Fachhochschule Westküste
Hochschule für Wirtschaft & Technik

Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Christopher Rotzlawski

Entwicklung eines umfeldsensorbasierten Navigationssystems für die Positionserkennung autonomer Fahrzeuge in Indoor-Bereichen

Fachhochschule Westküste
Fachbereich Technik

Fachhochschule Westküste - University of Applied Sciences
Faculty of Engineering

Hochschule für Angewandte Wissenschaften Hamburg
Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Hamburg University of Applied Sciences
Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Christopher Rotzlawski

Entwicklung eines umfeldsensorbasierten Navigationssystems für die Positionserkennung autonomer Fahrzeuge in Indoor-Bereichen

Masterarbeit eingereicht im Rahmen der Masterprüfung
im gemeinsamen Studiengang Mikroelektronische Systeme
am Fachbereich Technik
der Fachhochschule Westküste
und
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. Stephan Hussmann

Abgegeben am 19. März 2020

Christopher Rotzlawski

Thema der Masterarbeit

Entwicklung eines umfeldsensorbasierten Navigationssystems für die Positionserkennung autonomer Fahrzeuge in Indoor-Bereichen

Stichworte

Positionsbestimmung, Orientierungsbestimmung, Umfelderfassung, Raytracing-Simulation, Autonome Fahren, Robot Operating System (ROS), Linux, Ultraschall

Kurzzusammenfassung

Diese Arbeit befasst sich mit der Entwicklung eines Systems zur Bestimmung von Position und Orientierung autonomer Fahrzeuge in Indoor-Bereichen. Es wird ein Verfahren entwickelt, mit der eine Umgebungserkennung ermöglicht wird und so eine zeitweise, alternative Methode zur Satellitennavigation darstellt.

Christopher Rotzlawski

Title of the master thesis

Development of a surrounding sensorbased navigation system for the position detection of autonomous vehicles in indoor areas

Keywords

Position detection, Orientation detection, Surround sensing, Raytracing simulation, Autonomous driving, Robot Operating System (ROS), Linux, Ultrasound

Abstract

This thesis deals with the development of a system for the determination of position and orientation of autonomous vehicles in indoor areas. A method is being developed to enable environment detection, providing a temporary, alternative method of satellite navigation.

Inhaltsverzeichnis

1. Einführung und Motivation

Unter automatisiertem Fahren bezeichnet man Fahrerassistenzsysteme, mithilfe derer der Fahrzeugführer teilweise entlastet wird, wohingegen beim autonomen Fahren kein Fahrzeugführer mehr erforderlich sein soll. Hierbei sind teilautomatisierte Systeme schon länger als Fahrzeugausstattung verfügbar. Hochautomatisierte Systeme, die ohne menschliche Hilfe Teilbereiche des Fahrens abdecken, sind ebenfalls als Fahrzeugausstattung wählbar beziehungsweise stehen kurz vor der Serienreife. Des Weiteren werden weltweit vollautomatisierte und autonome Fahrzeuge, welche kein menschliches Eingreifen mehr erfordern sollen, auf unterschiedlichen Teststrecken erprobt (vgl. Ethik-Kommission, 2017, S. 6).

Dabei unterteilt man den Automatisierungsgrad in fünf Stufen (siehe Abbildung 1.1). Diese starten mit Stufe 1 beim assistierten Fahren. Hier übernimmt das Fahrzeug zum Beispiel die Längs- oder Querführung. Ab Stufe 2 wird von teilautomatisierten Fahren gesprochen, bei dem die Längs- und Querführung vom Fahrzeug übernommen wird. Hierbei muss der Fahrer jederzeit in der Lage sein, das Fahrzeug kurzfristig zu übernehmen. Bei Stufe 3, das hochautomatisierte Fahren, erkennt das Fahrzeug eigenständig die Systemgrenzen. Die Längs- und Querführung muss in diesem Fall nicht dauerhaft überwacht werden. Der Fahrer muss allerdings mit einer gewissen Zeitreserve in der Lage sein, das Fahrzeug nach Aufforderung zu übernehmen. Stufe 4 bedeutet vollautomatisiertes Fahren in spezifischen Anwendungsfällen, die vom Straßentyp, der Geschwindigkeit und den Umgebungsbedingungen abhängig sind. Hier wird die komplette Fahraufgabe an das Fahrzeug übergeben. Die letzte Stufe ist das fahrerlose, autonome Fahren. Dies bedeutet, dass das Fahrzeug alle Anwendungsfälle vollständig allein durchführen kann (vgl. VDA, 2015, S. 14).

Da sich ab der dritten Stufe die Verantwortung mehr Richtung Fahrzeug bewegt, ist es nötig, dass Fahrzeuge selbstständig navigieren können. Hierzu ist eine eigenständige Positionsbestimmung erforderlich. Dies geschieht insbesondere mithilfe von satellitengestützter Navigation (GNSS), wie zum Beispiel GPS oder GLONASS (vgl. Niehues, 2014, S. 1). Um in diesem Bereich robuste Gesamtsysteme mit einer hohen Einsatzbereitschaft zu realisieren, kommt es auf eine hohe Güte der Positionsbestimmung an. Eine bessere Güte resultiert dabei in einer verbesserten Performanz des Gesamtsystems (vgl. Niehues, 2014, S. 2). Dies stellt allerdings Fahrzeuge ab der dritten Automatisierungsstufe vor das Problem, dass eine zuverlässige Positionsbestimmung mithilfe von satellitengestützter Navigation von mehreren Faktoren deutlich beeinflusst werden kann, wie etwa der Beschaffenheit der Umgebung und des Wetters. So kann es dazu kommen, dass unterhalb von Baumkronen in

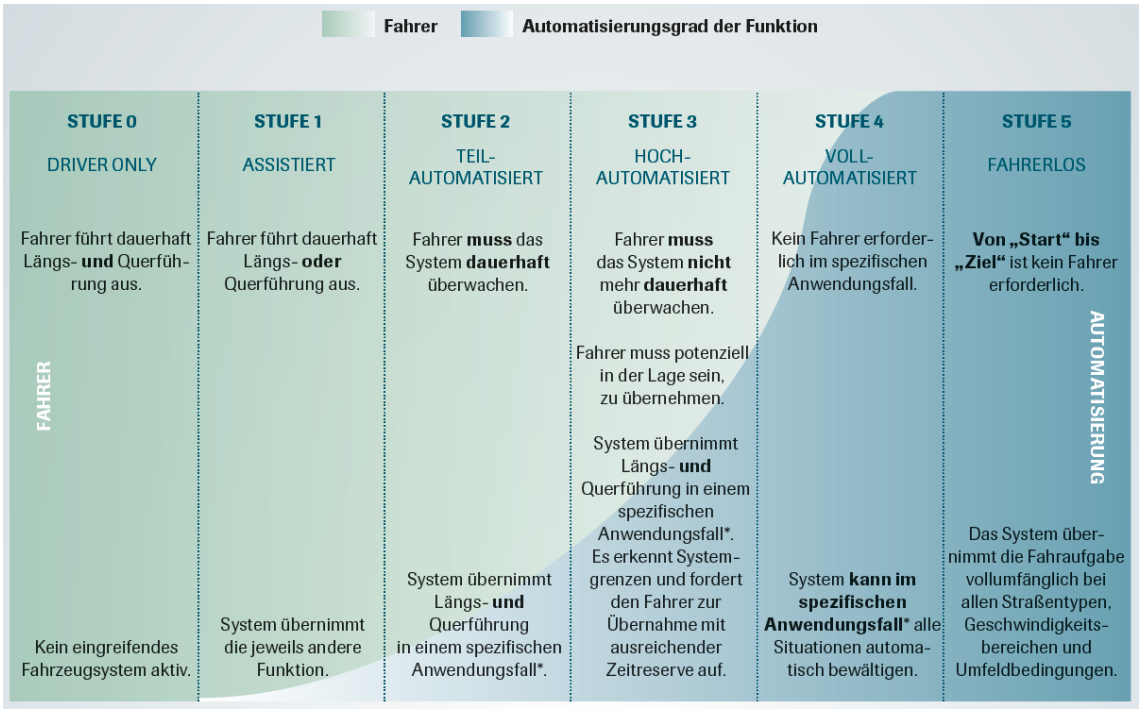


Abbildung 1.1.: Stufen des automatisierten und autonomen Fahrens (VDA, 2015, S. 15)

Alleen keine Verbindung zu Navigationssatelliten besteht (vgl. Dudek und Jenkin, 2016, S. 746). In *Urban Canyons*, also in Häuserschluchten, kann es zur Verfälschung des Satellitensignals kommen, da hier oft die Signale der Navigationssatelliten nicht direkt empfangen, sondern über Hauswände zum Empfänger reflektiert werden. Hierbei handelt es sich, wie in Abbildung 1.2 zu sehen, um den sogenannten *Multipath*-Fehler (vgl. NOAA, 2019). Wohingegen bei ungünstiger Beschaffenheit des Wetters, wie zum Beispiel starkem Regen oder Schneefall, die Verbindung zu den Navigationssatelliten auch bei optimaler Beschaffenheit der Umgebung abbrechen kann (vgl. Dudek und Jenkin, 2016, S. 746). Ein erster Ansatz, um die GNSS-basierte Positionsbestimmung kurzzeitig zu verbessern, ist die Kombination mit einem inertialen Navigationssystem (INS). Hierbei wird mithilfe von Inertialsensorik eine weitere zur GNSS-basierten Positionsbestimmung unabhängige Position bestimmt. Anschließend wird mit einem Kalman-Filter aus der GNSS-Position und der INS-Position eine verbesserte Fahrzeugposition ermittelt (vgl. Tanil u. a., 2019, S. 2).

Da bei einem nicht verfügbaren GNSS beziehungsweise einer schlechten Güte der GNSS-Positionsbestimmung ein INS nur kurzzeitig für eine Verbesserung der Positionsbestimmung sorgt, ist ein Ziel des *Urban Mobility Labs* die Entwicklung alternativer Methoden zur Positionsbestimmung von autonomen Fahrzeugen. Hierzu werden Sensorsysteme verwendet und entwickelt, mit deren Hilfe eine Erkennung von Umgebung und Position des Fahrzeugs ohne satellitengestützte Navigation realisiert werden soll. Zum Einsatz kommen hierbei unter

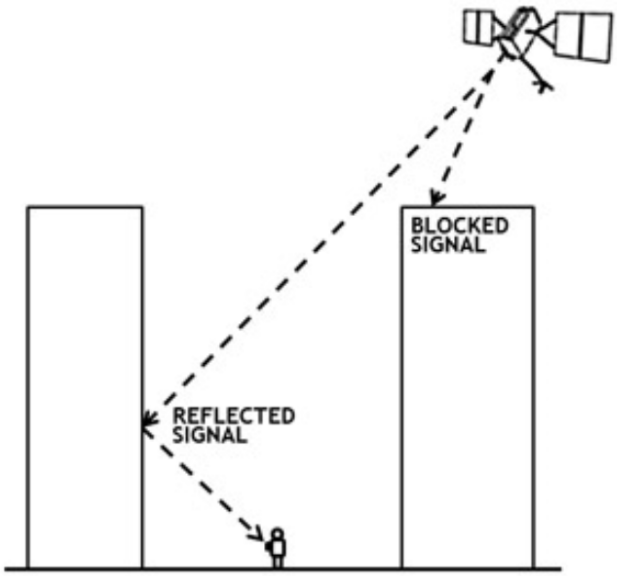


Abbildung 1.2.: Verfälschung des Satellitensignals durch den *Multipath*-Fehler (NOAA, 2019)

anderem Umfeldsensoren, wie zum Beispiel Ultraschall und LiDAR. Des Weiteren soll eine Positionsbestimmung mithilfe von kamerabasierten Systemen umgesetzt werden, die auf Stereo- und sphärische Kameras zurückgreifen.

Da Signale von Navigationssatelliten in Indoor-Bereichen nicht verfügbar beziehungsweise stark verfälscht sind und die Erkennung beziehungsweise Korrektur der Position kaum möglich ist, ist das Ziel dieser Arbeit die Entwicklung eines umfeldsensorbasierten Navigationssystems. Mit diesem soll eine Bestimmung von Position und Orientierung eines autonomen Fahrzeugs in Indoor-Bereichen, wie zum Beispiel Tiefgaragen, ermöglicht werden. Des Weiteren sollen verdeckte Objekte, die sich außerhalb des sichtbaren Bereichs befinden, erkannt werden. Dies soll mithilfe von Ultraschall als Umfeldsensorik realisiert werden, da dies mit der Auswertung von Mehrfachreflexionen möglich ist. Bei der Verwendung weiterer Umfeldsensorik neben Ultraschall sollen die Position und die Orientierung mithilfe von Sensordatenfusion aus den einzelnen Teilsystemen bestimmt werden. Hierbei soll das umfeldsensorbasierte Navigationssystem modular und skalierbar sein. Des Weiteren ist eine Integration des Systems in das Versuchsfahrzeug des *Urban Mobility Lab* geplant.

2. Stand der Technik

Das folgende Kapitel beschreibt gängige Grundbegriffen, die im Zusammenhang mit dieser Arbeit stehen. Hierbei wird zunächst auf das Thema Ultraschall eingegangen. Des Weiteren werden relevante Vorarbeiten des *Urban Mobility Lab* beschrieben.

2.1. Ultraschall

Im folgenden Abschnitt werden zunächst Grundlagen, die im Zusammenhang mit dem Thema Ultraschall stehen, beschrieben. Anschließend wird auf Ultraschallsensoren, die Abstandsmessung mit Ultraschallsensoren und deren Eigenschaften näher eingegangen. Abschließend wird der aktuelle Stand der Technik im Bezug auf Positionsbestimmung und Mapping mithilfe von Ultraschall beschrieben.

2.1.1. Grundlagen

Schallwellen sind mechanische Wellen, die sich in Gasen und anderen Medien ausbreiten. Dabei werden Schallwellen, deren Frequenz über 20 kHz liegt, als Ultraschall bezeichnet. Die Schallwellen breiten sich hierbei als longitudinale Druckwellen aus, bei dem der Druck der Schallwelle periodisch um den normalen Raumdruck schwankt (vgl. Eichler, 2014, S. 173).

Dabei ist die Schallgeschwindigkeit von der Umgebungstemperatur, dem Umgebungsdruck und der Dichte der Luft abhängig. So kann mit Formel 2.1 die Schallgeschwindigkeit in idealen Gasen berechnet werden. Hierbei ist γ das Verhältnis der spezifischen Wärmekapazität, P der Umgebungsdruck und ρ die temperaturabhängige Dichte der Luft (vgl. Albuquerque, 2013, S. 25).

$$c = \sqrt{\gamma \frac{P}{\rho}} \tag{2.1}$$

Hierbei beträgt die temperaturabhängige Dichte der Luft bei $0^\circ C$ 1,2922 $\frac{kg}{m^3}$ und bei $20^\circ C$ 1,2041 $\frac{kg}{m^3}$. Das Verhältnis der spezifischen Wärmekapazität von Luft ist dabei $\gamma = 1{,}402$. Der

Umgebungsdruck weist einen Wert von 101325 Pa auf. So erhält man bei einer Umgebungstemperatur von 0°C folgende Schallgeschwindigkeit (vgl. Albuquerque, 2013, S. 25):

$$c_0 = \sqrt{1,402 \frac{101325\,Pa}{1,2922\,\frac{kg}{m^3}}} = 331,57\,\frac{m}{s} \tag{2.2}$$

Für eine Umgebungstemperatur von 20°C hingegen berechnet sich eine Schallgeschwindigkeit von (vgl. Albuquerque, 2013, S. 25):

$$c_0 = \sqrt{1,402 \frac{101325\,Pa}{1,2041\,\frac{kg}{m^3}}} = 343,48\,\frac{m}{s} \tag{2.3}$$

Des Weiteren kann die Schallgeschwindigkeit als Funktion der absoluten Temperatur T_K dargestellt werden (siehe Formel 2.4). Hierbei stellt R die universelle Gaskonstante und M die Masse des Gases dar. Hierdurch ist die Schallgeschwindigkeit proportional zur Quadratwurzel der absoluten Temperatur (vgl. Albuquerque, 2013, S. 25).

$$c = \sqrt{\gamma \frac{RT_K}{M}} \tag{2.4}$$

Da die Schallgeschwindigkeit unter typischen Raumbedingungen bei Druckschwankungen näherungsweise konstant ist, kann diese als Funktion der Umgebungstemperatur beschrieben werden. Mit der folgenden Formel kann die Schallgeschwindigkeit berechnet werden, wobei die Temperatur θ in °C angegeben wird (vgl. Albuquerque, 2013, S. 25):

$$c_{Luft} = c_0 \sqrt{\frac{T_K}{273,15}} = 331,57 \sqrt{\frac{\theta}{273,15} + 1} \frac{m}{s} \tag{2.5}$$

Näherungsweise kann die Schallgeschwindigkeit mit der folgenden vereinfachten Formel berechnet werden (vgl. Eichler, 2014, S. 175):

$$c_{Luft} = \left(331,4 + 0,6 \frac{\theta}{^\circ C}\right) \frac{m}{s} \tag{2.6}$$

In Abbildung 2.1 ist ein Vergleich der Formeln 2.5 und 2.6 zu sehen. Hierbei ist zu erkennen, dass im Bereich von 0°C bis 30°C der Fehler, der durch die vereinfachte Formel verursacht wird, bei maximal 0,17 $\frac{m}{s}$ liegt und somit vernachlässigbar klein ist.

Die Schallgeschwindigkeit wird des Weiteren von der relativen Luftfeuchtigkeit beeinflusst. Hierbei erhöht sich diese bei einer Änderung der relativen Luftfeuchtigkeit von 0 % auf 100 % gleichzeitig um circa 0,2 $\frac{m}{s}$. Der dabei entstehende maximale Fehler beträgt bei einer Umgebungstemperatur von 20°C 0,058% und ist damit vernachlässigbar klein. Eine Kompen-

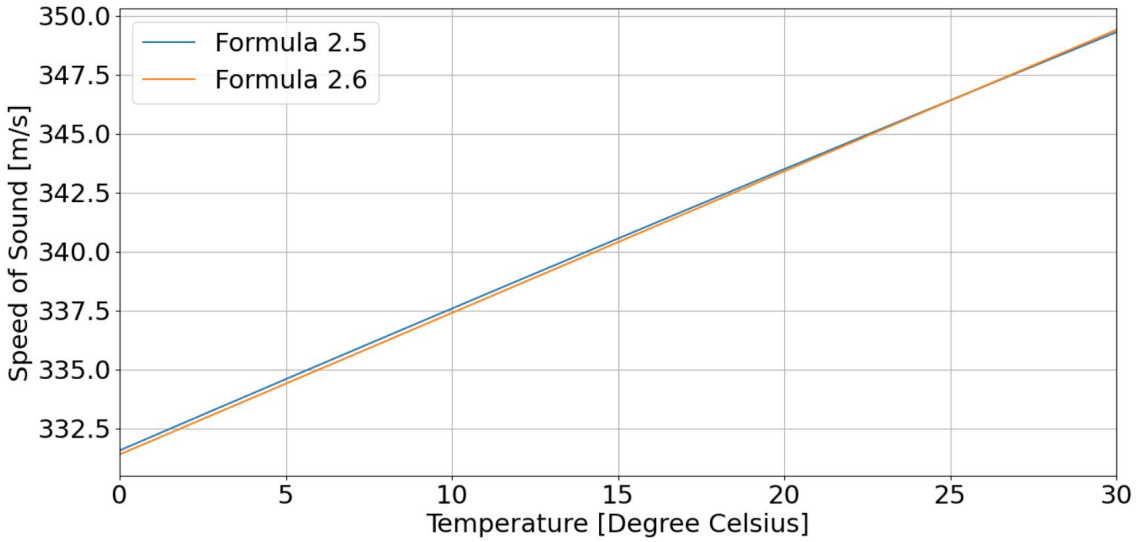


Abbildung 2.1.: Vergleich der Formeln 2.5 und 2.6 zur Berechnung der Schallgeschwindigkeit

sation der Messergebnisse ist somit in den meisten Fällen nicht erforderlich (vgl. Hering und Schönfelder, 2012, S. 277).

Ein weiterer Punkt, der die Schallwellen beeinflusst, ist die Absorption in Luft. Die Intensität I nimmt dabei mit zunehmenden Weg x ab. Hierbei hängt die Absorption von der Frequenz der Schallwellen ab, das heißt, dass mit zunehmender Frequenz die maximale Reichweite sinkt (siehe Abbildung 2.2). In die Berechnung der Intensität fließt des Weiteren ein Proportionalitätsfaktor a ein (vgl. Hering und Schönfelder, 2012, S. 110):

$$I = I_0 e^{-af^2 x} \tag{2.7}$$

Die Absorption wird dabei durch innere Reibung, Wärmeleitung und molekulare Absorption verursacht. Die innere Reibung beschreibt die dynamische Viskosität. Bei Absorption durch Wärmeleitung verursachen die Schallwellen hingegen periodische Temperaturänderungen innerhalb des Mediums, wodurch den Schallwellen Energie entzogen wird. Die durch die Schallwellen angeregten Rotations- und Schwingungsfreiheitsgrade der Moleküle führen zur molekularen Absorption. Hierbei wird den Schallwellen Translationsenergie entzogen (vgl. Lerch u. a., 2009, S. 35).

Des Weiteren wird die Dämpfung der Schallwellen vom Luftdruck beeinflusst. Hierbei nimmt mit steigendem Luftdruck die Absorption ab und die maximale Reichweite der Schallwellen zu (vgl. Hering und Schönfelder, 2012, S. 111).

Ein wichtiger Parameter für ultraschallbasierte Messverfahren ist die Wellenlänge λ (siehe Formel 2.8). Diese wird hierbei mithilfe der Schallgeschwindigkeit und der Frequenz f des

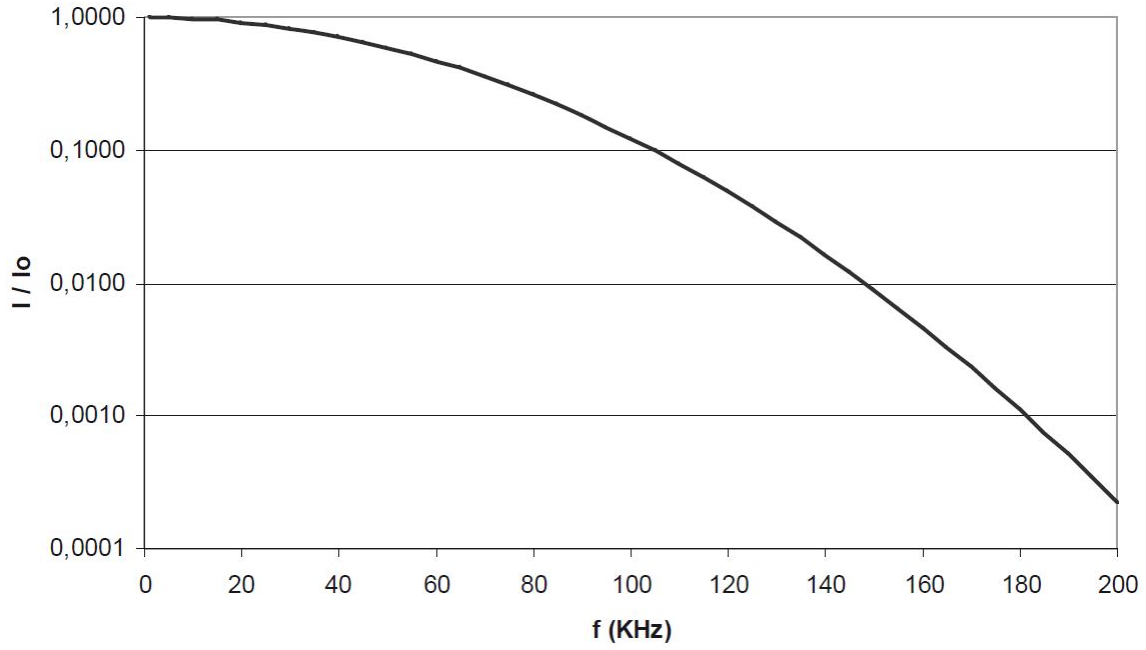


Abbildung 2.2.: Abhängigkeit der Schallintensität I von der Frequenz f (Hering und Schönfelder, 2012, S. 111)

Ultraschalls berechnet (vgl. Eichler, 2014, S. 164).

$$\lambda = \frac{c_{Luft}}{f} \tag{2.8}$$

In diesem Zusammenhang reichen die Anwendungsbereiche für Ultraschall von einfachen Distanzmessungen in Luft über komplexe Signalverarbeitungsprozesse auf Festkörperwellenbasis bis hin zu medizinischen Diagnostikverfahren. Aufgrund der hohen Dämpfung und sehr kleinen Wellenlänge geht hierbei der technisch nutzbare Bereich selten über eine Frequenz von 1 GHz hinaus (vgl. Lerch u. a., 2009, S. 573). Dabei stellt die Frequenz bei technischen Anwendungen einen Kompromiss zwischen Ortsauflösung und Dämpfung dar. Bei höheren Frequenzen ist die Ortsauflösung bedingt durch die kürzeren Wellenlängen höher. Gleichzeitig steigt hierbei aber auch die Dämpfung, wodurch die Reichweite sinkt (vgl. Lerch u. a., 2009, S. 575).

Gegenüber anderen Sensoren, wie zum Beispiel optischen Abstandssensoren, weisen Ultraschallsensoren den Nachteil der Richtcharakteristik beim Senden und Empfang der Schallwellen auf (siehe Abbildung 2.3). Hierdurch kann der ausgesendete Ultraschallpuls so gebündelt und gerichtet werden, dass dieser nicht zum Ultraschallsensor zurückkehrt (vgl. Lerch u. a., 2009, S. 575).

Eine weitere Eigenschaft von Ultraschallsensoren ist der Dopplereffekt. Hierbei wird durch

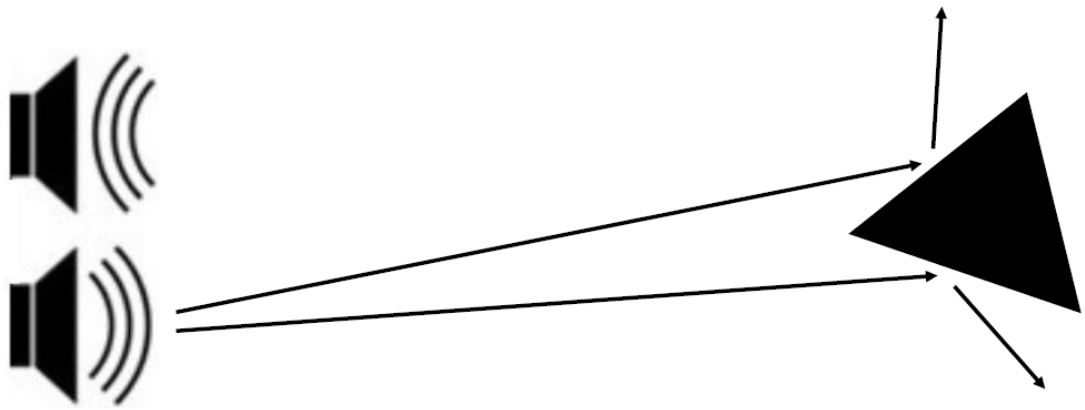


Abbildung 2.3.: Gerichtete Reflexion von Ultraschall an einem Objekt

sich bewegende Objekte die Frequenz der reflektierten Schallwellen verändert. Hierdurch kann es dazu kommen, dass die Frequenz der reflektierten Schallwellen außerhalb der Bandbreite des Ultraschallsensors liegt und diese nicht als Ultraschallecho erkannt werden. Im folgenden Beispiel wird die Frequenz der empfangenen Ultraschallechos berechnet (siehe Formel 2.9). Dabei beträgt die Sendefrequenz $f_S = 42\,kHz$, die Geschwindigkeit des Objektes $v_0 = 120\frac{km}{h}$ und die Umgebungstemperatur $20\,^\circ C$. Es ist zu erkennen, dass die empfangene Frequenz mit $f_E = 46{,}08\,kHz$ deutlich von der gesendeten Frequenz abweicht (vgl. Lerch u. a., 2009, S. 583).

$$f_E = f_S \left(1 + \frac{v_0}{c_{Luft}}\right) = 46,08\,kHz \tag{2.9}$$

2.1.2. Ultraschallsensoren

Ultraschallsensoren finden unter anderem Anwendung in Luft, Wasser, biologischem Gewebe oder Festkörpermaterialien.

Eine typische Anwendung in Luft ist dabei die Abstands- und Geschwindigkeitsmessung. Bei der Ermittlung des Abstands wird dabei die Laufzeit gemessen, die der ausgesendete Ultraschallpuls benötigt, um zum Sensor zurückzugelangen. Zur Ermittlung der Geschwindigkeit wird hingegen die Dopplerverschiebung gemessen, die das reflektierte Ultraschallecho aufweist. Ein für Ultraschallsensoren typischer Frequenzbereich liegt hierbei zwischen $25\,kHz$ und $500\,kHz$. Dabei ist ein Vorteil von Ultraschallsensoren im Vergleich zu optischen Sensoren, dass diese auch unter optisch nicht transparenten Umgebungsbedingungen ihre Anwendung finden und unempfindlich gegenüber Schmutz sind (vgl. Lerch u. a., 2009, S. 581).

Ein weiterer Vorteil von Ultraschallsensoren ist, dass diese im Vergleich zu anderen Abstandssensoren, kostengünstig und leicht realisiert werden können und einen geringen Energieverbrauch aufweisen. Teilweise sind Ultraschallsensoren in gewissen Anwendungsfällen, zum Beispiel Unterwasser oder bei geringer Sicht, die einzig realisierbaren Abstandssensoren (vgl. Kleeman und Kuc, 2016, S. 754).

Andere Ultraschallanwendungen befinden sich im Bereich der Messverfahren für biologische Gewebe. Dabei steht im Vordergrund, mithilfe der Ultraschallwellen innere Ortsinformationen über die zumessenden Objekte zu gewinnen, wie zum Beispiel bei der 3D-bildgebenden medizinischen Diagnostik. Die hierbei verwendeten Frequenzen liegen im Bereich zwischen 2,5 MHz und 15 MHz. Im Vergleich zu Gasen liegt dabei die Ausbreitungsgeschwindigkeit mit 1500 $\frac{m}{s}$ deutlich höher. Durch die höhere Ausbreitungsgeschwindigkeit erhöht sich die Wellenlänge proportional (siehe Formel 2.8). Da hierbei allerdings deutlich höhere Frequenzen verwendet werden, ist die Wellenlänge deutlich niedriger als bei Luftanwendungen. So beträgt bei 20° Umgebungstemperatur und einer Frequenz von 40 kHz die Wellenlänge 8,59 mm. Bei Anwendungen in biologischen Geweben beträgt die Wellenlänge bei einer Frequenz von 2,5 MHz hingegen 0,6 mm. Hierdurch wird eine deutlich höhere Auflösung realisiert (vgl. Lerch u. a., 2009, S. 587).

Zum Anwendungsbereich von Ultraschall gehören ebenfalls Unterwasseranwendungen wie das Echolotverfahren. Dabei werden zum Beispiel Tiefenbestimmungen zum Meeresboden durchgeführt. Des Weiteren können mithilfe der Echolotung Objekte geortet werden, die sich in einer vorgegebenen Richtung befinden, zum Beispiel senkrecht unter einem Schiff (vgl. Lerch u. a., 2009, S. 550). Die typische Schallgeschwindigkeit in Wasser bei einer Temperatur von 20°C beträgt 1484 $\frac{m}{s}$ (vgl. Lerch u. a., 2009, S. 540). Die hierbei verwendeten Frequenzen liegen für Tiefwasser bis 11 km bei bis zu 15 kHz. Für Anwendungen bis zu einer Tiefe von 3 km liegen die Frequenzen im Bereich von 30 kHz bis 100 kHz. In Flachwasser unter 300 m Tiefe werden hingegen Frequenzen über 200 kHz verwendet (vgl. Lerch u. a., 2009, S. 551).

Ein weiterer Anwendungsbereich liegt bei Messverfahren in Festkörpermaterialien, wie zum Beispiel die zerstörungsfreie Werkstoffprüfung, bei der Fehler im Materialinneren erkannt und visualisiert werden (vgl. Lerch u. a., 2009, S. 660).

2.1.2.1. Abstandsmessung

Die Abstandsmessung mithilfe von Ultraschallsensoren erfolgt zunächst mit dem Aussenden von einem Ultraschallpuls. Die Zeit, die der Ultraschallwandler, der die Ultraschallwellen erzeugt, zum Ausschwingen benötigt, bestimmt die Blindzone, in der keine Objekte erkannt

werden können, da hierbei die zurückkommenden reflektierten Echos nicht von dem Ausschwingen unterschieden werden können. Werden Ultraschallwellen von einem Objekt zurück zum Sensor reflektiert, versetzten diese den Ultraschallwandler in Schwingung, der hierdurch eine elektrische Spannung erzeugt. Überschreitet die Amplitude der elektrischen Spannung dabei einen festgelegten Schwellwert, wird das Signal als gültiges Ultraschallecho gewertet. Um die Distanz d zwischen Ultraschallsensor, als Sender und Empfänger des Ultraschalls, und detektierten Objekt zu ermitteln, wird die Schallgeschwindigkeit c_{Luft} und die Laufzeit t der Ultraschallwellen benötigt (vgl. Reif, 2014, S. 325):

$$d = \frac{c_{Luft}t}{2} \tag{2.10}$$

2.1.2.2. Messbereich

Die für den Messbereich von Ultraschallsensoren beeinflussenden Faktoren sind unter anderem die Sendefrequenz, Schallamplitude und Messempfindlichkeit. Mit steigender Frequenz sinkt gleichzeitig die maximale Reichweite. Der Vorteil von höheren Frequenzen ist, dass diese eine kürzere Zykluszeit zulassen (vgl. Hering und Schönfelder, 2012, S. 180).

Schallkeulen, an deren Grenze der Schalldruck des Sendepulses eine Dämpfung von 3 dB aufweist, haben dabei typischerweise einen Öffnungswinkel von 5° bis 8° (siehe Abbildung 2.4). Kleine Objekte mit einer ungeeigneten Form, wie zum Beispiel runde Objekte, können außerhalb der Schallkeule schwierig erkannt werden. Wohingegen große Objekte, die zum Beispiel eine flache Oberfläche aufweisen, auch außerhalb der Schallkeule detektiert werden können (vgl. Hering und Schönfelder, 2012, S. 179).

2.1.2.3. Einflüsse auf Messergebnisse und Reichweite

Die Eigenschaften von Objekten, die sich innerhalb des Messbereichs eines Ultraschallsensors befinden, beziehungsweise die Umgebungsbedingungen können teilweise sehr stark das Messergebnis und die Reichweite beeinflussen.

Zu den Objekteigenschaften, die die Ergebnisse und die Reichweite beeinflussen können, zählt zum Beispiel die Form. Bei runden Formen werden die Ultraschallwellen in unterschiedliche Richtungen reflektiert, wodurch nur ein geringer Teil zurück zum Ultraschallsensor gelangt (siehe Abbildung 2.5a). Hierbei nimmt dieser Effekt mit kleiner werdenden Radius zu (Hering und Schönfelder, 2012, S. 181).

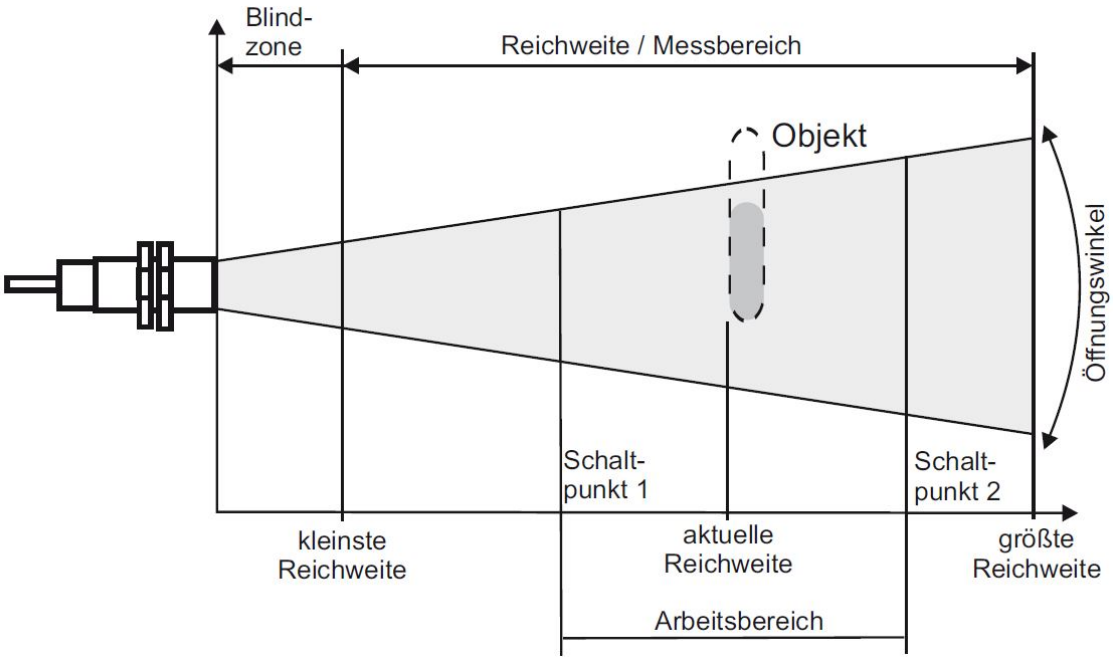


Abbildung 2.4.: Messbreich eines Ultraschallsensors (Hering und Schönfelder, 2012, S. 180)

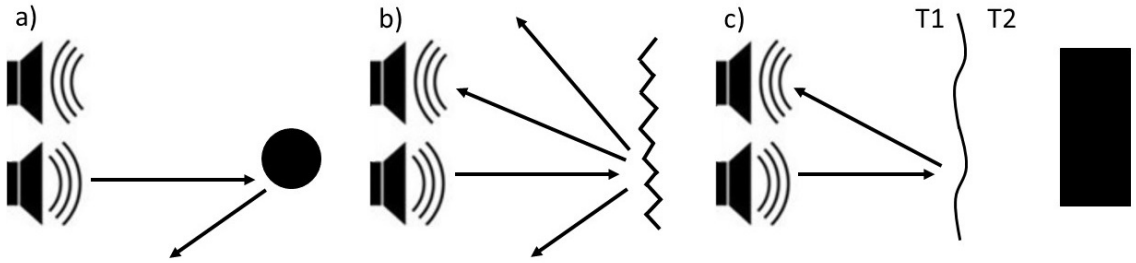


Abbildung 2.5.: Beispielhafte Beeinflussungen von Ultraschallmessungen

Des Weiteren kann die Materialhärte des Objekts den Anteil der reflektierten Schallwellen erheblich beeinflussen. Hierbei nimmt mit abnehmender Materialhärte die Absorption der Schallwellen zu (Hering und Schönfelder, 2012, S. 182).

Eine weitere wichtige Objekteigenschaft ist die Beschaffenheit der Oberfläche. So können bei rauen Oberflächen die Ultraschallwellen diffus reflektiert werden (siehe Abbildung 2.5b). Da in diesem Fall nicht alle Schallwellen in Richtung des Ultraschallsensors reflektiert werden, wird eine Erkennung des entsprechenden Objekts erschwert (Hering und Schönfelder, 2012, S. 182).

Die Objekttemperatur kann hierbei ebenfalls Beeinflussungen ausüben. Wärmekonvektionen können dazu führen, dass Ultraschallwellen abgelenkt beziehungsweise abgeschwächt werden, sodass das Objekt nicht erkannt wird. Des Weiteren kann es dazu kommen, dass die Schallwellen, wie in Abbildung 2.5c zu sehen, an der Grenzfläche der unterschiedlich temperierten Luftmassen reflektiert werden und nicht an dem Objekt (Hering und Schönfelder, 2012, S. 182).

Zu den Umgebungsbedingungen, die einen hohen Einfluss ausüben können, zählen starke Luftbewegungen, bei denen die Messergebnisse instabil werden können. Dies kommt allerdings erst ab Strömungsgeschwindigkeiten von einigen $\frac{m}{s}$ zu tragen (Hering und Schönfelder, 2012, S. 182).

2.1.3. Positionsbestimmung und Mapping

Für die Bereiche der Positionsbestimmung und des Mappings mithilfe von Ultraschall existieren unterschiedliche Methoden und Ansätze, auf die im Folgenden eingegangen wird. Ein weiterer Punkt, auf den im Folgenden ebenfalls eingegangen wird, sind Simulationsmethoden, mit denen ultraschallbasierte Positionsbestimmungen simuliert werden können. Hierzu ist in Tabelle 2.1 eine Übersicht der einzelnen Projekte der folgenden Abschnitte zu sehen.

Abschnitt	**Projekt**
Positionsbestimmung	Dynamisches, hybrides Ultraschall-Lokalisierungssystem (vgl. Seong und Byung, 2013)
	48-kanaliger Ultraschallring zur kontinuierlichen Echoverarbeitung (vgl. Browne und Kleeman, 2009)
Mapping	Multi-Ultraschallsensorsystem zur Echolokalisierung (vgl. Wu u. a., 2016)
	Generierung von 2D-Indoor-Karten mithilfe von Ultraschall (vgl. Rodin und Stajduhar, 2017)
	Indoor-Mapping für mobile Roboter mit einer Ultraschallsensorbank (vgl. Ilias u. a., 2016)
Raumakustik-Simulator für Ultraschall	Raumakustik-Simulator für Ultraschall (vgl. Albuquerque u. a., 2008), (vgl. Albuquerque, 2013)

Tabelle 2.1.: Übersicht der Projekte der folgenden Abschnitte

2.1.3.1. Positionsbestimmung

Bei der ultraschallbasierten Positionsbestimmung bestehen unterschiedliche Ansätze. Ein Ansatz ist, dass Ultraschallsender und -empfänger auf unterschiedliche Systeme aufgeteilt werden. Das heißt, dass die Ultraschallsender fest in einem Gebiet verbaut werden und die Ultraschallempfänger zum Beispiel auf einem mobilen Roboter platziert werden. Ein weiterer Ansatz ist, dass Ultraschallsender und -empfänger zusammen auf einem mobilen Roboter platziert werden.

Dynamisches, hybrides Ultraschall-Lokalisierungssystem

Bei dem dynamischen, hybriden Ultraschall-Lokalisierungssystem handelt es sich um ein Navigationssystem für autonome, mobile Indoor-Roboter, bei denen mehrere Ultraschall-distanzmessungen und ein extended Kalman-Filter (EKF) verwendet werden (vgl. Seong und Byung, 2013, S. 4562).

Hierbei werden Ultraschallsender an definierten Positionen der Raumdecke platziert (siehe Abbildung 2.6). Des Weiteren befinden sich drei Ultraschallempfänger auf dem mobilen Roboter. Unter Verwendung von Odometrie und den Ultraschallentfernungsmessungen wird mithilfe eines EKF-basierten Algorithmus die Position und die Orientierung bestimmt. Dabei wird eine dynamische Entfernungsschätzung angewendet, um die relevanten Signale von den Ultraschallsendern zu verfolgen und irrelevante auszublenden. So kann eine hybride Selbstlokalisierung ermöglichen werden (vgl. Seong und Byung, 2013, S. 4563).

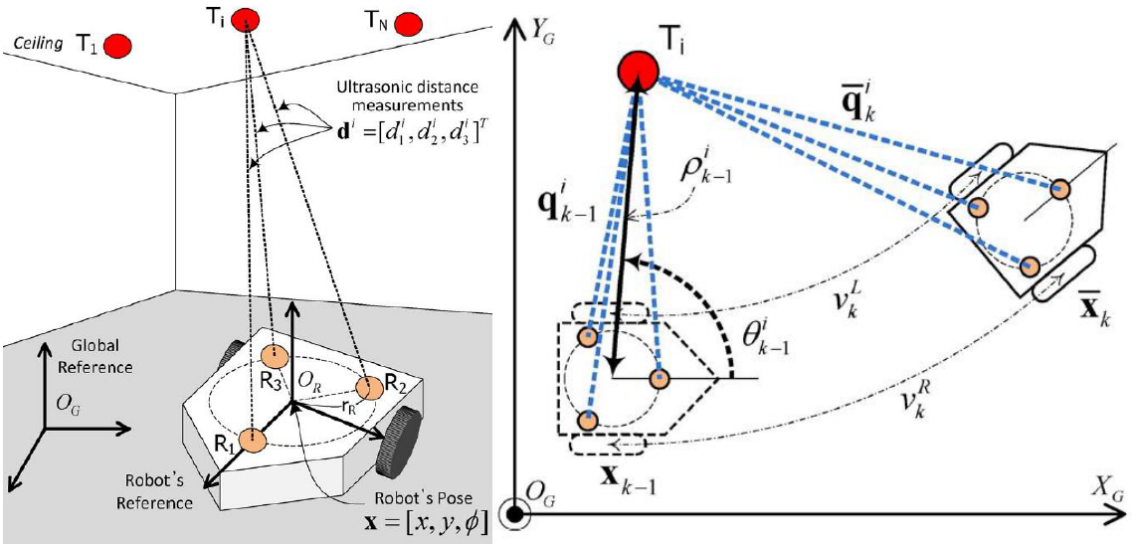


Abbildung 2.6.: Dynamisches, hybrides Ultraschall-Lokalisierungssystem (Seong und Byung, 2013, S. 4568)

Um die Abstände zwischen den Ultraschallempfängern des mobilen Roboters und den fest verbauten Ultraschallsendern zu ermitteln, werden die einzelnen Ultraschallsender durch eine Befehlsnachricht vom mobilen Roboter aktiviert. Dabei findet die Kommunikation für diese Befehlsnachrichten über RF-Module statt. Mithilfe des zeitlichen Abstands der Antwort des jeweiligen Ultraschallsenders über das RF-Modul und dem ausgesendeten Ultraschallpuls kann der mobile Roboter dem Abstand zu dem Ultraschallsender berechnen (vgl. Seong und Byung, 2013, S. 4564).

Die Vorteile eines solchen Systems sind, dass durch die bekannte Position der fest verbauten Ultraschallsender eine Positionsbestimmung mithilfe einfacher geometrischer Berechnungen möglich ist. Ein Nachteil des Systems ist hierbei allerdings, dass der mobile Roboter die fest verbauten Ultraschallsender benötigt und somit nicht flexibel im Bezug auf neue Einsatzorte ist.

48-kanaliger Ultraschallring zur kontinuierlichen Echoverarbeitung

Der 48-kanalige Ultraschallring ist ein echtzeitfähiges System, dass eine direkte Filterung auf der Systemhardware ermöglicht (siehe Abbildung 2.7a). Hierdurch wird, mit Anwendungsfokus auf mobilen Robotern, eine verringerte Latenz der Messungen realisiert. Der Ultraschallring wurde mit einem einzigen Ultraschallsender umgesetzt, der den Sendeimpuls in alle Richtungen gleichmäßig und gleichzeitig aussendet. Die 48 Ultraschallempfänger sind dabei auf zwei übereinanderliegenden Ebenen so angeordnet, dass eine 360°-Abdeckung ermöglicht wird (vgl. Browne und Kleeman, 2009, S. 4040).

Die Signalverarbeitung erfolgt dabei auf einem FPGA, auf dem zunächst die einzelnen Signale der Ultraschallempfänger gefiltert werden (siehe Abbildung 2.7b). Anschließend werden alle Signale in einem Multiplexer vereint. Abschließend zu einem Messdurchlauf werden in den Ultraschallechos einzelne Peaks ermittelt (vgl. Browne und Kleeman, 2009, S. 4043).

Wird ein zurückkommender Ultraschallpuls erkannt, werden die Daten in der Nachverarbeitung bearbeitet. Hierbei wird zunächst die Laufzeit des Ultraschallpulses berechnet. Im zweiten Schritt wird die Peilung des Ultraschallpulses ermittelt. Abschließend werden diese validiert, um redundante beziehungsweise unzuverlässige Ultraschallpulse zu entfernen (vgl. Browne und Kleeman, 2009, S. 4044).

Das 48-kanalige Ultraschallringsystem dient hierbei als Grundlage für Lokalisierungs- und Mappingprobleme (vgl. Browne und Kleeman, 2009, S. 4046). Dabei ist der Vorteil des Systems, das für jede Messung nur ein Ultraschallpuls ausgesendet wird und dieser von allen Ultraschallempfängern empfangen werden kann. Der Nachteil des Systems ist der relativ große und komplexe Hardwareaufbau.

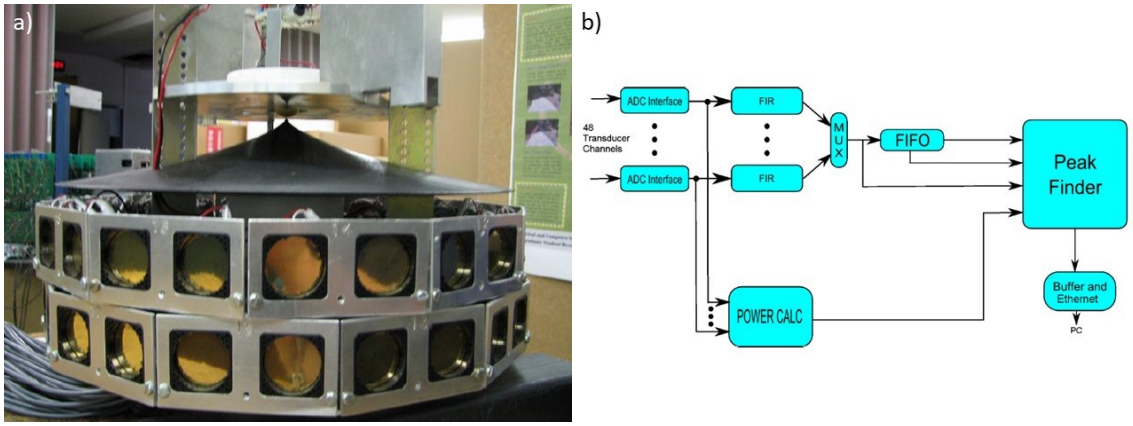


Abbildung 2.7.: a) 48-kanaliger Ultraschallring und b) FPGA-Signalverarbeitungspfad (Browne und Kleeman, 2009, S. 4043)

2.1.3.2. Mapping

Für das Mapping mithilfe von Ultraschallsensoren bestehen ebenfalls unterschiedliche Ansätze. Diese unterscheiden sich zum Großteil in der Anzahl der verwendeten Ultraschallsensoren. Hierbei werden mit einer Anzahl von einem Sensor bis hinzu 16 Sensoren Wände und Objekte kartiert.

Multi-Ultraschallsensorsystem zur Echolokalisierung

Das Projekt MUSSE (Multi-Ultrasonic-Sensor System for Echolocation) beschreibt die Umsetzung eines Multi-Ultraschallsensorsystems für das simultane Lokalisieren und Kartieren (SLAM) eines Open Source Roboters. Hierbei wird das System aufbauend auf dem Robot Operating System (ROS) implementiert. Das System basiert auf acht Ultraschallsensoren, die von einem zentralen Raspberry Pi gesteuert werden (siehe Abbildung 2.8). Hierbei kann das Sensorsystem auf einem Multiagentensystem, bestehend aus drei identischen Robotern von denen einer als Master operiert, angewendet werden (vgl. Wu u. a., 2016, S. 79).

Um Objekte zu erkennen, werden die Distanzen der ersten zurückkommenden Echos berechnet. Um hierbei eine gute Objekterkennung zu realisieren, werden die einzelnen Objekte mehrfach gemessen. Des Weiteren wird der Roboter nach den jeweiligen Messungen gedreht, um die einzelnen Messpunkte zu einem Objekt zu verbinden. Zur Messung werden hierbei die acht Ultraschallsensoren nacheinander einzeln angesteuert und ausgewertet (vgl. Wu u. a., 2016, S. 80).

Die eigentliche Kartierung erfolgt durch mehrere unabhängige Roboter. Dies reduziert zum einen die Zeit des Kartierens. Des Weiteren wird hierdurch der Einfluss von einzelnen Fehl-

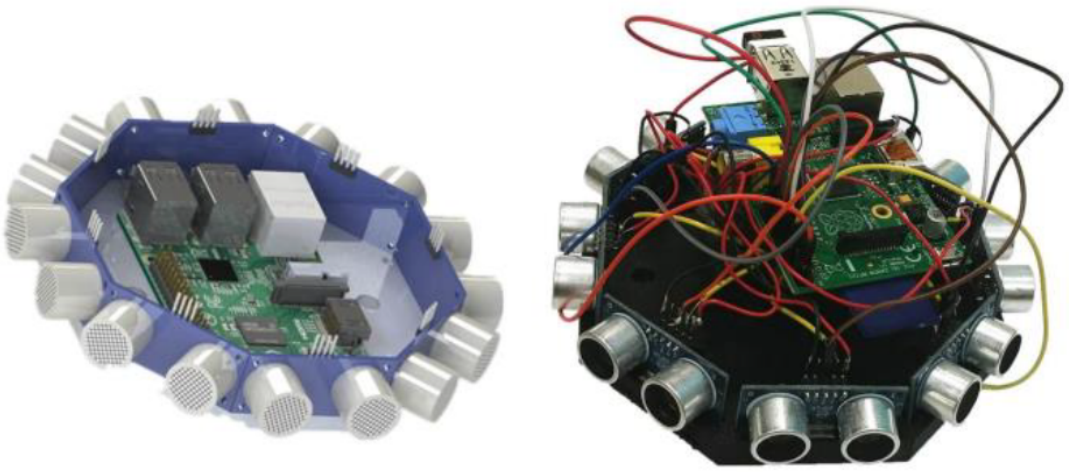


Abbildung 2.8.: Multi-Ultraschallsensorsystem zur Echolokalisierung (Wu u. a., 2016, S. 80)

messungen der Roboter auf das Gesamtergebnis reduziert. Das Mapping erfolgt hierbei durch eine gitterbasierte Karte (vgl. Wu u. a., 2016, S. 82).

Generierung von 2D-Indoor-Karten mithilfe von Ultraschall

Die Generierung von 2D-Indoor-Karten erfolgt mithilfe eines Ultraschallsensors, der auf einem autonom fahrenden Roboter montiert ist. Hierbei wird eine Methode verwendet, mit der unerwünschte Messergebnisse ausgefiltert werden. Bei den unerwünschten Messergebnissen handelt es sich um Präzisionsabnahme oder Ultraschallechos von verdeckten Objekten, die sich hinter einer Ecke verbergen (vgl. Rodin und Stajduhar, 2017, S. 1021).

Für das Mapping werden einfache Geometrien und eine Cluster-Analyse verwendet. Dabei dreht sich der Roboter im Kreis und misst die Abstände des ersten zurückkommenden Ultraschallechos, was in dem Fall eine Wand widerspiegelt. Um in den Messergebnissen die Wände zu erkennen und unerwünschte Messergebnisse zu filtern, wird ein K-Mean-Algorithmus als Cluster-Analyse verwendet. Die Filterung von relevanten Clustern, die Wände darstellen, und Clustern mit unerwünschten Messergebnissen wird anhand der Clustergröße durchgeführt. Dabei werden kleine Clustergrößen aus den Ergebnissen entfernt (vgl. Rodin und Stajduhar, 2017, S. 1023).

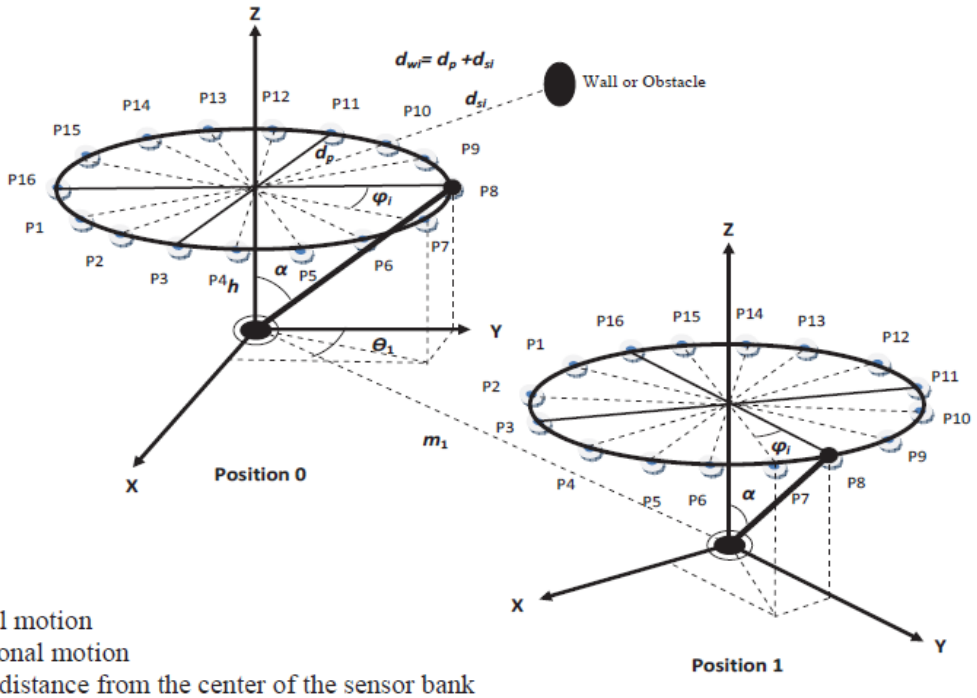
Indoor-Mapping für mobile Roboter mit einer Ultraschallsensorbank

Bei der Ultraschallbank für das Indoor-Mapping mobiler Roboter handelt es sich um ein System, bei dem 16 Ultraschallsensoren in einer sechseckigen Form angeordnet sind und das Umfeld zu 360° abdecken. Hierbei soll ein 2D- und 3D-Mapping in Echtzeit realisiert werden.

Das Ziel der Ultraschallbank ist dabei die genaue Kartierung von Wänden in Bezug auf deren Form und Umfang (vgl. Ilias u. a., 2016, S. 189).

Für die Ultraschallbank werden 16 Ultraschallsensoren, die jeweils um 22,5° versetzt sind, verwendet. Hierdurch sollen die Auswirkung von toten Winkeln bei den einzelnen Ultraschallsensoren reduzieren werden (vgl. Ilias u. a., 2016, S. 190).

Jeder Sensor bestimmt den jeweiligen Abstand zu einer Wand eigenständig. Anschließend werden mithilfe einer Koordinatentransformation die Abstände in eine Karte eingetragen. Nach jedem Distanzmessungszyklus dreht sich der mobile Roboter um einen definierten Winkel, um den Vorgang zu wiederholen (siehe Abbildung 2.9). Ebenfalls nach jedem Distanzmessungszyklus fährt der mobile Roboter auf eine andere Position, wobei nur soweit gefahren wird, dass keine Kollision mit einer Wand entsteht (vgl. Ilias u. a., 2016, S. 191).



θ - rotational motion
m - translational motion
d_p - sensors distance from the center of the sensor bank
φ_i - angular position of the sensor with respect to Z-axis (center of sensor bank)
α - sensor position with respect to the origin
h - center of sensor bank above from robot base along its Z-axis
d_{si} - distance between obstacle and sensor position
d_{wi} - distance between the center of sensor bank and obstacle

Abbildung 2.9.: Positionsveränderung der Ultraschallbank (Ilias u. a., 2016, S. 190)

Um aus der Kartierung Phantom-Punkte zu entfernen, wird im Anschluss ein Optimierungsalgorithmus angewandt. Bei diesem werden die empfangenen Echos der jeweiligen Ultra-

schallsensoren mit denen der Nachbarsensoren verglichen. Somit werden Echos, die zum Beispiel durch Mehrfachreflexion entstehen, ausgefiltert (vgl. Ilias u. a., 2016, S. 191).

Mit der Ultraschallbank können somit einfache geometrische Wandformen gemessen und kartiert werden (vgl. Ilias u. a., 2016, S. 193).

2.1.3.3. Raumakustik-Simulator für Ultraschall

Der Raumakustik-Simulator dient dem Vergleich verschiedener Ultraschall-Lokalisierungs-Algorithmen in kontrollierten Umgebungen. Hierbei wird die Annahme getroffen, dass eine Simulation deutlich schneller umgesetzt werden kann als ein reales Mess-Set-up. Ein weiterer Vorteil ist, dass eine Simulation verschiedener Algorithmen deutlich einfacher unter denselben Bedingungen durchgeführt werden kann, als Tests mit realen Ultraschallsignalen (vgl. Albuquerque, 2013, S. 21).

Dabei basiert der Raumakustik-Simulator auf einer hybriden Methode, bei der die Reflexionen als spiegelnd betrachtet werden. Beeinflussungsfaktoren bei der Simulation sind Reflexionen an Wänden und Objekten, die Schalldämpfung und die Eigenschaften des Ultraschallsensors (vgl. Albuquerque u. a., 2008, S. 1).

Mit dem Simulator werden Mechfachreflexionen simuliert, sodass nicht nur das Ultraschallecho mit der kürzesten Laufzeit ermittelt wird, sondern auch Ultraschallechos, die über mehrere Objekte zum Ultraschallempfänger reflektiert werden. Dabei setzt sich der Simulator aus einem Schallausbreitungsmodell, einem Mehrfachreflexionsmodell, einem Ultraschallsendermodell und einem Modell für das Rauschen zusammen (vgl. Albuquerque u. a., 2008, S. 2).

Das Schallausbreitungsmodell beachtet dabei den Temperatureinfluss auf die Schallgeschwindigkeit, die Streuung der Schallenergie im Verhältnis zum zurückgelegten Weg und die Abnahme der Schallenergie durch Luftverluste (vgl. Albuquerque u. a., 2008, S. 2).

Bei dem Mehrfachreflexionsmodell wird eine hybride Methode aus Raytracing und der Bildquellen-Methode angewendet. Hierbei betrachtet die Raytracing-Methode die Ultraschallwellen als eine große Anzahl an Stahlen, die von dem Ultraschallsender emittiert werden. Die einzelnen Pfade der Strahlen werden dabei berechnet. Bei einer Berührung eines Objekts tritt eine Reflexion auf, von der aus ein neuer Pfad berechnet wird. Um zu erkennen, ob ein Ultraschallempfänger von den einzelnen Strahlen getroffen wird, wird um die einzelnen Empfänger ein Schnittvolumen definiert. Mit der Raytracing-Methode sind genaue Simulationsergebnisse sehr rechenintensiv. Die Bildquellen-Methode hingegen basiert auf

dem Prinzip, dass alle Reflexionen spiegelnd sind. Hierbei wird bei einer Reflexion die Quelle an der Oberfläche des Reflexionsobjekts gespiegelt (siehe Abbildung 2.10a). Das Verhalten der realen Quelle wird dabei auf die gespiegelte virtuelle Quelle übertragen. Dieses Spiegeln wird bei weiteren Reflexionen fortgeführt. Die Bildquellen-Methode ist sehr genau, wobei die Anzahl der gespiegelten Bilder mit der Anzahl der Reflexionen schnell ansteigt. Bei der hybriden Methode wird bei dem Mehrfachreflexionsmodell zunächst die Bildquellen-Methode angewandt, um alle virtuellen Ultraschallquellen zu ermitteln. Anschließend werden die virtuellen Quellen mit der Raytracing-Methode überprüft, wobei nur Strahlen berücksichtigt werden, die von dieser erzeugt werden. So können die Ultraschallstrahlen vom Empfänger zur realen Quelle zurückverfolgt werden. Dabei wird mithilfe der hybriden Methode die Anzahl der Empfangswellen am Ultraschallempfänger deutlich reduziert (vgl. Albuquerque u. a., 2008, S. 3).

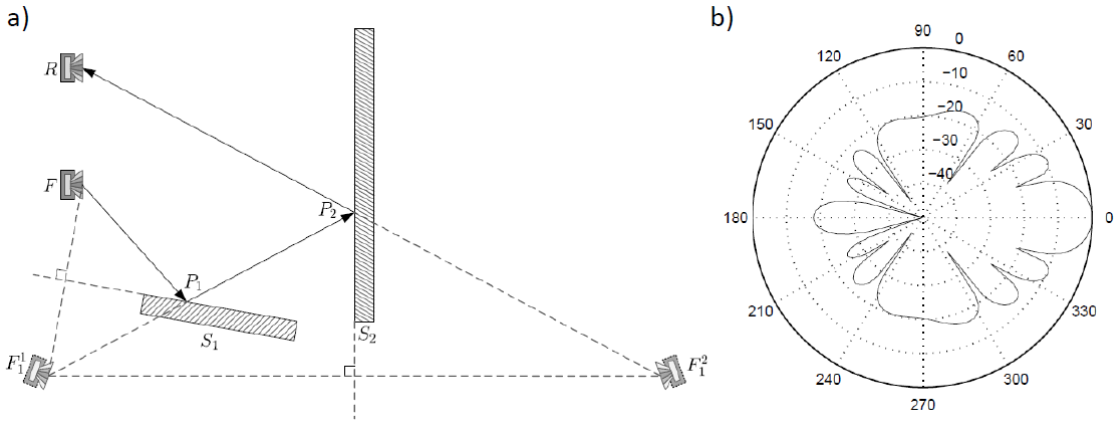


Abbildung 2.10.: a) Spiegelungen der Bildquellen-Methode (Albuquerque u. a., 2008, S. 3) und b) Abstrahlcharakteristik des Ultraschallsendermodells (Albuquerque u. a., 2008, S. 4)

Das Modell des Ultraschallsenders folgt der Annahme, dass dieser translatorische Bewegungen entlang einer Zylinderachse beim Aussenden von Ultraschallwellen durchführt. Die hierbei erzeugten Ultraschallwellen werden dabei im Bezug auf die Intensität nicht gleichmäßig in alle Richtungen ausgestrahlt. Zur Berechnung der Abstrahlcharakteristik, wie in Abbildung 2.10b zu sehen, wird eine Strahlfunktion verwendet (siehe Formel 2.11). In die Formel fließen dabei der Radius a des angenommenen Zylinders des Ultraschallsenders, die Wellenlänge λ und der Abstrahlwinkel θ ein. Des Weiteren wird innerhalb der Strahlfunktion eine Bessel-Funktion ersten Grades J_1 angewandt (vgl. Albuquerque u. a., 2008, S. 4).

$$H(a, \lambda, \theta) = 2\frac{J_1\left(\frac{2\pi a}{\lambda} sin\theta\right)}{\frac{2\pi a}{\lambda} sin\theta} \tag{2.11}$$

Abschließend wird mit einem Rausch-Modell auf die empfangenen Ultraschallechos ein weißes und gaußsches Rauschen modelliert. Die Stärke des Rauschens ist hierbei einstellbar (vgl. Albuquerque u. a., 2008, S. 4).

2.2. Vorarbeiten im Urban Mobility Lab

Zu den Vorarbeiten des *Urban Mobility Lab*, die für die Weiterarbeit im Rahmen dieser Masterarbeit genutzt werden, zählt unter anderem ein Entwicklungssystem für intelligente Ultraschallsensoren zur Lokalisation und Umgebungserkennung. Eine weitere Vorarbeit der Entwicklungen des *Urban Mobility Lab* ist eine 2D-Simulation für das Ultraschall-Raytracing. Abschließend wird der aktuelle Stand des Versuchsfahrzeugs beschrieben.

2.2.1. Entwicklungssystem für intelligente Ultraschallsensoren

Im Rahmen der Bachelorarbeit wurde ein Entwicklungssystem für intelligente Ultraschallsensoren konzipiert und umgesetzt. Mit dem System ist es möglich, Algorithmen und Methoden zur Lokalisation und Umgebungserkennung zu entwickeln (vgl. Rotzlawski, 2018, S. 3).

Das Entwicklungssystem besteht dabei aus mehreren Ultraschallsensoren, die beliebig angeordnet werden können (siehe Abbildung 2.11). Des Weiteren besteht das System aus einer Datenauswertung, die auf einem Rechner läuft, die die Ultraschallsensoren steuert und auswertet. Hierbei ist das System über ein Ethernetnetzwerk in einer Sterntopologie angeordnet (vgl. Rotzlawski, 2018, S. 35).

Zur Kommunikation zwischen der Datenauswertung und den Ultraschallsensoren wurden unterschiedliche Ethernetpakete definiert. Mit diesen Paketen kann man unter anderem den Status der Ultraschallsensoren abfragen. Des Weiteren wird den Sensoren über die Statusabfrage der Master- beziehungsweise Slavestatus mitgeteilt. Eine Messung der Umgebungsbedingungen kann ebenfalls angefordert werden. Die Ultraschallmessung wird über ein Anweisungspaket an den Master-Ultraschallsensor gestartet werden (vgl. Rotzlawski, 2018, S. 37).

Der Ultraschallsensor basiert auf einem *Arduino Due*. Für die Ethernetanbindung wurde das Board um ein Ethernet-Shield erweitert. Zur Messung der Umgebungsbedingungen besitzt jeder Ultraschallsensor einen entsprechenden Sensor. Für das Aussenden von Ultraschallpulsen besteht der Sendepfad aus einer Sendeverstärkerschaltung und einem Ultraschalllautsprecher. Die Ultraschallechos werden mit der Empfängerschaltung empfangen, welche

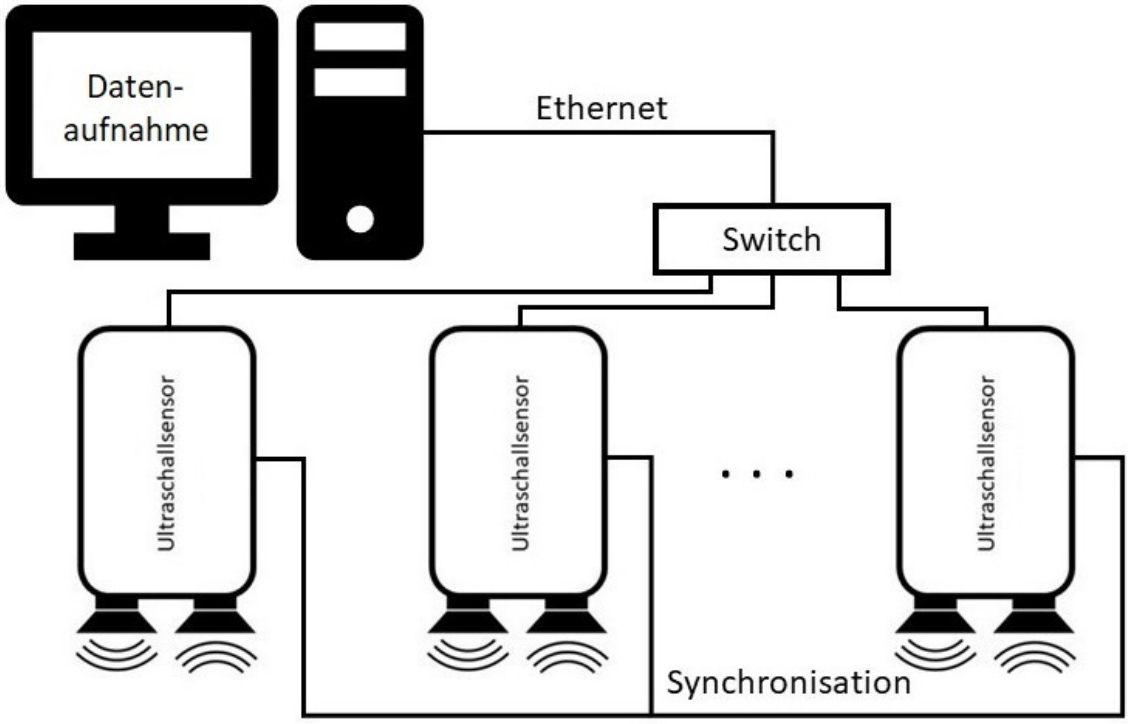


Abbildung 2.11.: Entwicklungssystem in Mehrfachanordnung (Rotzlawski, 2018, S. 36)

aus einem MEMS-Mikrofon und einer Verstärkerschaltung besteht. Die Spannungsversorgung des Ultraschallsensors erfolgt dabei über einen 5 V-Akku. Zur Synchronisation sind alle Ultraschallsensoren über eine Synchronisationsleitung verbunden (vgl. Rotzlawski, 2018, S. 39). Abbildung 2.12 zeigt die Architektur des Ultraschallsensors.

Der Messdurchlauf wird gestartet, indem der Mastersensor die Messung mithilfe der Synchronisationsleitung auf allen Sensoren gleichzeitig startet. Gleichzeitig zum Starten der Messung sendet der Mastersensor den Ultraschallpuls aus. Am Ende des Messdurchlaufs beendet der Mastersensor ebenfalls über die Synchronisationsleitung die Messung auf allen Sensoren. Der Ultraschallsendepuls kann hierbei variable in Bezug auf Frequenz und Dauer eingestellt werden (vgl. Rotzlawski, 2018, S. 49).

Im Vorfeld dieser Masterarbeit wurde eine Analyse zum Austausch der Sende- und Empfangspfade des Ultraschallsensors durchgeführt. Hierbei wurde der Ultraschallsensor mit dem Ultraschallmodul *MaxBotix XL-MaxSonar MB1300* verglichen (vgl. MaxBotix, 2015, S. 9). Da die empfangenen Echosignale des Ultraschallsensors deutlich verrauscht sind (vgl. Rotzlawski, 2018, S. 73), ist das Ergebnis des Vergleichs, dass mit dem *MaxBotix*-Modul eine höhere Reichweite der Ultraschallsensoren realisiert werden kann. Der Nachteil eines Austauschs ist hingegen, dass hierdurch der Sendepuls im Bezug auf Frequenz und Dauer nicht mehr einstellbar ist.

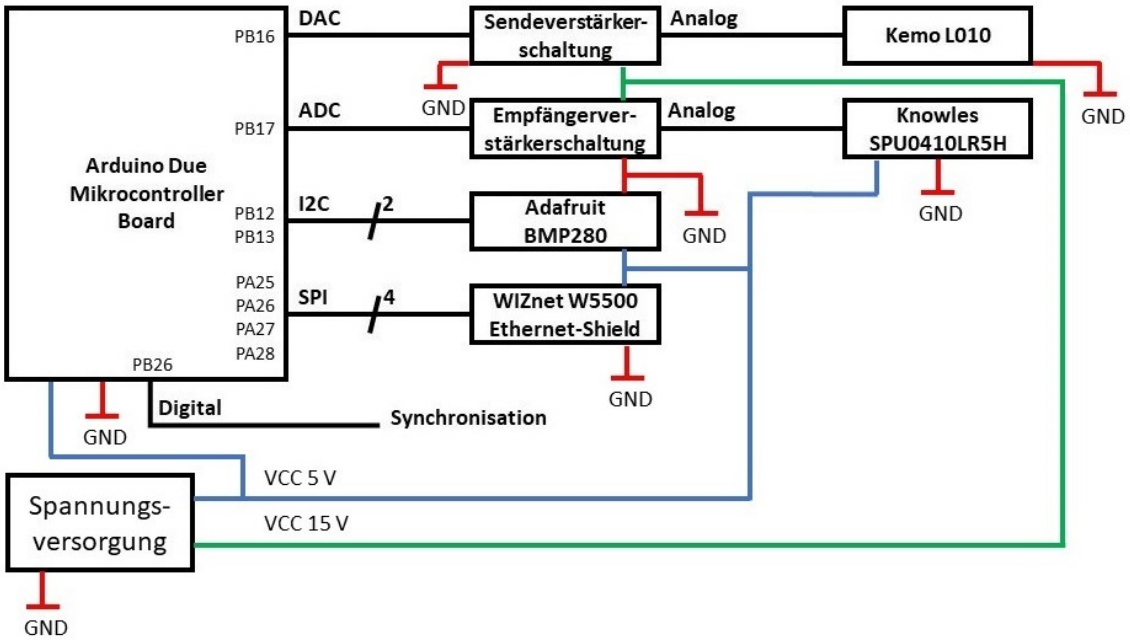


Abbildung 2.12.: Architektur des Ultraschallsensors (Rotzlawski, 2018, S. 39)

2.2.2. Simulation für das Ultraschall-Raytracing

Ein weiteres Projekt des *Urban Mobility Lab* ist eine 2D-Raytracing-Simulation für Ultraschall, die mit dem Raumakustik-Simulator aus Abschnitt 2.1.3.3 vergleichbar ist. Hierbei wurde mithilfe von *MATLAB* eine Simulation umgesetzt, mit der Ultraschallwellen in einzelne Strahlen aufgeteilt werden. Die einzelnen Pfade der Strahlen werden dabei berechnet. Trifft ein Strahl ein Objekt, wird dieser an dem Objekt reflektiert. Hiermit wird eine Simulation von Mehrfachreflexionen ermöglicht. Die Ergebnisse der Simulation können hierbei visualisiert werden.

In Abbildung 2.13 ist das Ergebnis der Pfade der einzelnen Strahlen zu sehen, wobei hier nur Strahlen geplottet werden, die einen Ultraschallsensor treffen. Hierbei wird ein schmaler Gang simuliert. Der rote beziehungsweise grüne Punkt stellen Ultraschallempfänger dar. Wohingegen der blaue Punkt einen Ultraschallsender und -empfänger darstellt. Die einzelnen Strahlen sind ebenfalls mithilfe der Farbe den einzelnen Ultraschallempfängern zuzuordnen.

Der dazugehörige zeitliche Verlauf der ankommenden Ultraschallechos ist in Abbildung 2.14 zu sehen. Die zeitlichen Verläufe der Ultraschallsensoren werden einzeln dargestellt, wobei *R1* den Ultraschallsender und -empfänger darstellt. *R2* und *R3* sind hingegen die Ultraschallempfänger. Zusätzlich wurden in Abbildung 2.14 die simulierten zeitlichen Verläufe der Ultraschallsensoren mit einer realen Messung des schmalen Gangs verglichen. Die einzelnen Peaks stellen die simulierten Ultraschallechos dar. Wohingegen die Hüllkurven die realen

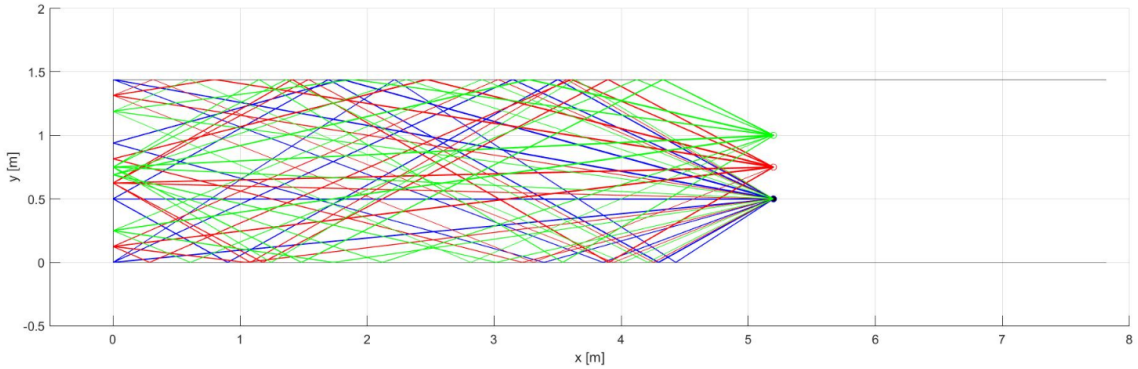


Abbildung 2.13.: Raytracing-Ergebins der Simulation

Messungen der Ultraschallsensoren darstellen. Hierbei wurde die reale Messung mit dem *MaxBotix XL-MaxSonar MB1300* als Ultraschallsensor durchgeführt (vgl. MaxBotix, 2015, S. 9). Die Ultraschallsensoren werden dabei identisch zur Abbildung 2.13 platziert.

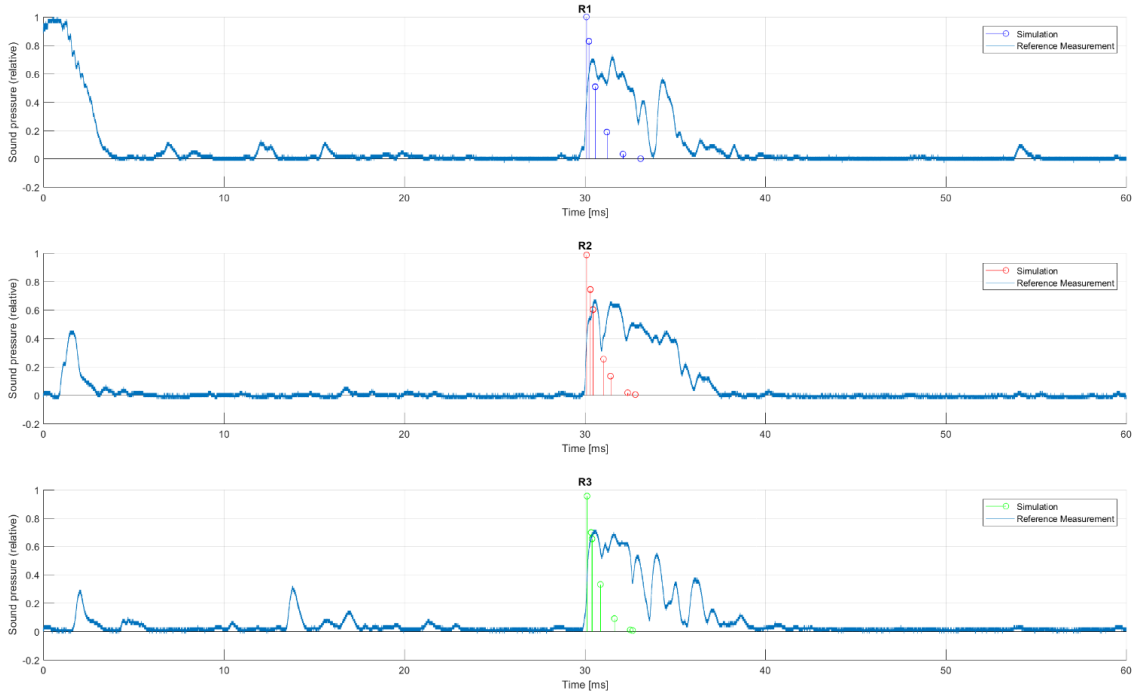


Abbildung 2.14.: Vergleich der 2D-Raytracing-Simulation mit einer realen Ultraschallmessung

Da es sich bei der Ultraschall-Raytracing-Simulation um eine zwei dimensionale Simulation handelt, werden Reflexionen, die zum Beispiel von Deckenlampen reflektiert werden, nicht berechnet. Des Weiteren werden hierdurch auch Mehrfachreflexionen über den Boden oder

der Decke, das heißt, die über drei Dimensionen verlaufen, nicht simuliert. Die dadurch entstehenden Abweichungen sind dabei deutlich zu erkennen.

2.2.3. Aktueller Stand des Versuchsfahrzeugs

Im Rahmen einer Bachelorarbeit im *Urban Mobility Lab* wurde das Versuchsfahrzeug mit einem Industrie-PC ausgestattet (vgl. Vater, 2018, S. 3). Des Weiteren ist das Versuchsfahrzeug mit einem Messaufbau auf dem Dach ausgestattet (siehe Abbildung 2.15). Abbildung 2.16 zeigt das Blockschaltbild des Messaufbaus mit den einzelnen Sensorsystemen.

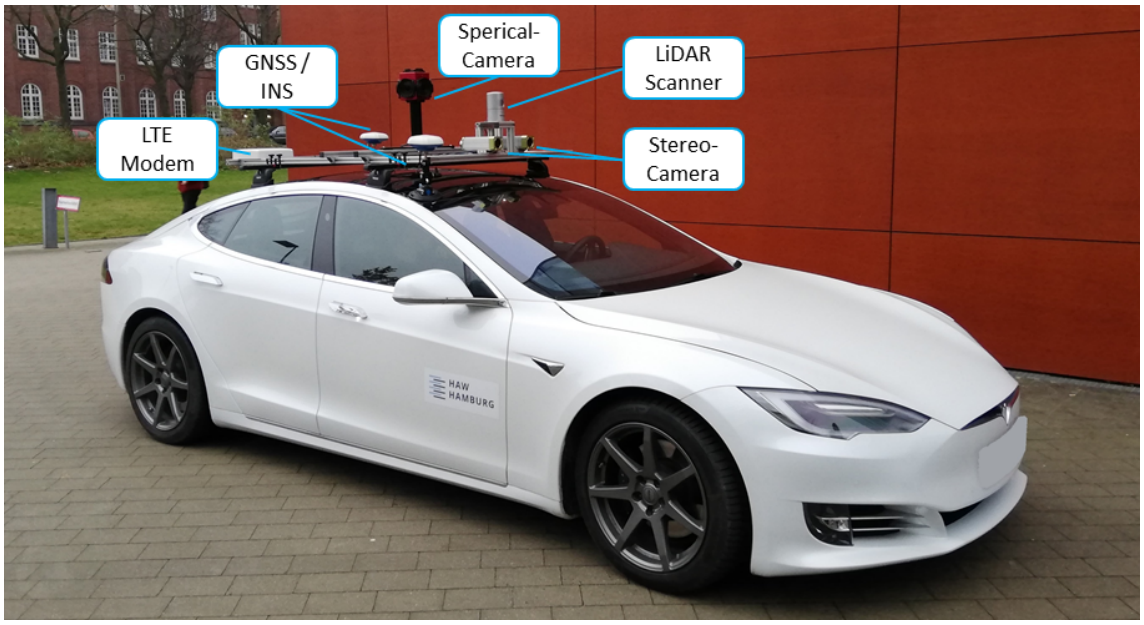


Abbildung 2.15.: Messaufbau auf dem Versuchsfahrzeug

Hierbei dient der Messaufbau zur Erforschung und Entwicklung von Systemen zur Fahrzeugpositionsbestimmung und Umgebungswahrnehmung. Mit dem GNSS/INS-System wird dabei eine hochpräzise Positionsbestimmung ermöglicht. Der LiDAR-Scanner hingegen dient zur Umgebungswahrnehmung, mit der ebenfalls eine Positionsbestimmung ermöglicht wird. Aktuelle Projekte beschäftigen sich mit der sphärischen und der Stereo-Kamera. Hierbei soll über Bilddaten der Umgebung eine Positionsbestimmung realisiert werden.

Mit einem weiteren Systembestandteil, dem LTE-Modem, ist es möglich, das Mobilfunknetz während der Fahrt zu vermessen. Mit einer nachgelagerten Datenauswertung wird eine messtechnische Beurteilung des Mobilfunknetzes realisiert.

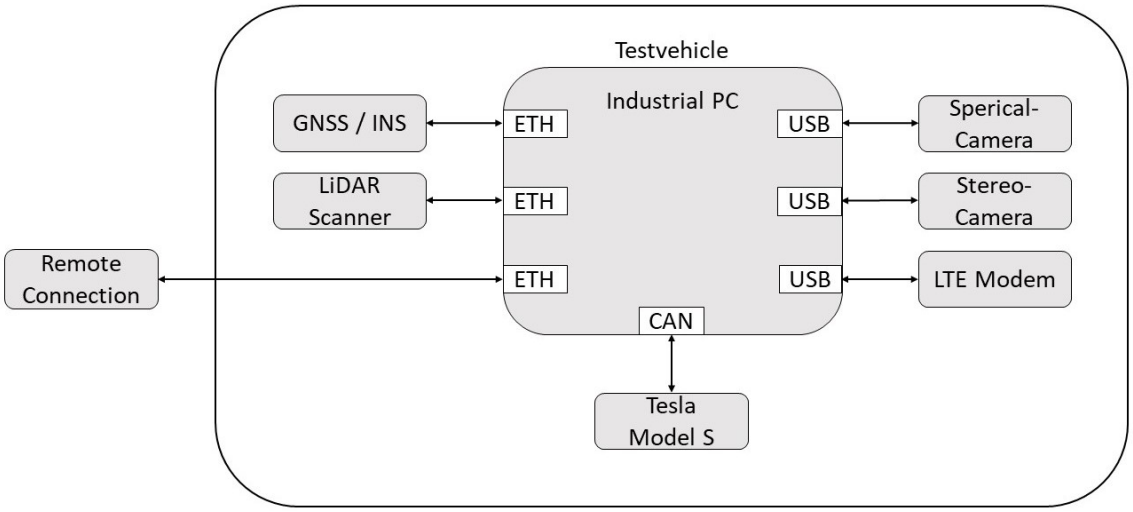


Abbildung 2.16.: Messsystemarchitektur des Versuchsfahrzeugs

Ebenfalls ein Bestandteil des Systems ist eine CAN-Schnittstelle. Hiermit kann die Kommunikation auf dem CAN-Bus des Fahrzeugs aufgenommen und analysiert werden. Abschließend besitzt der Messaufbau eine Remote-Schnittstelle, mit der der Industrie-PC gesteuert werden kann.

3. Analyse der Anforderungen

Die Analyse der Anforderungen befasst sich zunächst mit einem typischen Anwendungsfall für das Navigationssystem zur Positionserkennung autonomer Fahrzeuge. Abschließend werden die Anforderungen an das System analysiert und definiert.

3.1. Anwendungsfall zur Positionsbestimmung

Ein typischer Anwendungsfall des zu entwickelnden Systems befindet sich in Indoor-Bereichen, wie zum Beispiel Tiefgaragen oder Parkhäuser. Eine herkömmliche Positionsbestimmung mithilfe von GNSS-Satelliten ist in diesen Fällen nicht möglich, da hier die Satellitensignale über den *Multipath*-Fehler stark verfälscht werden beziehungsweise erst gar nicht am Fahrzeug ankommen. Eine Erkennung von verdeckten Objekten, die sich außerhalb des sichtbaren Bereichs befinden, ist mithilfe der regulären Anwendung von Umfeldsensorik in Fahrzeugen, wie zum Beispiel Kamerasysteme oder herkömmlichen Ultraschallsensoren, ebenfalls nicht möglich (siehe Abbildung 3.1). So werden verdeckte Objekte von Kamerasystemen generell nicht erkennbar. Eine Erkennung mithilfe von Ultraschallsensoren, bei der nur das erste zurückkommende Ultraschallecho ausgewertet wird, ist ebenfalls nicht möglich.

Mithilfe des umfeldsensorbasierten Navigationssystems soll eine Positionsbestimmung in einem im Vorfeld bekannten Indoor-Bereich ermöglicht werden. Dies soll mit der im *Urban Mobility Lab* vorhandenen Umfeldsensorik realisiert werden, wie in etwa dem *Velodyne HDL-32E* LiDAR-Scanner oder dem Ultraschallsensorsystem zur Entwicklung intelligenter Ultraschallsensoren aus Abschnitt 2.2.1. Ebenfalls soll mit dem umfeldsensorbasierten Navigationssystem eine Erkennung von verdeckten Objekten, die sich hinter einer Wand verbergen, ermöglicht werden.

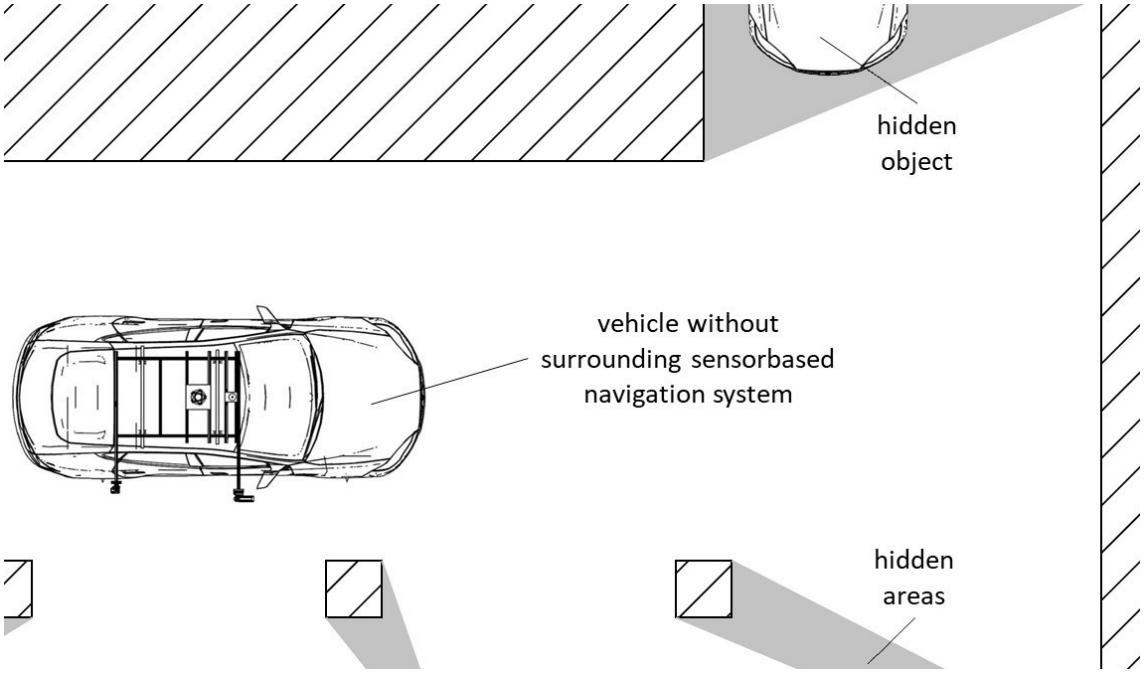


Abbildung 3.1.: Anwendungsfall mit einem Fahrzeug ohne umfeldsensorbasiertes Navigationssystem

3.2. Anforderungsanalyse

Die Anforderungen werden aufbauend auf der Aufgabenstellung und des Anwendungsfalls definiert. Hierbei werden die Anforderungen, wie in Tabelle 3.1 zu sehen, nach Möglichkeit mit einem Wert quantifiziert.

Um eine zeitweise, alternative Methode zur GNSS-basierten Navigation zu realisieren, ist es notwendig, mit dem System die Fahrzeugposition und -orientierung zu bestimmen. Hierbei muss das Navigationssystem auf mindestens ein Umfeldsensorsystem zurückgreifen. Dabei soll Ultraschall vorrangig als Umfeldsensorik verwendet werden. Des Weiteren soll eine Erkennung von Objekten ermöglicht werden, die sich nicht im direkten Sichtfeld befinden. Hierzu ist es notwendig, eine Erkennung der Umgebung und von Objekten zu realisieren. Um dies zu ermöglichen, wird die Detektionsreichweite der einzelnen Umfeldsensoriksysteme auf mindestens 5 m festgelegt.

Da sich in Indoor-Bereichen, wie zum Beispiel in Parkhäusern, Personen aufhalten, wird als maximale Geschwindigkeit 6 $\frac{km}{h}$ angenommen, was 1,67 $\frac{m}{s}$ entspricht. Dabei soll eine ausreichend hohe Messfrequenz erreicht werden. Hierbei soll spätestens nach einem gefahrenen Meter eine neue Position ermittelt werden. Hieraus folgt, dass die Messfrequenz des

Systems bei mindestens 1,67 Hz liegen soll (siehe Formel 3.1).

$$f_S = \frac{v_0}{\Delta x} = 1,67\,Hz \tag{3.1}$$

Um Kollisionen zu vermeiden, wird ein Sicherheitsabstand zu Objekten und Wänden von 0,5 m angenommen. Hierzu wird die Genauigkeit der Positionserkennung auf unter 0,05 m festgelegt, um bei einer Unterschreitung des Mindestabstands durch Ungenauigkeiten des Systems eine Kollision zu verhindern. Für die Orientierungserkennung hingegen wird eine Genauigkeit von unter 10° festgelegt.

Um die Position und Orientierung des Fahrzeugs bei der Verwendung von mindestens einem weiteren Umfeldsensorsystem zu bestimmen, muss diese in dem Fall durch eine Sensordatenfusion ermittelt werden. Hierbei soll das Navigationssystem modular um weitere Umfeldsensorsysteme erweiterbar sein. Um die Sicherheit des Fahrzeugs im Bezug auf die Positions- und Orientierungsbestimmung zu gewährleisten, müssen mögliche Fehler und Ausfälle der Teilsysteme erkannt werden. Hierbei soll das Navigationssystem weiterhin funktionsfähig sein.

Um mithilfe von Ultraschall eine Erkennung von Umgebung und Objekten eindeutig zu realisieren, soll dieses System aus mindestens zwei Sensoren bestehen. Diese sollen in Bezug auf Anordnung und Anzahl modular und skalierbar sein.

Da eine Integration des Navigationssystems in das Versuchsfahrzeug geplant ist, muss dies ermöglicht werden, ohne dass dadurch die Zulassung für den Straßenverkehr erlischt. Hierbei soll bei der Integration ein rückstandsfreier Ein- beziehungsweise Ausbau ermöglicht werden. Da bei dem Industrie-PC des Versuchsfahrzeugs als Betriebssystem *Ubuntu 16.04* zum Einsatz kommt, muss die Software des Navigationssystems auf diesem lauffähig sein. Hierbei soll das Navigationssystem in das Messsystem des Versuchsfahrzeugs integriert werden. Daraus folgt, dass die Implementierung des Navigationssystems unter Verwendung von ROS erfolgen soll. Die Spannungsversorgung des Systems soll dabei durch das Versuchsfahrzeug beziehungsweise dem Industrie-PC erfolgen. Hierzu muss das System auf 5 V beziehungsweise 12 V Spannungsversorgung ausgelegt werden.

Das Navigationssystem wird unter regengeschützten Bedingungen betrieben, da dieses für Indoor-Bereiche realisiert werden soll. Des Weiteren wird die Annahme getroffen, dass die Umgebungstemperatur in diesen Bereichen im Bereich von 0 °C bis 30 °C liegt.

Um die Positionsbestimmung des umfeldsensorbasierten Navigationssystems mit einer GNSS-basierten Positionsbestimmung vergleichbar zu machen, soll die Ausgabe der Position in Längen- und Breitengrad erfolgen.

ID	Beschreibung	Wert
1	Erkennung der Fahrzeugposition	
2	Erkennung der Fahrzeugorientierung	
3	Verwendung von Umfeldsensorsystemen	$n \geq 1$
4	Verwendung eines Ultraschallsensorsystems	
5	Erkennung von Objekten, die sich nicht im direkten Sichtfeld befinden	
6	Erkennung der Umgebung und von Objekten	
7	Möglichst hohe Detektionsreichweite der einzelnen Umfeldsensorsysteme	$x \geq 5\,m$
8	Ausreichend hohe Messfrequenz	$f \geq 1{,}67\,Hz$
9	Einsatzfähigkeit des Navigationssystems bei Schrittgeschwindigkeit	$v \leq 6\,\frac{km}{h}$
10	Möglichst hohe Genauigkeit der Positionserkennung	$\Delta x < 0{,}05\,m$
11	Möglichst hohe Genauigkeit der Orientierungserkennung	$\Delta\phi < 10^{\circ}$
12	Bei der Verwendung von mehreren Umfeldsensorsystemen Positions- und Orientierungsbestimmung durch Sensordatenfusion	
13	Navigationssystem modular erweiterbar um weitere Umfeldsensorsysteme	
14	Funktionsfähigkeit des Navigationssystems bei Ausfall von Teilsystemen	
15	Verwendung von mindestens zwei Ultraschallsensoren	$n \geq 2$
16	Modulare und skalierbare Anordnung der Ultraschallsensoren	
17	Integration in das Versuchsfahrzeug ohne Erlöschen der Straßenverkehrszulassung	
18	Rückstandsfreier Ein- und Ausbau für das Versuchsfahrzeug	
19	Software des Navigationssystems lauffähig auf *Ubuntu 16.04*	
20	Implementierung des Navigationssystems mithilfe von ROS	
21	Spannungsversorgung durch Versuchsfahrzeug beziehungsweise Industrie-PC	$5\,V$, $12\,V$
22	Betrieb unter regengeschützten Bedingungen	
23	Betrieb bei Umgebungstemperaturen für Indoor-Bereiche	$0\,^{\circ}C$ bis $30\,^{\circ}C$
24	Ausgabe der Position in geografischen Koordinaten	

Tabelle 3.1.: Anforderungen an das umfeldsensorbasierte Navigationssystem

4. Systementwurf

Aufbauend auf den analysierten Anforderungen wird in diesem Kapitel zunächst eine Systemarchitektur des umfeldsensorbasierten Navigationssystems konzipiert. Des Weiteren werden eine Architektur und eine Methode zur Positionsbestimmung des ultraschallbasierten Positionierungssystems entworfen. Abschließend wird ein Konzept zur Optimierung des Ultraschallsensorsystems entwickelt.

4.1. Systemarchitektur des umfeldsensorbasierten Navigationssystems

Um das umfeldsensorbasierte Navigationssystem zur Bestimmung von Position und Orientierung realisieren zu können, wird eine Systemarchitektur gewählt (siehe Abbildung 4.1), bei der die einzelnen positionsbestimmenden Systeme parallel angeordnet sind (vgl. Kichun u. a., 2015, S. 5121).

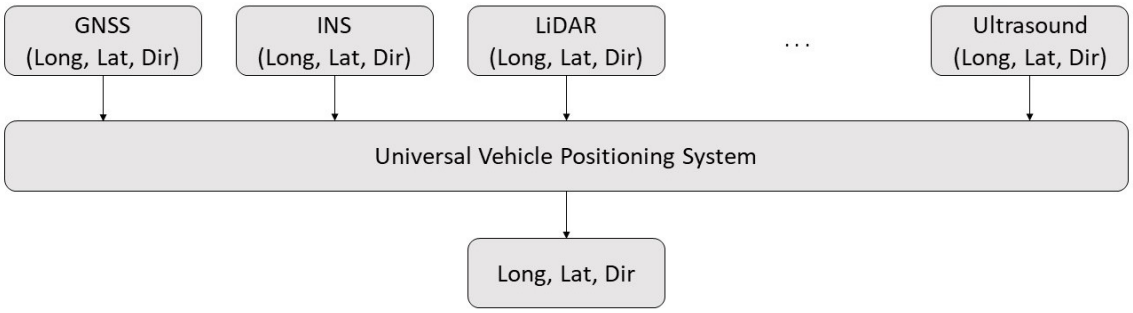


Abbildung 4.1.: Systemarchitekturentwurf des Gesamtsystems

Hierbei stellt das *Universal Vehicle Positioning System* (UVPS) das umfeldsensorbasierte Navigationssystem dar. Dabei werden die Daten der einzelnen Teilsysteme zu Positions- und Orientierungsdaten des Navigationssystems fusioniert. Des Weiteren wird das GNSS/INS-Systems in das UVPS integriert, da die umfeldsensorbasierten Systeme eine zeitweise, alternative Erweiterung zu diesem darstellen. Hierdurch kann ebenfalls die Funktionsfähigkeit

des Navigationssystems gewährleistet werden, wenn Teilsysteme ausfallen oder nicht verfügbar sind. Ein weiterer Punkt, der für die gewählte Systemarchitektur spricht, ist, dass das Navigationssystem modular und skalierbar um weitere umfeldsensorbasierte Sensorsysteme erweiterbar ist.

Die analysierten Anforderungen aus Tabelle 3.1, die mit der entworfenen Systemarchitektur erfüllt werden, sind in Tabelle 4.1 zu sehen.

ID	Beschreibung
12	Bei der Verwendung von mehreren Umfeldsensorsystemen Po-sitions- und Orientierungsbestimmung durch Sensordatenfusion
13	Navigationssystem modular erweiterbar um weitere Umfeldsensorsysteme
14	Funktionsfähigkeit des Navigationssystems bei Ausfall von Teilsystemen

Tabelle 4.1.: Erfüllte Anforderungen der Systemarchitektur des umfeldsensorbasierten Navigationssystems

4.2. Ultraschallbasiertes Positionierungssystem

Der folgende Abschnitt beschreibt die Konzeptionierung des ultraschallbasierten Positionierungssystems. Hierbei wird zunächst auf die Architektur des ultraschallbasierten Positionierungssystems eingegangen. Abschließend wird eine Methode zur ultraschallbasierten Positionsbestimmung entwickelt.

4.2.1. Architektur des ultraschallbasierten Positionierungssystems

Abbildung 4.2 zeigt die Architektur des ultraschallbasierten Positionierungssystems (UPS) und stellt eine dezentrale, hierarchische Struktur mit unterschiedlichen Ebenen dar (vgl. Gotlib u. a., 2019, S. 8). Die oberste Ebene ist der Industrie-PC des Versuchsfahrzeugs, auf dem das UVPS implementiert wird. Die untere Ebene hingegen ist das UPS.

Das UPS unterteilt sich des Weiteren nochmals in zwei hierarchische Ebenen. Hierbei ist die obere Ebene eine zentrale Recheneinheit, die das Positionierungssystem steuert und die Kommunikation mit dem UVPS übernimmt. Auf der unteren Ebene befinden sich die Ultraschallsensoren des Ultraschallsensorsystems aus Abschnitt 2.2.1.

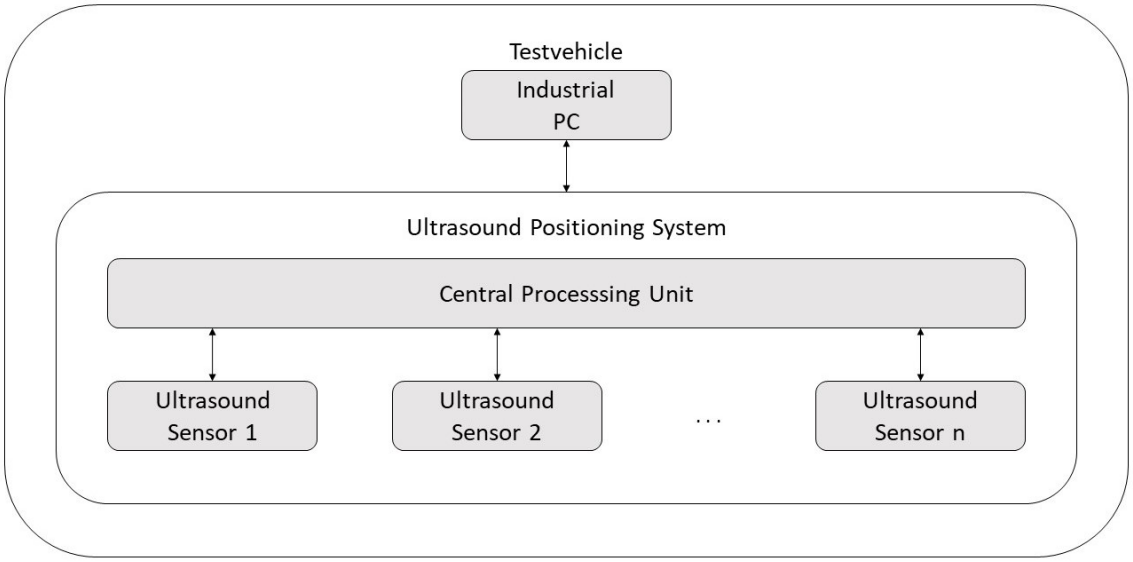


Abbildung 4.2.: Architekturentwurf des ultraschallbasierten Positionierungssystems

Diese Architektur ermöglicht eine modulare und skalierbare Anordnung der Ultraschallsensoren in Bezug auf Anzahl und Positionierung auf dem Fahrzeug. Ein möglicher Ausfall einzelner Ultraschallsensoren kann hierdurch ebenfalls abgefangen werden. Des Weiteren ist hierbei auch eine einfache Anbindung an den Industrie-PC des Versuchsfahrzeugs möglich. Ein weiterer Vorteil dieses dezentralen Architekturkonzepts ist, dass hierbei die Rechenlast zur Positionsbestimmung auf die zentrale Recheneinheit des UPS ausgelagert wird.

Tabelle 4.2 zeigt die Anforderungen aus Tabelle 3.1, die mit der entwickelten Architektur des UPS erfüllt werden.

ID	Beschreibung
3	Verwendung von Umfeldsensorsystemen
4	Verwendung eines Ultraschallsensorsystems
14	Funktionsfähigkeit des Navigationssystems bei Ausfall von Teilsystemen
15	Verwendung von mindestens zwei Ultraschallsensoren
16	Modulare und skalierbare Anordnung der Ultraschallsensoren
17	Integration in das Versuchsfahrzeug ohne Erlöschen der Straßenverkehrszulassung

Tabelle 4.2.: Erfüllte Anforderungen der Architektur des ultraschallbasierten Positionierungssystems

4.2.2. Methode zur ultraschallbasierten Positionsbestimmung

Für die Methode zur ultraschallbasierten Bestimmung von Position und Orientierung wird die Annahme getroffen, dass mithilfe von Ultraschallsensoren, die nicht nur das erste zurückkommende Ultraschallecho auswerten, die Umgebung durch Mehrfachreflexionen und mehrere zurückkommende Ultraschallechos charakterisiert werden kann (siehe Abbildung 4.3). Zur Charakterisierung einer Umgebung tragen markante Umgebungsbestandteile bei, wie zum Beispiel Wände oder Säulen. Mit der hierbei messbaren Ultraschallechosignatur ist eine Zuordnung zu einer Position und Orientierung möglich.

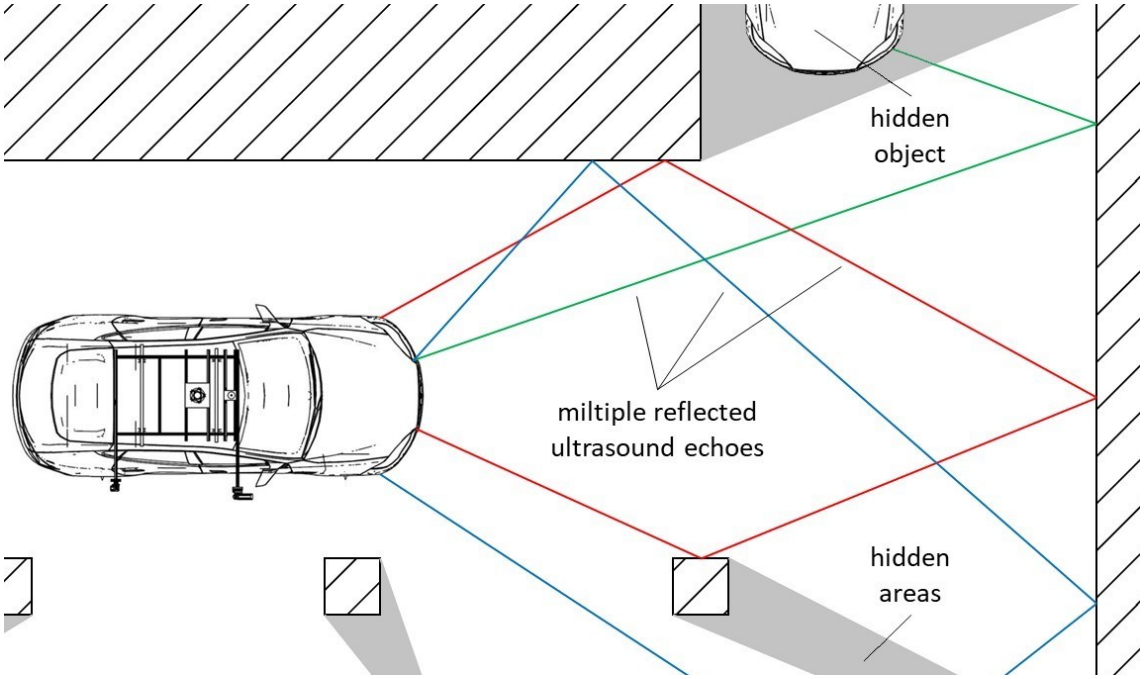


Abbildung 4.3.: Konzept zur Erkennung von Umgebung und verdeckten Objekten

Das Konzept sieht zur Positions- und Orientierungsbestimmung eine parallele Ultraschallmessung und -simulation vor (siehe Abbildung 4.4). Hierbei wird bei einer bereits bekannten Umgebung, in Form einer hinterlegten Karte, die Positionsermittlung durch einen Vergleich aus Messung und Simulation ermöglicht. Bei einer zu geringen Übereinstimmung wird dabei die Position optimiert, bis eine gültige Position gefunden wird.

Zur Simulation der Umgebung wird die Raytracing-Simulation aus Abschnitt 2.2.2 verwendet. Da die Ultraschallmessung eine Ultraschallsignatur einer 3D-Umgebung wiedergibt, wird die Simulation auf drei Dimensionen erweitert, um den Vergleich zwischen Messung und Simulation zu verbessern. Mit dem Vergleich zwischen Messung und Simulation können hierbei

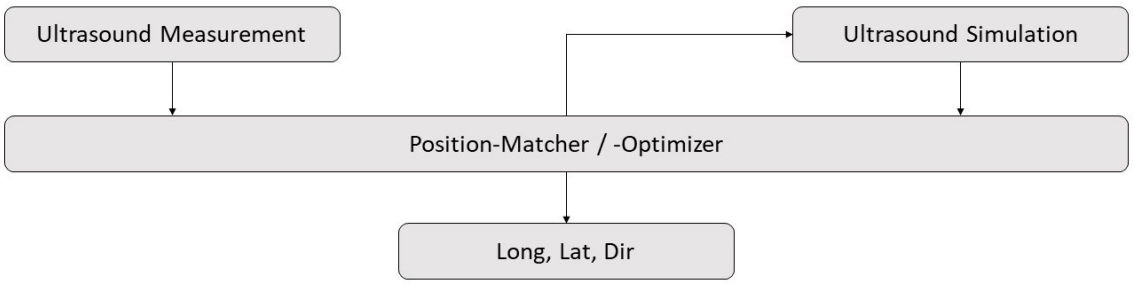


Abbildung 4.4.: Methodenkonzept der ultraschallbasierten Positions- und Orientierungsbestimmung

ebenfalls Objekte, die nicht fest in der Umgebung integriert sind beziehungsweise sich außerhalb des sichtbaren Bereichs befinden, erkannt werden.

In Tabelle 4.3 sind die aus Tabelle 3.1 erfüllten analysierten Anforderungen zu sehen.

ID	Beschreibung
1	Erkennung der Fahrzeugposition
2	Erkennung der Fahrzeugorientierung
5	Erkennung von Objekten, die sich nicht im direkten Sichtfeld befinden
6	Erkennung der Umgebung und von Objekten

Tabelle 4.3.: Erfüllte Anforderungen der Methode zur ultraschallbasierten Positionsbestimmung

4.3. Optimierung des Ultraschallsensors

Aufbauend auf dem Ultraschallsensorsystem aus Abschnitt 2.2.1 wird das Ultraschallsensorsystem optimiert und an die Aufgabe des UVPS angepasst. Hierzu wird die Architektur des Ultraschallsensors optimiert. Damit das Ultraschallsensorsystem flexibler eingesetzt werden kann, betrifft ein weiterer Optimierungsansatz die Spannungsversorgung des Ultraschallsensors. Des Weiteren wird ein Konzept zur verbesserten Synchronisation der Ultraschallsensoren untereinander entwickelt.

Architektur des Ultraschallsensors

Die Optimierung der Ultraschallsensorarchitektur sieht vor, dass diese im Hinblick auf eine modulare, skalierbare Anordnung und einer möglichst hohen Detektionsreichweite angepasste wird (siehe Abbildung 4.5). Bei dem Konzept der optimierten Ultraschallsensorarchitektur bleibt die variable Zuteilung des Masterstatus weiterhin erhalten, wodurch die Ultraschallsensoren identisch aufgebaut werden können.

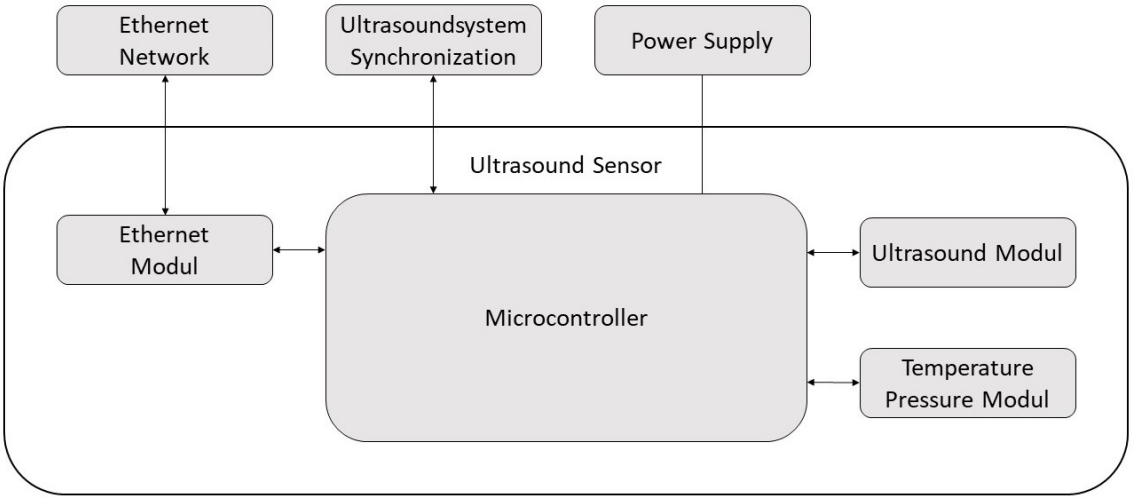


Abbildung 4.5.: Architekturentwurf des optimierten Ultraschallsensors

Die Analyse im Vorfeld dieser Arbeit hat gezeigt, dass durch den Austausch der Pfade für das Senden und Empfangen von Ultraschall eine höhere Detektionsreichweite realisiert werden kann. Um eine Detektionsreichweite von mindestens 5 m zu realisieren, werden diese durch ein Ultraschallmodul ausgetauscht, welches Senden und Empfangen kann. Die Module für das Messen von Umgebungstemperatur und -druck beziehungsweise der Ethernetkommunikation bleiben hingegen bestehen. Ebenfalls bestehen bleibt das Mikrocontrollerboard als Recheneinheit des Ultraschallsensors.

Spannungsversorgung des Ultraschallsensors

Da die Spannungsversorgung der Ultraschallsensoren mithilfe von integrierten 5 V-Akkus und der Erzeugung von unterschiedlichen Betriebsspannungen (vgl. Rotzlawski, 2018, S. 48) den Bauraum der Ultraschallsensoren erhöht und hierbei eine Überwachung der Akkuladestände nötig ist, wird die Spannungsversorgung von einer internen hinzu einer externen Spannungsversorgung umgewandelt. Hierbei wird die Spannungsversorgung über das Versuchsfahrzeug beziehungsweise dem Industrie-PC erfolgen. Hierzu werden alle Ultraschallsensoren von derselben Spannungsquelle versorgt.

Synchronisation der Ultraschallsensoren

Die Synchronisation der Ultraschallsensoren untereinander wird von einer einfachen Verbindungsleitung zu einer Ringleitung optimiert (siehe Abbildung 4.6). Hierbei wird die Funktion der Synchronisationsringleitung, dem vom Mastersensor gesteuerten Starten und Beenden einer Ultraschallmessung, beibehalten. Zusätzlich wird die Synchronisationsringleitung um die Funktion des Auslösens eines Resets der Ultraschallsensoren erweitert. Damit der Mastersensor nicht für jede Messung einzeln ausgewählt und gestartet werden muss, wird die Weitergabe des Masterstatus über die Synchronisationsringleitung realisiert. Hierdurch wird die modulare und skalierbare Anwendung der Ultraschallsensoren weiter erhöht.

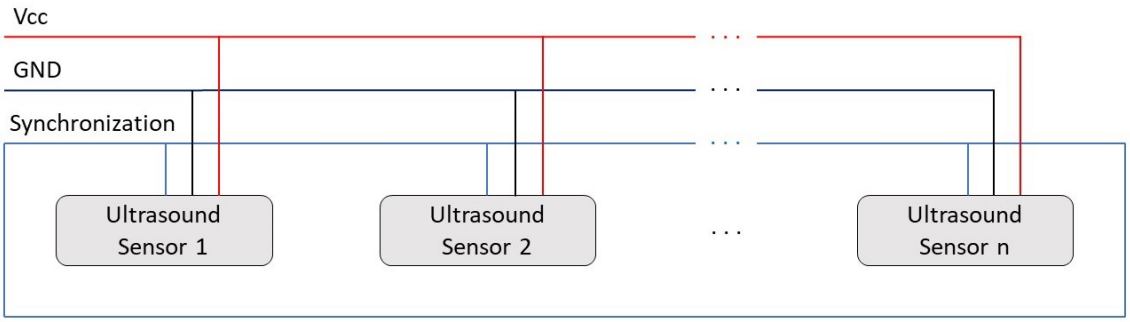


Abbildung 4.6.: Entwurf zur Synchronisation des Ultraschallsensorsystems

Die durch das Konzept des optimierten Ultraschallsensors erfüllten Anforderungen an das UVPS aus Tabelle 3.1 sind in Tabelle 4.4 zu sehen.

ID	Beschreibung
7	Möglichst hohe Detektionsreichweite der einzelnen Umfeldsensorsysteme
15	Verwendung von mindestens zwei Ultraschallsensoren
16	Modulare und skalierbare Anordnung der Ultraschallsensoren
21	Spannungsversorgung durch Versuchsfahrzeug beziehungsweise Industrie-PC

Tabelle 4.4.: Erfüllte Anforderungen des optimierten Ultraschallsensors

5. Realisierung

Die Realisierung beschreibt zunächst die Umsetzung des optimierten Ultraschallsensors. Aufbauend darauf folgt die Umsetzung des ultraschallbasierten Positionierungssystems. Den Kapitelabschluss bildet die Realisierung des umfeldsensorbasierten Navigationssystems als Gesamtsystem.

5.1. Ultraschallsensor

Aufbauend auf dem optimierten Konzept erfolgt folgend die Umsetzung der Ultraschallsensoren. Hierzu wird zunächst für das Ultraschallmodul eine Komponentenauswahl getroffen. Anschließend erfolgt eine Ausarbeitung einer definierten Ultraschallsensorarchitektur. Des Weiteren werden die Synchronisation und Spannungsversorgung der Ultraschallsensoren realisiert. Ein weiterer Bestandteil der Umsetzung ist das Hardwaredesign des Ultraschallsensors. Abschließend wird die Software des Ultraschallsensors an die Optimierungen angepasst.

5.1.1. Komponentenauswahl des Ultraschallmoduls

Als Ultraschallmodul wird ein *MaxBotix XL-MaxSonar*-Modul verwendet, da dieses die Erkennung von mehreren zurückkommende Ultraschallechos ermöglicht (vgl. MaxBotix, 2015, S. 1). Hierzu wird als Ultraschallmodul der *XL-MaxSonar 1360* verwendet, da dieser im Vergleich zum *XL-MaxSonar 1300* eine höhere Reichweite aufweist. Ein Vergleich der Spezifikationen des Ultraschallmoduls mit den Anforderungen auf Tabelle 3.1 ist in Tabelle 5.1 zu sehen.

Hierbei bietet das Ultraschallmodul mit bis zu 10,68 m eine hohe Detektionsreichweite, was eine ausreichend hohe Reichweite darstellt. Des Weiteren liefert das Ultraschallmodul ein analoges und ein digitales Ausgangssignal. Mit dem digitalen Ausgangssignal ist allerdings nur eine einfache Distanzmessung möglich. Das analoge Ausgangssignal liefert eine Hüllkurve der zurückkommenden Ultraschallechos, womit eine Erkennung von Objekten und der Umgebung ermöglicht wird. Der *XL-MaxSonar MB1360* hat eine feste Messzeit vom 100 ms.

Spezifikation	Wert	Anforderung
Detektionsbereich	0,2 ... 10,68 m	$x \geq 5\,m$
Messrate	max. 8,3 Hz	$f \geq 1{,}67\,Hz$
Messfrequenz	42 kHz	-
Ausgangssignal	Amplituden der Ultraschallechos (analog) einfache Distanz (digital)	-
Operationsmodi	frei laufend getriggert	-
Spannungsversorgung	3,3 ... 5 V	5 V, 12 V
Betriebstemperatur	-40 ... 65 $°C$	0 $°C$ bis 30 $°C$

Tabelle 5.1.: Spezifikationen des *MaxBotix XL-MaxSonar MB1360* im Vergleich zu den Anforderungen

Beim Start einer Messung benötigt das Modul zusätzlich 20,5 ms für die Kalibrierung. Hierdurch kann eine Messrate von 8,3 Hz realisiert werden (vgl. MaxBotix, 2015, S. 1).

Durch die kompakte Bauform und der integrierten Verstärkerschaltung kann der *XL-MaxSonar MB1360* leicht und platzsparend in den Ultraschallsensor integriert werden. Des Weiteren ist der Sensor durch seine Eigenschaften für Multisensorsysteme und den Einsatz für autonome mobile Systeme, wie etwa das autonome Fahren, geeignet.

Da eine Erkennung von mehreren zurückkommenden Ultraschallechos nur über den analogen Pfad des Moduls möglich ist, wird dieses an den A-/D-Wandler des *Arduino Dues* angeschlossen.

5.1.2. Vertiefung der Ultraschallsensorarchitektur

Im Folgenden wird die Architektur des Ultraschallsensors detaillierter definiert (siehe Abbildung 5.1). Hierzu werden die einzelnen Module in die Ultraschallsensorarchitektur integriert. Des Weiteren wird die Architektur um die Schnittstellenart, die die einzelnen Module am Mikrocontroller verwenden, erweitert.

Vergleichbar zur Ultraschallsensorarchitektur aus Abschnitt 2.2.1 sind das *Arduino Due* Mikrocontrollerboard und das *WIZnet W5500* Ethernet-Shield. Hierbei wird zur Kommunikation zwischen Mikrocontrollerboard und Ethernet-Shield weiterhin SPI als Schnittstelle verwendet. Durch das Ethernet-Shield ist der Ultraschallsensor weiterhin an ein Ethernetnetzwerk angeschlossen.

Das Modul zum Messen von Umgebungsdruck und -temperatur wird leicht angepasst. Da das *Adafruit BMP280* Board nur als Verbindungsboard dient, wird auf dieses verzichtet und

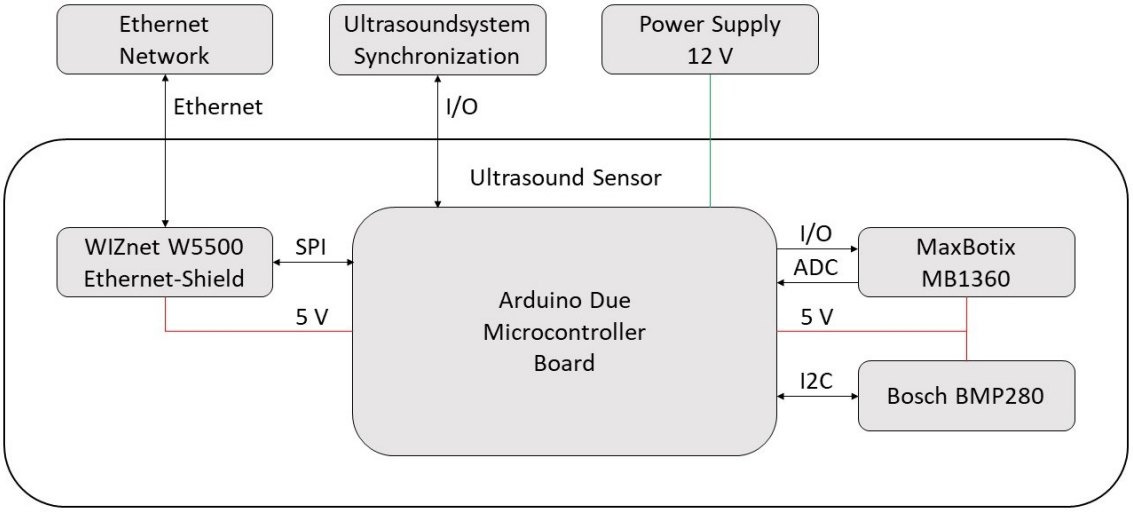


Abbildung 5.1.: Definierte Architektur des Ultraschallsensors

der *Bosch BMP280* direkt angesteuert. Als Schnittstelle zum Mikrocontrollerboard wird weiterhin I2C verwendet.

Durch die Verwendung des *MaxBotix* Ultraschallmoduls werden die Pfade zum Senden und Empfangen von Ultraschall eingespart, da diese auf dem Modul integriert sind. Um eine Messung des Ultraschallmoduls auszulösen, wird dieses an einen I/O-Pin des *Arduino Dues* angeschlossen. Die Schnittstelle zur Digitalisierung der empfangenen Ultraschallechos ist hingegen ein ADC-Pin des Mikrocontrollerboards.

Des Weiteren werden als Schnittstellen zur Synchronisation der Ultraschallsensoren untereinander und der Weitergabe des Masterstatus I/O-Pins des Mikrocontrollerboards verwendet.

Da der *Arduino Due* über einen integrierten Spannungsregler verfügt, der $5\,V$ zur Verfügung stellt, werden die einzelnen Module des Ultraschallsensors über diesen mit Betriebsspannung versorgt. Hierzu wird der *Arduino Due* an eine externe $12\,V$ Spannungsquelle angeschlossen.

Durch den Wegfall der Sende- und Empfangspfade und den Verzicht auf das *Adafruit* Board wird Bauraum eingespart, wodurch eine deutlich kompaktere Bauform realisiert wird.

5.1.3. Synchronisation und Spannungsversorgung der Ultraschallsensoren

Die Synchronisation der Ultraschallsensoren erfolgt, wie im Kapitel Systementwurf beschrieben, über eine Ringleitung. Hierbei wird die Ringleitung über die Funktion der Spannungsversorgung der Ultraschallsensoren erweitert (siehe Abbildung 5.2). Die Umsetzung der Ringleitung erfolgt als zehnadriges Flachbandkabel mit Pfostenverbindern (siehe Anhang A.1). Die Verbindung zu den einzelnen Ultraschallsensoren erfolgt dabei über je zwei Wannenstecker. Das Flachbandkabel hat hierbei einen Aderquerschnitt von 0,09 mm^2.

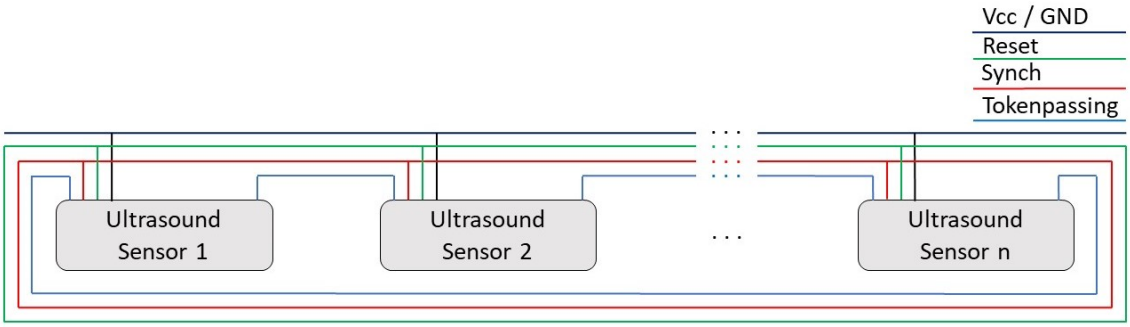


Abbildung 5.2.: Synchronisation und Spannungsversorgung der Ultraschallsensoren

Für die Aderanzahl, die für die Spannungsversorgung verwendet werden, wird die Annahme getroffen, dass maximal sechs Ultraschallsensoren mit einer Ringleitung synchronisiert und mit Spannung versorgt werden. Hierfür werden für die Spannungsversorgung drei Adern des Flachbandkabels verwendet, um die Stromstärke pro Ader auf maximal 1 A zu begrenzen. Des Weiteren werden vier Flachbandadern als Masse verwendet.

Die restlichen drei Flachbandadern dienen der Synchronisation der Ultraschallsensoren. Hierbei wird eine Ader zur Synchronisierung der Messung verwendet. Das heißt, dass die Ultraschallmessungen der Ultraschallsensoren synchronisiert gestartet und beendet werden. Über eine weitere Ader kann ein Reset der Ultraschallsensoren, die mit der Ringleitung verbunden sind, ausgelöst werden. Hierbei wird der Reset vom Ultraschallsensor mit dem Masterstatus ausgelöst.

Die letzte Ader wird für die Weitergabe des Masterstatus verwendet. Hiermit kann der Masterstatus über ein Token Passing nach einem Messdurchlauf an den jeweils nächsten Ultraschallsensor weitergegeben werden. Mit dem Token Passing wird realisiert, dass der jeweilige Messprozess nur einmal angestoßen und allen Ultraschallsensoren an der Ringleitung einmal der Masterstatus zugeteilt wird. Der Messprozess unterteilt sich dabei in unterschiedliche Messdurchläufe, bei denen der jeweilige Ultraschallsensor mit dem Masterstatus einen

Ultraschallpuls aussendet und den Messdurchlauf über die Synchronisationsleitung startet und wieder beendet.

5.1.4. Hardwaredesign

Dieser Abschnitt beschreibt zunächst das Platinenlayout, um die Synchronisation und Spannungsversorgung der Ultraschallsensoren zu realisieren. Des Weiteren wird mit dem Platinenlayout die Integration des Ultraschallmoduls und des Moduls zur Messung von Umgebungstemperatur und -druck in den Ultraschallsensor umgesetzt. Anschließend erfolgt die Realisierung des Ultraschallsensorgehäuses.

Platinendesign

Um eine möglichst kompakte Bauform zu realisieren und den Formfaktor des *Arduino Dues* zu bewahren (vgl. Arduino, 2020), wird die Platine neben dem Ethernet-Shield auf dem Mikrocontrollerboard platziert. Hierbei werden die Verbindungen zwischen Platine und Mikrocontrollerboard mithilfe von Stiftleisten hergestellt. Dabei wird die Grundfläche der Platine auf 53,4 mm x 35,95 mm festgelegt (siehe Anhang A.2 und A.3).

Damit auf der Platine eine Verbindung zur Ringleitung zum Synchronisieren und zur Spannungsversorgung realisiert wird, werden auf dieser zwei Wannenstecker platziert. Zur Stabilisierung der Spannungsversorgung wird ein 470 μF Kondensator verwendet. Des Weiteren wird die Spannungsversorgung mit dem 12 V Eingang des *Arduino Dues* verbunden (siehe Abbildung 5.3). Um die einzelnen Komponenten mit Spannung zu versorgen, wird die 5 V Spannung des Mikrocontrollerboards verwendet. Zusätzlich wird eine 3,3 V Spannung auf der Platine erzeugt.

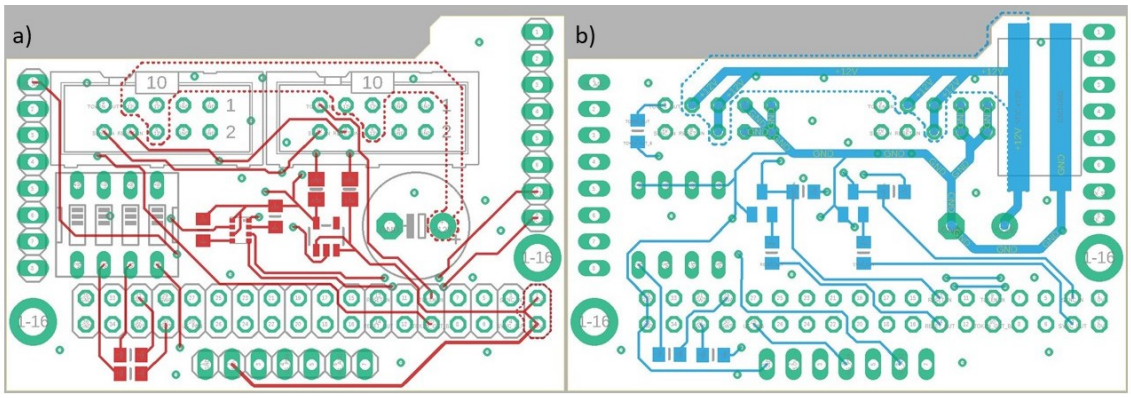


Abbildung 5.3.: Platinenlayout a) Oberseite und b) Unterseite

Ein weiterer Punkt ist die Zuweisung der Sensornummer. Da das softwaretechnische Zuweisen der Sensornummer fehleranfällig ist und der Ultraschallsensor bei einer neuen Sensornummer neu programmiert werden muss, wird auf der Platine ein 4-poliger DIP-Schalter platziert, über den die Sensornummer zugewiesen werden kann. Da der DIP-Schalter mit I/O-Pins des Mikrocontrollers verbunden ist und diese bei Betätigung auf Masse gezogen werden, werden zwischen den I/O-Pins und dem DIP-Schalter Widerstände platziert, um den Strom zu begrenzen.

Das Synchronisieren und Resetten der Ultraschallsensoren wird mithilfe identischer Open-Collector Schaltungen mit Pull-Up Widerstand realisiert. Hierbei werden die Leitungen mit dem Pull-Up Widerstand auf 3,3 V gehalten. Zum Resetten beziehungsweise Synchronisieren schaltet der Ultraschallsensor mit dem Masterstatus den NMOS-Transistor und zieht die jeweilige Leitung auf Masse.

Des Weiteren wird der Ausgangspin, der das Token des Masterstatus an den nächsten Ultraschallsensor weitergibt, über die Ringleitung mit dem Eingangspin des nächsten Ultraschallsensors verbunden. Hierbei wird ebenfalls ein Widerstand zwischen den Pins platziert, um den Strom zu begrenzen.

Der *Bosch BMP280* Sensor wird auf der Oberseite der Platine platziert und mit der erzeugten 3,3 V Spannung versorgt. Das Ultraschallmodul wird mit einer 90° Stiftleiste angebracht und in Längsrichtung des Mikrocontrollerboards ausgerichtet. Hierdurch wird eine flexible Verbindung zwischen Platine und Ultraschallmodul realisiert, wobei das Ultraschallmodul modular ausgetauscht werden kann.

Die aufgebaute und auf den *Arduinio Due* aufgesteckte Platine ist in Abbildung 5.4a zu sehen. Abbildung 5.4b zeigt die Verbindung zwischen der Spannungsversorgung und dem 12 V Eingang des *Arduino Dues*. Hierdurch kann ein *Arduino Due* mit externer Spannung versorgt werden, wodurch durch die Ringleitung die jeweils anderen *Arduino Dues* mit Spannung versorgt werden.

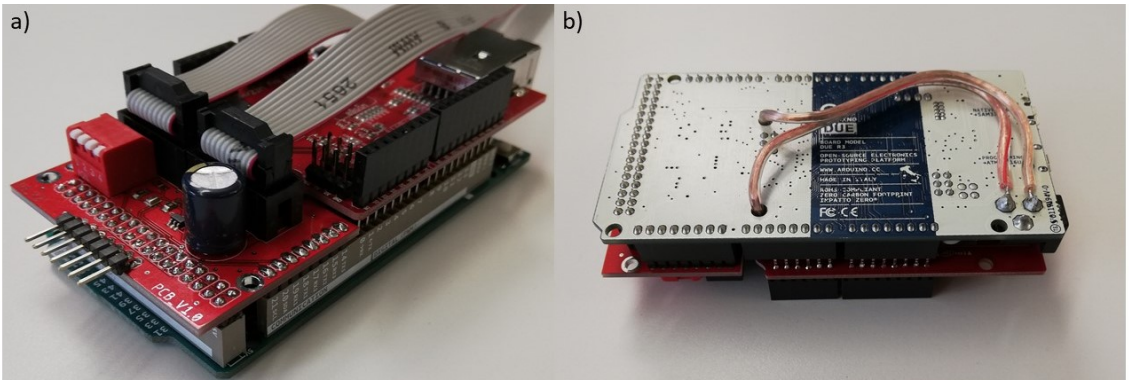


Abbildung 5.4.: a) aufgebaute Platine und b) Spannungsversorgung des *Arduino Dues*

Gehäuse

Damit der Ultraschallsensor flexibel an dem Versuchsfahrzeug angebracht werden kann, wird ein Gehäuse für diesen entworfen (siehe Abbildung 5.5). Hierzu nimmt das Gehäuse ein Grundvolumen von 68 mm x 43 mm x 125 mm ein (siehe Anhang A.4), wodurch eine kompakte Bauform erreicht wird und der Formfaktor des *Arduino Dues* mit zwei aufgesteckten Platinen erhalten bleibt.

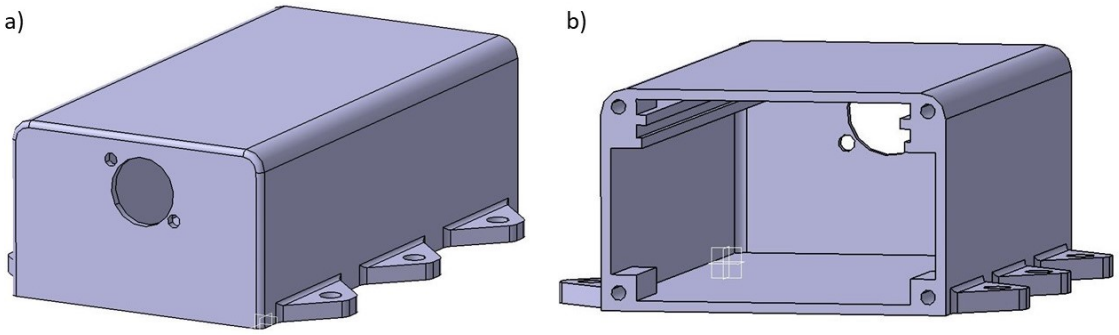


Abbildung 5.5.: CAD-Modell des Ultraschallsensorgehäuses a) Vorderansicht und b) Rückansicht

Die Verbindung des Ultraschallmoduls mit dem Gehäuse wird durch eine Verschraubung an der Vorderseite realisiert. Hierbei erfolgt die Anbringung von der Innenseite des Gehäuses, wobei das Ultraschallmodul durch eine Bohrung nach außen ragt. Die Positionierung des Mikrocontrollerboards erfolgt mithilfe von Führungsschienen. Hiermit kann das Mikrocontrollerboard in das Gehäuse geschoben werden, wobei die Führungsschienen die Zusammenführung der Verbindung zwischen der Platine und dem Ultraschallmodul gewährleisten.

Um das Mikrocontrollerboard in dem Gehäuse zu fixieren und um das Gehäuse zu verschließen, wird an der Rückseite des Gehäuses ein Deckel angebracht. Des Weiteren verfügt das Gehäuse zur flexiblen und modularen Anbringung an das Versuchsfahrzeug über sechs Laschen zur Befestigung. Hierbei wird das Ultraschallsensorgehäuse mit einem 3D-Druck als Rapid Prototyping Verfahren hergestellt. Des Weiteren ist in Abbildung 5.6 ein optischer Vergleich zwischen dem Ultraschallsensor aus Abschnitt 2.2.1 und dem optimierten Ultraschallsensor zu sehen.

5.1.5. Software der Ultraschallsensoren

Da das Hardwaredesign deutlich verändert wurde, ist ebenfalls eine Optimierung der Ultraschallsensorsoftware erforderlich (siehe Anhang C). Hierbei wird der Softwareablauf deutlich verschlankt.

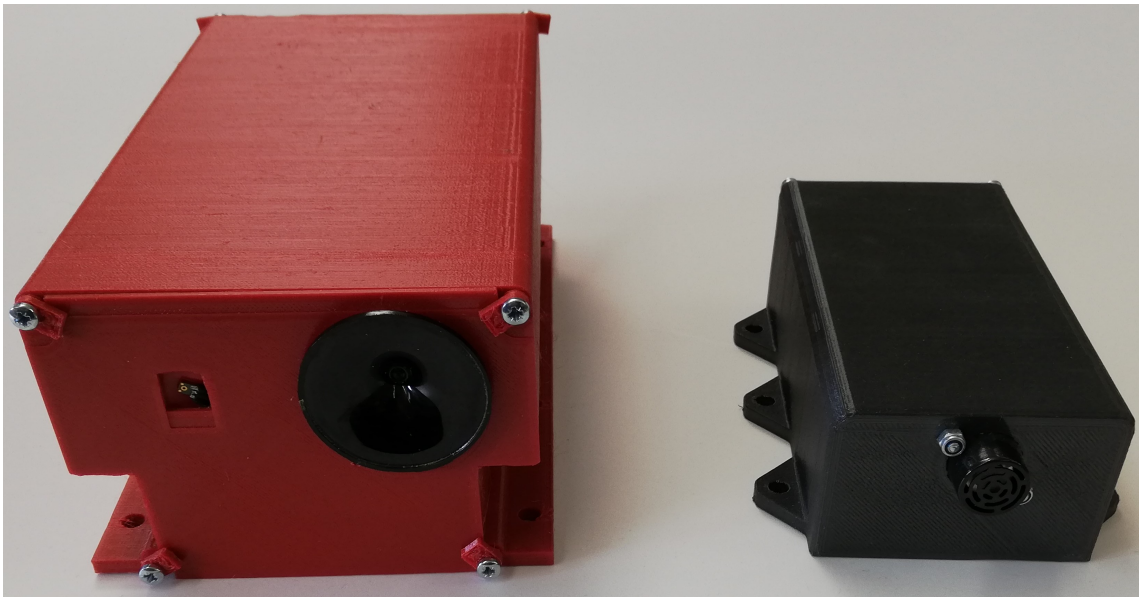


Abbildung 5.6.: Links der Ultraschallsensor aus Abschnitt 2.2.1 und rechts der optimierte Ultraschallsensor

Hierzu wird zunächst der Ablauf des Hauptprogramms angepasst (siehe Anhang B.1). Der Hauptablauf wird so angepasst, dass die Ultraschallsensoren automatisch während des Systemstarts den Slavestatus erhalten. Dadurch entfällt die Zuteilung des Slavestatus über das Ethernetprotokoll. Hierbei muss nur noch an den Ultraschallsensor, der den Masterstatus erhält und den Messprozess startet, ein Ethernetprotokoll gesendet werden. Des Weiteren wird der Systemstart der Ultraschallsensoren angepasst. Dieser erhält zusätzlich eine Power-Up-Phase von 200 ms für den Start des Ultraschallmoduls (vgl. MaxBotix, 2015, S. 6). Ebenfalls Bestandteil des Systemstarts ist die Abfrage der Sensornummer, mit der die IP-Adresse festgelegt wird. Dies geschieht über die Abfrage der entsprechenden Eingangspins des Mikrocontrollerboards. An den Abfragen des Status beziehungsweise der Anforderung einer Umgebungstemperatur und -druck Messung wird hingegen nichts verändert.

Ein weiterer Punkt der Optimierung ist die Anpassung des Slavemode. Hier erfolgt am Ende eines Messdurchlaufs, durch das Hinzufügen des Token Passings, eine Abfrage, ob der entsprechende Ultraschallsensor das Mastertoken erhalten hat und hierdurch einen neuen Messdurchlauf startet.

Im Bezug auf das verwendete Ultraschallmodul erfolgt die Optimierung des Mastermode. Hierbei wird am Start einer Messung zunächst das Ultraschallmodul angesteuert. Das Ultraschallmodul benötigt dabei 20,5 ms, um eine Messung zu starten, da dieses vor einer Messung kalibriert wird (vgl. MaxBotix, 2015, S. 6). Da der Ultraschallsendepfad entfernt wurde, wird die Erzeugung des Sendepulses nicht mehr benötigt und wird ebenfalls entfernt.

Des Weiteren wird die eigentliche Messung über die Ringleitung gestartet. Hierbei dauert eine Messung 100 ms. Am Ende einer Messung wird das Token Passing implementiert, bei dem das Mastertoken an den nächsten Ultraschallsensor weitergegeben wird. Das Token Passing wird so implementiert, dass dieses aktiviert beziehungsweise deaktiviert werden kann.

Das Resetten der Ultraschallsensoren erfolgt bei aktiviertem Token Passing dabei nach einem Messprozess. Das heißt, wenn alle Ultraschallsensoren für ein Messdurchlauf das Mastertoken hatten und dies wieder an den ersten Ultraschallsensor übergeben wird. Bei deaktiviertem Token Passing hingegen wird von dem jeweiligen Ultraschallsensor mit dem Mastertoken das Resetten direkt nach dem ersten Messdurchlauf durchgeführt.

Eine weitere Softwareoptimierung betrifft die Abtastrate des A-/D-Wandlers. Da das Ultraschallmodul eine deutlich niedrigere Frequenzbandbreite als das MEMS-Mikrofon des Ultraschallsensors aus Abschnitt 2.2.1 besitzt und dadurch eine Abtastrate von 200 kHz nicht erforderlich ist, wird diese reduziert. Die typische Bandbreite von Ultraschallsensoren liegt im Bereich von 1 kHz bis 2,5 kHz. Da in dem Datenblatt des Ultraschallmoduls diesbezüglich keine Angabe gemacht wird, wird eine Bandbreite von 2,5 kHz angenommen. Für diesen Fall sollte die Abtastrate unter Beachtung des Abtasttheorems bei mindestens 5 kHz liegen (siehe Formel 5.1). Um allerdings eine gute Unterscheidung von mehreren zurückkommenden Ultraschallechos zu realisieren und diesbezüglich eine hohe Auflösung zu erhalten, wird die Abtastrate auf 100 kHz festgelegt.

$$f_S \geq 2B_{US} \geq 5\ kHz \tag{5.1}$$

Da der Hauptprogrammablauf optimiert wurde, müssen mit dem Ethernetprotokoll deutlich weniger Parameter übertragen werden. Hierbei wird die Komplexität des Ethernetprotokolls reduziert. Des Weiteren werden dem Ethernetprotokoll Identifier- und Check-Bytes hinzugefügt, um Fehler in der Kommunikation abzufangen und ein Fehlverhalten der Ultraschallsensoren zu vermeiden. Das überarbeitete Ethernetprotokoll ist in Tabelle 5.2 zu sehen. Die Kommunikation wird ebenfalls so angepasst, dass für die Ethernetpakete nur ein Ethernetport verwendet wird.

Nachricht	Instruktion		Antwort		
	Identifier	**Check**	**Identifier**	**Check**	**Daten**
Status	0x4242	Counter 2 byte	0x4242	Counter	- - -
Temperatur & Druck	0xABCD	Counter 2 byte	0xABCD	Counter	- - -
			0xABCD	- - -	3 byte Druck 3 byte Temperatur 24 byte Kalibrierung
Messung	0xF00F	Counter 2 byte	0xF00F (Antwort)	Counter	- - -
			0xF00F (Messung)	- - -	2 byte UDP-Counter 1460 byte Daten
			0xF042 (Ende)	- - -	2 byte UDP-Counter
Fehler	- - -	- - -	0xFFFF	0xFFFF	- - -

Tabelle 5.2.: Aufbau des Ethernetprotokolls zur Kommunikation

5.2. Ultraschallbasiertes Positionierungssystem

Im folgenden Abschnitt wird zunächst die Auswahl der eingebetteten zentralen Recheneinheit beschrieben. Aufbauend darauf erfolgt eine detailliertere Architektur des ultraschallbasierten Positionierungssystems. Ein weiterer Punkt ist die Anordnung der Ultraschallsensoren. Anschließend wird die Software des ultraschallbasierten Positionierungssystems realisiert. Aufbauend auf der Software folgt die Umsetzung der Algorithmen zur ultraschallbasierten Positionsbestimmung.

5.2.1. Komponentenauswahl der eingebetteten zentralen Recheneinheit

Da die zentrale Recheneinheit für die Berechnungen und damit für die Positionsbestimmung des UPS zuständig ist, üben die Eigenschaften der zentralen Recheneinheit einen großen Einfluss auf die Leistungsfähigkeit des UPS aus. Hierbei wird als zentrale Recheneinheit ein eingebettetes System verwendet. Hierzu werden der *Raspberry Pi 4 Model B* (vgl. RaspberryPi, 2020, S. 1) und der *NVIDIA Jetson Nano* (vgl. NVIDIA, 2020a, S. 1) miteinander verglichen. Abschließend wird ein Board als eingebettete zentrale Recheneinheit ausgewählt.

Raspberry Pi 4 Model B

Das *Raspberry Pi 4 Model B* Board ist ein Einplatinencomputer mit vier ARM-Prozessoren und einem Takt von 1,5 GHz. Das Board verfügt über einen wählbaren Arbeitsspeicher von 1 GB bis 4 GB. Hauptspeicher ist auf dem Board keiner vorhanden. Dies wird hingegen mit einer SD-Karte gelöst, die ebenfalls mit dem Image des Betriebssystems geflasht wird. Die für den Einsatz als eingebettete zentrale Recheneinheit relevanten Schnittstellen sind jeweils zwei USB 2.0 und 3.0 Buchsen, eine WLAN- sowie eine Gigabit-Ethernet-Schnittstelle. Des Weiteren verfügt das Board über 40 GPIO-Pins. Ein weiteres Merkmal des *Raspberry Pi* ist eine im SoC integrierte GPU-Einheit, die im Datenblatt allerdings nicht weiter spezifiziert wird (vgl. RaspberryPi, 2020, S. 6).

Es wird für das Board eine Spannungsversorgung von 5 V benötigt. Hierbei sollte das Board mit mindestens 2,5 A in der Spitze versorgt werden können. Des Weiteren kann das Board über eine optionale Platine mithilfe von PoE mit Spannung versorgt werden (vgl. RaspberryPi, 2020, S. 6).

Die Maße des *Raspberry Pi 4 Model B* betragen 85 mm x 56 mm x 20 mm. Das Board verfügt über keine passive beziehungsweise aktive Kühlung, die allerdings nachgerüstet werden kann (vgl. RaspberryPi, 2020, S. 6).

NVIDIA Jetson Nano

Bei dem *NVIDIA Jetson Nano* handelt es sich um ein System on Module. Hierbei verfügt das Board über vier ARM-Prozessoren mit einem Takt von 1,43 GHz. Der Arbeitsspeicher beträgt dabei 4 GB. Das Board verfügt ebenfalls, wie das *Raspberry Pi* Board, über keinen integrierten Hauptspeicher. Dieser muss dem Board mithilfe einer SD-Karte zur Verfügung gestellt werden, auf der ebenfalls das Image des Betriebssystems gespeichert wird. Das Board verfügt als Schnittstelle über vier USB 3.0 Buchsen. Des Weiteren besitzt das Board eine Gigabit-Ethernet- und eine WLAN-Schnittstelle. Die Anzahl der GPIO-Pins beträgt 40. Des Weiteren verfügt das Board über eine Grafikeinheit mit 128 Kernen. Dabei eignet sich der *Jetson Nano* für Anwendungen im Bereich des maschinellen Lernens (vgl. NVIDIA, 2020a, S. 1).

Die Spannungsversorgung, die das Board benötigt, beträgt 5 V. Hierbei wird ein Maximalstrom von 4 A benötigt. Das Board kann dabei über PoE mit Spannung versorgt werden (vgl. NVIDIA, 2020a, S. 1).

Die Abmessungen des *NVIDIA Jetson Nano* betragen 100 mm x 80 mm x 29 mm. Des Weiteren verfügt das Board über einen Kühlkörper als passive Kühlung, wobei ein Lüfter als aktive Kühlung nachgerüstet werden kann (vgl. NVIDIA, 2020a, S. 1).

Auswahl der eingebetteten zentralen Recheneinheit

Vergleicht man die beiden eingebetteten Systeme miteinander, weisen beide identische Eigenschaften auf (siehe Tabelle 5.3). Hierbei besitzt der *Jetson Nano* einen etwas niedrigeren Prozessortakt. Des Weiteren benötigt der *Jetson Nano* mehr Strom. Dieser hat allerdings die Möglichkeit der Spannungsversorgung mithilfe von PoE bereits integriert und benötigt keine zusätzliche Platine. Betrachtet man die GPIO-Pins und die Schnittstellen, sind beide Systeme identisch ausgestattet. Ebenfalls identische Eigenschaften weisen die Systeme im Hinblick auf Arbeitsspeicher und Hauptspeicher auf. Im Bezug auf eine Kühlung ist der *Jetson Nano* bereits mit einem passiven Kühlkörper ausgestattet. Der *Paspberry Pi* weist in dem Vergleich eine deutlich geringere Abmessung auf. Ein weiterer Punkt ist die Grafikeinheit. Diese fällt beim *NVIDIA Jetson Nano* deutlich größer aus als bei dem *Raspberry Pi 4 Model B*.

Eigenschaft	**Raspberry Pi 4 Model B**	**NVIDIA Jetson Nano**
Prozessortakt	1,5 GHz	1,43 GHz
Prozessorkerne	4	4
Grafikeinheit	auf SoC integriert	Grafikeinheit mit 128 Kernen
Arbeitsspeicher	1 GB bis 4 GB	4 GB
Speicher	SD-Karte benötigt	SD-Karte benötigt
Schnittstellen	2x USB 2.0, 2x USB 3.0, Gigabit-Ethernet, WLAN	4x USB 3.0, Gigabit-Ethernet, WLAN
Anzahl GPIOs	40	40
Energieversorgung	5 V und 2,5 A, optional über PoE mit zusätzlicher Platine	5 V und 4 A, optional über PoE
Abmessungen	85 mm x 56 mm x 20 mm	100 mm x 80 mm x 29 mm
Kühlung	passiv und aktiv optional	passiver Kühlkörper, aktiv optional

Tabelle 5.3.: Gegenüberstellung der eingebetteten Systeme

Da der *Raspberry Pi* zwar im Bezug auf Abmessungen und Stromverbrauch bessere Eigenschaften aufweist, der *Jetson Nano* allerdings eine deutlich größere grafische Einheit besitzt, wird dieser als eingebettete zentrale Recheneinheit ausgewählt. Hierbei können große parallele Berechnungen, wie zum Beispiel Teile der Raytracing-Simulation, zur Entlastung des Hauptprozessors auf die grafische Einheit ausgelagert werden. Ein weiterer Vorteil des *NVIDIA Jetson Nano* ist die Auslegung auf maschinelles Lernen, wodurch die eingebettete zentrale Recheneinheit flexibler im Bezug auf mögliche Anwendungen in diesem Bereich wird.

5.2.2. Vertiefung der Architektur

Die vertiefte Architektur des UPS ist in Abbildung 5.7 zu sehen. Hierzu werden die einzelnen Teilsysteme und Bestandteile definiert und in das System integriert.

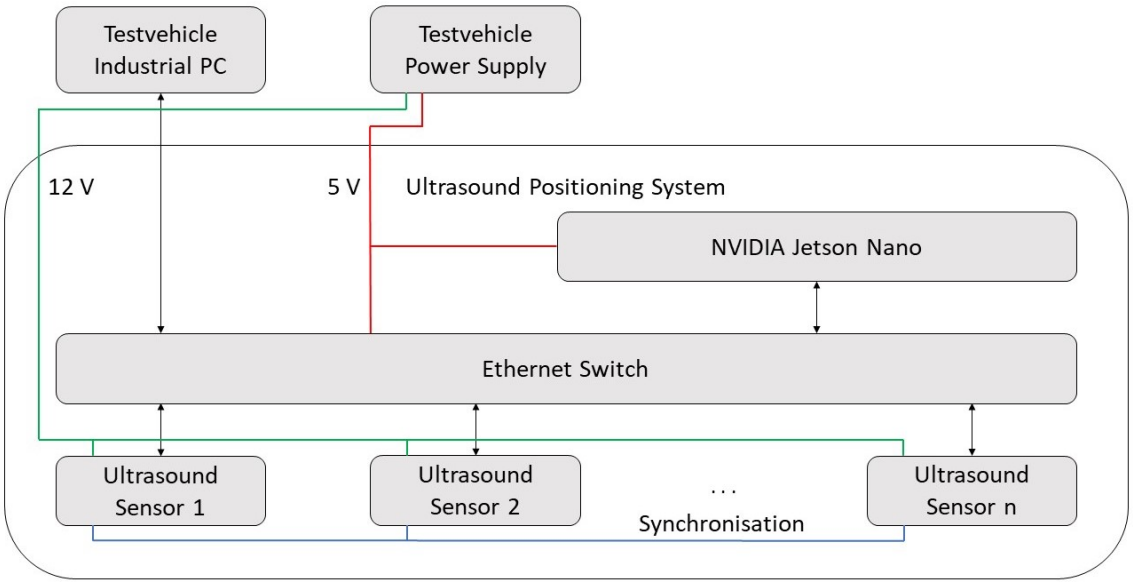


Abbildung 5.7.: Definierte Architektur des ultraschallbasierten Positionierungssystems

Hierbei wird weiterhin Ethernet als zentrales Kommunikationsnetz verwendet. Über das Ethernetnetzwerk erfolgt dabei sowohl die Kommunikation zwischen der eingebetteten zentralen Recheneinheit, dem *NVIDIA Jetson Nano*, und den Ultraschallsensoren, als auch zwischen der eingebetteten zentralen Einheit und dem Industrie-PC, auf dem das UVPS läuft. Des Weiteren wird für das Ethernetnetzwerk die Sterntopologie des Ultraschallsensorsystems aus Abschnitt 2.2.1 beibehalten. Hierbei dient ein Ethernet-Switch als zentraler Sternpunkt.

Ebenfalls werden die Ultraschallsensoren in variabler Anzahl und die Synchronisation der Ultraschallsensoren in die Architektur integriert.

Des Weiteren werden die externen Spannungsversorgungen in die Architektur integriert. Hierbei werden das *NVIDIA Jetson Nano* Board und der Ethernet-Switch mit $5\,V$ und die Ultraschallsensoren mit $12\,V$ versorgt. Dabei wird die $5\,V$ Spannungsversorgung zunächst mithilfe eines Akkus realisiert.

5.2.3. Ultraschallsensoranordnung

Um das UPS einfach zu halten, werden für die Umsetzung zunächst zwei unterschiedliche Sensoranordnungen verwendet. Hierdurch ist es möglich die Eigenschaften der Positions- und Orientierungsbestimmung mithilfe von Ultraschall zu erkennen und zu analysieren.

Hierzu wird zunächst das System mit nur einem Sensor aufgebaut. Dabei wird der Sensor an der Front auf der Längsachse des Fahrzeugs platziert. Bei dieser Sensoranordnung dürfte es zu Vieldeutigkeiten in der Positions- und Orientierungsbestimmung kommen. So ist zum Beispiel eine solche Vieldeutigkeit der Raytracing-Simulation in Abbildung 5.8 zu sehen.

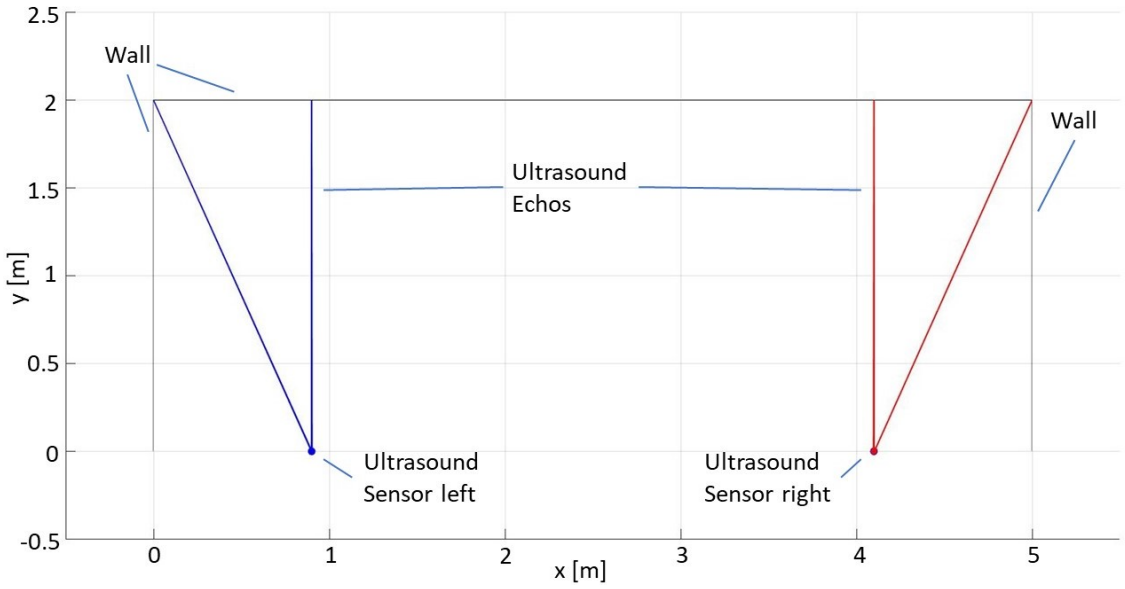


Abbildung 5.8.: Ultraschall-Raytracing Vergleich bei Verwendung eines Sensors

Hierbei ist der Ultraschallsensor bei der ersten Messung in Richtung der rechten Ecke und bei der zweiten Messung in Richtung der linken Ecke platziert. Dabei stellen die roten beziehungsweise blauen Linien die jeweils zurückkommenden Ultraschallechos dar. Obwohl die Ultraschallechos zum Teil aus unterschiedlichen Richtungen beim Sensor ankommen, sieht der zeitliche Verlauf der Messungen identisch aussehen (siehe Abbildung 5.9). So befinden sich zum Beispiel in einem quadratischen Raum, in dem sich keine anderen Objekte befinden, mit dieser Sensoranordnung pro Position jeweils sieben weitere Positionen mit einer identischen Ultraschallechosignatur.

Eine weitere Ultraschallsensoranordnung besteht aus zwei Ultraschallsensoren. Hierbei bleibt die Position des ersten Sensors erhalten. Der zweite Sensor wird 25 cm rechts neben dem erste platziert. Des Weiteren wird der zweite Sensor um 45° nach rechts verdreht.

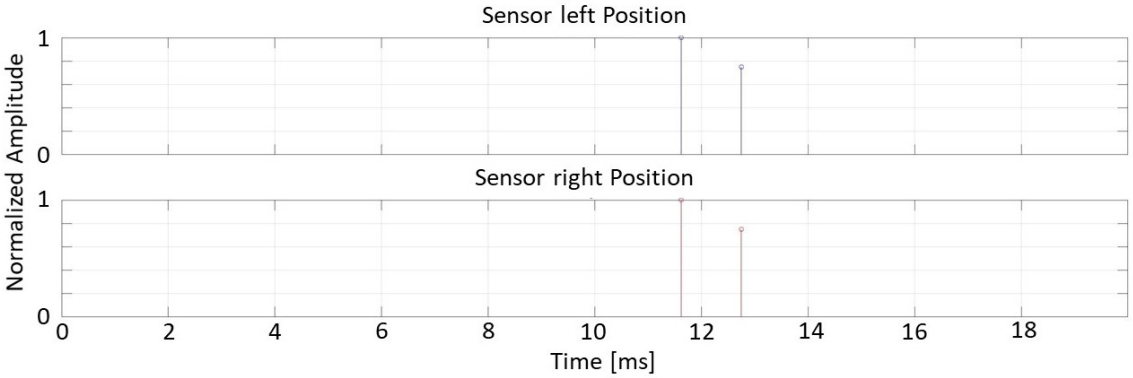


Abbildung 5.9.: Vergleich des zeitlichen Verlaufs der Ultraschallechos bei Verwendung eines Sensors

Durch eine solche asymmetrische Anordnung, bei der beide Sensoren separat einen Ultraschallpuls aussenden, wird die Anzahl an Vieldeutigkeiten pro Position und Orientierung reduziert. In Abbildung 5.10 sind die zurückkommenden Ultraschallechos zu sehen.

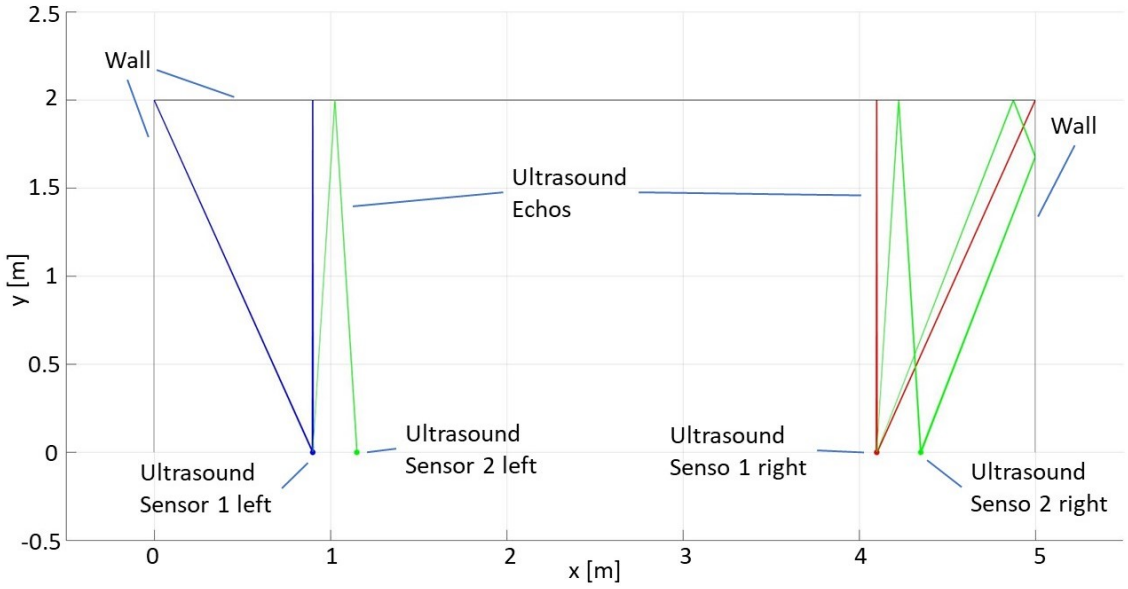


Abbildung 5.10.: Ultraschall-Raytracing Vergleich bei Verwendung von zwei asymmetrisch angeordneten Sensoren

Hierbei wird der Ultraschallpuls durch den ersten mittig platzierten Ultraschallsensor ausgesendet. So kann in dem Beispiel des quadratischen Raums unterschieden werden, ob die Sensoren in Richtung der rechten oder linken Ecke platziert sind, da mithilfe des zweiten Sensors in Richtung der rechten Ecke eine andere Ultraschallechosignatur empfangen wird

(siehe Abbildung 5.11). Hierbei reduziert sich bei einer Position die Anzahl möglicher anderer Positionen auf drei mit identischer Ultraschallechosignatur.

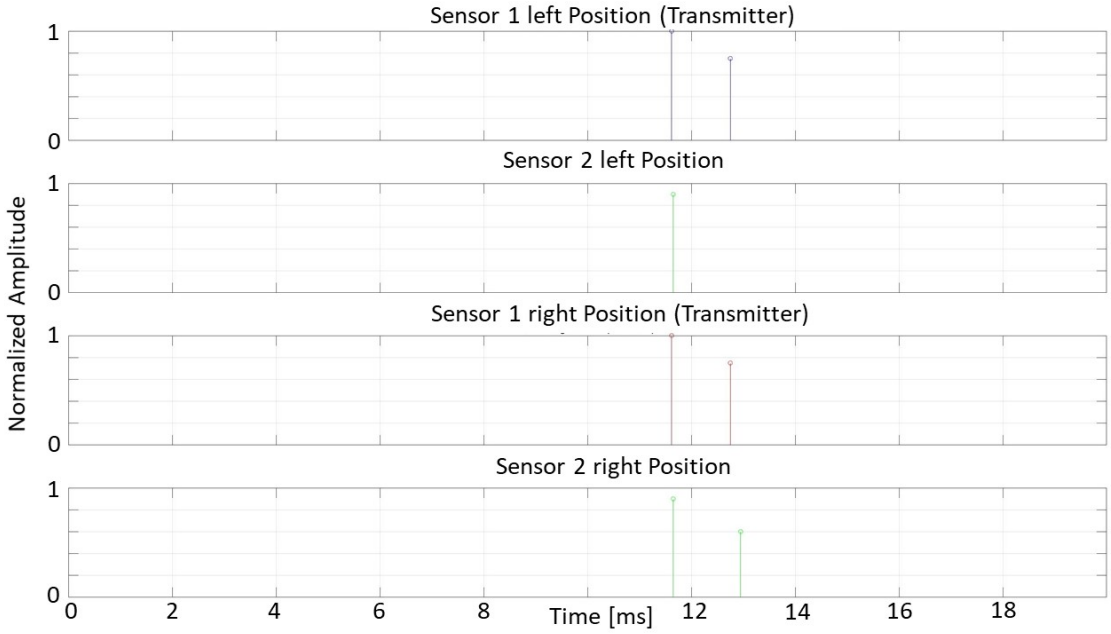


Abbildung 5.11.: Vergleich des zeitlichen Verlaufs der Ultraschallechos bei Verwendung von zwei Sensoren

5.2.4. Software des ultraschallbasierten Positionierungssystems

Die Realisierung der Software erfolgt aufbauend auf dem Meta-Betriebssystem ROS (vgl. ROS, 2020). Da ROS für modulare Systeme, bei denen die einzelnen Nodes mithilfe von Topics kommunizieren, ausgelegt ist, wird die Gesamtfunktionalität der Software in mehrere Softwaremodule unterteilt (siehe Abbildung 5.12). Hierbei erfolgt die Realisierung in den Programmiersprachen C++ und Python (siehe Anhang D). Die Programmabläufe sind zu den einzelnen Softwaremodulen in Anhang B.2 zu sehen. Des Weiteren wird als Betriebssystem der eingebetteten zentralen Recheneinheit ein von *NVIDIA* auf die ARM-Architektur angepasstes *Ubuntu 18.04* verwendet (vgl. NVIDIA, 2020b). Hierdurch fällt die Auswahl der ROS-Version auf *Melodic Morenia* (vgl. ROS, 2020).

Hierbei werden in den einzelnen Softwaremodulen mögliche Fehler abgefangen und behandelt. Um andere Systeme nicht zu beeinträchtigen oder die Fahrzeugposition zu verfälschen, wird das ultraschallbasierte Positionierungssystem im Fehlerfall beendet.

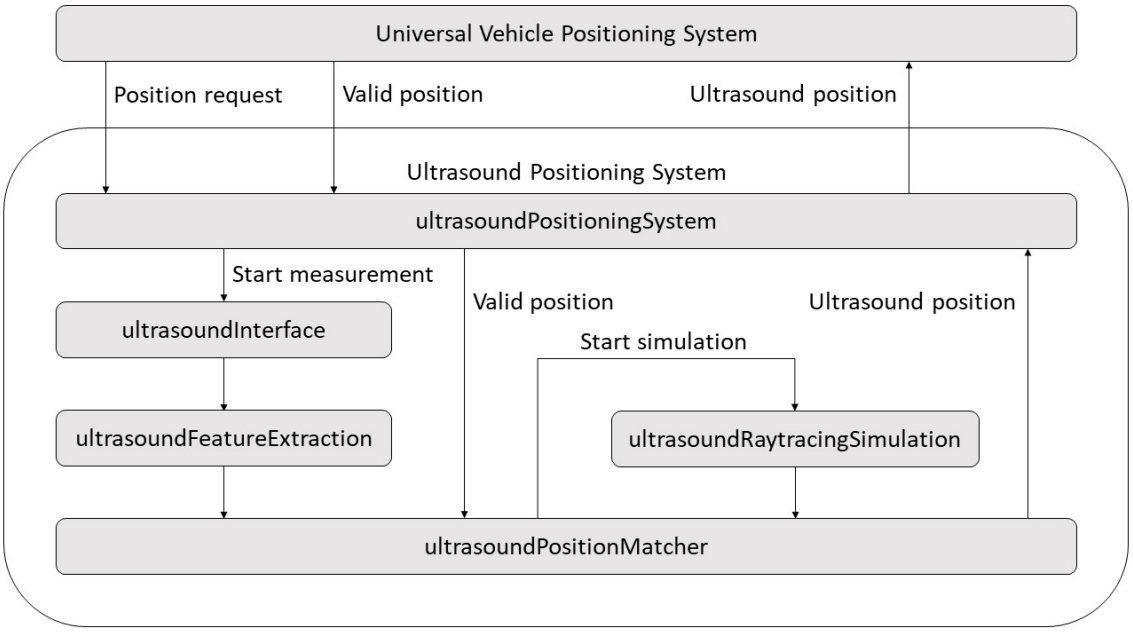


Abbildung 5.12.: Softwarearchitektur des ultraschallbasierten Positionierungssystems

Systemschnittstelle

Der Node *ultrasoundPositioningSystem* dient als Systemschnittstelle und übernimmt die Kommunikation mit dem UVPS. Des Weiteren steuert der Node das UPS.

Hierzu führt der Node beim Start des Systems einen Test der anderen Softwaremodule auf fehlerfreie Funktion durch. Bei einer fehlerfreien Funktion wird das System frei gegeben. Erfolgt vom UVPS eine Positionsanfrage, wird die Positionsbestimmung gestartet, indem die Ultraschallsensorschnittstelle aktiviert wird. Ebenfalls wird die gültige Fahrzeugposition, die vom UVPS mitgeteilt wird, im System aktualisiert. Des Weiteren übermittelt der Node die mithilfe von Ultraschall bestimmte Fahrzeugposition an das UVPS.

Ultraschallsensorschnittstelle

Das Softwaremodul *ultrasoundInterface* stellt die Ultraschallsensorschnittstelle zu den Ultraschallsensoren bereit, womit diese gesteuert und ausgewertet werden.

Bei einer angeforderten Ultraschallmessung startet das Modul einen Messdurchlauf der Ultraschallsensoren. Die ankommenden Ultraschallechos der einzelnen Ultraschallsensoren werden dabei überprüft und sortiert. Bei fehlerfreien Daten werden diese an die Ultraschallmerkmalsextraktion weitergegeben.

Ultraschallmerkmalsextraktion

Der *ultrasoundFeatureExtraction* Node verarbeitet die mit der Ultraschallsensorschnittstelle empfangenen Ultraschallechos und dient der Merkmalsextraktion.

Zur Verarbeitung der Ultraschallechos wird bei diesen zunächst der vom Ultraschallmodul der Ultraschallsensoren vorhandene Offset herausgerechnet. Anschließend werden die Ultraschallechos normalisiert. Des Weiteren werden die Echos der einzelnen Ultraschallsensoren zu einer Ultraschallsignatur zusammengefügt (vgl. Browne und Kleeman, 2009, S. 4043). Abschließend erfolgt die Weitergabe der Ultraschallsignatur an die Positionsbestimmung.

Des Weiteren ist dieser Node für eine mögliche Erweiterung um weitere Merkmalsextraktionen vorbereitet.

3D-Raytracing-Simulation

Das Modul *ultrasoundRaytracingSimulation* basiert auf der Simulation aus Abschnitt 2.2.2, welche auf drei Dimensionen erweitert und an die Anforderungen des UPS angepasst wird. Da ROS *MATLAB* als Programmiersprache nicht unterstützt, erfolgt die Implementierung mithilfe von Python (vgl. ROS, 2020).

Die Struktur der Simulation bleibt dabei bestehen. Hierbei werden zunächst bei den einzelnen Ultraschallstrahlen Objekttreffer und deren Refelxionsparameter berechnet. Anschließend erfolgt die Überprüfung, ob ein Ultraschallsensor von dem Strahl getroffen wird. Werden mehrere Treffer ermittelt, wird der Treffer mit der kürzesten Entfernung als gültig gewertet. Wird ein Sensor von einem Strahl getroffen, wird die Pfadberechnung des jeweiligen Strahls beendet. Ebenfalls beendet wird die Pfadberechnung, wenn die zurückgelegte Strahllänge einen sensorspezifischen Maximalwert überschreitet. So beträgt die maximale Strahllänge für das Ultraschallmodul zum Beispiel die doppelte Reichweite, also 21,36 m. Dieser Ablauf wird iterativ für jeden Ultraschallstrahl durchgeführt.

Der zu simulierende Bereich wird als objektbasierte Karte realisiert. Hierbei werden die einzelnen Objekte, wie zum Beispiel Wände, mithilfe von Ebenen in der Parameterdarstellung modelliert. Der Detaillierungsgrad der Karte stellt dabei einen Kompromiss dar. Mit einem niedrigeren Detaillierungsgrad wird eine kürzere Rechenzeit ermöglicht. Wohingegen eine Karte mit einem höheren Detaillierungsgrad die Simulationsergebnisse eine höhere Übereinstimmung mit der Ultraschallmessung aufweisen.

Die Anzahl der Strahlen, auf die der Ultraschallpuls aufgeteilt wird, errechnet sich aus den sensorspezifischen Parametern. Hierzu wird zunächst die Schrittweite w mithilfe der minimalen Objektgröße x, die noch detektiert werden soll, berechnet (siehe Formel 5.2). Die Schrittweite muss kleiner gleich der halben minimalen Objektgröße sein, um das Abtasttheorem einzuhalten. Anschließend erfolgt die Berechnung der Winkelauflösung α_R (siehe

Abbildung 5.13). Diese werden mit der Schrittweite und der maximalen Distanz d, an der Objekte mit der minimalen Objektgröße erkannt werden sollen, ermittelt (siehe Formel 5.3). Abschließend wird die Schallkeule des Ultraschallpulses mit der Winkelauflösung auf die einzelnen Strahlen aufgeteilt. Trifft man die Annahmen, dass Objekte mit einer Größe von $10\,cm$ in einer Entfernung von $6\,m$ noch detektiert werden sollen, beträgt die Winkelauflösung $0{,}0083\,rad$. Mit der Schallkeulengrenze des Ultraschallmoduls von ca. 33,7° (vgl. MaxBotix, 2015, S. 14) kommt man auf eine Anzahl von 16533 Strahlen.

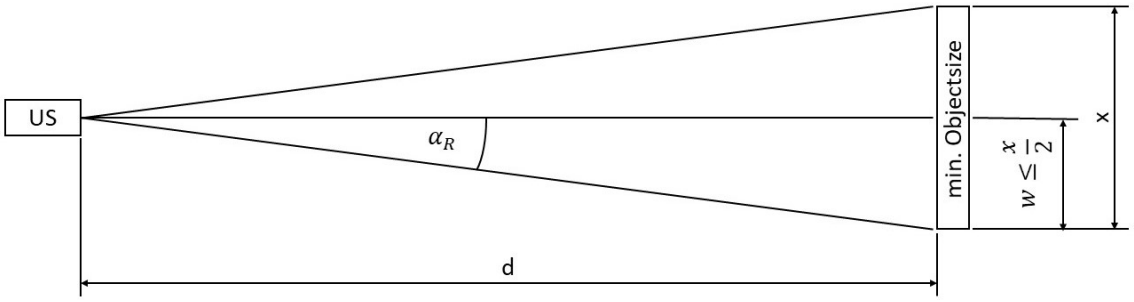


Abbildung 5.13.: Berechnung der Winkelauflösung der Ultraschallstrahlen

$$w \leq \frac{x}{2} \tag{5.2}$$

$$\alpha_R = arctan\left(\frac{w}{d}\right) \tag{5.3}$$

Des Weiteren wird für die Implementierung der Simulation die Annahme getroffen, dass das Fahrzeug eben auf dem Boden steht. Hierdurch bleiben die Höhe der Fahrzeugposition und der Elevationswinkel des Fahrzeugs zum Boden konstant, was eine Reduzierung der zu simulierenden Positionen zur Folge hat. Daraus folgt eine deutlich geringere Rechenlast bei der Bestimmung von Position und Orientierung.

Eine weitere Anpassung betrifft die Berechnung der Abstrahlcharakteristik des Ultraschallsenders. In Abbildung 5.14 ist der Vergleich der Abstrahlcharakteristik des Ultraschallmoduls mit der berechneten Abstrahlcharakteristik der Raytracing-Simulation zu sehen. Hierbei zeigt die grüne Kurve die Messkurve des Ultraschallmoduls und die blaue die berechnete Abstrahlcharakteristik der Formel 2.11 aus Abschnitt 2.1.3.3. Es ist zu sehen, dass die Berechnung deutlich von der Messung abweicht. Hierzu wird die Berechnung soweit angepasst, dass die Formel 2.11 auf einen Anteil von 20 % reduziert wird. So stellt die orangene Kurve, die optimierte Berechnung der Abstrahlcharakteristik, einen guten Kompromiss dar. Da der optimierte Ultraschallsensor einen Teil der Ultraschallechos durch das Gehäuse abschirmt, wird das Empfangen von Ultraschallechos in der Simulation auf $\pm 90°$ begrenzt.

Des Weiteren wird, da ein 3D-Raytracing rechenintensiv ist, die Simulation mithilfe von Multiprocessing auf mehrere Prozesse aufgeteilt. Hierbei wird eine dynamische Anzahl an Pro-

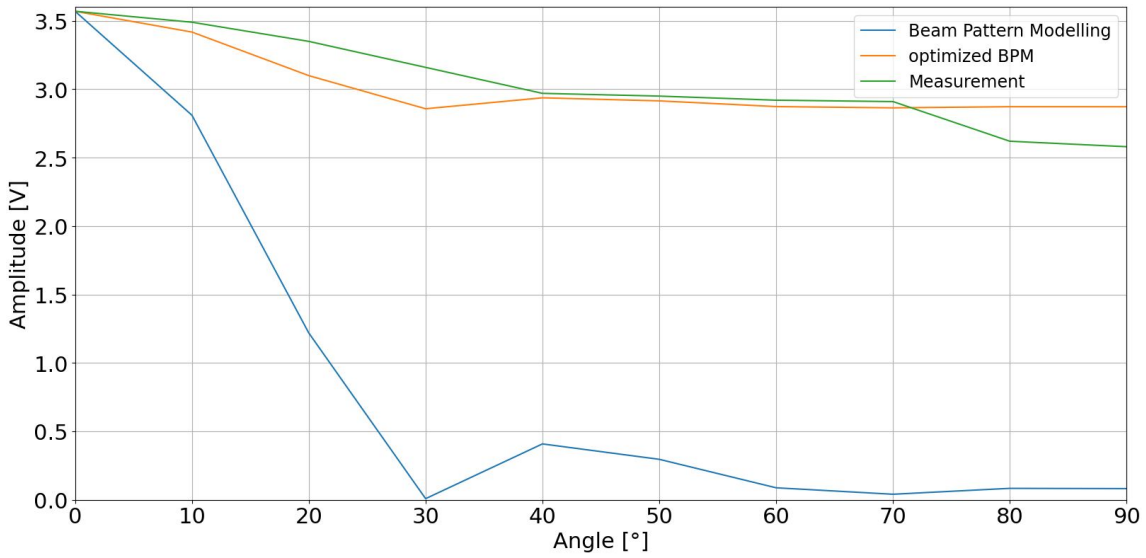


Abbildung 5.14.: Vergleich der Abstrahlcharakteristik des Ultraschallmoduls mit der Simulationsberechnung

zessen erzeugt, die sich an den verfügbaren Prozessorkernen orientiert. Eine Optimierung im Hinblick auf eine GPU-Auslagerung der Berechnungen ist bei der vorhandenen Struktur der Simulation nicht möglich. Hierbei ist durch eine GPU-Auslagerung eine hohe Anzahl an Arbeitsspeicherzugriffen erforderlich, wodurch eine kürzere Berechnungszeit nicht ermöglicht wird. Ein weiterer Punkt ist die Berechnungsdauer der einzelnen Strahlen. Hierbei benötigen die Strahlen eine unterschiedliche Anzahl an Rechenschritten. Bei einer GPU-Auslagerung werden alle Strahlen mit derselben Anzahl an Rechenschritten berechnet. Das heißt, auch wenn einzelne Strahlen ein Abbruchkriterium erreichen, werden diese weiter mitberechnet, bis der letzte Strahl das Abbruchkriterium erreicht hat.

Da eine Simulation von mehreren Positionen zeitintensiv ist und damit die geforderte Messfrequenz des Gesamtsystems nicht erreicht werden kann, ist die Simulation durch eine Look-Up-Tabelle austauschbar. Hierbei werden die Ergebnisse der einzelnen Positionen im Vorfeld berechnet und abgespeichert.

Das Ergebnis der Raytracing-Simulation ist zum einen der zeitliche Verlauf der zurückkommenden Ultraschallechos. Diese bestehen je Ultraschallecho aus einem Tupel mit Zeitangabe und Amplitude. Des Weiteren wird für jedes zurückkommenden Ultraschallecho der Strahlpfad ausgegeben, der visualisiert werden kann.

Positionsbestimmung

Die Positionsbestimmung wird durch den Node *ultrasoundPositionMatcher* durchgeführt. Dieser führt dabei den Vergleich der Ultraschallmessung mit der 3D-Raytracing-Simulation durch. Hierzu startet der Node nach Erhalt der verarbeiteten Ultraschallsignatur die Simulation und handelt die Simulationsergebnisse. Die eigentliche Bestimmung von Position und Orientierung erfolgt dabei innerhalb des Nodes mithilfe des Algorithmus zur Positionsbestimmung. Hierbei wird unter anderem die gültige Position des Fahrzeugs verwendet. Die Positionsparameter bestehen dabei aus der Position, der Orientierung und einem Genauigkeitswert. Abschließend erfolgt die Übermittlung der Positionsparameter an die Systemschnittstelle.

5.2.5. Algorithmen zur ultraschallbasierten Positionsbestimmung

Der Algorithmus führt den Vergleich zwischen Ultraschallmessung und Simulation durch, um somit eine Bestimmung von Position und Orientierung zu realisieren. Hierzu wird die Karte der Raytracing-Simulation in ein festes Raster aufgeteilt. Des Weiteren wird für die Orientierung eine feste Schrittweite implementiert. Dies minimiert die Anzahl an möglichen Positionen und reduziert somit den Rechenaufwand, um die Position und Orientierung zu bestimmen. Ein weiterer Punkt ist, dass hierdurch die Größe der Look-Up-Tabelle begrenzt wird.

Um die Simulationsergebnisse mit der Ultraschallmessung vergleichen zu können, wird aus den Ultraschalltupeln der Simulation eine Ultraschallsignatur erzeugt. Hierzu werden die Tupel zu einem zeitlichen Verlauf zusammengefügt und mit einer Hüllkurve eingehüllt. Abbildung 5.15 zeigt unterschiedliche Hüllkurven. Hierbei zeigt die blaue Kurve jeweils ein Ultraschallecho. Die orangenen Kurven zeigen unterschiedliche Hüllkurven, mit denen einzelne Peaks eingehüllt werden. Bei den Hüllkurven handelt es sich um eine cos-Funktion, eine Gauß-Funktion und eine cos^2-Funktion. Es ist zu erkennen, dass bei einzelnen Peaks der cos^2-Funktion die größte Übereinstimmung mit dem Ultraschallecho aufweist. Bei verrauschten Peaks, wenn sich mehrere Ultraschallechos überlagern, weist die cos-Funktion hingegen die größte Übereinstimmung auf. Da der Fokus auf der Detektion von mehreren zurückkommenden Ultraschallechos liegt, wird als Hüllkurve die cos-Funktion verwendet. Ein weiterer Punkt ist der Offset des Ultraschallmoduls, wodurch die Wurzeln der Peaks nicht sichtbar sind. Hierdurch weist die cos-Funktion in den Bereichen des Anfangs beziehungsweise Endes der einzelnen zurückkommenden Ultraschallechos eine höhere Übereinstimmung auf.

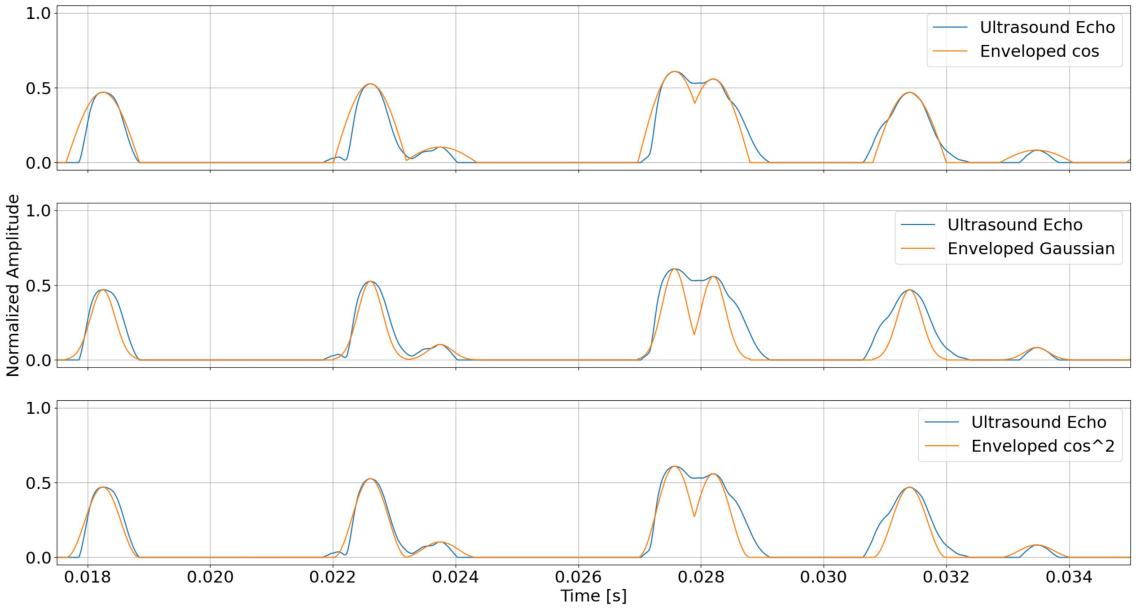


Abbildung 5.15.: Vergleich der Hüllkurven mit einem Ultraschallecho

Um die Signatur der eingehüllten Simulation mit der Signatur der Ultraschallsensoren zu vergleichen, wird eine Kreuzkorrelation verwendet. Hierbei werden die Signaturen pro Messdurchlauf einzeln mithilfe der Formel 5.4 verglichen (vgl. Ohm und Lüke, 2014, S. 215). Nach jeder Korrelation wird das Maximum in dem Korrelationsergebnis ermittelt, um mit diesem das t, der Position des Maximums, zu bestimmen. Da bei zwei Signalen, die zu 100% übereinstimmen, das Maximum der Korrelation in der Mitte der Zeitreihe liegt, beschreibt hierbei das t die Wahrscheinlichkeit der Position. Mit dem t und einer Gauß-Funktion wird anschließend im nächsten Schritt die Wahrscheinlichkeit bestimmt, mit der die verglichene Position bewertet werden kann. Abschließend wird aus den Wahrscheinlichkeiten der einzelnen Messdurchläufe eine Gesamtwahrscheinlichkeit berechnet. Dabei werden zur Positions- und Orientierungsbestimmung mehrere mögliche Positionen simuliert und mit der Ultraschallsignatur verglichen. Die Position mit der größten Gesamtwahrscheinlichkeit wird am Ende der Simulationen als gefundene Position gewertet. Abbildung 5.16 zeigt hierbei die Gauß-Funktion bei einer Zeitreihenlänge von 10000. Die optimale Übereinstimmung sollte dabei mit t bei 5000 liegen.

$$R_{xy}[n] = \sum_{\infty}^{m=-\infty} x^*[m]y[m+n] \tag{5.4}$$

Es werden zwei Strategien, mit der die möglichen Positionen, die simuliert und mit der Ultraschallsignatur verglichen werden, implementiert. Hierbei handelt es sich um eine globale Positionsbestimmung und einer Positionsvorhersage.

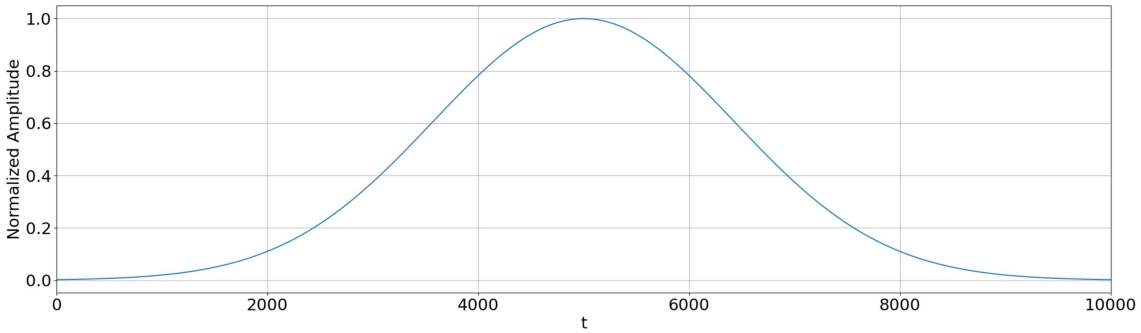


Abbildung 5.16.: Gauß-Funktion zur Bestimmung der Wahrscheinlichkeit einer gefundenen Position

Globale Positionsbestimmung

Bei der globalen Positionsbestimmung werden alle Positionen mit dazugehörigen Orientierungen der Karte simuliert und mit der Ultraschallsignatur verglichen. Hierbei wird die gültige beziehungsweise letzte bekannte Fahrzeugposition nicht zur Bestimmung der neuen Position und Orientierung verwendet. Ebenfalls findet keine Überprüfung auf Plausibilität der neuen Position und Orientierung statt. Hierdurch können die generelle Eigenschaft des UPS und die Auswirkungen der Vieldeutigkeiten auf dieses analysiert werden.

Positionsvorhersage

Die Positionsvorhersage wählt mögliche Positionen und Orientierungen aus, um diese zu simulieren und mit der Ultraschallsignatur zu vergleichen. Hierzu wird die letzte gültige beziehungsweise bekannte Fahrzeugposition verwendet. Des Weiteren ist hier nach einem Systemstart, wenn noch keine Fahrzeugposition vorhanden ist, ein hinterlegter Startpunkt nötig. Hierdurch wird die Anzahl der zu simulierenden und zu vergleichenden Positionen deutlich reduziert. Ebenfalls werden durch diese Vorgehensweise die Auswirkungen der Vieldeutigkeiten auf die Positions- und Orientierungsbestimmung reduziert. Bei dieser Vorgehensweise ist es nötig, dass bei einer falsch bestimmten Position und Orientierung, diese korrigiert werden, um eine zuverlässige Positions- und Orientierungsbestimmung zu realisieren.

So zeigt Abbildung 5.17 eine schematische Darstellung der Positionsvorhersage. Die blauen Kurven stellen dabei die möglichen Trajektorien des Fahrzeugs bei einem Positionswechsel dar. So kann in dem Beispiel das Fahrzeug nur die drei linken und drei rechten Positionen anfahren. Des Weiteren ist es möglich, dass das Fahrzeug auf der letzten bekannten Position stehen bleibt. Die grünen Pfeile zeigen hierbei die möglichen Orientierungen an, die das Fahrzeug auf den jeweiligen Positionen annehmen kann.

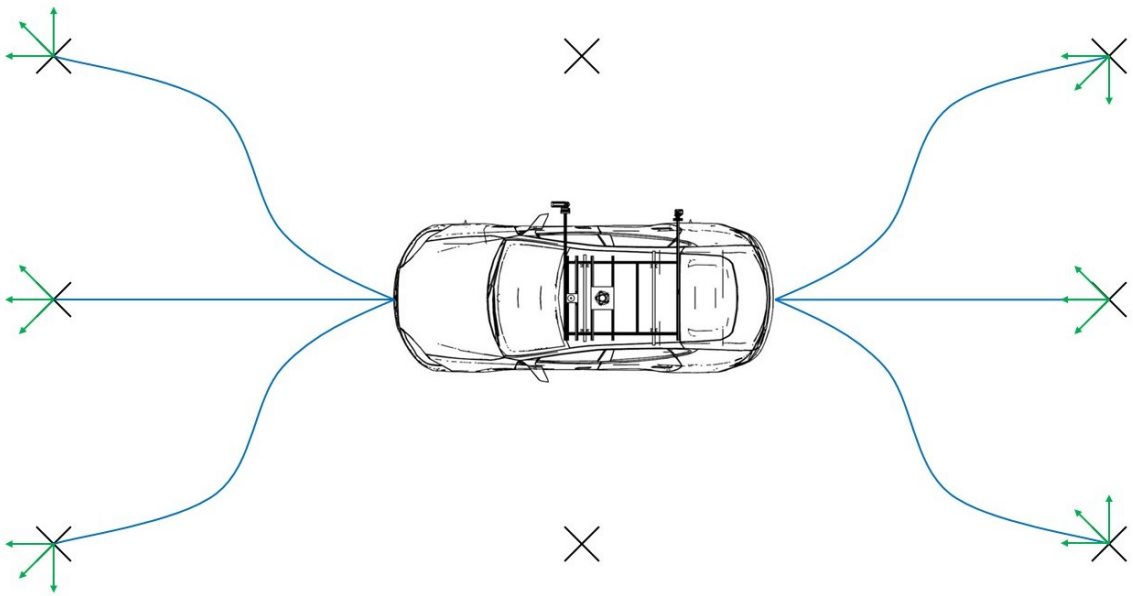


Abbildung 5.17.: Schematische Darstellung der Positionsvorhersage

5.3. Umfeldsensorbasiertes Navigationssystem

Aufbauend auf den Systementwurf wird die Architektur des UVPS weiter definiert. Hierzu wird parallel zum GNSS/INS-System des Versuchsfahrzeugs das UPS in das UVPS integriert (siehe Abbildung 5.18).

Des Weiteren wird eine Rückführung der Positionsdaten an das UPS hinzugefügt. Hiermit ist eine Korrektur der Position möglich und gegebenenfalls falsch detektierte Positionen üben keine große Auswirkung auf die Bestimmung von Position und Orientierung aus.

Das umfeldsensorbasierte Navigationssystem in Form des *Universal Vehicle Positioning System* stellt auf dem aktuellen Stand eine Schnittstelle zum UPS dar (siehe Anhang E). Eine Erweiterung um weitere umfeldsensorbasierte Positionierungssysteme ist möglich. Ebenfalls ist eine Erweiterung um eine Sensordatenfusion innerhalb des UVPS möglich, um zum Beispiel die GNSS/INS-Position mit dem UPS zu verbessern.

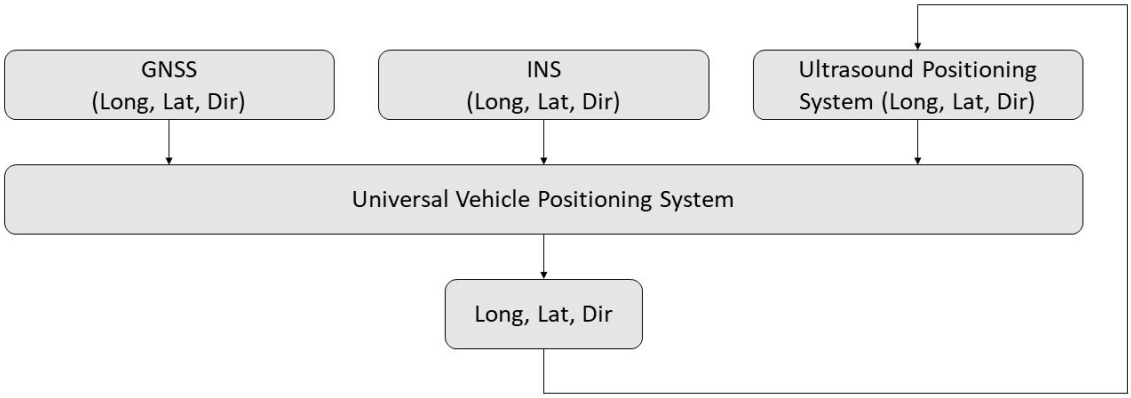


Abbildung 5.18.: Architektur des umfeldsensorbasierten Navigationssystems

6. Test und Bewertung

Um die Einsatzfähigkeit des umfeldsensorbasierten Navigationssystems darzustellen, wird zunächst der Ultraschallsensor getestet. Anschließend erfolgt ein Test des ultraschallbasierten Positionierungssystems. Des Weiteren erfolgt ein Funktionalitätstest des Gesamtsystems anhand eines Anwendungsfalls. Hierbei orientieren sich die Tests an den Anforderungen an das System. Den Abschluss des Kapitels bildet die Bewertung des umfeldsensorbasierten Navigationssystems.

6.1. Ultraschallsensor

Die Tests des optimierten Ultraschallsensors beschreiben die funktionalen Eigenschaften. Zunächst wird das zeitliche Verhalten der Ultraschallsensoren ermittelt. Abschließend wird die Spannungsversorgung des Ultraschallsensors beurteilt.

Die zeitliche Analyse befasst sich mit der Synchronisation der Ultraschallsensoren untereinander. Des Weiteren wird die Dauer eines Messdurchlaufs analysiert. Hierbei erfolgen die Tests mit zwei Ultraschallsensoren, die mit der Synchronisationsringleitung verbunden sind. Die Ermittlung des zeitlichen Verhaltens erfolgt dabei mit einem *PicoScope* (vgl. PicoTechnology, 2020).

Synchronisation der Ultraschallmessung

Zunächst wir die Synchronisation zum Starten und Beenden einer Messung getestet. Die Zeit, die die Open-Collector Schaltung vom softwaretechnischen Ansteuern bis zum Schalten der Synchronisationsleitung benötigt, beträgt 2,46 μs. Der Jitter bewegt sich im Bereich von einigen wenigen ns und ist damit vernachlässigbar klein. Da der Timer zur Zeitmessung erst mit dem Interrupt der Synchronisationsleitung gestartet wird, ist diese Verzögerung vernachlässigbar. In Abbildung 6.1 ist die zeitliche Abweichung beim Starten einer Messung der Ultraschallsensoren untereinander zu sehen. Diese beträgt 0,5 μs, wobei hier der Jitter ebenfalls vernachlässigbar klein ist. Hierbei stellt der obere Verlauf den Ultraschallsensor dar, der die Messung startet. Es ist zu erkennen, dass der auslösende Sensor den Interrupt

für den Messungsstart verzögert auslöst. So entsteht bei einer angenommenen Raumtemperatur von 20 $^{\circ}C$ beim Starten zwischen den Sensoren eine Distanzabweichung von 171,7 μm (siehe Formel 6.1).

$$E_D = c_{Luft} \Delta t = 171,7\,\mu m \tag{6.1}$$

Beim Beenden einer Messung ist der Ablauf der Verzögerung identisch. Das heißt, der auslösende Ultraschallsensor beendet die Messung nach dem zweiten Ultraschallsensor. Dabei beträgt die zeitliche Verzögerung 1,95 μs und der Jitter 0,46 μs. Da das Ultraschallmodul mit 100 kHz abgetastet wird und somit alle 10 μs, spielt die zeitliche Verzögerung beim Beenden des Messdurchlaufs keine Rolle.

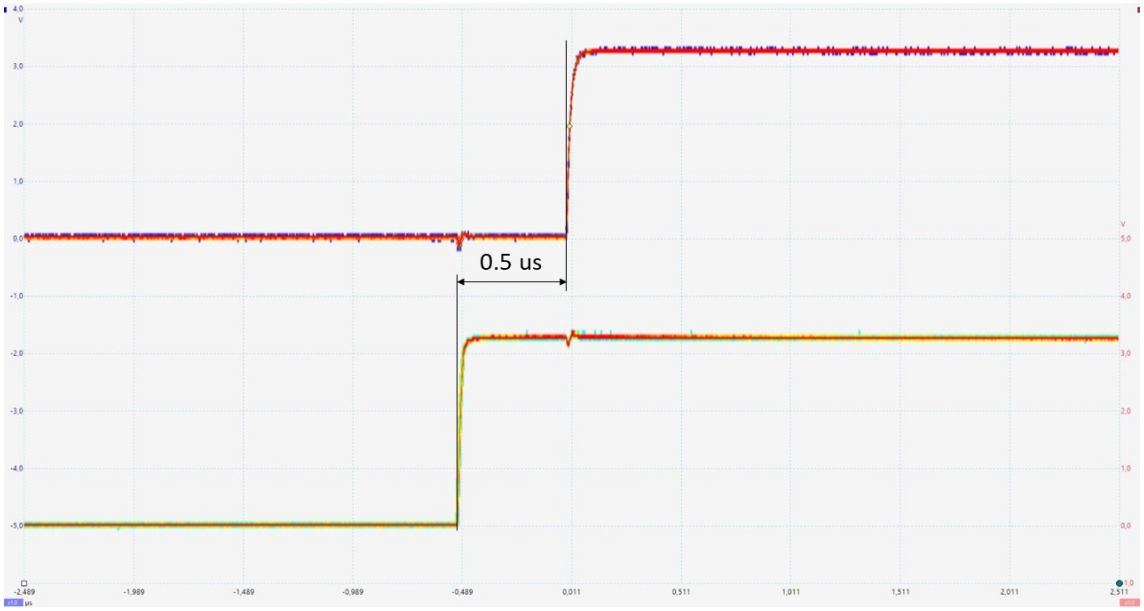


Abbildung 6.1.: Zeitliches Verhalten der Synchronisation des Messungsstarts

Resetten der Ultraschallsensoren

Ein weiterer Test ist die Messung des zeitlichen Verhaltens der Ultraschallsensoren untereinander im Bezug auf das Auslösen eines Resets. Hierbei beträgt die Zeitdifferenz zwischen softwaretechnischen Auslösen und Schalten der Resetleitung 2,37 μs. Der Jitter ist hierbei vernachlässigbar klein. Die zeitliche Differenz zwischen den Ultraschallsensoren beim Auslösen des Interrupts beträgt 0,4 μs und der Jitter 0,08 μs. Da das Resetten der Ultraschallsensoren am Ende eines Messprozesses durchgeführt wird und kein weiterer Messdurchlauf anschließend stattfindet, wirken sich die zeitlichen Abweichungen nicht auf die Messergebnisse aus.

Token Passing

Der Test des Token Passings ermittelt die Zeit, die für dieses benötigt wird. In Abbildung 6.2 stellt der obere Verlauf den Zeitpunkt der Tokenweitergabe des ersten Ultraschallsensors dar. Der untere Verlauf hingegen das Auslösen des Interrupts des zweiten Ultraschallsensors. Hierbei beträgt die Zeitdifferenz 26,1 μs. Der Jitter weist einen Wert von 0,4 μs auf. Da das Token Passing zwischen den Messdurchläufen stattfindet, wirkt sich die zeitliche Differenz nicht auf die Messergebnisse aus. Diese wirkt sich hingegen nur auf die Dauer eines gesamten Messprozesses des Ultraschallsensorsystems aus. Geht man von sechs Ultraschallsensoren an einer Ringleitung aus, so erhöht sich die Durchlaufzeit eines gesamten Messprozesses um maximal 157,8 μs. Da der Messprozess in diesem Fall insgesamt 720 ms dauert, ist die Dauer des Token Passings vernachlässigbar klein.

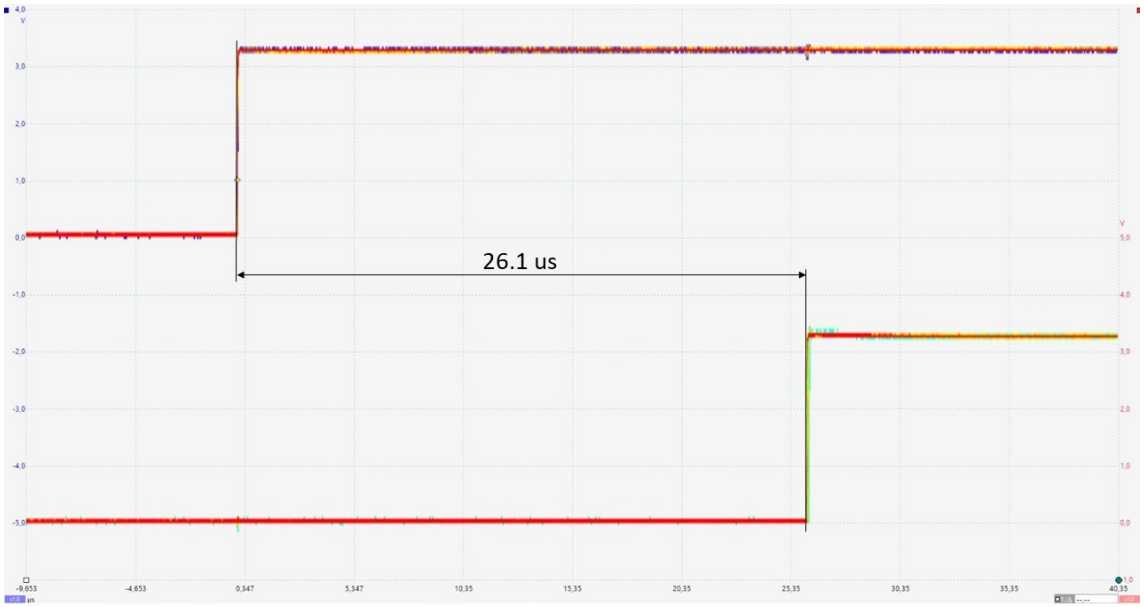


Abbildung 6.2.: Zeitliches Verhalten des Token Passings der Ultraschallsensoren

Dauer eines Messdurchlaufs

Bei der Ermittlung der Dauer eines Messdurchlaufs kommt der Verlauf aus Abbildung 6.3 heraus. Hier zeigt die steigende Flanke den Messungsstart und die fallende Flanke das Ende der Messung. Hierbei beträgt die Dauer 120 ms, wobei diese sich aus 20 ms Kalibrierzeit des Ultraschallmoduls und 100 ms der eigentlichen Messung zusammensetzen. Da die Darstellung sich im ms-Bereich befindet und der Jitter sich im ns-Bereich bewegt, ist dieser vernachlässigbar klein.

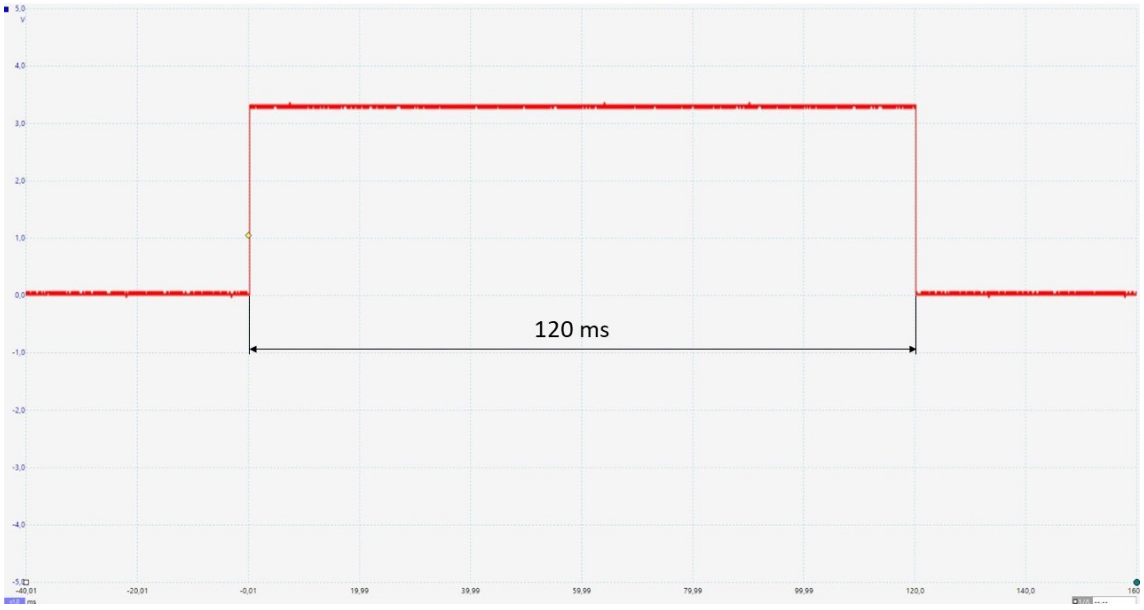


Abbildung 6.3.: Dauer eines Messdurchlaufs der Ultraschallsensoren

Spannungsversorgung

Zur Beurteilung der Spannungsversorgung wird der Strom gemessen, den ein einzelner Ultraschallsensor benötigt. Dies geschieht mit einem *PicoScope* und einem Shunt-Widerstand.

Das Ergebnis der Strommessung ist, dass der Ultraschallsensor im Idle-Zustand bei 12 V Spannungsversorgung ca. 120 mA benötigt. Im Betrieb benötigt der Ultraschallsensor in der Spitze bis zu 167 mA. Bei einer angenommenen maximalen Anzahl von 6 Ultraschallsensoren, die mit einer Ringleitung verbunden sind, bedeutet dies, dass insgesamt ein Maximalstrom von 1 A benötigt wird. Hierdurch wird eine einzelne Ader der Spannungsversorgung mit ca. 333 mA belastet. Da es sich hierbei allerdings um eine Ringleitung handelt, dürft dieser Wert deutlich niedriger liegen. Hierbei ist zu erkennen, dass die Spannungsversorgung eine ausreichend hohe Kapazität aufweist, da die einzelnen Adern mit maximal 1 A belastet werden dürfen.

6.2. Ultraschallbasiertes Positionierungssystem

Die folgenden Tests beschreiben das Verhalten des UPS. Hierzu erfolgt zunächst eine Abstandsmessung von mehreren Objekten. Darauf folgend wird ein Vergleich von einer Ul-

traschallmessung mit der 3D-Raytracing-Simulation durchgeführt. Abschließend erfolgt die Bewertung des zeitlichen Verhaltens des gesamten UPS.

Abstandsmessung

Die Ermittlung des maximalen Abstands erfolgt mit einem Ultraschallsensor und mehreren Objekten (siehe Abbildung 6.4). Als Objekte dienen hierbei zwei Pylonen und eine Trennwand. Die Pylonen sind dabei am seitlichen Rand der Ultraschallkeule platziert. Der Abstand zwischen den Pylonen und dem Ultraschallsensor betragen 4,33 m beziehungsweise 7,12 m. Die Trennwand wird in direkter Flucht mit dem Ultraschallsensor platziert. Mit dieser wird der maximal mögliche Abstand ermittelt, in dem Objekte erkannt werden können.

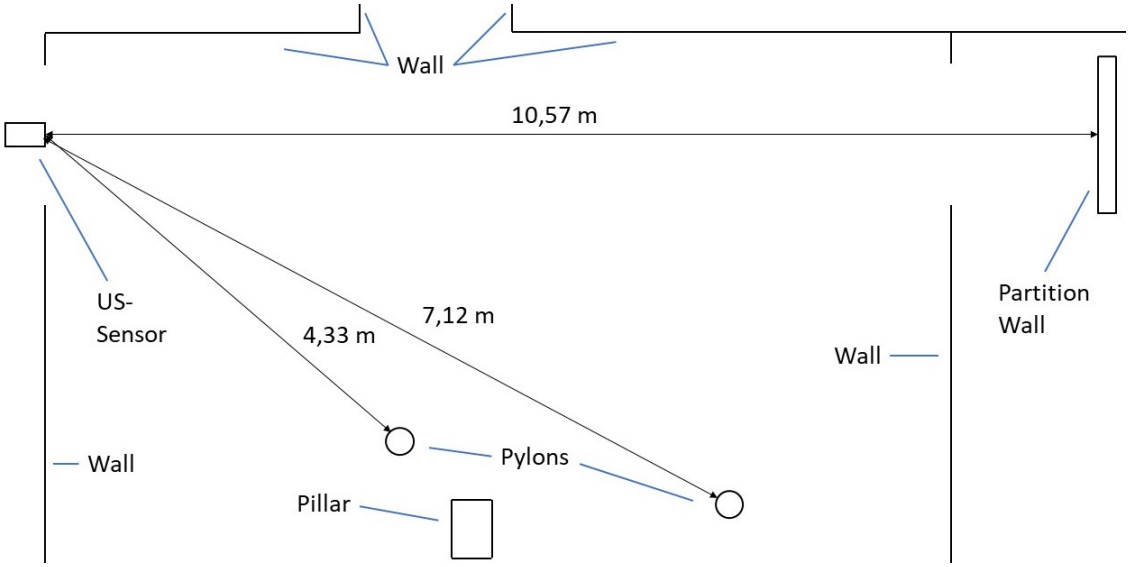


Abbildung 6.4.: Testszenario für die Abstandsmessung

Abbildung 6.5 zeigt das Ergebnis des Tests. Die Peaks mit den Nummern 2 und 6 sind hierbei die Ultraschallechos der zwei Pylonen. Mit den Nummern 1, 3 bis 5 und 7 markierte Peaks stellen Reflexionen von Raumbestandteilen dar, wie zum Beispiel Lampen oder Wände. Der bei diesem Test ermittelte maximale Abstand beträgt 10,57 m. Dieser wird durch die Peaks mit der Nummer 8 dargestellt, wobei der zweite Peak eine Mehrfachreflexion über den Boden ist. Des Weiteren zeigt dieser Test, dass mit dem UPS eine Erkennung von mehreren Objekten und der Umgebung möglich ist.

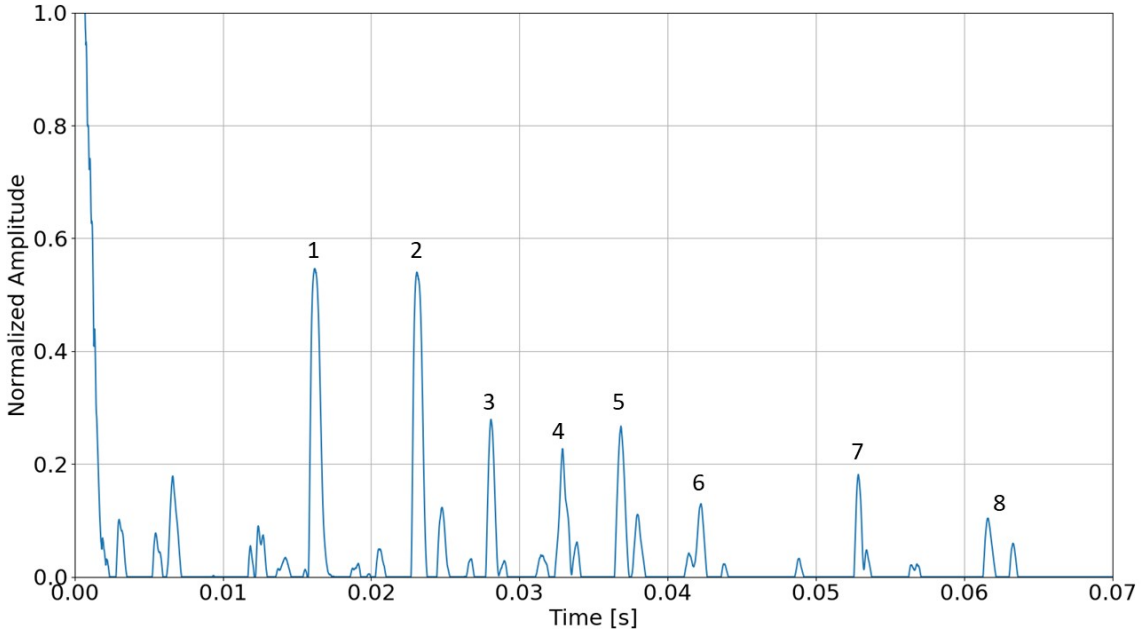


Abbildung 6.5.: Test zur Ermittlung des maximalen Abstands

3D-Raytracing-Simualtion

Der folgende Test vergleicht die Ergebnisse der 3D-Raytracing-Simulation mit den zurückkommenden Ultraschallechos des Ultraschallsensors. Der Test erfolgt dabei mit der ersten Ultraschallsensoranordnung, also einem Sensor. Die Grundabmessung des Testraums ist in Abbildung 6.6 zu sehen. Dieser hat dabei eine Höhe von 2,1 m. Das blaue Kreuz stellt hierbei den Koordinatenursprung dar. Das rote Kreuz hingegen ist die Testposition. Der Ultraschallsensor wird bei diesem Test auf einer Höhe von 1,12 m befestigt. Für die Messung wird eine Raumtemperatur von 21$^{\circ}C$ angenommen. Die minimale Objektgröße, die bei einer Entfernung von 6 m noch erkannt werden soll, beträgt bei diesem Test 1 cm.

In Abbildung 6.7 sind die, für diesen Test, visualisierten Strahlpfade der Simulation zu sehen. Die magentafarbenen Linien sind dabei die simulierten zurückkommenden Ultraschallechos. Hierbei ist zu erkennen, dass die Ultraschallechos nicht nur den direkten Weg nehmen, sondern auch über mehrere Flächen reflektiert werden können.

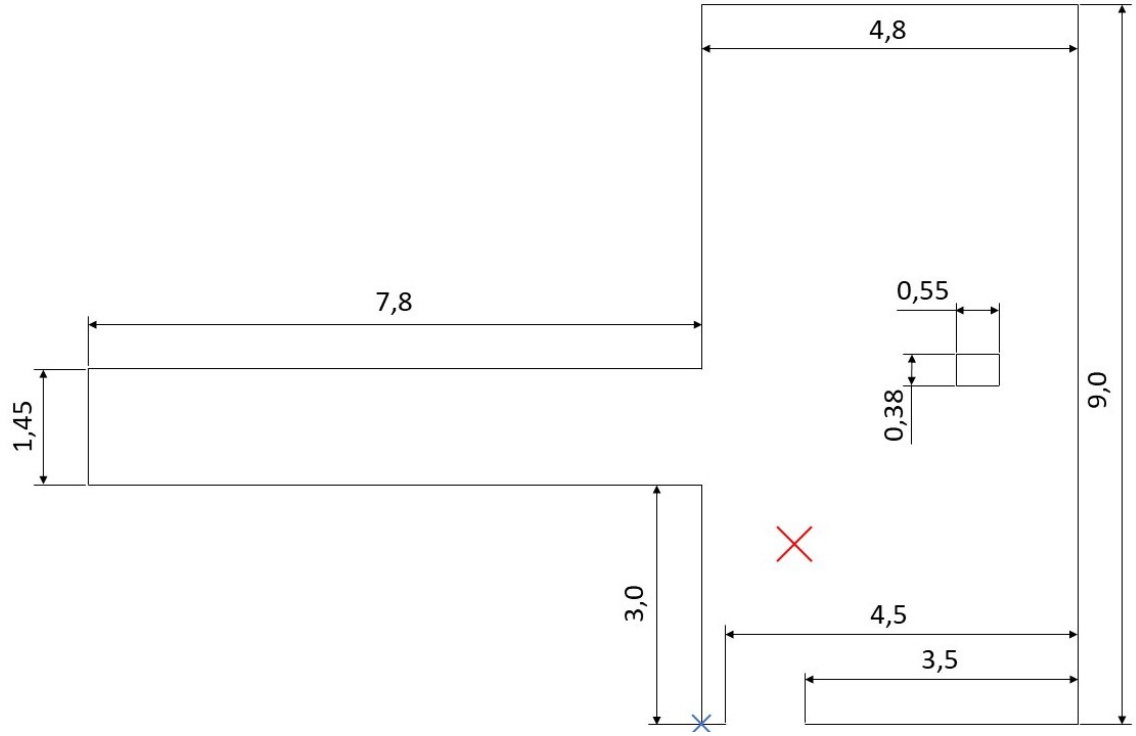


Abbildung 6.6.: Grundabmessung des Testraums

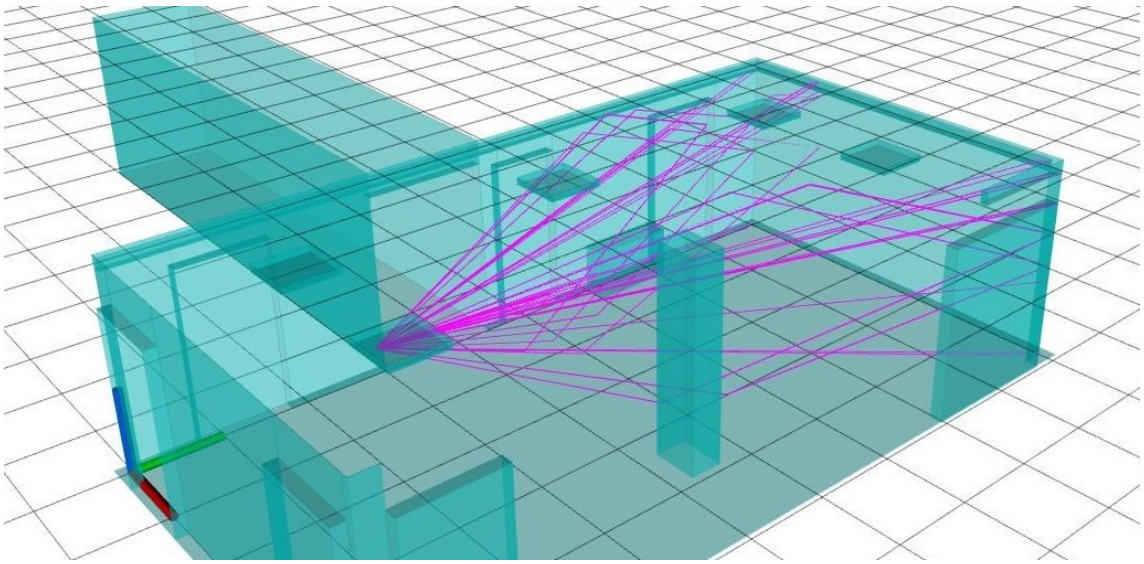


Abbildung 6.7.: Visualisierte Strahlpfade der 3D-Raytracing-Simulation

Abbildung 6.8 zeigt den Vergleich der Ultraschallmessung mit dem Ergebnis der Simulation. Die Nummern sind dabei den einzelnen Peaks beziehungsweise Peakclustern der Ultra-

schallmessung zuzuordnen. Hierbei ist zu erkennen, dass ein Teil der zurückkommenden Ultraschallechos, mit den Nummern 2 beziehungsweise 4 bis 6, eine hohe Übereinstimmung mit der Simulation aufweisen. Hingegen sind die ersten zwei Ultraschallechos mit der Nummer 1, die bei ca 0,015 s liegen, in dem Simulationsergebnis nicht vorhanden. Diese werden von der Decke des seitlichen Gangs reflektiert. Dies liegt an der angenommenen idealen Reflexion der Simulation. Des Weiteren weist das Ultraschallecho bei Nummer 3 zu dem simulierten Ultraschallecho eine leichte Verschiebung auf. Hierbei handelt es sich um ein verschiebbares Objekt im Raum.

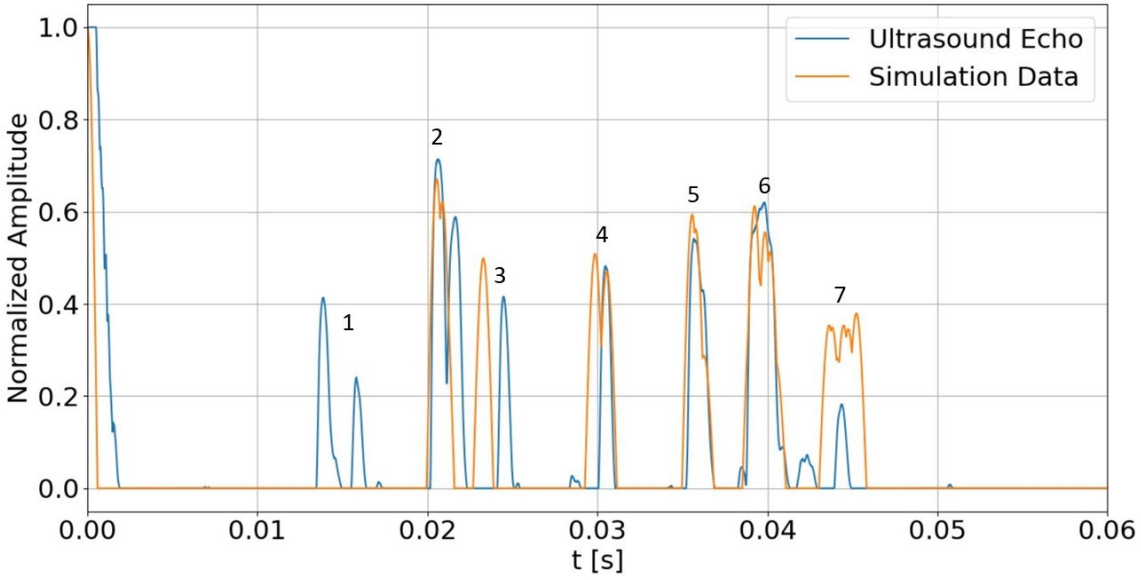


Abbildung 6.8.: Vergleich zwischen Ultraschallmessung und 3D-Raytracing-Simulation

Vergleicht man andere Positionen in dem Raum, so sind die Ergebnisse der Vergleiche identisch. Dabei sind bei diesem Test deutlich die Eigenschaften der 3D-Raytracing-Simulation zu erkennen. An realen Objekten werden hierbei die Ultraschallwellen nicht komplett in einem idealen Winkel reflektiert. So sind in den Ultraschallmessungen Echos zu sehen, die in der Simulation nicht zu sehen sind, da hier von idealen Reflexionen ausgegangen wird. Des Weiteren können verschiebbare Objekte einen Unterschied beim Vergleich zwischen Ultraschallmessung und Simulation verursachen, wenn diese ihre Position verändern. Ein weiterer Punkt stellt der Detaillierungsgrad der Karte dar. Dieser kann bei einer zu niedrigen Detaillierung zu schlechteren Übereinstimmungen führen.

Ein weiterer Punkt ist die Ausführungszeit der 3D-Raytracing-Simulation. Diese beträgt auf einem Rechner mit 14 CPU-Kernen, also 28 Prozessen und einem Takt von 2,87 GHz 0,42 min. Der Jitter beträgt hierbei 0,46 min. Es ist zu erkennen, dass eine zur Ultraschallmessung parallele Simulation zeitintensiv ist. Somit ist eine Bestimmung von Position und

Orientierung mit der Simulation im Vergleich zu einer Look-Up-Tabelle deutlich zeitintensiver.

Zeitliches Verhalten

Um das zeitliche Verhalten des UPS zu beurteilen, wird der Test mit den zwei verwendeten Sensoranordnungen durchgeführt. Hierbei wird zunächst das zeitliche Verhalten mit einem Ultraschallsensor ermittelt. Anschließend erfolgt der Test mit zwei Ultraschallsensoren. Dabei werden die beiden Algorithmen zur Positionsbestimmung einzeln betrachtet. Die Beurteilung des zeitlichen Verhaltens erfolgt mithilfe der Ausführungszeit des UPS, bis eine Position bestimmt wurde. Des Weiteren wird bei diesem Test die Positionsbestimmung mithilfe einer Look-Up-Tabelle als Simulationsersatz durchgeführt.

Die Ausführungszeit des UPS beträgt bei einem Sensor und der globalen Positionsbestimmung ca. 78,6 s. Hierbei beträgt der Jitter 3,6 s. Die Positionsvorhersage, bei der deutlich weniger Positionen verglichen werden, benötigt ca. 3,42 s. Der Jitter ist hierbei vernachlässigbar klein.

Das Ergebnis der Ausführungszeit für zwei Sensoren und der globalen Positionsbestimmung liegt bei 456,6 s. Der Jitter beträgt dabei 1,2 s. Die Positionsvorhersage benötigt hingegen 22,2 s, wobei der Jitter vernachlässigbar klein ist.

Es ist bei diesem Test deutlich zu erkennen, dass die Ausführungszeit exponentiell zur Anzahl der Ultraschallsensoren steigt. So benötigt das UPS für zwei Sensoren ca. sechsmal solange, wie für einen Sensor. Hierbei unterliegt die Ausführungszeit einer relativ hohen Schwankung, da das UPS mit *Ubuntu 18.04* auf einem nicht echtzeitfähigen Betriebssystem ausgeführt wird.

6.3. Anwendungsfall

Für den Anwendungsfall wird der Testraum der 3D-Raytracing-Simulation aus Abschnitt 6.2 verwendet. Hierbei wird der Raum in ein festes Rastermaß von 7 x 7 Positionen aufgeteilt. Das Rastermaß in x-Richtung beträgt dabei 0,6 m und in y-Richtung 1,125 m. Die Winkelschrittweite wird auf 45° festgelegt. Somit ist der Raum in 49 Positionen mit jeweils 8 möglichen Orientierungen aufgeteilt. Der gefahrene Testweg ist in Abbildung 6.9 zu sehen. Die roten Kreuze mit den grünen Pfeilen zeigen die jeweiligen Position mit Orientierung. Bei der Wahl der jeweils nächsten Position wird hierbei die Regel der Positionsvorhersage beachtet.

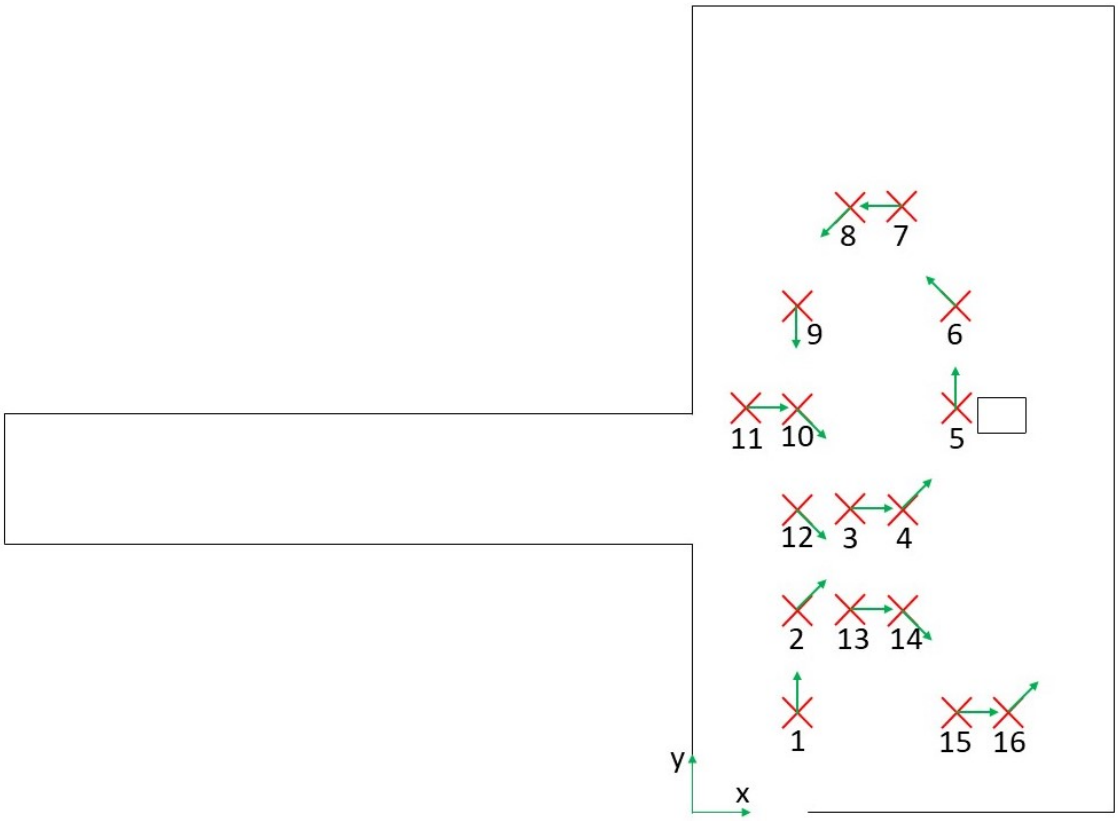


Abbildung 6.9.: Bewegungspfad des Anwendungsfalls im Testraum

Im Folgenden wird der Test zunächst mit dem Algorithmus der globalen Positionsbestimmung durchgeführt. Anschließend erfolgt der Test des Algorithmus mit der Positionsvorhersage. Die Testergebnisse werden jeweils mit den zwei entwickelten Ultraschallsensoranordnungen ermittelt. Hierbei werden die Ultraschallsensoren auf einer Höhe von 1,12 m platziert.

Globale Positionsbestimmung

Tabelle 6.1 zeigt die Ergebnisse des Algorithmus mit globaler Positionsbestimmung und den beiden Ultraschallsensoranordnungen. Hierbei werden die Sollpositionen mit den vom UPS bestimmten Positionen verglichen. Die mittleren Spalten stellen dabei die Ergebnisse der ersten Ultraschallsensoranordnung mit einem Sensor dar. Die Spalten auf der rechten Seite hingegen die zweite Ultraschallsensoranordnung mit zwei Sensoren. Es ist zu erkennen, dass beide Sensoranordnungen bezüglich fünf Positionen die Position korrekt bestimmen.

Nr	Sollposition	Ultraschallsensoranordnung			
	x ; y ; ϕ	1 Sensor		2 Sensoren	
1	1,2 ; 1,125 ; 90	1,2 ; 1,125 ; 90	✓	1,2 ; 7,875 ; 180	✗
2	1,2 ; 2,25 ; 45	2,4 ; 7,875 ; 270	✗	1,2 ; 7,875 ; 180	✗
3	1,8 ; 3,375 ; 0	3,0 ; 5,625 ; 180	✗	3,0 ; 5,625 ; 180	✗
4	2,4 ; 3,375 ; 45	1,2 ; 7,875 ; 180	✗	2,4 ; 1,125 ; 0	✗
5	3,0 ; 4,5 ; 90	3,0 ; 4,5 ; 270	✗	3,0 ; 4,5 ; 90	✓
6	3,0 ; 5,625 ; 135	3,0 ; 5,625 ; 135	✓	3,0 ; 5,625 ; 135	✓
7	2,4 ; 6,75 ; 180	2,4 ; 5,625 ; 180	✗	2,4 ; 5,625 ; 180	✗
8	1,8 ; 6,75 ; 225	3,0 ; 2,25 ; 315	✗	3,0 ; 2,25 ; 315	✗
9	1,2 ; 5,625 ; 270	2,4 ; 4,5 ; 315	✗	1,2 ; 7,875 ; 180	✗
10	1,2 ; 4,5 ; 315	1,2 ; 4,5 ; 315	✓	1,2 ; 4,5 ; 315	✓
11	0,6 ; 4,5 ; 0	2,4 ; 3,375 ; 315	✗	1,2 ; 6,75 ; 180	✗
12	1,2 ; 3,375 ; 315	2,4 ; 4,5 ; 315	✗	1,2 ; 7,875 ; 180	✗
13	1,8 ; 2,25 ; 0	3,0 ; 7,875 ; 180	✗	3,0 ; 1,125 ; 180	✗
14	2,4 ; 2,25 ; 315	2,4 ; 2,25 ; 315	✓	2,4 ; 2,25 ; 315	✓
15	3,0 ; 1,125 ; 0	3,0 ; 1,125 ; 0	✓	3,0 ; 1,125 ; 0	✓
16	3,6 ; 1,125 ; 45	1,2 ; 1,125 ; 225	✗	3,0 ; 2,25 ; 315	✗

Tabelle 6.1.: Auswertung der globalen Positionsbestimmung

Betrachtet man das Ergebnis der Position 5 mit einem Sensor, so ist die Eigenschaft eines symmetrischen Raums zu erkennen. Hierbei zeigt Abbildung 6.10 einen Vergleich der Ultraschallechos mit der Simulation der Sollorientierung mit 90°. Wohingegen Abbildung 6.11 den Vergleich mit der gefundenen Orientierung von 270° zeigt. Die beiden Simulationen weisen einen ähnlichen Verlauf auf, wobei der Simulationsverlauf der 270°-Orientierung bei dem großen Peakcluster eine etwas bessere Übereinstimmung zeigt. Hierdurch wird durch das UPS die Position mit der 270°-Orientierung ermittelt.

Abbildung 6.12 zeigt den Vergleich der Position 5 mit den zwei Ultraschallsensoren der zweiten Sensoranordnung. Der obere Verlauf zeigt die zusammengefügten Ultraschallsignale der beiden Ultraschallsensoren, bei dem der erste Sensor den Ultraschallpuls aussendet. Beim unteren Verlauf sendet hingegen der zweite Sensor den Puls aus. Die Verläufe aus Ultraschallsignal und simulierten Ultraschallsignalen weisen dabei eine hohe Übereinstimmung auf. Es ist hierbei zu erkennen, dass bei Verwendung von zwei Sensoren statt einem, die Vieldeutigkeiten zum Teil reduziert werden und die Position in diesem Fall korrekt erkannt wird.

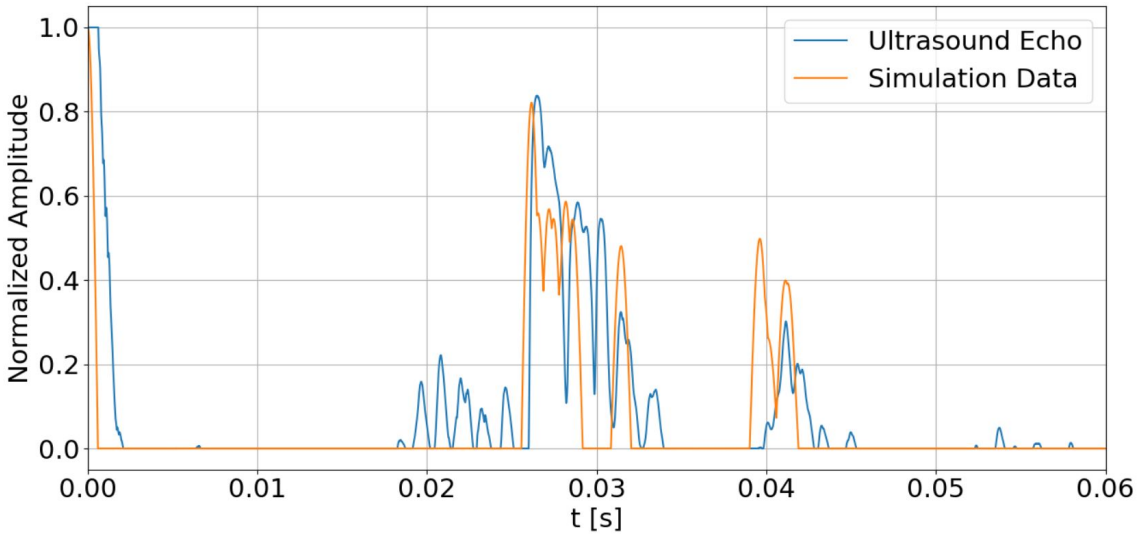


Abbildung 6.10.: Vergleich der Position 5 zwischen Ultraschallsignal und Simulation der Sollposition

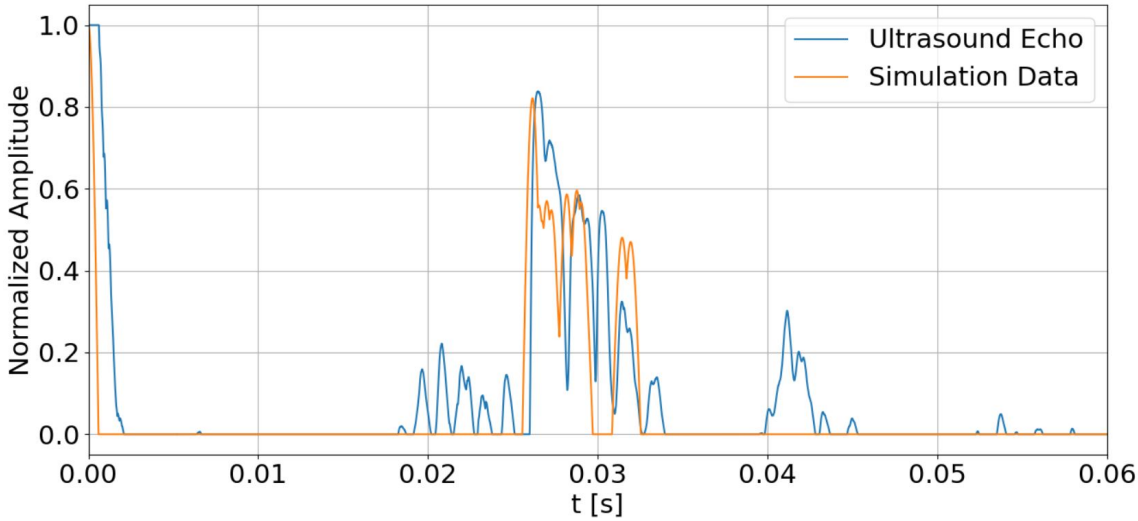


Abbildung 6.11.: Vergleich der Position 5 zwischen Ultraschallsignal und Simulation der bestimmten Position

Durch den Test der globalen Positionsbestimmung ist zu erkennen, dass bei einer ungeregelten Positionsbestimmung die Vieldeutigkeiten der Ultraschallsignaturen eine große Beeinflussung ausüben. Hierbei verhalten sich die zwei Ultraschallsensoranordnungen identisch und weisen ähnliche Ergebnisse auf.

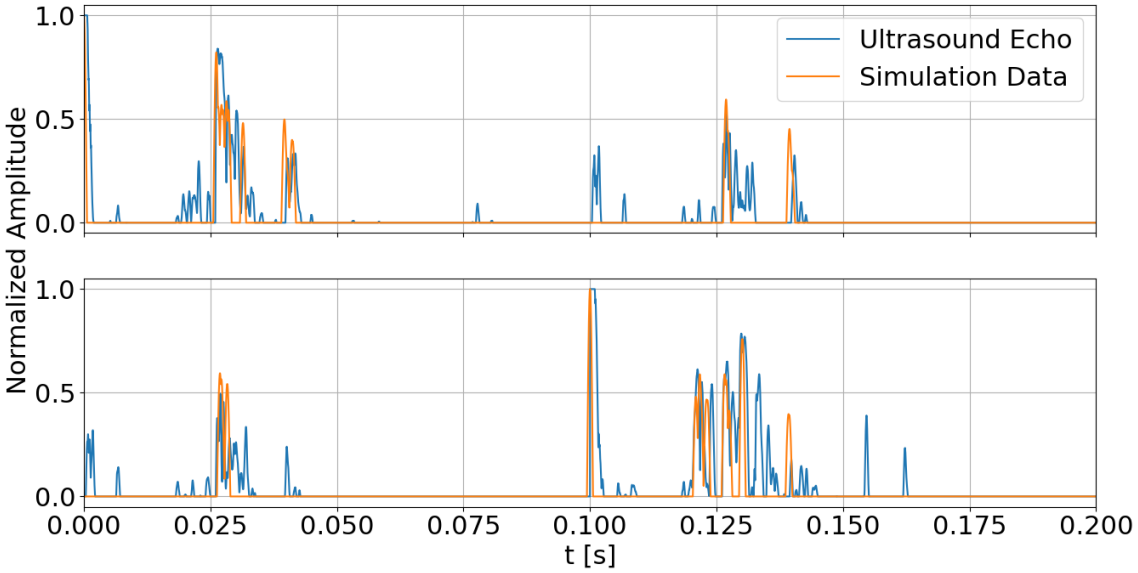


Abbildung 6.12.: Vergleich der Position 5 zwischen Ultraschallsignal und Simulation mit zwei Ultraschallsensoren

Positionsvorhersage

In Tabelle 6.2 ist das Ergebnis des Algorithmus mit Positionsvorhersage zu sehen. In den Testergebnissen sind ebenfalls beide Ultraschallsensoranordnungen wie in Tabelle 6.1 enthalten. Hierbei ist zu erkennen, dass die erste Sensoranordnung mit einem Ultraschallsensor acht Positionen richtig ermittelt. Die zweite Sensoranordnung mit zwei Ultraschallsensoren ermittelt hingegen elf Positionen richtig.

In Abbildung 6.13 ist der Vergleich der Position 16 zwischen den Ultraschallechos und der Simulation der Sollposition zu erkennen. Hierbei werden die Verläufe der zweiten Ultraschallsensoranordnung dargestellt. Der obere Plot zeigt die Signaturen, bei dem der erste Ultraschallsensor den Ultraschallpuls aussendet. Beim unteren Verlauf sendet der zweite um 45° im Uhrzeigersinn verdrehte Sensor den Ultraschallpuls aus. Des Weiteren stellen die einzelnen Plots die aus den Ultraschallsignalen der Ultraschallsensoren erstellten Ultraschallsignaturen dar. Abbildung 6.14 hingegen zeigt den Vergleich der Ultraschallechos mit dem Simulationsergebnis der vom UPS bestimmten Position. Dabei ist zu erkennen, dass bei der simulierten Sollposition der erste Sensor als Pulssender kaum Echos verursacht. Die Ultraschallsignaturen, die durch den zweiten Ultraschallsensor als Pulssender generiert werden, weisen hingegen einen identischen Verlauf auf. Dadurch, dass der Verlauf der durch den erste Ultraschallsensor generierten Ultraschallsignatur aus Abbildung 6.14 eine höhere Übereinstimmung aufweist, wird Position 16 in diesem Fall falsch bestimmt.

Nr	Sollposition	Ultraschallsensoranordnung			
	x ; y ; ϕ	1 Sensor		2 Sensoren	
1	1,2 ; 1,125 ; 90	1,2 ; 1,125 ; 90	✓	1,2 ; 1,125 ; 90	✓
2	1,2 ; 2,25 ; 45	1,2 ; 2,25 ; 90	✗	1,2 ; 3,375 ; 135	✗
3	1,8 ; 3,375 ; 0	1,2 ; 1,125 ; 135	✗	2,4 ; 2,25 ; 315	✗
4	2,4 ; 3,375 ; 45	2,4 ; 3,375 ; 45	✓	2,4 ; 3,375 ; 45	✓
5	3,0 ; 4,5 ; 90	3,0 ; 4,5 ; 90	✓	3,0 ; 4,5 ; 90	✓
6	3,0 ; 5,625 ; 135	3,0 ; 5,625 ; 135	✓	3,0 ; 5,625 ; 135	✓
7	2,4 ; 6,75 ; 180	2,4 ; 5,625 ; 180	✗	2,4 ; 6,75 ; 180	✓
8	1,8 ; 6,75 ; 225	3,0 ; 6,75 ; 180	✗	1,8 ; 6,75 ; 225	✓
9	1,2 ; 5,625 ; 270	1,8 ; 5,625 ; 225	✗	0,6 ; 4,5 ; 270	✗
10	1,2 ; 4,5 ; 315	1,2 ; 4,5 ; 315	✓	1,2 ; 4,5 ; 315	✓
11	0,6 ; 4,5 ; 0	0,6 ; 5,625 ; 0	✗	1,2 ; 4,5 ; 315	✗
12	1,2 ; 3,375 ; 315	1,2 ; 3,375 ; 315	✓	1,2 ; 3,375 ; 315	✓
13	1,8 ; 2,25 ; 0	1,2 ; 2,25 ; 225	✗	1,8 ; 2,25 ; 0	✓
14	2,4 ; 2,25 ; 315	2,4 ; 2,25 ; 315	✓	2,4 ; 2,25 ; 315	✓
15	3,0 ; 1,125 ; 0	3,0 ; 1,125 ; 0	✓	3,0 ; 1,125 ; 0	✓
16	3,6 ; 1,125 ; 45	3,6 ; 1,125 ; 315	✗	4,2 ; 1,125 ; 315	✗

Tabelle 6.2.: Auswertung der Positionsvorhersage

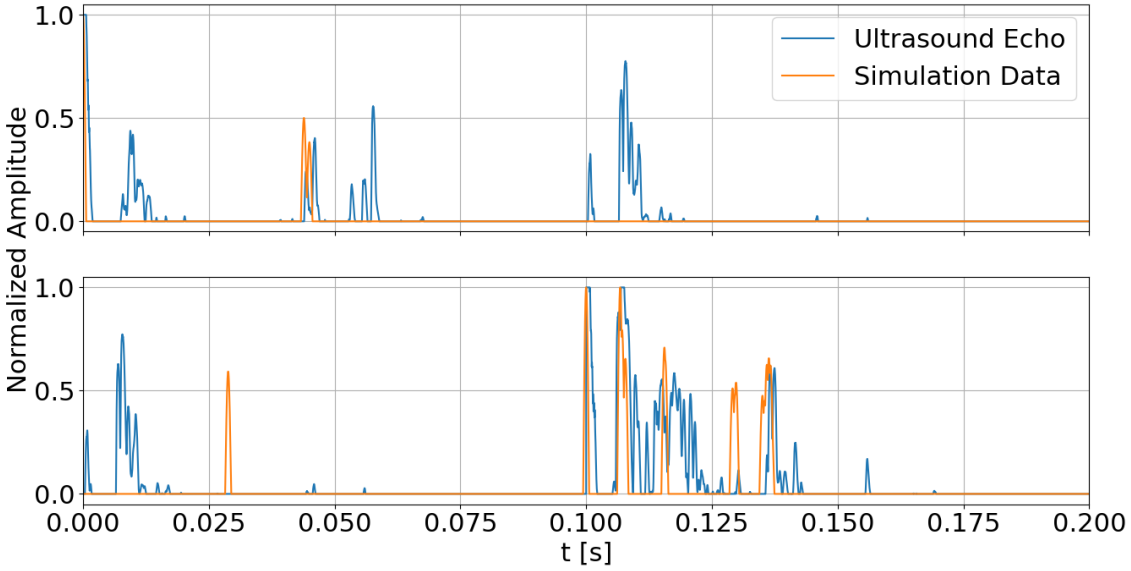


Abbildung 6.13.: Vergleich der Position 16 zwischen Ultraschallsignal und Simulation der Sollposition

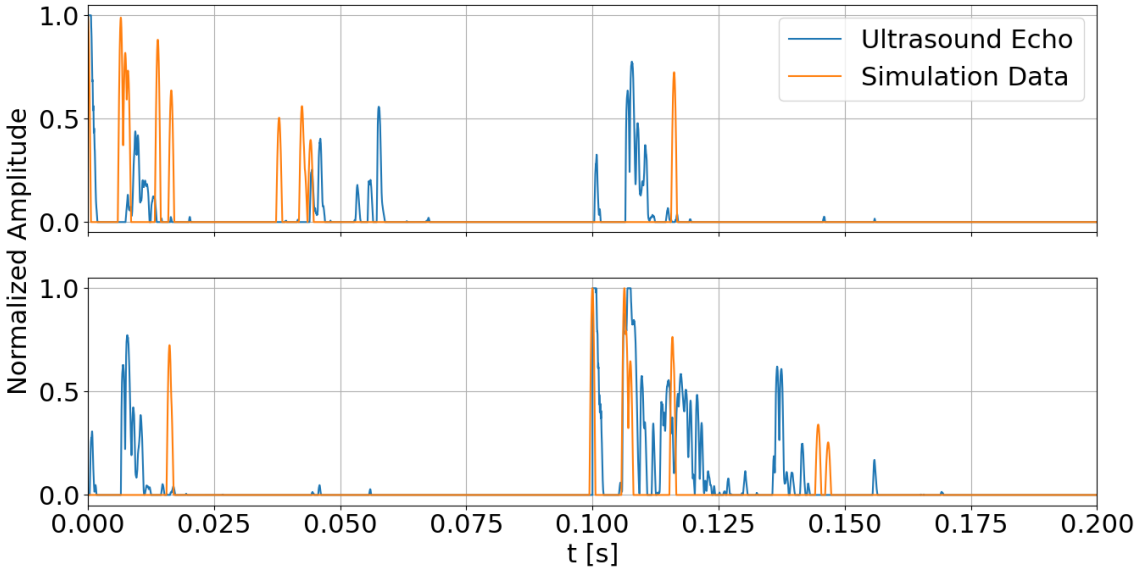


Abbildung 6.14.: Vergleich der Position 16 zwischen Ultraschallsignal und Simulation der bestimmten Position

Fazit des Anwendungsfalls

Vergleicht man die Ultraschallsensoranordnungen mithilfe des Anwendungsfalls miteinander, ist zu erkennen, dass die Anzahl der Vieldeutigkeiten reduziert werden kann, dies aber zum Teil bei einer ungünstigen Sensoranordnung zu neuen Vieldeutigkeiten führt. Ein weiteres Ergebnis des Anwendungsfalls ist, dass der Algorithmus mit der Positionsvorhersage eine deutlich geringere Fehleranfälligkeit im Bezug auf Vieldeutigkeiten aufweist. Es ist ebenfalls zur erkennen, dass mit einer höheren Anzahl an asymmetrisch angeordneten Ultraschallsensoren in Verbindung mit der Positionsvorhersage eine deutlich bessere Erkennung von Position und Orientierung realisiert werden kann. Der Anwendungsfall zeigt, dass eine Positions- und Orientierungsbestimmung mithilfe einer parallelen Ultraschallmessung und Simulation der Umgebung ermöglicht wird. Hierbei wurde eine Genauigkeit in x-Richtung von 0,6 m und in y-Richtung von 1,125 m erreicht. Die maximale Abweichung der Orientierung liegt in diesem Fall durch die 45°-Schrittweite bei 22,5°.

6.4. Bewertung des umfeldsensorbasierten Navigationssystems

Das umfeldsensorbasierte Navigationssystem wird anhand der analysierten Anforderungen aus Abschnitt 3.2 bewertet. Die Bewertung mit den ermittelten Werten ist dabei in Tabelle 6.3 zu sehen.

Aus der Durchführung des Anwendungsfalls folgt, dass eine Erkennung der Fahrzeugposition und -orientierung ermöglicht wird. Hierbei hängt das Ergebnis von der möglichen Symmetrie des Raums und der gewählten Ultraschallsensoranordnung ab.

Für das UVPS wird das UPS als Umfeldsensorsystem verwendet, wodurch sich eine Anzahl an Systemen von eins ergibt. Der ROS-Node ist hingegen für weitere Umfeldsensorsysteme vorbereitet und kann modular erweitert werden.

Durch die Tests der Abstandsmessung und der 3D-Raytracing-Simulation des UPS ist zu sehen, dass eine Erkennung von Umgebung und Objekten ermöglicht wird. Hierbei hat der Abstandstest eine maximale Detektionsreichweite von 10,57 m des UPS ergeben. Der Test der 3D-Raytracing-Simulation und der Anwendungsfall zeigen, dass die Simulation und die Ultraschallmessung zum Teil hohe Übereinstimmungen aufweisen. Hierdurch ist eine Erkennung von Objekten, die sich außerhalb des direkten Sichtfelds befinden, möglich. Dies stellt einen identifizierten Optimierungsansatz dar. Ein weiterer identifizierter Optimierungsansatz, der im weiteren Verlauf umgesetzt werden kann, ist die Ausgabe der Position in geografischen Koordinaten.

Betrachtet man das zeitliche Verhalten des UPS bei Verwendung des positionsvorhersagenden Algorithmus, so erhält man mit der Ausführungszeit bei der Ultraschallsensoranordnung mit einem Sensor eine Messfrequenz von 0,28 Hz. Die Messfrequenz bei Verwendung der zweiten Ultraschallsensoranordnung beläuft sich auf 0,05 Hz. In beiden Fällen wird die Zielmessfrequenz von mindesten 1,67 Hz deutlich unterschritten.

Da die Platzierung des UPS auf dem Dachgepäckträger des Versuchsfahrzeugs vorgesehen ist und bei maximalen Geschwindigkeiten von 6 $\frac{km}{h}$ keine großen Kräfte wirken, ist das System für diese Geschwindigkeiten einsatzfähig. Die beim Anwendungsfall ermittelte Genauigkeit der Positionserkennung beläuft sich in x-Richtung auf 0,6 m und in y-Richtung auf 1,125 m. Die erzielte Genauigkeit wird dabei durch das gewählte Rastermaß der Umgebungskarte beeinflusst. Durch eine feinere Einteilung der Umgebungskarte kann eine höher Genauigkeit erzielt werden. Die für den Anwendungsfall festgelegte Winkelschrittweite führt zu einer Genauigkeit der Orientierungserkennung von 45°. Hierbei kann ebenfalls durch eine feinere Winkelschrittweite eine höhere Genauigkeit erzielt werden.

Da für das UVPS mit dem UPS nur ein Umfeldsensorsystem verwendet wird, entfällt die Verwendung einer Sensordatenfusion zur Ermittlung von Fahrzeugposition und -orientierung. Des Weiteren werden mögliche Fehlfunktionen des UPS abgefangen. Bei einem Fehlverhalten wird das UPS beendet. Dies führt zu keiner Beeinträchtigung des UVPS. Hierdurch ist die Funktionsfähigkeit des UVPS bei einem Ausfall von Teilsystemen gewährleistet.

Analysiert man den festgelegten Adressraum des Ethernetnetzwerks des UPS, so sind 254 IP-Adressen möglich, wobei zwei davon für den Industrie-PC des Versuchsfahrzeugs und den *NVIDIA Jetson Nano* reserviert sind. Hierdurch sind in dem Ethernetnetzwerk theoretisch bis zu 252 Ultraschallsensoren möglich. Da die Abtastrate der Ultraschallsensoren von 200 kHz auf 100 kHz reduziert wurde, reduziert sich ebenfalls das Datenaufkommen pro Ultraschallsensor auf 1,91 $\frac{Mbit}{s}$ im Ethernetnetzwerk. Geht man von einem 100 $\frac{Mbit}{s}$-Ethernetnetzwerk aus, so ergibt sich eine maximale Anzahl von 52 Ultraschallsensoren.

Die Anordnung der Ultraschallsensoren des UPS ist hierbei modular und skalierbar im Bezug auf Anzahl und Positionierung. Durch die Möglichkeit der Platzierung der Ultraschallsensoren auf dem Dachgepäckträger des Versuchsfahrzeugs ist ein rückstandsfreier Ein- und Ausbau möglich. Ebenfalls wird eine Integration ohne Erlöschen der Straßenverkehrszulassung realisiert.

Das UVPS wurde mit *Ubuntu 16.04* als Basis implementiert. Für das UPS wird hingegen *Ubuntu 18.04* als Betriebssystem verwendet. Hierbei zeigt der Anwendungsfall, dass zwischen den Systemen eine problemlose Kommunikation möglich ist. Da es sich bei dem UPS um ein eingebettetes System handelt, zeigt die Verwendung eines anderen Betriebssystems keine Auswirkungen. Hierbei ist als Ergebnis zu erkennen, dass das UVPS auf *Ubuntu 16.04* lauffähig ist. Des Weiteren wurden sowohl das UVPS als auch das UPS mithilfe von ROS implementiert.

Als Spannungsversorgung für das UPS werden 5 V und 12 V verwendet. Hierbei werden die Ultraschallsensoren mit 12 V Spannung versorgt. Der *NVIDIA Jetson Nano* und der Ethernet-Switch werden hingegen mit 5 V Spannung versorgt. Da ein Betrieb unter regengeschützten Bedingungen vorgesehen ist, ist eine Abdichtung der Ultraschallsensoren nicht nötig. Hierbei ist ein Betrieb unter regengeschützten Bedingungen möglich. Da sich der analysierte Temperaturbereich für den Betrieb des UVPS innerhalb des standardmäßigen Betriebsbereichs von Elektronikkomponenten befindet, sollte ein Betrieb unter diesen Umgebungstemperaturen möglich sein. Da aber das *Arduino Due* Board keine Spezifikationen für Umgebungstemperaturen hat und die entwickelte Platine zur Synchronisation und Spannungsversorgung der Ultraschallsensoren in diesem Zusammenhang nicht getestet wurde, kann eine Beurteilung dieser Anforderung nicht erfolgen.

Betrachtet man die einzelnen Bewertungen der Anforderungen an das UVPS, ist zu erkennen, dass dieses für den geplanten Einsatz als zeitweise, alternative Methode zur satellitenbasierten Positionsbestimmung geeignet ist. Hierbei zeigen das UVPS und das UPS Potenzial für mögliche Verbesserungen.

ID	Beschreibung	Wert	Bewertung
1	Erkennung der Fahrzeugposition		✓
2	Erkennung der Fahrzeugorientierung		✓
3	Verwendung von Umfeldsensorsystemen	$n = 1$	✓
4	Verwendung eines Ultraschallsensorsystems		✓
5	Erkennung von Objekten, die sich nicht im direkten Sichtfeld befinden		✗
6	Erkennung der Umgebung und von Objekten		✓
7	Möglichst hohe Detektionsreichweite der einzelnen Umfeldsensorsysteme	$x \leq 10{,}57\,m$	✓
8	Ausreichend hohe Messfrequenz	$f \leq 0{,}28\,Hz$	✗
9	Einsatzfähigkeit des Navigationssystems bei Schrittgeschwindigkeit	$v \leq 6\,\frac{km}{h}$	✓
10	Möglichst hohe Genauigkeit der Positionserkennung	$\Delta x = 0{,}6\,m$ $\Delta y = 1{,}125\,m$	✗
11	Möglichst hohe Genauigkeit der Orientierungserkennung	$\Delta\phi = 45^\circ$	✗
12	Bei der Verwendung von mehreren Umfeldsensorsystemen Positions- und Orientierungsbestimmung durch Sensordatenfusion		–
13	Navigationssystem modular erweiterbar um weitere Umfeldsensorsysteme		✓
14	Funktionsfähigkeit des Navigationssystems bei Ausfall von Teilsystemen		✓
15	Verwendung von mindestens zwei Ultraschallsensoren	$n \leq 52$	✓
16	Modulare und skalierbare Anordnung der Ultraschallsensoren		✓
17	Integration in das Versuchsfahrzeug ohne Erlöschen der Straßenverkehrszulassung		✓
18	Rückstandsfreier Ein- und Ausbau für das Versuchsfahrzeug		✓
19	Software des Navigationssystems lauffähig auf *Ubuntu 16.04*		✓
20	Implementierung des Navigationssystems mithilfe von ROS		✓
21	Spannungsversorgung durch Versuchsfahrzeug beziehungsweise Industrie-PC	$5\,V$, $12\,V$	✓
22	Betrieb unter regengeschützten Bedingungen		✓
23	Betrieb bei Umgebungstemperaturen für Indoor-Bereiche	$0\,^\circ C$ bis $30\,^\circ C$	✗
24	Ausgabe der Position in geografischen Koordinaten		✗

Tabelle 6.3.: Bewertung des umfeldsensorbasierten Navigationssystems anhand der Anforderungen

7. Zusammenfassung

Im Rahmen dieser Arbeit wurde ein umfeldsensorbasiertes Navigationssystem für die Positionserkennung autonomer Fahrzeuge realisiert. Das System ermöglicht eine zu GNSS-basierten Systemen zeitweise, alternative Methode zur Positions- und Orientierungsbestimmung autonomer Fahrzeuge in Indoor-Bereichen. Dazu wird als Umfeldsensorsystem ein ultraschallbasiertes Positionierungssystem verwendet.

Das umfeldsensorbasierte Navigationssystem ist hierbei in Anzahl und Art der verwendeten Umfeldsensorsysteme modular und skalierbar. Des Weiteren ermöglicht das Navigationssystem eine leichte Integration in das Versuchsfahrzeug des *Urban Mobility Lab*.

Hierbei besteht das ultraschallbasierte Positionierungssystem aus einer eingebetteten zentralen Recheneinheit und einem synchronisierten Ultraschallsensorsystem. Das ultraschallbasierte Positionierungssystem ist dabei im Bezug auf Anzahl und Anordnung der verwendeten Ultraschallsensoren modular und skalierbar. Als Anordnungen wurden zwei unterschiedliche Ultraschallsensoranordnungen entwickelt. Mithilfe von Ultraschall und der Detektion von Mehrfachreflexionen wird eine Erkennung von Umgebung und Objekten realisiert. Durch eine parallele Anordnung von Ultraschallmessung und simulierten Ultraschallechos wird eine ultraschallbasierte Bestimmung von Position und Orientierung ermöglicht.

Die auf drei Dimensionen erweiterte Raytracing-Simulation des *Urban Mobility Lab* unterteilt die Ultraschallwellen in einzelne Strahlen und berechnet durch Mehrfachreflexionen die jeweiligen Pfade. Hierbei werden die Strahlen in einer im Vorfeld definierten Karte simuliert. Des Weiteren berechnet die Simulation die Abstrahlcharakteristik des Ultraschallsensors beim Aussenden und Empfangen der einzelnen Strahlen. Das Ergebnis der Raytracing-Simulation ist der zeitliche Verlauf der zurückkommenden Ultraschallechos.

Als Ultraschallsensorsystem wird ein im *Urban Mobility Lab* bereits vorhandenes System verwendet und optimiert. Hierzu wird das Senden und Empfangen von Ultraschall in ein Ultraschallmodul ausgelagert. Eine weitere Optimierung des Ultraschallsensors ist die Synchronisation der Ultraschallsensoren untereinander. Des Weiteren wird über die Synchronisation ein Token Passing realisiert, mit dem der jeweilige Ultraschallsensor bestimmt wird, der den Ultraschallpuls aussendet. Hierzu wird ein Platinendesign umgesetzt. Ebenfalls optimiert wird die Spannungsversorgung der Ultraschallsensoren. Um diese zu vereinfachen, wird die interne Spannungsversorgung zu einer externen Spannungsversorgung geändert.

Zur Bestimmung von Position und Orientierung wurden zwei Algorithmen implementiert. Die globale Positionsbestimmung vergleicht hierbei die aufgenommenen Ultraschallechos mit den simulierten Ultraschallechos jeder einzelnen Position und Orientierung. Der Algorithmus mit Positionsvorhersage hingegen verwendet die letzte bekannte Position und ermittelt mögliche Positionen, um die Simulationsergebnisse dieser mit dem aufgenommenen Ultraschallecho zu vergleichen.

Der abschließende Test führt zunächst eine Beurteilung des optimierten Ultraschallsensors durch. Der anschließende Test bewertet das ultraschallbasierte Positionierungssystem. Ein weiterer Test ist die Anwendung des umfeldsensorbasierten Navigationssystems anhand eines Anwendungsfalls. Des Weiteren erfolgt eine Bewertung des gesamten umfeldsensorbasierten Navigationssystems. Hierbei orientieren sich die Tests und Bewertungen an den analysierten Anforderungen.

Aus der Bewertung des umfeldsensorbasierten Navigationssystems folgt, dass dieses als zeitweise, alternative Methode zur Bestimmung von Position und Orientierung autonomer Fahrzeuge eingesetzt werden kann. Das umfeldsensorbasierte Navigationssystem ist hierbei eine Basis für weitere Optimierungen und Erweiterungen, die im Anschluss an diese Arbeit erfolgen können.

8. Ausblick

Das entwickelte umfeldsensorbasierte Navigationssystem zeigt Potenzial für mögliche Verbesserungen. Hierzu beschreibt der Ausblick die identifizierten Optimierungsansätze für das ultraschallbasierte Positionierungssystem. Ein weiterer Punkt ist die Weiterentwicklung des umfeldsensorbasierten Navigationssystems. Abschließend wird auf die Integration des Systems in das Versuchsfahrzeug eingegangen und dessen Anwendung beschrieben.

8.1. Optimierung des ultraschallbasierten Positionierungssystems

Ein Ansatz zur Optimierung des ultraschallbasierten Positionierungssystems ist die Detektion von verdeckten Objekten. Hierbei kann bei einer korrekt bestimmten Position und Orientierung die Ultraschallsignatur mit der Signatur der Simulation verglichen werden. Bei einem ausreichend hohen Detailgrad der Simulationskarte kann die Signatur der Simulation von der Ultraschallsignatur subtrahiert werden. Die übrig bleibenden Ultraschallechos könnten hierbei Aufschluss über verdeckte Objekte geben. Hierzu kann sich zum Beispiel ein Machine-Learning-Algorithmus eignen.

Ein weiterer identifizierter Optimierungsansatz beschreibt die Bestimmung von Position und Orientierung. Hierzu kann diese mithilfe eines Machine-Learning-Algorithmus realisiert werden, wie zum Beispiel dem *Deep Gaussian Process Regression Model* (vgl. Teng u. a., 2018, S. 1). Um in diesem Fall einen großen Datensatz unter definierten Bedingungen zu erhalten, eignet sich zur Erzeugung die 3D-Raytracing-Simulation. Da die Simulation allerdings ideale Bedingungen darstellt, können die Simulationsergebnisse mit einem künstlichen Rauschen versehen werden. Erzeugt man so aus den einzelnen Simulationsergebnissen mehrere Datenpunkte mit unterschiedlichen Rauchfaktoren, erhöht dies die Datenvielfalt des Datensatzes. Mit einem Machine-Learning-Algorithmus kann es ebenfalls möglich sein, von einer unbekannten Umgebung, von der noch keine Karte vorhanden ist, automatisiert eine Karte zu erstellen.

Um das ultraschallbasierte Positionierungssystem besser in das umfeldsensorbasierte Navigationssystem zu integrieren, ist es nötig, die Positions- und Orientierungsangaben an

geografische Koordinaten anzupassen. Dies kann mithilfe eines Wrappers leicht realisiert werden. Dabei ist es nötig, genaue geografische Koordinaten der jeweiligen Karte zu ermitteln.

Da das ultraschallbasierte Positionierungssystem nicht in allen Bereichen einsetzbar ist, eignet sich die Verwendung von sogenannten *Geofences*. Hierbei wird das Gebiet, in dem das System angewendet werden kann, mithilfe von geografischen Koordinaten abgegrenzt. Bei der Einfahrt in ein solches Gebiet kann damit die Aktivierung des Systems erfolgen. Solche Gebiete können zum Beispiel in Kartendiensten wie etwa *OpenStreetMap* integriert werden.

8.2. Optimierung des umfeldsensorbasierten Navigationssystems

Der Optimierungsansatz des umfeldsensorbasierten Navigationssystems betrifft die Sensordatenfusion. Diese ist nötig, um die Position und Orientierung des Fahrzeugs mithilfe von mehreren Systemen zu bestimmen. Hierzu kann zum Beispiel der im Rahmen einer Bachelorarbeit im *Urban Mobility Lab* entwickelte Kalman-Filter erweitert werden (vgl. Mohr, 2016, S. 3). Somit kann die GNSS-basierte Bestimmung von Position und Orientierung mithilfe des ultraschallbasierten Positionierungssystems bei stark verfälschten beziehungsweise nicht vorhandenen Satellitensignalen zeitweise verbessert werden.

Des Weiteren können mithilfe einer Sensordatenfusion im *Urban Mobility Lab* entwickelte Positionierungssystem in das umfeldsensorbasierte Navigationssystem integriert werden. Hierzu zählen unter anderem das im Rahmen eine Masterarbeit entwickelte hochpräzise System zur Positionsbestimmung von Fahrzeugen in urbanen Gebieten (vgl. Wegner, 2017, S. 3) und das im Rahmen einer Bachelorarbeit entwickelte LiDAR-basierte SLAM-System (vgl. Vater, 2018, S. 3). Weitere System, die in das umfeldsensorbasierte Navigationssystem integriert werden können, sind die Entwicklungen aus zwei aktuellen Masterarbeiten des *Urban Mobility Lab* zur Positionsbestimmung mithilfe einer sphärischen- beziehungsweise einer Stereo-Kamera.

8.3. Integration in das Versuchsfahrzeug

Im Anschluss an diese Arbeit kann die Integration des umfeldsensorbasierten Navigationssystems in das Versuchsfahrzeug erfolgen. Da die Ultraschallsensoren einen rückstandsfreien Ein- beziehungsweise Ausbau ermöglichen, kann die Integration auf dem Dachgepäckträger des Versuchsfahrzeugs ohne Erlöschen der Straßenverkehrszulassung erfolgen. Hierbei

ist durch die Eigenschaften des Dachgepäckträgers eine variable Anordnung der Ultraschallsensoren möglich. In Abbildung 8.1 ist eine mögliche asymmetriesche Sensoranordnung zu sehen, die aus sechs Ultraschallsensoren besteht. Hierbei stellen die roten Linien die Grenzen der jeweiligen Ultraschallkeulen dar. Durch die asymmetrische Anordnung und der teilweisen Überschneidung der Ultraschallkeulen kann die Auswirkung von Vieldeutigkeiten auf die Bestimmung von Position und Orientierung minimiert werden. Hierbei ist eine einfache Portierung der Software auf den Industrie-PC des Versuchsfahrzeugs durch die Verwendung von *Ubuntu 16.04* als Betriebssystem und ROS als Meta-Betriebssystem möglich.

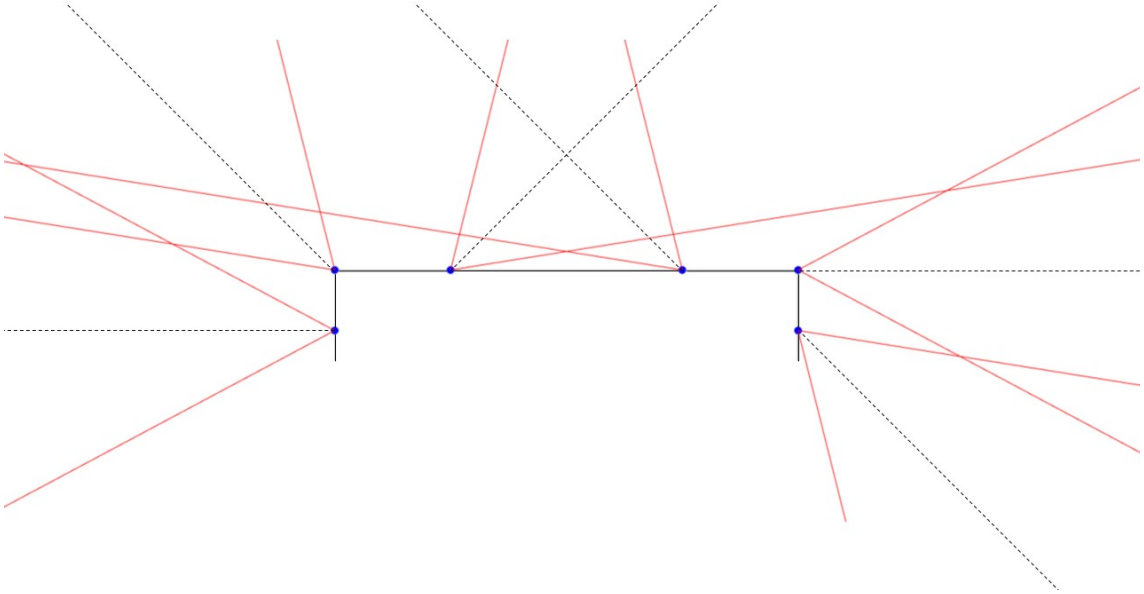


Abbildung 8.1.: Mögliche Sensoranordnung der Ultraschallsensoren auf dem Versuchsfahrzeug

Abkürzungsverzeichnis

ADC	Analog to Digital Converter
CAN	Controller Area Network
EKF	Extended Kalman-Filter
GLONASS	Global Navigation Satellite System
GNSS	Global Navigation Satellite System
GPS	Global Positioning System
INS	Inertial Navigation System
LiDAR	Light Detection and Ranging
MUSSE	Multi-Ultrasonic-Sensor System for Echolocation
PoE	Power over Ethernet
ROS	Robot Operating System
SLAM	Simultaneous Localization and Mapping
SoC	System on Chip
UPS	Ultrasound Positioning System / ultraschallbasiertes Positionierungssystem
UVPS	Universal Vehicle Positioning System / umfeldsensorbasiertes Navigationssystem

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Albuquerque u. a. 2008] ALBUQUERQUE, Daniel ; VIEIRA, Jose ; BASTOS, Carlos A. C.: Room Acoustics Simulator for Ultrasonic Robot Location. In: *Researchgate* (2008), Januar, S. 1 – 5. – URL https://www.researchgate.net/publication/228888202_Room_Acoustics_Simulator_for_Ultrasonic_Robot_Location. – Abruf: 2019-12-04

[Albuquerque 2013] ALBUQUERQUE, Daniel F.: *Sistema de Localizacao com Ultrassons - Ultrasonic Location System*. Aveiro, Universidade de Aveiro, Dissertation, 2013. – URL http://repositorio.ipv.pt/handle/10400.19/3036. – Abruf: 2018-05-07

[Arduino 2020] ARDUINO: *Website Arduino Due*. 2020. – URL https://store.arduino.cc/usa/arduino-due. – Abruf: 2020-03-14

[Browne und Kleeman 2009] BROWNE, Damien ; KLEEMAN, Lindsay: An Advanced Sonar Ring Design with 48 Channels of Continuous Echo Processing using Matched Filters. In: *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems* (2009), Oktober, S. 4040 – 4046. – URL https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5354228. – Abruf: 2019-09-27. – ISSN 2153-0866

[Dudek und Jenkin 2016] DUDEK, Gregory ; JENKIN, Michael: *Springer Handbook of Robotics*. Kap. Inertial Sensing, GPS and Odometry, S. 737 – 752. Berlin : Springer, 2016. – ISBN 978-3-319-32550-7

[Eichler 2014] EICHLER, Jürgen: *Physik für das Ingenieurstudium: Prägnant mit vielen Kontrollfragen und Beispielaufgaben*. 5. vollst. überarb. u. erw. Aufl. Wiesbaden : Springer Vieweg, 2014. – URL https://link.springer.com/book/10.1007%2F978-3-658-04626-2. – Abruf: 2018-02-27. – ISBN 978-3-658-04626-2

[Ethik-Kommission 2017] ETHIK-KOMMISSION: *Automatisiertes und vernetztes Fahren*. Berlin : Bundesministerium für Verkehr und digitale Infrastruktur, Juni 2017. – URL https://www.bmvi.de/SharedDocs/DE/Publikationen/DG/bericht-der-ethik-kommission.pdf?__blob=publicationFile. – Abruf: 2019-10-19

[Gotlib u. a. 2019] GOTLIB, Adam ; LUKOJC, Kornelia ; SZCZYGIELSKI: Localization-based software architecture for 1:10 scale autonomous car. In: *2019 International Interdisciplinary PhD Workshop (IIPhDW)* (2019), Mai, S. 7 – 11. – URL https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8755418. – Abruf: 2020-01-06. ISBN 978-1-7281-0423-2

[Hering und Schönfelder 2012] HERING, Ekbert ; SCHÖNFELDER, Gert: *Sensoren in Wissenschaft und Technik: Funktionsweise und Einsatzgebiete*. Wiesbaden : Vieweg + Teubner, 2012. – URL https://link.springer.com/book/10.1007%2F978-3-8348-8635-4. – Abruf: 2018-02-27. – ISBN 978-3-8348-0169-2

[Ilias u. a. 2016] ILIAS, Bukhari ; SHUKOR, Shazmin A. A. ; ADOM, Abdul H. ; IBRAHIM, Mohd F. ; YAACOB, Sazali: A Novel Indoor Mobile Robot Mapping with USB-16 Ultrasonic Sensor Bank and NWA Optimization Algorithm. In: *2016 IEEE Symposium on Computer Applications & Industrial Electronics (ISCAIE)* (2016), Mai, S. 189 – 194. – URL https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7575061. – Abruf: 2019-09-27. ISBN 978-1-5090-1543-6

[Kichun u. a. 2015] KICHUN, Jo ; KIM, Junsoo ; KIM, Dongchul ; JANG, Chulhoon ; SUNWOO, Mayoungho: Development of Autonomous Car - Part II: A Case Study on the Implementation of an Autonomous Driving System Based on Distributed Architecture. In: *IEEE Transactions on Industrial Electronics* Vol. 62 (2015), August, Nr. 8, S. 5119 – 5132. – URL https://ieeexplore.ieee.org/document/7056521. – Abruf: 2019-08-03. – ISSN 1557-9948

[Kleeman und Kuc 2016] KLEEMAN, Lindsay ; KUC, Roman: *Springer Handbook of Robotics*. Kap. Sonar Sensing, S. 753 – 781. Berlin : Springer, 2016. – ISBN 978-3-319-32550-7

[Lerch u. a. 2009] LERCH, Reinhard ; SESSLER, Gerhard ; WOLF, Dietrich: *Technische Akustik: Grundlagen und Anwendungen*. Berlin : Springer, 2009. – URL https://link.springer.com/book/10.1007%2F978-3-540-49833-9. – Abruf: 2018-02-27. – ISBN 978-3-540-49833-9

[MaxBotix 2015] MAXBOTIX: *XL-MaxSonar - EZ Series Datasheet*. 2015. – URL https://www.maxbotix.com/documents/XL-MaxSonar-EZ_Datasheet.pdf. – Abruf: 2020-02-24

[Mohr 2016] MOHR, Philipp: *Bachelorthesis: Development of a Kalman Filter for Multiple-Sensor Embedded Systems in Urban Public Transportation Applications*. Hamburg : Hochschule für Angewandte Wissenschaften Hamburg, 2016. – URL http://edoc.sub.uni-hamburg.de/haw/volltexte/2017/3853/. – Abruf: 2019-07-23

[Niehues 2014] NIEHUES, Daniel: *Hochgenaue Positionsbestimmung von Fahrzeugen als Grundlage autonomer Fahrregime im Hochgeschwindigkeitsbereich*. Dresden, Technische Universität Dresden, Dissertation, Mai 2014. – URL http://tud.qucosa.de/api/qucosa%3A26709/attachment/ATT-0/. – Abruf: 2019-10-20

[NOAA 2019] NOAA: *GPS Accuracy*. National Oceanic and Atmospheric Administration, 2019. – URL https://www.gps.gov/systems/gps/performance/accuracy/. – Abruf: 2019-10-20

[NVIDIA 2020a] NVIDIA: *NVIDIA Jetson Nano System-on-Module Datasheet*. 2020. – URL https://developer.download.nvidia.com/assets/embedded/secure/jetson/Nano/docs/JetsonNano_DataSheet_DS09366001v1.0.pdf?0I_b6PxOtlRxoiJ79MGMIndw3pXN-LmS5DsltZD2zCcdSSa-PT5ZCHZD_we_9GNm3DMInxwWBhNK2BiqzhvyTQWnpfXHmJt20V1egbTLClkrav496giMvg3HooAREzOQxvjgrtP48DujW_gMjy7tRKaj8otrdRaIIcDfr0PGGJYGGd45A. – Abruf: 2020-02-28

[NVIDIA 2020b] NVIDIA: *Website NVIDIA L4T*. 2020. – URL https://developer.nvidia.com/embedded/linux-tegra. – Abruf: 2020-03-14

[Ohm und Lüke 2014] OHM, Jens R. ; LÜKE, Hans D.: *Signalübertragung - Grundlagen der digitalen und analogen Nachrichtenübertragungssysteme*. 12. Aufl. Berlin : Springer Vieweg, 2014. – URL https://link.springer.com/book/10.1007%2F978-3-642-53901-5. – Abruf: 2018-05-09. – ISBN 978-3-642-53901-5

[PicoTechnology 2020] PICOTECHNOLOGY: *Website Pico Technology*. 2020. – URL https://www.picotech.com/. – Abruf: 2020-03-14

[RaspberryPi 2020] RASPBERRYPI: *Raspberry Pi 4 Model B Datasheet*. 2020. – URL https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2711/rpi_DATA_2711_1p0_preliminary.pdf. – Abruf: 2020-02-28

[Reif 2014] REIF, Konrad: *Automobilelektronik: Eine Einführung für Ingenieure*. 5. überarb. Aufl. Wiesbaden : Springer Vieweg, 2014. – URL https://link.springer.com/book/10.1007%2F978-3-658-05048-1. – Abruf: 2018-02-27. – ISBN 978-3-658-05048-1

[Rodin und Stajduhar 2017] RODIN, R. ; STAJDUHAR, I.: The Challenge of Measuring Distance to Obstacles for The Purpose of Generating a 2-D Indoor Map Using an Autonomous Robot Equipped with an Ultrasonic Sensor. In: *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)* (2017), Mai, S. 1021 – 1026. – URL https://ieeexplore.ieee.org/

stamp/stamp.jsp?tp=&arnumber=7973574. – Abruf: 2019-09-27. ISBN 978-953-233-090-8

[ROS 2020] ROS: *Website Robot Operating System (ROS)*. 2020. – URL http://wiki.ros.org/de/ROS/Introduction. – Abruf: 2020-03-14

[Rotzlawski 2018] ROTZLAWSKI, Christopher: *Bachelorarbeit: Konzeption und Umsetzung eines Entwicklungssystems für intelligente Ultraschallsensoren in Mehrfachanordnung zur Lokalisation und Umgebungserkennung in komplexen Umgebungen*. Hamburg : Hochshule für Angewandte Wissenschaften Hamburg, 2018. – URL http://edoc.sub.uni-hamburg.de/haw/volltexte/2018/4340/. – Abruf: 2019-07-23

[Seong und Byung 2013] SEONG, Jim K. ; BYUNG, Kook K.: Dynamic Ultrasonic Hybrid Localization System for Indoor Mobile Robots. In: *IEEE Transactions on Industrial Electronics* Vol. 60 (2013), Oktober, Nr. 10, S. 4562 – 4573. – URL https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6290366. – Abruf: 2019-09-27. – ISSN 1557-9948

[Tanil u. a. 2019] TANIL, Cagatay ; KHANAFSEH, Samer ; JOERGER, Mathieu ; KUJUR, Birendra ; KRUGER, Brett ; GROOT, Lance de ; PERVAN, Boris: Optimal INS/GNSS Coupling for Autonomous Car Positioning Integrity. In: *Proceedings of the 32nd International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS+ 2019)* (2019), September, S. 3123 – 3140. – URL https://www.researchgate.net/profile/Cagatay_Tanil/publication/336180346_Optimal_INSGNSS_Coupling_for_Autonomous_Car_Positioning_Integrity/links/5d938262a6fdcc2554aba25d/Optimal-INS-GNSS-Coupling-for-Autonomous-Car-Positioning-Integrity.pdf. – Abruf: 2019-10-19

[Teng u. a. 2018] TENG, Fei ; TAO, Wenyuan ; OWN, Chung-Ming: Localization Reliability Improvement Using Deep Gaussian Process Regression Model. In: *Sensors - Open Access Journal* Vol. 18 (2018), November, S. 1 – 19. – URL https://www.mdpi.com/1424-8220/18/12/4164. – Abruf: 2020-02-14. – ISSN 1424-8220

[Vater 2018] VATER, Tobias: *Bachelorthesis: Inline-Auswertung und -Korrektur multipler, komplexer, fusionierter, bildgebender Sensoren für das automatisierte Fahren*. Hamburg : Hochshule für Angewandte Wissenschaften Hamburg, 2018. – URL http://edoc.sub.uni-hamburg.de/haw/volltexte/2018/4304/. – Abruf: 2019-07-23

[VDA 2015] VDA: *Automatisierung*. Berlin : Verband der Automobilindustrie, September 2015. – URL https://www.vda.de/de/services/Publikationen/automatisierung.html. – Abruf:2019-10-20

[Wegner 2017] WEGNER, Mario: *Masterthesis: Konzeption und Entwicklung eines hochpräzisen Systems zur Positionsbestimmung von Fahrzeugen in urbanen Gebieten*. Hamburg : Hochschule für Angewandte Wissenschaften Hamburg, 2017. – URL http://edoc.sub.uni-hamburg.de/haw/volltexte/2017/3957/. – Abruf: 2019-07-23

[Wu u. a. 2016] WU, Xiaodong ; ABRAHANTES, Miguel ; EDINGTON, Mark: MUSSE: A Designed Multi-Ultrasonic-Sensor System for Echolocation on Multiple Robots. In: *2016 Asia-Pacific Conference on Intelligent Robot Systems (ACIRS)* (2016), Juli, S. 79 – 83. – URL https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7556192. – Abruf: 2019-09-27. ISBN 978-1-5090-1362-3

A. Hardwaredesign

A.1. Pinbelegung der Synchronisationsleitung

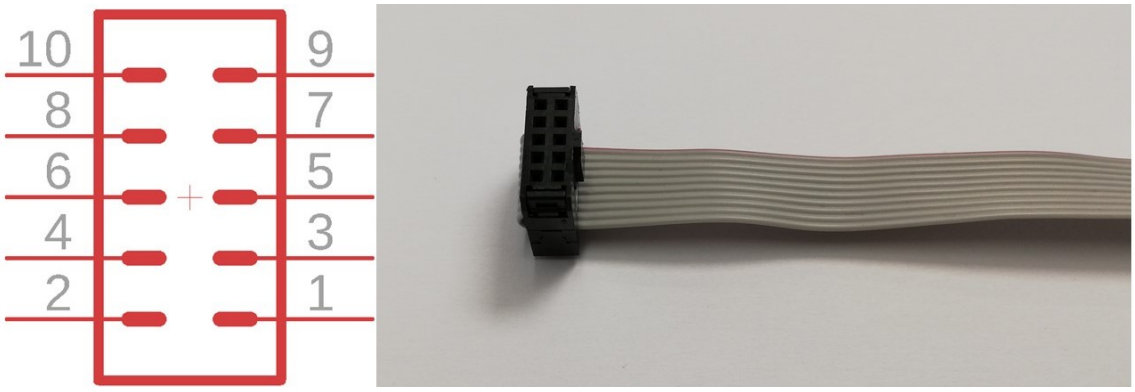


Abbildung A.1.: Pinbelegung der Synchronisationsleitung

Pin	Belegung	Pin	Belegung
1	Ground	2	Ground
3	Ground	4	Ground
5	12 V	6	12 V
7	12 V	8	Reset
9	Token Passing	10	Synchronisierung

Tabelle A.1.: Pinbelegung der Synchronisationsleitung

A.2. Platinenlayout

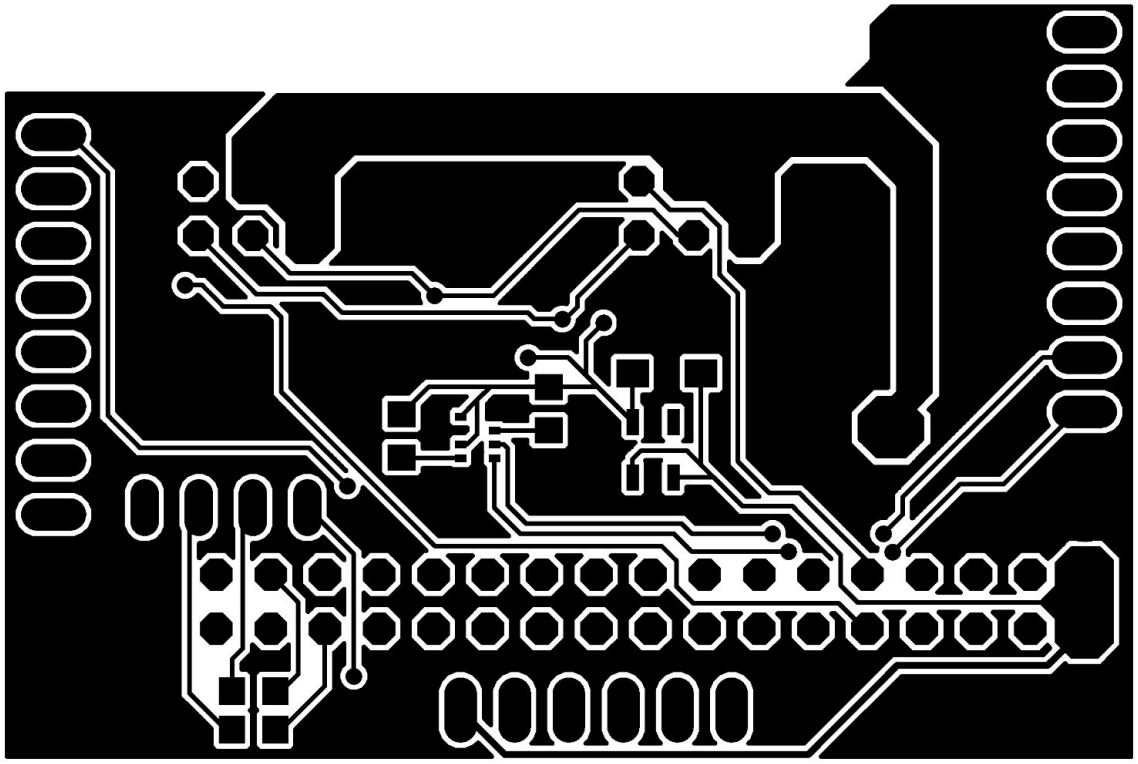


Abbildung A.2.: Platinenlayout der Oberseite

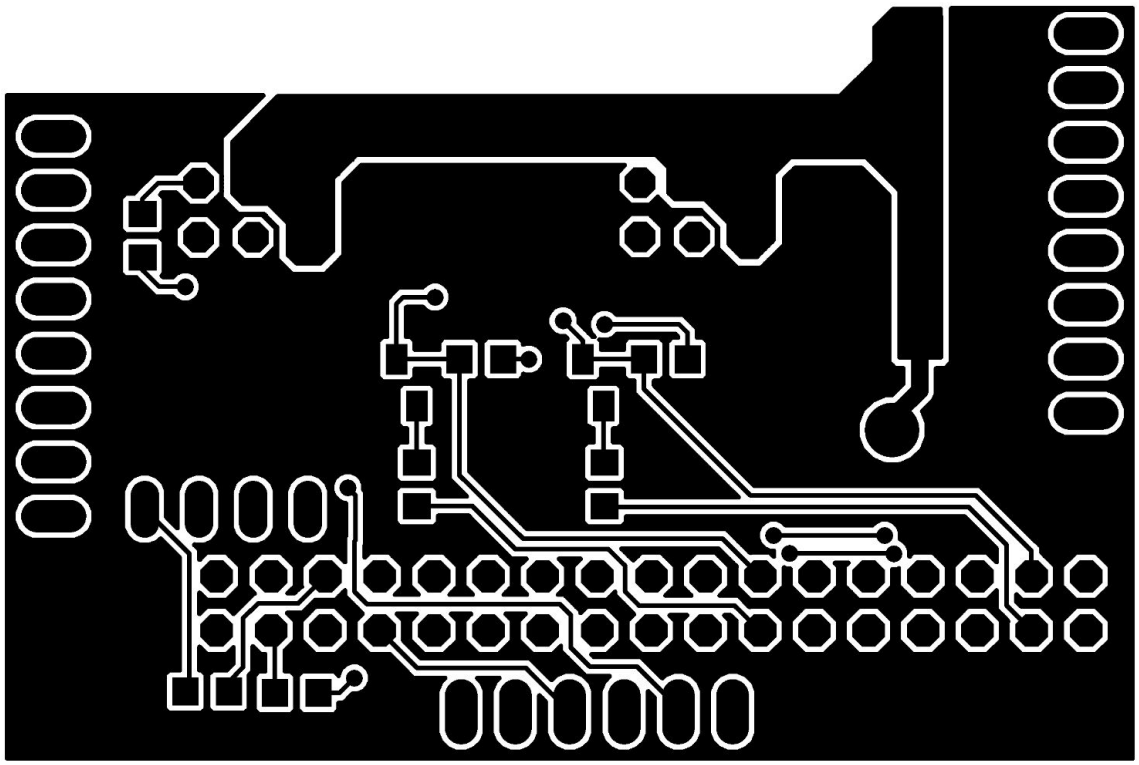
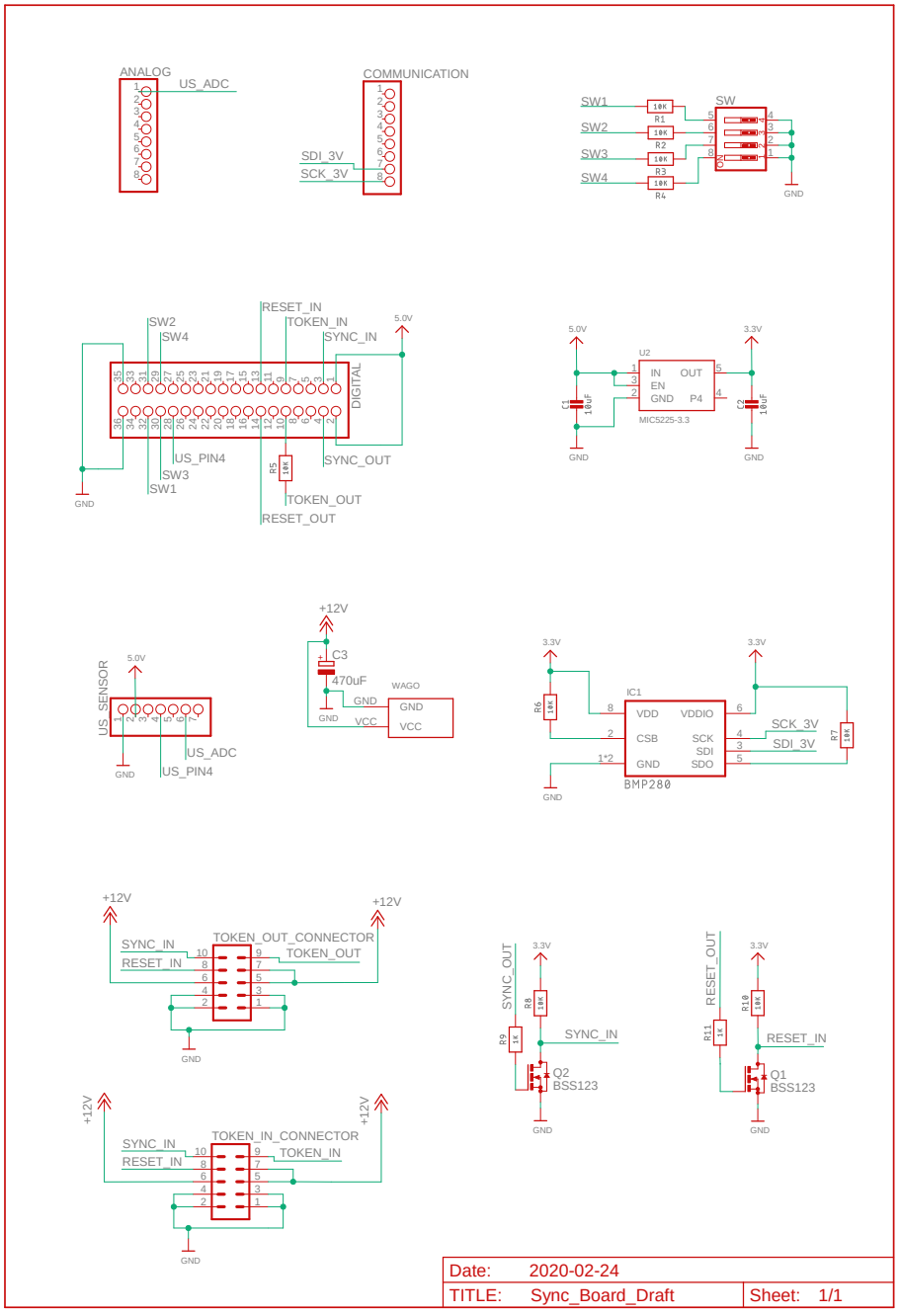


Abbildung A.3.: Platinenlayout der Unterseite

A.3. Schaltplan der Platine



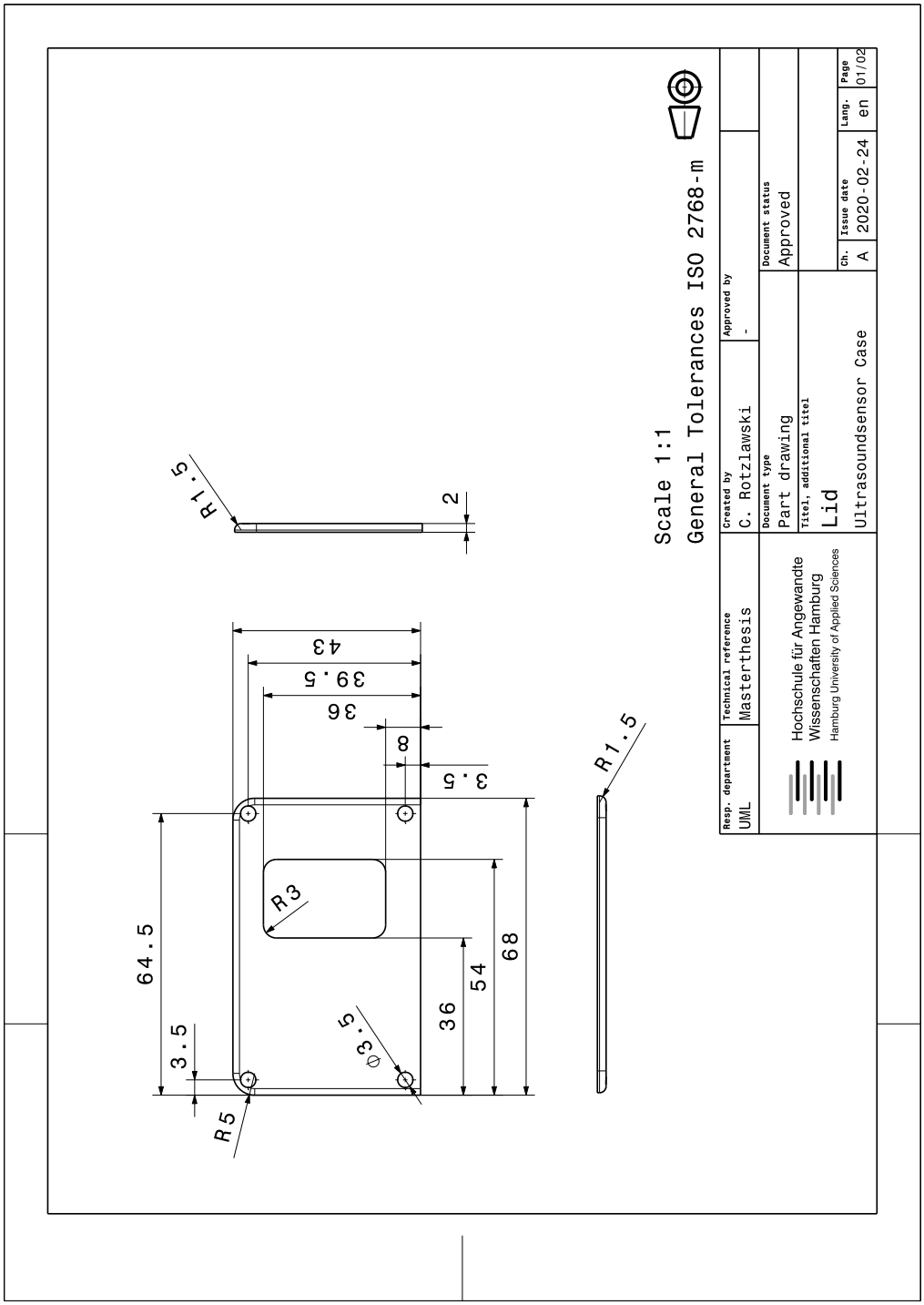
Sync Board	**Arduino**	**Sync Board**	**Arduino**
US_ADC	ADC10	US_PIN4	D47
SDI_3V	TWI1 SDA	SCK_3V	TWI1 SCL
SW1	D51	SW2	D50
SW3	D49	SW4	D48
RESET_IN	D32	RESET_OUT	D33
TOKEN_IN	D28	TOKEN_OUT	D29
SYNC_IN	D24	SYNC_OUT	D25

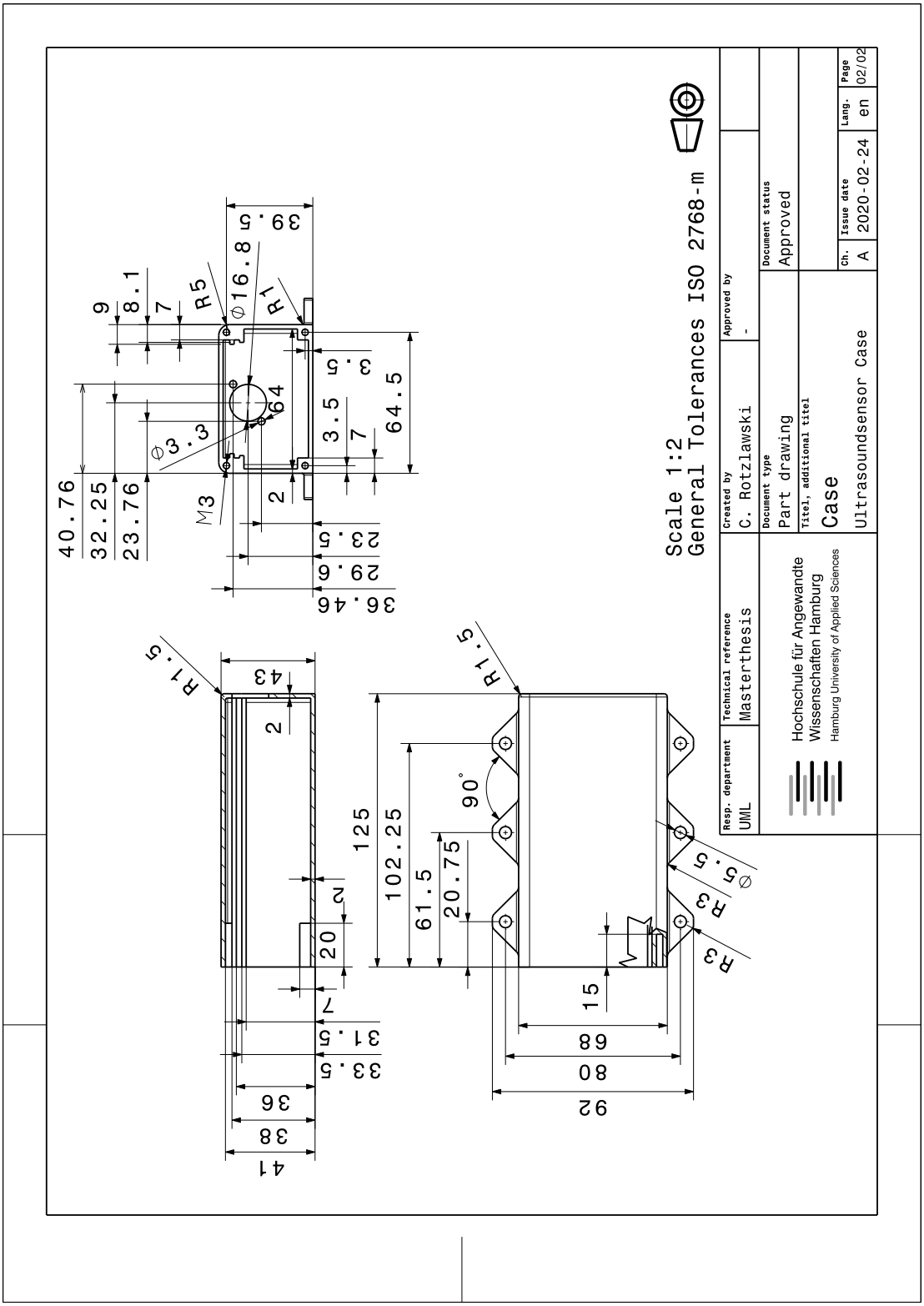
Tabelle A.2.: Pinbelegung des Synchronisationsboards auf dem Arduino Due

Bezeichnung	**Bauteil / Wert**
R1 ... R8, R10	10 $k\Omega$ SMD-Widerstand
R9, R11	1k $k\Omega$ SMD-Widerstand
C1, C2	10 μF SMD-Kondensator
C3	470 μF Elektrolytkondensator
Q1, Q2	BSS123 N-MOSFET
U2	MIC5225-3.3 Spannungsregler
IC1	Bosch BMP280
SW	Dip-Schalter 4-polig
WAGO	SMD-Leiterplattenklemme 2-polig
US_SENSOR	Stiftleiste 7-polig
ANALOG, COMMUNICATION	Stiftleiste 8-polig
DIGITAL	Stiftleiste 2x18-polig
TOKEN_IN_CONNECTOR, TOKEN_OUT_CONNECTOR	Wannenstecker 2x5-polig

Tabelle A.3.: Bauteilliste des Synchronisationsboards

A.4. Technische Zeichnungen des Ultraschallsensorgehäuses




Scale 1:2
General Tolerances ISO 2768-m
Resp. department
UML
Technical reference
Masterthesis
Created by
C. Rotzlawski
Approved by
-
Document type
Part drawing
Document status
Approved
Titel, additional titel
Case
Ultrasoundsensor Case
Ch.
A
Issue date
2020-02-24
Lang.
en
Page
02/02
Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences
39.5
16.8
8.1
9
7
R5
R1
3.5
64.5
40.76
32.25
23.76
M3
36.46
29.6
23.5
43
2
20
125
102.25
61.5
20.75
90°
15
68
80
92
41
38
36
33.5
31.5

B. Programmabläufe

B.1. Programmabläufe des Ultraschallsensors

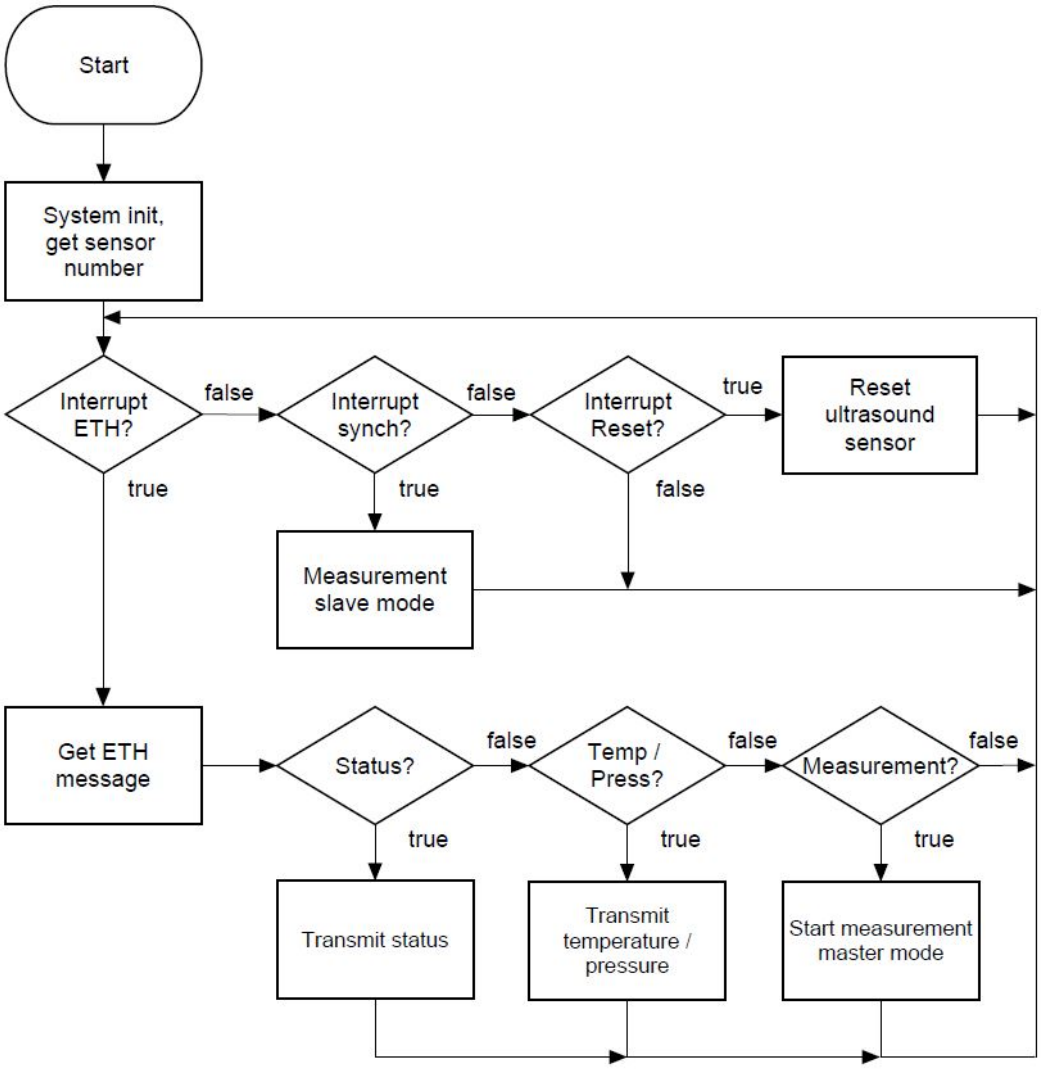


Abbildung B.1.: Ablauf des Hauptprogramms des Ultraschallsensors

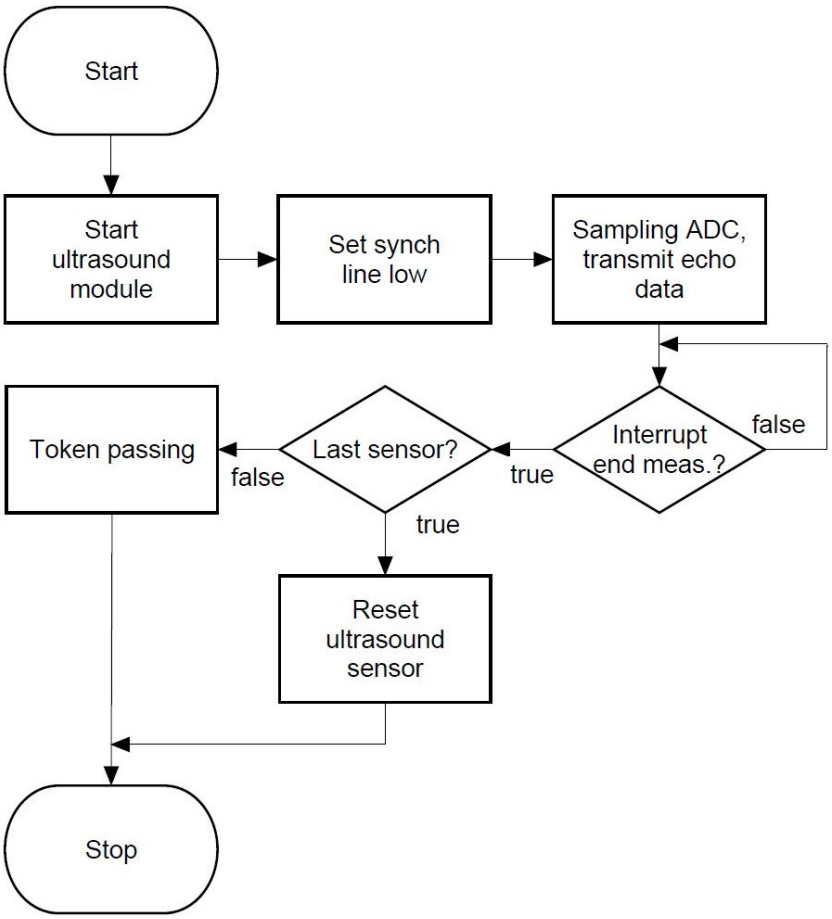


Abbildung B.2.: Ablauf des Mastermode des Ultraschallsensors

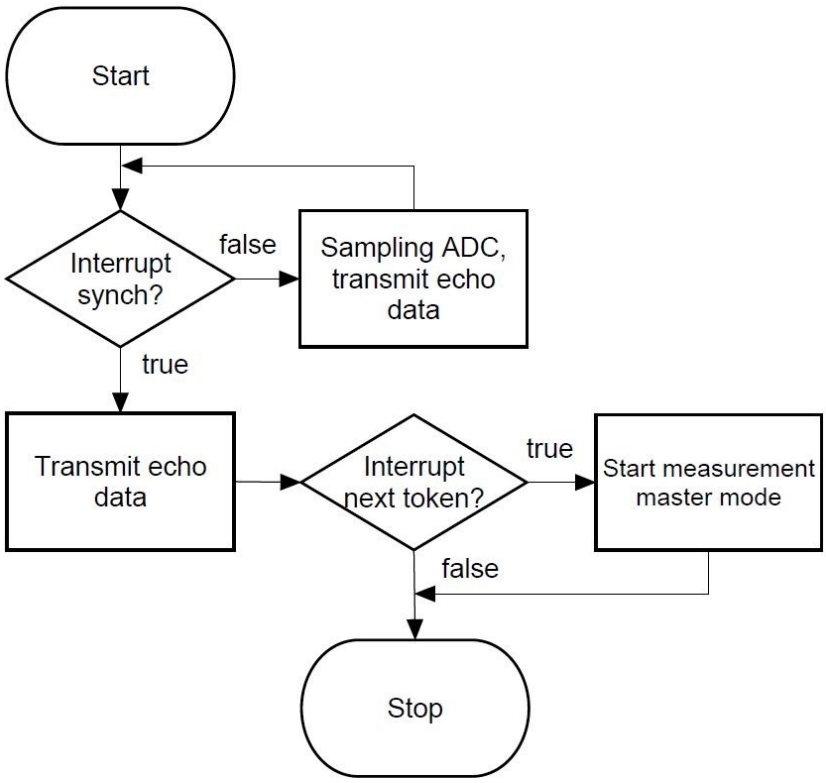


Abbildung B.3.: Ablauf des Slavemode des Ultraschallsensors

B.2. Programmabläufe des ultraschallbasierten Positionierungssystems

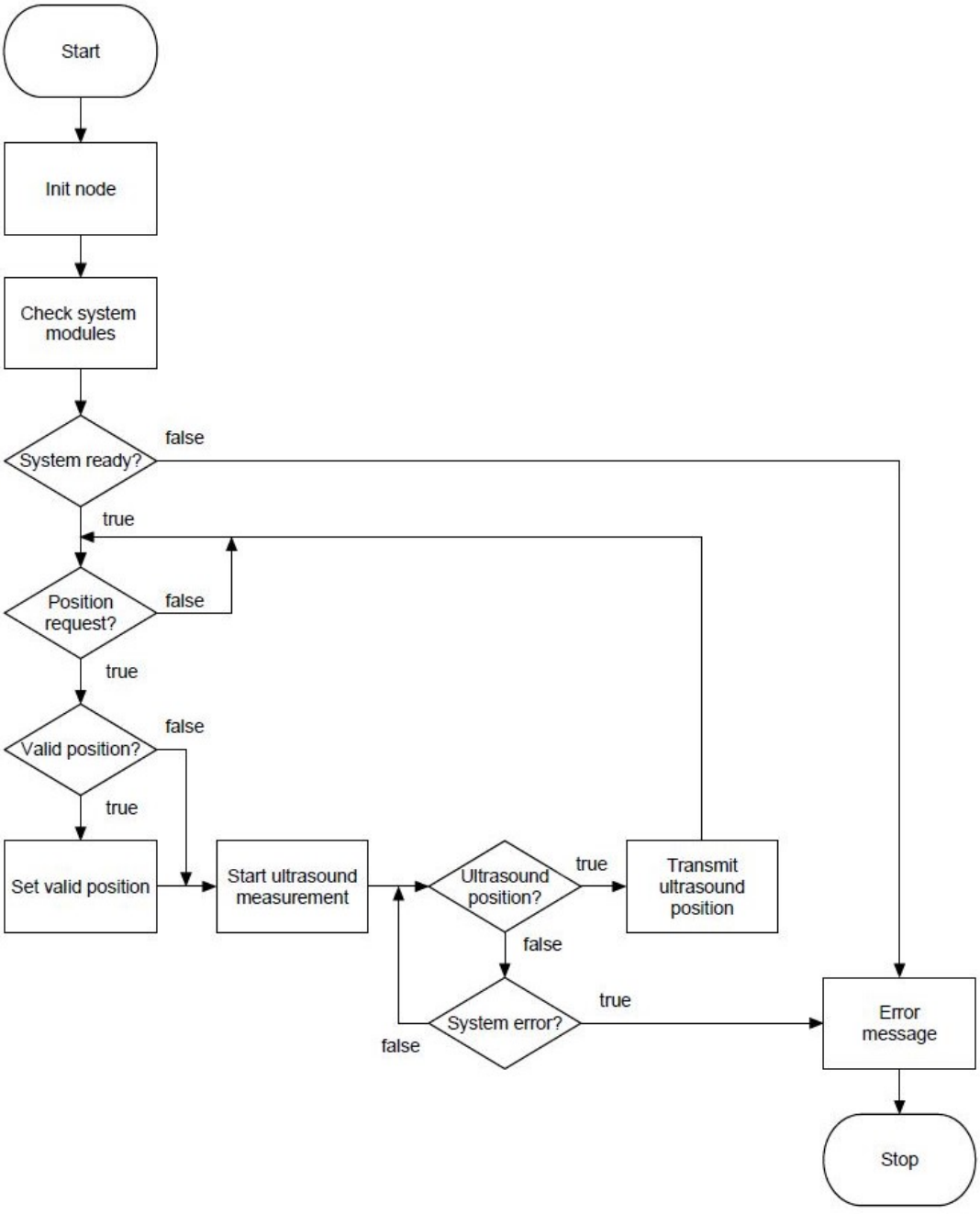


Abbildung B.4.: Ablauf der Systemschnittstelle des ultraschallbasierten Positionierungssystems

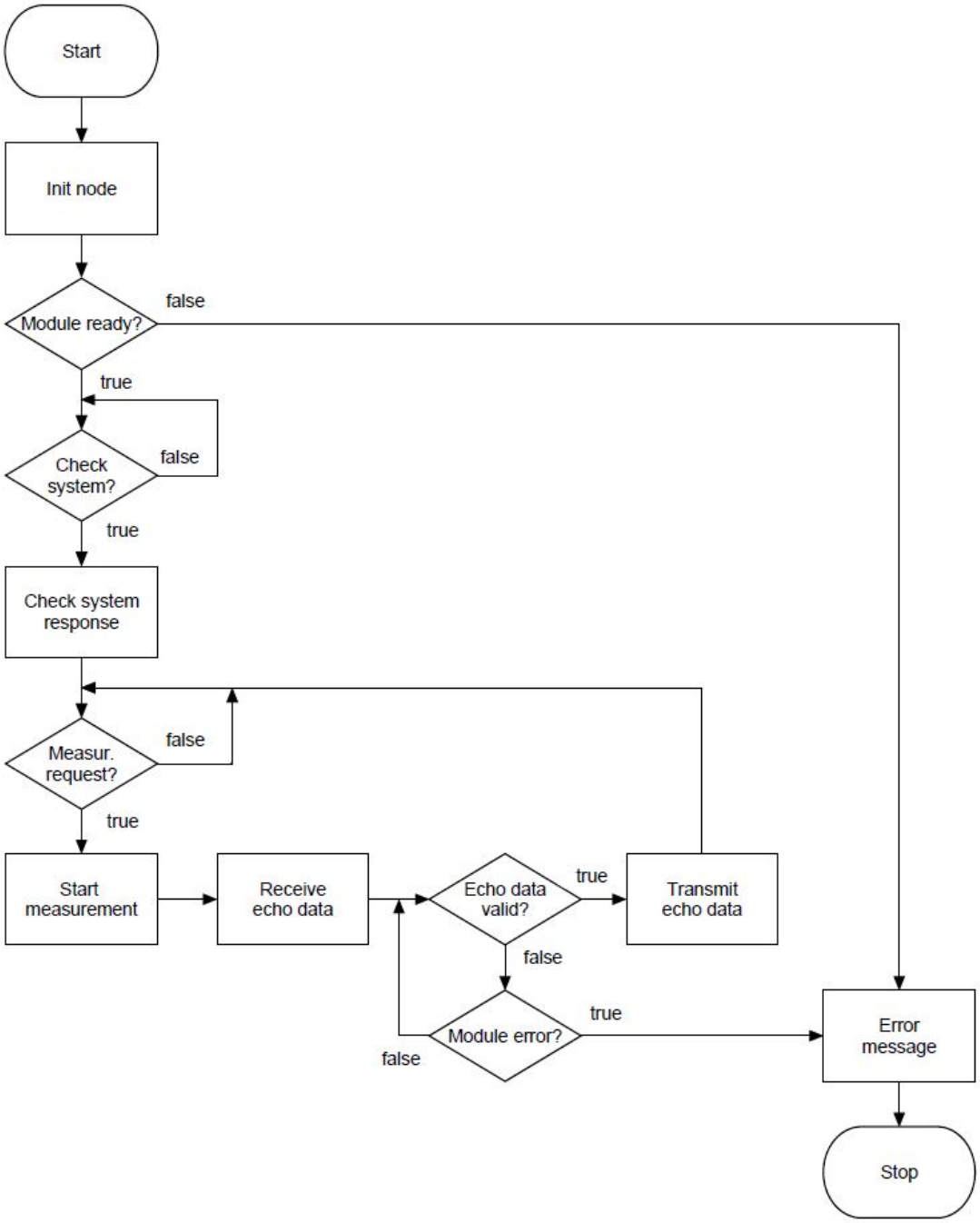


Abbildung B.5.: Ablauf der Ultraschallsensorschnittstelle des ultraschallbasierten Positionierungssystems

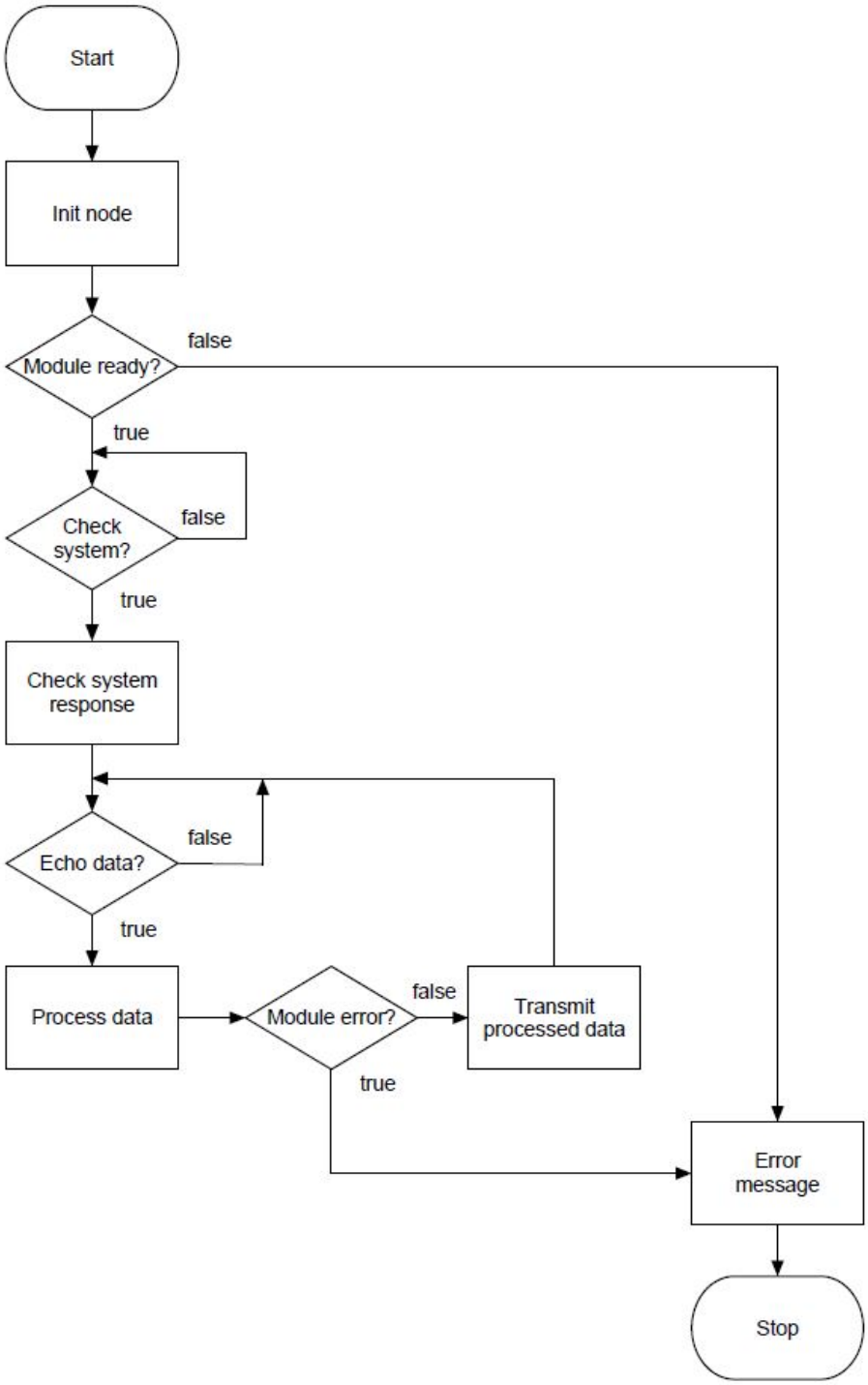


Abbildung B.6.: Ablauf der Ultraschallmerkmalsextraktion des ultraschallbasierten Positionierungssystems

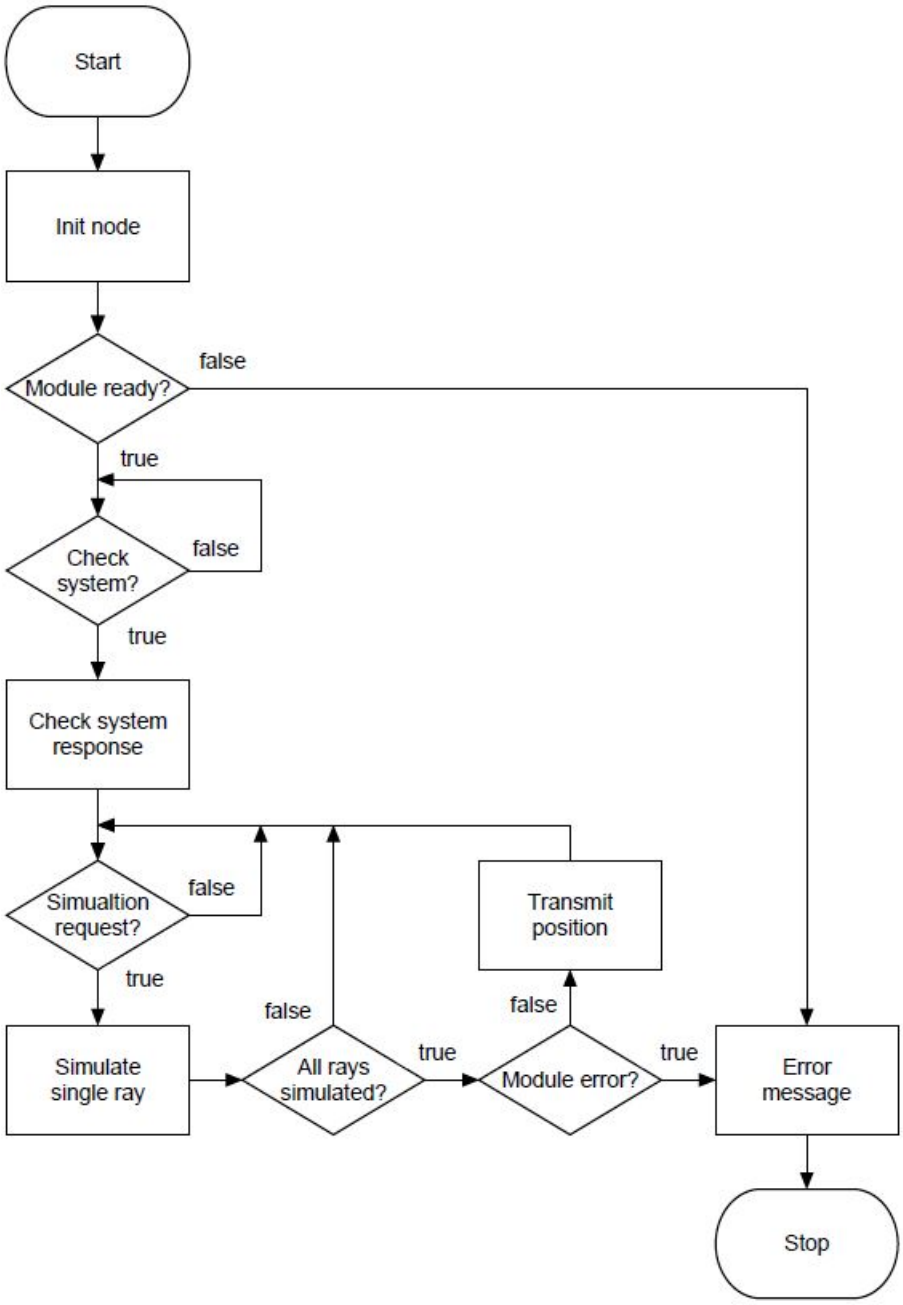


Abbildung B.7.: Ablauf der 3D-Raytracing-Simulation des ultraschallbasierten Positionierungssystems

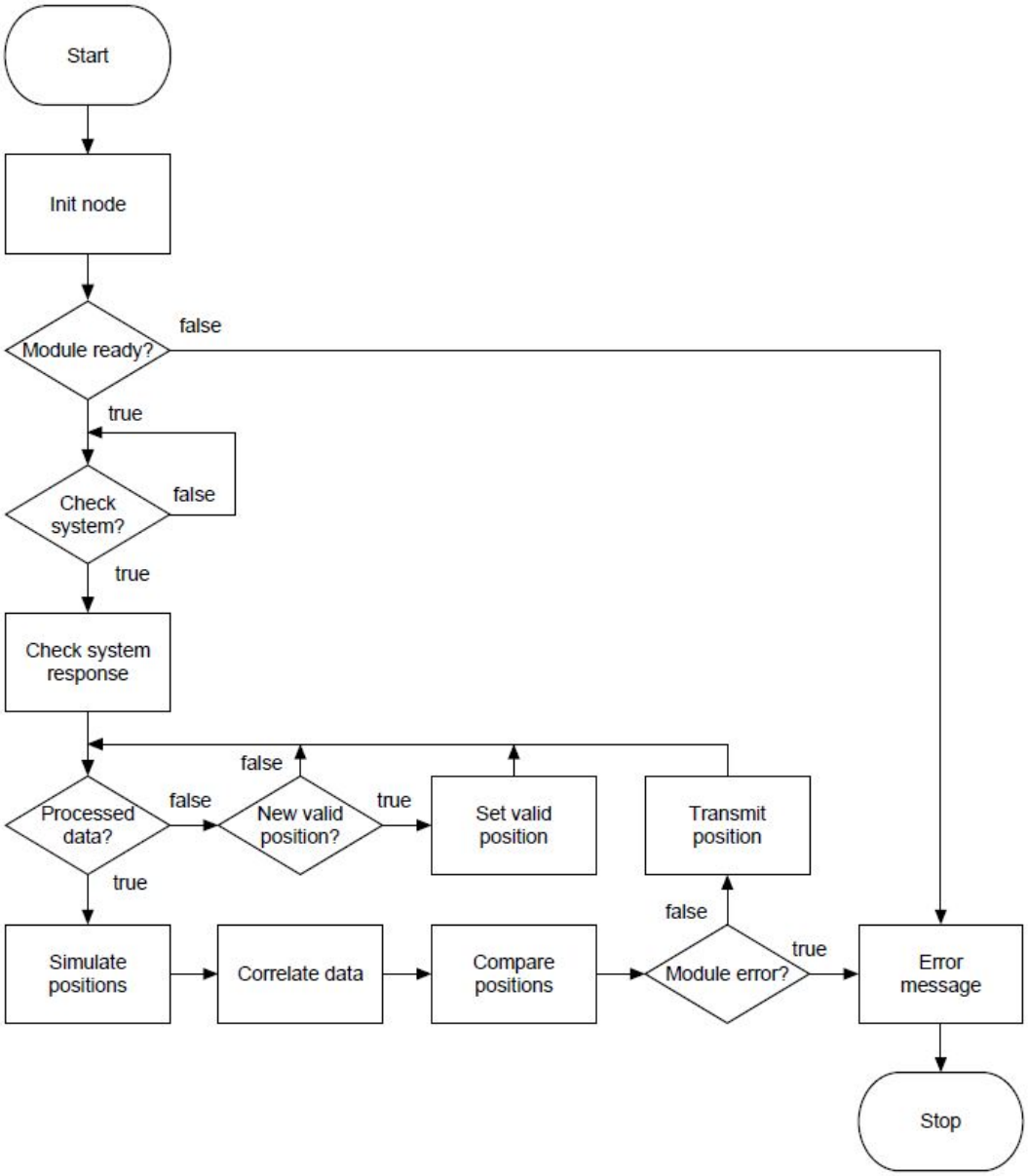


Abbildung B.8.: Ablauf der Positionsbestimmung des ultraschallbasierten Positionierungssystems

C. Quellcode des Ultraschallsensors

main.c

```
1  /*****************************************************************/
2  /* University: HAW Hamburg                                       */
3  /* Author: Christopher Rotzlawski                                */
4  /*                                                               */
5  /* Project: US_Sensor_Firmware                                   */
6  /* Version: 1.0                                                  */
7  /*****************************************************************/
8
9  #include <asf.h>
10 #include <conf_board.h>
11 #include <conf_uart_serial.h>
12 #include <string.h>
13 #include <w5500.h>
14 #include <bmp280.h>
15
16 /*****************************************************************/
17 /*                          Definitions                          */
18 /*****************************************************************/
19 // Debug mode
20 #define DEBUGGING            // Uncommented for save ethernet transmission
21 // Token passing mode
22 //#define TOKEN_PASSING      // Uncomment for token passing
23 // Sensor number automatic
24 #define AUTO_SENSOR_NUMBER
25 // Sensor number
26 #define SENSOR 1             // Default
27 #define SENSOR_OFFSET 101
28 // Master mode
29 #define MASTER 1
30 #define SLAVE 0
31 // Sensor mode definitions
32 #define STATUS_MODE 0x4242
33 #define TEMP_PRESS_MODE 0xABCD
34 #define MEASUREMENT_MODE 0xF00F
35 // General definitions
36 #define MEGA_HZ 1000000
37 #define KILO_HZ 1000
38 #define DIV_MS 1000
39 #define DIV_US 1000000
40 #define BYTE_0_MASK 0xFF
41 #define BYTE_1_MASK 0xFF00
42 #define BYTE_2_MASK 0xFF0000
43 #define BYTE_3_MASK 0xFF000000
44 #define BYTE_0_SHIFT 0
45 #define BYTE_1_SHIFT 8
46 #define BYTE_2_SHIFT 16
47 #define BYTE_3_SHIFT 24
48 // SPI definitions
49 #define SPI_CLOCK 21
50 #define SPI_CHIP_SEL 0
51 #define SPI_CHIP_PCS spi_get_pcs(SPI_CHIP_SEL)
52 #define SPI_CLK_POLARITY 1
53 #define SPI_CLK_PHASE 0
54 #define SPI_DLYBS 0x00
55 #define SPI_DLYBCT 0x00
56 // TWI definitions
57 #define TWI_CLK 400000
58 // Ethernet Shield definitions
59 #define MAX_DATA_FRAME_SIZE 20
60 #define BUFFER_ECHO_SIZE_KB 2
61 #define BUFFER_TEMP_PRESS_KB 2
62 #define BUFFER_STATUS_KB 2
63 #define BUFFER_RX_SIZE_KB 2
64 #define MAX_ETHERNET_PAYLOAD 1460
65 #define TRANSMIT_SOCKET 0
66 #define RECEIVE_SOCKET TRANSMIT_SOCKET
67 #define SEND_MANUAL 1
```

```
68  #define SEND_AUTO 0
69  // Communication
70  #define END_MEASUREMENT 0xF0420000
71  #define NULL_MESSAGE 0x0000
72  #define ERROR_MESSAGE 0xFFFFFFFF
73  // ADC definitions
74  #define SAMPLE_RATE_ADC_KHZ 100
75  #define ADC_CLOCK 6400000
76  // Interrupt definitions
77  #define INT_PRIOR_ETHERNET 3
78  #define INT_PRIOR_ADC 2
79  #define INT_PRIOR_TC 1
80  // Timer definitions
81  #define TIMER_MCK_2 2
82  #define POWER_UP 200              // Wait time for power up in ms
83  #define START_MEASUREMENT 20      // Time to start measurment in ms
84  #define MEASUREMENT 100           // Measurement time in ms
85  #define TC_CHANNEL0 0
86  #define TC_CHANNEL1 1
87  #define TC_CHANNEL2 2
88  #define TC_CHANNEL3 3
89
90  /*****************************************************************/
91  /*                          Declarations                         */
92  /*****************************************************************/
93  // Network informations of the Sensor
94  wiz_NetInfo Sensor_Info = {
95      .mac = {0x42,0x42,0x42,0x42,0x42,SENSOR},
96      .ip = {192,168,42,(SENSOR+SENSOR_OFFSET)},
97      .sn = {255,255,255,0},
98      .gw = {192,168,42,0},
99  };
100 const uint8_t destination_ip[4] = {192,168,42,SENSOR_OFFSET-1};
101 const uint8_t port_base[2] = {0xC3,0x50};       // 50000
102 // Power up
103 volatile uint8_t power_nready = 1;
104 // Master status
105 volatile uint8_t master_status = SLAVE;
106 // UDP package counter
107 volatile uint16_t udp_counter = 0;
108
109 /*****************************************************************/
110 /*                      Function Prototypes                      */
111 /*****************************************************************/
112 static void system_init(void);
113 static void spi_master_init(void);
114 static void spi_transmit(uint32_t addr,uint8_t *data,uint8_t data_len);
115 static void spi_receive(uint32_t addr,uint8_t *data,uint8_t data_len);
116 static void ethernet_init(void);
117 static void ethernet_init_socket(uint8_t socket,uint8_t buffer_size);
118 static void ethernet_transmit_echo(uint16_t *data,uint8_t send_manuell);
119 static void ethernet_transmit_data(uint32_t *data,uint8_t data_len);
120 static void ethernet_receive_data(uint32_t *data);
121 static void twi_init(void);
122 static void bmp280_sensor_init(void);
123 static void bmp280_get_calibration(uint8_t *bmp280_calib);
124 static void bmp280_measurement(void);
125 static void adc_init_config(void);
126 static void timer_init(void);
127 static void timer_power_up(void);
128 static void nvicc_init(void);
129 static void sync_pins_init(void);
130 static void Int_Handler_Ethernet(uint32_t ul_id,uint32_t ul_mask);
131 static void Int_Handler_Sync(uint32_t ul_id,uint32_t ul_mask);
132 static void Int_Handler_Next_Token(uint32_t ul_id,uint32_t ul_mask);
133 static void Int_Handler_Reset(uint32_t ul_id,uint32_t ul_mask);
134 static void init_sensor_pins(void);
135 static void init_sensor_number_pins(void);
136 static void get_sensor_number(uint8_t *number);
137 // Debug function
138 #ifdef DEBUGGING
139 static void debugging_init(void);
140 static void configure_console(void);
141 #endif
142
143 /*****************************************************************/
144 /*                          Main Function                        */
145 /*****************************************************************/
146 int main(void)
147 {
148     // Init microcontroller
149     system_init();
150
151 #ifdef DEBUGGING
152     puts("System started\r");
```

```
153 #endif
154
155     while (1){
156         __asm__ __volatile__ ("nop");
157     }
158
159     return 0;
160 }
161
162 /*****************************************************************/
163 /*                      Interrupt Handler                        */
164 /*****************************************************************/
165 // ADC handler
166 void ADC_Handler(void)
167 {
168     static uint16_t adc_value;
169
170     // Get value from register
171     adc_value = (uint16_t)(ADC->ADC_LCDR & ADC_LCDR_LDATA_Msk);
172
173     // Transmit value via ethernet in automatic mode
174     ethernet_transmit_echo(&adc_value,SEND_AUTO);
175 }
176
177 // Timer0 A1 power up
178 void TC1_Handler(void)
179 {
180     // Disable interrupt Timer0 A1
181     tc_get_status(TC0,TC_CHANNEL1);
182     tc_disable_interrupt(TC0,TC_CHANNEL1,TC_IER_CPCS);
183
184     // Set power up ready
185     power_nready = 0;
186 }
187
188 // Timer0 A2 start
189 void TC2_Handler(void)
190 {
191     // Disable interrupt Timer0 A2 start time
192     tc_get_status(TC0,TC_CHANNEL2);
193     tc_disable_interrupt(TC0,TC_CHANNEL2,TC_IER_CPCS);
194
195     // Set US-Sensor pin 4 low
196     ioport_set_pin_level(PIO_PC16_IDX,IOPORT_PIN_LEVEL_LOW);
197
198     // Set sync link high
199     ioport_set_pin_level(PIO_PA14_IDX,IOPORT_PIN_LEVEL_HIGH);
200
201     // Start Timer1 A3 for measurement time
202     tc_enable_interrupt(TC1,TC_CHANNEL0,TC_IER_CPCS);
203     tc_start(TC1,TC_CHANNEL0);
204 }
205
206 // Timer1 A3 measurement
207 void TC3_Handler(void)
208 {
209     // Set sync link low
210     ioport_set_pin_level(PIO_PA14_IDX,IOPORT_PIN_LEVEL_LOW);
211
212     // Disable interrupt Timer1 A3 measurement time
213     tc_get_status(TC1,TC_CHANNEL0);
214     tc_disable_interrupt(TC1,TC_CHANNEL0,TC_IER_CPCS);
215
216     // Enable reset interrupt
217     pio_enable_interrupt(PIOD,PIO_PD10);
218
219     // Set next token high
220     ioport_set_pin_level(PIO_PD6_IDX,IOPORT_PIN_LEVEL_HIGH);
221 }
222
223 // Ethernet shield handler
224 static void Int_Handler_Ethernet(uint32_t ul_id,uint32_t ul_mask)
225 {
226     uint8_t clear_reg[1] = {Sn_IR_RECV};
227     uint32_t data_receive[2];
228     uint32_t data_transmit[1];
229     uint16_t identifier;
230     uint16_t check;
231
232     // Disable interrupt ethernet shield
233     pio_disable_interrupt(PIOB,PIO_PB25);
234
235     // Get ethernet data
236     ethernet_receive_data(&data_receive[0]);
237     spi_transmit((Sn_IR(RECEIVE_SOCKET) | _W5500_SPI_WRITE_),&clear_reg[0],1);
```

```
238
239         // Get identifier and check
240         identifier = (uint16_t)(data_receive[0] >> BYTE_2_SHIFT);
241         check = (uint16_t)(data_receive[0] & 0xFFFF);
242
243         // Status query
244         if (identifier == STATUS_MODE)
245         {
246             // Response for status query
247             data_transmit[0] = (STATUS_MODE << BYTE_2_SHIFT) | check;
248
249             // Transmit response
250             ethernet_transmit_data(&data_transmit[0],1);
251
252             // Enable interrupt ethernet shield
253             pio_enable_interrupt(PIOB,PIO_PB25);
254
255             // Enable interrupt sync
256             pio_enable_interrupt(PIOB,PIO_PB26);
257         }
258         // Temperature and pressure query
259         else if (identifier == TEMP_PRESS_MODE)
260         {
261             // Response for temperature and pressure query
262             data_transmit[0] = (TEMP_PRESS_MODE << BYTE_2_SHIFT) | check;
263
264             // Transmit response
265             ethernet_transmit_data(&data_transmit[0],1);
266
267             // Start measuring and transmit data
268             bmp280_measurement();
269
270             // Enable interrupt ethernet shield
271             pio_enable_interrupt(PIOB,PIO_PB25);
272         }
273         // Measurement mode
274         else if (identifier == MEASUREMENT_MODE)
275         {
276             // Response for master
277             data_transmit[0] = (MEASUREMENT_MODE << BYTE_2_SHIFT) | check;
278
279             // Transmit response
280             ethernet_transmit_data(&data_transmit[0],1);
281
282             // Set master status
283             master_status = MASTER;
284
285             // Set US-Sensor pin 4 high
286             ioport_set_pin_level(PIO_PC16_IDX,IOPORT_PIN_LEVEL_HIGH);
287
288             // Start timer0 A2 start
289             tc_enable_interrupt(TC0,TC_CHANNEL2,TC_IER_CPCS);
290             tc_start(TC0,TC_CHANNEL2);
291         }
292         // Default
293         else
294         {
295             // Response for error
296             data_transmit[0] = ERROR_MESSAGE;
297
298             // Transmit response
299             ethernet_transmit_data(&data_transmit[0],1);
300
301             // Enable interrupt ethernet shield
302             pio_enable_interrupt(PIOB,PIO_PB25);
303         }
304     }
305
306     // Sync handler measurement
307     static void Int_Handler_Sync(uint32_t ul_id,uint32_t ul_mask)
308     {
309         uint16_t null_value = NULL_MESSAGE;
310         uint32_t data_transmit[1];
311
312         // Disable interrupt PB26
313         pio_disable_interrupt(PIOB,PIO_PB26);
314
315         // If pin level high, start ADC
316         if (!ioport_get_pin_level(PIO_PB26_IDX))
317         {
318             // Change interrupt edge
319             pio_handler_set(PIOB,ID_PIOB,PIO_PB26,(PIO_PULLUP | PIO_IT_RISE_EDGE),Int_Handler_Sync);
320
321             // Start timer0 A0 ADC
322             tc_start(TC0,TC_CHANNEL0);
```

```
323             adc_enable_interrupt(ADC,ADC_IER_DRDY);
324
325             // Set next token low
326             ioport_set_pin_level(PIO_PD6_IDX,IOPORT_PIN_LEVEL_LOW);
327
328             // Disable next token interrupt
329             pio_disable_interrupt(PIOD,PIO_PD3);
330
331             // Disable reset interrupt
332             pio_disable_interrupt(PIOD,PIO_PD10);
333         }
334         // If pin level low, stop ADC
335         else
336         {
337             // Stop Timer0 A0 ADC
338             tc_stop(TC0,TC_CHANNEL0);
339             adc_disable_interrupt(ADC,ADC_IER_DRDY);
340
341             // Enable next token interrupt
342             pio_enable_interrupt(PIOD,PIO_PD3);
343
344             // Enable reset interrupt
345             pio_enable_interrupt(PIOD,PIO_PD10);
346
347             // Transmit remaining echo data
348             ethernet_transmit_echo(&null_value,SEND_MANUAL);
349
350             // Response for status end measurement
351             data_transmit[0] = END_MEASUREMENT | (udp_counter - 1);
352
353             // Transmit status response
354             ethernet_transmit_data(&data_transmit[0],1);
355
356             // Change interrupt edge
357             pio_handler_set(PIOB,ID_PIOB,PIO_PB26,(PIO_PULLUP | PIO_IT_FALL_EDGE),Int_Handler_Sync);
358         }
359
360         // Enable interrupt PB26
361         pio_enable_interrupt(PIOB,PIO_PB26);
362     }
363
364     // Next token handler (next master)
365     static void Int_Handler_Next_Token(uint32_t ul_id,uint32_t ul_mask)
366     {
367         if (master_status)
368         {
369             // Set reset high
370             ioport_set_pin_level(PIO_PC1_IDX,IOPORT_PIN_LEVEL_HIGH);
371         }
372         else
373         {
374     #ifdef TOKEN_PASSING
375             // Disable interrupts
376             pio_disable_interrupt(PIOD,PIO_PD3);
377             pio_disable_interrupt(PIOD,PIO_PD10);
378
379             // Set US-Sensor pin 4 high
380             ioport_set_pin_level(PIO_PC16_IDX,IOPORT_PIN_LEVEL_HIGH);
381
382             // Start timer0 A2 start
383             tc_enable_interrupt(TC0,TC_CHANNEL2,TC_IER_CPCS);
384             tc_start(TC0,TC_CHANNEL2);
385     #else
386             // Set reset high
387             ioport_set_pin_level(PIO_PC1_IDX,IOPORT_PIN_LEVEL_HIGH);
388     #endif
389         }
390     }
391
392     // Reset handler for end measurment
393     static void Int_Handler_Reset(uint32_t ul_id,uint32_t ul_mask)
394     {
395         // Disable interrupts
396         pio_disable_interrupt(PIOD,PIO_PD3);
397         pio_disable_interrupt(PIOD,PIO_PD10);
398
399         // Set pin level low
400         ioport_set_pin_level(PIO_PA14_IDX,IOPORT_PIN_LEVEL_LOW);
401         ioport_set_pin_level(PIO_PC1_IDX,IOPORT_PIN_LEVEL_LOW);
402         ioport_set_pin_level(PIO_PD6_IDX,IOPORT_PIN_LEVEL_LOW);
403
404         // Set UDP package counter to null
405         udp_counter = 0;
406
407         // Set master status
```

```
408     master_status = SLAVE;
409
410     // Enable interrupt ethernet shield
411     pio_enable_interrupt(PIOB,PIO_PB25);
412 }
413
414 /*****************************************************************/
415 /*                      Function Implementation                  */
416 /*****************************************************************/
417 // Initialize microcontrollerboard
418 static void system_init(void)
419 {
420     sysclk_init();                          // Init system clock
421     board_init();                           // Init board
422     ioport_init();                          // Init ports
423     spi_master_init();                      // Init SPI
424 #ifdef DEBUGGING
425     debugging_init();                       // Init debug mode
426     configure_console();                    // Configure UART for debug message output
427 #endif
428     init_sensor_number_pins();              // Init pins for sensor number
429     ethernet_init();                        // Init ethernet shield
430     twi_init();                             // Init I2C
431     bmp280_sensor_init();                   // Init BMP280 Sensor
432     timer_power_up();                       // Init Timer power up and start
433     do
434     {
435         __asm__ __volatile__ ("nop");
436     } while(power_nready == 1);             // Wait for sensor power up
437     adc_init_config();                      // Init ADC
438     timer_init();                           // Init Timer
439     nvicc_init();                           // Init NVICC
440     init_sensor_pins();                     // Init US-Sensor pins
441     sync_pins_init();                       // Init sync pin
442 }
443
444 // Init SPI master
445 static void spi_master_init(void)
446 {
447     spi_enable_clock(SPI_MASTER_BASE);
448     spi_disable(SPI_MASTER_BASE);
449     spi_reset(SPI_MASTER_BASE);
450     spi_set_lastxfer(SPI_MASTER_BASE);
451     spi_set_master_mode(SPI_MASTER_BASE);
452     spi_disable_mode_fault_detect(SPI_MASTER_BASE);
453     spi_set_peripheral_chip_select_value(SPI_MASTER_BASE,SPI_CHIP_PCS);
454     // SPI mode 2
455     spi_set_clock_polarity(SPI_MASTER_BASE,SPI_CHIP_SEL, SPI_CLK_POLARITY);
456     spi_set_clock_phase(SPI_MASTER_BASE,SPI_CHIP_SEL,SPI_CLK_PHASE);
457     spi_set_bits_per_transfer(SPI_MASTER_BASE,SPI_CHIP_SEL,SPI_CSR_BITS_8_BIT);
458     // Safe transfer at 21 MHz
459     spi_set_baudrate_div(SPI_MASTER_BASE,SPI_CHIP_SEL,(sysclk_get_peripheral_hz() / SPI_CLOCK / MEGA_HZ));
460     // Disable delay before and between transfers
461     spi_set_transfer_delay(SPI_MASTER_BASE,SPI_CHIP_SEL,SPI_DLYBS,SPI_DLYBCT);
462     spi_enable(SPI_MASTER_BASE);
463 }
464
465 // Transmit data via SPI
466 static void spi_transmit(uint32_t addr,uint8_t *data,uint8_t data_len)
467 {
468     static uint8_t frame[MAX_DATA_FRAME_SIZE];
469
470     // Ethernet shield address and control
471     frame[0] = (addr & BYTE_2_MASK) >> BYTE_2_SHIFT;
472     frame[1] = (addr & BYTE_1_MASK) >> BYTE_1_SHIFT;
473     frame[2] = (addr & BYTE_0_MASK) >> BYTE_0_SHIFT;
474     // Raw data
475     for (uint8_t i = 0; i < data_len; i++)
476     {
477         frame[i+3] = data[i];
478     }
479
480     // Transmit data
481     for (uint8_t i = 0; i < data_len + 3; i++)
482     {
483         // If TX register not empty, wait
484         while (!(SPI_MASTER_BASE->SPI_SR & SPI_SR_TDRE));
485         SPI_MASTER_BASE->SPI_TDR = SPI_TDR_TD(frame[i]);
486     }
487 }
488
489 // Receive data via SPI
490 static void spi_receive(uint32_t addr,uint8_t *data,uint8_t data_len)
491 {
492     static uint8_t frame[MAX_DATA_FRAME_SIZE];
```

```
493     uint8_t frame_len = 3 + data_len;
494
495     // Ethernet shield address and control
496     frame[0] = (addr & BYTE_2_MASK) >> BYTE_2_SHIFT;
497     frame[1] = (addr & BYTE_1_MASK) >> BYTE_1_SHIFT;
498     frame[2] = (addr & BYTE_0_MASK) >> BYTE_0_SHIFT;
499
500     // Receive data
501     for (uint8_t i = 0; i < frame_len; i++)
502     {
503         while (!(SPI_MASTER_BASE->SPI_SR & SPI_SR_TDRE));
504         SPI_MASTER_BASE->SPI_TDR = SPI_TDR_TD(frame[i]);
505         frame[i] = (uint8_t)(SPI_MASTER_BASE->SPI_RDR & SPI_RDR_RD_Msk);
506     }
507
508     // Delay of 3 bits
509     for (uint8_t i = 0; i < data_len; i++)
510     {
511         data[i] = frame[i+3];
512     }
513 }
514
515 // Init ethernet shield
516 static void ethernet_init(void)
517 {
518     uint8_t phy_reset[1] = {0xFF & PHYCFGR_RST};
519     uint8_t phy_config[1] = {~PHYCFGR_RST | PHYCFGR_OPMD | PHYCFGR_OPMDC_100F};
520     uint8_t phy_check[3] = {};
521     uint8_t mode_reg[1] = {MR_RST | MR_FARP | MR_WOL | MR_PB};
522     uint8_t interrupt_mask[1] = {0};
523     uint8_t socket_int_mask[1] = {0x1};
524     uint8_t socket_buf_rx_size[1] = {BUFFER_RX_SIZE_KB};
525     uint8_t socket_ready[3] = {Sn_CR_SEND};
526
527 #ifdef AUTO_SENSOR_NUMBER
528     // Get sensor number
529     uint8_t sensor_number;
530     get_sensor_number(&sensor_number);
531
532     // Set sensor number
533     Sensor_Info.mac[5] = sensor_number;
534     Sensor_Info.ip[3] = sensor_number + SENSOR_OFFSET;
535 #endif
536
537     // Set pins as input, not to disturb SPI
538     ioport_enable_pin(PIO_PB27_IDX);
539     ioport_set_pin_dir(PIO_PB27_IDX,IOPORT_DIR_INPUT);
540     ioport_enable_pin(PIO_PD8_IDX);
541     ioport_set_pin_dir(PIO_PD8_IDX,IOPORT_DIR_INPUT);
542     ioport_enable_pin(PIO_PD7_IDX);
543     ioport_set_pin_dir(PIO_PD7_IDX,IOPORT_DIR_INPUT);
544
545     // Reset and configure PHY register
546     spi_transmit((PHYCFGR | _W5500_SPI_WRITE_),&phy_reset[0],1);
547     spi_transmit((PHYCFGR | _W5500_SPI_WRITE_),&phy_config[0],1);
548     do
549     {
550         // Wait if link down
551         spi_receive(PHYCFGR,&phy_check[0],3);
552     } while (!(phy_check[2] & PHYCFGR_LNK_ON));
553
554     // Configure mode register and interrupts
555     spi_transmit((MR | _W5500_SPI_WRITE_),&mode_reg[0],1);
556     spi_transmit((_IMR_ | _W5500_SPI_WRITE_),&interrupt_mask[0],1);
557     spi_transmit((SIMR | _W5500_SPI_WRITE_),&socket_int_mask[0],1);
558
559     // Configure sensor ethernet address
560     spi_transmit((SHAR | _W5500_SPI_WRITE_),&Sensor_Info.mac[0],6);
561     spi_transmit((GAR | _W5500_SPI_WRITE_),&Sensor_Info.gw[0],4);
562     spi_transmit((SUBR | _W5500_SPI_WRITE_),&Sensor_Info.sn[0],4);
563     spi_transmit((SIPR | _W5500_SPI_WRITE_),&Sensor_Info.ip[0],4);
564
565     // Init sockets
566     ethernet_init_socket(TRANSMIT_SOCKET,BUFFER_ECHO_SIZE_KB);
567
568     // Set RX buffer size
569     spi_transmit((Sn_RXBUF_SIZE(RECEIVE_SOCKET) | _W5500_SPI_WRITE_),&socket_buf_rx_size[0],1);
570
571     // Transmit NULL data
572     spi_transmit((Sn_CR(TRANSMIT_SOCKET) | _W5500_SPI_WRITE_),&socket_ready[0],2);
573     do
574     {
575         // Wait if data not transmit
576         spi_receive((Sn_IR(TRANSMIT_SOCKET)),&socket_ready[0],3);
577     } while (!(socket_ready[2] & Sn_IR_SENDOK));
```

```
578 }
579
580 // Init ethernet shield sockets
581 static void ethernet_init_socket(uint8_t socket,uint8_t buffer_size)
582 {
583     uint8_t socket_mode[1] = {Sn_MR_UDP};
584     uint8_t socket_buf_size[1];
585     uint8_t socket_int_en[1] = {Sn_IR_RECV};
586     uint8_t socket_open[1] = {Sn_CR_OPEN};
587     uint8_t socket_check[3] = {};
588     uint8_t port[2];
589     uint8_t ip[4];
590
591     socket_buf_size[0] = buffer_size;
592     port[0] = port_base[0];
593     port[1] = port_base[1] + socket;
594
595     for (uint8_t i = 0; i < 4; i++)
596     {
597         ip[i] = destination_ip[i];
598     }
599
600     // Configure socket registers
601     spi_transmit((Sn_MR(socket) | _W5500_SPI_WRITE_),&socket_mode[0],1);
602     spi_transmit((Sn_PORT(socket) | _W5500_SPI_WRITE_),&port[0],2);
603     spi_transmit((Sn_DPORT(socket) | _W5500_SPI_WRITE_),&port[0],2);
604     spi_transmit((Sn_DIPR(socket) | _W5500_SPI_WRITE_),&ip[0],4);
605     spi_transmit((Sn_TXBUF_SIZE(socket)| _W5500_SPI_WRITE_),&socket_buf_size[0],1);
606     spi_transmit((Sn_IMR(socket) | _W5500_SPI_WRITE_),&socket_int_en[0],1);
607     spi_transmit((Sn_CR(socket) | _W5500_SPI_WRITE_),&socket_open[0],1);
608     do
609     {
610         // Wait if socket is not in UDP mode
611         spi_receive(Sn_SR(socket),&socket_check[0],3);
612     } while (!(socket_check[2] & SOCK_UDP));
613 }
614
615 // Transmit echo data via ethernet
616 static void ethernet_transmit_echo(uint16_t *data,uint8_t send_manuell)
617 {
618     static uint8_t socket_send[1] = {Sn_CR_SEND};
619     static uint8_t socket_tx_wr_ptr[4] = {};
620     static uint16_t transmit_counter = 0x0000;
621     static uint8_t data_frame[MAX_DATA_FRAME_SIZE];
622     static uint32_t addr = 0;
623     static uint16_t tx_wr_ptr = 0x0000;
624
625     // If send manual, transmit echo data in TX register
626     if (send_manuell == SEND_MANUAL)
627     {
628         // Built TX write pointer frame
629         socket_tx_wr_ptr[0] = ((tx_wr_ptr + 0x0) & BYTE_1_MASK) >> BYTE_1_SHIFT;
630         socket_tx_wr_ptr[1] = ((tx_wr_ptr + 0x0) & BYTE_0_MASK) >> BYTE_0_SHIFT;
631
632         // Set TX write pointer
633         spi_transmit((Sn_TX_WR(TRANSMIT_SOCKET) | _W5500_SPI_WRITE_),&socket_tx_wr_ptr[0],2);
634
635         // Transmit echo data package
636         spi_transmit((Sn_CR(TRANSMIT_SOCKET) | _W5500_SPI_WRITE_),&socket_send[0],1);
637
638         // Set transmit counter and to null
639         transmit_counter = 0x00;
640     }
641     // Transmit echo data in automatic mode
642     else
643     {
644         // Ethernet shield address and control
645         addr = (WIZCHIP_TXBUF_BLOCK(TRANSMIT_SOCKET) << 3) | _W5500_SPI_WRITE_ | ((uint32_t)tx_wr_ptr << 8);
646
647         // Get tx_wr_ptr and transmit identifier and udp_counter
648         if (transmit_counter == 0)
649         {
650             // Get TX write pointer
651             spi_receive(Sn_TX_WR(TRANSMIT_SOCKET),&socket_tx_wr_ptr[0],4);
652
653             tx_wr_ptr = ((socket_tx_wr_ptr[2] << BYTE_1_SHIFT) | socket_tx_wr_ptr[3]);
654
655             // Ethernet shield address and control
656             addr = (WIZCHIP_TXBUF_BLOCK(TRANSMIT_SOCKET) << 3) | _W5500_SPI_WRITE_ | ((uint32_t)tx_wr_ptr << 8);
657
658             // Built data frame, identifier and UDP counter
659             data_frame[0] = (MEASUREMENT_MODE & BYTE_1_MASK) >> BYTE_1_SHIFT;
660             data_frame[1] = (MEASUREMENT_MODE & BYTE_0_MASK) >> BYTE_0_SHIFT;
661             data_frame[2] = (udp_counter & BYTE_1_MASK) >> BYTE_1_SHIFT;
662             data_frame[3] = (udp_counter & BYTE_0_MASK) >> BYTE_0_SHIFT;
```

```
663
664             // Transmit data frame to TX register, 4 byte
665             spi_transmit(addr,&data_frame[0],4);
666
667             // Build data frame, echo data
668             data_frame[0] = (*data & BYTE_1_MASK) >> BYTE_1_SHIFT;
669             data_frame[1] = (*data & BYTE_0_MASK) >> BYTE_0_SHIFT;
670
671             // Increment transmit counter, pointer and UDP counter
672             transmit_counter += 0x2;
673             tx_wr_ptr += 0x4;
674             udp_counter += 1;
675         }
676         // Transmit echo data 4 byte
677         else if (transmit_counter == 2)
678         {
679             // Built data frame
680             data_frame[2] = (*data & BYTE_1_MASK) >> BYTE_1_SHIFT;
681             data_frame[3] = (*data & BYTE_0_MASK) >> BYTE_0_SHIFT;
682
683             // Transmit echo data to TX register, 4 byte
684             spi_transmit(addr,&data_frame[0],4);
685
686             // Increment transmit counter and pointer
687             transmit_counter += 0x2;
688             tx_wr_ptr += 0x4;
689         }
690         // Transmit echo data 2 byte
691         else if (transmit_counter > 2)
692         {
693             // Built data frame
694             data_frame[0] = (*data & BYTE_1_MASK) >> BYTE_1_SHIFT;
695             data_frame[1] = (*data & BYTE_0_MASK) >> BYTE_0_SHIFT;
696
697             // Transmit echo data to TX register, 2 byte
698             spi_transmit(addr,&data_frame[0],2);
699
700             // Increment transmit counter and pointer
701             transmit_counter += 0x2;
702             tx_wr_ptr += 0x2;
703
704             // Set tx_wr_ptr and send package
705             if (transmit_counter >= MAX_ETHERNET_PAYLOAD)
706             {
707                 // Built TX write pointer frame
708                 socket_tx_wr_ptr[0] = (tx_wr_ptr & BYTE_1_MASK) >> BYTE_1_SHIFT;
709                 socket_tx_wr_ptr[1] = (tx_wr_ptr & BYTE_0_MASK) >> BYTE_0_SHIFT;
710
711                 // Set TX write pointer
712                 spi_transmit((Sn_TX_WR(TRANSMIT_SOCKET) | _W5500_SPI_WRITE_),&socket_tx_wr_ptr[0],2);
713
714                 // Transmit echo data package
715                 spi_transmit((Sn_CR(TRANSMIT_SOCKET) | _W5500_SPI_WRITE_),&socket_send[0],1);
716
717                 // Set pointer to null
718                 transmit_counter = 0x0;
719             }
720         }
721     }
722 }
723
724 // Transmit data via ethernet
725 static void ethernet_transmit_data(uint32_t *data,uint8_t data_len)
726 {
727     uint8_t socket_send[1] = {Sn_CR_SEND};
728     uint8_t socket_tx_wr_ptr[4] = {};
729     uint8_t data_frame[MAX_DATA_FRAME_SIZE];
730     uint32_t addr;
731     static uint16_t tx_wr_ptr = 0x0000;
732
733     // Get TX write pointer
734     spi_receive(Sn_TX_WR(TRANSMIT_SOCKET),&socket_tx_wr_ptr[0],4);
735
736     tx_wr_ptr = ((socket_tx_wr_ptr[2] << BYTE_1_SHIFT) | socket_tx_wr_ptr[3]);
737
738     // Transmit data to TX register
739     for (uint8_t i = 0; i < data_len; i++)
740     {
741         // Ethernet shield address and control
742         addr = (WIZCHIP_TXBUF_BLOCK(TRANSMIT_SOCKET) << 3) | _W5500_SPI_WRITE_ | ((uint32_t)tx_wr_ptr << 8);
743
744         // Built data frame
745         data_frame[0] = (data[i] & BYTE_3_MASK) >> BYTE_3_SHIFT;
746         data_frame[1] = (data[i] & BYTE_2_MASK) >> BYTE_2_SHIFT;
747         data_frame[2] = (data[i] & BYTE_1_MASK) >> BYTE_1_SHIFT;
```

```
748             data_frame[3] = (data[i] & BYTE_0_MASK) >> BYTE_0_SHIFT;
749
750             // Transmit data to TX register
751             spi_transmit(addr,&data_frame[0],4);
752
753             // Increment pointer
754             tx_wr_ptr += 0x4;
755         }
756
757         // Built TX write pointer frame
758         socket_tx_wr_ptr[0] = (tx_wr_ptr & BYTE_1_MASK) >> BYTE_1_SHIFT;
759         socket_tx_wr_ptr[1] = (tx_wr_ptr & BYTE_0_MASK) >> BYTE_0_SHIFT;
760
761         // Set TX write pointer and transmit data
762         spi_transmit((Sn_TX_WR(TRANSMIT_SOCKET) | _W5500_SPI_WRITE_),&socket_tx_wr_ptr[0],2);
763         spi_transmit((Sn_CR(TRANSMIT_SOCKET) | _W5500_SPI_WRITE_),&socket_send[0],1);
764     }
765
766     // Receive data via ethernet
767     static void ethernet_receive_data(uint32_t *data)
768     {
769         static uint8_t set_recv[1] = {Sn_CR_RECV};
770         static uint8_t received_data_size[4] = {};
771         static uint8_t rx_read_ptr[4] = {};
772         static uint16_t data_size;
773         static uint16_t read_ptr = 0x0000;
774         static uint8_t data_frame[MAX_DATA_FRAME_SIZE];
775
776         // Ethernet shield address and control
777         uint32_t addr = (WIZCHIP_RXBUF_BLOCK(RECEIVE_SOCKET) << 3) | ((uint32_t)read_ptr << BYTE_1_SHIFT);
778
779         // Get received data size
780         spi_receive(Sn_RX_RSR(RECEIVE_SOCKET),&received_data_size[0],4);
781
782         // Calculate data size and RX read pointer
783         data_size = (received_data_size[2] << BYTE_1_SHIFT) | received_data_size[3];
784         read_ptr += data_size;
785
786         // Built RX read pointer
787         rx_read_ptr[0] = (read_ptr & BYTE_1_MASK) >> BYTE_1_SHIFT;
788         rx_read_ptr[1] = (read_ptr & BYTE_0_MASK) >> BYTE_0_SHIFT;
789
790         // Receive data from RX register
791         spi_receive(addr,&data_frame[0],(data_size+2));
792
793         // Built data frame
794         data[0] = 0;
795         data[0] |= (uint32_t)(data_frame[data_size-2] << BYTE_3_SHIFT);
796         data[0] |= (uint32_t)(data_frame[data_size-1] << BYTE_2_SHIFT);
797         data[0] |= (uint32_t)(data_frame[data_size-0] << BYTE_1_SHIFT);
798         data[0] |= (uint32_t)(data_frame[data_size+1] << BYTE_0_SHIFT);
799
800         // Set RX read pointer
801         spi_transmit((Sn_RX_RD(RECEIVE_SOCKET) | _W5500_SPI_WRITE_),&rx_read_ptr[0],2);
802         spi_transmit((Sn_CR(RECEIVE_SOCKET) | _W5500_SPI_WRITE_),&set_recv[0],1);
803     }
804
805     // Init I2C
806     static void twi_init(void)
807     {
808         twi_options_t opt;
809
810         // Enable peripheral clock for TWI1
811         pmc_enable_periph_clk(ID_TWI1);
812
813         // Define I2C clock
814         opt.master_clk = sysclk_get_peripheral_hz();
815         opt.speed = TWI_CLK;
816
817         // Set I2C clock
818         twi_master_init(TWI1,&opt);
819     }
820
821     // Init BMP280 Sensor
822     static void bmp280_sensor_init(void)
823     {
824         twi_packet_t reset;
825         twi_packet_t status;
826
827         uint8_t bmp280_reset[1] = {BMP280_SOFT_RESET_CODE};
828         uint8_t bmp280_get_status[1];
829
830         // Define soft reset
831         reset.chip = BMP280_I2C_ADDRESS2;
832         reset.addr[0] = BMP280_RST_REG;
```

```
833         reset.addr_length = 1;
834         reset.buffer = &bmp280_reset[0];
835         reset.length = 1;
836 
837         // Define status register
838         status.chip = BMP280_I2C_ADDRESS2;
839         status.addr[0] = BMP280_STAT_REG;
840         status.addr_length = 1;
841         status.buffer = &bmp280_get_status[0];
842         status.length = 1;
843 
844         // Reset BMP280
845         twi_master_write(TWI1,&reset);
846 
847         // Wait if BMP280 is not reset
848         do
849         {
850             twi_master_read(TWI1,&status);
851         } while (bmp280_get_status[0] & BMP280_STATUS_REG_IM_UPDATE__MSK);
852     }
853 
854     // Get calibration data
855     static void bmp280_get_calibration(uint8_t *bmp280_calib)
856     {
857         twi_packet_t read_calib;
858 
859         // Define calibration register
860         read_calib.chip = BMP280_I2C_ADDRESS2;
861         read_calib.addr[0] = BMP280_TEMPERATURE_CALIB_DIG_T1_LSB_REG;
862         read_calib.addr_length = 1;
863         read_calib.buffer = &bmp280_calib[0];
864         read_calib.length = BMP280_CALIB_DATA_SIZE;
865 
866         // Get calibration data via I2C
867         twi_master_read(TWI1,&read_calib);
868     }
869 
870     // Start measuring and transmit data
871     static void bmp280_measurement(void)
872     {
873         uint8_t bmp280_calib[BMP280_CALIB_DATA_SIZE];
874         uint32_t transmit_data[8];
875 
876         // Get calibration data
877         bmp280_get_calibration(&bmp280_calib[0]);
878 
879         twi_packet_t ctrl_meas;
880         twi_packet_t measurement;
881         twi_packet_t status;
882 
883         // Define temperature oversampling
884         uint8_t temperature_oversampling = (BMP280_ULTRAHIGHRESOLUTION_OVERSAMP_TEMPERATURE
885                                             << BMP280_CTRL_MEAS_REG_OVERSAMP_TEMPERATURE__POS)
886                                             & BMP280_CTRL_MEAS_REG_OVERSAMP_TEMPERATURE__MSK;
887         // Define pressure oversampling
888         uint8_t pressure_oversampling = (BMP280_ULTRAHIGHRESOLUTION_OVERSAMP_PRESSURE
889                                         << BMP280_CTRL_MEAS_REG_OVERSAMP_PRESSURE__POS)
890                                         & BMP280_CTRL_MEAS_REG_OVERSAMP_PRESSURE__MSK;
891 
892         uint8_t bmp280_meas[6];
893         uint8_t bmp280_ctrl_meas[1] = {temperature_oversampling | pressure_oversampling | BMP280_FORCED_MODE};
894         uint8_t bmp280_get_status[1];
895 
896         // Define control measurement register
897         ctrl_meas.chip = BMP280_I2C_ADDRESS2;
898         ctrl_meas.addr[0] = BMP280_CTRL_MEAS_REG;
899         ctrl_meas.addr_length = 1;
900         ctrl_meas.buffer = &bmp280_ctrl_meas[0];
901         ctrl_meas.length = 1;
902 
903         // Define data start address
904         measurement.chip = BMP280_I2C_ADDRESS2;
905         measurement.addr[0] = BMP280_PRESSURE_MSB_REG;
906         measurement.addr_length = 1;
907         measurement.buffer = &bmp280_meas[0];
908         measurement.length = 6;
909 
910         // Define status register
911         status.chip = BMP280_I2C_ADDRESS2;
912         status.addr[0] = BMP280_STAT_REG;
913         status.addr_length = 1;
914         status.buffer = &bmp280_get_status[0];
915         status.length = 1;
916 
917         // Start measurement
```

```
918         twi_master_write(TWI1, &ctrl_meas);
919
920         // Wait if measurement not started
921         do
922         {
923             twi_master_read(TWI1,&status);
924         } while (!(bmp280_get_status[0] & BMP280_STATUS_REG_MEASURING__MSK));
925
926         // Wait if measurement not finished
927         do
928         {
929             twi_master_read(TWI1,&status);
930         } while (bmp280_get_status[0] & BMP280_STATUS_REG_MEASURING__MSK);
931
932         // Get temperature and pressure
933         twi_master_read(TWI1,&measurement);
934
935         // Pressure
936         transmit_data[0] = (TEMP_PRESS_MODE << BYTE_2_SHIFT) | (bmp280_meas[0] << 4) | ((bmp280_meas[1] & 0xF0) >> 4);
937         // Temperature
938         transmit_data[1] = ((bmp280_meas[1] & 0x0F) << 28) | ((bmp280_meas[2] & 0xF0) << 20) | (bmp280_meas[3] << 12)
939                          | (bmp280_meas[4] << 4) | ((bmp280_meas[5] & 0xF0) >> 4);
940         // Calibration data
941         transmit_data[2] = (bmp280_calib[0] << BYTE_3_SHIFT) | (bmp280_calib[1] << BYTE_2_SHIFT)
942                          | (bmp280_calib[2] << BYTE_1_SHIFT) | (bmp280_calib[3] << BYTE_0_SHIFT);
943         transmit_data[3] = (bmp280_calib[4] << BYTE_3_SHIFT) | (bmp280_calib[5] << BYTE_2_SHIFT)
944                          | (bmp280_calib[6] << BYTE_1_SHIFT) | (bmp280_calib[7] << BYTE_0_SHIFT);
945         transmit_data[4] = (bmp280_calib[8] << BYTE_3_SHIFT) | (bmp280_calib[9] << BYTE_2_SHIFT)
946                          | (bmp280_calib[10] << BYTE_1_SHIFT) | (bmp280_calib[11] << BYTE_0_SHIFT);
947         transmit_data[5] = (bmp280_calib[12] << BYTE_3_SHIFT) | (bmp280_calib[13] << BYTE_2_SHIFT)
948                          | (bmp280_calib[14] << BYTE_1_SHIFT) | (bmp280_calib[15] << BYTE_0_SHIFT);
949         transmit_data[6] = (bmp280_calib[16] << BYTE_3_SHIFT) | (bmp280_calib[17] << BYTE_2_SHIFT)
950                          | (bmp280_calib[18] << BYTE_1_SHIFT) | (bmp280_calib[19] << BYTE_0_SHIFT);
951         transmit_data[7] = (bmp280_calib[20] << BYTE_3_SHIFT) | (bmp280_calib[21] << BYTE_2_SHIFT)
952                          | (bmp280_calib[22] << BYTE_1_SHIFT) | (bmp280_calib[23] << BYTE_0_SHIFT);
953
954         // Transmit data via ethernet
955         ethernet_transmit_data(&transmit_data[0],8);
956     }
957
958     // Init ADC
959     static void adc_init_config(void)
960     {
961         pmc_enable_periph_clk(ID_ADC);
962         adc_init(ADC, sysclk_get_cpu_hz(),ADC_CLOCK,ADC_STARTUP_TIME_4);
963         adc_configure_timing(ADC,1,ADC_SETTLING_TIME_3,1);
964         adc_enable_tag(ADC);
965         adc_stop_sequencer(ADC);
966         // ADC channel 10 == Arduino Due A8
967         adc_enable_channel(ADC,ADC_CHANNEL_10);
968         adc_disable_anch(ADC);
969         adc_set_channel_input_gain(ADC,ADC_CHANNEL_10,ADC_GAINVALUE_0);
970         adc_disable_channel_input_offset(ADC,ADC_CHANNEL_10);
971         // Enable interrupt
972         NVIC_SetPriority(ADC_IRQn,INT_PRIOR_ADC);
973         NVIC_EnableIRQ(ADC_IRQn);
974     }
975
976     // Timer power up init and start
977     static void timer_power_up(void)
978     {
979         // Timer0 A1 for power up
980         ioport_set_pin_mode(PIO_PA2_IDX,IOPORT_MODE_MUX_A);
981         ioport_disable_pin(PIO_PA2_IDX);
982
983         pmc_enable_periph_clk(ID_TC1);
984
985         // Calculate power up time
986         uint32_t time_power_up = sysclk_get_cpu_hz() / DIV_MS / TIMER_MCK_2 * POWER_UP;
987
988         // Init timer for power up
989         tc_init(TC0, TC_CHANNEL1,0 | TC_CMR_WAVE | TC_CMR_CPCTRG | TC_CMR_ACPA_CLEAR | TC_CMR_ACPC_SET
990                 | TC_CMR_TCCLKS_TIMER_CLOCK1 | TC_CMR_CPCDIS);
991
992         // Set power up time
993         TC0->TC_CHANNEL[TC_CHANNEL1].TC_RA = time_power_up;
994         TC0->TC_CHANNEL[TC_CHANNEL1].TC_RC = time_power_up;
995
996         // Enable interrupt
997         NVIC_DisableIRQ(TC1_IRQn);
998         NVIC_ClearPendingIRQ(TC1_IRQn);
999         NVIC_SetPriority(TC1_IRQn,INT_PRIOR_TC);
1000        NVIC_EnableIRQ(TC1_IRQn);
1001
1002        // Start timer0 A1 channel1
```

```
1003        tc_enable_interrupt(TC0,TC_CHANNEL1,TC_IER_CPCS);
1004        tc_start(TC0,TC_CHANNEL1);
1005    }
1006
1007    // Init Timer
1008    static void timer_init(void)
1009    {
1010        // Timer0 A0 for ADC
1011        pmc_enable_periph_clk(ID_TC0);
1012
1013        // Calculate ADC sample time
1014        uint32_t time_adc = sysclk_get_cpu_hz() / SAMPLE_RATE_ADC_KHZ / KILO_HZ / TIMER_MCK_2;
1015
1016        // Init timer for ADC
1017        pio_configure_pin(PIO_PB25_IDX,(PIO_INPUT | PIO_DEFAULT));
1018        tc_init(TC0, TC_CHANNEL0,0 | TC_CMR_CPCTRG | TC_CMR_WAVE | TC_CMR_ACPA_CLEAR | TC_CMR_ACPC_SET | TC_CMR_TCCLKS_TIMER_CLOCK1);
1019
1020        // Set sample time
1021        TC0->TC_CHANNEL[TC_CHANNEL0].TC_RA = time_adc/2;
1022        TC0->TC_CHANNEL[TC_CHANNEL0].TC_RC = time_adc;
1023        adc_configure_trigger(ADC,ADC_TRIG_TIO_CH_0,0);
1024
1025        // Timer0 A2 for start measurement
1026        ioport_set_pin_mode(PIO_PA5_IDX,IOPORT_MODE_MUX_A);
1027        ioport_disable_pin(PIO_PA5_IDX);
1028
1029        pmc_enable_periph_clk(ID_TC2);
1030
1031        // Calculate start time
1032        uint32_t time_start = sysclk_get_cpu_hz() / DIV_MS / TIMER_MCK_2 * START_MEASUREMENT;
1033
1034        // Init timer for start measurement
1035        tc_init(TC0, TC_CHANNEL2,0 | TC_CMR_WAVE | TC_CMR_CPCTRG | TC_CMR_ACPA_CLEAR | TC_CMR_ACPC_SET
1036                | TC_CMR_TCCLKS_TIMER_CLOCK1 | TC_CMR_CPCDIS);
1037
1038        // Set start time
1039        TC0->TC_CHANNEL[TC_CHANNEL2].TC_RA = time_start;
1040        TC0->TC_CHANNEL[TC_CHANNEL2].TC_RC = time_start;
1041
1042        // Enable interrupt
1043        NVIC_DisableIRQ(TC2_IRQn);
1044        NVIC_ClearPendingIRQ(TC2_IRQn);
1045        NVIC_SetPriority(TC2_IRQn,INT_PRIOR_TC);
1046        NVIC_EnableIRQ(TC2_IRQn);
1047
1048        // Timer1 A3 for measurement time
1049        ioport_set_pin_mode(PIO_PB0_IDX, IOPORT_MODE_MUX_B);
1050        ioport_disable_pin(PIO_PB0_IDX);
1051
1052        pmc_enable_periph_clk(ID_TC3);
1053
1054        // Calculate measurement time
1055        uint32_t time_measurement = sysclk_get_cpu_hz() / DIV_MS / TIMER_MCK_2 * MEASUREMENT;
1056
1057        // Init timer for measurement time
1058        tc_init(TC1, TC_CHANNEL0,0 | TC_CMR_WAVE | TC_CMR_CPCTRG | TC_CMR_ACPA_CLEAR | TC_CMR_ACPC_SET
1059                | TC_CMR_TCCLKS_TIMER_CLOCK1 | TC_CMR_CPCDIS);
1060
1061        // Set measurement time
1062        TC1->TC_CHANNEL[TC_CHANNEL0].TC_RA = time_measurement;
1063        TC1->TC_CHANNEL[TC_CHANNEL0].TC_RC = time_measurement;
1064
1065        // Enable interrupt
1066        NVIC_DisableIRQ(TC3_IRQn);
1067        NVIC_ClearPendingIRQ(TC3_IRQn);
1068        NVIC_SetPriority(TC3_IRQn,INT_PRIOR_TC);
1069        NVIC_EnableIRQ(TC3_IRQn);
1070    }
1071
1072    // Init NVICC
1073    static void nvicc_init(void)
1074    {
1075        WDT->WDT_MR = WDT_MR_WDDIS;
1076
1077        pmc_enable_periph_clk(ID_PIOB);
1078        pmc_enable_periph_clk(ID_PIOD);
1079
1080        // Set ethernet handler
1081        pio_set_debounce_filter(PIOB,PIO_PB25,10);
1082        pio_handler_set(PIOB,ID_PIOB,PIO_PB25,(PIO_PULLUP | PIO_IT_FALL_EDGE),Int_Handler_Ethernet);
1083
1084        // Set sync handler
1085        pio_set_debounce_filter(PIOB,PIO_PB26,10);
1086        pio_handler_set(PIOB,ID_PIOB,PIO_PB26,(PIO_PULLUP | PIO_IT_FALL_EDGE),Int_Handler_Sync);
1087
```

```
1088        // Set next token handler
1089        pio_set_debounce_filter(PIOD,PIO_PD3,10);
1090        pio_handler_set(PIOD,ID_PIOD,PIO_PD3,(PIO_PULLUP | PIO_IT_RISE_EDGE),Int_Handler_Next_Token);
1091
1092        // Set reset handler
1093        pio_set_debounce_filter(PIOD,PIO_PD10,10);
1094        pio_handler_set(PIOD,ID_PIOD,PIO_PD10,(PIO_PULLUP | PIO_IT_FALL_EDGE),Int_Handler_Reset);
1095
1096        // Disable interrupts PIOB
1097        pio_disable_interrupt(PIOB,0xFFFFFFFF);
1098        pio_get_interrupt_status(PIOB);
1099
1100        // Disable interrupts PIOD
1101        pio_disable_interrupt(PIOD,0xFFFFFFFF);
1102        pio_get_interrupt_status(PIOD);
1103
1104        // Enable interrupts PIOB
1105        NVIC_DisableIRQ((IRQn_Type)ID_PIOB);
1106        NVIC_ClearPendingIRQ((IRQn_Type)ID_PIOB);
1107        NVIC_SetPriority((IRQn_Type)ID_PIOB,INT_PRIOR_ETHERNET);
1108        NVIC_EnableIRQ((IRQn_Type)ID_PIOB);
1109
1110        // Enable interrupts PIOD
1111        NVIC_DisableIRQ((IRQn_Type)ID_PIOD);
1112        NVIC_ClearPendingIRQ((IRQn_Type)ID_PIOD);
1113        NVIC_SetPriority((IRQn_Type)ID_PIOD,INT_PRIOR_ETHERNET);
1114        NVIC_EnableIRQ((IRQn_Type)ID_PIOD);
1115
1116        // Enable interrupt ethernet shield
1117        pio_enable_interrupt(PIOB,PIO_PB25);
1118    }
1119
1120    // Init sync pins
1121    static void sync_pins_init(void)
1122    {
1123        // Set PA14 as output for sync measurement
1124        pmc_enable_periph_clk(ID_PIOA);
1125        //pio_set_debounce_filter(PIOA,PIO_PA14,10);
1126        ioport_enable_pin(PIO_PA14_IDX);
1127        ioport_set_pin_dir(PIO_PA14_IDX,IOPORT_DIR_OUTPUT);
1128        ioport_set_pin_level(PIO_PA14_IDX,IOPORT_PIN_LEVEL_LOW);
1129
1130        // Set PD6 as output for next token
1131        pmc_enable_periph_clk(ID_PIOD);
1132        //pio_set_debounce_filter(PIOD,PIO_PD6,10);
1133        ioport_enable_pin(PIO_PD6_IDX);
1134        ioport_set_pin_dir(PIO_PD6_IDX,IOPORT_DIR_OUTPUT);
1135        ioport_set_pin_level(PIO_PD6_IDX,IOPORT_PIN_LEVEL_LOW);
1136
1137        // Set PC1 as output for reset
1138        pmc_enable_periph_clk(ID_PIOC);
1139        //pio_set_debounce_filter(PIOC,PIO_PC1,10);
1140        ioport_enable_pin(PIO_PC1_IDX);
1141        ioport_set_pin_dir(PIO_PC1_IDX,IOPORT_DIR_OUTPUT);
1142        ioport_set_pin_level(PIO_PC1_IDX,IOPORT_PIN_LEVEL_LOW);
1143    }
1144
1145    // Init pins for US-Sensor
1146    static void init_sensor_pins(void)
1147    {
1148        // Configure PC16  for US-Sensor pin 4
1149        pmc_enable_periph_clk(ID_PIOC);
1150        pio_set_debounce_filter(PIOC,PIO_PC16,10);
1151        ioport_enable_pin(PIO_PC16_IDX);
1152        ioport_set_pin_dir(PIO_PC16_IDX,IOPORT_DIR_OUTPUT);
1153        ioport_set_pin_level(PIO_PC16_IDX,IOPORT_PIN_LEVEL_LOW);
1154    }
1155
1156    // Init pins for sensor number
1157    static void init_sensor_number_pins(void)
1158    {
1159        // Configure PC12 ... PC15 for sensor number
1160        pmc_enable_periph_clk(ID_PIOC);
1161
1162        // PC12
1163        pio_set_debounce_filter(PIOC,PIO_PC12,10);
1164        ioport_enable_pin(PIO_PC12_IDX);
1165        ioport_set_pin_dir(PIO_PC12_IDX,IOPORT_DIR_INPUT);
1166
1167        // PC13
1168        pio_set_debounce_filter(PIOC,PIO_PC13,10);
1169        ioport_enable_pin(PIO_PC13_IDX);
1170        ioport_set_pin_dir(PIO_PC13_IDX,IOPORT_DIR_INPUT);
1171
1172        // PC14
```

```
1173        pio_set_debounce_filter(PIOC,PIO_PC14,10);
1174        ioport_enable_pin(PIO_PC14_IDX);
1175        ioport_set_pin_dir(PIO_PC14_IDX,IOPORT_DIR_INPUT);
1176
1177        // PC15
1178        pio_set_debounce_filter(PIOC,PIO_PC15,10);
1179        ioport_enable_pin(PIO_PC15_IDX);
1180        ioport_set_pin_dir(PIO_PC15_IDX,IOPORT_DIR_INPUT);
1181    }
1182
1183    // Get sensor number form io-pins
1184    static void get_sensor_number(uint8_t *number)
1185    {
1186        *number = 0 | !ioport_get_pin_level(PIO_PC15_IDX);
1187        *number |= (!ioport_get_pin_level(PIO_PC14_IDX) << 1);
1188        *number |= (!ioport_get_pin_level(PIO_PC13_IDX) << 2);
1189        *number |= (!ioport_get_pin_level(PIO_PC12_IDX) << 3);
1190    }
1191
1192    #ifdef DEBUGGING
1193    // Init debug mode
1194    static void debugging_init(void)
1195    {
1196        // Configure PC23
1197        ioport_enable_pin(PIO_PC23_IDX);
1198        ioport_set_pin_dir(PIO_PC23_IDX,IOPORT_DIR_OUTPUT);
1199        ioport_set_pin_level(PIO_PC23_IDX,IOPORT_PIN_LEVEL_LOW);
1200
1201        // Configure PC24
1202        ioport_enable_pin(PIO_PC24_IDX);
1203        ioport_set_pin_dir(PIO_PC24_IDX,IOPORT_DIR_OUTPUT);
1204        ioport_set_pin_level(PIO_PC24_IDX,IOPORT_PIN_LEVEL_LOW);
1205
1206        // Configure PC25
1207        ioport_enable_pin(PIO_PC25_IDX);
1208        ioport_set_pin_dir(PIO_PC25_IDX,IOPORT_DIR_OUTPUT);
1209        ioport_set_pin_level(PIO_PC25_IDX,IOPORT_PIN_LEVEL_LOW);
1210    }
1211
1212    // Configure UART for debug message output
1213    static void configure_console(void)
1214    {
1215        const usart_serial_options_t uart_serial_options = {
1216            .baudrate = CONF_UART_BAUDRATE,
1217            .paritytype = CONF_UART_PARITY
1218        };
1219
1220        /* Configure console UART. */
1221        sysclk_enable_peripheral_clock(CONSOLE_UART_ID);
1222        stdio_serial_init(CONF_UART, &uart_serial_options);
1223    }
1224    #endif
```

D. Quellcode des ultraschallbasierten Positionierungssystems

D.1. Systemschnittstelle

ultrasoundPositioningSystem.py

```
#!/usr/bin/env python3
#*********************************************************************#
#* University: HAW Hamburg                                           *#
#* Author: Christopher Rotzlawski                                    *#
#*                                                                   *#
#* Project: Surrounding Sensor-based Naviagtionssystem               *#
#* Version: 1.0                                                      *#
#*********************************************************************#
import rospy
import rospkg
from commonMsgUVPS.msg import checkUltrasoundSystem
from commonMsgUVPS.msg import responseCheckUltrasoundSystem
from commonMsgUVPS.msg import startUltrasoundMeasurement
from commonMsgUVPS.msg import ultrasoundSystemPosition
from commonMsgUVPS.msg import ultrasoundPositionRequest
from commonMsgUVPS.msg import responseUltrasoundPosition
from commonMsgUVPS.msg import validUltrasoundPosition
import time

class ultrasoundPositioningSystem:
    # Initialize class
    def __init__(self):
        # Definitions
        self.PUBLISHCHECK = 'checkUltrasoundSystem'
        self.PUBLISHSTART = 'startUltrasoundMeasurement'
        self.PUBLISHPOSITION = 'responseUltrasoundPostion'
        self.PUBLISHVALID = 'validUltrasoundPostion'
        self.LISTNERCHECK = 'responseUltrasoundSytem'
        self.LISTNERPOSITION = 'ultrasoundSystemPosition'
        self.LISTNERREQUEST = 'ultrasoundPositionRequest'
        self.ROSNODE = 'ultrasoundPositioningSystem'

        self.systems = ['ultrasoundInterface','ultrasoundFeatures','ultrasoundSimulation','ultrasoundMatcher']
        self.systemReadyCounter = 0
        self.numberSystems = len(self.systems)
        self.systemReady = False

        self.errorCounter = 0
        error = False

        # Start node and publisher
        rospy.init_node(self.ROSNODE, disable_signals=True)

        self.pubCheck = rospy.Publisher(self.PUBLISHCHECK, checkUltrasoundSystem, queue_size=10)
        self.pubStart = rospy.Publisher(self.PUBLISHSTART, startUltrasoundMeasurement, queue_size=10)
        self.pubPosition = rospy.Publisher(self.PUBLISHPOSITION, responseUltrasoundPosition, queue_size=10)
        self.pubValidPosition = rospy.Publisher(self.PUBLISHVALID, validUltrasoundPosition, queue_size=10)

        self.messageCheck = checkUltrasoundSystem()
        self.messageStart = startUltrasoundMeasurement()
        self.messagePosition = responseUltrasoundPosition()
        self.messageValidPosition = validUltrasoundPosition()

        # Check number of system components
        while self.pubCheck.get_num_connections() < 3:#len(self.systems)
            self.errorCounter += 1

```

```
58              if self.errorCounter >= 50:
59                  error = True
60                  print("ERROR: not enough system components available in ultrasoundPositioningSystem!")
61                  print("Number of expected system componets: "+str(len(self.systems)))
62                  print("Number of connected system components: "+str(self.pubCheck.get_num_connections()))
63                  break
64
65              rospy.sleep(0.5)
66
67          if not error:
68              # Wait for startup of system components
69              print('Waiting for start of system components...')
70              rospy.sleep(5)
71
72              # Start listner callbacks
73              self.listner()
74          else:
75              print('ERROR: start ultrasound position system failed!')
76
77      # Start listner callbacks
78      def listner(self):
79          rospy.Subscriber(self.LISTNERCHECK, responseCheckUltrasoundSystem, self.listnerCheckSystem)
80          rospy.Subscriber(self.LISTNERPOSITION, ultrasoundSystemPosition, self.listnerPosition)
81
82          time.sleep(0.1)
83
84          self.checkSystem()
85
86          rospy.spin()
87
88      # Check system components
89      def checkSystem(self):
90          self.messageCheck.system = self.systems[self.systemReadyCounter]
91          self.pubCheck.publish(self.messageCheck)
92
93      # Listner for check system response
94      def listnerCheckSystem(self,msg):
95          if msg.ready and msg.system == self.systems[self.systemReadyCounter]:
96              self.systemReadyCounter += 1
97
98              if self.systemReadyCounter < len(self.systems):
99                  self.checkSystem()
100             else:
101                 self.systemReady = True
102                 print("ultrasoundPositioningSystem ready.")
103                 rospy.Subscriber(self.LISTNERREQUEST, ultrasoundPositionRequest, self.listnerPositionRequest)
104
105         else:
106             print("ERROR: component "+msg.system+" not ready!")
107             rospy.signal_shutdown("ERROR: ultrasoundPositioningSystem not ready!")
108
109     # Listner for ultrasound position data
110     def listnerPosition(self,data):
111         self.messagePosition.longitude = data.longitude
112         self.messagePosition.latitude = data.latitude
113         self.messagePosition.direction = data.direction
114         self.messagePosition.accuracy = data.accuracy
115
116         self.pubPosition.publish(self.messagePosition)
117
118     # Listner for position request
119     def listnerPositionRequest(self,data):
120         if data.getPosition:
121             self.startUltrasoundPositioningSystem()
122
123         elif data.validPosition:
124             self.messageValidPosition.latitudeValid = data.latitudeOld
125             self.messageValidPosition.longitudeValid = data.longitudeOld
126             self.messageValidPosition.directionValid = data.directionOld
127
128             self.pubValidPosition.publish(self.messageValidPosition)
129
130     # Start ultrasound measurement
131     def startUltrasoundPositioningSystem(self):
132         if self.systemReady:
133             print("ultrasoundPositioningSystem started.")
134
135             self.messageStart.start = True
136             self.pubStart.publish(self.messageStart)
137         else:
138             print("ERROR: ultrasoundPositioningSystem not ready.")
139             rospy.signal_shutdown("ERROR: ultrasoundPositioningSystem not ready!")
140
141 if __name__ == '__main__':
142     try:
```

```
143         node = ultrasoundPositioningSystem()
144     except rospy.ROSInterruptException:
145         print('ERROR: start ultrasound positioning system failed!')
```

D.2. Ultraschallsensorschnittstelle

ultrasoundInterface.cpp

```
1  /*****************************************************************/
2  /* University: HAW Hamburg                                       */
3  /* Author: Christopher Rotzlawski                                */
4  /*                                                               */
5  /* Project: Surrounding Sensor-based Naviagtionssystem           */
6  /* Version: 1.0                                                  */
7  /*****************************************************************/
8  #include "../include/ultrasoundInterface/ultrasoundInterfaceNode.hpp"
9
10 #define PACKAGENAME "ultrasoundPositioningSystem"
11 #define NODENAME "ultrasoundInterface"
12 #define ECHOTOPICNAME "rawUltrasoundData"
13 #define RESPONSENAME "responseUltrasoundSytem"
14 #define LISTNERCHECKNAME "checkUltrasoundSystem"
15 #define LISTNERSTARTNAME "startUltrasoundMeasurement"
16 #define FILEPATH (ros::package::getPath(PACKAGENAME)+"/config/positioningSystemConfig.json").c_str()
17
18 // Start node
19 int main(int argc,char **argv)
20 {
21     uml::ultrasoundInterfaceNode* node = new uml::ultrasoundInterfaceNode(FILEPATH);
22
23     node->startNode(argc,argv,NODENAME,ECHOTOPICNAME,RESPONSENAME,LISTNERCHECKNAME,LISTNERSTARTNAME);
24
25     std::cout << "ERROR: System not ready. Node ultrasoundInterface stopped." << std::endl;
26
27     return 0;
28 }
```

ultrasoundInterfaceNode.hpp

```
1  /*****************************************************************/
2  /* University: HAW Hamburg                                       */
3  /* Author: Christopher Rotzlawski                                */
4  /*                                                               */
5  /* Project: Surrounding Sensor-based Naviagtionssystem           */
6  /* Version: 1.0                                                  */
7  /*****************************************************************/
8  #ifndef ULTRASOUNDINTERFACENODE_HPP
9  #define ULTRASOUNDINTERFACENODE_HPP
10
11 // Includes
12 #include <ros/ros.h>
13 #include <ros/package.h>
14 #include "commonMsgUVPS/rawUltrasoundData.h"
15 #include "commonMsgUVPS/startUltrasoundMeasurement.h"
16 #include "commonMsgUVPS/checkUltrasoundSystem.h"
17 #include "commonMsgUVPS/responseCheckUltrasoundSystem.h"
18 #include "../include/ultrasoundInterface/USSensorAPI.hpp"
19 #include <vector>
20 #include <string>
21
22 namespace uml
23 {
24     class ultrasoundInterfaceNode
25     {
26         // Attributes
27     private:
28         bool error;
29         bool systemReady;
30         bool sendInstruction;
31         int masterSensor;
32         int count;
33         int numberSensors;
```

```
34          int dataLen;
35          double temperature;
36          double pressure;
37          std::vector<uint16_t> measurementResults;
38          uml::USSensorAPI* sensorSystem;
39          ros::Subscriber subCheckSystem;
40          ros::Subscriber subStartMeasurement;
41          ros::Publisher pubCheckSystem;
42          ros::Publisher pubEchoData;
43          commonMsgUVPS::rawUltrasoundData echoDataMsg;
44          commonMsgUVPS::responseCheckUltrasoundSystem responseSystemMsg;
45
46          // Methods
47      public:
48          ultrasoundInterfaceNode(std::string filepath);
49          ~ultrasoundInterfaceNode(void);
50          void startNode(int argc,char** argv,std::string nodeName,std::string echoName,std::string responseName
51                         ,std::string listnerCheckName,std::string listnerStartName);
52      private:
53          void checkCallback(const commonMsgUVPS::checkUltrasoundSystemPtr& msg);
54          void startCallback(const commonMsgUVPS::startUltrasoundMeasurementPtr& msg);
55      };
56  }
57
58  #endif // ULTRASOUNDINTERFACENODE_HPP
```

ultrasoundInterfaceNode.cpp

```
1   /*****************************************************************/
2   /* University: HAW Hamburg                                       */
3   /* Author: Christopher Rotzlawski                                */
4   /*                                                               */
5   /* Project: Surrounding Sensor-based Naviagtionssystem           */
6   /* Version: 1.0                                                  */
7   /*****************************************************************/
8   #include "ultrasoundInterfaceNode.hpp"
9
10  namespace uml
11  {
12      // Initialize node
13      ultrasoundInterfaceNode::ultrasoundInterfaceNode(std::string filepath)
14      {
15          error = false;
16          systemReady = false;
17          sendInstruction = false;
18          masterSensor = 0;
19          count = 0;
20          numberSensors = 0;
21          dataLen = 0;
22
23          sensorSystem = new uml::USSensorAPI(filepath);
24
25          systemReady = sensorSystem->getSystemStatus();
26          if(systemReady) sensorSystem->getTemperaturePressure(&temperature,&pressure);
27
28          // Check system
29          if(systemReady)
30          {
31              if(sensorSystem->getNumberSensors(&numberSensors))
32              {
33                  systemReady = false;
34                  std::cout << "ERROR: Can't get number of sensors!" << std::endl;
35              }
36              else measurementResults.reserve(10000 * numberSensors);
37          }
38      }
39
40      ultrasoundInterfaceNode::~ultrasoundInterfaceNode()
41      {
42          // NOP
43      }
44
45      // Start node
46      void ultrasoundInterfaceNode::startNode(int argc,char** argv,std::string nodeName,std::string echoName,std::string responseName
47                                              ,std::string listnerCheckName,std::string listnerStartName)
48      {
49          ros::init(argc,argv,nodeName);
50          ros::NodeHandle n;
51
52          subCheckSystem = n.subscribe(listnerCheckName,1000,&ultrasoundInterfaceNode::checkCallback,this);
53          subStartMeasurement = n.subscribe(listnerStartName,1000,&ultrasoundInterfaceNode::startCallback,this);
```

```
54          pubCheckSystem = n.advertise<commonMsgUVPS::responseCheckUltrasoundSystem>(responseName,1000);
55          pubEchoData = n.advertise<commonMsgUVPS::rawUltrasoundData>(echoName,1000);
56
57          ros::spin();
58      }
59
60      // Calback for check system request
61      void ultrasoundInterfaceNode::checkCallback(const commonMsgUVPS::checkUltrasoundSystemPtr& msg)
62      {
63          if(msg->system == "ultrasoundInterface")
64          {
65              responseSystemMsg.system = msg->system;
66              responseSystemMsg.ready = systemReady;
67
68              pubCheckSystem.publish(responseSystemMsg);
69          }
70      }
71
72      // Callback for start ultrasound measurement
73      void ultrasoundInterfaceNode::startCallback(const commonMsgUVPS::startUltrasoundMeasurementPtr& msg)
74      {
75          masterSensor = 0;
76          count = 0;
77          sendInstruction = true;
78
79          while(count < numberSensors && systemReady)
80          {
81              echoDataMsg.header.stamp = ros::Time::now();
82
83              sensorSystem->measurement(&measurementResults,&dataLen,&masterSensor,&sendInstruction);
84              systemReady = sensorSystem->getSystemStatus();
85
86              echoDataMsg.pressure = pressure;
87              echoDataMsg.temperature = temperature;
88              echoDataMsg.numberSensors = numberSensors;
89              echoDataMsg.dataLen = dataLen;
90              echoDataMsg.echoData = measurementResults;
91
92              pubEchoData.publish(echoDataMsg);
93
94              measurementResults.clear();
95
96              masterSensor++;
97              count++;
98
99              //sendInstruction = false;
100         }
101     }
102 }
```

USSensorAPI.hpp

```
1   /*****************************************************************************/
2   /* University: HAW Hamburg                                                   */
3   /* Author: Christopher Rotzlawski                                            */
4   /*                                                                           */
5   /* Project: Surrounding Sensor-based Naviagtionssystem                       */
6   /* Version: 1.0                                                              */
7   /*****************************************************************************/
8   #ifndef USSENSORAPI_HPP
9   #define USSENSORAPI_HPP
10
11  // Includes
12  #include "SensorAPI.hpp"
13  #include <string>
14  #include <iostream>
15  #include <vector>
16
17  namespace uml
18  {
19      class USSensorAPI : private SensorAPI
20      {
21          // Attributes
22      private:
23          std::vector<std::vector<uint16_t>> echoData;
24          std::vector<uint16_t> checkUDPCounter;
25
26          // Methods
27      public:
28          USSensorAPI(std::string filePath);
29          ~USSensorAPI();
```

```
30          bool measurement(std::vector<uint16_t>* results,int* vecLen,int* startSensor,bool* sendInstruction);
31          bool getSystemStatus(void);
32          bool getTemperaturePressure(double* temperature,double* pressure);
33          bool calculateTemperature(uint8_t* rawData,double* temperature);
34          bool calculatePressure(uint8_t* rawData,double* pressure,double *temperature);
35          bool getNumberSensors(int* numSensors);
36      };
37  }
38
39  #endif // USSENSORAPI_HPP
```

USSensorAPI.cpp

```
1   /*****************************************************************************/
2   /* University: HAW Hamburg                                                   */
3   /* Author: Christopher Rotzlawski                                            */
4   /*                                                                           */
5   /* Project: Surrounding Sensor-based Naviagtionssystem                       */
6   /* Version: 1.0                                                              */
7   /*****************************************************************************/
8   #include "USSensorAPI.hpp"
9
10  #define SENSORSYSTEM "US_Sensor"
11
12  namespace uml
13  {
14      // Initialize API
15      USSensorAPI::USSensorAPI(std::string filePath)
16      {
17          bool error = false;
18
19          systemReady = false;
20          configFilePath = filePath;
21
22          // Initialize socket
23          error = initSocket();
24
25          // Check system
26          if(!error && sensorsystem.compare(SENSORSYSTEM))
27          {
28              error = true;
29              std::cout << "Wrong config file." << std::endl;
30          }
31
32          // Check sensors
33          if(!error)
34          {
35              error = checkSensors();
36              if(error) error = checkNumSensors();
37              else systemReady = true;
38          }
39
40          if(!error)
41          {
42              // Initialize echoData with identifier
43              for(int i = 0; i < numberSensors; i++)
44              {
45                  std::vector<uint16_t> initVec;
46                  echoData.push_back(initVec);
47                  echoData[i].push_back(i);
48                  echoData[i].reserve(10000 + 1);
49              }
50
51
52              // Clear checkUDPCounter
53              for(int i = 0; i < numberSensors; i++) checkUDPCounter.push_back(0);
54          }
55
56          // If error print number of found sensors
57          if(error)
58          {
59              std::cout << "ERROR: Systemstart failed! " << std::endl;
60          }
61      }
62
63      // Destructor to close sockets
64      USSensorAPI::~USSensorAPI()
65      {
66          bool error = false;
67
68          error = closeSocket();
```

```
69      }
70
71      // Ultrasound measurement function
72      bool USSensorAPI::measurement(std::vector<uint16_t>* results,int* vecLen,int* startSensor,bool* sendInstruction)
73      {
74          bool error = false;
75          bool recLoop = true;
76          std::string srcIP;
77          int sensorNum;
78          char chrSrcAddr[INET_ADDRSTRLEN];
79          uint8_t checkSumByteOne;
80          uint8_t checkSumByteTwo;
81          int16_t checkMsgLen;
82          uint8_t buffer[1470];
83          uint16_t lenBuffer = sizeof(buffer);
84          int countEnd = 0;
85          struct sockaddr_in src_addr;
86          socklen_t lenSrcAddr = sizeof(src_addr);
87
88          // Check if nullptr
89          if((results == nullptr)) error = true;
90          if((vecLen == nullptr)) error = true;
91          if((startSensor == nullptr)) error = true;
92          if((sendInstruction == nullptr)) error = true;
93
94          // If sendInstruction
95          if(*sendInstruction && !error)
96          {
97              // Clear checkUDPCounter
98              for(int i = 0; i < numberSensors; i++) checkUDPCounter[i] = 0;
99
100             // Build message
101             checkSumByteOne = checkSum & 0xFF;
102             checkSumByteTwo = (checkSum & 0xFF00) >> 8;
103             uint8_t data[4] = {measurementMsg[0],measurementMsg[1],checkSumByteTwo,checkSumByteOne};
104             uint8_t lenData = sizeof(data);
105
106             // Increment checkSum
107             checkSum++;
108
109             // Send message
110             sendMessage(&dstIP[*startSensor],&data[0],&lenData);
111
112             // Check response
113             checkMsgLen = recvfrom(sock,&buffer[0],lenBuffer,0,(struct sockaddr*)&src_addr,&lenSrcAddr);
114
115             // Get response IP
116             inet_ntop(AF_INET,&src_addr.sin_addr,chrSrcAddr,INET_ADDRSTRLEN);
117             srcIP = chrSrcAddr;
118
119             // Check response and IP
120             for(int i = 0; i < lenData; i++)
121             {
122                 if(!(buffer[i] == data[i]))
123                 {
124                     error = true;
125                     std::cout << "error" << std::endl;
126                 }
127             }
128
129             if(srcIP.compare(dstIP[*startSensor])) error = true;
130             if(checkMsgLen != 4) error = true;
131
132             if(error) std::cout << "Failure in send instructions." << std::endl;
133         }
134
135         // Get messages and sort
136         while(!error && recLoop)
137         {
138             // Get data
139             checkMsgLen = recvfrom(sock,&buffer[0],lenBuffer,0,(struct sockaddr*)&src_addr,&lenSrcAddr);
140
141             // Get response IP
142             inet_ntop(AF_INET,&src_addr.sin_addr,chrSrcAddr,INET_ADDRSTRLEN);
143             srcIP = chrSrcAddr;
144
145             // Check if timeout
146             if(checkMsgLen <= 0)
147             {
148                 std::cout << "Missing identifier in echo data response." << std::endl;
149                 error = true;
150                 break;
151             }
152
153             // Get sensor number
```

```
154             sensorNum = -1;
155             for(int i = 0; i < numberSensors; i++)
156             {
157                 if(srcIP == dstIP[i])
158                 {
159                     sensorNum = i;
160                 }
161             }
162 
163             // Check if IP in range
164             if(!(sensorNum >= 0 && sensorNum < numberSensors)) continue;
165             // Check if end measurement
166             else if(buffer[0] == endMeasurement[0] && buffer[1] == endMeasurement[1])
167             {
168                 countEnd++;
169                 if(countEnd == numberSensors) recLoop = false;
170             }
171             // Else push data in vector
172             else
173             {
174                 // Check if data fits into buffer
175                 if(echoData[sensorNum].size() + (checkMsgLen - 4) / 2 > 10000 + 1)
176                 {
177                     std::cout << "Failure in push data into buffer. Too much data from sensor " << sensorName[sensorNum]
178                             << "." << std::endl;
179                     error = true;
180                 }
181                 // Check UDP counter
182                 else if(!(buffer[2] == (checkUDPCounter[sensorNum] >> 8)) || !(buffer[3] == (checkUDPCounter[sensorNum] & 0xFF)))
183                 {
184                     printf("%d  %d\n",buffer[3],(checkUDPCounter[sensorNum] & 0xFF));
185                     std::cout << "Missing UDP package from sensor " << sensorName[sensorNum] << "." << std::endl;
186                     error = true;
187                 }
188                 // Push data in vector
189                 else
190                 {
191                     for(int i = 4; i < checkMsgLen; i += 2)
192                     {
193                         echoData[sensorNum].push_back(buffer[i] << 8 |buffer[i+1]);
194                     }
195 
196                     // Increment UDP counter
197                     checkUDPCounter[sensorNum]++;
198                 }
199 
200                 // Check if error
201                 if(error) break;
202             }
203         }
204 
205         // Check sensor data length
206         if(!error)
207         {
208             for(int i = 0; i < numberSensors; i++)
209             {
210                 if(echoData[i].size() < 10000 - 5)
211                 {
212                     std::cout << "Difference between sensor data length to big." << std::endl;
213                     std::cout << "Threshold: " << 10000 - 5 << std::endl;
214                     std::cout << "Sensor " << sensorName[i] << " lenght of data: " << echoData[i].size() << std::endl;
215                     error = true;
216                 }
217                 else while(echoData[i].size() < 10000+1) echoData[i].push_back(0);
218             }
219         }
220 
221         // Check if error
222         if(error) systemReady = false;
223         // Else push echo data in results
224         else for(int i = 0; i < numberSensors; i++) results->insert(results->end(),echoData[i].begin(),echoData[i].end());
225 
226         *vecLen = results->size();
227 
228         // Clear echo data vector
229         for(int i = 0; i < numberSensors; i++)
230         {
231             echoData[i].clear();
232             echoData[i].push_back(i);
233             echoData[i].reserve(10000 + 1);
234         }
235 
236         return error;
237     }
238 
```

```
239     // Environmental measurement function
240     bool USSensorAPI::getTemperaturePressure(double* temperature,double* pressure)
241     {
242         bool error = false;
243         bool errorLoop;
244         struct sockaddr_in src_addr;
245         socklen_t lenSrcAddr = sizeof(src_addr);
246         char chrSrcAddr[INET_ADDRSTRLEN];
247         std::string strSrcAddr;
248         uint8_t checkSumByteOne;
249         uint8_t checkSumByteTwo;
250         uint8_t buffer[32];
251         uint8_t lenBuffer = sizeof(buffer);
252         int16_t checkMsgLen;
253         double temperatureTemp[numberSensors];
254         double pressureTemp[numberSensors];
255
256         if((temperature == nullptr)) error = true;
257         if((pressure == nullptr)) error = true;
258
259         for(int i = 0; i < numberSensors; i++)
260         {
261             errorLoop = false;
262
263             // Build message
264             checkSumByteOne = checkSum & 0xFF;
265             checkSumByteTwo = (checkSum & 0xFF00) >> 8;
266             uint8_t data[4] = {tempMsg[0],tempMsg[1],checkSumByteTwo,checkSumByteOne};
267             uint8_t lenData = sizeof(data);
268
269             // Increment checkSum
270             checkSum++;
271
272             // Send message
273             sendMessage(&dstIP[i],&data[0],&lenData);
274
275             // Get response
276             checkMsgLen = recvfrom(sock,&buffer[0],lenBuffer,0,(struct sockaddr*)&src_addr,&lenSrcAddr);
277
278             // Get response IP
279             inet_ntop(AF_INET,&src_addr.sin_addr,chrSrcAddr,INET_ADDRSTRLEN);
280             strSrcAddr = chrSrcAddr;
281
282             // Check response and IP
283             for(int i = 0; i < lenData; i++)
284                 if(!(buffer[i] == data[i])) errorLoop = true;
285             if(strSrcAddr.compare(dstIP[i])) errorLoop = true;
286             if(checkMsgLen != 4) errorLoop = true;
287
288             // Clear buffer
289             for(int8_t i = 0; i < lenBuffer; i++) buffer[i] = 0;
290
291             // Get data
292             checkMsgLen = recvfrom(sock,&buffer[0],lenBuffer,0,(struct sockaddr*)&src_addr,&lenSrcAddr);
293
294             // Get response IP
295             inet_ntop(AF_INET,&src_addr.sin_addr,chrSrcAddr,INET_ADDRSTRLEN);
296             strSrcAddr = chrSrcAddr;
297
298             // Check response and IP
299             for(int i = 0; i < lenData/2; i++)
300                 if(!(buffer[i] == data[i])) errorLoop = true;
301             if(strSrcAddr.compare(dstIP[i])) errorLoop = true;
302             if(checkMsgLen != 32) errorLoop = true;
303
304             // If error print message
305             if(errorLoop) std::cout << "No temperature response form sensor " << i+1 << std::endl;
306
307             if(!errorLoop)
308             {
309                 // Calculate sensor temperature
310                 if(calculateTemperature(&buffer[0],&temperatureTemp[i]))
311                 {
312                     errorLoop = true;
313                     std::cout << "Calculating temperature sensor " << i+1 << " failed." << std::endl;
314                 }
315
316                 // Calculate sensor pressure
317                 if(calculatePressure(&buffer[0],&pressureTemp[i],&temperatureTemp[i]))
318                 {
319                     errorLoop = true;
320                     std::cout << "Calculating pressure sensor " << i+1 << " failed." << std::endl;
321                 }
322             }
323
```

```
324             // Clear buffer
325             for(int8_t i = 0; i < lenBuffer; i++) buffer[i] = 0;
326 
327             // Check if error
328             if(errorLoop) error = true;
329         }
330 
331         // Calculate temperature and pressure
332         if(!error)
333         {
334             *temperature = 0;
335             *pressure = 0;
336             for(int8_t i = 0; i < numberSensors; i++)
337             {
338                 *temperature += temperatureTemp[i];
339                 *pressure += pressureTemp[i];
340             }
341             *temperature /= numberSensors;
342             *pressure /= numberSensors;
343         }
344         else systemReady = false;
345 
346         return error;
347     }
348 
349     // Calculate temperature
350     // Caluculation in Bosch BMP280 datasheet p. 23
351     bool USSensorAPI::calculateTemperature(uint8_t* rawData,double* temperature)
352     {
353         bool error = false;
354         int32_t adcT;
355         uint16_t digT1;
356         int16_t digT2,digT3;
357         double var1,var2,tFine;
358 
359         // Check if nullptr
360         if(rawData == nullptr || temperature == nullptr) error = true;
361 
362         if(!error)
363         {
364             // Get values
365             try
366             {
367                 adcT = rawData[5] << 16 | rawData[6] << 8 | rawData[7];
368                 digT1 = rawData[9] << 8 | rawData[8];
369                 digT2 = rawData[11] << 8 | rawData[10];
370                 digT3 = rawData[13] << 8 | rawData[12];
371             }
372             catch(const std::exception& e)
373             {
374                 error = true;
375             }
376 
377             // Calculate temperature
378             if(!error)
379             {
380                 var1 = (((double)adcT) / 16384.0 - ((double)digT1) / 1024.0) * ((double)digT2);
381                 var2 = ((((double)adcT) / 131072.0 - ((double)digT1) / 8192.0) * (((double)adcT) / 131072.0 - ((double)digT1)
382                         / 8192.0)) * ((double)digT3);
383                 tFine = var1 + var2;
384                 *temperature = tFine / 5120.0;
385             }
386         }
387 
388         return error;
389     }
390 
391     // Calculate pressure
392     // Caluculation in Bosch BMP280 datasheet p. 23
393     bool USSensorAPI::calculatePressure(uint8_t* rawData,double* pressure,double *temperature)
394     {
395         bool error = false;
396         int32_t adcP;
397         uint16_t digP1;
398         int16_t digP2,digP3,digP4,digP5,digP6,digP7,digP8,digP9;
399         double var1,var2,tFine;
400 
401         // Check if nullptr
402         if(rawData == nullptr || pressure == nullptr) error = true;
403 
404         if(!error)
405         {
406             // Get values
407             try
408             {
```

```
409                 adcP = rawData[2] << 16 | rawData[3] << 8 | rawData[4];
410                 digP1 = rawData[15] << 8 | rawData[14];
411                 digP2 = rawData[17] << 8 | rawData[16];
412                 digP3 = rawData[19] << 8 | rawData[18];
413                 digP4 = rawData[21] << 8 | rawData[20];
414                 digP5 = rawData[23] << 8 | rawData[22];
415                 digP6 = rawData[25] << 8 | rawData[24];
416                 digP7 = rawData[27] << 8 | rawData[26];
417                 digP8 = rawData[29] << 8 | rawData[28];
418                 digP9 = rawData[31] << 8 | rawData[30];
419             }
420             catch(const std::exception& e)
421             {
422                 error = true;
423             }
424 
425             // Calculate temperature
426             if(!error)
427             {
428                 tFine = *temperature * 5120.0;
429                 var1 = (tFine / 2.0) - 64000.0;
430                 var2 = var1 * var1 * ((double)digP6) / 32768.0;
431                 var2 = var2 + var1 * ((double)digP5) * 2.0;
432                 var2 = (var2 / 4.0) + (((double)digP4) * 65536.0);
433                 var1 = (((double)digP3) * var1 * var1 / 524288.0 + ((double)digP2) * var1) / 524288.0;
434                 var1 = (1.0 + var1 / 32768.0) * ((double)digP1);
435                 *pressure = 1048576.0 - (double)adcP;
436                 *pressure = (*pressure - (var2 / 4096.0)) * 6250.0 / var1;
437                 var1 = ((double)digP9) * *pressure * *pressure / 2147483648.0;
438                 var2 = *pressure * ((double)digP8) / 32768.0;
439                 *pressure = *pressure + (var1 + var2 + ((double)digP7)) / 16.0;
440             }
441         }
442 
443         return error;
444     }
445 
446     // Get number of sensors
447     bool USSensorAPI::getNumberSensors(int* numSensors)
448     {
449         bool error = false;
450 
451         // Check if number of sensors unequal / greater than null
452         if(numberSensors > 0) *numSensors = numberSensors;
453         else error = true;
454 
455         return error;
456     }
457 
458     // Get system status
459     bool USSensorAPI::getSystemStatus(void)
460     {
461         return systemReady;
462     }
463 }
```

SensorAPI.hpp

```
1  /*****************************************************************************/
2  /* University: HAW Hamburg                                                   */
3  /* Author: Christopher Rotzlawski                                            */
4  /*                                                                           */
5  /* Project: Surrounding Sensor-based Naviagtionssystem                       */
6  /* Version: 1.0                                                              */
7  /*****************************************************************************/
8  #ifndef SENSORAPI_HPP
9  #define SENSORAPI_HPP
10 
11 // Includes
12 #include <string>
13 #include <cstring>
14 #include <fstream>
15 #include <sstream>
16 #include <iostream>
17 #include <vector>
18 #include <unistd.h>
19 #include <sys/socket.h>
20 #include <sys/ioctl.h>
21 #include <net/if.h>
22 #include <netinet/in.h>
23 #include <arpa/inet.h>
```

```
24  #include <jsoncpp/json/reader.h>
25
26  namespace uml
27  {
28      class SensorAPI
29      {
30          // Attributes
31      protected:
32          std::string ip;
33          std::string srcIP;
34          std::vector<std::string> dstIP;
35          std::vector<std::string> sensorName;
36          int port;
37          int sock;
38          int sensorOffset;
39          int numberSensors;
40          int systemSensors;
41          bool systemReady;
42          uint16_t checkSum;
43          std::string networkIface;
44          std::string sensorsystem;
45          std::string configFilePath;
46          struct sockaddr_in sockAddr;
47          struct sockaddr_in destinationAddr;
48          uint8_t statusMsg[2];
49          uint8_t tempMsg[2];
50          uint8_t measurementMsg[2];
51          uint8_t endMeasurement[2];
52
53          // Methods
54      public:
55          SensorAPI();
56          ~SensorAPI();
57      protected:
58          bool initSocket(void);
59          bool getMachineIP(void);
60          bool readConfig(void);
61          bool openSocket(void);
62          bool closeSocket(void);
63          void setDestination(void);
64          void sendMessage(std::string* dstIP,uint8_t* messagePtr,uint8_t* messageLen);
65          bool checkSensors(void);
66          bool checkStatus(std::string* dstIP);
67          bool checkNumSensors(void);
68      };
69  }
70
71  #endif // SENSORAPI_HPP
```

SensorAPI.cpp

```
1   /*****************************************************************/
2   /* University: HAW Hamburg                                       */
3   /* Author: Christopher Rotzlawski                                */
4   /*                                                               */
5   /* Project: Surrounding Sensor-based Naviagtionssystem           */
6   /* Version: 1.0                                                  */
7   /*****************************************************************/
8   #include "SensorAPI.hpp"
9
10  namespace uml
11  {
12      // Methode implementation
13      SensorAPI::SensorAPI()
14      {
15          // NOP
16      }
17
18      SensorAPI::~SensorAPI()
19      {
20          // NOP
21      }
22
23      // Initialize socket
24      bool SensorAPI::initSocket(void)
25      {
26          bool error = false;
27
28          error = readConfig();
29          if(!error) error = getMachineIP();
30          if(!error) error = openSocket();
```

```
31          if(!error) setDestination();
32
33          return error;
34      }
35
36      // Get IP of the machine
37      bool SensorAPI::getMachineIP(void)
38      {
39          bool error = false;
40
41          int netSock;
42          struct ifreq ifr;
43
44          // Open socket
45          netSock = socket(AF_INET,SOCK_DGRAM,0);
46
47          // Get IPv4 adress
48          ifr.ifr_addr.sa_family = AF_INET;
49
50          // Get ethernet adress
51          strncpy(ifr.ifr_name,networkIface.c_str(),IFNAMSIZ-1);
52
53          ioctl(netSock,SIOCGIFADDR,&ifr);
54
55          // Close socket
56          close(netSock);
57
58          // Set ip
59          ip = inet_ntoa(((struct sockaddr_in *)&ifr.ifr_addr)->sin_addr);
60
61          try
62          {
63              if(!ip.compare(ip.size()-2,2,".0"))
64              {
65                  std::cout << "Get wrong machine IP: " << ip << std::endl;
66                  error = true;
67              }
68              else if(ip.compare(srcIP))
69              {
70                  std::cout << "Get wrong machine IP: " << ip << std::endl;
71                  std::cout << "Expected IP: " << srcIP << std::endl;
72                  error = true;
73              }
74          }
75          catch(const std::exception& e)
76          {
77              std::cout << "Get machine IP failed. Failure in str.compare()." << std::endl;
78              error = true;
79          }
80
81          return error;
82      }
83
84      // Read config file
85      bool SensorAPI::readConfig(void)
86      {
87          bool error = false;
88
89          Json::Reader reader;
90          Json::Value file;
91
92          // Load config file
93          std::ifstream configFile(configFilePath);
94
95          // Check if file exist
96          if(configFile.fail()) error = true;
97
98          if(!error)
99          {
100             // Parse config file
101             reader.parse(configFile,file);
102
103             // Get values
104             try
105             {
106                 sensorsystem.assign(file["positioningSystem"].asString());
107                 networkIface.assign(file["network"]["iface"].asString());
108                 port = file["network"]["port"].asInt();
109                 srcIP = file["network"]["ip"].asString();
110                 numberSensors = file["sensors"].size();
111                 statusMsg[0] = file["messages"]["status"][0].asInt();
112                 statusMsg[1] = file["messages"]["status"][1].asInt();
113                 tempMsg[0] = file["messages"]["temp"][0].asInt();
114                 tempMsg[1] = file["messages"]["temp"][1].asInt();
115                 measurementMsg[0] = file["messages"]["measurement"][0].asInt();
```

```
116                 measurementMsg[1] = file["messages"]["measurement"][1].asInt();
117                 endMeasurement[0] = file["messages"]["end measurement"][0].asInt();
118                 endMeasurement[1] = file["messages"]["end measurement"][1].asInt();
119
120                 dstIP.reserve(numberSensors);
121                 sensorName.reserve(numberSensors);
122                 for(int i = 0; i < numberSensors; i++)
123                 {
124                     dstIP.push_back(file["sensors"][i]["sensorIp"].asString());
125                     sensorName.push_back(file["sensors"][i]["name"].asString());
126                 }
127             }
128             catch(const std::exception& e)
129             {
130                 error = true;
131             }
132         }
133
134         if(error) std::cout << "Read config file failed." << std::endl;
135
136         return error;
137     }
138
139     // Open socket
140     bool SensorAPI::openSocket(void)
141     {
142         bool error = false;
143         struct timeval tv;
144
145         // Open socket
146         if((sock = socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP)) == -1) error = true;
147
148         if(!error)
149         {
150             // Set adress
151             memset((char*)&sockAddr,0,sizeof(sockAddr));
152             sockAddr.sin_addr.s_addr = inet_addr(ip.c_str());
153             sockAddr.sin_family = AF_INET;
154             sockAddr.sin_port = htons(port);
155
156             // Bind socket
157             if(bind(sock,(struct sockaddr*)&sockAddr,sizeof(sockAddr)) == -1) error = true;
158
159             // Set timeout
160             tv.tv_sec = 0;
161             tv.tv_usec = 100000;
162             if(setsockopt(sock,SOL_SOCKET,SO_RCVTIMEO,(struct timeval *)&tv,sizeof(struct timeval))) error = true;
163         }
164
165         if(error) std::cout << "Open socket failed." << std::endl;
166
167         return error;
168     }
169
170     // Close socket
171     bool SensorAPI::closeSocket(void)
172     {
173         bool error = false;
174
175         error =  close(sock);
176
177         if(error) std::cout << "Close socket failed." << std::endl;
178
179         return error;
180     }
181
182     // Set Destination
183     void SensorAPI::setDestination(void)
184     {
185         // Set adress
186         memset((char*)&destinationAddr,0,sizeof(destinationAddr));
187         destinationAddr.sin_family = AF_INET;
188         destinationAddr.sin_port = htons(port);
189     }
190
191     // Send message
192     void SensorAPI::sendMessage(std::string* dstIP,uint8_t* messagePtr,uint8_t* messageLen)
193     {
194         // Set destination IP
195         destinationAddr.sin_addr.s_addr = inet_addr((*dstIP).c_str());
196
197         // Send message
198         sendto(sock,messagePtr,*messageLen,0,(struct sockaddr*)&destinationAddr,sizeof(sockAddr));
199     }
200
```

```
201     // Check sensors
202     bool SensorAPI::checkSensors(void)
203     {
204         bool error = false;
205         bool errorTemp = false;
206 
207         systemSensors = 0;
208         checkSum = 0;
209 
210         for(int i = 0; i < numberSensors; i++)
211         {
212             // Check sensor
213             if(checkStatus(&dstIP[i])) errorTemp = true;
214             else errorTemp = false;
215 
216             // Check if error
217             if(!errorTemp) systemSensors++;
218             else
219             {
220                 std::cout << sensorName[i] << " not found!" << std::endl;
221                 error = true;
222             }
223         }
224 
225         return error;
226     }
227 
228     // Check status
229     bool SensorAPI::checkStatus(std::string* dstIP)
230     {
231         bool error = false;
232         uint8_t checkSumByteOne;
233         uint8_t checkSumByteTwo;
234         uint8_t buffer[4] = {0,0,0,0};
235         uint8_t lenBuffer = sizeof(buffer);
236 
237         // Check if ptr is nullptr
238         if(dstIP == nullptr) error = true;
239 
240         if(!error)
241         {
242             // Build message
243             checkSumByteOne = checkSum & 0xFF;
244             checkSumByteTwo = (checkSum & 0xFF00) >> 8;
245             uint8_t data[4] = {statusMsg[0],statusMsg[1],checkSumByteTwo,checkSumByteOne};
246 
247             // Send message
248             sendMessage(dstIP,&data[0],&lenBuffer);
249 
250             // Check response
251             recv(sock,&buffer[0],sizeof(buffer),0);
252             for(int i = 0; i < lenBuffer; i++)
253                 if(!(buffer[i] == data[i])) error = true;
254 
255             // Clear buffer
256             for(int8_t i = 0; i < lenBuffer; i++) buffer[i] = 0;
257 
258             // Increment checkSum
259             checkSum++;
260         }
261 
262         return error;
263     }
264 
265     // Check number of sensors
266     bool SensorAPI::checkNumSensors(void)
267     {
268         bool error = false;
269 
270         // If number of found sensors != expected sensors print message
271         if(systemSensors != numberSensors)
272         {
273             std::cout << "Expected number of sensors: " << numberSensors << std::endl;
274             std::cout << "Found number of sensors: " << systemSensors << std::endl;
275 
276             error = true;
277         }
278 
279         return error;
280     }
281 }
```

D.3. Ultraschallmerkmalsextraktion

ultrasoundFeatureExtraction.cpp

```
1  /*****************************************************************/
2  /* University: HAW Hamburg                                       */
3  /* Author: Christopher Rotzlawski                                */
4  /*                                                               */
5  /* Project: Surrounding Sensor-based Naviagtionssystem           */
6  /* Version: 1.0                                                  */
7  /*****************************************************************/
8  #include <iostream>
9  #include <ros/package.h>
10 #include "../include/ultrasoundFeatureExtraction/extractFeatures.hpp"
11
12 #define PACKAGENAME "ultrasoundPositioningSystem"
13 #define NODENAME "ultrasoundFeatureExtraction"
14 #define SUBSCRIBECHECK "checkUltrasoundSystem"
15 #define SUBSCRIBECHODATA "rawUltrasoundData"
16 #define PUBLISHCHECKRESPONSE "responseUltrasoundSytem"
17 #define PUBLISHFEATURES "ultrasoundFeatures"
18 #define PUBLISHNORMALIZED "ultrasoundNormalized"
19 #define CONFIGFILEPATH (ros::package::getPath(PACKAGENAME)+"/config/positioningSystemConfig.json").c_str()
20
21 // Start node
22 int main(int argc,char** argv)
23 {
24     bool error = false;
25
26     uml::extractFeatures* extractFeatures = new uml::extractFeatures(CONFIGFILEPATH);
27     error = !extractFeatures->getStatus();
28
29     if(!error)
30         extractFeatures->startNode(argc,argv,NODENAME,SUBSCRIBECHODATA,PUBLISHFEATURES,PUBLISHNORMALIZED
31                                    ,SUBSCRIBECHECK,PUBLISHCHECKRESPONSE);
32     else
33         std::cout << "ERROR: System not ready. Node ultrasoundFeatureExtraction stopped." << std::endl;
34
35     if(!error)
36         std::cout << "Node ultrasoundFeatureExtraction stopped." << std::endl;
37
38     return 0;
39 }
```

extractFeatures.hpp

```
1  /*****************************************************************/
2  /* University: HAW Hamburg                                       */
3  /* Author: Christopher Rotzlawski                                */
4  /*                                                               */
5  /* Project: Surrounding Sensor-based Naviagtionssystem           */
6  /* Version: 1.0                                                  */
7  /*****************************************************************/
8  #ifndef EXTRACTFEATURES_HPP
9  #define EXTRACTFEATURES_HPP
10
11 #include <iostream>
12 #include <string>
13 #include <fstream>
14 #include <vector>
15 #include <jsoncpp/json/reader.h>
16 #include <ros/ros.h>
17 #include "commonMsgUVPS/checkUltrasoundSystem.h"
18 #include "commonMsgUVPS/responseCheckUltrasoundSystem.h"
19 #include "commonMsgUVPS/rawUltrasoundData.h"
20 #include "commonMsgUVPS/ultrasoundFeatures.h"
21 #include "commonMsgUVPS/normalizedUltrasoundData.h"
22
23 //#define FEATURES
24
25 namespace uml
26 {
27     class extractFeatures
28     {
29         // Attributes
30     private:
31         std::string configFilePath;
32         std::vector<std::string> sensorName;
```

```
33          std::vector<std::vector<uint16_t>> echoData;
34          std::vector<std::vector<double>> normalizedData;
35          std::vector<std::vector<double>> maxima;
36          std::vector<std::vector<double>> timeMaxima;
37          Json::Value configFile;
38          uint8_t numberSensors;
39          double sampleRate;
40          int measurementOffset;
41          int dataLength;
42          bool featureExtractionReady;
43          ros::Subscriber subCheckSystem;
44          ros::Subscriber subEchoData;
45          ros::Publisher pubCheckSystem;
46          ros::Publisher pubFeatures;
47          ros::Publisher pubNormalized;
48          commonMsgUVPS::responseCheckUltrasoundSystem responseSystem;
49          commonMsgUVPS::ultrasoundFeatures messageFeatures;
50          commonMsgUVPS::normalizedUltrasoundData messageNormalized;
51
52          // Methodes
53      public:
54          extractFeatures(std::string filePath);
55          ~extractFeatures();
56          bool getStatus();
57          void startNode(int argc,char** argv,std::string nodeName,std::string subscribeEchoData,std::string publishFeatures
58                         ,std::string publishNormalized,std::string subscribeCheck,std::string publishCheckResponse);
59      private:
60          bool readConfig();
61          void listnerCheckSystem(const commonMsgUVPS::checkUltrasoundSystemPtr& msg);
62          void listnerEchoData(const commonMsgUVPS::rawUltrasoundDataPtr& msg);
63          void publishFeatures();
64          void publishNormalized();
65          bool splitData(const commonMsgUVPS::rawUltrasoundDataPtr& msg);
66          bool normalizeData();
67          bool findMaxima();
68          void setMaxima(uint16_t maximum,int timeValue,int sensorNumber);
69          void convertTime(double& temperature);
70      };
71  }
72
73  #endif // EXTRACTFEATURES_HPP
```

extractFeatures.cpp

```
1   /*****************************************************************************/
2   /* University: HAW Hamburg                                                   */
3   /* Author: Christopher Rotzlawski                                            */
4   /*                                                                           */
5   /* Project: Surrounding Sensor-based Naviagtionssystem                       */
6   /* Version: 1.0                                                              */
7   /*****************************************************************************/
8   #include "extractFeatures.hpp"
9
10  namespace uml
11  {
12      // Initialize node
13      extractFeatures::extractFeatures(std::string filePath)
14      {
15          bool error = false;
16          configFilePath = filePath;
17
18          // Read config file
19          error = readConfig();
20
21          // Initialize sensorData and maxima vector
22          if(!error)
23          {
24              for(int i = 0; i < numberSensors; i++)
25              {
26                  std::vector<uint16_t> intVec;
27                  std::vector<double> doubleVec;
28                  std::vector<double> doubleVecShort(5,0.0);
29                  echoData.push_back(intVec);
30                  normalizedData.push_back(doubleVec);
31                  maxima.push_back(doubleVecShort);
32                  timeMaxima.push_back(doubleVecShort);
33              }
34          }
35
36          // Check if error
37          if(error) featureExtractionReady = false;
```

```
38          else featureExtractionReady = true;
39      }
40
41      extractFeatures::~extractFeatures()
42      {
43          // NOP
44      }
45
46      // Read config file
47      bool extractFeatures::readConfig()
48      {
49          bool error = false;
50          Json::Reader reader;
51
52          // Load config file
53          std::ifstream configStream(configFilePath);
54
55          // Check if file exist
56          if(configStream.fail()) error = true;
57
58          if(!error)
59          {
60              // Parse config file
61              reader.parse(configStream,configFile);
62
63              // Get number and names of sensors and sample rate os system
64              try
65              {
66                  numberSensors = configFile["sensors"].size();
67                  sampleRate = configFile["sensorProperties"]["sampleRate[kHz]"].asDouble()*1000.0;
68                  measurementOffset = configFile["sensorProperties"]["measurementOffset"].asInt();
69
70                  for(int i = 0; i < numberSensors; i++)
71                      sensorName.push_back(configFile["sensors"][i]["name"].asString());
72              }
73              catch (const std::exception& e)
74              {
75                  error = true;
76              }
77          }
78
79          if(error) std::cout << "Read config file failed." << std::endl;
80
81          return error;
82      }
83
84      // Get status of node
85      bool extractFeatures::getStatus()
86      {
87          return featureExtractionReady;
88      }
89
90      // Start node
91      void extractFeatures::startNode(int argc,char** argv,std::string nodeName,std::string subscribeEchoData,std::string publishFeatures
92                                      ,std::string publishNormalized,std::string subscribeCheck,std::string publishCheckResponse)
93      {
94          ros::init(argc,argv,nodeName);
95          ros::NodeHandle n;
96
97          subCheckSystem = n.subscribe(subscribeCheck,10,&extractFeatures::listnerCheckSystem,this);
98          subEchoData = n.subscribe(subscribeEchoData,10,&extractFeatures::listnerEchoData,this);
99          pubCheckSystem = n.advertise<commonMsgUVPS::responseCheckUltrasoundSystem>(publishCheckResponse,10);
100         pubFeatures = n.advertise<commonMsgUVPS::ultrasoundFeatures>(publishFeatures,10);
101         pubNormalized = n.advertise<commonMsgUVPS::normalizedUltrasoundData>(publishNormalized,10);
102
103         ros::spin();
104     }
105
106     // Callback for check system
107     void extractFeatures::listnerCheckSystem(const commonMsgUVPS::checkUltrasoundSystemPtr& msg)
108     {
109         if(msg->system == "ultrasoundFeatures")
110         {
111             responseSystem.system = msg->system;
112             responseSystem.ready = featureExtractionReady;
113
114             pubCheckSystem.publish(responseSystem);
115         }
116     }
117
118     // Callback for extract features
119     void extractFeatures::listnerEchoData(const commonMsgUVPS::rawUltrasoundDataPtr& msg)
120     {
121         bool error = false;
122
```

```
123         // Split echoData
124         error = splitData(msg);
125
126 #ifndef FEATURES
127         // Normalize data
128         if(!error) error = normalizeData();
129
130         // Publish message
131         if(!error) publishNormalized();
132 #else
133         // Find maxima in sensorData
134         if(!error) error = findMaxima();
135
136         // Convert time values of maxima
137         if(!error) convertTime(msg->temperature);
138
139         // Publish message
140         if(!error) publishFeatures();
141 #endif // FEATURES
142
143         // Clear sensorData vector
144         for(int i = 0; i < numberSensors; i++)
145         {
146             std::fill(maxima[i].begin(),maxima[i].end(),0.0);
147             std::fill(timeMaxima[i].begin(),timeMaxima[i].end(),0.0);
148             echoData[i].clear();
149             normalizedData[i].clear();
150         }
151     }
152
153     // Publish extracted features
154     void extractFeatures::publishFeatures()
155     {
156         std::vector<double> tempVector;
157
158         for(int i = 0; i < numberSensors; i++)
159             tempVector.insert(tempVector.end(),maxima[i].begin(),maxima[i].end());
160         messageFeatures.maxima = tempVector;
161
162         tempVector.clear();
163
164         for(int i = 0; i < numberSensors; i++)
165             tempVector.insert(tempVector.end(),timeMaxima[i].begin(),timeMaxima[i].end());
166         messageFeatures.timeMaxima = tempVector;
167
168         pubFeatures.publish(messageFeatures);
169     }
170
171     // Publish normalized echo data
172     void extractFeatures::publishNormalized()
173     {
174         std::vector<double> tempVector;
175
176         for(int i = 0; i < numberSensors; i++)
177         {
178             //tempVector.push_back(static_cast<double>(i));
179             tempVector.insert(tempVector.end(),normalizedData[i].begin(),normalizedData[i].end());
180         }
181
182         messageNormalized.dataLen = tempVector.size();
183         messageNormalized.numberSensors = numberSensors;
184         messageNormalized.normalizedData = tempVector;
185
186         pubNormalized.publish(messageNormalized);
187     }
188
189     // Split echo data
190     bool extractFeatures::splitData(const commonMsgUVPS::rawUltrasoundDataPtr& msg)
191     {
192         bool error = false;
193
194         dataLength = msg->dataLen/numberSensors;
195
196         try
197         {
198             for(int i = 0; i < numberSensors; i++)
199             {
200                 echoData[i].insert(echoData[i].begin(),&msg->echoData[(i*dataLength+(i+1)*measurementOffset)+1]
201                                    ,&msg->echoData[(i+1)*dataLength]);
202                 while(echoData[i].size() < (dataLength-1)) echoData[i].push_back(0);
203             }
204         }
205         catch(const std::exception& e)
206         {
207             error = true;
```

```
208             std::cout << "Split data failed." << std::endl;
209         }
210 
211         return error;
212     }
213 
214     // Normalize echo data
215     bool extractFeatures::normalizeData()
216     {
217         bool error = false;
218         double tempValue;
219 
220         try
221         {
222             for(int i = 0; i < numberSensors; i++)
223             {
224                 for(int j = 0; j < dataLength-1; j++)
225                 {
226                     tempValue = (static_cast<double>(echoData[i][j])/static_cast<double>(0xFFF)-0.3)/(1-0.3);
227 
228                     if(tempValue < 0.0) tempValue = 0.0;
229 
230                     normalizedData[i].push_back(tempValue);
231                 }
232             }
233         }
234         catch(const std::exception& e)
235         {
236             error = true;
237             std::cout << "Normalize data failed." << std::endl;
238         }
239 
240         return error;
241     }
242 
243     // Find peaks in echo data
244     bool extractFeatures::findMaxima()
245     {
246         bool error = false;
247         int timeValue = 0;
248         uint16_t localMaxima;
249 
250         try
251         {
252             for(int i = 0; i < numberSensors; i++)
253             {
254                 localMaxima = 0.0;
255 
256                 for(int j = 300; j < dataLength-1; j++)
257                 {
258                     if(echoData[i][j] > 1433 && echoData[i][j] > echoData[i][j-1] && echoData[i][j] > echoData[i][j+1])
259                     {
260                         if(echoData[i][j] > localMaxima)
261                         {
262                             localMaxima = echoData[i][j];
263                             timeValue = j;
264                         }
265                     }
266 
267                     if(timeValue != 0 && j-timeValue > 50)
268                     {
269                         setMaxima(localMaxima,timeValue,i);
270                         timeValue = 0;
271                         localMaxima = 0.0;
272                     }
273                 }
274             }
275         }
276         catch(const std::exception& e)
277         {
278             error = true;
279             std::cout << "Find maxima in sensor data failed." << std::endl;
280         }
281 
282         return error;
283     }
284 
285     // Set peaks in echo data
286     void extractFeatures::setMaxima(uint16_t maximum,int timeValue,int sensorNumber)
287     {
288         int index = 0;
289         double maximumTemp;
290 
291         // Find lowest value in maxima
292         for(int i = 1; i < 5; i++)
```

```
293                 if(maxima[sensorNumber][i] < maxima[sensorNumber][index]) index = i;
294
295             // Cast maximum to double
296             maximumTemp = (static_cast<double>(maximum)/static_cast<double>(0xFFF)-0.3)/(1-0.3);
297
298             // Set maxima
299             if(maximumTemp > maxima[sensorNumber][index])
300             {
301                 maxima[sensorNumber][index] = maximumTemp;
302                 timeMaxima[sensorNumber][index] = static_cast<double>(timeValue);
303             }
304         }
305
306         // Convert time
307         void extractFeatures::convertTime(double& temperature)
308         {
309             for(int i = 0; i < numberSensors; i++)
310                 for(int j = 0; j < 5; j++) timeMaxima[i][j] = timeMaxima[i][j]/sampleRate;
311         }
312     }
```

D.4. 3D-Raytracing-Simulation

ultrasoundRaytracingSimulation.py

```
1   #!/usr/bin/env python3
2   #*********************************************************************#
3   #* University: HAW Hamburg                                           *#
4   #* Author: Christopher Rotzlawski                                    *#
5   #*                                                                   *#
6   #* Project: Surrounding Sensor-based Naviagtionssystem               *#
7   #* Version: 1.0                                                      *#
8   #*********************************************************************#
9   import rospy
10  import rospkg
11  import os
12  import simulateImpulseResponse
13  from commonMsgUVPS.msg import checkUltrasoundSystem
14  from commonMsgUVPS.msg import responseCheckUltrasoundSystem
15  from commonMsgUVPS.msg import startUltrasoundSimulation
16  from commonMsgUVPS.msg import visualizeUltrasoundRays
17  from commonMsgUVPS.msg import simulatedImpulseResponse
18
19  class ultrasoundRaytracingSimulation:
20      # Initialize node
21      def __init__(self):
22          self.PACKAGENAME = 'ultrasoundPositioningSystem'
23          self.ROSNODE = os.path.basename(os.path.dirname(os.path.realpath(__file__)))
24          self.PUBLISHCHECKRESPONSE = 'responseUltrasoundSytem'
25          self.PUBLISHVISUALIZERAYS = 'ultrasoundSimulationRaytrace'
26          self.PUBLISHIMPULSERESPONSE = 'simulatedImpulseResponse'
27          self.LISTNERCHECK = 'checkUltrasoundSystem'
28          self.LISTNERSTART = 'startUltrasoundSimulation'
29          self.CONFIGFILEPATH = '/config/positioningSystemConfig.json'
30          self.MAPFILEPATH = '/config/map.json'
31
32          rospack = rospkg.RosPack()
33          rospack.list()
34          configFilePath = rospack.get_path(self.PACKAGENAME) + self.CONFIGFILEPATH
35          mapFilePath = rospack.get_path(self.PACKAGENAME) + self.MAPFILEPATH
36
37          # Initialize simulation
38          self.simulation = simulateImpulseResponse.simulateImpulseResponse(configFilePath,mapFilePath)
39
40          self.pubCheckResponse = rospy.Publisher(self.PUBLISHCHECKRESPONSE, responseCheckUltrasoundSystem, queue_size=10)
41          self.pubVisualizeRays = rospy.Publisher(self.PUBLISHVISUALIZERAYS, visualizeUltrasoundRays, queue_size=10)
42          self.pubImpulseResponse = rospy.Publisher(self.PUBLISHIMPULSERESPONSE, simulatedImpulseResponse, queue_size=10)
43
44          self.checkResponseMessage = responseCheckUltrasoundSystem()
45          self.visualizeMessage = visualizeUltrasoundRays()
46          self.impulseResponseMessage = simulatedImpulseResponse()
47
48          rospy.init_node(self.ROSNODE)
49
50      # Publish simulation results
51      def talker(self,data):
52          timeStamp = data.header.stamp
```

```
53          latitude = data.latitude
54          longitude = data.longitude
55          direction = data.direction
56          temperature = data.temperature
57          transmitter = data.transmitterSensor
58
59          if self.simulation.systemReady:
60              error,impulseResponse,rayTrace = self.simulation.simulate(longitude,latitude,direction,temperature,transmitter)
61
62              self.impulseResponseMessage.length = len(impulseResponse)
63              self.impulseResponseMessage.impulseResponse = impulseResponse
64              self.pubImpulseResponse.publish(self.impulseResponseMessage)
65
66              if not error:
67                  # Publish raytrace
68                  for i in range(len(rayTrace)):
69                      self.visualizeMessage.rayTrace = rayTrace[i]
70                      self.visualizeMessage.length = len(rayTrace[i])
71                      self.pubVisualizeRays.publish(self.visualizeMessage)
72
73      # Listner for check system
74      def checkSytem(self,data):
75          if data.system == 'ultrasoundSimulation':
76              self.checkResponseMessage.system = 'ultrasoundSimulation'
77              self.checkResponseMessage.ready = self.simulation.systemReady
78
79              self.pubCheckResponse.publish(self.checkResponseMessage)
80
81      # Start subscriber
82      def listner(self):
83          rospy.Subscriber(self.LISTNERCHECK,checkUltrasoundSystem,self.checkSytem)
84          rospy.Subscriber(self.LISTNERSTART,startUltrasoundSimulation,self.talker)
85          rospy.spin()
86
87  if __name__ == '__main__':
88      try:
89          node = ultrasoundRaytracingSimulation()
90          node.listner()
91      except rospy.ROSInterruptException:
92          print('ERROR: start ultrasound raytracing simulation failed!')
```

simulateImpulseResponse.py mit Simulation

```
1   #!/usr/bin/env python3
2   #*****************************************************************************#
3   #* University: HAW Hamburg                                                   *#
4   #* Author: Christopher Rotzlawski                                            *#
5   #*                                                                           *#
6   #* Project: Surrounding Sensor-based Naviagtionssystem                       *#
7   #* Version: 1.0                                                              *#
8   #*****************************************************************************#
9   import getParameter
10  import math
11  import numpy
12  import json
13  from scipy.special import jv
14  from multiprocessing import Process,Queue,cpu_count
15
16  import time
17
18  class simulateImpulseResponse:
19      # Initialize simulation
20      def __init__(self,configFilePath,mapFilePath):
21          self.systemReady = False
22
23          self.parameter = getParameter.getParameter(configFilePath,mapFilePath)
24
25          self.systemReady = self.parameter.isParameterReady()
26
27      # Simulate position
28      def simulate(self,positionX,positionY,vehicleDirection,temperature,transmitter):
29          error = True
30          impulseResponse = []
31          rayTrace = []
32
33          if self.systemReady:
34              self.parameter.setTemperature(temperature)
35
36              vehicleDirection = vehicleDirection*math.pi/180
37              vehicleDirection = vehicleDirection%(2*math.pi)
38              vehiclePosition = [positionX,positionY,0.0]
```

```
39
40              startX = positionX+self.parameter.sensors[transmitter][1]*math.cos(vehicleDirection)
41                     -self.parameter.sensors[transmitter][2]*math.sin(vehicleDirection)
42              startY = positionY+self.parameter.sensors[transmitter][1]*math.sin(vehicleDirection)
43                     +self.parameter.sensors[transmitter][2]*math.cos(vehicleDirection)
44              startZ = self.parameter.sensors[transmitter][3]
45              startOfRay = [startX,startY,startZ]
46
47              startTime = time.time()
48
49              # Get number of workers
50              numberProcesses = cpu_count()-2
51              numberChunks = numberProcesses*2
52
53              # Split simulation angles for multiprocessing
54              simulationAngels = []
55              simulationAngelSize = int(self.parameter.numberSimulationAngels/numberChunks)
56              for i in range(numberChunks):
57                  if i < numberChunks-1:
58                      simulationAngels.append(self.parameter.simulationAngels[i*simulationAngelSize:(i+1)*simulationAngelSize])
59                  else:
60                      simulationAngels.append(self.parameter.simulationAngels[i*simulationAngelSize:])
61
62              # Start processes
63              self.queue = []
64              processes = []
65              for i in range(numberProcesses):
66                  self.queue.append(Queue())
67                  processes.append(Process(target=self.simulateRays,args=(simulationAngels[i],startOfRay,vehicleDirection
68                                  ,vehiclePosition,transmitter,i)))
69                  processes[i].start()
70                  print("Process "+str(i)+" started.")
71
72              simulationAngleMarker = i
73
74              # Get simulation results
75              results = []
76              activeProcesses = numberProcesses
77              while activeProcesses > 0:
78                  for i in range(numberProcesses):
79                      if self.queue[i]:
80                          try:
81                              results.append(self.queue[i].get(False))
82
83                              if simulationAngleMarker < numberChunks-1:
84                                  print("Process "+str(i)+" succeeded.")
85                                  simulationAngleMarker += 1
86                                  self.queue[i] = Queue()
87                                  processes[i] = Process(target=self.simulateRays,args=(simulationAngels[simulationAngleMarker]
88                                                  ,startOfRay,vehicleDirection,vehiclePosition,transmitter,i))
89                                  processes[i].start()
90                                  print("Process "+str(i)+" restarted.")
91                              else:
92                                  self.queue[i] = None
93                                  activeProcesses -= 1
94                                  print("Process "+str(i)+" finished.")
95                          except:
96                              pass
97
98                  time.sleep(0.01)
99
100             # Split and sort results
101             impulseResponse = []
102             rayTrace = []
103             for result in results:
104                 output = json.loads(result)
105                 if not output[0]:
106                     impulseResponse += output[1]
107                     rayTrace += output[2]
108
109             endTime = time.time()
110             print(str(endTime-startTime)+" s")
111             print("Hits: "+str(len(rayTrace)))
112             print("SimAngles: "+str(self.parameter.numberSimulationAngels))
113
114         return error,impulseResponse,rayTrace
115
116     # Simulate rays
117     def simulateRays(self,simulationAngels,startOfRay,vehicleDirection,vehiclePosition,transmitter,processNumber):
118         error = False
119         rayTrace = []
120         impulseResponse = []
121         sensorStartRay = startOfRay
122
123         # For each simulationAngels
```

```
124         for ray in simulationAngels:
125             rayLoop = True
126 
127             startOfRay = sensorStartRay
128 
129             rayOrientationAzimut = vehicleDirection+self.parameter.sensors[transmitter][4]+ray[1]
130             rayOrientationAzimut = rayOrientationAzimut%(2*math.pi)
131             rayOrientationElevation = ray[0]
132             rayOrientation = [rayOrientationElevation,rayOrientationAzimut]
133 
134             reflexionElement = self.parameter.sensors[transmitter][0]
135             newReflexionElement = ""
136 
137             rayLength = 0.0
138             newRayLength = 0.0
139             newReflectionFactor = 0.0
140             rayLengthObject = 0.0
141             rayLengthSensor = 0.0
142             newRayTrace = []
143 
144             angle = self.calculateVectorAngle(ray[1],ray[0])
145             rayIntensity = self.beamPattern(angle)
146 
147             sensorNumber = -1
148 
149             for i in range(len(startOfRay)):
150                 newRayTrace.append(startOfRay[i])
151 
152             while rayLoop:
153                 for mapObject in self.parameter.objects:
154                     if mapObject[0] != reflexionElement:
155                         # Check if object is hit
156                         isHit,pointObjectHit,rayLengthObject = self.calculateRayHit3D(startOfRay,rayOrientation,mapObject)
157 
158                         # Calculate angle of reflexion
159                         if isHit:
160                             error,reflexionDirection = self.calculateAngle(mapObject,pointObjectHit,startOfRay)
161 
162                         if isHit and not error:
163                             # Check ray length
164                             if rayLengthObject < newRayLength or not newRayLength:
165                                 newRayLength = rayLengthObject
166                                 newStartOfRay = pointObjectHit
167                                 newRayOrientation = reflexionDirection
168                                 newReflexionElement = mapObject[0]
169                                 newReflectionFactor = mapObject[4]
170 
171                 # Check if sensor is hit
172                 isHit,sensorNumber,rayLengthSensor = self.calculateSensorHit3D(startOfRay,rayOrientation,vehiclePosition
173                                                                                ,vehicleDirection,reflexionElement)
174 
175                 # Check ray and sensor direction
176                 if isHit:
177                     isHit,angle = self.checkSensorRayDirection(rayOrientation,vehicleDirection,sensorNumber)
178 
179                 # Get sensor hit
180                 if isHit:
181                     if rayLengthSensor < newRayLength or not newRayLength:
182                         newRayLength = rayLengthSensor
183                         newReflexionElement = self.parameter.sensors[sensorNumber][0]
184                         positionX = vehiclePosition[0]+self.parameter.sensors[sensorNumber][1]*math.cos(vehicleDirection)
185                                     -self.parameter.sensors[sensorNumber][2]*math.sin(vehicleDirection)
186                         positionY = vehiclePosition[1]+self.parameter.sensors[sensorNumber][1]*math.sin(vehicleDirection)
187                                     +self.parameter.sensors[sensorNumber][2]*math.cos(vehicleDirection)
188                         positionZ = self.parameter.sensors[sensorNumber][3]+vehiclePosition[2]
189                         newStartOfRay = [positionX,positionY,positionZ]
190                         newReflectionFactor = self.beamPattern(angle)
191                         rayLoop = False
192 
193                 # Check if ray hits
194                 if newReflexionElement != "":
195                     rayLength += newRayLength
196                     startOfRay = newStartOfRay
197                     rayOrientation = newRayOrientation
198                     reflexionElement = newReflexionElement
199                     newReflexionElement = ""
200                     newRayLength = 0.0
201                     rayIntensity *= newReflectionFactor
202                     newReflectionFactor = 0.0
203 
204                     for i in range(len(startOfRay)):
205                         newRayTrace.append(startOfRay[i])
206 
207                     # Check if length of ray is in range when sensor is hit
208                     if rayLength <= self.parameter.maxRayLength and not rayLoop:
```

```
209                         # impulseResponse = [sensor,timeValue,intensity,...]
210                         timeValue = rayLength/self.parameter.speedOfSound
211                         impulseResponse.append(sensorNumber)
212                         impulseResponse.append(timeValue)
213                         if timeValue <= 0.015:
214                             rayIntensityTemp = 1.0
215                         else:
216                             rayIntensityTemp = 1.0-16*(timeValue-0.015)
217                         rayIntensity = rayIntensity*rayIntensityTemp
218                         impulseResponse.append(rayIntensity)
219                         rayTrace.append(newRayTrace)
220
221                     # Check ray length
222                     elif rayLength > self.parameter.maxRayLength:
223                         rayLoop = False
224
225                     # Check amplitude of ray
226                     if rayIntensity < 0.05:
227                         rayLoop = False
228
229                 else:
230                     rayLoop = False
231
232             # Check return vectors
233             if len(impulseResponse) < 1:
234                 error = True
235             else:
236                 error = False
237
238             outputString = json.dumps([error,impulseResponse,rayTrace])
239             self.queue[processNumber].put(outputString)
240
241         # Check if ray hits object
242         def calculateRayHit3D(self,startOfRay,rayOrientation,mapObject):
243             error = False
244             isHit = False
245             equationMatrix = [[0.0,0.0,0.0],[0.0,0.0,0.0],[0.0,0.0,0.0]]
246             equationResolution = [0.0,0.0,0.0]
247             equationParameter = [0.0,0.0,0.0]
248             pointObjectHit = [0.0,0.0,0.0]
249             rayLength = 0.0
250
251             rayDirectionX = math.cos(rayOrientation[1])
252             rayDirectionY = math.sin(rayOrientation[1])
253             rayDirectionZ = math.sin(rayOrientation[0])
254             rayDirectionVector = [rayDirectionX,rayDirectionY,rayDirectionZ]
255
256             for i in range(len(equationMatrix)):
257                 equationMatrix[i] = [mapObject[2][i],mapObject[3][i],-rayDirectionVector[i]]
258                 equationResolution[i] = startOfRay[i]-mapObject[1][i]
259
260             # Calculate equation parameter [u,v,t] of vectorU, vectorV and rayDirection
261             try:
262                 equationParameter = numpy.linalg.solve(equationMatrix,equationResolution)
263             except:
264                 error = True
265
266             # Check if point hits object
267             if not error:
268                 isHit = self.checkHitObject(equationParameter)
269
270                 if isHit:
271                     # Direction parameter of equationParameter is length of ray
272                     rayLength = equationParameter[2]
273                     for i in range(len(pointObjectHit)):
274                         pointObjectHit[i] = startOfRay[i]+equationParameter[2]*rayDirectionVector[i]
275
276             return isHit,pointObjectHit,rayLength
277
278         # Check if ray hits sensor
279         def calculateSensorHit3D(self,startOfRay,rayOrientation,vehiclePosition,vehicleDirection,reflexionElement):
280             rayLengthSensor = 0.0
281             rayLength = 0.0
282             sensorPosition = [0.0,0.0,0.0]
283             lambdaVec = [0.0,0.0,0.0]
284             sensorNumber = -1
285
286             rayDirectionX = math.cos(rayOrientation[1])
287             rayDirectionY = math.sin(rayOrientation[1])
288             rayDirectionZ = math.sin(rayOrientation[0])
289             rayDirectionVector = [rayDirectionX,rayDirectionY,rayDirectionZ]
290
291             try:
292                 for i in range(len(self.parameter.sensors)):
293                     isHit = True
```

```
294
295                 if self.parameter.sensors[i][0] != reflexionElement:
296                     # Calculate position of sensor
297                     sensorPosition[0] = vehiclePosition[0]+self.parameter.sensors[i][1]*math.cos(vehicleDirection)
298                                         -self.parameter.sensors[i][2]*math.sin(vehicleDirection)
299                     sensorPosition[1] = vehiclePosition[1]+self.parameter.sensors[i][1]*math.sin(vehicleDirection)
300                                         +self.parameter.sensors[i][2]*math.cos(vehicleDirection)
301                     sensorPosition[2] = self.parameter.sensors[i][3]
302
303                     # Calculate parameter of vector direction
304                     directionValue = 0.0
305                     directionParameter = 0.0
306                     for j in range(len(rayDirectionVector)):
307                         if round(rayDirectionVector[j],3):
308                             directionValue = rayDirectionVector[j]
309                             directionParameter = (sensorPosition[j]-startOfRay[j])/directionValue
310                             break
311
312                     # Check if ray hits sensor
313                     squareValue = 0.0
314                     for j in range(len(sensorPosition)):
315                         value = startOfRay[j]+directionParameter*rayDirectionVector[j]
316                         squareValue += (value-sensorPosition[j])*(value-sensorPosition[j])
317
318                     squareValue = math.sqrt(squareValue)
319                     if squareValue > self.parameter.sensorDiameter:
320                         isHit = False
321
322                     # Calculate length of ray
323                     if isHit:
324                         error,rayLengthSensor = self.calculateRayLength(startOfRay,sensorPosition)
325
326                         if not error:
327                             if rayLengthSensor < rayLength or not rayLength:
328                                 rayLength = rayLengthSensor
329                                 sensorNumber = i
330
331         except:
332             isHit = False
333
334         if rayLength:
335             isHit = True
336         else:
337             isHit = False
338
339         return isHit,sensorNumber,rayLength
340
341     # Check if ray has correct direction to hit sensor
342     def checkSensorRayDirection(self,rayOrientation,vehicleDirection,sensorNumber):
343         isHit = False
344         sensorOrientationAzimut = 0.0
345         rayOrientationAzimut = 0.0
346         rayOrientationElevation = 0.0
347
348         try:
349             sensorOrientationAzimut = vehicleDirection+self.parameter.sensors[sensorNumber][4]
350             rayOrientationAzimut = rayOrientation[1]-math.pi-sensorOrientationAzimut
351             rayOrientationAzimut = rayOrientationAzimut%(2*math.pi)
352             rayOrientationElevation = -rayOrientation[0]
353             rayOrientationElevation = rayOrientationElevation%(2*math.pi)
354
355             angle = self.calculateVectorAngle(rayOrientationAzimut,rayOrientationElevation)
356
357             if angle <= self.parameter.angelBorder and angle >= -self.parameter.angelBorder:
358                 isHit = True
359
360         except:
361             isHit = False
362
363         return isHit,angle
364
365     # Check if object hit in boundary
366     def checkHitObject(self,parameter):
367         isHit = False
368         # Check if parameter of vectorU 0 < u < 1
369         if parameter[0] > 0.0 and parameter[0] < 1.0:
370             # Check if parameter of vectorV 0 < v < 1
371             if parameter[1] > 0.0 and parameter[1] < 1.0:
372                 # Check if object is in line of sight, parameter of rayDirection is t > 0
373                 if parameter[2] > 0.0:
374                     isHit = True
375
376         return isHit
377
378     # Calculate reflexion angle
```

```
379     def calculateAngle(self,mapObject,pointObjectHit,startOfRay):
380         error = False
381         norm = [0.0,0.0,0.0]
382         perpendicularPoint = [0.0,0.0,0.0]
383         vectorPerpendicular = [0.0,0.0,0.0]
384         mirrorPoint = [0.0,0.0,0.0]
385         deltaMirrorHit = [0.0,0.0,0.0]
386         reflexionDirection = [0.0,0.0]
387         pointParameter = 0.0
388         pointResult = 0.0
389 
390         try:
391             # Calculate narmal vector of plane
392             norm[0] = mapObject[2][1]*mapObject[3][2]-mapObject[2][2]*mapObject[3][1]
393             norm[1] = mapObject[2][2]*mapObject[3][0]-mapObject[2][0]*mapObject[3][2]
394             norm[2] = mapObject[2][0]*mapObject[3][1]-mapObject[2][1]*mapObject[3][0]
395 
396             # Calculate point on perpendicular staright
397             for i in range(len(norm)):
398                 pointParameter += norm[i]*norm[i]
399                 pointResult += startOfRay[i]*norm[i]-pointObjectHit[i]*norm[i]
400 
401             parameterStraight = pointResult/pointParameter
402 
403             for i in range(len(norm)):
404                 perpendicularPoint[i] = pointObjectHit[i]+parameterStraight*norm[i]
405 
406             # Vector form startOfRay to perpendicularPoint
407             for i in range(len(vectorPerpendicular)):
408                 vectorPerpendicular[i] = perpendicularPoint[i]-startOfRay[i]
409 
410             # Calculate mirror point
411             for i in range(len(mirrorPoint)):
412                 mirrorPoint[i] = startOfRay[i]+2*vectorPerpendicular[i]
413 
414             # Calculate delta between mirrorPoint and pointObjectHit
415             for i in range(len(deltaMirrorHit)):
416                 deltaMirrorHit[i] = mirrorPoint[i]-pointObjectHit[i]
417 
418             # Calculate azimut and elevation of reflexion
419             azimut = math.atan2(deltaMirrorHit[1],deltaMirrorHit[0])
420             lengthXY = math.sqrt(deltaMirrorHit[0]*deltaMirrorHit[0]+deltaMirrorHit[1]*deltaMirrorHit[1])
421             elevation = math.atan2(deltaMirrorHit[2],lengthXY)
422             reflexionDirection = [elevation,azimut]
423         except:
424             print("Failure in calculate reflexion angle.")
425             error = True
426 
427         return error,reflexionDirection
428 
429     # Calculate ray length
430     def calculateRayLength(self,startOfRay,pointObjectHit):
431         error = False
432         squareSum = 0.0
433         rayLength = 0.0
434 
435         try:
436             for i in range(len(startOfRay)):
437                 deltaPoint = pointObjectHit[i]-startOfRay[i]
438                 squareSum += deltaPoint*deltaPoint
439 
440             rayLength = math.sqrt(squareSum)
441         except:
442             print("Failure in calculate ray length.")
443             error = True
444 
445         return error,rayLength
446 
447     # Calculate vector angle
448     def calculateVectorAngle(self,azimut,elevation):
449         hPoint = math.tan(azimut)*self.parameter.adjacentSide
450         bPoint = math.tan(elevation)*self.parameter.adjacentSide
451         rLength = math.sqrt(hPoint*hPoint+bPoint*bPoint)
452         angle = math.atan(rLength/self.parameter.adjacentSide)
453 
454         return angle
455 
456     # Beam pattern modelling
457     def beamPattern(self,angle):
458         # Calculate bessel function
459         b = 0.49
460         value = (2*math.pi*self.parameter.sensorPiston/self.parameter.lambdaWave)*math.sin(angle)
461         j1 = jv(1,value)
462 
463         if value != 0.0:
```

```
464             h = 2*j1/value
465         else:
466             h = 1
467
468         hb = abs(h*(b+(1-b)*math.cos(angle)))*0.2+0.8
469
470         return hb
```

simulateImpulseResponse.py ohne Simulation

```
1   #!/usr/bin/env python3
2   #*****************************************************************************#
3   #* University: HAW Hamburg                                                   *#
4   #* Author: Christopher Rotzlawski                                            *#
5   #*                                                                           *#
6   #* Project: Surrounding Sensor-based Naviagtionssystem                       *#
7   #* Version: 1.0                                                              *#
8   #*****************************************************************************#
9   import getParameter
10  import os
11  import os.path
12  import ast
13  import rospkg
14
15  class simulateImpulseResponse:
16      # Initialize simulation
17      def __init__(self,configFilePath,mapFilePath):
18          self.systemReady = False
19          error = False
20
21          rospack = rospkg.RosPack()
22          rospack.list()
23
24          SIMULATIONRESULTSPATH = rospack.get_path('ultrasoundPositioningSystem')+'/src/ultrasoundRaytracingSimulation/simResults.csv'
25
26          self.parameter = getParameter.getParameter(configFilePath,mapFilePath)
27
28          error = self.readSimualtionResults(SIMULATIONRESULTSPATH)
29
30          if not error:
31              self.systemReady = self.parameter.isParameterReady()
32          else:
33              self.systemReady = False
34
35      # Load simulation data
36      def simulate(self,positionX,positionY,vehicleDirection,temperature,transmitter):
37          error = True
38          self.impulseResponse = []
39          self.rayTrace = []
40
41          if self.systemReady:
42              # Get simulation results
43              self.getValues(positionX,positionY,vehicleDirection,transmitter)
44
45              if len(self.rayTrace) > 0:
46                  error = False
47
48          return error,self.impulseResponse,self.rayTrace
49
50      # Read simulation file
51      def readSimualtionResults(self,simulationresultspath):
52          error = False
53
54          # Check if file exists
55          if type(simulationresultspath) is not str:
56              error = True
57              print("Wrong simualtion results file path.")
58          else:
59              error = not os.path.isfile(simulationresultspath)
60              if error:
61                  print("Simulation results file doesn't exist.")
62
63          if not error:
64              with open(simulationresultspath,'r') as readFile:
65                  self.data = readFile.read().split('\n')
66
67          return error
68
69      # Get simulation values
70      def getValues(self,positionX,positionY,vehicleDirection,transmitter):
71          for i in range(0,len(self.data)-1,3):
```

```
72              positionValues = list(map(float,self.data[i].split(',')))
73              if positionValues[0] == positionX and positionValues[1] == positionY:
74                  if positionValues[2] == vehicleDirection:
75                      if positionValues[4] == transmitter:
76                          self.impulseResponse = ast.literal_eval(self.data[i+1])
77                          self.rayTrace = ast.literal_eval(self.data[i+2])
```

getParameter.py

```
1   #!/usr/bin/env python3
2   #*****************************************************************************#
3   #* University: HAW Hamburg                                                   *#
4   #* Author: Christopher Rotzlawski                                            *#
5   #*                                                                           *#
6   #* Project: Surrounding Sensor-based Naviagtionssystem                       *#
7   #* Version: 1.0                                                              *#
8   #*****************************************************************************#
9   import os.path
10  import json
11  import math
12  import numpy
13
14  class getParameter:
15      # Initialize parameter
16      def __init__(self,configFilePath,mapFilePath):
17          self.parameterReady = False
18          self.temperature = 20 #default
19
20          # Check if file exists
21          if type(configFilePath) is not str:
22              error = True
23              print("Wrong config file path.")
24          else:
25              error = not os.path.isfile(configFilePath)
26              if error:
27                  print("Config file doesn't exist.")
28
29          if type(mapFilePath) is not str:
30              error = True
31              print("Wrong map file path.")
32          else:
33              error = not os.path.isfile(mapFilePath)
34              if error:
35                  print("Map file doesn't exist.")
36
37          # Parse json files
38          if not error:
39              with open(configFilePath,'r') as configFile:
40                  self.parsedConfig = json.load(configFile)
41              with open(mapFilePath,'r') as mapFile:
42                  self.parsedMap = json.load(mapFile)
43          else:
44              print("Read files failed.")
45
46          # Read config file
47          if not error:
48              error = self.readConfig()
49              if error:
50                  print("Get config parameter failed.")
51
52          # Read map file
53          if not error:
54              error = self.loadMap()
55              if error:
56                  print("Get map parameter failed.")
57
58          if not error:
59              self.parameterReady = True
60              self.calculateSimulationParameter()
61              self.calculateSimulationAngels()
62
63      # Get parameter status
64      def isParameterReady(self):
65          return self.parameterReady
66
67      # Read config
68      def readConfig(self):
69          error = False
70
71          try:
72              self.readEnvironment()
```

```
73              self.readSensorProperties()
74              self.readSensorsystem()
75          except:
76              error = True
77
78          return error
79
80      # Read environmental values
81      def readEnvironment(self):
82          self.maxSimulationTime = self.parsedConfig["generalEnvironment"]["maxSimulationTime[s]"]
83          self.minObjectSize = self.parsedConfig["generalEnvironment"]["minObjectSize[m]"]
84
85      # Read sensor properties
86      def readSensorProperties(self):
87          self.maxRayLength = self.parsedConfig["sensorProperties"]["maxRayLength[m]"]
88          self.maxDistance = self.parsedConfig["sensorProperties"]["maxDistance[m]"]
89          self.beamSpread = self.parsedConfig["sensorProperties"]["beamSpread[deg]"]
90          self.angelBorder = self.beamSpread*math.pi/180
91          self.sensorDiameter = self.parsedConfig["sensorProperties"]["sensorDiameter[m]"]
92          self.sensorPiston = self.parsedConfig["sensorProperties"]["sensorPistonRadius[m]"]
93          self.sensorFrequency = self.parsedConfig["sensorProperties"]["frequency[kHz]"]*1000
94
95      # Read system parameter
96      def readSensorsystem(self):
97          self.numberSensors = len(self.parsedConfig["sensors"])
98          self.sensors = []
99
100         for i in range(self.numberSensors):
101             sensorName = self.parsedConfig["sensors"][i]["name"]
102             positionX = self.parsedConfig["sensors"][i]["simPositionX[m]"]
103             positionY = self.parsedConfig["sensors"][i]["simPositionY[m]"]
104             positionZ = self.parsedConfig["sensors"][i]["simPositionZ[m]"]
105             direction = self.parsedConfig["sensors"][i]["simDirectionToVehicle[deg]"]*math.pi/180
106             self.sensors.append([sensorName,positionX,positionY,positionZ,direction])
107
108     # Calculate simulation parameter
109     def calculateSimulationParameter(self):
110         w = self.minObjectSize/2
111         self.angleResolution = 2*math.atan((w/2)/(self.maxDistance))
112         self.speedOfSound = 331.4+0.6*self.temperature #default
113         self.lambdaWave = self.speedOfSound/self.sensorFrequency
114
115         self.unitCircle = 1
116         self.adjacentSide = self.unitCircle/math.tan(self.angelBorder)
117
118     # Set temperature
119     def setTemperature(self,temperature):
120         self.temperature = temperature
121         self.speedOfSound = 331.4+0.6*self.temperature
122         self.lambdaWave = self.speedOfSound/self.sensorFrequency
123
124     # Calculate rays
125     def calculateSimulationAngels(self):
126         self.simulationAngels = []
127
128         # Elevation
129         for el in numpy.arange(0,self.angelBorder,self.angleResolution):
130             bPoint = math.tan(el)*self.adjacentSide
131
132             # Azimut
133             for az in numpy.arange(0,self.angelBorder,self.angleResolution):
134                 hPoint = math.tan(az)*self.adjacentSide
135
136                 # Check length
137                 rLength = math.sqrt(bPoint*bPoint+hPoint*hPoint)
138                 if rLength <= self.unitCircle:
139                     # [Elevation,Azimut] in all quadrant Q1 = [El,Az], Q2 = [El,-Az], Q3 = [-El,-Az], Q4 = [-El,Az]
140                     self.simulationAngels.append([el,az])
141                     if az != 0.0:
142                         self.simulationAngels.append([el,-az])
143                     if el != 0.0:
144                         self.simulationAngels.append([-el,az])
145                     if el != 0.0 and az != 0.0:
146                         self.simulationAngels.append([-el,-az])
147                 else:
148                     break
149
150         self.numberSimulationAngels = len(self.simulationAngels)
151
152     # Load map
153     def loadMap(self):
154         error = False
155         self.numberMapObjects = len(self.parsedMap["objects"])
156         self.objects = []
157
```

```
158         try:
159             for i in range(self.numberMapObjects):
160                 name = self.parsedMap["objects"][i]["name"]
161                 positionStart = self.parsedMap["objects"][i]["positionStart"]
162                 vectorU = self.parsedMap["objects"][i]["vectorU"]
163                 vectorV = self.parsedMap["objects"][i]["vectorV"]
164                 reflectionFactor = self.parsedMap["objects"][i]["reflectionFactor"]
165 
166                 self.objects.append([name, positionStart, vectorU, vectorV, reflectionFactor])
167         except:
168             error = True
169 
170         return error
```

D.5. Positionsbestimmung

ultrasoundPositionMatcher.py

```
1   #!/usr/bin/env python3
2   #*********************************************************************#
3   #* University: HAW Hamburg                                           *#
4   #* Author: Christopher Rotzlawski                                    *#
5   #*                                                                   *#
6   #* Project: Surrounding Sensor-based Naviagtionssystem               *#
7   #* Version: 1.0                                                      *#
8   #*********************************************************************#
9   import rospy
10  import rospkg
11  import os
12  import json
13  import math
14  import numpy
15  from scipy import signal
16  from commonMsgUVPS.msg import checkUltrasoundSystem
17  from commonMsgUVPS.msg import responseCheckUltrasoundSystem
18  from commonMsgUVPS.msg import normalizedUltrasoundData
19  from commonMsgUVPS.msg import startUltrasoundSimulation
20  from commonMsgUVPS.msg import simulatedImpulseResponse
21  from commonMsgUVPS.msg import ultrasoundSystemPosition
22  from commonMsgUVPS.msg import validUltrasoundPosition
23
24  class ultrasoundPositionMatcher:
25      # Initialize node
26      def __init__(self):
27          self.PACKAGENAME = 'ultrasoundPositioningSystem'
28          self.ROSNODE = os.path.basename(os.path.dirname(os.path.realpath(__file__)))
29          self.PUBLISHCHECKRESPONSE = 'responseUltrasoundSytem'
30          self.PUBLISHSIMULATION = 'startUltrasoundSimulation'
31          self.PUBLISHPOSITION = 'ultrasoundSystemPosition'
32          self.LISTNERCHECK = 'checkUltrasoundSystem'
33          self.LISTNERIMPULSERESPONSE = 'simulatedImpulseResponse'
34          self.LISTNERULTRASOUNDDATA = 'ultrasoundNormalized'
35          self.LISTNERVALID = 'validUltrasoundPostion'
36          self.POSITIONSFILEPATH = '/config/positions.json'
37
38          rospack = rospkg.RosPack()
39          rospack.list()
40          self.positionsFilePath = rospack.get_path(self.PACKAGENAME) + self.POSITIONSFILEPATH
41
42          self.numberSensors = 0
43          self.sensorData = []
44          self.tempPosition = []
45          self.simulationPosition = []
46          self.impulseResponses = []
47          self.accuracyValues = []
48
49          self.position = None
50          self.positionValue = None
51          self.accuracy = None
52
53          self.systemReady = not self.readConfig()
54
55          self.pubCheckResponse = rospy.Publisher(self.PUBLISHCHECKRESPONSE, responseCheckUltrasoundSystem, queue_size=10)
56          self.pubStartSimulation = rospy.Publisher(self.PUBLISHSIMULATION, startUltrasoundSimulation, queue_size=10)
57          self.pubPosition = rospy.Publisher(self.PUBLISHPOSITION, ultrasoundSystemPosition, queue_size=10)
58
59          self.checkResponseMessage = responseCheckUltrasoundSystem()
```

```
60          self.startSimulationMessage = startUltrasoundSimulation()
61          self.positionMessage = ultrasoundSystemPosition()
62
63          rospy.init_node(self.ROSNODE)
64
65      # Publish position
66      def publishPosition(self):
67          self.positionMessage.latitude = self.position[1]
68          self.positionMessage.longitude = self.position[0]
69          self.positionMessage.direction = self.position[4]
70          self.positionMessage.accuracy = self.accuracy
71
72          self.pubPosition.publish(self.positionMessage)
73
74      # Set simulation request
75      def startSimulation(self):
76          self.startSimulationMessage.latitude = self.tempPosition[0][1]
77          self.startSimulationMessage.longitude = self.tempPosition[0][0]
78          self.startSimulationMessage.direction = self.tempPosition[0][4]
79          self.startSimulationMessage.temperature = 20.0
80          self.startSimulationMessage.transmitterSensor = self.tempPosition[0][5]
81
82          self.pubStartSimulation.publish(self.startSimulationMessage)
83
84          # Set simulation position
85          if self.tempPosition[0][5] == 0:
86              self.simulationPosition = [self.tempPosition[0][0], self.tempPosition[0][1], self.tempPosition[0][2]
87                                         ,self.tempPosition[0][3], self.tempPosition[0][4]]
88
89          self.tempPosition.pop(0)
90
91      # Start subscriber
92      def listner(self):
93          rospy.Subscriber(self.LISTNERCHECK, checkUltrasoundSystem, self.listnerCheckSytem)
94          rospy.Subscriber(self.LISTNERULTRASOUNDDATA, normalizedUltrasoundData, self.listnerUltrasoundData)
95          rospy.Subscriber(self.LISTNERIMPULSERESPONSE, simulatedImpulseResponse, self.listnerSimulation)
96          rospy.Subscriber(self.LISTNERVALID, validUltrasoundPosition, self.listnerValidPosition)
97
98          rospy.spin()
99
100     # Listner for check system
101     def listnerCheckSytem(self,data):
102         if data.system == 'ultrasoundMatcher':
103             self.checkResponseMessage.system = 'ultrasoundMatcher'
104             self.checkResponseMessage.ready = self.systemReady
105
106             self.pubCheckResponse.publish(self.checkResponseMessage)
107
108     # Listner to start position matching
109     def listnerUltrasoundData(self,ultrasoundData):
110         self.splitUltrasoundData(ultrasoundData)
111
112         #self.position = None
113         self.positionValue = None
114         self.accuracy = None
115
116         # If all ultrasound data are received, start position estimation
117         if len(self.sensorData) == self.numberSensors:
118             #self.iterationNumber = 1
119             self.tempTemp = [None,None,None,None,None,None,None,None,None,None]
120             self.positionEstimation()
121
122     # Listner to get simulation data
123     def listnerSimulation(self,simulationData):
124         self.sortImpulseResponses(simulationData)
125
126         # If all simulation data are received, correlate data
127         if len(self.impulseResponses) == len(self.sensorData):
128             positionValue,accuracy = self.correlateData()
129
130             tempValue = 100000000
131             tempPos = -1
132             for q in range(len(self.tempTemp)):
133                 if self.tempTemp[q] == None:
134                     tempPos = q
135                     break
136                 elif self.tempTemp[q][1] < positionValue and self.tempTemp[q][1] < tempValue:
137                     tempPos = q
138                     tempValue = positionValue
139
140             if tempPos > -1:
141                 self.tempTemp[tempPos] = [self.simulationPosition,positionValue,accuracy]
142
143             # Save best position
144             if not self.positionValue or self.positionValue < positionValue:
```

```
145                 self.position = self.simulationPosition
146                 self.positionValue = positionValue
147                 self.accuracy = accuracy
148 
149             # If simulation positions left, start next simulation
150             if len(self.tempPosition) > 0:
151                 self.startSimulation()
152             else:
153                 self.publishPosition()
154         else:
155             self.startSimulation()
156 
157     # Listner to set valid position
158     def listnerValidPosition(self,data):
159         self.position[0] = data.longitudeValid
160         self.position[1] = data.latitudeValid
161         self.position[4] = data.directionValid
162 
163     # Read config file
164     def readConfig(self):
165         error = False
166 
167         # Check if file exists
168         if type(self.positionsFilePath) is not str:
169             error = True
170             print("Wrong config file path.")
171         else:
172             error = not os.path.isfile(self.positionsFilePath)
173             if error:
174                 print("Positions file doesn't exist.")
175 
176         # Parse json file
177         if not error:
178             with open(self.positionsFilePath,'r') as positionsFile:
179                 parsedFile = json.load(positionsFile)
180         else:
181             print("Read file failed.")
182 
183         # Get positions
184         if not error:
185             try:
186                 self.getPositions(parsedFile)
187             except:
188                 error = True
189                 print("Get positions failed.")
190 
191         return error
192 
193     # Get position parameter
194     def getPositions(self,parsedFile):
195         self.numberPositions = len(parsedFile["positions"])
196         self.seedPoint = []
197         self.positions = []
198 
199         for i in range(self.numberPositions):
200             name = parsedFile["positions"][i]["name"]
201             centerX = parsedFile["positions"][i]["centerX[m]"]
202             centerY = parsedFile["positions"][i]["centerY[m]"]
203             lengthX = parsedFile["positions"][i]["lengthX[m]"]
204             lengthY = parsedFile["positions"][i]["lengthY[m]"]
205 
206             if i == 0:
207                 direction = parsedFile["positions"][i]["direction"]
208                 self.seedPoint = [centerX,centerY,lengthX,lengthY,direction]
209                 self.position = self.seedPoint
210             else:
211                 self.positions.append([name,centerX,centerY,lengthX,lengthY])
212 
213     # Split ultrasound data
214     def splitUltrasoundData(self,ultrasoundData):
215         self.numberSensors = ultrasoundData.numberSensors
216 
217         self.dataLength = int(ultrasoundData.dataLen/self.numberSensors)
218 
219         if len(self.sensorData) >= self.numberSensors:
220             self.sensorData = []
221         self.sensorData.append(ultrasoundData.normalizedData)
222 
223     # Estimate position
224     def positionEstimation(self):
225         #-----------------------------------------------------------------
226         # Globale position
227         """posXTemp = self.positions[0][1]
228         posYTemp = self.positions[0][2]
229         lenX = self.positions[0][3]/4.0
```

```
230             lenY = self.positions[0][4]/4.0
231
232             for deltaX in range(-3,4):
233                 posX = posXTemp+deltaX*lenX
234
235                 for deltaY in range(-3,4):
236                     posY = posYTemp+deltaY*lenY
237
238                     for deltaDir in range(8):
239                         direction = deltaDir*360.0/8.0
240
241                         for sensor in range(self.numberSensors):
242                             self.tempPosition.append([posX,posY,lenX,lenY,direction,sensor])
243
244             self.startSimulation()"""
245
246             #-----------------------------------------------------------
247             # Predicted area
248             if not self.position:
249                 self.position = self.seedPoint
250
251             for sensor in range(self.numberSensors):
252                 self.tempPosition.append([self.position[0],self.position[1],self.position[2],self.position[3],self.position[4],sensor])
253
254             for deltaX in range(-1,2):
255                 posX = self.position[0]+deltaX*self.position[2]
256
257                 for deltaY in range(-1,2):
258                     posY = self.position[1]+deltaY*self.position[3]
259
260                     delta = [posX-self.position[0],posY-self.position[1]]
261                     angle = (math.atan2(delta[1],delta[0])/math.pi*180.0)%360.0
262
263                     for deltaDir in range(-2,3):
264                         direction = (self.position[4]+deltaDir*45.0)%360.0
265
266                         angleDiffOld = (angle-self.position[4])%180
267                         if angleDiffOld > 90:
268                             angleDiffOld -= 180
269
270                         angleDiffNew = (self.position[4]+angleDiffOld-direction)%360
271                         if angleDiffNew > 180:
272                             angleDiffNew -= 360
273
274                         if deltaX == 0 and deltaY == 0:
275                             samePos = True
276                         else:
277                             samePos = False
278
279                         if angleDiffOld >= -65.0 and angleDiffOld <= 65.0:
280                             if angleDiffNew >= -65.0 and angleDiffNew <= 65.0 and not samePos:
281
282                                 for sensor in range(self.numberSensors):
283                                     self.tempPosition.append([posX,posY,self.position[2],self.position[3],direction,sensor])
284
285             self.startSimulation()
286
287         # Sort simulation data
288         def sortImpulseResponses(self,simulationData):
289             self.tempResponses = [[0.0]*self.dataLength for i in range(self.numberSensors)]
290             for i in range(0,simulationData.length,3):
291                 sensorIndex = int(simulationData.impulseResponse[i])
292                 timeIndex = int(simulationData.impulseResponse[i+1]/0.1*self.dataLength+0.5)
293                 if simulationData.impulseResponse[i+2] > self.tempResponses[sensorIndex][timeIndex]:
294                     self.tempResponses[sensorIndex][timeIndex] = simulationData.impulseResponse[i+2]
295
296             for i in range(self.numberSensors):
297                 self.tempResponses[i][0] = 1.0
298
299             self.envelopeSimulationData()
300
301             if len(self.impulseResponses) >= self.numberSensors:
302                 self.impulseResponses = []
303
304             self.impulseResponses.append([])
305             for i in range(self.numberSensors):
306                 self.impulseResponses[len(self.impulseResponses)-1] += self.envelopedResponses[i]
307
308         # Envelope simulation data
309         def envelopeSimulationData(self):
310             self.envelopedResponses = [[0.0]*self.dataLength for i in range(self.numberSensors)]
311             for i in range(self.numberSensors):
312                 for j in range(self.dataLength-60):
313                     if self.tempResponses[i][j] != 0.0:
314                         for k in range(-60,61):
```

```
315                         value = self.tempResponses[i][j]*math.cos(2*math.pi/240*k)
316                         if value > self.envelopedResponses[i][j+k] and j+k >= 0:
317                             self.envelopedResponses[i][j+k] = value
318
319     # Correlate data
320     def correlateData(self):
321         correlateResult = []
322         maxValue = [0.0]*self.numberSensors
323         pos = [0]*self.numberSensors
324         for i in range(self.numberSensors):
325             correlateResult.append(numpy.correlate(self.sensorData[i][:], self.impulseResponses[i][:], 'same'))
326
327             for j in range(len(correlateResult[i])):
328                 if correlateResult[i][j] > maxValue[i]:
329                     pos[i] = j
330                     maxValue[i] = correlateResult[i][j]
331
332         positionValue = 0.0
333         accuracy = 0.0
334
335         if len(self.accuracyValues) < 1:
336             self.accuracyValues = signal.gaussian(len(correlateResult[0]), std=len(correlateResult[0])/7)
337
338         for i in range(self.numberSensors):
339             accuracyValue = self.accuracyValues[pos[i]]
340             positionValue += maxValue[i]*accuracyValue
341             accuracy += accuracyValue
342         positionValue /= self.numberSensors
343         accuracy /= self.numberSensors
344
345         return positionValue,accuracy
346
347 if __name__ == '__main__':
348     try:
349         node = ultrasoundPositionMatcher()
350         node.listner()
351     except rospy.ROSInterruptException:
352         print('ERROR: start ultrasound position matcher failed!')
```

E. Quellcode des umfeldsensorbasierten Navigationssystems

universalVehiclePositioningSystemNode.py

```
#!/usr/bin/env python3
#*********************************************************************#
#* University: HAW Hamburg                                           *#
#* Author: Christopher Rotzlawski                                    *#
#*                                                                   *#
#* Project: Surrounding Sensor-based Naviagtionssystem               *#
#* Version: 1.0                                                      *#
#*********************************************************************#
import rospy
import os
from commonMsgUVPS.msg import ultrasoundPositionRequest
from commonMsgUVPS.msg import responseUltrasoundPosition
from sbg_ros_driver.msg import SbgGpsRaw
from sbg_ros_driver.msg import SbgImuData

class unversalVehiclePositioningSystem:
    # Initialize node
    def __init__(self):
        self.ROSNODE = os.path.basename(os.path.dirname(os.path.realpath(__file__)))
        self.PUBLISHPOSITIONREQUEST = 'ultrasoundPositionRequest'
        self.LISTNERPOSITION = 'responseUltrasoundPostion'

        self.pub = rospy.Publisher(self.PUBLISHPOSITIONREQUEST, ultrasoundPositionRequest, queue_size=10)
        rospy.init_node(self.ROSNODE)

        rospy.Subscriber(self.LISTNERPOSITION, responseUltrasoundPosition, self.listnerPosition)
        self.message = ultrasoundPositionRequest()

        # Position request
        self.publishPositionRequest()

        rospy.spin()

    # Publish position request
    def publishPositionRequest(self):
        newPos = input('New valid position? (Y/N)')
        if newPos == 'Y' or newPos == 'y':
            self.message.getPosition = False
            self.message.validPosition = True
            self.message.latitudeOld = float(input('X: '))
            self.message.longitudeOld = float(input('Y: '))
            self.message.directionOld = float(input('Dir: '))

        else:
            input('Position request...')

            self.message.getPosition = True
            self.message.validPosition = False
            self.message.latitudeOld = 0.0
            self.message.longitudeOld = 0.0
            self.message.directionOld = 0.0

        self.pub.publish(self.message)

    # Listner for ultrasound position
    def listnerPosition(self,data):
        print('------ Position ------')
        print('X: '+str(data.longitude))
        print('Y: '+str(data.latitude))
        print('Direction: '+str(data.direction))
        print('Accuracy: '+str(data.accuracy))

        self.publishPositionRequest()

```

```
65  if __name__ == '__main__':
66      try:
67          node = unversalVehiclePositioningSystem()
68      except rospy.ROSInterruptException:
69          pass
```

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit nach § 22 (6) PStO 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, 19. März 2020	
Ort, Datum	Unterschrift