies nicht der Fall, so erfolgt zunächst eine Zwischenspeicherung der Daten. Solange kein Zeilenumbruch in den Daten enthalten ist werden diese an die bereits gespeicherten Daten angehängt. Wird ein Zeilenumbruch in den Daten erkannt, so wird dieser Teil an die zwischengespeicherten Daten angehängt und in die Pipe geschrieben. Daraufhin werden die übrigen Daten erneut auf einen Zeilenumbruch geprüft und entsprechend behandelt, bis alle empfangenen Daten entweder in die Pipe geschrieben oder zwischengespeichert sind. Auf diese Weise werden alle möglichen Fälle behandelt und nur vollständige Nachrichten in die Pipe geschrieben.

Der laufende Server kann durch den Aufruf der Methode *Server::stop()* beendet werden. Dies schließt alle Verbindungen. Außerdem verfügt die Klasse Methoden zur Ausgabe der IP-Adressen aller verbundenen Klienten, sowie zum Senden von Daten an die Klienten.

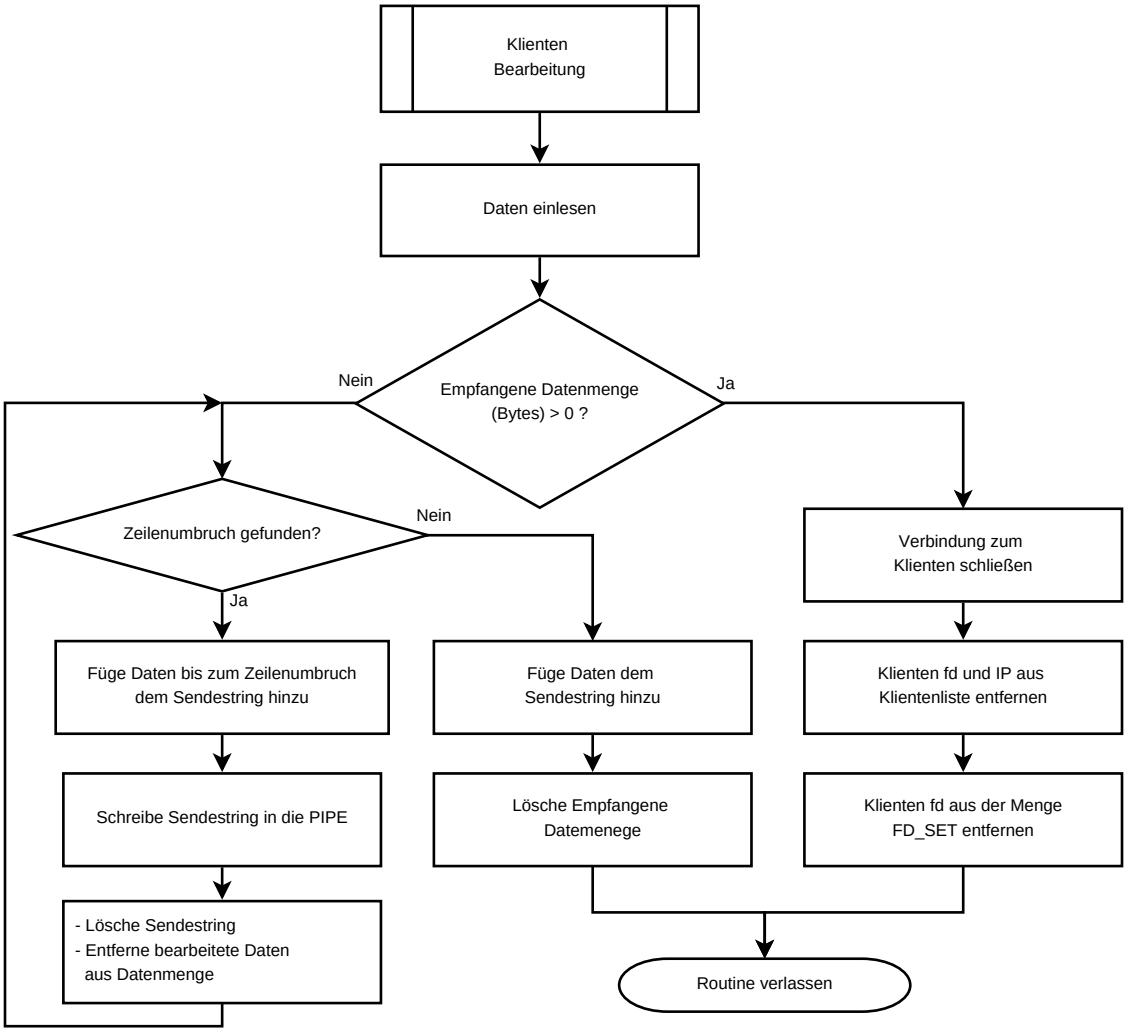

Abbildung 5.6.: Programmablaufplan für die Klienten-Bearbeitung

Klasse: SensorboxServer: Diese Klasse *SensorboxServer* übernimmt die meisten Aufgaben der Datenaufnahme. Hierzu zählt die Ansteuerung der Sensoreinheiten und die Verarbeitung der gesendeten Messdaten. Das Klassendiagramm ist in Abbildung B.2 dargestellt, hierbei ist zunächst eine Übertragung von Ultraschall- und LIDAR-Messwerten über je eine TCP-Verbindung vorgesehen. Hinzu kommt ein zusätzlicher Server, über den Befehle gesendet werden. Jeder Server ist an einen eigenen Port gebunden, welcher bei Aufruf des Konstruktors übergeben werden. Außerdem übernimmt diese Klasse das Speichern der Rohdaten und die Bereitstellung der Messwerte innerhalb des Programms. Für diesen Zweck verfügt sie über acht Ringpuffer, die die Messwerte (Sonar und LIDAR) jeder Sensoreinheit enthalten.

Diese Funktionen werden in der Methode *SensorboxServer::start()* realisiert, deren Ablauf-

plan in Abbildung 5.7 dargestellt ist. Nach der Initialisierung folgt das Starten der drei Server für LIDAR-, Ultraschallwerte und Befehle in jeweils einem Thread. Ab diesem Zeitpunkt können Klienten, also die Sensoreinheiten, eine Verbindung zu den Servern aufbauen. Die angemeldeten Sensoreinheiten werden nun identifiziert, indem zunächst die IP-Adressen der verbundenen Klienten, beziehungsweise Sensoreinheiten vom Befehlsserver angefordert werden. Nun wird an jede dieser Adressen der Befehl *Identify* gesendet, welcher von der Sensoreinheit mit dem eigenen Namen (0, 1, 2 oder 3) beantwortet wird. Die Identifizierung erfolgt sekündlich, bis alle vier Sensoreinheiten mit den Servern verbunden sind. Daraufhin wird der Befehl *StartMeasurement* an alle Sensoreinheiten gesendet, so dass der Messvorgang startet. Die von den Sensoreinheiten unterstützen Befehle sind in Tabelle 5.16 aufgeführt. Auch hier erfolgt das Warten auf Daten durch die Verwendung von synchronen IO-Multiplexen, was hier auf eine Bereitschaft zum Auslesen der Server-Pipes angezeigt wird. Darauf folgt das Einlesen und Verarbeiten der empfangenen Daten. Das Einlesen erfolgt zeilenweise und beginnt mit der Umwandlung der Daten in Form eines Objektes der Klasse *std::string* in ein Objekt der Klasse *MeasurePoint_Laser* oder *MeasurePoint_Sonar*. Diese werden lokal in einer Datei gespeichert und in den einheitenspezifischen Puffer geschrieben.

Für jede Sensoreinheit existieren zwei Ringpuffer, einen für LIDAR- und einer für Ultraschallwerte. Damit die Messdaten der verschiedenen Sensoreinheiten ausgewertet werden können, müssen die Daten in den Puffern zeitsynchron sein. Daher werden nach jedem Speichern die Zeitstempel der Messwerte miteinander verglichen. Wird beispielsweise ein LIDAR-Messpunkt von Sensoreinheit 0 in das Pufferelement 10 geschrieben, so folgt danach ein Vergleich der Zeitstempel aller vier LIDAR-Puffer von Element 10. Ergibt sich hierbei eine größere Zeitdifferenz als die Abtastrate der Sensoren, wird ein Fehlerzähler (*SyncErrorCtr*) inkrementiert. Überschreitet dieser Zähler den Wert *MAX_SYNC_ERROR*, werden alle vier Puffer zurückgesetzt.

Dieser Vorgang wird wiederholt, bis die Pipe keine Daten mehr enthält. Ein Abbruch der Routine kann durch den Aufruf der Methode *SensorboxServer::stop()* erreicht werden, wodurch der Indikator *stopServer* gesetzt wird.

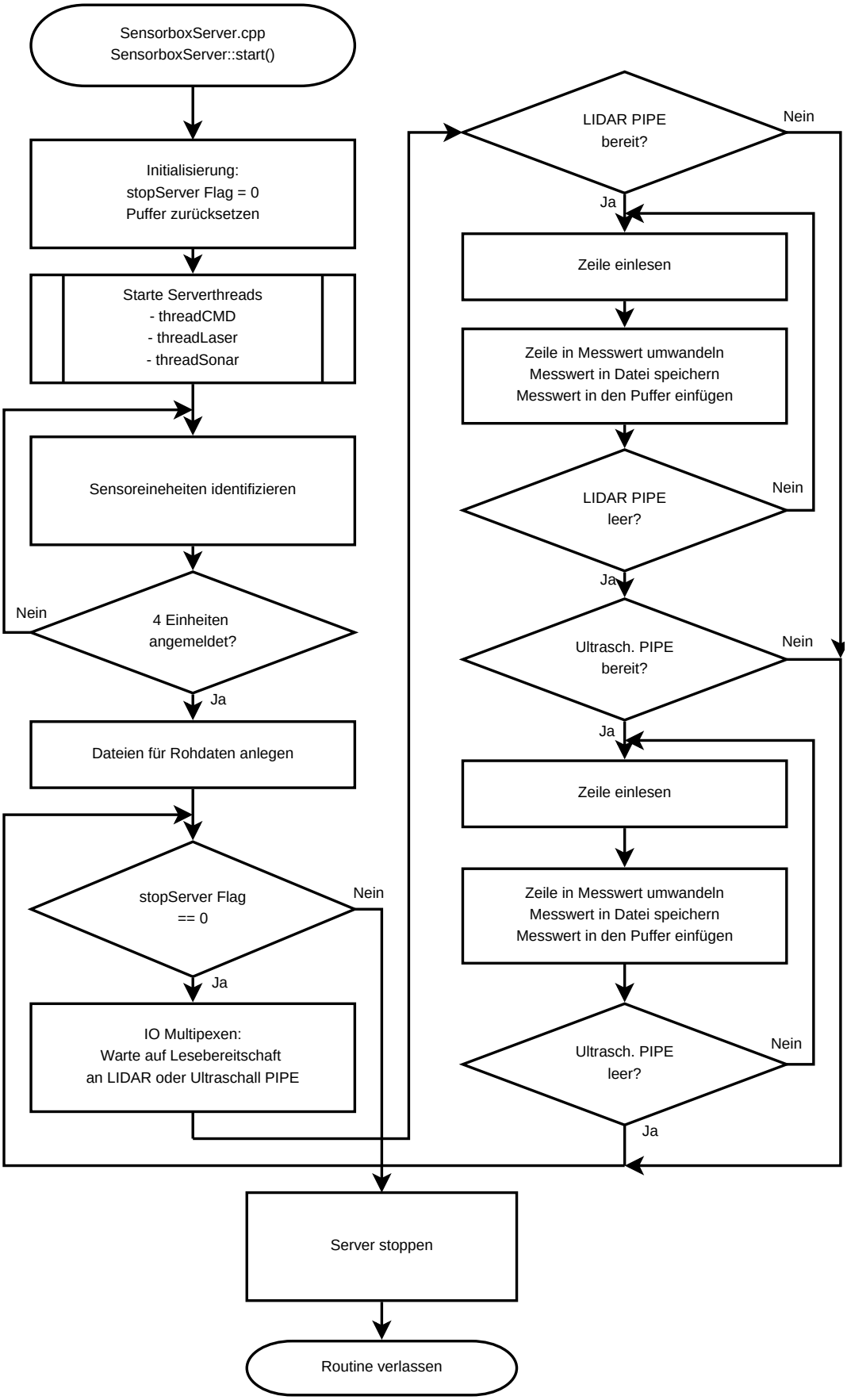

Abbildung 5.7.: Programmablaufplan der Methode SensorboxServer::start()

Klassen: MeasuringPoint, MeasurePoint_Laser und MeasurePoint_Sonar: In diesen Klassen sind Messpunkte definiert. Einer besteht aus einem Zeitstempel und dem Messwert. *MeasureingPoint* ist eine abstrakte Klasse, die den grundsätzlichen Aufbau eines Messpunktes und den Zeitstempel definiert. Dieser hat eine Auflösung von 1 ms und wird als *unsigned long long* absolut als UNIX-Zeit gespeichert. Die Klasse *MeasurePoint_Laser* erbt die Eigenschaften von *MeasuringPoint* und fügt den Messwert vom Typ *float* hinzu. Gleiches gilt für die Klasse *MeasurePoint_Sonar*, nur dass hier ein Messwert aus einem *std::array* mit Elementen vom Typ *int* besteht.

Die Klassen besitzen eine Methode *getString()*, welche aus einem Zeitstempel und Messwert ein Objekt vom Typ *std::string* erzeugt. Dieses kann dann zur Übertragung über die TCP-Verbindung genutzt werden. Es stehen drei Konstruktoren zur Erzeugung eines Objekt zu Verfügung, ein Aufruf ohne Parameter generiert einen leeren Messpunkt. Der Konstruktor kann auch mit einem Zeitstempel als Parameter aufgerufen werden, um diesen direkt festzulegen. Zuletzt kann ein Messpunkt aus einem *std::string* erzeugt werden, der durch die *getString()* Methode erstellt wurde. Der Messpunkt enthält dann Zeitstempel und Messwert. Die Klassendiagramme sind im Anhang B.1 dargestellt.

Routine: Main: Die Hauptroutine erzeugt zunächst ein Objekt vom Typ SensorboxServer mit den Port-Nummern für die drei Server. Die verwendeten Port-Nummern sind in Tabelle 5.6 aufgelistet. Das Objekt wird nun in einem eigenen Thread durch Aufruf der Methode *ServerboxServer::start()* gestartet. Die Routine gibt dann sekündlich die aktuellen LIDAR-Werte der vier Sensoreinheiten und gegebenenfalls Fehlermeldungen aus.

Tabelle 5.6.: Port-Nummern der Server

Server	Port-Nummer
LIDAR	1475
Ultraschall	1476
Befehle	1477

Datenverarbeitung

Eine Echtzeit-Datenverarbeitung findet im ersten Prototyp nicht statt.

Referenzposition

Als Programm für die Referenzposition wird das im HAW-Datenlogger [33] verwendete Programm *GPS-Logger* verwendet. Zur Anpassung an das System und die neue Hardware werden die folgenden Änderungen vorgenommen:

- Zielverzeichnis der Rohdaten auf */home/pi/DataGPS/* geändert
- Serielle Schnittstelle (UART) von */dev/ttyAMA0* auf */dev/serial0* geändert
- Programmaufruf des Unterprogramms zur Auswertung der Rohdaten entfernt
- Temperatur und Luftdruckmessung deaktiviert

Referenzgeschwindigkeit

Zur Realisierung des ersten Prototyps werden die CAN-Daten auf einem externen Windows 10 Rechner aufgenommen. Hierfür wird das im Treiber des *IXXAT USB-to-CAN V2* enthaltende Programm *canAnalyzer3 mini* verwendet.

Zur Bestimmung der Geschwindigkeit des Fahrzeuges muss diese aus allen CAN-Nachrichten herausgefiltert werden. Da die hierfür benötigte CAN-Matrix für das verwendete Applikationsfahrzeug nicht öffentlich verfügbar ist, ist eine experimentelle Analyse erforderlich. Als Ausgangspunkt wird eine online verfügbare Tabelle [15] verwendet. Bezüglich der Fahrzeuggeschwindigkeit kann der Inhalt der Tabelle in einem eigenen Versuch bestätigt werden. Es handelt sich um einen *High Speed CAN-Bus* mit einer Übertragungsrate von 500 kb/s. Die zur Bestimmung der Geschwindigkeit relevanten Daten sind in Tabelle 5.7 dargestellt.

Tabelle 5.7.: Geschwindigkeitssignal in CAN-Daten

CAN Identifier	Nachrichten-Bytes
0x201	4,5

Die Geschwindigkeit in Kilometer pro Stunde kann durch die Gleichung

$$v = \frac{(b_4 \cdot 2^8) + b_5}{100 \frac{h}{km}} \tag{5.2}$$

berechnet werden. Die Geschwindigkeit wird so mit einer Auflösung von 0.01 km/h dargestellt. Eine Aussage über die Genauigkeit kann nicht getroffen werden. Die Rate des Geschwindigkeitssignals beträgt 100 Hz.

Zeitsynchronität

Damit alle Funktionseinheiten des Sensorsystems mit der gleichen Systemzeit arbeiten, wird der Linux *Daemon PTPd* [25] verwendet. Dieser implementiert das Precision Time Protocol (PTP) der Version 2, welches eine präzise Zeitsynchronität, zwischen den in einem LAN befindlichen Geräten, ermöglicht. Bei Softwarelösungen werden Sychronisationsgenauigkeiten zwischen 10 - 100 μs erreicht [31]. Das Programm wird beim Bootvorgang des Einplatinencomputers automatisch gestartet. Die zentrale Einheit übernimmt im System die Aufgabe des *Masters*.

5.2.9. Gehäuse

Als Gehäuse für die zentrale Einheit wird das Gehäuse des HAW-Datenloggers verwendet. Hierbei wird zusätzlich die Ethernet-Buchse nach außen geführt.

5.3. Sensoreinheit

5.3.1. Einplatinencomputer

Als Einplatinencomputer der Sensoreinheit wird, wie für die zentrale Einheit, ein *Raspberry Pi 3 Model B* mit gleicher Distribution verwendet. Dieser verfügt somit ebenfalls über die in Tabelle 5.1 aufgeführten Schnittstellen.

5.3.2. Debug-Schnittstelle

Als Debug-Schnittstelle wird das integrierte WLAN-Modul des Einplatinencomputer verwendet. Hierfür wird das von der zentralen Einheit bereitgestellte Drahtlosnetzwerk genutzt. Jede Sensoreinheit meldet sich im Netzwerk *SensorboxServer* mit einer festen IP-Adresse an und ist somit ebenfalls über das SSH-Protokoll erreichbar. Die Adressen der Sensoreinheiten sind in Tabelle 5.8 dargestellt.

Tabelle 5.8.: Netzwerkadressen der Sensoreinheiten

Name	IP-Adresse
Sensoreinheit 0	192.168.2.10
Sensoreinheit 1	192.168.2.11
Sensoreinheit 2	192.168.2.12
Sensoreinheit 3	192.168.2.13

5.3.3. LIDAR-Sensor

Als LIDAR-Sensor wird der *SF02/F Rangefinder* von *LightWare Optoelectronics Ltd.* verwendet (Abbildung 5.8). Das Modul beinhaltet alle für eine LIDAR-Messung benötigten optischen und elektrischen Komponenten, sowie die Software für eine *time-of-flight*-Messung [22]. Die relevanten Eigenschaften sind in Tabelle 5.9 dargestellt.

Tabelle 5.9.: Spezifikation SF02/F Rangefinder

Spezifikation	**Wert**	**Bemerkung**
Operationsbereich	0 m - 50 m	
Messrate	32 Hz	Nicht einstellbar
Laserklasse	1M	
Wellenlänge	905 nm (Infrarot)	Nicht sichtbar für das menschliche Auge
Öffnungswinkel	0.2°	
Messauflösung	1 cm	
Messgenauigkeit	± 5 cm	Abhängig von Distanz, siehe Figure 20 im Datenblatt [22]
Extern Triggerbar	nein	
Schnittstellen	UART, I^2C, USB, analog	
Betriebsspannung	5 V oder 6,5 V - 9 V	
Arbeitstemperatur	0 ... 40°C	

Die Messrate kann nun in die im Systementwurf entwickelten Gleichungen 4.3 und 4.9 eingesetzt werden. Bei Schrittgeschwindigkeit ist so, im Fall der höchsten Geschwindigkeit eine räumliche Auflösung von

$$\Delta x_{max} = \frac{v_{max}}{f_{mess}} = \frac{1\frac{m}{s}}{32Hz} = 3,125cm \tag{5.3}$$

zu erreichen. Die maximal messbare Geschwindigkeit, definiert durch die Messperiode und dem Abstand r_A, kann ebenfalls berechnet werden.

$$v'_{max} = \frac{r_A}{T_{mess}} = \frac{0.51m}{31,25ms} = 16\frac{m}{s} \tag{5.4}$$

Messauslösung

Der Sensor bietet verschiedene Möglichkeiten zur Anforderung eines Messwertes, eine periodische Messwertausgabe ist nicht vorgesehen. Die Messwertanforderung kann direkt über die verwendete Schnittstelle oder über einen Trigger-Pin erfolgen. Allerdings handelt es sich hierbei nicht um eine Auslösung der Messung, sondern um eine Anforderung des aktuellen

Messwerts. Daher ist es nicht möglich, auf diese Weise eine Information über die Aktualität der Messung zu erhalten. Aus diesem Grund stellt der Sensor ein *Timer-Sync*-Signal zur Verfügung, welches mit einer steigenden Flanke signalisiert, dass ein Messwert zur Verfügung steht. Dieser kann dann über eine beliebige Schnittstelle abgefragt werden.

Abbildung 5.8.: SF02/F Rangefinder von *LightWare Optoelectronics Ltd.* Bild aus Datenblatt [22]

Abbildung 5.9.: XL-MaxSonar - AE Serie von *MaxBotix Inc.* Bild aus Datenblatt [16]

5.3.4. Ultraschallsensor

Als Ultraschallsensor wird ein *XL-Maxsonar-AE MB1300* Sensor [16] verwendet. Dieser ist in Abbildung 5.9 dargestellt und besitzt die in Tabelle 5.10 aufgeführten Eigenschaften.

Tabelle 5.10.: Spezifikation XL MaxSonar-AE-MB1300

Spezifikation	**Wert**	**Bemerkung**
Operationsbereich	0,2 m - 10,68 m	
Messrate	10 Hz	maximal
Extern Triggerbar	ja	
Schnittstellen	Digital (UART) Amplitudenanalog	Das amplitudenanaloge Signal wird in Form einer Analogspannung bereitgestellt
Auflösung	1 cm	Nur digitale Schnittstelle
Genauigkeit	keine Angabe	Abhängig von Temperatur und Luftdruck
Betriebsspannung	3,3 V - 5 V	
Arbeitstemperatur	-40 °C bis 65 °C	0 °C bis 40 °C empfohlen

Der Sensor ermöglicht eine amplitundenanaloge Messung. Diese wird allerdings in Form einer Analogspannung bereitgestellt und muss somit digitalisiert werden, um eine Auswertung auf dem Einplatinencomputer zu ermöglichen. Für die benötigte Analog / Digitalwandlung wird ein *High-Precision AD/DA Board* von *Waveshare* [6] verwendet. Hierbei handelt es sich um eine zum *Raspberry Pi 3 B+* passende Aufsatzplatine, welche einen *ADS1256* [20] zur A/D-Wandlung und einen *DAC8532* zur D / A-Wandlung von *Texas Instruments* verwendet. Für die Aufnahme des Sensorsignals wird ausschließlich der A / D-Wandler verwendet, dessen relevante Spezifikationen in Tabelle 5.11 dargestellt sind.

Tabelle 5.11.: Spezifikation ADS1256 im Aufbau *High-Precision AD/DA Board*[6]

Spezifikation	**Wert**	**Bemerkung**
Schnittstelle	SPI	
Auflösung	24 Bit	
Abtastrate	30000; 15000; 7500; 3750; 2000; 1000; 500; 100; 60; 50; 30; 25; 15; 10; 5; 2,5 SPS	einstellbar
Anzahl Kanäle	8	
Analog Spannungsbereich	-5 V bis 5 V	
Arbeitstemperatur	-40°C bis 105°C	

Das Ausgangssignal des Sensors wird mit dem Kanal 1 des *ADS1256* verbunden. Für die Kommunikation mit dem Einplatinencomputer wird die SPI-Schnittstelle verwendet. Der Einplatinencomputer übernimmt hierbei die Rolle des Masters und der *ADS1256* die Rolle des Slaves. Es werden fünf Signale benötigt:

- SPI MISO: Signalleitung für Kommunikation vom Slave zum Master.
- SPI MOSI: Signalleitung für Kommunikation vom Master zum Slave.
- SPI CLK: Taktsignal für die Datenübertragung, bereitgestellt vom Master.
- CS: *Low*-Pegel am *ADS1256* aktiviert SPI-Schnittstelle.
- DRDY: Eine fallende Flanke gibt an, dass ein neuer Messwert vorliegt.

Da die Distanz zu echogebenden Objekten durch die doppelte Schalllaufzeit (Hin- und Rückweg) gegeben ist, definiert die Abtastrate des amplitudenanlogen Signals die mögliche Auflösung der Distanzmessung. Mit einem Skalierungsfaktor $k = 58\frac{\mu s}{cm}$ [16] kann ein Zeitpunkt t_1 ab Messbeginn t_0 in eine Distanz r überführt werden.

$$r = \frac{t_1 - t_0}{k} \tag{5.5}$$

Die Auflösung der Distanzmessung Δr ist somit abhängig von der Abtastrate f_A des *ADS1256* und kann für die verwendeten Rate von 7500 SPS berechnet werden. Eine höhere Abtastrate ist nicht möglich (siehe Abschnitt 5.3.7).

$$\Delta r = \frac{\frac{1}{f_A}}{k} = \frac{1}{f_A \cdot k} = \frac{1}{7500 s^{-1} \cdot 58 \frac{\mu s}{cm}} = 2,3 cm \tag{5.6}$$

5.3.5. Stereo-Kamera

Die Realisierung der Stereo-Kamera wird im Rahmen einer Studienarbeit durchgeführt [29]. Diese wird durch zwei *DELOCK 96371* USB-Kameras realisiert, die mit einem Abstand von 21 cm parallel zueinander an der Sensoreinheit befestigt sind. Die Spezifikationen der Kamera sind in Tabelle 5.12 aufgeführt.

Tabelle 5.12.: Spezifikation DELOCK 96371

Spezifikation	**Wert**	**Bemerkung**
Auflösung	5,05 Megapixel	2592 x 1944 Pixel
Schnittstelle	USB	
Fokus	0,3 m - unendlich	Fixfokus
Filter	Kein Infrarotfilter	
Bildwinkel	48°	
Temperaturbereich	0°C - 50°C	

Zur Aufnahme der Bilder wird die API *Video4Linux* verwendet. Da die Kamera keinen Infrarotfilter besitzt, ist sie in der Lage, das ausgesendete Licht des LIDAR-Sensors aufzunehmen. Die Kamera ist so konfiguriert, dass sie Schwarzweißbilder mit einer Auflösung von 640 x 480 Pixel aufnimmt.

5.3.6. Gesamtsystem

Grundlage der Sensoreinheit ist der Einplatinencomputer, welcher die Ansteuerung und das Auslesen der Sensoren übernimmt. Außerdem werden dort die Messwerte verarbeitet und an die zentrale Einheit gesendet. Die Verteilung der Schnittstellen des Einplatinencomputers ist in Tabelle 5.1 dargestellt. Da bei dem verwendeten *LIDAR*-Sensor keine externe Triggerung möglich ist, wird das am Sensor zur Verfügung stehende *Timer-Sync* Signal zur Auslösung aller Messungen verwendet. Dieses wird an einen GPIO-Pin des Einplatinencomputers angeschlossen.

Tabelle 5.13.: Interne Schnittstellenverteilung der Sensoreinheit

Schnittstelle	Ausführung	Anzahl	Komponenten
USB 2.0	A-Buchse	4	Delock 96371 (2x)
10/100 MBit-Ethernet	RJ45 Buchse	1	Verbindung zur zentralen Einheit
2,4GHz WLAN b/g/n	-	1	Debug-Schnittstelle
Bluetooth 4.1	-	1	-
GPIO 3,3 V	Stiftleiste 2,54, 2x20	26	Ultraschall-Trigger, LIDAR-Trigger
I^2C	Stiftleiste 2,54	1	-
SPI	Stiftleiste 2,54	1	ADS1256
UART	Stiftleiste 2,54	1	SF02/F Rangefinder
HDMI	HDMI-Buchse	1	-
DSI	ZIF 15	1	-
CSI-2	ZIF 15	1	-

Das Blockschaltbild der Sensoreinheit ist in Abbildung 5.10 dargestellt. Die Sensoren sind entsprechend der Tabelle 5.13 angeschlossen. Zur Wahrung der Übersichtlichkeit wird auf die Darstellung der Energieversorgung der einzelnen Module verzichtet. Die verwendeten Betriebsspannungen sind in Tabelle 5.14 dargestellt, alle Module teilen sich eine gemeinsame Masse.

Tabelle 5.14.: Betriebsspannungen der Module

Modul	Betriebsspannung
SF02/F Rangefinder	5 V
XL_MaxSonar	3,3 V
ADS1256	5 V
Delock 96371	5 V (USB)

Die Energieversorgung der Sensoreinheit erfolgt über das Power over Ethernet (PoE)-Verfahren. Da der verwendete Einplatinencomputer dieses Verfahren nicht unterstützt, wird ein PoE-Splitter verwendet. Alternativ kann anstatt des PoE-Verfahrens auch ein USB-Akkumulator zur Energieversorgung der Einheit verwendet werden, diese erfolgt dann über die *Micro*-USB-Schnittstelle des Einplatinencomputers.

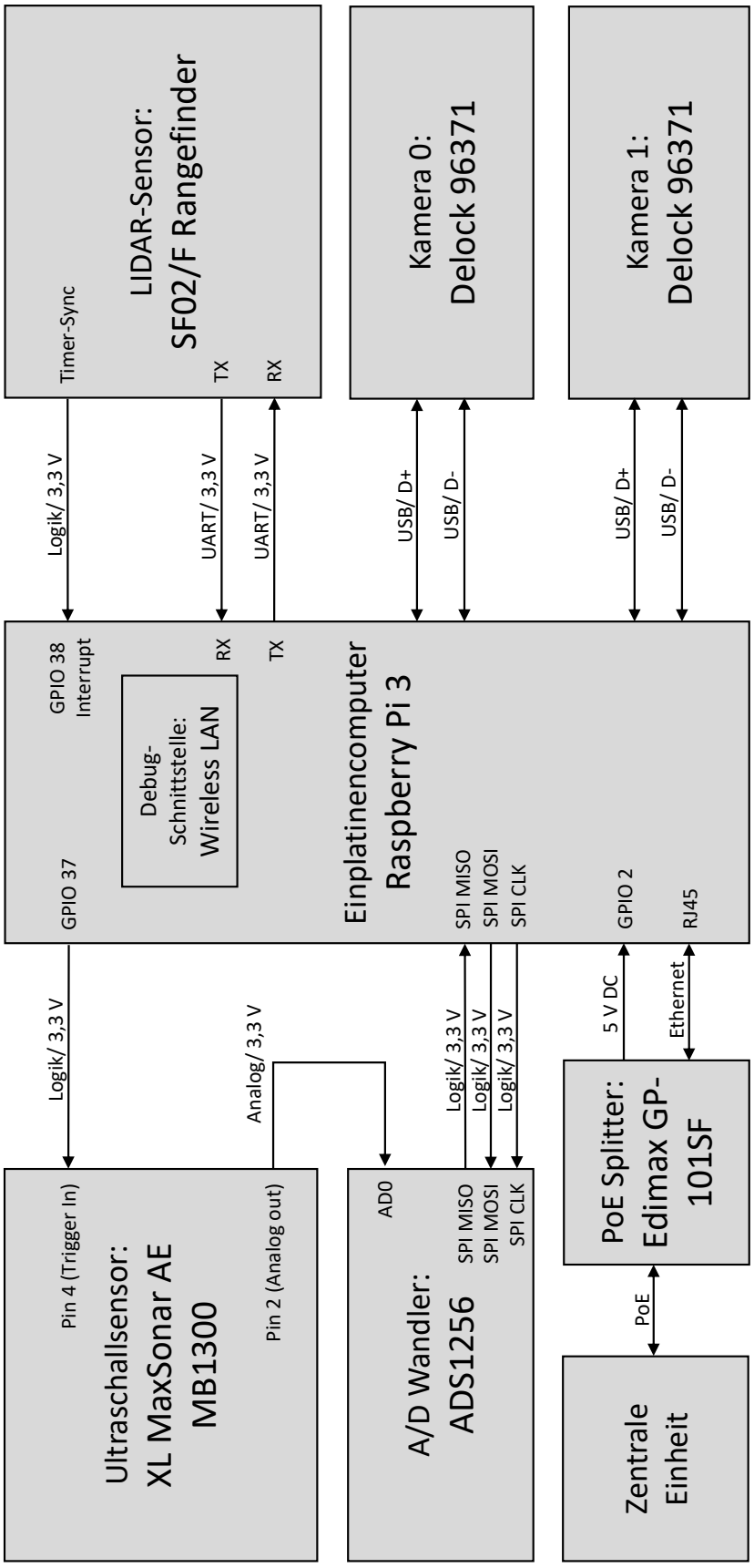

Abbildung 5.10.: Blockschaltbild der Sensoreinheit

Um eine Auswertung der Messdaten übersichtlicher zu gestalten, soll die Ansteuerung der Sensoren synchron erfolgen. Da der LIDAR-Sensor im Vergleich zu den anderen Sensoren nicht direkt ansteuerbar ist, wird die Messrate zur Synchronisierung aller Sensoren verwendet. Das vom Einplatinencomputer zu realisierende Zeitverhalten ist in Abbildung 5.11 dargestellt.

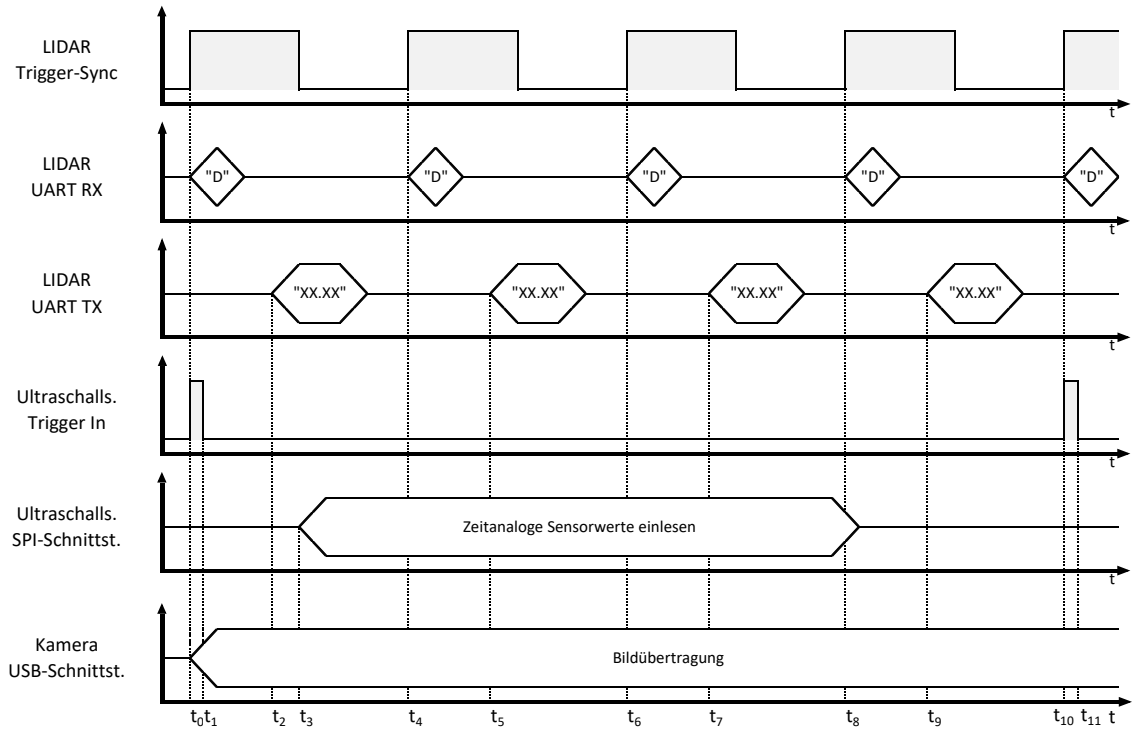

Abbildung 5.11.: Zeitlicher Ablauf Sensoransteuerung

Eine steigende Flanke am *Trigger-Sync* des LIDAR-Sensor gibt an, dass ein neuer Messwert zur Verfügung steht. Dies ist zum Zeitpunkt t_0 der Fall. Daraufhin wird über die *LIDAR-RX* Leitung das ASCII-Symbol 'D' gesendet, welches den aktuellen Messwert vom LIDAR-Sensor anfordert. Gleichzeitig wird der *Trigger-in* des Ultraschallsensors gesetzt, um eine Ultraschallmessung auszulösen. Außerdem werden zwei Bildaufnahmen über die USB-Schnittstellen gestartet. Zum Zeitpunkt t_1 wird der *Trigger-in* des Ultraschallsensors wieder zurückgesetzt, dies erfolgt nach der im Datenblatt [16] angegeben Zeit. Die Antwort des LIDAR-Sensors findet zum Zeitpunkt t_2 statt und wird über die *LIDAR-TX* als ASCII-Kette übertragen. Das Einlesen der zeitanalogen Sensorwerte beginnt bei t_3, da der Ultraschallsensor laut Datenblatt [16] 20,5 ms für die Kalibrierung und interne Berechnungen benötigt, bevor die Ultraschallwellen gesendet werden. Zum Zeitpunkt t_4 steht ein neuer Messwert

am LIDAR zur Verfügung, welcher durch das Senden des Zeichens 'D' angefragt und zum Zeitpunkt t_5 übertragen wird. Ein Setzen des Ultraschall *Trigger-in* findet nicht statt, da sich der Sensor noch in der Messaufnahme befindet. Gleiches gilt für die Zeitpunkte t_6, t_7, t_8 und t_9. Die Messung des Ultraschallsensors ist zum Zeitpunkt t_{10} abgeschlossen, so dass eine neue Messung durch setzen des *Trigger-in* Signals gestartet werden kann. Die Messwerte des LIDAR-Sensors werden zur gleichen Zeit angefordert.

Dieses Verfahren stellt sicher, dass die Messungen der verschiedenen Sensoren immer im gleichen Zeitbezug stehen und dem selben Zeitstempel zugeordnet werden können.

5.3.7. Software

Die Software der Sensoreinheit setzt sich aus zwei Komponenten zusammen, dem Programm zur Sensorauswertung und dem Programm zur Realisierung der Zeitsynchronität aller Systemkomponenten. Beide werden in den folgenden Abschnitten beschrieben.

Sensoransteuerung

Das Programm der Sensoreinheit hat die Aufgabe, das in Abbildung 5.11 dargestellte Zeitverhalten zu realisieren und die Messdaten an einen Server zu senden. Es ist in der Programmiersprache *C++* geschrieben und es besteht aus mehreren Klassen, deren grundsätzliche Aufgaben in Tabelle 5.15 kurz beschrieben sind. Eine ausführliche Beschreibung der wichtigsten Klassen erfolgt in den folgenden Abschnitten. Der Quellcode ist im Anhang B.2.2 enthalten.

Tabelle 5.15.: Übersicht der Klassen des Programms der Sensoreinheit

Klassenname	Beschreibung
SF02_F_Rangefinder	Klasse zur Ansteuerung SF02/F LIDAR. Die Kommunikation erfolgt über eine UART-Schnittstelle.
XL_MaxSonar	Klasse zur Ansteuerung XL MaxSonar Ultraschallsensors. Hierbei wird eine amplitudenanaloge Messung durchgeführt. Enthält ein Objekt der Klasse ADS1256_ISR zum Einlesen der Analogwerte.
ADS1256_ISR	Klasse zur Ansteuerung eines ADS1256 Analog/Digitalwandlers. Die Kommunikation erfolgt über eine SPI-Schnittstelle.
V4L_camera	Klasse zur Ansteuerung einer *Video4Linux*-Kamera [29].
Client	Klasse zum Aufbau einer TCP/IP-Verbindung zu einem Server. Die Kommunikation erfolgt bidirektional, es werden Objekte vom Typ *std::string* gesendet.
ConfigData	In dieser Klasse sind für das Programm wichtige Informationen und Statusflags enthalten. Bei Initialisierung wird eine Konfigurationsdatei eingelesen.
MeasuringPoint	Abstrakte *Interface*-Klasse, für Messpunkte. Definiert die Nutzung des Zeitstempels.
MeasurePoint_Laser	Diese Klasse erbt die Eigenschaften von *MeasuringPoint* und fügt einen LIDAR-Messwert hinzu.
MeasurePoint_Sonar	Diese Klasse erbt die Eigenschaften von *MeasuringPoint* und fügt eine Ultraschallmessung hinzu.
WiringPiInit	Klasse zur Initialisierung der verwendeten GPIO-Bibliotheken.

Klasse: SF02_F_Rangefinder Die Klasse *SF02_F_Rangefinder* enthält alle benötigten Variablen und Methoden, um eine Messung mit diesem LIDAR-Sensor durchzuführen. Das Klassendiagramm ist in Abbildung 5.12 dargestellt. Bei Aufruf des Konstruktors wird eine serielle Schnittstelle geöffnet. Danach stehen zwei Methoden zur Verfügung mit denen einen Messung durchgeführt werden kann. *takeMeasurement_string()* sendet das Trigger-Signal (ASCII-Zeichen 'D') über die serielle Schnittstelle. Danach wartet die Methode auf die Antwort des Sensors, liest diese Zeichenweise ein und fügt sie in einem *std::string* zusammen. Durch die Verwendung von synchronem I/O Multiplexen kann auf zyklisches Abfragen (polling) verzichtet werden. Außerdem erfolgt nach 30 ms ohne Antwort des Sensors ein Abbruch der Messung. In diesem Fall wird der *std::string* 'no value' zurückgegeben und als aktuelle Messung gespeichert. Da es sich hierbei um ein zeitunkritisches Abfragen eines Sensorwertes handelt, wird die Aussendung der Trigger-Nachricht um 1 ms verzögert, um Rechenressourcen direkt nach der Trigger-Auslösung für andere Programmteile bereitstellen zu können.

Die Methode *takeMeasurement_float()* ruft die oben beschriebene Funktion *takeMeasurement_string()* auf und wandelt den Empfangen *std::string* in den Datentyp *float* um. Sollte der empfangene *std::string*, zum Beispiel durch eine Zeitüberschreitung, nicht in eine Zahl umgewandelt werde können, wird der Wert *-1* zurückgegeben und gespeichert. Die Klasse besitzt außerdem den Indikator *ready*, der den Status des Objektes angibt und nur bei ent-

sprechendem Zustand eine Messung zulässt. So wird verhindert, dass bei Parallelisierung mehrere Prozesse oder Threads gleichzeitig eine Messung auslösen.

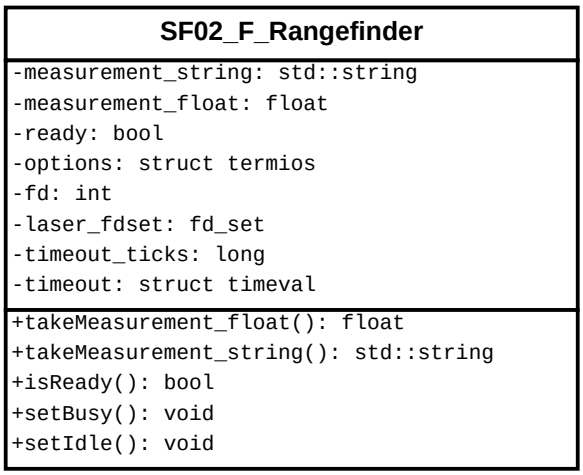

Abbildung 5.12.: Klassendiagramm für SF02_F_Rangefinder

Klasse: XL_MaxSonar und ADS1256_ISR Wie die Klasse *SF02_F_Ragefinder* enthält auch die Klasse *XL_MAXSonar* alle Variablen und Methoden, die für eine amplitudenanloge Messung benötigt werden. Das Klassendiagramm ist in Abbildung 5.13 dargestellt. Um die vom Sensor ausgehenden Analogspannungen einzulesen, besitzt die Klasse ein Objekt vom Typ *ADS1256_ISR*, welches das Einlesen der Analogwerte übernimmt. Bei Aufruf des Konstruktors wird der GPIO-Pin übergeben, der den Sendepuls des Ultraschallsensors auslöst. Außerdem wird ein Objekt vom Typ *ADC1256_ISR* mit einer Abtastrate von 7500 Hz erzeugt. Eine höhere Abtastrate ist auf Grund der verwendeten ISR nicht möglich. Die Methode *takeMeasurement()* für eine Ultraschallmessung aus, indem zunächst durch Setzen des *triggerOutputPins* eine Messung ausgelöst wird. Nach der im Datenblatt [16] gegebenen Wartezeit wird dann eine Messung des amplitudenanalogen Echosignal gestartet. Die Messung erzeugt einen *std::vector* vom Typ *int* und der Länge *SONAR_NUMBER_OF_MEASUREMENTS*, der die Messwerte enthält. Bei einer Abtastrate f_{ADC} von 7500 Hz und Messzeit t_{mess} von 60 ms ergibt sich für den Wert n_{mess} (*SONAR_NUMBER_OF_MEASUREMENTS*)

$$n_{mess} = f_{ADC} \cdot t_{Mess} = 7500Hz \cdot 60ms = 450. \tag{5.7}$$

Nach Aufnahme der Werte wird der *std:vector* gespeichert und zurückgegeben. Auch Klasse *XL_MaxSonar* besitzt einen Indikator *ready*, der verhindert, dass mehrere Prozesse oder Threads gleichzeitig eine Messung starten.

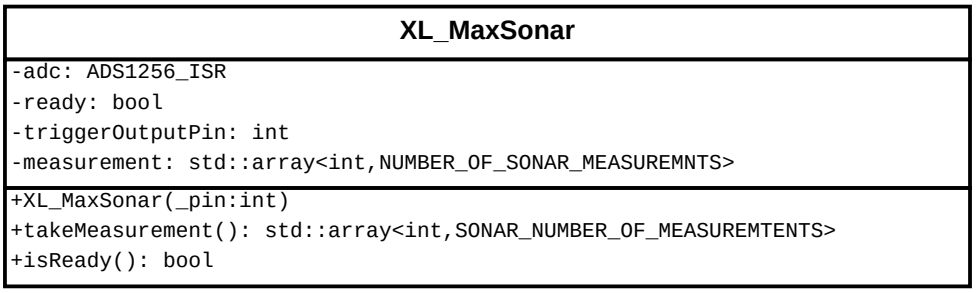

Abbildung 5.13.: Klassendiagramm für XL_MaxSonar

Die Klasse *ADS1256_ISR* enthält alle Variablen und Methoden zur Ansteuerung des *ADS1256*. Die Kommunikation erfolgt über die SPI-Schnittstelle. Hierfür wird die C-Bibliothek *BCM2835* [24] verwendet. Zur Erstellung der Klasse wird der in der Dokumentation des *Waveshare High-Precision AD_DA Board* bereitgestellte Beispielquelltext als Vorlage verwendet. Durch zusätzliche Verwendung der Bibliothek *WiringPi* [15] und die damit mögliche Nutzung von Interrupt-Routinen, ist eine Messung ohne aktives Warten (polling) möglich. Dies ist für die geforderte Parallelität unbedingt notwendig. Da die Reaktionszeit der Interrupt-Routine durchschnittlich 20 μs und bis zu 33 μs beträgt (siehe Anhang A.1) wird die Abtastrate auf 7500 Hz gesetzt. Das Klassendiagramm ist im Anhang B.1 dargestellt.

Klasse: V4L_camera Diese Klasse ermöglicht die Aufnahme eines Bildes mit einer *Video4Linux* kompatiblen Kamera. Die Klasse ist im Rahmen einer Studienarbeit entwickelt worden [29]. Nach der Aufnahme des Bildes, ausgelöst durch die Funktion *takePicture*, wird dieses lokal gespeichert.

Klasse: Client Die Objekte der Klasse *Client* dienen zur Kommunikation mit einem Server, der sich im Netzwerk befindet. Bei Aufruf des Konstruktors wird zunächst nur ein Objekt erzeugt und die Variablen auf ihre Default-Werte gesetzt. Mit der Methode *connectTo()* kann dann eine TCP-Verbindung zu einem Server aufgebaut werden. Hierfür wird die IP-Adresse und der Port des Servers übergeben. Bei erfolgreichem Verbindungsaufbau wird der Statusindikator *connected* gesetzt. Die Kommunikation erfolgt über Objekte vom Typ *std::string*, welche mit der Methode *sendData()* an den Server gesendet werden. Die Methode *receiveData()* dient zum Empfangen von Nachrichten, welche blockend auf Nachrichten vom Server wartet. Auch dies ist durch synchrones I/O Multiplexen realisiert, so dass kein aktives Warten benötigt wird.

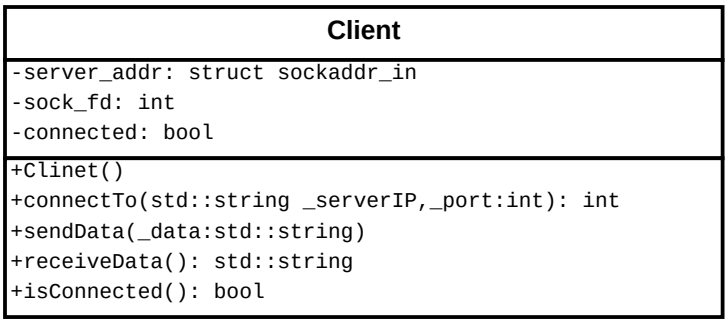

Abbildung 5.14.: Klassendiagramm für Client

Klasse: ConfigData Die Klasse *ConfigData* enthält alle Variablen, die für den Betrieb des Programms benötigt werden. Außerdem wird in dieser Klasse eine Konfigurationsdatei eingelesen, so dass bestimmt Änderungen ohne kompilieren durchgeführt werden können. Folgende Konfigurationen sind in der Datei definiert:

- Name der Sensoreinheit: *boxname*
- Internet Protocol (IP)-Adresse des Servers: *serverIP*
- Port-Nummer für LIDAR-Messungen: *laserPort*
- Port-Nummer für Ultraschallmessungen: *sonarPort*
- Port-Nummer für Steuer-Nachrichten: *cmdPort*
- Einstellung, ob eine LIDAR Messung durchgeführt werden soll *Sonar Measurements: (yes/no)*
- Einstellung, ob eine Ultraschallmessung durchgeführt werden soll *Sonar Measurements: (yes/no)*
- Einstellung, ob eine Stereo-Kamera-Messung durchgeführt werden soll *Sonar Measurements: (yes/no)*

Weiterhin sind in einem Objekt der Klasse Indikatoren enthalten, mit denen der Messvorgang gestartet und gestoppt werden kann. Der Zugriff erfolgt über getter-Methoden, das Klassendiagramm ist im Anhang B.1 dargestellt.

Klassen: MeasuringPoint, MeasurePoint_Laser und MeasurePoint_Sonar Die Klassen sind in Abschnitt 5.2.8 beschrieben.

Routine: *Main* Der Programmablaufplan der Hauptroutine ist in Abbildung 5.15 dargestellt. Sie startet zunächst mit der Initialisierung der Sensoren, hierbei wird für jeden Sensor ein

Objekt der entsprechenden Klasse angelegt. Für die Stereo-Kamera werden zwei Objekte der Klasse *V4L-camera* angelegt. Das darauf folgende Einlesen der Konfigurationsdatei setzt Name, IP-Adresse des Server und die benötigten Port-Nummern fest. Als nächstes wird die Interrupt Service Routine gestartet, diese prüft den *Timer-Sync*-Pin des *SF02 F Rangefinders* auf eine steigende Flanke und ruft in diesem Fall die Funktion *triggerISR* auf, welche in Abbildung 5.16 dargestellt und im folgenden Abschnitt beschrieben wird. Zur Kommunikation mit dem Server werden nun drei Objekte vom Typ *Client* erzeugt, die je eine TCP-Verbindung für LIDAR-Messwerte und Ultraschall-Messwerte aufbauen, sowie eine Verbindung zum Austausch von Befehlsnachrichten. Der Verbindungsaufbau und deren Überwachung wird im parallel laufenden Thread durchgeführt, welcher die Funktion *threadClient* durchläuft (Anhang B.3). Hierbei wird zyklisch geprüft, ob die Verbindung noch aktiv ist, um bei einem Verbindungsabbruch einen neuen Verbindungsversuch zu starten. Zuletzt wird in einer Schleife auf ankommende Befehle des Servers gewartet. Diese werden über die Befehlsverbindung als Objekt vom Typ *std::string* übertragen. Durch den Befehl *StopProgram* kann die Schleife verlassen und das Programm beendet werden. Die möglichen Befehle, welche an die Sensoreinheit gesendet werden können, sind in Tabelle 5.16 aufgelistet. Der Empfang von Nachrichten erfolgt ebenfalls über synchrones I/O Multiplexen, so dass kein aktives Warten stattfindet.

Tabelle 5.16.: Übersicht der Klassen des Programms der Sensoreinheit

Befehl	**Beschreibung**
StartMeasurement	Aufnahme und Senden von Messdaten wird gestartet.
StopMeasurement	Aufnahme und Senden von Messdaten wird gestoppt, das Programm läuft weiter.
Identify	Die Sensoreinheit sendet den eigenen Namen an den Befehlsserver zurück 'My name is *boxName*'.
StopProgram	Die Ausführung des Programms wird beendet.

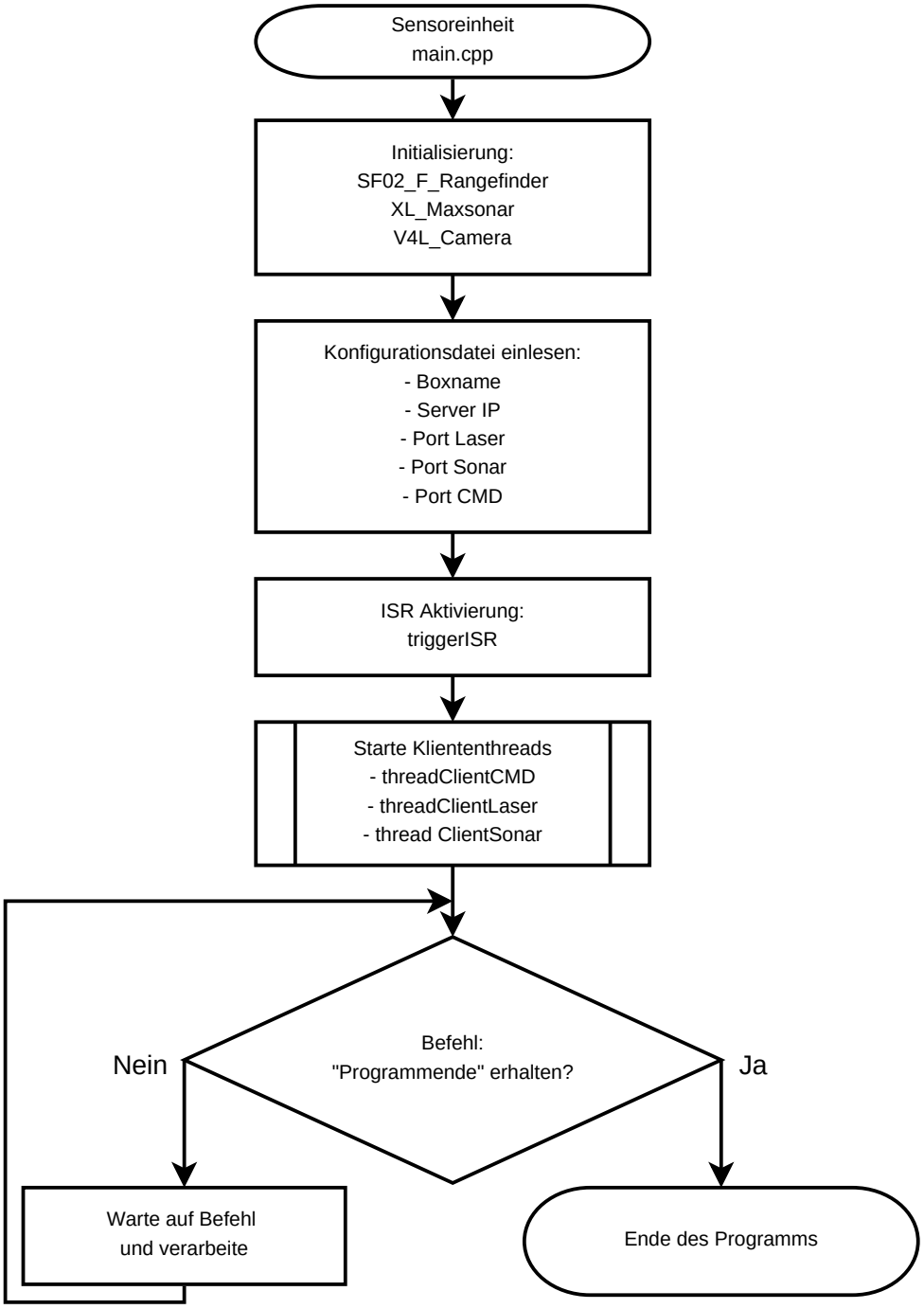

Abbildung 5.15.: Programmablauf der *main* Routine

Routine: *triggerISR* Das Zeitverhalten der Sensoreinheit wird durch die in Abbildung 5.16 dargestellte Interrupt Service Routine (ISR) realisiert. Da hier, wie in der Klasse *AD1256_ISR*, ebenfalls die ISR der Bibliothek *WiringPi* verwendet wird, muss die Reaktionszeit der Routine von ca. 20 μs berücksichtigt werden. Ausgelöst durch eine steigende Flanke am *TimerSync* des *SF02/F Rangefinders* wird die *triggerISR*-Funktion aufgerufen. Zunächst wird im Objekt der Klasse *ConfigData* geprüft, ob die Messung durch den Befehl *startMeasurement* gestartet wurde. Ist dies nicht der Fall, so wird die Routine wieder verlassen. Bei gestarteter Messung erfolgt die Erstellung eines Zeitstempels, der die aktuelle und absolute Systemzeit enthält. Dieser entspricht der *Unix-Zeit*, aufgelöst in Millisekunden. Es folgt die Prüfung, ob der LIDAR-Sensor bereit für eine Messung ist. Fällt die Prüfung positiv aus, so wird ein Thread der Funktion *threadLaser* gestartet. Nach dem Start des Threads wird die Kamera auf Bereitschaft geprüft. Diese wird zusätzlich mit dem Ultraschallsensor synchronisiert, da die Anforderung des Bildes die SPI-Schnittstelle beeinflussen kann. Beim gleichzeitigen Starten wird dies verhindert. Sind beide Bedingungen erfüllt, so wird auch hier der Thread zur Bildaufnahme gestartet. Zuletzt wird der Thread *threadSonar* zur Ultraschallmessung erzeugt, sofern der Ultraschallsensor bereit zur Messung ist. Alle Threads werden bei Erzeugung mit einer Referenz auf das Sensorobjekt und dem zu Beginn erstellten Zeitstempel aufgerufen. Die Programmablaufpläne der Funktionen *threadLaser* und *threadSonar* sind im Anhang B.3 dargestellt und haben grundsätzlich den gleichen Ablauf. Der Unterschied liegt darin, dass unterschiedliche Messpunkte erstellt werden (MeasurePoint_Laser und MeasurePoint_Sonar). Beide beginnen zunächst mit dem Sperren des Sensorobjektes, damit es nicht zu einer parallelen Messung kommen kann. Nun wird ein leerer Messpunkt mit dem zu Beginn erzeugten Zeitstempel angelegt, die Messung wird gestartet und dann dem Messpunkt hinzugefügt. Zur temporären Sicherung wird dieser lokal in einer Datei auf dem Einplatinencomputer gespeichert, so dass jeweils die Daten des letzten Messvorgangs ausgelesen können. Die dauerhafte Sicherung der Messdaten erfolgt in der zentralen Einheit an die der Messpunkt danach gesendet wird. Die Nachrichten sind für LIDAR und Ultraschallmessungen gleich formatiert und enthalten eine Einheitenkennung der Form *#boxNumber:*, gefolgt vom 13-stelligen Zeitstempel, welcher durch das Symbol *';'* von dem darauffolgenden Messwert getrennt wird. Ein LIDAR-Messpunkt von Sensoreinheit 1, aufgenommen um 123456789123 mit einer gemessenen Distanz ist demnach auf folgende Weise formatiert:

LIDAR-Nachricht: *#1:1234567890123;2.45*

Eine Ultraschallnachricht sendet anstatt eines Messwertes alle aufgenommenen Analogwerte getrennt durch das Symbol *';'*.

Ultraschall-Nachricht: *#1:1234567890123;1234567;2345678;3456789; ... ; 5678901*

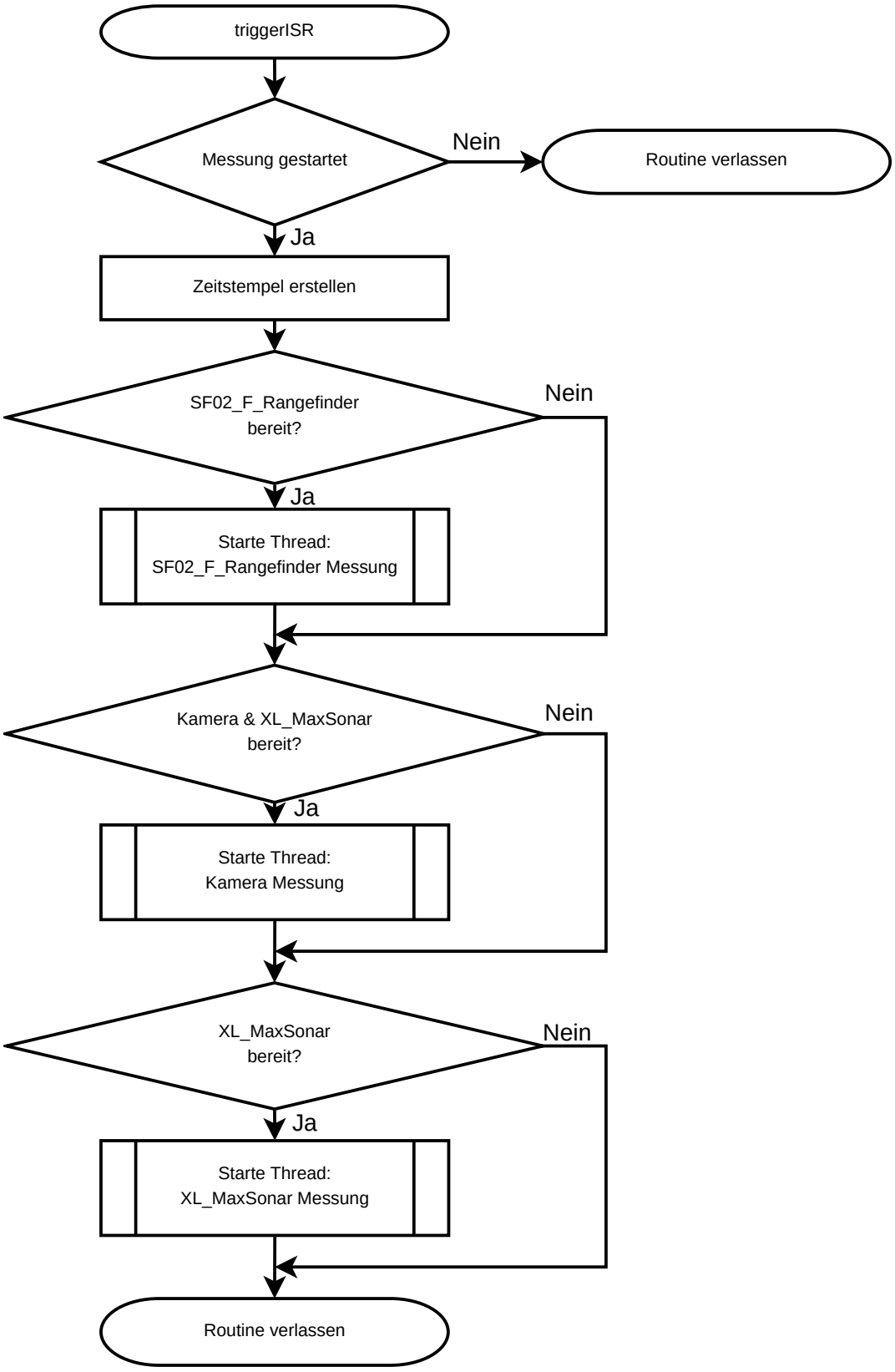

Abbildung 5.16.: Programmablauf der *triggerISR* Routine

Zeitsynchronisation

Auch hier wird für die Zeitsynchronisation das Precision Time Protocol (PTP), realisiert durch den *Linux Daemon PTPd*, verwendet. Dieser startet, wie bei der zentralen Einheit beim Bootvorgang des Einplatinencomputers. Die Sensoreinheiten treten in der Rolle des *Slaves* auf und übernehmen die Systemzeit der zentralen Einheit.

5.3.8. Datenaufkommen

Das Datenaufkommen, welches von einer Sensoreinheit erzeugt wird, ist in Tabelle 5.17 aufgeführt. Es setzt sich aus den Messungen der einzelnen Komponenten zusammen. Die Übertragung und Speicherung erfolgt in ASCII-Zeichen, daher müssen auch Trennzeichen berücksichtigt werden. Für den ersten Prototyp wird von einer Übertragung der Stereo-Kamera-Werte abgesehen.

Tabelle 5.17.: Datenaufkommen der Komponenten

Komponente	Speicher/ Messwert	Messperiode	Speicher/ Sekunde	Übertragung an die zentrale Einheit
LIDAR-Sensor	18 B	32 ms	562,5 B/s	ja
Ultraschallsensor	3613 B	128 ms	27,8 kB/s	ja
Stereo Kamera	200 kB	10 s	20 kB/	nein

Jede Sensoreinheit überträgt somit 28,4 kB/s Messdaten an die zentrale Einheit.

5.3.9. Gehäuse

Die Realisierung des Gehäuses der Sensoreinheit wird in einer Studienarbeit [29] durchgeführt. Diese sieht einen Aluminiumwinkel als Träger vor, der alle Komponenten trägt. LIDAR-Sensor und Ultraschall-Sensor sind hierbei zentral positioniert und von der Stereo-Kamera umgeben (Abbildung 5.17). Einplatinencomputer, LIDAR-Sensor und *Power over Ethernet*-Splitter sind an der Innenseite des Winkels angebracht (Abbildung 5.18).

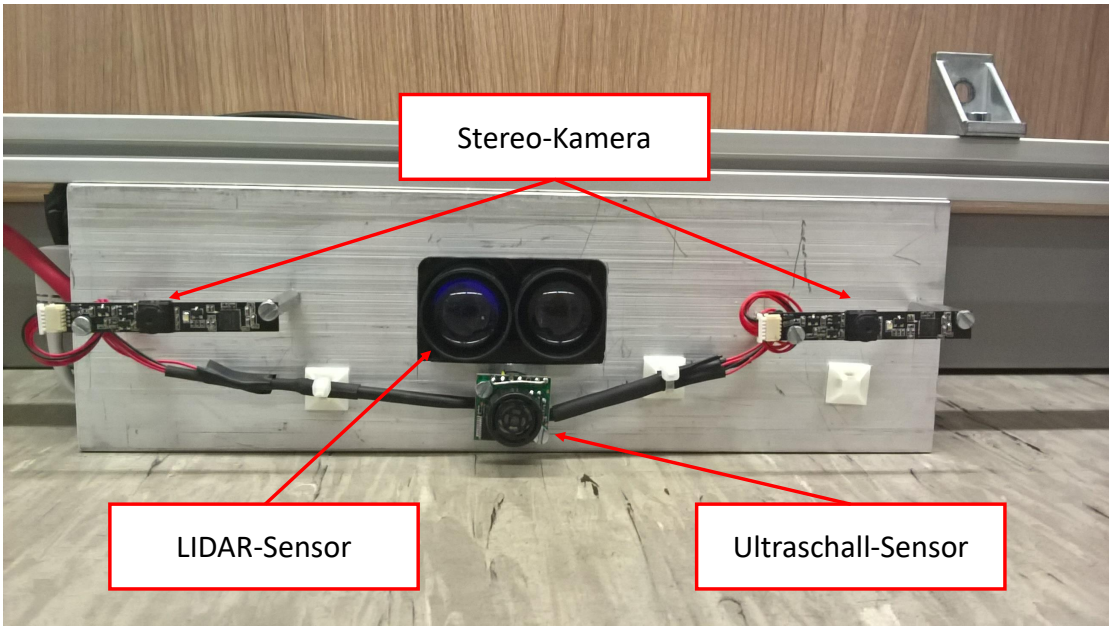

Abbildung 5.17.: Gehäuse der Sensoreinheit, Frontansicht

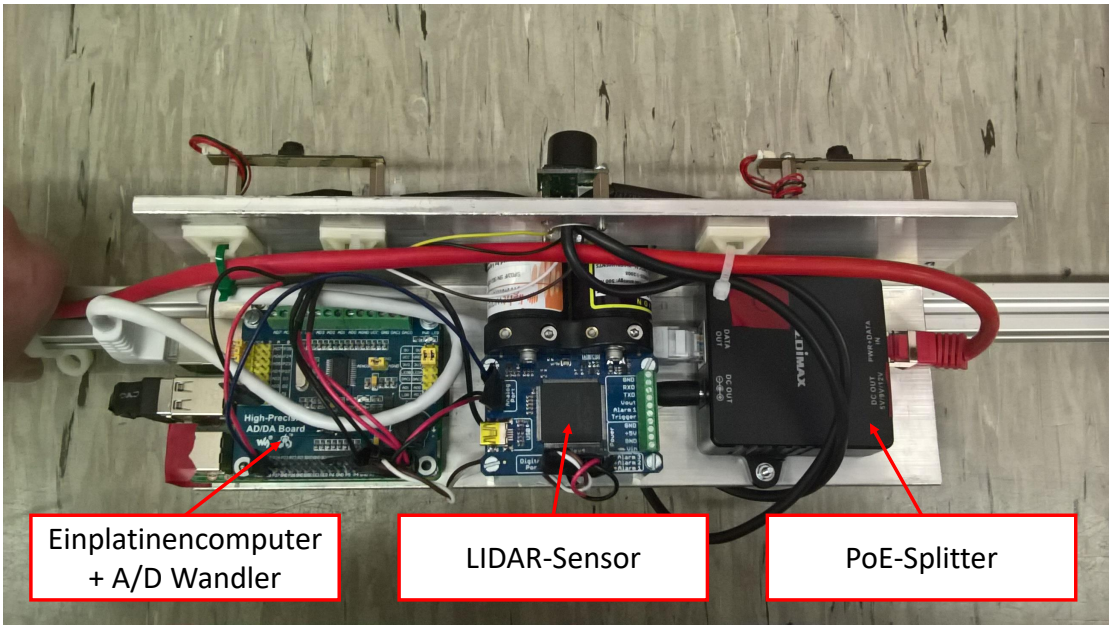

Abbildung 5.18.: Gehäuse der Sensoreinheit, Untersicht

5.4. Nachträgliche Datenauswertung

Die nachträgliche Datenauswertung zur Bestimmung der Position und Geschwindigkeit wird mit *MATLAB* durchgeführt. Als Basisdaten werden die in der zentralen Einheit gespeicherten Rohdaten verwendet. Hierfür wird zunächst eine gemeinsame Zeitbasis aller Sensoren erzeugt. Die LIDAR-Messpunkte werden in Zeitstempel und Messwert aufgeteilt und aus den GPS-Daten die Positionsdaten, Zeitstempel und Geschwindigkeit extrahiert. Aus den CAN-Daten wird nach Gleichung 5.2 die Geschwindigkeit des Fahrzeuges ermittelt.

Die LIDAR-Distanzmessungen werden auf Messwerte kleiner als 15 cm geprüft, welche dann gelöscht und linear interpoliert werden. Dies soll mögliche Fehlmessungen durch Interferenzen zwischen den parallel liegenden Sensoren herausfiltern (Abschnitt 6.1.4). Es folgt eine lineare Interpolation aller Rohdaten auf einen äquidistanten Zeitvektor t_{intp} mit einer Auflösung von einer Millisekunde.

5.4.1. Bestimmung der Position

Zur Bestimmung der Position wird vorausgesetzt, dass der zu bestimmende Positionsbereich mehrmals durchfahren wird. Da zu dieser Zeit noch keine präzise Karte zur Verfügung steht, soll diese, für die Realisierung des Prototyps, aus einer der Durchfahrten erzeugt werden. Zu diesem Zweck findet zunächst eine Extraktion der Durchfahrten aus den Rohdaten statt. Diese Daten werden dann durch Integration der CAN-Geschwindigkeit in eine Ortsfolge umgewandelt. Die Messwerte der ersten Durchfahrt werden als *Tarveling Edge* (Gleichung 4.2) definiert und dienen damit als globale Karte.

Die Positionsbestimmung wird dann mit der zweiten Durchfahrt durchgeführt, diese soll über einen einfachen Algorithmus erfolgen, welcher auf der quadratischen Abweichung (RMS) basiert. Die Messungen nach links und rechts werden getrennt behandelt. Somit werden zwei lokale Karten erstellt $m_{local,l}$ und $m_{local,r}$, die ortsabhängige LIDAR-Messwerte beinhalten. Es entstehen somit zwei eindimensionale Karten. Die Größe der Karten ist frei wählbar und abhängig von den zu erkennenden Strukturen in den Fassaden.

Für die Positionsbestimmung wird nun der Ort gesucht, an dem die lokale und globale Karte die geringste quadratische Abweichung aufweisen. Hierfür wird ein Rechteckfenster, so groß wie die lokale Karte schrittweise über die globale Karte geschoben. In jedem Schritt wird die quadratische Abweichung berechnet und gespeichert. Nach vollständigem Durchlauf entspricht der Punkt, an dem die niedrigste Abweichung aufgetreten ist, der geschätzten Position.

5.4.2. Bestimmung der Geschwindigkeit

Die zeitliche Differenz der parallel liegenden Sensoren, woraus sich die Geschwindigkeit ergibt, soll durch eine Kreuzkorrelation beider Sensorwerte ermittelt werden. Hierfür wird ein Algorithmus entwickelt, der aus zwei LIDAR-Messvektoren (Gleichung 4.5) den Zeitabstand bestimmt. Hierbei muss beachtet werden, dass durch die Interpolation die Größe des verwendeten Messvektor $\vec{z}$ nicht mehr von der Messrate der Sensoren, sondern vom Zeitvektor t_{intp} abhängig ist. Der dadurch entstehende Betrachtungszeitraum t_n ist somit durch

$$t_n = n \cdot 1\,ms \tag{5.8}$$

gegeben. Wobei n für die Länge des Vektors $\vec{z}$ steht.

Der Algorithmus beginnt mit dem Entfernen des Mittelwertes aus beiden Messvektoren, dies ist für eine Kreuzkorrelation notwendig. Die mittelwertfreien Folgen werden dann mit einer diskreten Kreuzkorrelation auf Ähnlichkeit geprüft. Das Maximum dieser Folge gibt dann den Punkt mit der höchsten Übereinstimmung an. Die Distanz vom Zentrum der Folge entspricht dann der Zeitverschiebung. Diese kann dann über den Abstand zwischen den parallel liegenden Sensoren r_A in eine Geschwindigkeit überführt werden. Die Messungen zu beiden Seiten werden hier getrennt behandelt, so dass zwei Geschwindigkeitsinformationen zur Verfügung stehen.

6. Test und Bewertung

6.1. Sensoreinheit

Zur Überprüfung der Sensoreinheit werden zunächst die Sensoren separat getestet und bewertet.

6.1.1. LIDAR-Sensor

Zunächst wird der Messvorgang des LIDAR-Sensors überprüft, indem die relevanten Spannungspegel mit einem *PicoScope 3406B* gemessen werden. Hierbei wird der in Abbildung 5.11 dargestellte Signalverlauf erwartet. Das Ergebnis der Messung ist in Abbildung 6.1 dargestellt, der Verlauf entspricht dem erwarteten Verhalten. Die Verzögerungszeit vom Auftreten der positiven Flanke am *Tigger-Sync* (blau) bis zum Aussenden der Sensordatenanforderung (gelb) beträgt 1,2 ms. Diese setzt sich aus einer Interrupt Auslösezeit von 20 μs, der im Programm definierten Wartezeit von 1 ms und der Zeit zur Ansteuerung der UART-Schnittstelle zusammen, welche nach dieser Messung auf ca. 180 μs gesetzt werden kann. Die Antwort des LIDAR-Sensors erfolgt 2,8 ms nach dem Aussenden der Datenanforderung. Somit beträgt die Gesamtdauer einer Sensorwertabfrage 3,5 ms.

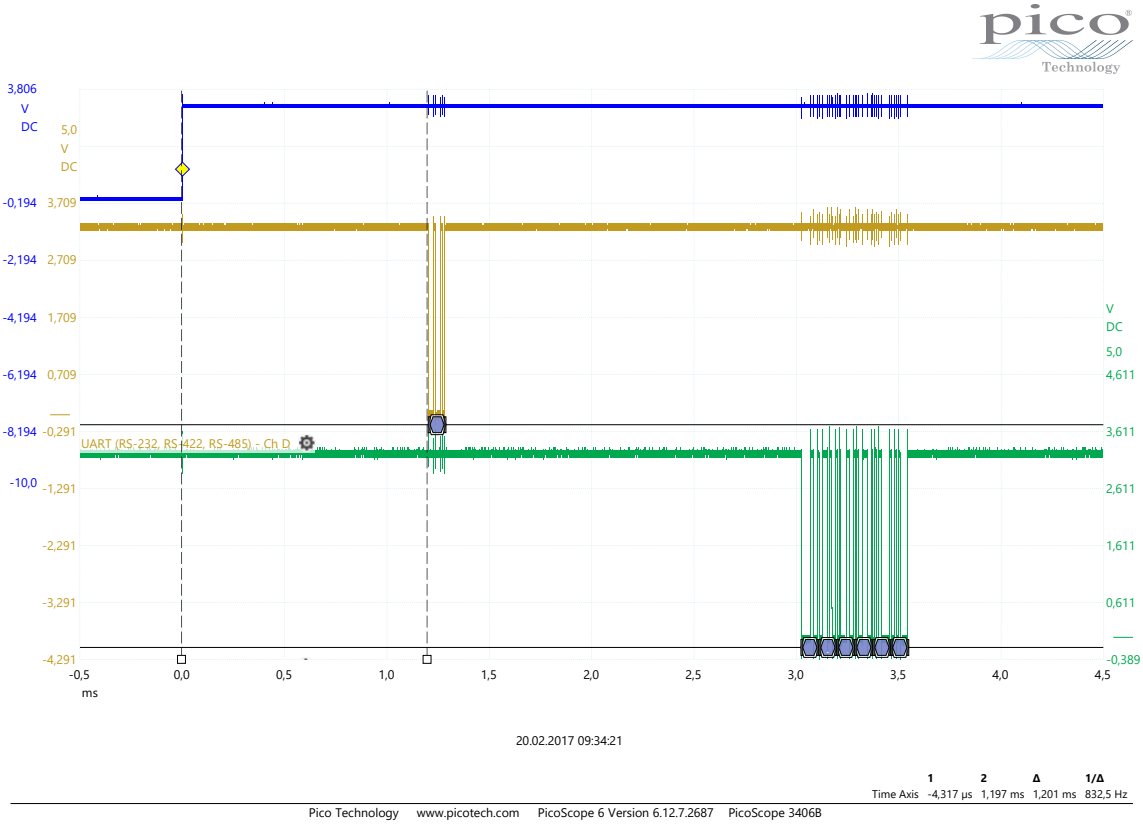

Abbildung 6.1.: Signalverlauf einer LIDAR Messung; Blau: Trigger-Sync; Gelb: UART(LIDAR) Rx; Grün: UART(LIDAR) Tx

Außerdem wird eine Überprüfung der Genauigkeit durchgeführt, die Ergebnisse sind in Tabelle 6.1 dargestellt. Die Messung ist in einem Innenraum durchgeführt worden. Die Ergebnisse entsprechen den im Datenblatt [22] angegebenen Genauigkeiten.

Tabelle 6.1.: SF02/F Rangefinder Genauigkeit

Referenzdistanz [m]	0,50	1	2	3	4	5	6	7
Referenzgenauigkeit (Güteklasse 2) [mm]	0,4	0,5	0,7	0,9	1,1	1,3	1,5	1,7
SF02/F Mittelwert μ_0 [m]	0,5005	0,9938	2,0147	3,0105	3,9698	4,9837	5,9446	6.9455
SF02/F Standardabweichung σ [m]	0,0022	0,0049	0,005	0,0033	0,0038	0,0049	0,0054	0,0059
Abweichung [mm]	0,5	6,2	-14,7	-10,5	30,2	16,3	55,4	54,5

Der LIDAR-Sensor zeigt somit das erwartete Verhalten. Über einen Zeitraum von 13 Minuten wird mit Hilfe der Zeitstempel geprüft, ob die Daten vollständig aufgenommen wurden. Hierfür

wird der zeitliche Abstand nebeneinanderliegender Zeitstempel auf Zeiten größer als die Messperiode von 32 ms geprüft. Aus dem Test ergibt sich eine Erfolgsquote von 99,93 %. Der Grund für den Datenverlust muss noch genauer untersucht werden. Es liegt nahe, dass dies auf eine nicht ausgelöste ISR zurückzuführen ist.

6.1.2. Ultraschallsensor

Zum Test des Ultraschallsensors wird zunächst der *AD1256* geprüft. Hierfür wird der Wandler im Spannungsbereich von 0 V bis 3,3 V vermessen, da das amplitudenanaloge Signale des *XL MaxSonar* in diesem Bereich liegen kann. Die Messwerte x_D können durch die Gleichung

$$U_A = \frac{2 \cdot 5V}{2^{24}} \cdot x_D \tag{6.1}$$

in eine Spannung U_A umgerechnet werden. Der Spannungsbereich wird in 0,1 V Schritten durchfahren, um Offset- und Verstärkungsfehler zu bestimmen.

$$U_{offset} = \quad 0,0096V \tag{6.2}$$

$$U_{gain} = \quad -0,0251V \tag{6.3}$$

Mit diesen Werten können die Messwerte korrigiert werden:

$$U_{A,korr} = U_A - U_{offset} - (U_{gain} - U_{offset} \cdot \frac{x_D}{n_{max}}) \tag{6.4}$$

Wobei U_A für die gemessene Spannung, x_D für den diskret gemessene Stufe und n_{max} für die maximal erwartete Stufe bei 3,3 V steht.

$$n_{max} = \frac{U_{A,max} \cdot 2^{24}}{2 \cdot 5V} = \frac{3,3V \cdot 2^{24}}{10V} \tag{6.5}$$

Der *ADS1256* ist somit in der Lage, die Signale des Ultraschallsensors mit ausreichender Genauigkeit aufzunehmen.

Zur Bewertung des Ultraschallsensors wird der Signalverlauf betrachtet, welcher in Abbildung 5.11 dargestellt ist. Der mit einem *PicoScope 3406B* gemessene Signalverlauf ist in Abbildung 6.2 dargestellt. Die notwendigen 20,5 ms Wartezeit, bevor das Aussenden des Ultraschallpulses beginnt, ist deutlich zu erkennen. Ab diesem Zeitpunkt startet die Datenübertragung über die SPI-Schnittstelle (grün). Das aufzunehmende amplitudenanaloge Signal (gelb) wird somit vollständig eingelesen. Die Daten werden wie in der Realisierung beschrieben gespeichert, die Funktionalität ist somit gegeben. Da der Ultraschallsensor nicht

für die entwickelte Positions- und Geschwindigkeitsbestimmung erforderlich ist, wird auf eine Spezifizierung des Sensors verzichtet.

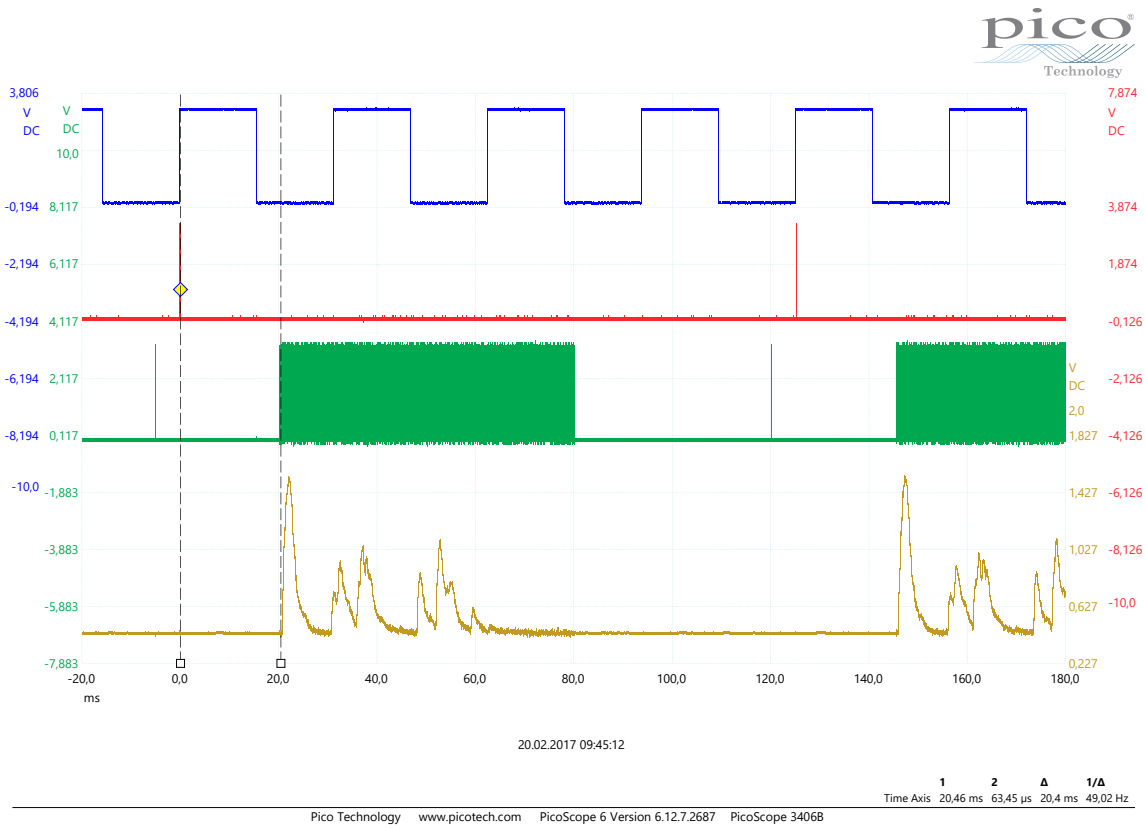

Abbildung 6.2.: Signalverlauf des Ultraschallsensors; Blau: Trigger-Sync; Rot: Trigger In; Grün: SPI-Schnittstelle (MISO) ; Gelb: Amplitudenanaloges Signal

6.1.3. Stereo-Kamera

Eine ausführliche Bewertung der Stereo-Kamera ist in der Studienarbeit, in der auch die Realisierung durchgeführt wurde [29], enthalten. Die Funktion der synchronen Bildaufnahme ist gegeben. Langzeittests zeigen aber, dass die Kameras unabhängig voneinander nach unregelmäßiger Zeit ihre Funktionsfähigkeit verlieren.

6.1.4. Gesamtsystem

Die grundsätzliche Funktionalität der Sensoreinheit ist gegeben, das parallele Starten der Messungen ist in Abbildung 6.3 dargestellt. Die Messung zeigt die gleichzeitige Ansteuerung

von LIDAR- und Ultraschallsensor, ausgelöst durch eine steigende Flanke am *Tigger In*. Der zeitliche Ablauf entspricht dem in der Realisierung vorgegebenen Ablauf (Abbildung 5.11). Sind alle Messungen aktiviert (LIDAR, Ultraschall und Stereo-Kamera), so treten Probleme mit den ISR auf. Trotz des synchronisierten Startens von Ultraschallsensor und Stereo-Kamera (siehe 5.3.7, Routine: triggerISR) werden nicht alle Flanken vom Einplatinencomputer erkannt, was zu Fehlern beim Auslesen der Analogwerte des A / D-Wandlers und zu nicht erkannten Flanken am *Trigger Sync* führen kann. Daher sollte auf eine gleichzeitige Messung von Stereo-Kamera und Ultraschallsensor verzichtet werden.

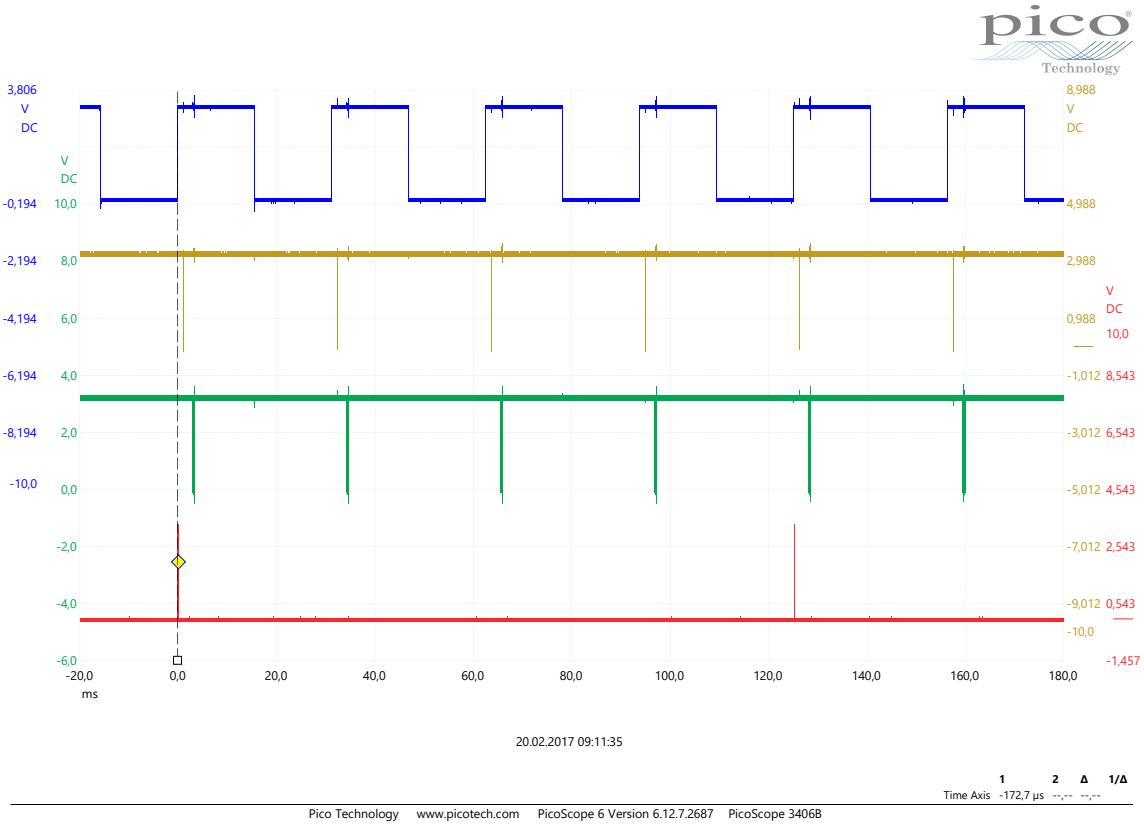

Abbildung 6.3.: Paralleles Auslesen von LIDAR- und Ultraschallsensor; Blau: Trigger-Sync; Gelb: UART(LIDAR) Rx; Grün: UART(LIDAR) Tx; Rot: Trigger In (Ultraschall)

Durch die teilweise verzögerten Messauslösungen entsprechen die Zeitstempel t_{mess} der Messpunkte nicht dem exakten Zeitpunkt der Messung t_{real}. Die wirklichen Zeitstempel können für die LIDAR- und Ultraschallsensoren mit den folgenden Gleichungen berechnet werden.

$$t_{real,LIDAR} = t_{mess} + t_{ISR} = t_{mess} + 20\mu s \tag{6.6}$$

$$t_{real,Ultras.} = t_{mess} + t_{ISR} + t_{cal} = t_{mess} + 20\mu s + 20,5ms \tag{6.7}$$

Wobei t_{ISR} für die Auslösezeit der ISR des Einplatinencomputers steht und t_{cal} für die vom Ultraschallsensor benötigte Kalibrierungszeit steht, die bis zum Aussenden des Ultraschallpulses vergehen. Da die Auslösezeit der ISR unterhalb der zeitlichen Auflösung liegt, kann diese vernachlässigt werden.

6.2. Sensorsystem

6.2.1. Zeitsynchronisation

Die durch das Precision Time Protocol (PTP) realisierte Zeitsynchronisation der Funktionskomponente wird ebenfalls geprüft. Zu diesem Zweck wird ein Testprogramm verwendet, das zu jeder vollen hundertstel Sekunde, also alle 10 Millisekunden, einen GPIO-Pin für eine Millisekunde setzt. Dieses Programm wird zeitgleich auf der zentralen Einheit und einer Sensoreinheit durchgeführt, so dass die zeitliche Verschiebung beider GPIO-Pins gemessen werden kann. Dieser lässt so auf die Synchronität der Systemuhren schließen. Die Schaltzeit der GPIO-Pins kann hierbei vernachlässigt werden.

Die Messung wird mit einem *PicoScope 3406B* durchgeführt, welches im persistenten Modus *Persistace Mode* betrieben wird. Abbildung 6.4 zeigt das Ergebnis einer 15-minütigen Messung an beiden GPIO-Pins. Der Modus der Zeitverzögerung beträgt 2,3 μs, die maximale Zeitdifferenz liegt bei 9,956 μs. Bei einer Zeitstempelauflösung von einer Millisekunde liegt der zu erwartende Synchronisationsfehler bei maximal 1 %.

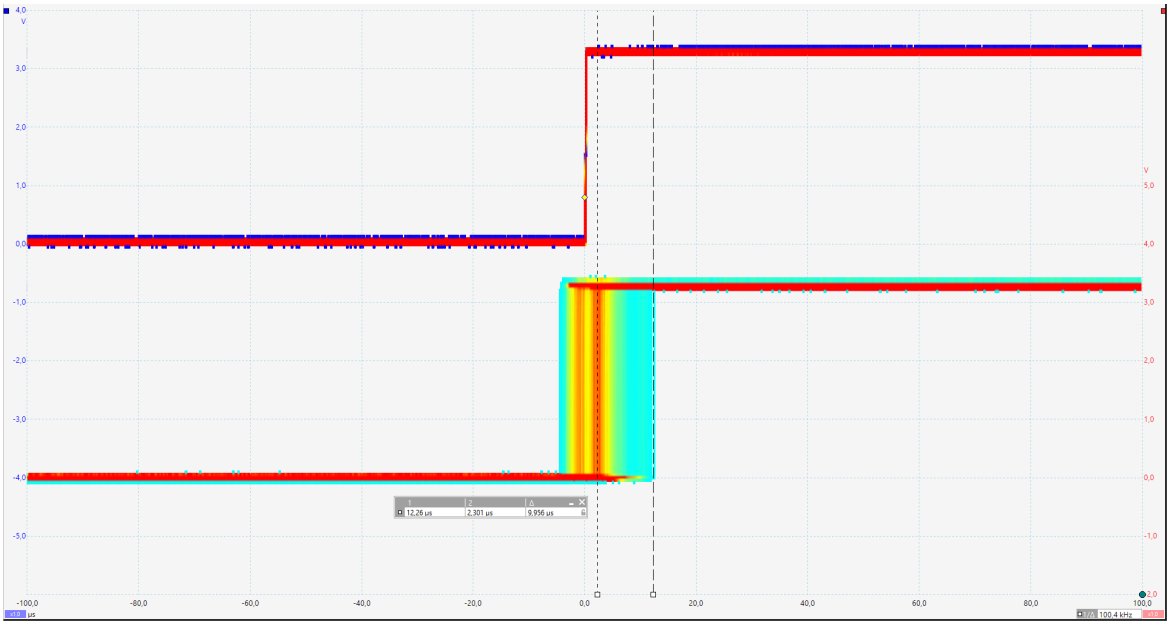

Abbildung 6.4.: Test der Zeitsynchronisation; Oben: GPIO-Pin der zentralen Einheit (Trigger); Unten: GPIO-Pin der Sensoreinheit

6.2.2. Ausgabeverzögerung

Zur Überprüfung der Ausgabeverzögerung wird die Laufzeit der Messpunkte betrachtet. Diese ist von der Erzeugung bis zum Zeitpunkt, an dem der Messpunkt in den Puffer der zentralen Einheit geschrieben wird, definiert. Es handelt sich also ausschließlich um die Übertragungsgeschwindigkeit, da im ersten Prototyp noch keine Auswertung erfolgt. Zur Bestimmung der Laufzeit wird der Zeitstempel des Messpunktes von der aktuellen Systemzeit der zentralen Einheit abgezogen, die Differenz steht dann für die Laufzeit des Messpunktes. Das Ergebnis einer Messung bestehend aus 480000 LIDAR-Messpunkten ist als Histogramm in Abbildung 6.5 dargestellt. Der Modus der Übertragungsdauer liegt bei 4 ms. Es treten aber auch Verzögerungen größer als 50 ms auf, dies kommt mit einer relativen Häufigkeit von $2 \cdot 10^{-5}$ zwar nur sehr selten vor. Die Verzögerungszeiten können dann aber bis zu 5 Sekunden betragen. Dies ist auf ein Problem des Einplatinencomputers beim Schreiben in Dateien zurückzuführen, welches mit einem einfachen Testprogramm rekonstruierbar ist. Hierbei wird in einer Endlosschleife in eine Datei geschrieben und die Dauer des Speichervorgangs gemessen. Zusammenfassend werden 99 % der Messpunkte innerhalb von 10 ms übertragen. Die vereinzelt auftretenden Verzögerungen führen aufgrund ausreichend großer Puffer zu keinem Datenverlust, für eine Echtzeit-Datenauswertung muss dies aber beachtet werden.

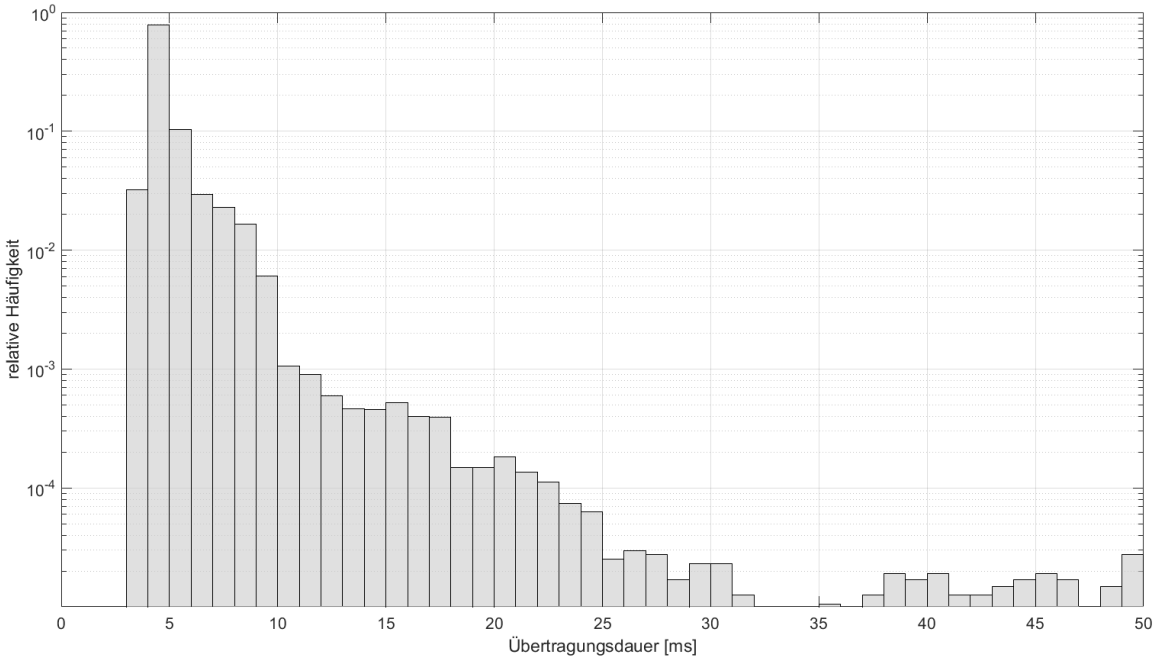

Abbildung 6.5.: Histogramm der Übertragungsdauer

6.2.3. Gesamtsystem

Die Funktionsweise des Sensorsystems ist somit gegeben, die Kommunikation zwischen der zentralen Einheit und den Sensoreinheiten wird erfolgreich ausgeführt. Bei gleichzeitiger Messung aller LIDAR-Sensoren treten, trotz gebündelter Lichtstrahlen, Interferenzen zwischen den parallel liegenden LIDAR-Sensoren auf. Dies führt dazu, dass vereinzelt die Distanz 0 gemessen wird oder Messfehler auftreten (Abbildung A.2). Die Interferenz kann allerdings leicht erkannt werden, da diese zeitgleich bei beiden Sensoren auftritt.

6.3. Nachträgliche Datenauswertung

Das Sensorsystem wird im Rahmen einer Testfahrt auf eine praxisnahe Funktionsweise getestet. Die während dieser Fahrt gesammelten Daten können dann zur nachträglichen Auswertung verwendet werden, um die entwickelten Algorithmen zu testen. Die Messung wird unter, für den GNSS-Sensor, guten Bedingungen durchgeführt, so sind durchschnittlich 9 Satelliten sichtbar. Da bei der Messung eine Zeitverschiebung zwischen der externen Erfassung der CAN-Nachrichten und dem Sensorsystem aufgetreten ist, müssen die Zeitstempel der Sensor-Messwerte um 25 s erhöht werden. Für die Testfahrt sind alle LIDAR-Sensoren

aktiv, Ultraschallmessungen werden nur von den Sensoreinheiten 1 und 2 durchgeführt. Die Einheiten 0 und 3 führen Stereo-Kamera-Messungen durch.

6.3.1. Bestimmung der Position

Zur Überprüfung des Positionsbestimmungsalgorithmus wird die gleiche Strecke dreifach durchfahren. Die gefahrene Route ist in Abbildung 6.6 darstellt, hierfür werden die aufgenommenen GPS-Daten verwendet. Bei der Route handelt es sich um eine Kreisfahrt, so ist das Ende einer Fahrt gleichzeitig der Beginn einer neuen Fahrt. Es zeigt sich, dass trotz guter Bedingungen für den GNSS-Sensor die Routen deutliche Abweichungen aufweisen und teilweise auch innerhalb von Gebäuden liegen. Die Länge einer Route beträgt ca. 720 m.

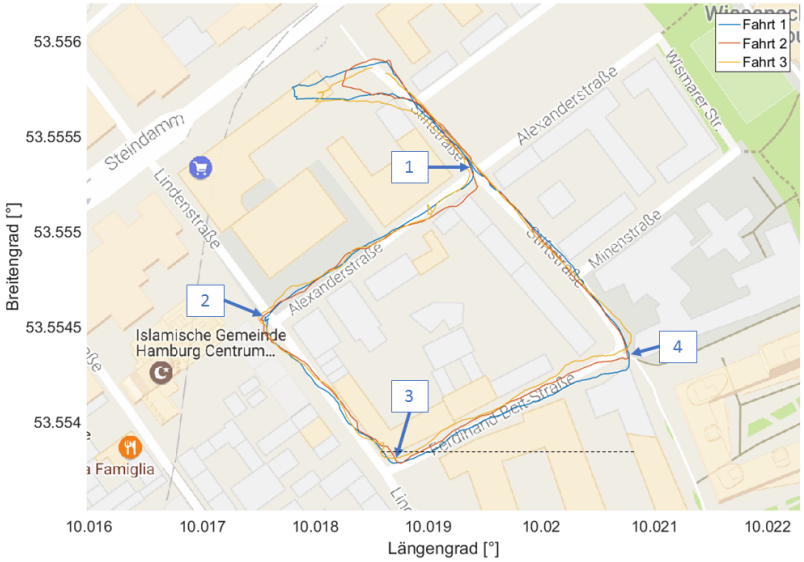

Abbildung 6.6.: Route der Testfahrt

Die LIDAR-Messwerte werden nun mit dem Geschwindigkeitssignal vom CAN-Bus des Fahrzeuges in eine Ortsfunktion umgewandelt, die örtliche Auflösung beträgt 1 cm. Die Karte, welche die 720 m lange Teststrecke repräsentiert, besitzt somit Vektoren der Länge 72000. Jedes Element der Vektoren kann so einer zurückgelegten Strecke dieser *Traveling Edge* zugeordnet werden. Abbildung 6.7 zeigt die aufgenommenen *Urban Canyons*, diese weisen an vielen Stellen eine deutliche Ähnlichkeit auf. Zur Orientierung werden die in Abbildung 6.6 markierten Punkte ebenfalls im *Urban Canyon* für die ersten Fahrt markiert. Die erste Fahrt wird als globale Karte verwendet, so dass Fahrt 2 und 3 für die Erzeugung der lokalen Karten verwendet werden und so für die Echtzeitmessungen des Fahrzeuges stehen. Da die

erfassten GPS-Positionen deutliche Ungenauigkeiten aufweisen wird für die Validierung des ersten Prototyps betrachtet, ob eine Positionsbestimmung grundsätzlich möglich ist. Hierfür wird geprüft, wie hoch die Rate des erfolgreichen Wiedererkennens von Strukturen ist. Außerdem soll experimentell ermittelt werden, welche lokale Kartengröße optimal ist.

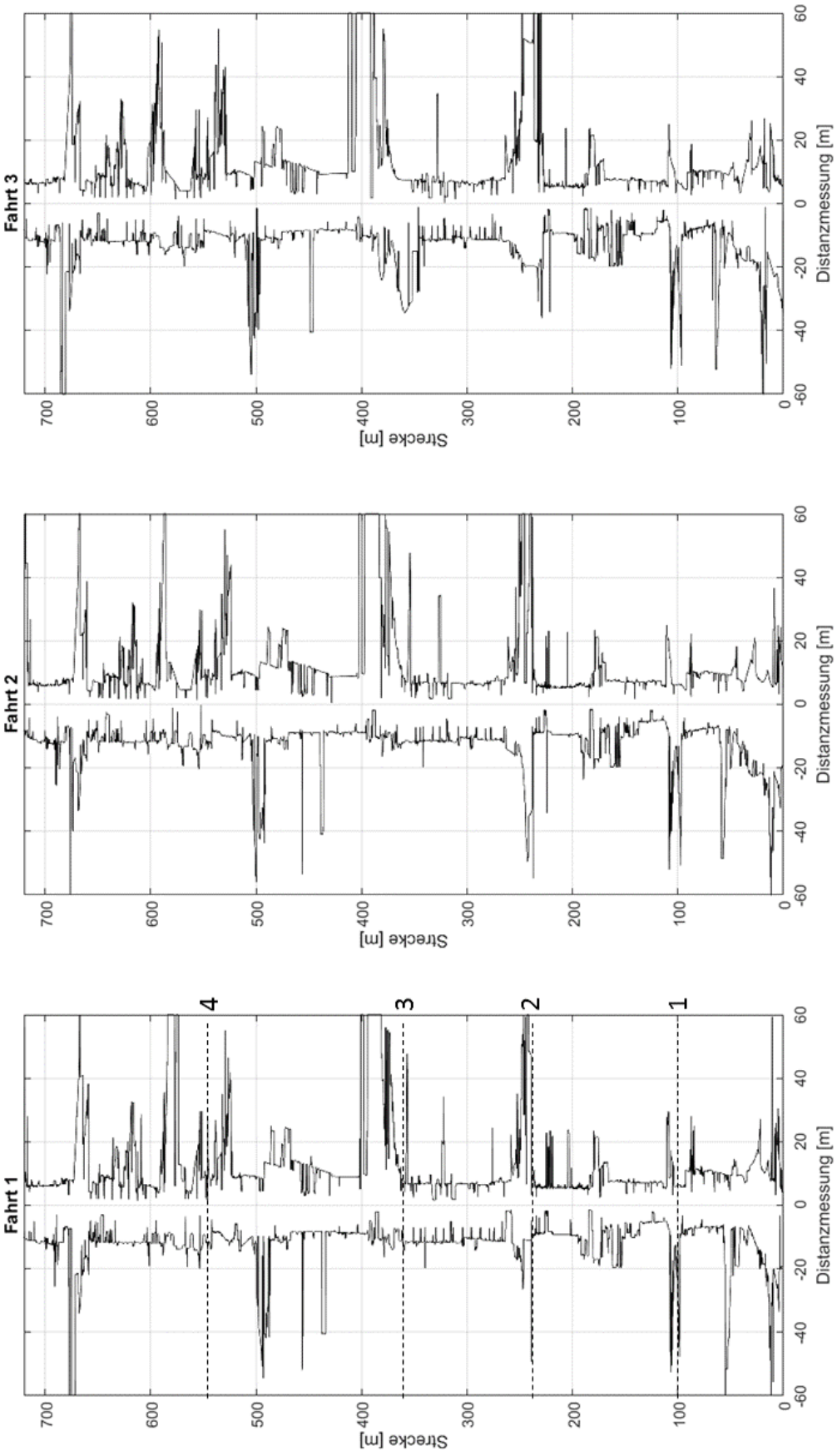

Abbildung 6.7.: *Urban Canyons* der Testroute

Zur Bestimmung der Erfolgsrate für die Positionsbestimmung werden die Fahrten 2 und 3 einzeln betrachtet. Die Rate soll durch die Positionsbestimmung an Referenzpunkten ermittelt werden. Für jeden Referenzpunkt werden zwei lokale Karten ($m_{local,r}$ $m_{local,l}$) erzeugt, deren Position dann durch den Positions-Algorithmus bestimmt wird. Die Referenzpunkte werden mit einem Abstand von 1 m über die gesamte Stecke verteilt. Im Idealfall kann so jede lokale Karte am richtigen Ort in der globalen Karte gefunden werden, so dass eine Gerade mit der Steigung 1 entsteht. Die Fahrten 1 bis 3 verfügen nicht über die gleiche Gesamtsteckenlänge da beispielsweise Kurven mit unterschiedlichen Radien durchfahren werden. Daher werden auch Positionen, die $\pm$ 5 % der Gesamtlänge von der erwarteten Position abweichen, als richtig geschätzt bewertet. Abbildung 6.8 zeigt ein Beispiel hierfür, die gestrichelten Linien symbolisieren das Toleranzband. Die Erfolgsrate berechnet sich dann aus dem Verhältnis der Punkte innerhalb des Toleranzbandes und der Gesamtanzahl.

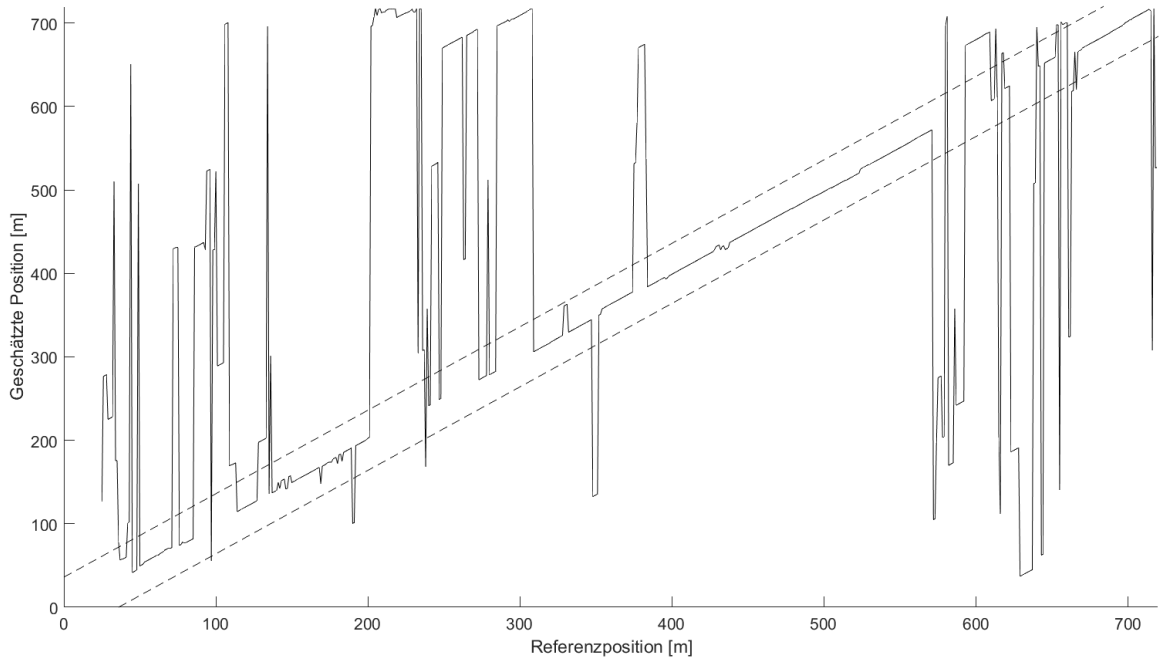

Abbildung 6.8.: Methode zur Erfolgsratenbestimmung bei einer Kartengröße $n_l = 25m$

Tabelle 6.2 zeigt das Ergebnis der Messung, hierbei werden die Erfolgsraten in Abhängigkeit der Größe der lokalen Karte n_l aufgezeigt. Die Erfolgsrate steigt proportional zur Größe der lokalen Karten, dies ist darauf zurückzuführen, dass in größeren Karten mehr markante Bereiche enthalten sind. Es ist aber zu beachten, dass diese Bereiche auch weiter zurückliegen können. Die Positionsbestimmung wird dadurch auch von der Genauigkeit des Geschwindigkeitssignals abhängig. Daher kann die Karte nicht beliebig groß gewählt werden. Die Ergebnisse zeigen, dass ab einer Kartengröße von 5 m eine Erfolgsrate von über 50 %

erreicht werden kann. Dies zeigt, dass eine Positionsbestimmung anhand von *Urban Canyons* möglich ist. Die verhältnismäßig niedrigen Erfolgsraten sind darauf zurückzuführen, dass die *Urban Canyons* zwar markante Verläufe zeigen, diese sich aber mit dem verwendeten Algorithmus nicht unterscheidbar sind. Außerdem spielen dynamische Objekte in der Umgebung eine große Rolle.

Außerdem zeigt sich, dass die Positionsbestimmung zur rechten Seite fast immer erfolgreicher ist, als die Messung zur linken Seite. Grund hierfür ist, dass Messungen zur linken Seite durch Gegenverkehr gestört werden können.

Tabelle 6.2.: Erfolgsraten der Positionsbestimmung bei verschiedenen Kartengrößen n_l

Messung	$n_l = 0,5m$	$n_l = 1m$	$n_l = 5m$	$n_l = 10m$	$n_l = 25m$	$n_l = 50m$	$n_l = 100m$
Fahrt 1; $m_{local,l}$	0,3167	0,3853	0,5510	0,5845	0,6230	0,6478	0,7242
Fahrt 1; $m_{local,r}$	0,3333	0,3950	0,5720	0,6310	0,6518	0,7104	0,7952
Fahrt 2; $m_{local,l}$	0,3194	0,3574	0,5077	0,5592	0,5281	0,5478	0,5129
Fahrt 2; $m_{local,r}$	0,3236	0,3435	0,5343	0,5408	0,5683	0,6985	0.7758

6.3.2. Bestimmung der Geschwindigkeit

Die Geschwindigkeitsbestimmung wird im gesamten Zeitbereich der Testfahrt durchgeführt. Als Referenzsignal wird das Geschwindigkeitssignal aus den CAN-Daten verwendet. Hierfür wird der Algorithmus an äquidistant verteilten Zeitpunkten angewendet. Der Abstand dieser Punkte beträgt 100 ms. Zu jedem dieser Zeitpunkte werden die relevanten Zeitfolgen der beiden parallel liegenden LIDAR-Sensoren extrahiert und die Geschwindigkeit für diesen Zeitpunkt durch den Algorithmus bestimmt.

Für die Bewertung der Geschwindigkeitsbestimmung wird Rate der erfolgreich bestimmten Geschwindigkeiten ermittelt. Werte, die innerhalb der in Tabelle 3.1 geforderten Genauigkeit liegen, werden als erfolgreich gewertet. Die zu erreichende Genauigkeit beträgt 1 km/h, also 0,277 m/s. Dies wird bei verschiedenen Betrachtungsdauern t_n durchgeführt. Abbildung 6.9 zeigt exemplarisch das Ergebnis der Geschwindigkeitsbestimmung. Hierbei sind große Abweichungen von der Referenzgeschwindigkeit zu erkennen. Es wird aber auch deutlich, dass die Geschwindigkeitsbestimmung an bestimmten Stellen erfolgreich ist. Die Fehler überwiegen allerdings deutlich.

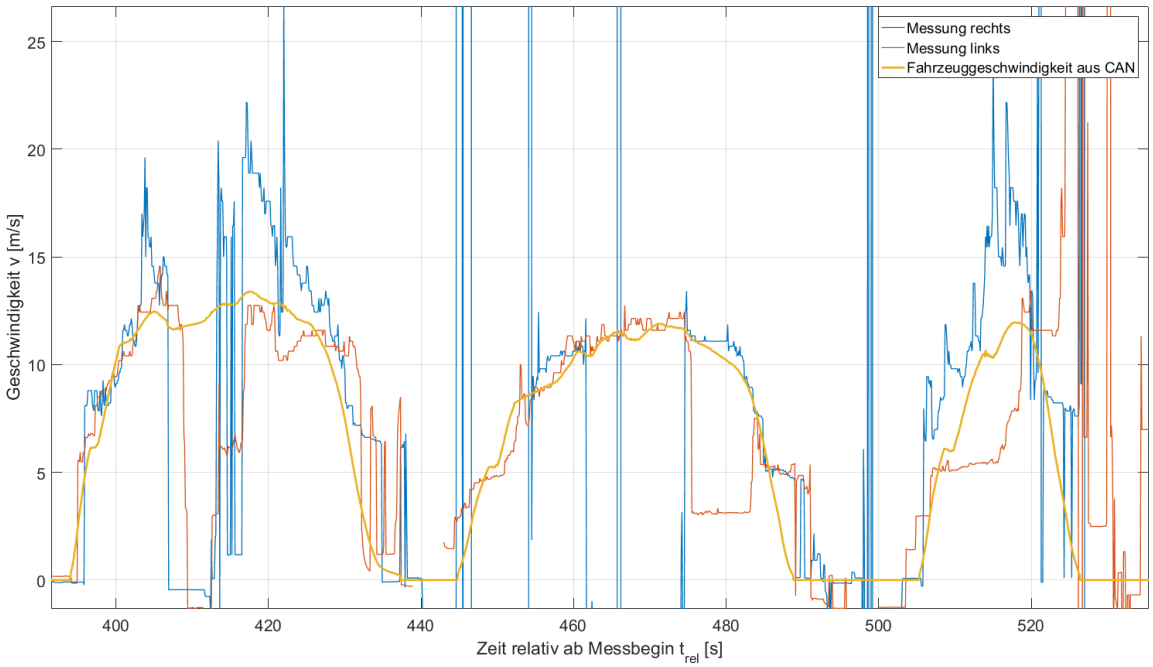

Abbildung 6.9.: Geschwindigkeitsbestimmung bei $t_n = 8$

Die ist auch bei verschiedenen Betrachtungsdauern sichtbar, die Erfolgsrate der Bestimmung liegt bei nur 10 % (Tabelle 6.3). Die grundsätzliche Funktion ist somit zwar gegeben, der verwendete Algorithmus führt aber nicht zu einem akzeptablen Ergebnis.

Tabelle 6.3.: Erfolgsraten der Geschwindigkeitsbestimmung bei verschiedenen Betrachtungsdauern n_l

Messung	$t_n = 0.5s$	$t_n = 1s$	$t_n = 2s$	$t_n = 5s$	$t_n = 8s$	$t_n = 10s$	$t_n = 20s$
Messung links	0,0355	0,0548	0,0655	0,1168	0,1427	0,1588	0,1424
Messung rechts	0,0289	0,0375	0,0580	0,0999	0,1232	0,1143	0,0832

6.4. Zusammenfassung

Zum Abschluss wird dargestellt, inwieweit die in Kapitel 3 beschriebenen Anforderungen erfüllt werden. Die in Tabelle 3.1 enthaltenen Anforderungen an die Messdaten können nicht erfüllt werden, da im Rahmen des ersten Prototyps zunächst eine Aussage über die Machbarkeit getroffen wird. Hierbei zeigen die Ergebnisse der Positionserkennung, dass dies anhand von *Urban Canyons* mit dem entwickelten Algorithmus möglich ist. Für die Geschwindigkeitsmessung ist dies nicht der Fall, mit dem entwickelten Algorithmus konnte keine ausreichende Genauigkeit erreicht werden (Tabelle 6.4).

Tabelle 6.4.: Bewertung der Messdatenanforderungen

Messdaten	**Bewertung**	**erfüllt?**
Position	Erkennungsrate über 50 %, Aussage zur Genauigkeit kann nicht getroffen werden	-
Geschwindigkeit	Erkennungsrate 10 %, keine Zuverlässige Geschwindigkeitserkennung möglich	- -

(++ = voll erfüllt; + = größtenteils erfüllt; - = teilweise nicht erfüllt; - - = nicht erfüllt)

Die technischen Anforderungen aus Tabelle 3.2 sind in Tabelle 6.5 bewertet. Diese können nahezu vollständig erfüllt werden. Eine Nutzer-Schnittstelle ist über ein WLAN-Netztwerk realisiert und ermöglicht so eine vollen Zugriff auf alle Funktionseinheiten. Die verwendete Speicherkarte ermöglicht eine Speicherung der Rohdaten von weit mehr als einer Stunde. Die Möglichkeit für weitere Auswertungen ist durch Ultraschallsensor und Stereo-Kamera gegeben. Da beide Sensoren nicht gleichzeitig an einer Sensoreinheit aktiviert sein können, ist dies mit kleinen Einschränkungen erfüllt. Auch die geforderte Ausgabeverzögerung kann auf Grund vereinzelnd auftretender Verzögerungen von über 50 ms nur bedingt erfüllt werden. Auch handelt es sich hierbei um die Übertragungsgeschwindigkeit, so dass die Zeit zur Auswertung noch zuzufügen ist, sobald einen Echtzeit-Datenverarbeitung implementiert wird. Da die Testfahrt bei Geschwindigkeiten über der Vorgegebenen durchgeführt wurde, ist eine Funktionsfähigkeit auch bei höheren Geschwindigkeiten gegeben. Die Abtastgeschwindigkeit ist durch die Abtastrate der LIDAR-Sensoren gegeben, damit kann die geforderte Abtastrate erfüllt werden. Durch die Verwendung von GPS-Empfänger und Aufzeichnung des Fahrzeug-CAN-Busses können die Anforderungen nach Referenzsystemen voll erfüllt werden.

Tabelle 6.5.: Bewertung der Technische Anforderungen

Funktion	**Anforderung**	**erfüllt?**
Nutzer-Schnittstelle	Drahtlos	++
Datenaufzeichnung	Rohdatensicherung für 1 Stunde	++
Erweiterbarkeit	Möglichkeit zur Funktionserweiterung und weiteren Auswertung	+
Ausgabeverzögerung	Maximal 50 ms	+
Fahrzeuggeschwindigkeit	0,1 m/s - 1 m/s, oder höher	++
Ausgaberate	Mindestens 10 Hz	++
Referenzsystem Position	GNSS	++
Referenzsystem Geschwindigkeit	Odometrie	++

(++ = voll erfüllt; + = größtenteils erfüllt; - = teilweise nicht erfüllt; - - = nicht erfüllt)

Zuletzt werden die Umweltanforderungen aus Tabelle 3.3 betrachtet. Hier können alle Anforderungen erfüllt werden, durch die Befestigung am Dachgepäckträger des Fahrzeuges ist eine rückstandsfreie Demontage möglich. Eine Montage ist in unter 30 Minuten möglich. Die Fahrzeugzulassung bleibt durch Verwendung eines CAN-Kopplers mit Straßenzulassung erhalten, so dass ein Einsatz im öffentlichen Straßenverkehr möglich ist. Die Energieversorgung erfolgt aus dem Fahrzeugnetz, so kann ein beliebig langer Einsatzzeitraum erreicht werden. Auch der Temperaturbereich wird von jeder verwendeten Komponente erfüllt.

Tabelle 6.6.: Bewertung der Umweltanforderungen

Funktion	**Anforderung**	**erfüllt?**
Applikationsfahrzeug	Mazda 5 (2006 USA)	n.n.
Einbauart	Rückstandsfreie Demontage	++
Fahrzeugzulassung	Darf nicht erlöschen	++
Energieversorgung	Energieversorgung aus Fahrzeug oder Akkubetrieb	++
Temperaturbereich	0°C - 40°C	++

(++ = voll erfüllt; + = größtenteils erfüllt; - = teilweise nicht erfüllt; - - = nicht erfüllt; n.n. = nicht notwendig)

7. Abschluss der Arbeit

7.1. Fazit

Im Rahmen der Arbeit wurde ein Sensorsystem zur Positionsbestimmung in urbanen Gebieten konzeptioniert und ein erster Prototyp realisiert. Dieser wurde dann in einer Testfahrt einem Praxistest unterzogen. Das System nutzt die Strukturen, die an Häuserfassaden auftreten, um daran eine Positionsbestimmung durchzuführen. Außerdem wird die Geschwindigkeit des Fahrzeuges bestimmt.

Das System besteht aus vier unabhängigen Sensoreinheiten, die unabhängig voneinander Distanzmessungen zu beiden Seiten orthogonal zur Fahrtrichtung eines Fahrzeuges durchführen. Jede Einheit besitzt einen LIDAR- Sensor, einen Ultraschallsensor und eine Stereo-Kamera. Die aufgenommenen Messdaten werden dann an eine zentrale Einheit gesendet. Diese führt im ersten Prototyp die Sicherung der Daten und eine Sychronisationsprüfung aus. Die Auswertung der Daten und damit auch die Bestimmung von Position und Geschwindigkeit findet beim ersten Prototyp in einer nachträglichen Auswertung statt.

Die Funktion des ersten Prototypen ist gegeben, so findet eine zuverlässige Aufnahme der Messwerte statt. Auch die Übertragung der Messdaten erfolgt ohne Messdatenverlust. Es treten aber betriebssystembedingte Verzögerungen in der Datenübertragung auf, dies führt aber nicht zu einem Datenverlust. Für eine Auswertung in Echtzeit muss dies aber berücksichtigt werden.

Für eine nachträgliche Auswertung wurden zwei Algorithmen entwickelt, die die Positions- und Geschwindigkeitsbestimmung realisieren. Es zeigt sich, dass der Geschwindigkeits-Algorithmus zwar grundsätzlich korrekte Geschwindigkeiten bestimmt, die Zuverlässigkeit aber nicht ausreicht. Der Positionsbestimmungs-Algorithmus zeigt, dass eine Positionsbestimmung anhand von Gebäudefassaden möglich ist. Die vermessenen Strukturen der Teststrecke können an vielen Streckenteilen eindeutig wiedererkannt werden. Es treten aber Orte auf, an denen eine Positionsbestimmung nicht möglich ist. Das Ergebnis der Testfahrt zeigt aber, dass eine Weiterentwicklung des Systems lohnenswert ist.

7.2. Ausblick

Die Entwicklung des Sensorsystems ist nach Erstellung eines ersten Prototypen noch nicht abgeschlossen. Die Weiterarbeit teilt sich in zwei Teile auf: die Verbesserung vom Sensorsystem und die Weiterentwicklung der Algorithmen.

Zur Weiterentwicklung des Sensorsystems sollte zunächst das verwendete Betriebssystem der Funktionseinheiten stärker an die Anforderungen angepasst werden. Dies kann durch die Verwendung von Echtzeit-Distributionen, wie zum Beispiel *RTLinux* realisiert werden. So sollten die teilweise auftretenden Verzögerungen dann nicht mehr vorkommen. Auch sollte es dann möglich sein, Sonar- und Stereo-Kamera gleichzeitig in einer Sensoreinheit aktivieren zu können. Um das System für den Echtzeit-Einsatz vorzubereiten muss die Auswertung des CAN-Signals mit dem gleichzeitigem Herausfiltern der Fahrzeuggeschwindigkeit auf dem Einplatinencomputer realisiert werden. So kann dann auch die Datenauswertung in Echtzeit auf dem Einplatinencomputer stattfinden.

Da der entwickelte Algorithmus zunächst nur auf Verfügbarkeit getestet wurde, soll auch dieser weiterentwickelt und auf Genauigkeit geprüft werden. So sollte zunächst eine Fusion von den Messungen nach links und rechts durchführt werden. Dies könnte beispielsweise durch Bestimmung der *Canyon*-Breite erfolgen. Hierbei muss aber beachtet werden, dass die Messungen zur rechten Seite zu zuverlässigeren Ergebnissen führen. Auch können die zusätzlich aufgenommenen Messwerte der Ultraschallsensoren mit einbezogen werden. Zur Bewertung der Genauigkeit kann aus den Stadt-Höhenprofil-Daten, welche über das *Transparenzportal Hamburg* zur Verfügung steht, eine Karte erstellt werden, die dann als Referenz dient.

Alle Funktionen müssen dann in weiteren Testfahrten validiert werden. Diese sollten bei verschiedenen Geschwindigkeit stattfinden, um die Grenzen des Systems zu bestimmen.

A. Anhang

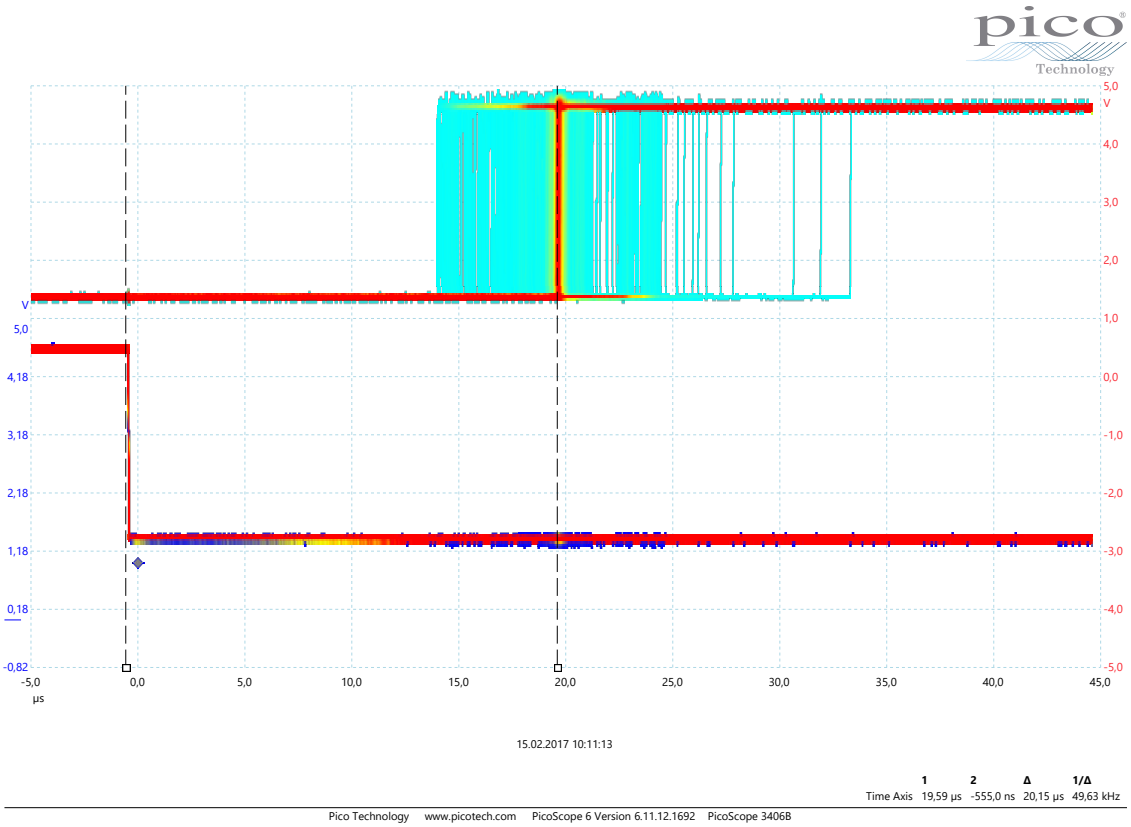

Abbildung A.1.: Reaktionszeit der WiringPi Interrupt-Routine; Fallende Flanke löst Interrupt aus; steigende Flanke zeigt Zeitpunkt des Eintritts an. Aufgezeichnet mit *Persistence Mode* des *PicoScope3406B*

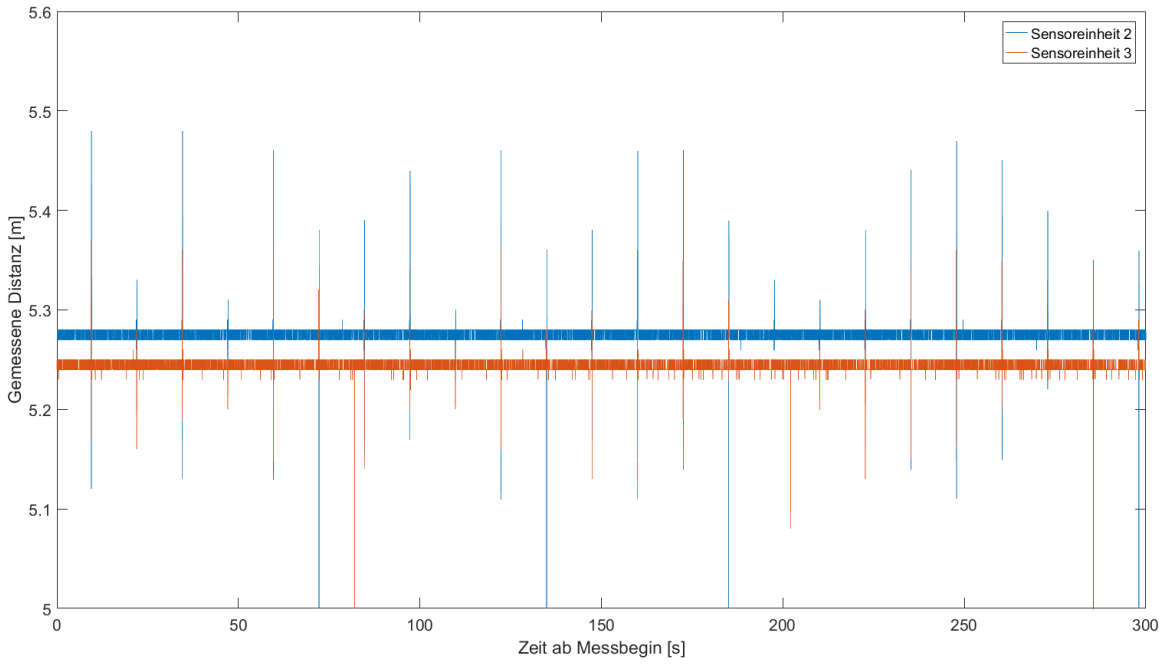

Abbildung A.2.: Interferenz zwischen parallelen LIDAR-LIDAR

B. Software

B.1. Klassendiagramme

B.1.1. Zentrale Einheit

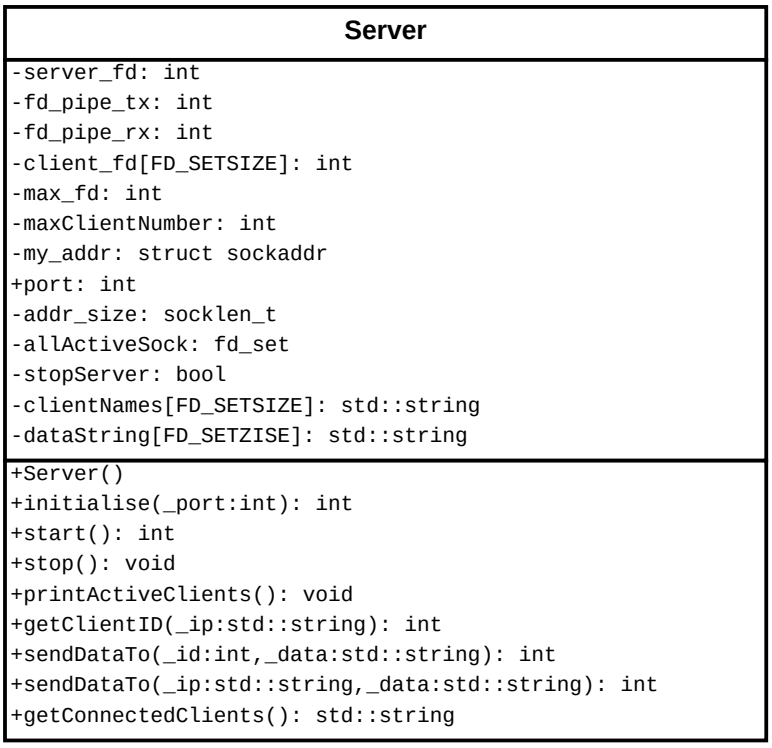

Abbildung B.1.: Klassendiagramm für Server

SensorboxServer

```
-serverLaser: Server
-serverSonar: Server
-serverCMD: Server
-fd_pipe_laser: int
-fd_pipe_laser: int
-fd_pipe_cmd: int
-fd_max: int
-laserBuf_ptr[NUMBER_OF_BOXES]: int
-sonarBuf_ptr[NUMBER_OF_BOXES]: int
-laserBufferReady: bool
-sonarBufferReady: bool
-laserSynchErrorCtr: int
-sonarSynchErrorCtr: int
-syncLaserBuf[NUMBER_OF_BOXES]: std::array<MeasurePoint_Laser, BUFSIZE>
-syncSonarBuf[NUMBER_OF_BOXES]: std::array<MeasurePoint_Sonar, BUFSIZE>
-boxNameList: std::array<std::string, NUMBER_OF_BOXES>
-boxNumberList: std::array<int, MAX_NUMBER_OF_BOXES>
-stopServer: bool
-LaserFiles[4]: std::ofstream
-SonarFiles[4]: std::ofstream
```

```
+SensorboxServer(_cmdPort:int,_laserPort:int,
                 _sonarPort:int)
+start(): void
+stop(): void
+sendCommandTo(_ip:std::string,_command:std::string): std::string
+broadcast(_dataString:std::string): int
+getLaserDataFrom(_boxNr:int,_size:int): std::vector<MeasurePoint_Laser>
+getLaserDataFrom(_boxNr0:int,_boxNr1:int,
                  _size:int): std::array<std::vector<MeasurePoint_Laser>,
                  2>
+getLaserDataFrom(_boxNr0:int,_boxNr1:int,
                  _boxNr2:int,_size:int): std::array<std::vector<MeasurePoint_Laser>,
                  3>
+getLaserDataFrom(_boxNr0,_boxNr1,_boxNr2,
                  _boxNr3,_size:int): std::array<std::vector<MeasurePoint_Laser>,
                  4>
-serverThread(server:Server): static void
-readLine(_fd:int): std::string
-addLaserData(_data:std::string): int
-addSonarData(_data:std::string): int
-identifyBoxes(): int
-checkLaserBufferSync(_buff_ptr:int): int
-checkSonarBufferSync(_buff_ptr:int): int
-getCurrentSonarBuffPtr(): int
-getCurrentLaserBuffPtr(): int
+readLaserBufferFrom(_boxNr:int,_size:int,
                     _buf_ptr:int): std::vector<MeasurePoint_laser>
-createTimestamp(): unsigned long long
```

Abbildung B.2.: Klassendiagramm für SensorboxServer

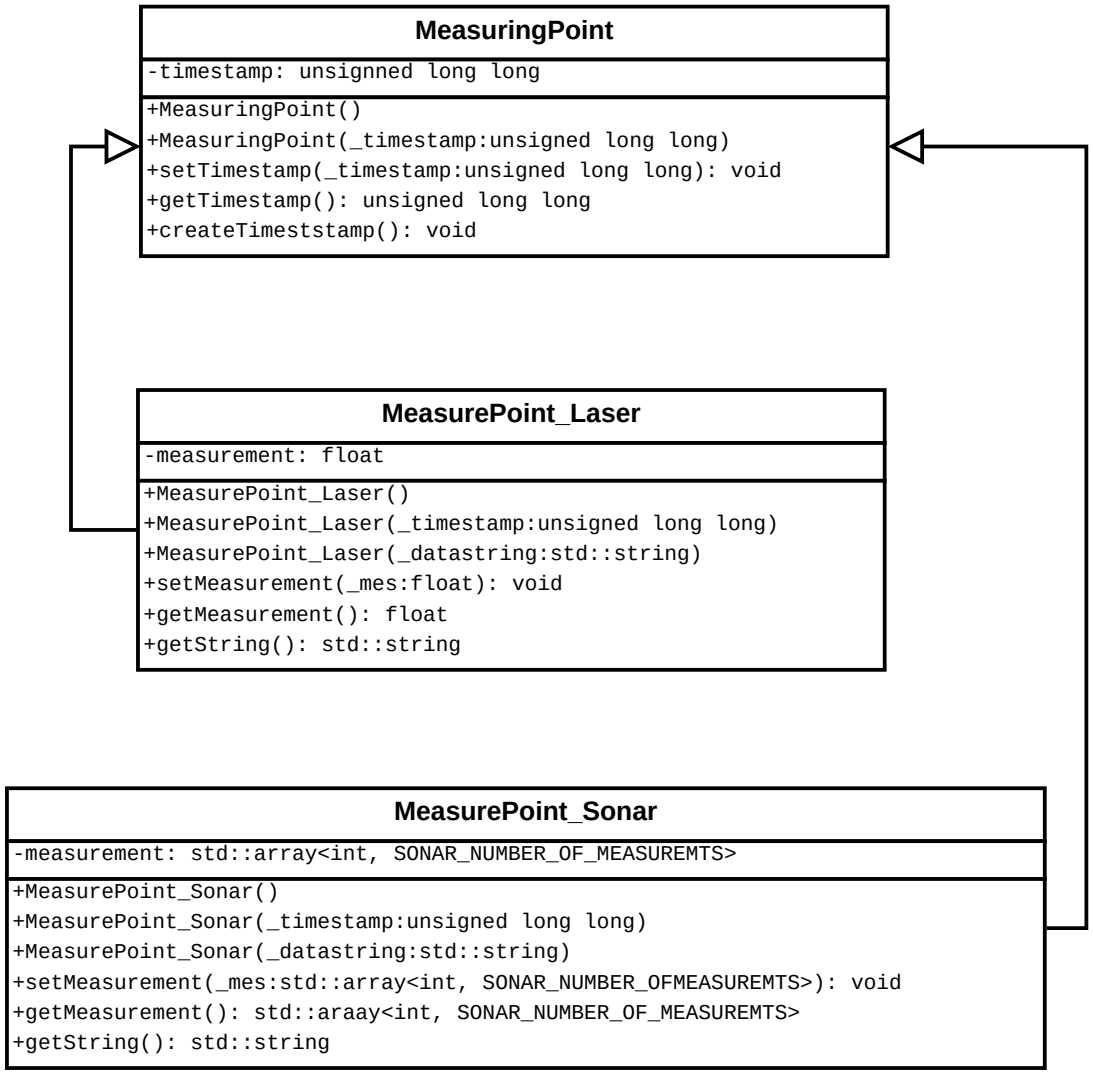

Abbildung B.3.: Klassendiagramme für MeasuringPoint, MeasurePoint_Laser und MeasurePoint_Sonar

B.1.2. Sensoreinheit

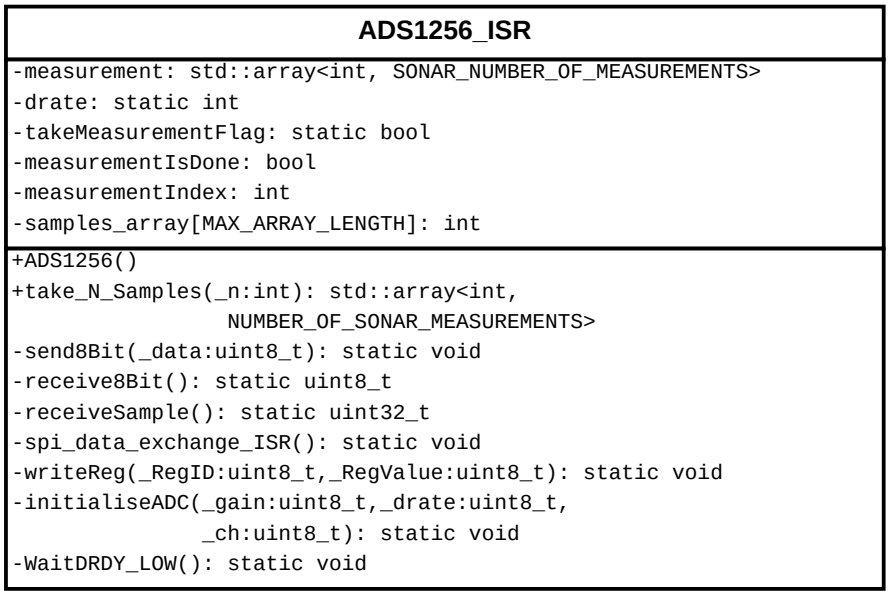

Abbildung B.4.: Klassendiagramm für ADS1256_ISR

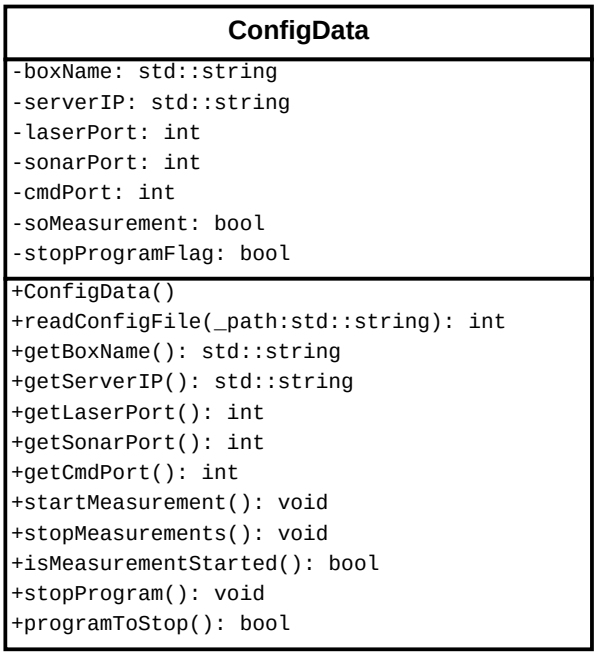

Abbildung B.5.: Klassendiagramm für ConfigData

B.2. Quellcode

B.2.1. Zentrale Einheit

Server

CPP-Datei

```
// Server.cpp
// Project: Sensorbox Server
// Brief:   This Class defines methods for a Server Program using sockets.
//          The Data is provided via Pipes, one for TX and one for RX
//
//           This program is part of the masterthesis "Konzeption
//          und Entwicklung eines hochpräzisen Systems zur
//          Positionsbestimmung von Fahrzeugen in urbanen Gebieten"
// Author:  Mario Wegner
// Date:    10/2016

#include "Server.h"

/** \brief  Constructor initialises variables and sets all client names to "empty".
 *          The maximum number of clients is defined by FD_SETSIZE from the OS
 */
Server::Server()
{
    maxClientNumber = -1;

    // Sett all Client Names (IP Addresses to empty
    for(int j = 0; j < FD_SETSIZE; j++)
    {
        clientNames[j] = "empty";
    }
}

Server::~Server()
{
    close(server_fd);
}

/** \brief  This function sets up a server on _port, also a pipe will be
 *          created. Socket will be in listen mode after running the method
 *
 * \param Port of the sever
 *
 * \return File descriptor of the created pipe
 *
 */
int Server::initialise(int _port)
{
    // Safe port number
    this->port = _port;

    // Create a Socket for Internet communication (PF_INET), and TCP Sessions (SOCK_STREAM)
    server_fd = socket(PF_INET, SOCK_STREAM, 0);
    if(server_fd == -1)
    {
        std::cout << "Server::initialise():␣Error:␣socket()" << std::endl;
        return -1;
    }

    // Set max_fd, needed for IO Multiplexing
    max_fd = server_fd;

    // set Socket options
    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(port);
    my_addr.sin_addr.s_addr = INADDR_ANY;

    // Bind Socket to port
    if(bind(server_fd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr_in)) != 0)
    {
        std::cout << "Server::initialise():␣Error:␣Bind()" << std::endl;
        return -1;
    }

    // Go to listen staus
```

```
    if(listen(server_fd, FD_SETSIZE) != 0)
    {
        std::cout << "Server::initialise():␣Error:␣Listen()" << std::endl;
        return -1;
    }

    // Clear all clients
    for(i=0; i<FD_SETSIZE; i++)
    {
        client_fd[i] = -1;
    }

    // IO Multiplexing set server_fd
    FD_ZERO(&allActiceSock); // Clear all active socks
    FD_SET(server_fd,&allActiceSock); // add server sock



    int fd_pipe[2];

    // Create two pipes for RX and TX
    if(pipe(fd_pipe) == -1)
    {
        std::cout << "Server::initialise():␣Error:␣pipe()" << std::endl;
        return -1;
    }

    // Set rx pipe
    fd_pipe_rx = fd_pipe[0];
    fcntl(fd_pipe_rx, F_SETFL, O_NONBLOCK); // Don't block
    // Check if new fd is new max
    if(fd_pipe_rx > max_fd)
    {
        max_fd = fd_pipe_rx;
    }
    // set tx pipe
    fd_pipe_tx = fd_pipe[1];
    //fcntl(fd_pipe_tx, F_SETFL, O_NONBLOCK);
    if(fd_pipe_rx > max_fd)
    {
        max_fd = fd_pipe_tx;
    }

    return fd_pipe_rx;
}

/** \brief  This function is the main server function, it is a non returning function
 *          handling Clients (register and close) and Data. It will run until ::stop()
 *          is called
 */

int Server::start()
{
    // init Variables
    stopServer = false;
    char * buffer;
    struct sockaddr_in client_addr;

    // allocate space for receive buffer
    buffer = (char*) malloc ((sizeof(char) * BUF) + 1);

    // Main while loop (tob be stopped by ::stop())
    while(stopServer == false)
    {
        int newsock_fd, readsock_fd;
        int length;
        int numberOfReadyFD;

        fd_set tempSock = allActiceSock;

        //std::cout << "Select in" << std::endl;

        // IO Multiplexing: Wait for Data, new Clients or closed clients
        numberOfReadyFD = select(max_fd+1, &tempSock, NULL, NULL, NULL);

        //std::cout << numberOfReadyFD << std::endl;

        // accept new Client
        if(FD_ISSET(server_fd, &tempSock))
        {
            addr_size = sizeof(struct sockaddr_in);

            // accept new client
            newsock_fd = accept(server_fd, (struct sockaddr *)&client_addr, &addr_size);
            // add new client fd to client list
```

```
    // run through complete list
    for(i = 0; FD_SETSIZE; i++)
    {
        // check if entry is free
        if(client_fd[i] < 0)
        {
            // add client
            client_fd[i] = newsock_fd;
            break;
        }
    }
    // add new client to IO Multiplexing list
    FD_SET(newsock_fd, &allActiceSock);

    // update max fd
    if(newsock_fd > max_fd)
    {
        max_fd = newsock_fd;
    }

    // Safe Client IP-Address
    clientNames[i] = inet_ntoa(client_addr.sin_addr);
    std::cout << "Client␣connected␣Nr.:␣" << i << "␣at␣" << inet_ntoa(client_addr.sin_addr) << std::endl;

    // Needed for next for-loop
    if(i > maxClientNumber)
        maxClientNumber = i;


    if(--numberOfReadyFD <= 0)
        continue; // End for
}

// receive new data, check every client for new data
for(i=0; i<=maxClientNumber; i++)
{
    // check if client is registered at this element
    if((readsock_fd = client_fd[i]) < 0)
        continue;

    // Check if this client received data
    if(FD_ISSET(readsock_fd, &tempSock))
    {

        // read data from client
        length = read(readsock_fd, buffer, BUF);
        // Check if client send empty message -> connection closed by client
        if(length == 0)
        {
            // close connection
            close(readsock_fd);
            FD_CLR(readsock_fd, &allActiceSock);
            // delete from client list
            client_fd[i] = -1;
            clientNames[i] = "empty";
            std::cout << "Client␣closed:␣Nr:␣" << i << std::endl;
        }
        // else read data
        else
        {
            // convert to std::string
            std::string data(buffer,length);

            // handle complete data string
            while(data.empty() == false)
            {
                // find linebreak
                int pos = data.find("\r\n");

                // if lineend found
                if(pos != std::string::npos)
                {
                    // add data to client based std::string
                    dataString[i] += data.substr(0,pos+2);

                    char * sendString = new char [dataString[i].length()+1];
                    std::strcpy (sendString, dataString[i].c_str());

                    // write string to pipe
                    write(fd_pipe_tx, sendString, dataString[i].length());
                    free(sendString);
                    // delete handled data from received string
                    data.erase(0,pos+2);
                    // clear client based string
```

```
                            dataString[i].clear();
                            }
                        // if no linebreak found
                        else
                        {
                            // add data to client based string
                            dataString[i] += data;
                            // all data is now handled -> clear string
                            data.clear();
                        }
                    }
                }
                // all actions handled?
                if(--numberOfReadyFD <= 0)
                    break;
            }
        }
    }
    return 0;
}


/** \brief This function prints all registered Client IPs
 *
 *
 */

void Server::printActiveClients()
{
    for(int j = 0; j<=maxClientNumber; j++)
    {
        if(clientNames[j] != "empty")
            std::cout << clientNames[j] << std::endl;
    }
}

/** \brief This function is a getter for the ID of a client
 *
 * \param IP address of the client
 * \return client ID
 *
 */

int Server::getClientID(std::string _clientIP)
{
    // compare IPs and return with ID
    for(int j = 0; j <= maxClientNumber; j++)
    {
        if(clientNames[j] == _clientIP)
            return j;
    }
    return -1;
}

/** \brief This Function is used to send Data to a specific Client identified by client id
 *
 * \param Client ID
 * \param  Data to be send
 * \return Number of Bytes send
 *
 */
int Server::sendDataTo(int _id, std::string _dataString)
{
    int length;

    // convert std::string to c-string
    char * sendString = new char [_dataString.length()+1];
    std::strcpy (sendString, _dataString.c_str());

    // Send data to client
    length = send(client_fd[_id], sendString, _dataString.length()+1,MSG_DONTWAIT); // Non waiting if TCP Buffer full
    free(sendString);
    //std::cout << _dataString << " to id = " << _id << std::endl;
    if(length == -1)
    {
        std::cout << "Client::sendData:␣sending␣failed!" << std::endl;
    }
    return length;
}

/** \brief  This function is used to send Data to a specific Client identified by client IP
 *
 * \param IP of the client
 * \param Data to be send
 * \return Number of Bytes send
```

```
 *
 */

int Server::sendDataTo(std::string _ip, std::string _dataString)
{
    // get the client ID
    int id = this->getClientID(_ip);

    int length = this->sendDataTo(id, _dataString);

    return length;
}

/** \brief This function is a getter for all connected Client IP
 *
 * \return std::string of all connected clients separated by linebreak (\r\n)
 *
 */
std::string Server::getConnectedClients()
{
    std::string clientIPs;

    // run through all clients and add IP to string
    for(int j = 0; j<FD_SETSIZE; j++)
    {
        if(clientNames[j] != "empty")
        {
            clientIPs += clientNames[j];
            clientIPs += "\r\n";
        }
    }
    return clientIPs;
}

/** \brief This function is used to stop the server, all clients will be deleted atart-loop
 *          will be stopped
 *
 */

void Server::stop()
{
    stopServer = true;
    // clear all clients
    for(int i=0; i< FD_SETSIZE; i++)
    {
        close(client_fd[i]);
        client_fd[i] = -1;
        clientNames[i] = "empty";
    }
    // close server
    close(server_fd);
}
```

Header-Datei

```
#ifndef SERVER_H
#define SERVER_H

#include <iostream>
#include <unistd.h>
#include <stdlib.h>
#include <queue>
#include <string>
#include <cstring>
#include <fcntl.h>

#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define BUF 16383


class Server
{
    public:
        Server();
        virtual ~Server();
        int initialise(int);
        int start();
        void printActiveClients();
        int getClientID(std::string);
        int sendDataTo(int, std::string);
```

```
        int sendDataTo(std::string, std::string);
        std::string getConnectedClients();
        void stop();

    private:
        int         server_fd;
        int         fd_pipe_tx, fd_pipe_rx;
        int         client_fd[FD_SETSIZE];
        int         max_fd, i, maxClientNumber;
        struct      sockaddr_in my_addr;
        int         port;
        socklen_t   addr_size;
        fd_set      allActiceSock;
        //struct      timeval tv;
        bool        stopServer;
        std::string clientNames[FD_SETSIZE];
        std::string dataString[FD_SETSIZE];
};

#endif // SERVER_H
```

SensorboxServer

CPP-Datei

```
// SensorboxServer.cpp
// Project: Sensorbox Server
// Brief:   This Class defines methods for a server program to receive and process
//          the data from multiple SensorBoxes
//
//
//           This program is part of the masterthesis "Konzeption
//          und Entwicklung eines hochpräzisen Systems zur
//          Positionsbestimmung von Fahrzeugen in urbanen Gebieten"
// Author:  Mario Wegner
// Date:    10/2016

#include "SensorboxServer.h"

/** \brief  Constructor: Creates objects of type "Server", there will be one Server
 *          for every sensor type and one additional command server.
 *
 * \param Portnumber for the Command Server
 * \param Portnumber for the laser measurements
 * \param Portnumber for the sonar measurements
 *
 */
SensorboxServer::SensorboxServer(int _cmdPort, int _laserPort, int _sonarPort)
{
    std::cout << "Initialise␣Server..."  << std::endl;

    // Set BufferFlags not ready
    laserBufferReady = false;
    sonarBufferReady = false;

    // Clear Box names on init
    for(int j = 0; j<NUMBER_OF_BOXES; j++)
    {
        boxNameList[j] = "empty";
        boxNumberList[j] = -1;
    }


    // Initialise Command Server ----------------------------------------
    fd_pipe_cmd =  serverCMD.initialise(_cmdPort);
    // Check if Pipe has been created
    if(fd_pipe_cmd == -1)
    {
        // If not stop
        std::cout <<  "Set␣up␣Laser-Server␣failed!" << std::endl;
        return;
    }
    // fd_max is needed to use select() later
    fd_max = fd_pipe_cmd;



    // Initialise Laser Server ------------------------------------------
```

```
    fd_pipe_laser = serverLaser.initialise(_laserPort);
    // Check if Pipe has been created
    if(fd_pipe_laser == -1)
    {
        // If not stop
        std::cout << "Set up Laser-Server failed!" << std::endl;
        return;
    }
    // If the new pipe fd is higher then the current fd_max
    // set this pipe fd as new max
    if(fd_pipe_laser > fd_max)
    {
        fd_max = fd_pipe_laser;
    }



    // Initialise Sonar Server ---------------------------------------------
    fd_pipe_sonar = serverSonar.initialise(_sonarPort);
    // Check if Pipe has been created
    if(fd_pipe_sonar == -1)
    {
        // If not stop
        std::cout << "Set up Sonar-Server failed!" << std::endl;
        return;
    }
    // If the new pipe fd is higher then the current fd_max
    // set this pipe fd as new max
    if(fd_pipe_sonar > fd_max)
    {
        fd_max = fd_pipe_sonar;
    }

    std::cout << "Server ready!" << std::endl;
}

SensorboxServer::~SensorboxServer()
{
    //dtor
}


/** \brief This is the main function of this class, is sets up the server and wait for
 *          the boxes to connect, than it will read data from the pipes and handles commands
 *
 */
void SensorboxServer::start()
{
    stopServer = false;
    std::cout << "Start" << std::endl;
    fd_set activeSock;
    std::string data;

    // Reset Buffer Pointer
    for(int i = 0; i<NUMBER_OF_BOXES; i++)
    {
        laserBuf_ptr[i] = 0;
        sonarBuf_prt[i] = 0;
    }

    // Sets FD_SET to laser and sonar pipe, this is later needed
    // to uses IO multiplexing (select())
    FD_ZERO(&activeSock);
    FD_SET(fd_pipe_laser, &activeSock);
    FD_SET(fd_pipe_sonar, &activeSock);

    // Start Server Threads, 1 Laser, 1 Sonar, 1 Command
    std::thread tLaser(serverThread, std::ref(serverLaser));
    std::thread tSonar(serverThread, std::ref(serverSonar));
    std::thread tCMD(serverThread, std::ref(serverCMD));

    int anz = 0;

    // Wait until all boxes are connected
    while(anz < NUMBER_OF_BOXES)
    {
        sleep(1);
        // Identify all momentarily connected boxes
        anz = identifyBoxes();
    }

    std::cout << "Boxes connected: " << anz << std::endl;


    // Get actual time
```

```
struct timeval tv;
gettimeofday(&tv, NULL);

unsigned long long time =
    (unsigned long long)(tv.tv_sec) * 1000 +
    (unsigned long long)(tv.tv_usec) / 1000;


// Create files, one for each box and sensor
LaserFiles[0].open("/home/pi/Data_Laser/" + std::to_string(time) + "_0_Laser.txt");
LaserFiles[1].open("/home/pi/Data_Laser/" + std::to_string(time) + "_1_Laser.txt");
LaserFiles[2].open("/home/pi/Data_Laser/" + std::to_string(time) + "_2_Laser.txt");
LaserFiles[3].open("/home/pi/Data_Laser/" + std::to_string(time) + "_3_Laser.txt");

SonarFiles[0].open("/home/pi/Data_Sonar/" + std::to_string(time) + "_0_Sonar.txt");
SonarFiles[1].open("/home/pi/Data_Sonar/" + std::to_string(time) + "_1_Sonar.txt");
SonarFiles[2].open("/home/pi/Data_Sonar/" + std::to_string(time) + "_2_Sonar.txt");
SonarFiles[3].open("/home/pi/Data_Sonar/" + std::to_string(time) + "_3_Sonar.txt");

broadcast("StartMeasurement");

//
while(stopServer == false)
{

    fd_set tempSock = activeSock;

    // IO Multiplexing waiting for data from Sonar or Laser Pipe
    select(fd_max+1,&tempSock, NULL, NULL, NULL);

    // Check for Laser Data
    if(FD_ISSET(fd_pipe_laser, &tempSock))
    {
        //std::cout << "while Laser" << std::endl;

        // Read Laser Data line by line until pipe is cleard
        while(true)
        {
            // Read line from pipe
            data = readLine(fd_pipe_laser);
            //std::cout << "readline: " << runTime<< std::endl;


            // Break if no data in pipe
            if(data == "")
            {
                //std::cout << "Laser break" << std::endl;
                break;
            }

            // Add data to buffer, safe data to file
            addLaserData(data);
            //std::cout << "Laser Data: " << runTime<< std::endl;

        }

    }

    // Check for Sonar data
    if(FD_ISSET(fd_pipe_sonar, &tempSock))
    {
        // Read Sonar data line by line
        while(true)
        {
            // Read line from pipe
            data = readLine(fd_pipe_sonar);

            // Break if pipe is empty
            if(data == "")
            {
                //std::cout << "Sonar break" << std::endl;
                break;
            }
            // Add data to buffer, safe data to files
            //std::cout << "Sonar Data: " << data;
            addSonarData(data);
        }
    }
    //std::cout << " * "<< std::endl;
}
// End While

// Close Files
LaserFiles[0].close();
LaserFiles[1].close();
```

```
    LaserFiles[2].close();
    LaserFiles[3].close();

    SonarFiles[0].close();
    SonarFiles[1].close();
    SonarFiles[2].close();
    SonarFiles[3].close();

    // Stop Server
    serverCMD.stop();
    serverLaser.stop();
    serverSonar.stop();


}


/** \brief This function is used to run server as thread
 *
 * \param Reference of Server Object
 *
 */
void SensorboxServer::serverThread(Server& server)
{
    // Start the Server
    server.start();
}

/** \brief  This function reads an ASCII line from a File descriptor. A
 *          line is detected by receiving the symbol "\n"
 *
 * \param File descriptor to be read
 * \return std::string line
 *
 */
std::string SensorboxServer::readLine(int _fd)
{
    // Init Variable
    int anz;                // Number of received Bytes
    char c = 'x';           // Reveive Buffer for one char
    std::string line = "";  // Received String

    // Read char by char until return
    while(true)
    {
        // Read one char
        anz = read(_fd,&c,1);
        //std::cout << anz << " ";
        // If one char has been received
        if(anz == 1)
        {
            // Check if line end
            if(c == '\n')
            {
                // If yes, add to string and return the string
                line += c;
                //std::cout << std::endl;
                return line;
            }
            // If no end line received, add char to string
            else
            {
                line += c;
            }
        }
        // If no data could be received (pipe empty) return an empty string
        else
        {
            //std::cout << std::endl;
            return "";
        }
    }
}


/** \brief  This function converts a string to a MeasurePoint_Laser and adds it
 *          to a box specific ringbuffer. Also the Boxnumber will be read. The function
 *          expects a string in the following form:
 *
 *          #1:1465986486184;6.456
 *              - #1: = Boxnumber (here 1)
 *              - 1465986486184 = Timestamp of measurement
 *              - 6.456 = Distance measurement as float
 *
```

```
 * \param std::string with data
 * \return  0 if data was added
 *          -1 if an error appears
 *
 */
int SensorboxServer::addLaserData(std::string _data)
{
    long long start = createTimestamp();
    // Init variables
    int pos, boxNr, temp_ptr;

    // Find the symbol "#" to identify the boxnumber
    pos = _data.find("#"); // Find
    if(pos != std::string::npos)
    {
        // If "#" has been found
        boxNr = _data[pos+1] - '0';     // Convert Char to Integer
        _data.erase(pos,3);             // erase Boxname from string

        // Safe Data to file
        LaserFiles[boxNr] << _data;

        laserBuf_ptr[boxNumberList[boxNr]]++;
        if(laserBuf_ptr[boxNumberList[boxNr]]>= BUFSIZE)
        {
            laserBuf_ptr[boxNumberList[boxNr]] = 0;
            laserBufferReady = true;
        }
        // Safe data to box-specific ringbuffer
        temp_ptr = laserBuf_ptr[boxNumberList[boxNr]];

        syncLaserBuf[boxNumberList[boxNr]][temp_ptr] = MeasurePoint_Laser(_data);

        // Calculate time needed for data transportation
        long long transpTime = createTimestamp() - syncLaserBuf[boxNumberList[boxNr]][temp_ptr].getTimestamp();

        /*
        if(transpTime > 100)
        {
            std::cout << " Transp: " << transpTime << std::endl;
            std::cout << _data << std::endl;
        }
        */

        // Check if buffer are synchronous
        checkLaserBufferSync(temp_ptr);
    }
    // If no "#" was found, return with error
    else
    {
        std::cout << "Error:␣Sensorbox::addLaserData:␣Could␣not␣identify␣boxnumber␣from:␣" << _data << std::endl;
        return -1;

    }
    return 0;
}

/** \brief  This function converts a string to a MeasurePoint_Sonar and adds it
 *          to a box specific ringbuffer. Also the Boxnumber will be read. The function
 *          expects a string in the following form:
 *
 *          #1:1465986486184;245064,243563 ... 343564
 *              - #1: = Boxnumber (here 1)
 *              - 1465986486184 = Timestamp of measurement
 *              - 245064,243563 ... 343564 = all analog measurements
 *
 * \param std::string with data
 * \return  0 if data was added
 *          -1 if an error appears
 *
 */
int SensorboxServer::addSonarData(std::string _data)
{
    // Init variables
    int pos, boxNr, temp_ptr;

    // Find the symbol "#" to identify the boxnumber
    pos = _data.find("#"); // Find


    if(pos != std::string::npos)
    {
        // If "#" has been found, extract and convert boxnumber an erase
        // the box-identifier from string
        boxNr = _data[pos+1] - '0'; // Convert Char to Integer
```

```
        _data.erase(pos,3);

        // Safe data to file
        SonarFiles[boxNr] << _data;

        sonarBuf_prt[boxNumberList[boxNr]]++;
        if(sonarBuf_prt[boxNumberList[boxNr]]>= BUFSIZE)
        {
            sonarBuf_prt[boxNumberList[boxNr]] = 0;
            sonarBufferReady = true;
        }
        // Safe data to box-specific ringbuffer
        temp_ptr = sonarBuf_prt[boxNumberList[boxNr]];
        syncSonarBuf[boxNumberList[boxNr]][temp_ptr] = MeasurePoint_Sonar(_data);

        // Check if Buffer are synchronous
        //checkSonarBufferSync(temp_ptr);
    }
    // If no "#" was found, return with error
    else
    {
        std::cout << "Error: Sensorbox::addSonarData: Could not identify boxnumber from: " << _data << std::endl;
        return -1;
        //throw "Error";
    }
    return 0;
}

/** \brief  This function can be used to connect the boxnumber with the IP-adress of the
 *          Sensorboxes. The function send an "idetify command" to every connected client.
 *
 *
 * \return int: Number of boxes that were identified
 *
 */
int SensorboxServer::identifyBoxes()
{
    // Init variable
    int boxesConnected = 0;
    int pos;
    // Set answerstring: This is the expected answer replied by the client
    std::string answerstring = "My name is ";

    // clear the list that hold the boxnumber-IP connection
    for(int j = 0; j<NUMBER_OF_BOXES; j++)
    {
        boxNameList[j] = "empty";
        boxNumberList[j] = -1;
    }

    // Call for every connected IP. IPs will be separated by a linebreak
    std::string names = serverCMD.getConnectedClients();


    int boxNumberCtr = 0;
    while(names.empty() == false)
    {
        // Find linebreak
        pos = names.find("\r\n");
        if(pos != std::string::npos)
        {
            // If linebreak was found, extract IP and erase from string
            std::string ip = names.substr(0,pos);
            names.erase(0,pos+2);

            // Sens "Identify" command to extracted IP
            std::string response = sendCommandTo(ip,"Identify");
            //std::cout << "Response: " << response << std::endl;

            // Find answerstring in response
            int pos = response.find(answerstring);
            if(pos != std::string::npos)
            {
                // If answerstring was found, extract boxname
                response.erase(0,pos+answerstring.length());

                // Convert boxname to integer and add ip to list
                try
                {
                    int boxNumber = std::stoi(response);
                    boxNumberList[boxNumber] = boxNumberCtr;
                    boxNameList[boxNumberCtr] = ip;
                    //std::cout << "Client " << ip << " connected as: '" << boxNumber << "'" << std::endl;
                    boxesConnected++;
                    boxNumberCtr++;
```

```
                }
                catch(...)
                {
                    std::cout << "Error: SensorboxServer::identifyBoxes: Could not resolve boxname: '" << response << "'"
                          << std::endl;
                }
            }
            else
            {
                // If answerstring was not found, go to next IP
                std::cout << "Error:SensorboxServer::identifyBoxes: No suitable answer received: '" << response << "'" <<
                      std::endl;
            }
        }
        else
        {
            return -1;
        }
    }
    return boxesConnected;
}

/** \brief  This function sends a command to a sensorbox. The function will wait 3 sec for
 *          an answer from the box.
 *
 * \param std::string IP address of the sensorbox
 * \param std::string Commandmesssage
 * \return std::string  Answer from Sensorbox
 *                      "NACK" after timeout
 *
 */
std::string SensorboxServer::sendCommandTo(std::string ip, std::string _dataString)
{
    // Set Timeout to 3 sec
    struct timeval timeout;
    timeout.tv_sec = 3;
    timeout.tv_usec = 0;

    // Send command string to ip-address
    serverCMD.sendDataTo(ip, _dataString);

    // Prepare select()
    fd_set cmd_fdset;

    FD_ZERO(&cmd_fdset);
    FD_SET(fd_pipe_cmd, &cmd_fdset);

    // Wait for answer with 3 sec timeout
    select(fd_pipe_cmd+1,&cmd_fdset, NULL, NULL, &timeout);

    // Check for data in pipe from cmd server
    if(FD_ISSET(fd_pipe_cmd, &cmd_fdset))
    {
        // read line from pipe
        std::string answer = readLine(fd_pipe_cmd);
        return answer;
    }
    // If no data in pipe, eg. Timeout
    return "NACK";
}

/** \brief  This function, like "sendCommandTo" sends a command, but this time to all
 *          connected sensorboxes. The difference is, that all commands will be send
 *          without waiting for an answer. Only works for commands that expect an "ACK"
 *          from the serverbox.
 *
 * \param std::string Command
 * \return  int 0 if broadcast was successful
 *              -1 if one or more "ACK" were missing
 *
 */
int SensorboxServer::broadcast(std::string _dataString)
{
    // Init variables
    int j, messagesSend = 0, messagesReceived = 0;
    fd_set cmd_fdset;
    std::string ip;

    // Set Timeout to 3 sec
    struct timeval timeout;
    timeout.tv_sec = 3;
    timeout.tv_usec = 0;

    // Send data to all known boxes
    for(j = 0; j < NUMBER_OF_BOXES; j++)
```

```
    {
        ip = boxNameList[j];
        // Check if list entry is not empty
        if(ip != "empty")
        {
            // Send data to box
            serverCMD.sendDataTo(ip, _dataString);
            messagesSend++;
        }
    }

    // Prepare select()
    FD_ZERO(&cmd_fdset);
    FD_SET(fd_pipe_cmd, &cmd_fdset);

    while(messagesSend <= messagesReceived)
    {
        // Wait for answer
        int rec = select(fd_pipe_cmd+1,&cmd_fdset, NULL, NULL, &timeout);
        // Read all answers (if more than one answers in buffer
        for(j = 0; j < rec; j++)
        {
            if(FD_ISSET(fd_pipe_cmd, &cmd_fdset))
            {
                std::string answer = readLine(fd_pipe_cmd);
                // Check for "ACK"
                if(answer == "ACK")
                {
                    messagesReceived++;
                }
            }
            else
            {
                // If something else or nothing received, return with error
                return -1;
            }
        }
    }
    return 0;
}

/** \brief  This function checks if the laserMeasurementBuffer are still synchronised
 *          by checking the timestamps of the previous entries. If the time distance is
 *          bigger than expected for 'MAX_SYNC_ERROR' times in a row the buffer pointer will be set to
 *           0 and the laserBufferReady flag will be set false.
 *
 * \param int current buffer pointer

 * \return int  0  if buffer are synchronised
 *              -1 if buffer has been reseted
 *
 */
int SensorboxServer::checkLaserBufferSync(int _buff_ptr)
{
        unsigned long long timestamp[NUMBER_OF_BOXES];
        long long difftime;

    int laserSyncSucsessCtr = 0;
    // Zero pointer will not be checked to prevent a deadlock after buffer reset
    if(_buff_ptr != 0)
    {
        // Get the previous timestamp of the first box in buffer
        timestamp[0] = syncLaserBuf[0][_buff_ptr-1].getTimestamp();

        // Run for all boxes starting with one
        for(int i = 1; i < NUMBER_OF_BOXES; i++)
        {
            // Get previous timestamp
            timestamp[i] = syncLaserBuf[i][_buff_ptr-1].getTimestamp();
            // Calculate time difference
            difftime = (timestamp[i] - timestamp[0]);
            //std::cout << i << " - " << 0 << " = " << difftime << std::endl;
            //std::cout << i << " - 0 = " << difftime << std::endl;
            if(std::abs(difftime) > LASER_SAMPLE_PERIOD_ms)
            {
                // If time difference is to big: increase Error counter
                laserSynchErrorCtr++;
                //std::cout << "Synch Error Laser! " << laserSynchErrorCtr << std::endl;
                if(laserSynchErrorCtr >= MAX_SYNC_ERROR)
                {
                    // Reset all laserBuffer pointer and lock buffer
                    laserBufferReady = false;
                    laserSynchErrorCtr = 0;
                    std::cout << "SensorboxServer::checkLaserBufferSync():␣Buffer␣out␣of␣sync␣->␣reset␣buffer;␣
                          Timedifference:␣" << difftime << "␣ms" << std::endl;
```

```
                    for(int j = 0; j < NUMBER_OF_BOXES; j++)
                    {
                        laserBuf_ptr[j] = 0;
                    }
                    return -1;
                }
            }
            else
            {
                laserSyncSucsessCtr++;
            }

        }
        if(laserSyncSucsessCtr >= NUMBER_OF_BOXES-1)
        {
            laserSynchErrorCtr = 0;
        }
        //std::cout << std::endl;
    }
    return 0;
}

/** \brief  This function checks if the sonarMeasurementBuffer are still synchronised
 *          by checking the timestamps of the previous entries. If the time distance is
 *          bigger than expected for 'MAX_SYNC_ERROR' times in a row the buffer pointer will be set to
 *           0 and the sonarBufferReady flag will be set false.
 *
 * \param int current buffer pointer

 * \return int  0  if buffer are synchronised
 *              -1 if buffer has been reseted
 *
 */
int SensorboxServer::checkSonarBufferSync(int _buff_ptr)
{
        unsigned long long timestamp[NUMBER_OF_BOXES];
                 long long difftime;
    int sonarSyncSucsessCtr = 0;
    // Zero pointer will not be checked to prevent a deadlock after buffer reset
    if(_buff_ptr != 0)
    {
        // Get the previous timestamp of the first box in array
        timestamp[0] = syncSonarBuf[0][_buff_ptr-1].getTimestamp();

        // Run for all boxes starting with one
        for(int i = 1; i < NUMBER_OF_BOXES; i++)
        {
            // Get previous timestamp
            timestamp[i] = syncSonarBuf[i][_buff_ptr-1].getTimestamp();
            // Calculate time difference
            difftime = (timestamp[i] - timestamp[0]);
            //std::cout << i << " - 0 = " << difftime << std::endl;

            if(std::abs(difftime) > SONAR_SAMPLE_PERIOD_ms)
            {
                // If time difference is to big: increase Error counter
                sonarSynchErrorCtr++;
                //std::cout << "Synch Error Sonar! " << sonarSynchErrorCtr << std::endl;
                if(sonarSynchErrorCtr >= MAX_SYNC_ERROR)
                {
                    // Reset all laserBuffer pointer and lock buffer
                    sonarBufferReady = false;
                    sonarSynchErrorCtr = 0;
                    std::cout << "SensorboxServer::checkSonarBufferSync():␣Buffer␣out␣of␣sync␣->␣reset␣buffer;␣
                         Timedifference:␣" << difftime << "␣ms" << std::endl;
                    for(int j = 0; j < NUMBER_OF_BOXES; j++)
                    {
                        sonarBuf_prt[j] = 0;
                    }
                    return -1;
                }
            }
            else
            {
                // If time difference is ok, reset the error counter
                sonarSynchErrorCtr++;
            }
        }
        if(sonarSyncSucsessCtr >= NUMBER_OF_BOXES-1)
        {
            sonarSynchErrorCtr = 0;
        }
    }
    return 0;
```

```
}

/** \brief  This function is used to find the latest Buffer pointer from all laser measurements
 *          The latest pointer means the pointer, where data from ALL boxes is available
 *
 * \return Pointer number of latest synchronised Buffer entry
 *
 */

int SensorboxServer::getCurrentLaserBuffPtr()
{
    // Find smallest buffer pointer
    int min_buff_ptr = BUFSIZE;         // Set Pointer to max
    int temp_ptr;

    // Run through all boxes
    for(int i = 0; i < NUMBER_OF_BOXES; i++)
    {
        temp_ptr = laserBuf_ptr[i];
        // Check if pointer is near buffer overflow
        if(temp_ptr + MAX_SYNC_ERROR > BUFSIZE)
        {
            // If yes, pointer is negativ
            temp_ptr = temp_ptr - BUFSIZE;
        }
        // Is current pointer the smallest?
        if(temp_ptr < min_buff_ptr)
        {
            min_buff_ptr = temp_ptr;
        }
    }
    // Make negative pointer positive
    if(min_buff_ptr < 0)
    {
        min_buff_ptr = min_buff_ptr + BUFSIZE;
    }

    return min_buff_ptr;
}

/** \brief  This function is used to find the latest Buffer pointer from all laser measurements
 *          The latest pointer means the pointer, where data from ALL boxes is available
 *
 * \return Pointer number of latest synchronised Buffer entry
 *
 */
int SensorboxServer::getCurrentSonarBuffPtr()
{
    // Find smallest buffer pointer
    int min_buff_ptr = BUFSIZE;         // Set Pointer to max
    int temp_ptr;

    for(int i = 0; i < NUMBER_OF_BOXES; i++)
    {
        temp_ptr = sonarBuf_prt[i];
        // Check if pointer is near buffer overflow
        if(temp_ptr + MAX_SYNC_ERROR > BUFSIZE)
        {
            // If yes, pointer is negativ
            temp_ptr = temp_ptr - BUFSIZE;
        }
        // Is current pointer the smallest?
        if(temp_ptr < min_buff_ptr)
        {
            min_buff_ptr = temp_ptr;
        }
    }
    // Make negative pointer positive
    if(min_buff_ptr < 0)
    {
        min_buff_ptr = min_buff_ptr + BUFSIZE;
    }

}


/** \brief  This function is a getter for the Laser data buffer for ONE Box, the requested array length
 *          may not be higher than the buffer size
 *
 * \param Number of the box the data is needed from
 * \param  Number of Data points needed
 * \return std::vector with latest data points
 *
 */
std::vector<MeasurePoint_Laser> SensorboxServer::getLaserDataFrom(int _boxNr, int _size)
```

```
{
    std::vector<MeasurePoint_Laser> measurements;
    int current_ptr;

    // Check if given size is not bigger than the buffer
    if(_size > (BUFSIZE - MAX_SYNC_ERROR))
    {
        throw "Error!";
    }

    // Check if Buffer full
    //std::cout << "Buff ready " << laserBufferReady << std::endl;
    if(laserBufferReady == false)
    {
        throw "Error";
    }

    // Get the current buffer pointer
    current_ptr = getCurrentLaserBuffPtr();

    // Read Data from buffer
    measurements = readLaserBufferFrom(_boxNr, _size, current_ptr);

    return measurements;
}

/** \brief  This function is a getter for the Laser data buffer for TWO Boxes, the requested array length
 *          may not be higher than the buffer size
 *
 * \param Number of the box the data is needed from
 * \param  Number of Data points needed
 * \return std::array of std::vector with latest data points
 *
 */
std::array<std::vector<MeasurePoint_Laser>,2> SensorboxServer::getLaserDataFrom(int _boxNr0, int _boxNr1, int _size)
{
    std::array<std::vector<MeasurePoint_Laser>,2> measurements;
    int current_ptr;

    // Check if given size is not bigger than the buffer
    if(_size > (BUFSIZE - MAX_SYNC_ERROR))
    {
        throw "Error!";
    }

    // Check if Buffer full
    //std::cout << "Buff ready " << laserBufferReady << std::endl;
    if(laserBufferReady == false)
    {
        throw "Error";
    }

    // Get current buffer pointer
    current_ptr = getCurrentLaserBuffPtr();

    // Read data from buffer
    measurements[0] = readLaserBufferFrom(_boxNr0, _size, current_ptr);
    measurements[1] = readLaserBufferFrom(_boxNr1, _size, current_ptr);

    return measurements;
}

/** \brief  This function is a getter for the Laser data buffer for THREE Boxes, the requested array length
 *          may not be higher than the buffer size
 *
 * \param Number of the box the data is needed from
 * \param  Number of Data points needed
 * \return std::array of std::vector with latest data points
 *
 */
std::array<std::vector<MeasurePoint_Laser>,3> SensorboxServer::getLaserDataFrom(int _boxNr0, int _boxNr1, int _boxNr2,
      int _size)
{
    std::array<std::vector<MeasurePoint_Laser>,3> measurements;
    int current_ptr;

    // Check if given size is not bigger than the buffer
    if(_size > (BUFSIZE - MAX_SYNC_ERROR))
    {
        throw "Error!";
    }

    // Check if Buffer full
    //std::cout << "Buff ready " << laserBufferReady << std::endl;
    if(laserBufferReady == false)
```

```
    {
        throw "Error";
    }

    // Get current buffer pointer
    current_ptr = getCurrentLaserBuffPtr();


    // Read data from buffer
    measurements[0] = readLaserBufferFrom(_boxNr0, _size, current_ptr);
    measurements[1] = readLaserBufferFrom(_boxNr1, _size, current_ptr);
    measurements[2] = readLaserBufferFrom(_boxNr2, _size, current_ptr);

    return measurements;
}

/** \brief  This function is a getter for the Laser data buffer for FOUR Boxes, the requested array length
 *          may not be higher than the buffer size
 *
 * \param Number of the box the data is needed from
 * \param  Number of Data points needed
 * \return std::array of std::vector with latest data points
 *
 */
std::array<std::vector<MeasurePoint_Laser>,4> SensorboxServer::getLaserDataFrom(int _boxNr0, int _boxNr1, int _boxNr2,
     int _boxNr3, int _size)
{
    std::array<std::vector<MeasurePoint_Laser>,4> measurements;
    int current_ptr;

    // Check if given size is not bigger than the buffer
    if(_size > (BUFSIZE - MAX_SYNC_ERROR))
    {
        throw "Error!";
    }

    // Check if Buffer full
    //std::cout << "Buff ready " << laserBufferReady << std::endl;
    if(laserBufferReady == false)
    {
        throw "Error";
    }

    // Get current buffer pointer
    current_ptr = getCurrentLaserBuffPtr();

    // Read data from buffer
    measurements[0] = readLaserBufferFrom(_boxNr0, _size, current_ptr);
    measurements[1] = readLaserBufferFrom(_boxNr1, _size, current_ptr);
    measurements[2] = readLaserBufferFrom(_boxNr2, _size, current_ptr);
    measurements[3] = readLaserBufferFrom(_boxNr3, _size, current_ptr);

    return measurements;
}


/** \brief This function is used to read data from the laser buffer
 *
 * \param Number of the box the data is needed
 * \param  Number Measurepoints needed
 * \param Buffer pointer with latest Measurepoint
 * \return
 *
 */
std::vector<MeasurePoint_Laser> SensorboxServer::readLaserBufferFrom(int _boxNr, int _size, int _buf_ptr)
{
    std::vector<MeasurePoint_Laser> measurementVector;
    int temp = _buf_ptr - _size;

    // Ringbuffer overflow must be catched

    // No overflow will occure
    if(temp >= 0)
    {
        // Run buffer reverse
        for(int i = _buf_ptr; i> temp; i--)
        {
            // Add MeasurePoint at the end of buffer
            measurementVector.push_back(syncLaserBuf[boxNumberList[_boxNr]][i]);
        }
    }

    // If overflow will occure
    else if(temp < 0)
    {
```

```
        // Run buffer unter pointer is 0
        for(int i = _buf_ptr; i >= 0; i--)
        {
            // Add MeasurePoint at the end of buffer
            measurementVector.push_back(syncLaserBuf[boxNumberList[_boxNr]][i]);
        }

        // Calculate rest
        int rest = BUFSIZE + (_buf_ptr - _size);

        // run buffer
        for(int i = BUFSIZE-1; i > rest; i--)
        {
            // Add MeasurePoint at the end of buffer
            measurementVector.push_back(syncLaserBuf[boxNumberList[_boxNr]][i]);
        }
    }

    return measurementVector;

}

/** \brief  This function sets a stop flag, that will stop while loop in
 *          ::start()
 *
 *
 */
void SensorboxServer::stop()
{
    stopServer = true;
}


/** \brief  This function will create a timestamp of actual time
 *          resulution in ms
 *
 * \return unsigned long long timestamp
 *
 */
unsigned long long SensorboxServer::createTimestamp()
{
    struct timeval tv;
    gettimeofday(&tv, NULL);

    unsigned long long time =
        (unsigned long long)(tv.tv_sec) * 1000 +
        (unsigned long long)(tv.tv_usec) / 1000;

    return time;
}
```

Header-Datei

```
#ifndef SENSORBOXSERVER_H
#define SENSORBOXSERVER_H

#include <iostream>
#include <thread>
#include <mutex>
#include <string>
#include <array>
#include <vector>
#include <fstream>
#include <sys/time.h>
#include <stdlib.h>


#include "Server.h"
#include "MeasurePoint_Laser.h"
#include "MeasurePoint_Sonar.h"

#define SENSORBOX0_IP "192.168.2.10"
#define SENSORBOX1_IP "192.168.2.11"
#define SENSORBOX2_IP "192.168.2.12"
#define SENSORBOX3_IP "192.168.2.13"

#define LASER_SAMPLE_PERIOD_ms 32
#define SONAR_SAMPLE_PERIOD_ms 120
#define MAX_SYNC_ERROR 150

#define NUMBER_OF_BOXES 4
#define MAX_NUMBER_OF_BOXES 9
#define BUFSIZE 300
```

```
#define BOX0 0
#define BOX1 1
#define BOX2 2
#define BOX3 3
#define BOX4 4
#define BOX5 5
#define BOX6 6
#define BOX7 7
#define BOX8 8
#define BOX9 9

class SensorboxServer
{
public:
    SensorboxServer(int, int, int);
    virtual ~SensorboxServer();
    //int         sendDataToBox(int, std::string);
    void        start();
    void        stop();
    std::string sendCommandTo(std::string, std::string);
    int broadcast(std::string);
    std::vector<MeasurePoint_Laser>    getLaserDataFrom(int, int);
    std::array<std::vector<MeasurePoint_Laser>,2> getLaserDataFrom(int, int, int);
    std::array<std::vector<MeasurePoint_Laser>,3> getLaserDataFrom(int, int, int, int);
    std::array<std::vector<MeasurePoint_Laser>,4> getLaserDataFrom(int, int, int, int, int);

private:

    Server serverLaser;
    Server serverSonar;
    Server serverCMD;
    int fd_pipe_laser, fd_pipe_sonar, fd_max, fd_pipe_cmd;
    int laserBuf_ptr[NUMBER_OF_BOXES], sonarBuf_prt[NUMBER_OF_BOXES];
    bool laserBufferReady, sonarBufferReady;
    int laserSynchErrorCtr, sonarSynchErrorCtr;
    std::array<MeasurePoint_Laser, BUFSIZE> syncLaserBuf[NUMBER_OF_BOXES];
    std::array<MeasurePoint_Sonar, BUFSIZE> syncSonarBuf[NUMBER_OF_BOXES];
    std::array<std::string, NUMBER_OF_BOXES > boxNameList;
    std::array<int        , MAX_NUMBER_OF_BOXES > boxNumberList;

    std::ofstream LaserFiles[4];
    std::ofstream SonarFiles[4];
    std::ofstream LatencyFile;
    bool stopServer;

    static void serverThread(Server&);
    std::string readLine(int);
    int addLaserData(std::string);
    int addSonarData(std::string);
    int identifyBoxes();
    int checkLaserBufferSync(int);
    int checkSonarBufferSync(int);
    int getCurrentSonarBuffPtr();
    int getCurrentLaserBuffPtr();
    std::vector<MeasurePoint_Laser> readLaserBufferFrom(int, int, int);
    unsigned long long createTimestamp();


};

#endif // SENSORBOXSERVER_H
```

MeasuringPoint

CPP-Datei

```
#include <iostream>
#include <sys/time.h>
#include <string>

#include "MeasuringPoint.h"

using namespace std;

MeasuringPoint::MeasuringPoint()
{
    //createTimestamp();
}
```

```
MeasuringPoint::MeasuringPoint(unsigned long long _timestamp)
{
    this->timestamp = _timestamp;
}


unsigned long long MeasuringPoint::createTimestamp()
{
    struct timeval tv;
    gettimeofday(&tv, NULL);

    unsigned long long time =
    (unsigned long long)(tv.tv_sec) * 1000 +
    (unsigned long long)(tv.tv_usec) / 1000;

    return time;
}
unsigned long long MeasuringPoint::getTimestamp()
{
    return timestamp;
}

std::string MeasuringPoint::getString()
{
    return "Baseclass␣default\n";
}

void MeasuringPoint::setTimestamp(unsigned long long _timestamp)
{
    this->timestamp = _timestamp;
}
```

Header-Datei

```
#ifndef MEASURINGPOINT_H
#define MEASURINGPOINT_H

#include <string>

#define BUF_LEN_SONAR 2000

class MeasuringPoint
{
    public:
        MeasuringPoint();
        MeasuringPoint(unsigned long long);
        unsigned long long  getTimestamp();
        void setTimestamp(unsigned long long);

        // interface framework
        virtual std::string getString();

    private:
        unsigned long long  timestamp;
        unsigned long long  createTimestamp();

};

#endif // MEASURINGPOINT_H
```

MeasurePoint_Laser

CPP-Datei

```
#include <iostream>
#include <string>
#include "MeasurePoint_Laser.h"
#include "MeasuringPoint.h"

using namespace std;

MeasurePoint_Laser::MeasurePoint_Laser() : MeasuringPoint()
{
    this-> measurement = -1;
}

MeasurePoint_Laser::MeasurePoint_Laser(unsigned long long _timestamp) : MeasuringPoint(_timestamp)
```

```
{
    this-> measurement = -1;
}

MeasurePoint_Laser::MeasurePoint_Laser(std::string _datastring)
{
    int pos = _datastring.find(";");

    if(pos != std::string::npos)
    {
        try
        {
            // Extract Timestamp
            setTimestamp(stoull(_datastring.substr(0,pos)));
            // Extract Measurement
            setMeasurement(stof(_datastring.substr(pos+1,100)));
        }
        catch(...)
        {
            std::cout << "!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!" << std::endl;
            setTimestamp(0);
            setMeasurement(-1);
        }
    }
    else
    {
        std::cout << "!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!" << std::endl;
        setTimestamp(0);
        setMeasurement(-1);
    }
}


MeasurePoint_Laser::~MeasurePoint_Laser()
{

}

void MeasurePoint_Laser::setMeasurement(float mes)
{
    this->measurement = mes;
}

float MeasurePoint_Laser::getMeasurement()
{
    return measurement;
}

std::string MeasurePoint_Laser::getString()
{
    std::ostringstream outstream;

    outstream << this->getTimestamp() << ";" << this->measurement;
    //std::string output =std::to_string(this->getTimestamp()) + ";" + to_string(measurement);

    return outstream.str();
}
```

Header-Datei

```
#ifndef MEASUREPOINT_LASER_H
#define MEASUREPOINT_LASER_H

#include <sstream>

#include "MeasuringPoint.h"

class MeasurePoint_Laser : public MeasuringPoint
{
public:
    MeasurePoint_Laser();
    MeasurePoint_Laser(unsigned long long);
    MeasurePoint_Laser(std::string);
    virtual ~MeasurePoint_Laser();
    void    setMeasurement(float);
    float   getMeasurement();

    std::string  getString();


private:
    float measurement;
};
```

```
#endif // MEASUREPOINT_LASER_H
```

MeasurePoint_Sonar

CPP-Datei

```
#include <iostream>
#include <string>

#include "MeasurePoint_Sonar.h"
#include "MeasuringPoint.h"

MeasurePoint_Sonar::MeasurePoint_Sonar() : MeasuringPoint()
{

}

MeasurePoint_Sonar::MeasurePoint_Sonar(unsigned long long _timestamp) : MeasuringPoint(_timestamp)
{

}

MeasurePoint_Sonar::MeasurePoint_Sonar(std::string _datastring) : MeasuringPoint()
{
    int pos = _datastring.find(";");
    int i;

    if(pos != std::string::npos)
    {
        //std::cout << _datastring << std::endl;
        try
        {
        setTimestamp(stoull(_datastring.substr(0,pos)));
        }
        catch(...)
        {
                setTimestamp(0);
        }
        _datastring.erase(0,pos+1);
        for(i=0; i<SONAR_NUMBER_OF_MEASUREMENTS-1; i++)
        {
            pos = _datastring.find(";");
            if(pos != std::string::npos)
            {
                try
                {
                    measurement_array[i] = stoi(_datastring.substr(0,pos));
                    _datastring.erase(0,pos+1);
                }
                catch(...)
                {
                    std::cout << "Error: MeasurePoint_Sonar::MeasurePoint_Sonar(std::string _datastring): " <<
                          _datastring << std::endl;
                    measurement_array[i] = -1;
                }
            }
            else
            {
                measurement_array[i] = -1;
            }
        }
        // The last value has no ";" at the end
        try
        {
            measurement_array[i] = stoi(_datastring);
        }
        catch(...)
        {
            measurement_array[i] = -1;
        }
    }
}

MeasurePoint_Sonar::~MeasurePoint_Sonar()
{
    //dtor
}
```

```
void MeasurePoint_Sonar::setMeasurement(std::array<int, SONAR_NUMBER_OF_MEASUREMENTS> _mes)
{
    //for(int i = 0; i < SONAR_NUMBER_OF_MEASUREMENTS; i++)
    //    this->measurement_array[i] = mes[i];
    this->measurement_array = _mes;
}

std::array<int, SONAR_NUMBER_OF_MEASUREMENTS> MeasurePoint_Sonar::getMeasurement()
{
    return measurement_array;
}

std::string MeasurePoint_Sonar::getString()
{
    std::string output = std::to_string(this->getTimestamp()) ;

    for(int i = 0; i<SONAR_NUMBER_OF_MEASUREMENTS; i++)
    {
        output = output +  ";" + std::to_string(measurement_array[i]);
    }

    return output;
}
```

Header-Datei

```
#ifndef MEASUREPOINT_SONAR_H
#define MEASUREPOINT_SONAR_H

#include <string>
#include <array>

//#include "XL_MaxSonar.h"
#include "MeasuringPoint.h"

#define SONAR_NUMBER_OF_MEASUREMENTS 450

class MeasurePoint_Sonar : public MeasuringPoint
{
public:
    MeasurePoint_Sonar();
    MeasurePoint_Sonar(unsigned long long);
    MeasurePoint_Sonar(std::string);
    virtual ~MeasurePoint_Sonar();
    void    setMeasurement(std::array<int, SONAR_NUMBER_OF_MEASUREMENTS>);
    std::array<int, SONAR_NUMBER_OF_MEASUREMENTS> getMeasurement();

    std::string getString();

private:
    //int     measurement[SONAR_NUMBER_OF_MEASUREMENTS];
    std::array<int, SONAR_NUMBER_OF_MEASUREMENTS> measurement_array;

};

#endif // MEASUREPOINT_SONAR_H
```

B.2.2. Sensoreinheit

SF02_F_Rangefinder

CPP-Datei

```
// SF02_F_Rangefinder.cpp
// Project: SensorBox LIDAR Class
// Brief:   This Class defines methods to for the SF02_F_Rangefinder
//          using the UART interface
//
//          This program is part of the masterthesis "Konzeption
//          und Entwicklung eines hochpraezisen Systems zur
//          Positionsbestimmung von Fahrzeugen in urbanen Gebieten"
// Author:  Mario Wegner
// Date:    10/2016
```

```
#include "SF02_F_Rangefinder.h"

#include <iostream>
#include <string>
#include <time.h>



/** \brief  Constructor: Opens UART interface and sets
 *          Object ready
 *
 */
SF02_F_Rangefinder::SF02_F_Rangefinder()
{
    //std::cout << "Konstructor" << std::endl;
    // Clear measurement
    measurement_string = "";
    measurement_float = -1;
    fd = 0;

    // Calculations for UART Timeout
    int timeout_ms = 30;
    timeout_ticks = timeout_ms * CLOCKS_PER_SEC * 0.001;

    // Set Timeout for select() to 31 ms
    timeout.tv_sec = 0;
    timeout.tv_usec = 30000;



    try
    {
        // Open interface
        //fd = open("/dev/serial0", O_RDWR | O_NOCTTY | O_NDELAY);
        fd = open("/dev/ttyS0", O_RDWR | O_NOCTTY | O_NDELAY);

        if(fd>= 0)
        {
            // Config UART Interface
            cfsetispeed(&options, B115200);     // Bautrate Rx
            cfsetospeed(&options, B115200);     // Bautrate Tx

            options.c_cflag &= ~PARENB;         // no Paritaetsbit
            options.c_cflag &= ~CSTOPB;         // one Stopbit
            options.c_cflag &= ~CSIZE;
            options.c_cflag |= CS8;             // 8 Datenbits
            options.c_cflag |= (CLOCAL | CREAD);
            options.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG);
            options.c_iflag = IGNPAR;
            options.c_oflag = ~OPOST;
            options.c_cc[VMIN] = 1;           //
            options.c_cc[VTIME] = 25;          // Timeout in x*0.1s

            tcflush(fd,TCIOFLUSH);              // clear buffer

            if(tcsetattr(fd,TCSAFLUSH,&options) != 0)
                throw "SF02_F_Rangefinder::Constructor:␣Error:␣Start␣serieal␣interface";
        }
        else
        {
            throw "SF02_F_Rangefinder::Constructor:␣Error:␣Device␣not␣found";
        }

        // Set device ready
        this->ready = true;
    }
    catch(std::string s)
    {
        std::cout << s << std::endl;
    }

    // Prepare for select()
    FD_ZERO(&laser_fdset);
    FD_SET(fd, &laser_fdset);

    return;
}


/** \brief  This function performs a measurement by sending the symbol 'D' to the Rangefinder.
 *          The Rangefinder will answer with his actual measurement with following format:
 *
 *          "12,34\r\n"
 *
```

```
 *          The function will return the received data as std::sting
 *
 * \return std::string measurement
 *
 */
std::string SF02_F_Rangefinder::takeMeasurement_string()
{
    // Set device not ready
    //this->ready = false;
    try
    {
        usleep(1000);
        // clear variable
        measurement_string.clear();
        char c = 'x';
        int anz;

        // clear buffer
        tcflush(fd,TCIOFLUSH);

        // safe timeout starttime = time since OS start
        clock_t starttime = clock();

        // Trigger measurement by sending "D" (see Datasheet)
        write(fd,"D",1);

        // Wait for the Rangefinder
        //usleep(20000);

        // Receive Measurement "XX.XX\r\n"
        //do
        timeout.tv_sec = 0;
        timeout.tv_usec = 30000;
        select(fd+1, &laser_fdset,NULL,NULL,&timeout);//&timeout);

        if(FD_ISSET(fd,&laser_fdset))
        {
            while(true)
            {
                anz = read(fd, &c,1);
                //std::cout << "anz: " << anz << " " << hex << (int)c << dec << std::endl;
                if(anz == 1)
                {
                    if(c == '\r' || c == '\n')
                        break;
                    else
                    {
                        //std::cout << hex << (int)c << std::endl;
                        measurement_string = measurement_string + c;
                        //std::cout << measurement_string << std::endl;
                    }
                }

                // Timeout counter
                //    if(starttime+timeout_ticks < clock())
                //    {
                //        std::cout << "SF02_F_Rangefinder::takeMeasurement: Error TIMEOUT: " << starttime+timeout_ticks
                      - clock() << std::endl;
                //        throw "SF02_F_Rangefinder::takeMeasurement: Error: UART Timeout";
                //    }


                //usleep(100);

            }
        }
        else
        {
            std::cout << "SF02_F_Rangefinder::takeMeasurement:␣Error␣TIMEOUT:␣" << clock() - starttime << std::endl;
            throw "SF02_F_Rangefinder::takeMeasurement:␣Error:␣UART␣Timeout";
        }

        // Read until the char '\n is send
        //while(c != 0xd);

        // Set device ready
        //this->ready = true;

        return measurement_string;
    }

    // Catch Error and print them via cout
    catch(std::string& s)
    {
        std::cout << s << std::endl;
```

```
        return "no value";
    }
    catch(...)
    {
        std::cout << "boing" << std::endl;
        return "no value";

    }

}

/** \brief  This function calls the function takeMeasurement_string and
 *          converts the String to a float
 *
 * \return float measurement
 *
 */
float   SF02_F_Rangefinder::takeMeasurement_float()
{
    std::string temp_string;
    try
    {
        measurement_float = 0;
        temp_string = this->takeMeasurement_string();
        //std::cout << "'" << temp_string << "'" << std::endl;
        measurement_float = std::stof(temp_string);

        //this->ready = true;
        return measurement_float;
    }
    catch(...)
    {
        cout << "makeMeasurement_float: Something went wrong! '"<< temp_string <<"'" << endl;
        //this->ready = true;
        return -1;
    }
}
/** \brief This function is a getter for the ready-state
 *
 * \return bool ready
 *
 */
bool SF02_F_Rangefinder::isReady()
{
    return ready;
}

void SF02_F_Rangefinder::setBusy()
{
    this->ready = false;
}

void SF02_F_Rangefinder::setIdle()
{
    this->ready = true;
}
```

Header-Datei

```
// SF02_F_Rangefinder.h
//
// Header File for SF02/F Laser Rangefinder
// Contains Class "Laser"
//
//
// Date: 27.10.2016
// By Mario Wegner

#include <iostream>
#include <string>

#include <time.h>
#include <termios.h>            //Used for UART
#include <unistd.h>                     //Used for UART
#include <fcntl.h>                      //Used for UART

using namespace std;

class SF02_F_Rangefinder
{
public:
    SF02_F_Rangefinder();
    float       takeMeasurement_float();
    string      takeMeasurement_string();
```

```
    bool    isReady();
    void    setBusy();
    void    setIdle();

private:
    int             fd;
    struct termios  options;
    long            timeout_ticks;
    bool            ready;
    fd_set          laser_fdset;
    struct timeval  timeout;

    string measurement_string;
    float measurement_float;

};
```

XL_MaxSonar

CPP-Datei

```
// XL_MaxSonar.cpp
// Project: SensorBox Sonar Class
// Brief:   This Class defines methods to for the XL_MaxSonar Ultrasonic sensor
//          using an instance of ADS1256_ISR (24 Bit A/D Converter)
//
//          This program is part of the masterthesis "Konzeption
//          und Entwicklung eines hochpraezisen Systems zur
//          Positionsbestimmung von Fahrzeugen in urbanen Gebieten"
// Author:  Mario Wegner
// Date:    10/2016

#include <iostream>
#include <unistd.h> // for sleep

#include <wiringPi.h>
#include <bcm2835.h>

#include "XL_MaxSonar.h"

/** \brief  Constructor for Class XL_MaxSonar, an instance of ADS1256_ISR
 *          will be created. Also the pin to Trigger a measurement will be
 *          defined. At last the device will be set ready.
 *
 * \param   int trigger_pin
 * \return
 *
 */
XL_MaxSonar::XL_MaxSonar(int _pin) : adc(ADS1256_7500SPS)
{
    this->triggerOutputPin = _pin;
    // Define pin as output
    bcm2835_gpio_fsel(triggerOutputPin, BCM2835_GPIO_FSEL_OUTP);

    this->ready = true;
}
/** \brief  This function performs an sonar measurement by setting the trigger-pin
 *          high for 100 us then read analog values for 60 ms and returns pointer to array
 *
 * \return std::array<int,SONAR_NUMBER_OF_MEASUREMENTS> measurements
 *
 */
std::array<int,SONAR_NUMBER_OF_MEASUREMENTS> XL_MaxSonar::takeMeasurement()
{
    // set device not ready
    //ready = false;

    // Set triggerPin 1
    bcm2835_gpio_write(triggerOutputPin,HIGH);
    // Wait 1 msgpio rea
    bcm2835_delayMicroseconds(100);
    // Set triggerPin 0
    bcm2835_gpio_write(triggerOutputPin,LOW);

    // Wait for Trigger delay
    bcm2835_delayMicroseconds(20000);

    try
    {
```

```
        measurement = adc.take_N_Samples(SONAR_NUMBER_OF_MEASUREMENTS);
    }
    catch(std::string s) // Error handling
    {
        std::cout << s << std::endl;

        // clear array if error
        for(int i = 0; i < SONAR_NUMBER_OF_MEASUREMENTS; i++)
        {
            this->measurement[i] = 0;
        }
        ready = true;
        return measurement;
    }

    //ready = true;

    return measurement;

}

/** \brief This function is a getter for the ready flag
 *
 * \return bool ready
 *
 */

/** \brief This function is a getter for the ready-state
 *
 * \return bool ready
 *
 */
bool XL_MaxSonar::isReady()
{
    return ready;
}

void XL_MaxSonar::setBusy()
{
    this->ready = false;
}

void XL_MaxSonar::setIdle()
{
    this->ready = true;
}
```

Header-Datei

```
#ifndef XL_MAXSONAR_H
#define XL_MAXSONAR_H

#include "ADS1256_ISR.h"
#include <array>


// 100 ms Sample Time: NUMBER_OF_MEASUREMENTS = ADC_SAMPLETIME * 0,06 s
#define SONAR_NUMBER_OF_MEASUREMENTS 450

class XL_MaxSonar
{
public:
    XL_MaxSonar(int);
    std::array<int,SONAR_NUMBER_OF_MEASUREMENTS> takeMeasurement();
    bool isReady();
    void setBusy();
    void setIdle();

private:
    std::array<int,SONAR_NUMBER_OF_MEASUREMENTS> measurement;
    int triggerOutputPin;
    ADS1256_ISR adc;
    bool ready;

};

#endif // XL_MAXSONAR_H
```

ADS1256_ISR

CPP-Datei

```
#include "ADS1256_ISR.h"

// Initialise static variables - Needed to be static because of wiringPiISR
uint32_t    ADS1256_ISR::samples_array[MAX_ARRAY_LENGTH];
uint8_t     ADS1256_ISR::drate          = 0;
bool        ADS1256_ISR::isr_idle       = true;
bool        ADS1256_ISR::takeMeasurementsFlag = false;
uint32_t    ADS1256_ISR::measurementIndex = 0;
uint32_t    ADS1256_ISR::numberOfMeasurements = 0;
bool        ADS1256_ISR::measurementIsDone = false;


/** \brief  Constructor, opens the SPI interface using bcm2835 library, resets
 *          and initialises the ADC
 * \param   int drate, samplerate of ADC
 *
 */
ADS1256_ISR::ADS1256_ISR(int _drate)
{
    drate = _drate;

    // Initialise SPI interface
    if (!bcm2835_init())
        throw "ADS1256::Constructor:␣Error:␣SPI␣Init";
    bcm2835_spi_begin();
    bcm2835_spi_setBitOrder(BCM2835_SPI_BIT_ORDER_LSBFIRST );
    bcm2835_spi_setDataMode(BCM2835_SPI_MODE1);
    bcm2835_spi_setClockDivider(BCM2835_SPI_CLOCK_DIVIDER_128);

    bcm2835_gpio_fsel(CS_PIN, BCM2835_GPIO_FSEL_OUTP);//
    //bcm2835_gpio_fsel(BCM2835_SPI_CS0, BCM2835_GPIO_FSEL_OUTP);//
    bcm2835_spi_chipSelect(BCM2835_SPI_CS_NONE);

    bcm2835_gpio_write(CS_PIN, HIGH);
    bcm2835_gpio_fsel(DRDY_PIN, BCM2835_GPIO_FSEL_INPT);
    bcm2835_gpio_set_pud(DRDY_PIN, BCM2835_GPIO_PUD_UP);

    bcm2835_delayMicroseconds(50);

    send8Bit(CMD_RESET);

    bcm2835_delayMicroseconds(10);

    initialiseADC(0, drate, 0);

    wiringPiISR (DRDY_PIN, INT_EDGE_FALLING, &(ADS1256_ISR::spi_data_exchange_ISR));
}

ADS1256_ISR::~ADS1256_ISR()
{
    //dtor
}

void ADS1256_ISR::send8Bit(uint8_t _data)
{
    CS_0();
    bcm2835_spi_transfer(_data);
    CS_1();
}

uint8_t ADS1256_ISR::receive8Bit()
{

    uint8_t read = 0;
    read = bcm2835_spi_transfer(0xff);
    return read;
}

uint32_t ADS1256_ISR::receiveSample()
{
    uint8_t receive_buff[3];
    uint32_t sample;

    CS_0();
    receive_buff[0] = receive8Bit();
    receive_buff[1] = receive8Bit();
    receive_buff[2] = receive8Bit();
    CS_1();
```

```
    sample = ((uint32_t)receive_buff[0] << 16) & 0x00FF0000;
    sample |= ((uint32_t)receive_buff[1] << 8);
    sample |= receive_buff[2];


    return sample;
}

void ADS1256_ISR::writeReg(uint8_t _RegID, uint8_t _RegValue)
{
    CS_0();
    bcm2835_spi_transfer(CMD_WREG | _RegID);    /*Write command register */
    bcm2835_spi_transfer(0x00);         /*Write amount of Bytes (N -1) */
    bcm2835_spi_transfer(_RegValue);    /*send register value */
    CS_1();
}

void ADS1256_ISR::initialiseADC(uint8_t _gain, uint8_t _drate, uint8_t _ch)
{
    uint8_t statusREG_conf  = (0 << 3) | (1 << 2) | (0 << 1);

    uint8_t muxREG_conf     = 0x08;

    uint8_t adconREG_conf   = (0 << 5) | (0 << 3) | (_gain << 0);

    uint8_t drateREG_conf   = _drate;

    //WaitDRDY_LOW();
    writeReg(REG_Status,statusREG_conf);
    WaitDRDY_LOW();
    writeReg(REG_MUX,muxREG_conf);
    WaitDRDY_LOW();
    writeReg(REG_ADCON,adconREG_conf);
    WaitDRDY_LOW();
    writeReg(REG_DRATE,drateREG_conf);
}

void ADS1256_ISR::WaitDRDY_LOW()
{
    uint32_t i;

    for (i = 0; i < 500000; i++)
    {
        usleep(10);
        if (DRDY_IS_LOW())
        {
            break;
        }
    }
    if (i >= 500000)
    {
        std::cout << "ADS1256_WaitDRDY_LOW()␣Time␣Out␣..." << std::endl;
    }
}

std::array<int,SONAR_NUMBER_OF_MEASUREMENTS> ADS1256_ISR::take_N_Samples(int _n)
{
    WaitDRDY_LOW();
    send8Bit(CMD_RDATAC);

    //uint32_t *samples_ptr = new uint32_t[_n];
    measurementIndex = 0;
    numberOfMeasurements = _n;

    // activate ISR measurement
    takeMeasurementsFlag = true;
    measurementIsDone = false;
    //cout << "Start mes" << endl;

    // Wait for measurement
    bcm2835_delayMicroseconds(100000);

    while(true)
    {
        if(measurementIsDone == true)
            break;
        usleep(100);
    }

    //deactivate measurement
    takeMeasurementsFlag == false;
    send8Bit(CMD_SDATAC);

    // copy to output
    for(int i = 0; i < numberOfMeasurements; i++)
```

```
    {
        measurement[i] = samples_array[i];
    }

    return measurement;

}

void ADS1256_ISR::spi_data_exchange_ISR()
{
    //cout << "ADC Trigger!" << endl;
    if(isr_idle == true && takeMeasurementsFlag == true)
    {
        isr_idle = false;

        if(measurementIndex < numberOfMeasurements)
        {
            samples_array[measurementIndex] = receiveSample();
            measurementIndex++;
        }
        else
        {
            measurementIsDone = true;
        }

        isr_idle = true;
    }
}
```

Header-Datei

```
#ifndef ADS1256_ISR_H
#define ADS1256_ISR_H

#include <unistd.h>
#include <iostream>
#include <array>

#include <wiringPi.h>
#include <bcm2835.h>

#define MAX_ARRAY_LENGTH 2000

#define DRDY_PIN    RPI_GPIO_P1_11
#define CS_PIN      RPI_GPIO_P1_15

#define DRDY_PIN_WIRINGPI 0

#define CHANNEL_0   0
#define CHANNEL_1   1

#define CS_1() bcm2835_gpio_write(CS_PIN,HIGH)
#define CS_0()  bcm2835_gpio_write(CS_PIN,LOW)

#define DRDY_IS_LOW()   ((bcm2835_gpio_lev(DRDY_PIN)== 0))
#define DRDY_IS_HIGH()  ((bcm2835_gpio_lev(DRDY_PIN)== 1))

#define SPI_SPEED_1953kHz 1953000

#define CMD_WREG    0x50
#define CMD_SYNC    0xFC
#define CMD_WAKEUP  0x00
#define CMD_RDATA   0x01
#define CMD_RDATAC  0x03
#define CMD_SDATAC  0x0F
#define CMD_STANDBY 0xFD
#define CMD_RESET   0xFE

#define REG_Status  0x00
#define REG_MUX     0x01
#define REG_ADCON   0x02
#define REG_DRATE   0x03
#define REG_IO      0x04


#define ADS1256_GAIN_1 0

#define ADS1256_30000SPS    0xf0
#define ADS1256_15000SPS    0xe0
#define ADS1256_7500SPS     0xD0
#define ADS1256_3750SPS     0xC0
#define ADS1256_100SPS      0x82
#define ADS1256_10SPS       0x23
```

```
#define SONAR_NUMBER_OF_MEASUREMENTS 450



class ADS1256_ISR
{

public:
    ADS1256_ISR(int);
    virtual ~ADS1256_ISR();
    std::array<int, SONAR_NUMBER_OF_MEASUREMENTS> take_N_Samples(int);

private:
    std::array<int, SONAR_NUMBER_OF_MEASUREMENTS> measurement;
    static uint8_t  drate;
    static bool     isr_idle;
    static bool     takeMeasurementsFlag;
    static bool     measurementIsDone;
    static uint32_t measurementIndex;
    static uint32_t samples_array[MAX_ARRAY_LENGTH];
    static uint32_t numberOfMeasurements;

    static void     send8Bit(uint8_t);
    static uint8_t  receive8Bit();
    static uint32_t receiveSample();
    static void     spi_data_exchange_ISR();
    static void     writeReg(uint8_t, uint8_t);
    static void     initialiseADC(uint8_t, uint8_t, uint8_t);
    static void     WaitDRDY_LOW();

};

#endif // ADS1256_ISR_H
```

Client

CPP-Datei

```
// Project: Client.cpp
// Brief:   Short and simple client class using sockets
//          Source: "C und Linux - Die Moeglichkeiten des Betriebssystems
//          mit eigenen Programmen nutzen"
//          This program is part of the masterthesis "Konzeption
//          und Entwicklung eines hochpräzisen Systems zur
//          Positionsbestimmung von Fahrzeugen in urbanen Gebieten"
// Author:  Mario Wegner
// Date:    10/2016

#include "Client.h"


/** \brief Constructor initializes with "not connected"
 *
 */
Client::Client()
{
    connected = false;
}

Client::~Client()
{
    close(sock_fd);
}

/** \brief  connectTo: Try to connect to server. If successful, fd_socket
 *          will be set. Connected flag will be set true
 *
 * \param std::string _serverIP IP address of the Server
 * \param int _port Port of the server
 * \return  0 if connected
 *          -1 if not connected
 */
int Client::connectTo(std::string _serverIP, int _port)
{
    // Default vlaues
    connected = false;
    sock_fd = -1;
```

```
    // Add IP and Port to struckt (preparation for socket
    server_addr.sin_family = PF_INET;
    server_addr.sin_port = htons(_port);        // add port
    inet_aton(_serverIP.c_str(), &(server_addr.sin_addr)); // add IP

    // Create Socket
    sock_fd = socket(AF_INET, SOCK_STREAM,0);


    int err = -1;

    // Connect to server
    err = connect(sock_fd, (struct sockaddr *)&server_addr, sizeof(struct sockaddr_in));
    if(err == 0)
    {
        // If connection successful -> connected flag = true, return 0
        connected = true;
        return 0;
    }

    // Return -1 if not connected
    return -1;
}

/** \brief  sendData can be used to send a data string to the server. a linebreak will
 *          be appended to string
 *
 * \param   std::string _data Data string with data to send
 * \return  int length: number of bytes send
 *          -1 if data could not be send
 */

int Client::sendData(std::string _dataString)
{
    // Check Client is connected to server
    if(connected == false)
        // If not return -1
        return -1;

    int length;

    // append linebreak
    _dataString.append("\r\n");

    // Convert std::string to c_string
    char * sendString = new char [_dataString.length()+1];
    std::strcpy (sendString, _dataString.c_str());

    // send c_string
    length = send(sock_fd, sendString, _dataString.length(),MSG_DONTWAIT); // Non waiting if TCP Buffer full
    free(sendString);

    if(length == -1)
        std::cout << "Client::sendData:␣sending␣failed!" << std::endl;

    // return number of bytes send
    return length;
}

/** \brief  receiveData will block until Data arrives at socket from server,
 *          if connection closed by server client will clos connection to
 *
 * \return  std::string data
 *          "ConnectionClosed" if connection closed
  *         " "error: Not connected" if not connected to server
 *
 */

std::string Client::receiveData()
{
    // check if connected to Server
    if(connected == false)
        return "error:␣Not␣connected\r\n";

    // Init variables
    char buffer[16383];
    int length;

    // Create fd_set
    fd_set input_fdset;     // Init
    FD_ZERO(&input_fdset);  // clear fd_set
    FD_SET(sock_fd, &input_fdset);  // add socket_fd to fd_set

    // block until data arrives at socket_fd, no timeout
    select(sock_fd+1, &input_fdset,NULL,NULL,NULL);
```

```
    // read data from socket_fd
    length = recv(sock_fd, buffer, 16383, 0);

    //std::cout << "Buffer: " << buffer << std::endl;
    // If no data received -> Connection closed
    if(length == 0)
    {
        // Close connection
        std::cout << "Client::receiveData:␣Connection␣closed␣by␣server" << std::endl;
        close(sock_fd);
        connected = false;
        return "ConnectionClosed";

    }

    // Convert c_string to std::string
    std::string data(buffer);

        //std::cout << "Data: " << buffer << " " << length << std::endl;
    // return data
    return data;
}

/** \brief isConnected is a getter for the connected flag
 *
 * \return  true if client is connected to server
 *          false if client is not connected to server
 *
 */

bool Client::isConnected()
{
    return connected;
}
```

Header-Datei

```
#ifndef CLIENT_H
#define CLIENT_H

#include <iostream>
#include <string>
#include <cstring>
#include <unistd.h>

#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>


#define NUMBER_OF_BUFFERSTRINGS     100

class Client
{
    public:
        Client();
        virtual         ~Client();
        int             connectTo(std::string, int);
        int             sendData(std::string);
        std::string     receiveData();
        bool            isConnected();

    private:
        struct  sockaddr_in server_addr;
        int     sock_fd;
        bool    connected;

};

#endif // CLIENT_H
```

ConfigData

CPP-Datei

```
// main.cpp
// Project: SensorBox ConfigData.cpp
// Brief:   This class holds all the configData
//          needed to run the sensorbox.
//          The data is saved in a config file
//          This program is part of the masterthesis "Konzeption
//          und Entwicklung eines hochpraezisen Systems zur
//          Positionsbestimmung von Fahrzeugen in urbanen Gebieten"
// Author:  Mario Wegner
// Date:    10/2016

#include "ConfigData.h"

/** \brief Constructor sets every data to default
 *
 */

ConfigData::ConfigData()
{
    // Ser every variable to default
    boxName = "";
    serverIP = "";
    laserPort = -1;
    sonarPort = -1;
    cmdPort = -1;
    doMeasurement = false;
    stopProgramFlag = false;
    sonarMeasurements = true;
    laserMeasurements = true;
    camMeasurements = true;

}

ConfigData::~ConfigData()
{
    //dtor
}

/** \brief  readConficFile reads the config.txt and initializes
 *              - Boxname
 *              - Server IP
 *              - Laser Port
 *              - Sonar Port
 *              - CMD Port
 * \param std::string with path to "config.txt" ("./" if confic.txt in program folder
 * \return  0 if config file read
 *          -1 if an error occured
 */

int ConfigData::readConfigFile(std::string _path)
{
    // Define varables
    std::string filename = "config.txt";
    std::string line;
    int pos;

    // Lines to find
    std::string boxNumber   = "BoxName:";
    std::string serverIP    = "Server␣IP-Address:";
    std::string laserPort   = "Laser␣Port␣Nr.:";
    std::string sonarPort   = "Sonar␣Port␣Nr.:";
    std::string cmdPort     = "CMD␣Port␣Nr.:";
    std::string sonarMes    = "Sonar␣Measurements:";
    std::string camMes      = "Camera␣Measurements:";

    // Open config File
    std::ifstream file_config(_path + filename);
    if(!file_config)
    {
        std::cout << "Error:␣Main::readConfigFile:␣Could␣not␣open␣config␣file" << std::endl;
        return -1;
    }

    // Read Config File line by line
    while(std::getline(file_config, line))
    {
        pos = line.find(boxNumber);
        if(pos!=std::string::npos)
        {
            std::string data = line.substr(pos+boxNumber.length(),1000);
            this->boxName = data;
        }
        pos = line.find(serverIP);
        if(pos!=std::string::npos)
        {
```

```
            std::string data = line.substr(pos+serverIP.length(),1000);
            this->serverIP = data;
        }
        pos = line.find(laserPort);
        if(pos!=std::string::npos)
        {
            std::string data = line.substr(pos+laserPort.length(),1000);
            this->laserPort = std::stoi(data);
        }
        pos = line.find(sonarPort);
        if(pos!=std::string::npos)
        {
            std::string data = line.substr(pos+sonarPort.length(),1000);
            this->sonarPort = std::stoi(data);
        }
        pos = line.find(cmdPort);
        if(pos!=std::string::npos)
        {
            std::string data = line.substr(pos+cmdPort.length(),1000);
            this->cmdPort = std::stoi(data);
        }
        pos = line.find(sonarMes);
        if(pos!=std::string::npos)
        {

            std::string data = line.substr(pos+sonarMes.length(),1000);
            std::cout << data << std::endl;
            if(data == "no")
            {
                this->sonarMeasurements = false;
            }
        }
        pos = line.find(camMes);
        if(pos!=std::string::npos)
        {

            std::string data = line.substr(pos+camMes.length(),1000);
            std::cout << data << std::endl;
            if(data == "no")
            {
                this->camMeasurements = false;
            }
        }
    }

    // Check if BoxName is set
    if(this->boxName== "")
    {
        std::cout << "Error:␣Main::readConfigFile:␣No␣Boxname␣found" << std::endl;
        return -1;
    }
    return 0;
}

/** \brief gertter for BoxName
 *
 */

std::string ConfigData::getBoxName()
{
    return boxName;
}

std::string ConfigData::getBoxIdentifier()
{
    return "#" + boxName + ":";
}

/** \brief getter for Server IP
 *
 */
std::string ConfigData::getServerIP()
{
    return serverIP;
}

/** \brief getter for Laser Port
 *
 */

int ConfigData::getLaserPort()
{
    return laserPort;
}
```

```
/** \brief getter for Sonar Port
 *
 */

int ConfigData::getSonarPort()
{
    return sonarPort;
}

/** \brief getter for CMD Port
 *
 */
int ConfigData::getCmdPort()
{
    return cmdPort;
}


/** \brief set doMeasurement flag to true
 *
 */
void ConfigData::startMeasurement()
{
    doMeasurement = true;
}
/** \brief set doMeasurement flag to false
 */
void ConfigData::stopMeasurement()
{
    doMeasurement = false;
}

/** \brief getter for doMeasurement flag
 *
 */
bool ConfigData::isMeasurementStarted()
{
    return doMeasurement;
}

/** \brief set stopProgramFlag to true
 *
 */
void ConfigData::stopProgram()
{
    stopProgramFlag = true;
}
/** \brief getter for stopProgramFlag
 *
 */
bool ConfigData::programToStop()
{
    return stopProgramFlag;
}

bool ConfigData::sonarMeasurement()
{
    return sonarMeasurements;
}

void ConfigData::startSonar()
{
    sonarMeasurements = true;
}

void ConfigData::stopSonar()
{
    sonarMeasurements = false;
}

bool ConfigData::laserMeasurement()
{
    return laserMeasurements;
}

void ConfigData::startLaser()
{
    laserMeasurements = true;
}

void ConfigData::stopLaser()
{
    laserMeasurements = false;
}
```

```
bool ConfigData::camMeasurement()
{
    return camMeasurements;
}

void ConfigData::startCam()
{
    camMeasurements = true;
}

void ConfigData::stopCam()
{
    camMeasurements = false;
}
```

Header-Datei

```
#ifndef CONFIGDATA_H
#define CONFIGDATA_H

#include <string>
#include <iostream>
#include <fstream>

class ConfigData
{
public:
    ConfigData();
    virtual ~ConfigData();
    int readConfigFile(std::string);
    std::string getBoxName();
    std::string getBoxIdentifier();
    std::string getServerIP();
    int         getLaserPort();
    int         getSonarPort();
    int         getCmdPort();
    void startMeasurement();
    void stopMeasurement();
    bool isMeasurementStarted();
    void stopProgram();
    bool programToStop();
    bool sonarMeasurement();
    void startSonar();
    void stopSonar();
    bool camMeasurement();
    void startCam();
    void stopCam();
    bool laserMeasurement();
    void startLaser();
    void stopLaser();


private:
    std::string boxName;
    std::string serverIP;
    int         laserPort;
    int         sonarPort;
    int         cmdPort;
    bool        doMeasurement;
    bool        stopProgramFlag;
    bool        sonarMeasurements;
    bool        laserMeasurements;
    bool        camMeasurements;

};

#endif // CONFIGDATA_H
```

WiringPiInit

CPP-Datei

```
#include "WiringPiInit.h"

#include <wiringPi.h>
#include <bcm2835.h>
```

```
WiringPiInit::WiringPiInit()
{
    wiringPiSetupGpio();
    bcm2835_init();
}

WiringPiInit::~WiringPiInit()
{
    //dtor
}
```

Header-Datei

```
#ifndef WIRINGPIINIT_H
#define WIRINGPIINIT_H

class WiringPiInit
{
    public:
        WiringPiInit();
        virtual ~WiringPiInit();

    protected:

    private:
};

#endif // WIRINGPIINIT_H
```

B.2.3. Nachträgliche Datenauswertung

Geschwindigkeit

```
%% Analysis and implentation of Speed-Algorithm
% AuswertungGeschwindigkeit.m
%
% Author: Mario Wegner
% Date: 1/2017
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

lidar_distance = 0.51; % m
correlation_window = 8000; % ms

%% Lidar Error handling Linearse measurements under 15 cm
r_lidar0_filtered = r_lidar0;
r_lidar1_filtered = r_lidar1;
r_lidar2_filtered = r_lidar2;
r_lidar3_filtered = r_lidar3;

i = 2;
for i = 2 : 1 :size(r_lidar3,1)
     if r_lidar3_filtered(i) < 0.15
          r_lidar3_filtered(i) = 0.5*(r_lidar3_filtered(i-1)+ r_lidar3_filtered(i+1));
    end
end

i = 2;
for i = 2 : 1 :size(r_lidar2,1)
     if r_lidar2_filtered(i) < 0.15
          r_lidar2_filtered(i) = 0.5*(r_lidar2_filtered(i-1)+ r_lidar2_filtered(i+1));
    end
end

i = 2;
for i = 2 : 1 :size(r_lidar1,1)
     if r_lidar1_filtered(i) < 0.15
          r_lidar1_filtered(i) = 0.5*(r_lidar1_filtered(i-1)+ r_lidar1_filtered(i+1));
    end
end
i = 2;
for i = 2 : 1 :size(r_lidar0,1)
     if r_lidar0_filtered(i) < 0.15
          r_lidar0_filtered(i) = 0.5*(r_lidar0_filtered(i-1)+ r_lidar0_filtered(i+1));
    end
```

```
end

%% Interploate to 1ms resolution
% Create time vector
t_start = 1490360200000;
t_stop  = t_gps(end);
t = (t_start : t_stop)';

% Interpolation Speed to 1 ms
can_speed_ipol = interp1(t_can,can_speed,t);
GPS_speed_ipol = interp1(t_gps,GPS_speed,t);

% Cancel time differences and interpolate
r_lidar0_ipol = interp1(t_lidar0,r_lidar0_filtered,t-25000);
r_lidar1_ipol = interp1(t_lidar1,r_lidar1_filtered,t-25000);
r_lidar2_ipol = interp1(t_lidar2,r_lidar2_filtered,t-25000);
r_lidar3_ipol = interp1(t_lidar3,r_lidar3_filtered,t-25000);
%%

% Calculate speed from LIDAR
corr_time_ms_left = [];
corr_time_ms_right = [];
corr_speed_right = [];
corr_speed_left = [];
t_corr_r = [];
t_corr_l = [];

offset = 200000;
%% Start Loop left Side
for i = 1+offset: 100 : size(t,1)- offset
    % Extract measurements
    r_temp0 = r_lidar0_ipol(i-correlation_window:i);
    r_temp1 = r_lidar1_ipol(i-correlation_window:i);

    % Subtract mean
    r_temp0 = (r_temp0 - mean(r_temp0));
    r_temp1 = (r_temp1 - mean(r_temp1));

    % correlate
    corr = xcorr(r_temp0 ,r_temp1);

    % Find element with highest correlation
    max_index = find(corr == max(corr(:)));

    % If multiple found, no correleation. Set index minimal speed
    if size(max_index,1) > 1
        max_index = correlation_window;
    elseif max_index < 1
        max_index = correlation_window;
    end

    %corr_time_ms = max_index-correlation_window
    corr_time_ms_left = [corr_time_ms_left; max_index-correlation_window];
    corr_speed_left = [corr_speed_left; lidar_distance / (corr_time_ms_left(end)*0.001)]; % m/s
    t_corr_l = [t_corr_l; t(i)];
end
% Interploate to 1 ms resolution
corr_speed_left_ipol = interp1(t_corr_l,corr_speed_left,t);

% Start Loop; right side
for i = 1+offset: 100 : size(t,1)
    % Extract measurements
    r_temp0 = r_lidar3_ipol(i-correlation_window:i);
    r_temp1 = r_lidar2_ipol(i-correlation_window:i);

    %Substract mean
    r_temp0 = (r_temp0 - mean(r_temp0));% .* hamming(correlation_window+1);
    r_temp1 = (r_temp1 - mean(r_temp1));% .* hamming(correlation_window+1);

    % Correlate
    corr = xcorr(r_temp0 ,r_temp1);

    % Find element with max correlation
    max_index = find(corr == max(corr(:)));

    % If multiple found, no correleation. Set index minimal speed
    if size(max_index,1) > 1
        max_index = 1;
    elseif max_index < 1
        max_index = 1;
    end

    %corr_time_ms = max_index-correlation_window
    corr_time_ms_right = [corr_time_ms_right; max_index-correlation_window];
    corr_speed_right = [corr_speed_right; lidar_distance/(corr_time_ms_right(end)*0.001)]; % m/s
```

```
    t_corr_r = [t_corr_r; t(i)];
end
corr_speed_right_ipol = interp1(t_corr_r,corr_speed_right,t);

%% Evaluation

% Calculate errors
e_l = can_speed_ipol(1+offset: size(t,1)- offset)/3.6 - corr_speed_left_ipol(1+offset: size(t,1)- offset);
e_r = can_speed_ipol(1+offset: size(t,1)- offset)/3.6 - corr_speed_right_ipol(1+offset: size(t,1)- offset);

% Find elements in Window
in_l = find(abs(e_l) < 1/3.6);
in_r = find(abs(e_r) < 1/3.6);

% Calculate Rate
rate_l = size(in_l,1)/size(e_l,1)
rate_r = size(in_r,1)/size(e_r,1)
```

Position

```
%% Analysis and implentation of Speed-Algorithm
% AuswertungPosition.m
%
% Author: Mario Wegner
% Date: 1/2017
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Lidar Error handling Linearse measurements under 15 cm
r_lidar0_filtered = r_lidar0;
r_lidar1_filtered = r_lidar1;
r_lidar2_filtered = r_lidar2;
r_lidar3_filtered = r_lidar3;

i = 2;
for i = 2 : 1 :size(r_lidar3,1)
     if r_lidar3_filtered(i) < 0.15
          r_lidar3_filtered(i) = 0.5*(r_lidar3_filtered(i-1)+ r_lidar3_filtered(i+1));
    end
end

i = 2;
for i = 2 : 1 :size(r_lidar2,1)
     if r_lidar2_filtered(i) < 0.15
          r_lidar2_filtered(i) = 0.5*(r_lidar2_filtered(i-1)+ r_lidar2_filtered(i+1));
    end
end

i = 2;
for i = 2 : 1 :size(r_lidar1,1)
     if r_lidar1_filtered(i) < 0.15
          r_lidar1_filtered(i) = 0.5*(r_lidar1_filtered(i-1)+ r_lidar1_filtered(i+1));
    end
end
i = 2;
for i = 2 : 1 :size(r_lidar0,1)
     if r_lidar0_filtered(i) < 0.15
          r_lidar0_filtered(i) = 0.5*(r_lidar0_filtered(i-1)+ r_lidar0_filtered(i+1));
    end
end

% Interpolate Measurements to ms resolution
% Create Time vector
t_start = 1490360200000;
t_stop  = t_gps(end);
t = (t_start : t_stop)';

% Interpolate to 1 ms resolution
can_speed_ipol = interp1(t_can,can_speed,t);
GPS_speed_ipol = interp1(t_gps,GPS_speed,t);
GPS_direction_ipol = interp1(t_gps,GPS_direction,t);
GPS_lat_ipol = interp1(t_gps,GPS_lat,t);
GPS_lon_ipol = interp1(t_gps,GPS_lon,t);

% Cancel time differences and interpolate
r_lidar0_ipol = interp1(t_lidar0,r_lidar0_filtered,t-25000);
r_lidar1_ipol = interp1(t_lidar1,r_lidar1_filtered,t-25000);
r_lidar2_ipol = interp1(t_lidar2,r_lidar2_filtered,t-25000);
r_lidar3_ipol = interp1(t_lidar3,r_lidar3_filtered,t-25000);

%% Extract Rounds
% First round (start 13:10:46; end 13:13:37)
startdate_index_1   = find(131046 == floor(GPS_time));% 7558; % [131046.001000000]
```

```
stopdate_index_1    = find(131337 == floor(GPS_time));% 9268; %[131337.001000000]
start_time_1    = t_gps(startdate_index_1);
stop_time_1     = t_gps(stopdate_index_1);
start_index_1 = find( floor(start_time_1(1)) == t(:));
stop_index_1 = find( floor(stop_time_1(1)) == t(:));
t_1 = 0 : (stop_index_1 - start_index_1);

% Extraxt values for round
GPS_lat_1   = GPS_lat_ipol(start_index_1:stop_index_1);
GPS_lon_1   = GPS_lon_ipol(start_index_1:stop_index_1);
can_speed_1 = can_speed_ipol(start_index_1:stop_index_1)/3.6; % m/s
route_1     = cumsum(can_speed_1)/1000; % Samplerate 1 kHz

r_lidar_0_1   = r_lidar0_ipol(start_index_1:stop_index_1);
r_lidar_1_1   = r_lidar1_ipol(start_index_1:stop_index_1);
r_lidar_2_1   = r_lidar2_ipol(start_index_1:stop_index_1);
r_lidar_3_1   = r_lidar3_ipol(start_index_1:stop_index_1);


% Second Round (start 13:13:37; end 13:16:05)
startdate_index_2   = find(131337 == floor(GPS_time));
stopdate_index_2    = find(131605 == floor(GPS_time));
start_time_2    = t_gps(startdate_index_2);
stop_time_2     = t_gps(stopdate_index_2);
start_index_2 = find( floor(start_time_2(1)) == t);
stop_index_2 = find( floor(stop_time_2(1)) == t);
t_2 = 0 : (stop_index_2 - start_index_2);

% Extraxt values for round
GPS_lat_2 = GPS_lat_ipol(start_index_2:stop_index_2);
GPS_lon_2 = GPS_lon_ipol(start_index_2:stop_index_2);
can_speed_2 = can_speed_ipol(start_index_2:stop_index_2)/3.6; % m/s
route_2     = cumsum(can_speed_2)/1000; % Samplerate 1 kHz
r_lidar_0_2   = r_lidar0_ipol(start_index_2:stop_index_2);
r_lidar_1_2   = r_lidar1_ipol(start_index_2:stop_index_2);
r_lidar_2_2   = r_lidar2_ipol(start_index_2:stop_index_2);
r_lidar_3_2   = r_lidar3_ipol(start_index_2:stop_index_2);


% Third Round (start 13:16:05; end 13:20:07)
startdate_index_3   = find(131606 == floor(GPS_time));
stopdate_index_3    = find(132007 == floor(GPS_time));
start_time_3    = t_gps(startdate_index_3);
stop_time_3     = t_gps(stopdate_index_3);
start_index_3 = find( floor(start_time_3(1)) == t);
stop_index_3 = find( floor(stop_time_3(1)) == t);
t_3 = 0 : (stop_index_3 - start_index_3);

% Extraxt values for round
GPS_lat_3   = GPS_lat_ipol(start_index_3:stop_index_3);
GPS_lon_3   = GPS_lon_ipol(start_index_3:stop_index_3);
can_speed_3 = can_speed_ipol(start_index_3:stop_index_3)/3.6; % m/s
route_3     = cumsum(can_speed_3)/1000; % Samplerate 1 kHz
r_lidar_0_3   = r_lidar0_ipol(start_index_3:stop_index_3);
r_lidar_1_3   = r_lidar1_ipol(start_index_3:stop_index_3);
r_lidar_2_3   = r_lidar2_ipol(start_index_3:stop_index_3);
r_lidar_3_3   = r_lidar3_ipol(start_index_3:stop_index_3);

%% Transform to Grid

grid_size = 0.01; % m
window_size = 25 / grid_size;     % m
step_wide = 1; % m
tolerance = 0.05; % of one

s = (0.01 : grid_size : 720)';
[x, index_1] = unique(route_1);
r_lidar_1_1_grid = interp1(route_1(index_1), r_lidar_1_1(index_1),s);
r_lidar_2_1_grid = interp1(route_1(index_1), r_lidar_2_1(index_1),s);
r_lidar_0_1_grid = interp1(route_1(index_1), r_lidar_0_1(index_1),s+0.51);
r_lidar_3_1_grid = interp1(route_1(index_1), r_lidar_3_1(index_1),s+0.51);
can_speed_1_grid = interp1(route_1(index_1), can_speed_1(index_1),s);
GPS_lat_1_grid = interp1(route_1(index_1), GPS_lat_1(index_1), s);
GPS_lon_1_grid = interp1(route_1(index_1), GPS_lon_1(index_1), s);

[x, index_2] = unique(route_2);
r_lidar_1_2_grid = interp1(route_2(index_2), r_lidar_1_2(index_2),s);
r_lidar_2_2_grid = interp1(route_2(index_2), r_lidar_2_2(index_2),s);
r_lidar_0_2_grid = interp1(route_2(index_2), r_lidar_0_2(index_2),s+0.51);
r_lidar_3_2_grid = interp1(route_2(index_2), r_lidar_3_2(index_2),s+0.51);
can_speed_2_grid = interp1(route_2(index_2), can_speed_2(index_2),s);
GPS_lat_2_grid = interp1(route_2(index_2), GPS_lat_2(index_2), s);
GPS_lon_2_grid = interp1(route_2(index_2), GPS_lon_2(index_2), s);

[x, index_3] = unique(route_3);
```

```
r_lidar_1_3_grid = interp1(route_3(index_3), r_lidar_1_3(index_3),s);
r_lidar_2_3_grid = interp1(route_3(index_3), r_lidar_2_3(index_3),s);
r_lidar_0_3_grid = interp1(route_3(index_3), r_lidar_0_3(index_3),s+0.51);
r_lidar_3_3_grid = interp1(route_3(index_3), r_lidar_3_3(index_3),s+0.51);
can_speed_3_grid = interp1(route_3(index_3), can_speed_3(index_3),s);
GPS_lat_3_grid = interp1(route_3(index_3), GPS_lat_3(index_3), s);
GPS_lon_3_grid = interp1(route_3(index_3), GPS_lon_3(index_3), s);


%% Generate map
% first Round is used as reference -> line c11
% c stands for one line between two notes
% It contains map data and the corresbonding position
c = [];
c.route = s;
c.left  = r_lidar_1_1_grid;
c.right = r_lidar_2_1_grid;
c.lat   = GPS_lat_1_grid;
c.lon   = GPS_lon_1_grid;


%% Find correlation by RMS
rms_r = [];
rms_l = [];
x_r = [];
x_l = [];
x_soll = [];
position = [];

% Test Loop
for j = window_size+1 : step_wide/grid_size : c.route(end)/grid_size

    point_to_find = j
    % Extract Measurements
    zl = r_lidar_1_2_grid(point_to_find-window_size:point_to_find);
    zr = r_lidar_2_2_grid(point_to_find-window_size:point_to_find);

%% RMS Algorithm
for i = window_size+1 :50: c.route(end)/grid_size

    % Difference between Measurement and Map
    difference_r = zr - c.right(i-window_size:i);
    difference_l = zl - c.left(i-window_size:i);

    % Calculate RMS Error and safe position
    rms_r = [rms_r ; rms(difference_r)];
    rms_l = [rms_l ; rms(difference_l)];
    position = [position ; i];
end
    % Find element with smalest error
    index_l = find(rms_l == min(rms_l));
    index_r = find(rms_r == min(rms_r));
    % Save this as Position
    x_l =  [x_l position(index_l(1))];
    x_r =  [x_r position(index_r(1))];

    rms_r = [];
    rms_l = [];
    position = [];
    x_soll = [x_soll j];
       % pause;
end

%% Analysis

% Calculate error
e_l = x_soll - x_l;
e_r = x_soll - x_r;

% Find elements in tolerace
in_l = find(abs(e_l) < 72000*tolerance);
in_r = find(abs(e_r) < 72000*tolerance);

% Calculate success rate
rate_l = size(in_l,2)/size(e_l,2);
rate_r = size(in_r,2)/size(e_r,2);
```

B.3. Programmablaufpläne

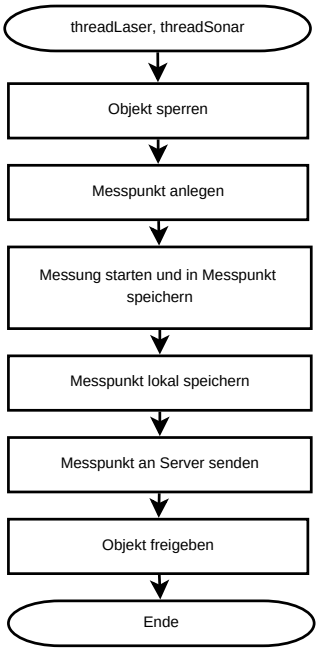

Abbildung B.6.: Programmablaufplan der Funktionen *threadLaser* und *threadSonar*

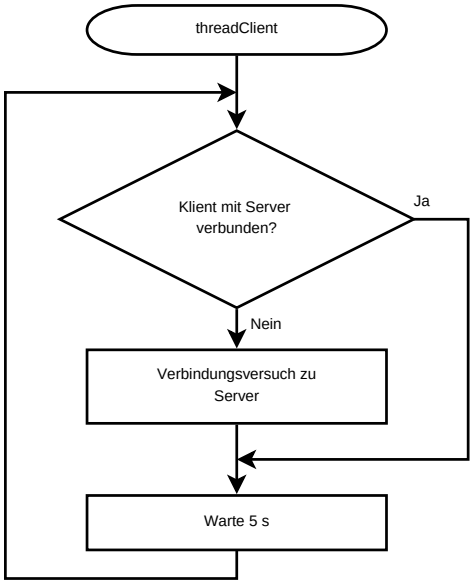

Abbildung B.7.: Programmablaufplan der Funktion threadClient

Tabellenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[1] Bundesnetzagentur. Allgemeinzuteilung von frequenzen fuer kraftfahrzeug-kurzstreckenradare im frequenzbereich 21,65- 26,65 ghz, 2012. https://www.bundesnetzagentur.de/SharedDocs/Downloads/DE/Sachgebiete/Telekommunikation/Unternehmen_Institutionen/Frequenzen/Allgemeinzuteilungen/2012_41_KfzKurzstreckenradar24GHz_pdf aufgerufen am 6.2.2017.

[2] Bundesnetzagentur. Allgemeinzuteilung von frequenzen fuer kraftfahrzeug-kurzstreckenradare im frequenzbereich 77 -81 ghz, 2014. https://www.bundesnetzagentur.de/SharedDocs/Downloads/DE/Sachgebiete/Telekommunikation/Unternehmen_Institutionen/Frequenzen/Allgemeinzuteilungen/2004_59_KfzKurzstreckenradar79GHz_pdf aufgerufen am 6.2.2017.

[3] Howie Choset and Keiji Nagatani. *Topological Simultaneous Localization and Mapping (SLAM): Toward Exact Localization Without Explicit Localization*. IEEE, 2001.

[4] Continental Corp. Srr 20x /-2 /-2c /-21 datasheet, 2016. http://www.conti-online.com/www/download/industrial_sensors_de_en/themes/download/srr20x_datasheet_en.pdf aufgerufen am 6.2.2017.

[5] Paloma de la Puente and Diego Rodriguez-Losada. *Feature based graph SLAM with high level representation using rectangles*. Elsevier, 2014.

[6] Waveshare Electronics. High-precision ad/da board, 2016. http://www.waveshare.com/wiki/High-Precision_AD/DA_Board aufgerufen am 25.3.2017.

[7] U.S. Air Force. Gps accuracy, 2017. http://www.gps.gov/systems/gps/performance/accuracy/ aufgerufen am 7.4.2017.

[8] Raspberry Pi Foundation. Dokumentation: Raspberry pi 3 model b, 2017. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/ aufgerufen am 28.2.2017.

[9] Raspberry Pi Foundation. Download raspian jessie lite, 2017. https://www.raspberrypi.org/downloads/raspbian/ aufgerufen am 28.2.2017.

[10] Bundesanstalt für Straßenwesen. Rechtsfolgen zunehmender fahrzeugautomatisierung, 2012. http://www.bast.de/DE/Publikationen/Foko/Downloads/2012-11.pdf?__blob=publicationFile aufgerufen am 7.4.2017.

[11] Arturo Gil, Miguel Julia, and Oscar Reinoso. *Occupancy grid based graph-SLAM using the distance transform, SURF features and SGD*. Elsevier, 2014.

[12] Bosch GmbH. Stereo video camera, 2017. http://products.bosch-mobility-solutions.com/en/de/_technik/component/CO_PC_DA_Traffic_Jam_Assist__CO_PC_Driver-Assistance_5056.html?compId=5824 aufgerufen am 31.3.2017.

[13] Bosch Engineering GmbH. Long-range-radar lrr3 datasheet, 2010. http://www.bosch-engineering.de/media/de/pdfs/einsatzgebiete_1/produktdatenblaetter/120903_LRR3_EN_V05_final.pdf aufgerufen am 6.2.2017.

[14] Richard Günter, Tobias Wenzel, Mario Wegner, and Rasmus Rettig. *Big data driven dynamic driving cycle development for busses in urban public transportation*. Transportation Research Part D: Transport and Environment Volume 51, March 2017, Pages 276 to 289, 2017.

[15] Gordon Henderson. Wiring pi; gpio interface library for the raspberry pi, 2016. http://wiringpi.com/ aufgerufen am 15.2.2017.

[16] MaxBotix Inc. Xl-maxsonar-ez/ae series datasheet, 2005. http://www.maxbotix.com/documents/XL-MaxSonar-EZ_Datasheet.pdf aufgerufen am 5.2.2017.

[17] MaxBotix Inc. I2cxl-maxsonar datasheet, 2012. http://www.maxbotix.com/documents/I2CXL-MaxSonar-EZ_Datasheet.pdf aufgerufen am 5.2.2017.

[18] Parallax Inc. Ping))) ultrasonic distance sensor datasheet, 2012. https://www.parallax.com/sites/default/files/downloads/28015-PING-Documentation-v1.6.pdf aufgerufen am 5.2.2017.

[19] Tesla Inc. Hardware für autonomes fahren in allen fahrzeugen, 2017. https://www.tesla.com/de_DE/autopilot aufgerufen am 7.4.2017.

[20] Texas Instruments. Ads1256/ads1255 datasheet, 2013. im Anhang als: DatasheetADS1256.pdf.

[21] Ulla Kirch and Peter Prinz. *C++ - Lernen und professionell anwenden*. mitp Verlags GmbH und Co. KG, 2015. ISBN: 978-3-95845-028-8.

[22] LightWare Optoelectronics Ltd. Sf02 f laser rangefinder datasheet, 2016. im Anhang als: SF02 - Laser Rangefinder Manual - Rev 11.pdf.

[23] Murata Manufacturing. Product specification ma40s4r, 2015. http://www.murata.com/en-us/products/productdata/8797589274654/MASPOPRE.pdf?1450668610000 aufgerufen am 5.2.2017.

[24] Mike McCauley. C library for broadcom bcm 2835 as used in raspberry pi, 2016. http://www.airspayce.com/mikem/bcm2835/ aufgerufen am 15.2.2017.

[25] Wojciech Owczarek. Precision time protocol daemon, 2015. https://github.com/ptpd/ptpd aufgerufen am 5.4.2017.

[26] Konrad Reif. *Fahrstabilisierungssysteme und Fahrerassistenzsysteme*. Vieweg + Teubner, 2010. ISBN: 978-3-8348-1314-5.

[27] Konrad Reif. *Sensoren im Kraftfahrzeug*. Springer Vieweg, 2012. ISBN: 978-3-8348-1778-5.

[28] SanDisk. Unterschied zwischen speed class, uhs speed class, und speed ratings (lese- und schreibgeschwindigkeiten) bei sd/sdhc/sdxc karten, 2017. https://kb-de.sandisk.com/app/answers/detail/a_id/8054/~/unterschied-zwischen-speed-class%2C-uhs-speed-class%2C-und-speed-ratings-%28lese--und aufgerufen am 2.4.2017.

[29] Jakob Sannert. *Studienarbeit:*. Hochschule für Angewandte Wissenschaften Hamburg, 2017.

[30] Bruno Siciliano and Oussama Khatib. *Springer Handbook of Robotics*. Springer Verlag Berlin Heidelberg, 2016. ISBN: 978-3-319-32552-1.

[31] EndRun Technologies. White paper: Precision time protocol (ptp/ieee-1588), 2011. https://www.endruntechnologies.com/pdf/PTP-1588.pdf aufgerufen am 5.4.2017.

[32] Sebastian Thrun. *Probalistic robotics*. The MIT Press, 2005. ISBN: 978-0-262-20162-9.

[33] Mario Wegner. *Bachelorthesis: Entwicklung eines autonom arbeitenden GPS-Datenloggers mit hoher Updatefrequenz für die Anwendung in Nahverkehrsbussen*. Hochschule für Angewandte Wissenschaften Hamburg, 2015.

[34] Hermann Winner. *Handbuch Fahrerassistenzsysteme*. Vieweg + Teubner, 2015. ISBN: 978-3-658-05733-6.

[35] Jürgen Wolf. *Linux-Unix-Programmierung : das umfassende Handbuch*. Rheinwerk Verlag, 2016. ISBN: 3-8362-3772-5.

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Hamburg, 10. April 2017	
Ort, Datum	Unterschrift