Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Christopher Rotzlawski

Konzeption und Umsetzung eines Entwicklungssystems für intelligente Ultraschallsensoren in Mehrfachanordnung zur Lokalisation und Umgebungserkennung in komplexen Umgebungen

Fakultät Technik und Informatik
Department Fahrzeugtechnik und Flugzeugbau

Faculty of Engineering and Computer Science
Department of Automotive and Aeronautical Engineering

Christopher Rotzlawski

Konzeption und Umsetzung eines Entwicklungssystems für intelligente Ultraschallsensoren in Mehrfachanordnung zur Lokalisation und Umgebungserkennung in komplexen Umgebungen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Mechatronik
am Department Fahrzeugtechnik und Flugzeugbau
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Erstprüfer : Prof. Dr. rer. nat. Rasmus Rettig
Zweitprüfer : Prof. Dr.-Ing. Jochen Maaß

Abgabedatum: 27.05.2018

Zusammenfassung

Christopher Rotzlawski

Thema der Bachelorarbeit

Konzeption und Umsetzung eines Entwicklungssystems für intelligente Ultraschallsensoren in Mehrfachanordnung zur Lokalisation und Umgebungserkennung in komplexen Umgebungen

Stichworte

Ultraschallsensor, Mikrocontroller, Ultraschall-Lautsprecher, MEMS-Mikrofon, Temperatursensor, Drucksensor, Ethernet-Netzwerk, Ultraschallechodatenauswertung, Objekterkennung, Abstandsmessung, Mehrfachanordnung, Signalverarbeitung

Kurzzusammenfassung

In dieser Arbeit wird ein Entwicklungssystem für intelligente Ultraschallsensoren realisiert. Das System ermöglicht eine Entwicklung von Algorithmen und Methoden zur Lokalisation und Umgebungserkennung in komplexen Umgebungen. Dazu kann eine Datenauswertung der Ultraschallechos im Anschluss an einen Messdurchlauf vorgenommen werden. Für eine spätere Datenauswertung der Messungen werden die Ultraschallechos gespeichert.

Christopher Rotzlawski

Title of the paper

Conception and implementation of a development system for intelligent ultrasonic sensors in multiple configuration for localization and environment detection in complex environments

Keywords

Ultrasonic sensor, Microcontroller, Ultrasonic-loudspeaker, MEMS-microphone, Temperature sensor, Pressure sensor, Ethernet-network, Ultrasonic echo data analysis, Object detection, Distance measurement, Multiple configuration, signal processing

Abstract

In this thesis, a development system for intelligent ultrasonic sensors is implemented. The system enables the development of algorithms and methods for localization and environment detection in complex environments. For this purpose, a data analysis of the ultrasonic echoes can be made following a measurement run. For a later data analysis of the measurement, the ultrasonic echoes are stored.

Inhaltsverzeichnis

1. Einführung und Motivation

Fahrerassistenzsysteme (FAS) stellen dem Fahrer Informations-, Assistenz- und Sicherheitsfunktionen zur Verfügung. Dies geschieht mithilfe intelligenter Sensorik. Das Ziel dabei ist ein lernendes, autonomes Fahrzeug der Stufe 5 (vgl. BASt, 2012), welches seine Umgebung wahrnehmen und interpretieren kann. Ein Beispiel für intelligente Fahrzeugsensorik ist der Ultraschallsensor. Dabei sind Ultraschallsensoren in FAS für den Ultranahbereich bis 5 m zuständig. Die Verwendung reicht dabei von einfachen Abstandswarnungen bis zum Vermessen von Parklücken und dem anschließenden semi- beziehungsweise vollautomatischen Einparken (vgl. Reif, 2010, S. 109). Ein weiterer Einsatzbereich von Ultraschallsensoren in Fahrzeugen ist die Überwachung des toten Winkels. Hierbei werden Fahrzeuge in diesem Bereich erkannt und dem Fahrer signalisiert. Des Weiteren kann dies bei einem automatischen Spurwechsel genutzt werden (vgl. Ingle und Phute, 2016, S. 370).

Ein Ziel der Entwicklung im *Urban Mobility Lab* ist dabei die Lokalisation von Objekten und Umgebungserkennung in komplexen Umgebungen. Die Anforderungen an das zu entwickelnde System sind dabei eine flexible Anzahl an Ultraschall-Sendern und -Empfängern im System. Des Weiteren soll eine Zeitanalyse der empfangenen Ultraschallechosignale möglich sein. Für den Betrieb in komplexen Umgebungen sollten Vielfachechos aufgezeichnet werden können. Eine Erkennung von Objekten, welche optisch noch nicht erfassbar sind, sollte möglich sein. Eine weitere Anforderung an das System ist eine Erhöhung der Pulszykluszeiten. Des Weiteren sollen mit einem solchen System die Auswirkungen von Sendefrequenz und Signalformen auf die Messergebnisse analysierbar sein.

Ziel dieser Arbeit ist der Entwurf und die Umsetzung eines flexiblen und skalierbaren Entwicklungssystems für intelligente Ultraschallsensoren bestehend aus Hardware und Software. Die Hardware des Systems, bestehend aus Ultraschallsensoren, soll einstellbare Sendefrequenzen und Signalformen ermöglichen. Ein weiteres Ziel ist die Erfassung und Auswertung von Vielfachechos in komplexen Umgebungen, mit der verdeckte Objekte erfasst werden können. Des Weiteren soll eine Verwendung der Ergebnisse aus Vorarbeiten analysiert werden.

Die Vorgehensweise in dieser Arbeit orientiert sich dabei an dem V-Modell (vgl. Vogel Heuser, 2003, S. 26). Hierbei werden zunächst in einer Analyse die Anforderungen an das Entwicklungssystem analysiert. Dies geschieht auf Basis der Nutzeranforderungen und eines

Anwendungsfalls. Das anschließende Konzept befasst sich zunächst mit der Systemarchitektur für das Entwicklungssystem, auf deren Basis die Konzeption des Ultraschallsensors erfolgt. In der Umsetzung wird zunächst das Entwicklungssystem bestehend aus Ultraschallsensoren und Datenaufnahme umgesetzt. Aufbauend darauf erfolgt die Umsetzung des Ultraschallsensors. Abschließend wird die Datenaufnahme implementiert. Die Tests befassen sich zunächst mit der Überprüfung der einzelnen Komponenten. Anschließend erfolgt der Test des Entwicklungssystems auf die Funktionalität. Den Abschluss der Arbeit bildet der Test des Entwicklungssystems anhand eines Anwendungsfalls.

2. Grundlagen

Das Kapitel Grundlagen beschäftigt sich mit gängigen Grundbegriffen, die in Zusammenhang mit dieser Arbeit stehen. Hierzu wird zunächst auf das Thema Ultraschall eingegangen. Weiterführend werden Ultraschallsensoren erklärt. Abschließend wird der Stand der Technik, welcher die Ergebnisse aus Vorarbeiten beinhaltet, beschrieben.

2.1. Ultraschall

Als Schall bezeichnet man mechanische Wellen, die sich in Gasen oder anderen Medien ausbreiten. Dabei wird zwischen Infraschall, Hörschall, Ultraschall und Hyperschall unterschieden. Bei Infraschall liegt die Frequenz unter 16 Hz. Daran anschließend liegt der Bereich des Hörschalls mit einer Frequenz von 16 Hz bis 20 kHz. Frequenzen über 20 kHz, die außerhalb des menschlichen Wahrnehmungsbereichs liegen, bezeichnet man als Ultraschall beziehungsweise ab einer Frequenz von über 1 GHz als Hyperschall (vgl. Eichler, 2014, S. 173).

Ultraschall breitet sich dabei in Gasen und Flüssigkeiten als longitudinale Druckwelle aus, wobei der Druck p periodisch um den normalen Druck p_0 schwankt. Der Druck lässt sich dabei aus dem normalen Druck, der Amplitude der Schallwelle $\hat{p}$, der Frequenz f, der Ausbreitung in x-Richtung und der Wellenlänge λ berechnen (vgl. Eichler, 2014, S. 173):

$$p = p_0 + \hat{p} sin\left(2\pi\left(ft - \frac{x}{\lambda}\right)\right) \tag{2.1}$$

Der effektive Schalldruck p_{eff} lässt sich mit der Amplitude der Schallwelle berechnen (vgl. Eichler, 2014, S. 173):

$$p_{eff} = \frac{\hat{p}}{\sqrt{2}} \tag{2.2}$$

Die Berechnung der Schallgeschwindigkeit c in Gasen und Flüssigkeiten erfolgt mit dem Kompressionsmodul K und der Dichte ρ (vgl. Eichler, 2014, S. 174):

$$c = \sqrt{\frac{K}{\rho}} \tag{2.3}$$

Wobei das Kompressionsmodul mithilfe des Adiabatenexponenten κ bestimmt wird (vgl. Eichler, 2014, S. 174):

$$K = \kappa p \tag{2.4}$$

Geht man für die vereinfachte Berechnung von idealem Gas aus, lässt sich der Druck mit der Dichte, der speziellen Gaskonstanten R' und der Temperatur T berechnen (vgl. Eichler, 2014, S. 175):

$$p = \rho R' T \tag{2.5}$$

Setzt man die Formel 2.5 in die Formel 2.3 ein, resultiert für die Schallgeschwindigkeit (vgl. Eichler, 2014, S. 175):

$$c = \sqrt{\kappa R' T} \tag{2.6}$$

Dabei beträgt die Schallgeschwindigkeit bei einer Temperatur von 0 °C in Luft zum Beispiel 331,4 $\frac{m}{s}$. Da die Schallgeschwindigkeit von der Temperatur abhängig ist, muss man diese berücksichtigen. Dies kann näherungsweise mit folgender Formel berechnet werden, wobei die Temperatur θ in °C angegeben wird (vgl. Eichler, 2014, S. 175):

$$c_{Luft} = \left(331,4 + 0,6\frac{\theta}{^\circ C}\right)\frac{m}{s} \tag{2.7}$$

Ebenfalls ist die Schallgeschwindigkeit von der relativen Luftfeuchte abhängig. Bei einer Änderung der relativen Luftfeuchte von 0 % auf 100 % erhöht sich gleichzeitig die Schallgeschwindigkeit um ca. 0,2 $\frac{m}{s}$. Da diese Abhängigkeit sehr gering ausfällt, ist eine Kompensation für genaue Messergebnisse nicht erforderlich (vgl. Hering und Schönfelder, 2012, S. 277).

Eine weitere wichtige Messgröße für Schall ist der Schallpegel L_P. Der Schallpegel beschreibt die Schallimmission an einem bestimmten Ort. Der Schallpegel kann mithilfe des effektiven Schalldrucks berechnet werden (vgl. Eichler, 2014, S. 178):

$$L_P = 20log\left(\frac{p_{eff}}{p_{eff,0}}\right)dB \tag{2.8}$$

Mit dem Bezugswert $p_{eff,0}$ (vgl. Eichler, 2014, S. 178):

$$p_{eff,0} = 2 * 10^{-5} Pa \tag{2.9}$$

Ein weiterer wichtiger Punkt bei Schall ist die Absorption. Hierbei nimmt die Intensität I der Schallwelle mit zunehmendem Weg x ab. Die Dämpfung der Schallwelle ist dabei von der Frequenz abhängig. Dabei gilt, je höher die Frequenz ist, umso niedriger ist die Reichweite der Schallwelle. Die Intensität wird mit der Frequenz und dem Proportionalitätsfaktor a

berechnet (vgl. Hering und Schönfelder, 2012, S. 110):

$$I = I_0 e^{-af^2x} \tag{2.10}$$

Mit der Intensität I_0 an der Schallquelle (vgl. Eichler, 2014, S. 177):

$$I_0 = 10^{-12} \frac{W}{m^2} \tag{2.11}$$

Ebenso für die Dämpfung der Schallwellen ist der Luftdruck entscheidend. So nimmt mit höherem Luftdruck die Absorption ab und die Reichweite zu (vgl. Hering und Schönfelder, 2012, S. 111).

Trifft eine Schallwelle in Luft auf ein anderes Medium, wird ein Teil in das Medium transmittiert und ein anderer Teil wird von der Grenzfläche reflektiert (vgl. Hering und Schönfelder, 2012, S. 111).

Die Verwendung von Ultraschall reicht von einfachen Distanzmessungen in Luft über komplexe Signalverarbeitungsprozesse auf Basis von Festkörperwellen bis hin zu medizinischen Diagnostikverfahren. Dabei geht aufgrund hoher Dämpfung und sehr kleiner Wellenlängen der technisch nutzbare Bereich selten über 1 GHz hinaus (vgl. Lerch u. a., 2009, S. 573).

Die Frequenz stellt bei technischen Anwendungen einen Kompromiss zwischen Ortsauflösung und Dämpfung dar. Bei höheren Frequenzen ist die Ortsauflösung durch die kürzeren Wellenlängen höher, gleichzeitig steigt aber auch die Dämpfung, wodurch die Reichweite sinkt (vgl. Lerch u. a., 2009, S. 575).

Ein Nachteil von Ultraschall gegenüber optischen Abstandssensoren ist die Richtcharakteristik beim Empfang von Schallwellen (siehe Abbildung 2.1). Dabei kann der ausgesendete Schall von einem Objekt so gebündelt und gerichtet reflektiert werden, dass dieser nicht zum Ultraschallsensor zurückgelangt (vgl. Lerch u. a., 2009, S. 575).

Abbildung 2.1.: Gebündelte und gerichtete Reflexion von Ultraschall an einem Objekt

Ein weiterer Nachteil von Ultraschallsensoren gegenüber optischen Abstandssensoren ist der Doppler-Effekt. Hierbei wird die Frequenz der Ultraschallwellen verändert, wenn diese, von einem sich bewegenden Objekt, reflektiert werden (vgl. Lerch u. a., 2009, S. 583). Hierdurch kann es dazu kommen, dass die Frequenz der reflektierten Ultraschallwellen außerhalb der Bandbreite des Ultraschallsensors liegt und diese nicht als Ultraschallechos erkannt werden. Formel 2.12 (vgl. Lerch u. a., 2009, S. 583) zeigt die empfangene Frequenz bei einer Sendefrequenz f_S = 40 kHz und einem Objekt, welches sich mit einer Geschwindigkeit von ν_0 = 100 $\frac{km}{h}$ auf den Ultraschallsensor zubewegt. Als Temperatur wird dabei 20 °C angenommen.

$$f_E = f_S \left(1 + \frac{\nu_0}{c}\right) = 43,24\ kHz \tag{2.12}$$

2.2. Ultraschallsensoren

Ultraschallsensoren dienen zur Abstands- und Geschwindigkeitsmessung. Hierbei werden Laufzeitmessungen, sogenannte Time of Flight (TOF) Messungen, zur Ermittlung des Abstands durchgeführt. Beziehungsweise wird zur Geschwindigkeitsmessung die Dopplerverschiebung des reflektierten Ultraschallechos gemessen. Der bei Ultraschallsensoren verwendete typische Frequenzbereich liegt zwischen 25 und 500 kHz. Ein Vorteil von Ultraschallsensoren ist, dass diese auch in optisch nicht transparenten Umgebungen verwendet werden können (vgl. Lerch u. a., 2009, S. 581).

Des Weiteren handelt es sich bei Ultraschallsensoren, im Vergleich zu anderen Abstandssensoren, um kostengünstige und leichte Sensoren, die einen geringen Energieverbrauch benötigen. In manchen Anwendungsfällen, zum Beispiel Unterwasser oder bei geringer Sicht, sind Ultraschallsensoren die einzig realisierbaren Abstandssensoren (vgl. Siciliano und Khatib, 2016, S. 754).

Weitere Ultraschallanwendungen sind Messverfahren in flüssigen Medien beziehungsweise biologischem Gewebe, zum Beispiel bildgebende medizinische Diagnostik. Hierbei werden mithilfe der Ultraschallwellen Ortsinformationen gewonnen. Die verwendeten Frequenzen liegen bei der medizinischen Diagnostik im Bereich von 2,5 MHz bis 15 MHz. Die Ausbreitungsgeschwindigkeit der Ultraschallwellen liegt hierbei mit ca. 1500 $\frac{m}{s}$ deutlich höher als in Gasen (vgl. Lerch u. a., 2009, S. 587). So sind in der medizinischen Diagnostik 3D-bildgebende-Verfahren möglich, die mit Hilfe von 2D-Matrix-Ultraschallwandlern, zum Beispiel Capacitive Micromachined Ultrasound Transducers (CMUT), realisiert werden (vgl. Lerch u. a., 2009, S. 604).

Des Weiteren wird Ultraschall für Messverfahren in Festkörper Materialien eingesetzt, wie die zerstörungsfreie Werkstoffprüfung. Hierbei sollen Fehler im Materialinneren erkannt und sichtbar gemacht werden (vgl. Lerch u. a., 2009, S. 660).

2.2.1. Ultraschallwandler

Das Hauptbauteil eines Ultraschallsensors ist der Ultraschallwandler. Dieser sendet Ultraschallwellen aus und empfängt das zurückkommende Echo. Dabei wird dieser mit einem Leistungsverstärker periodisch angesteuert. Durch das Anlegen einer elektrischen Wechselspannung an den Wandler wird dieser zum Schwingen angeregt, wodurch die Ultraschallwellen entstehen (vgl. Hering und Schönfelder, 2012, S. 178).

Die zwei gebräuchlichsten Wandler für Anwendungen in der Umgebungsluft sind dabei piezoelektrische und elektrostatische Wandler. Vorteil von elektrostatischen Wandlern ist die hohe Bandbreite von zum Beispiel 20 kHz (vgl. SensComp, 2013, S. 2). Der Nachteil von elektrostatischen Wandlern ist, dass diese eine hohe Offsetspannung von über 100 V benötigen. Piezoelektrische Wandler haben als Vorteil, dass diese bei niedrigen Wechselspannungen unter 30 V arbeiten. Als Nachteil haben piezoelektrische Wandler eine geringe Bandbreite von maximal 3 kHz (vgl. ProWave-Electronic, 2005, S. 1). Zudem sind diese als Sender und Empfänger geeignet (vgl. Siciliano und Khatib, 2016, S. 759). Eine weitere Variante sind mikroelektromechanische Systeme (MEMS), welche auf einem Siliziumchip realisiert werden. Dabei arbeiten diese als elektrostatisch-kapazitive Wandler. Die Frequenz bei MEMS reicht hin bis zu mehreren MHz (vgl. Siciliano und Khatib, 2016, S. 760).

2.2.2. Abstandsmessung

Die Abstandsmessung startet mit dem Aussenden eines kurzen Schallwellenpaketes. Dabei breiten sich die Ultraschallwellen in der umgebenden Luft aus. Die Zeit, die der Ultraschallwandler zum Ausschwingen benötigt, bestimmt die Blindzone, in der keine Objekte erkannt werden können, da in dieser Zeit keine reflektierten Schallwellen erkannt werden. Nachdem die Schallwellen auf ein Objekt treffen und von diesem reflektiert werden, erreichen diese den Sensor und versetzen diesen in Schwingung. Überschreitet dabei die Amplitude der elektrischen Wechselspannung einen festgelegten Schwellwert, wird das Signal als gültiges Ultraschallecho gewertet. Mit der hieraus ermittelten Laufzeit der Ultraschallwellen kann der Abstand zum Objekt bestimmt werden. Um die Distanz d zwischen Ultraschallsensor und Objekt zu ermitteln, benötigt man die Schallgeschwindigkeit und die Laufzeit t (vgl. Reif, 2014, S. 325):

$$d = \frac{c_{Luft} t}{2} \tag{2.13}$$

Ein typischer Signalverlauf eines Ultraschallsensors wird in Abbildung 2.2 gezeigt.

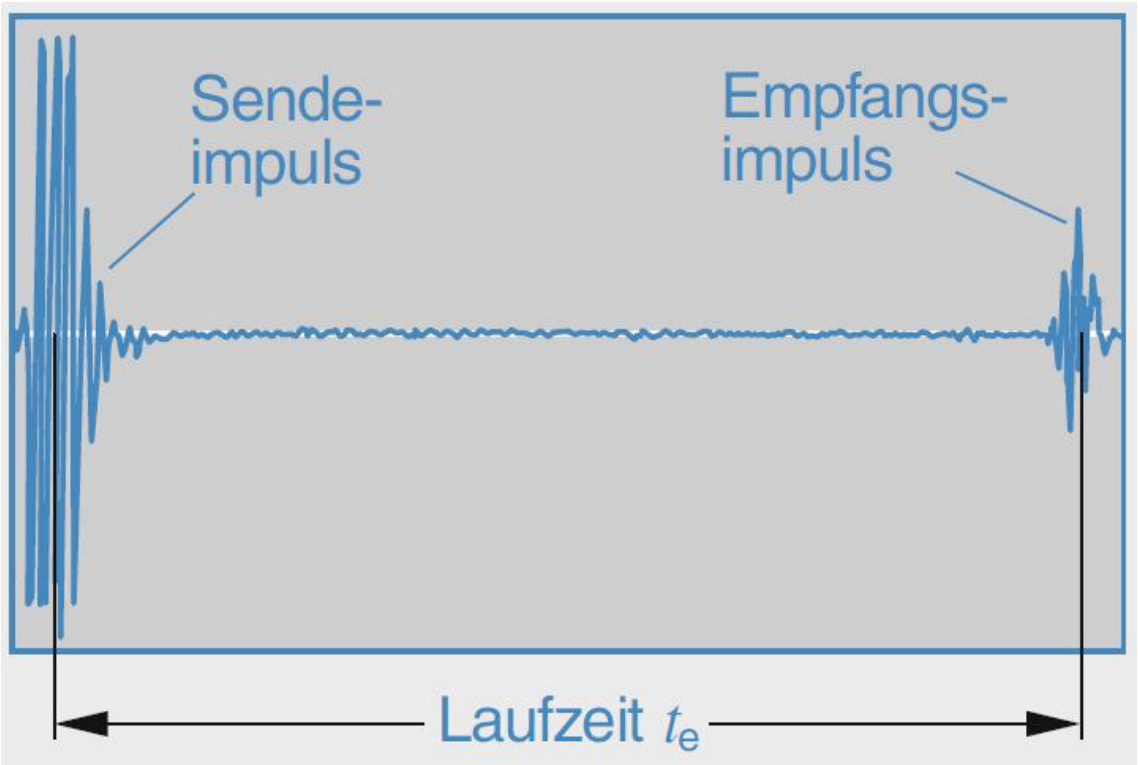


Abbildung 2.2.: Typischer Messzyklus eines Ultraschallsensors (Reif, 2010, S. 131)

Messbereich

Der Messbereich von Ultraschallsensoren wird von mehreren Faktoren wie Sendefrequenz, Schallamplitude und Messempfindlichkeit des Ultraschallsensors bestimmt. Mit höherer Sendefrequenz sinkt die maximale Reichweite. Gleichzeitig sinkt der Bereich der Blindzone ebenfalls. Der Vorteil von höheren Sendefrequenzen ist, dass diese kürzere Zykluszeiten erlauben (vgl. Hering und Schönfelder, 2012, S. 180).

Die Schallkeulen der ausgesendeten Ultraschallwellen haben typischerweise einen Öffnungswinkel von 5° bis 8° (siehe Abbildung 2.3). Die Grenze dieser Schallkeule entspricht einer Dämpfung von 3 dB, das heißt, dass der Schalldruck an dieser Stelle auf die Hälfte des ursprünglichen Wertes zurückfällt. Objekte, die die entsprechende Größe und Form besitzen, können auch außerhalb dieser Grenze erkannt werden (vgl. Hering und Schönfelder, 2012, S. 179).

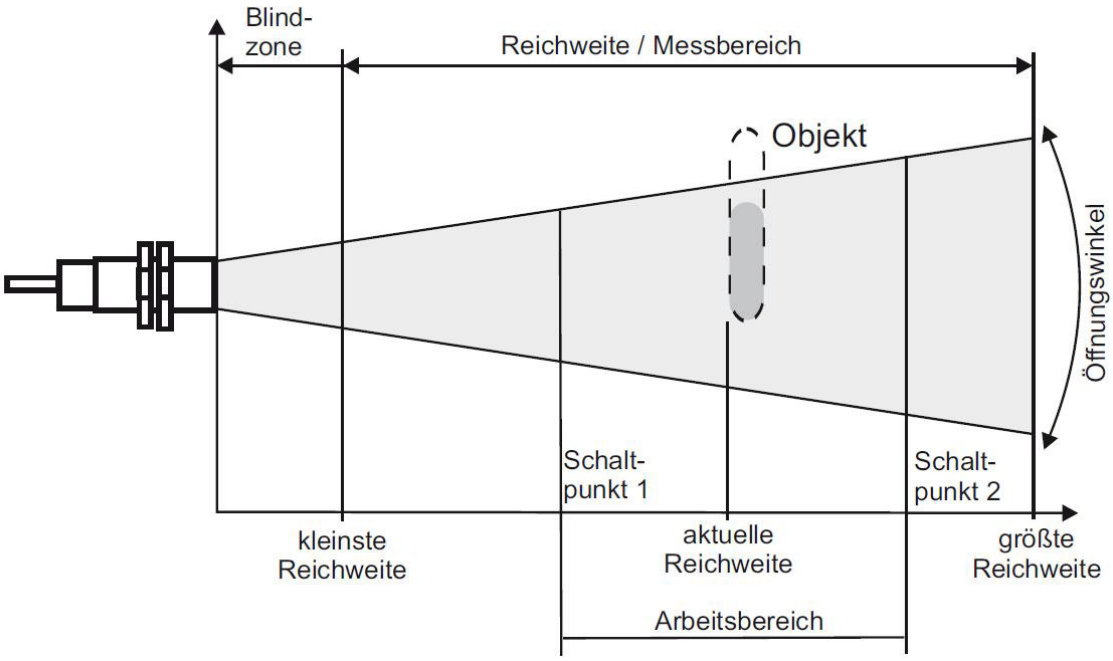


Abbildung 2.3.: Messbreich und Reichweite eines Ultraschallsensors (Hering und Schönfelder, 2012, S. 180)

Einflüsse auf Messergebnisse und Reichweite

Die Reichweite und Messergebnisse eines Ultraschallsensors können teilweise sehr stark von Objekten im Messbereich oder der Umwelt beeinflusst werden (siehe Tabelle 2.1).

Objekteinfluss	**Umwelteinfluss**
Form	Luftbewegungen
Oberflächenbeschaffenheit	
Materialhärte	
Objekttemperatur	

Tabelle 2.1.: Übersicht über Einflüsse auf Messergebnisse und Reichweite eines Ultraschallsensors

Zu den Eigenschaften von Objekten beziehungsweise deren Zuständen, die die Messergebnisse beeinflussen können, zählt zum Beispiel die Form. Bei konvexen Formen werden die Ultraschallwellen in unterschiedliche Richtungen reflektiert, was dazu führt, dass nur ein sehr kleiner Teil der Ultraschallwellen den Ultraschallsensor erreicht (siehe Abbildung 2.4.a). Dieser Effekt nimmt mit kleiner werdenden Radius zu (vgl. Hering und Schönfelder, 2012, S. 181).

Eine weitere wichtige Objekteigenschaft ist die Oberflächenbeschaffenheit. Bei rauen Oberflächen können die Ultraschallwellen diffus reflektiert werden. Dies führt dazu, wie in Abbildung 2.4.b zu sehen, dass eine Erkennung des entsprechenden Objekts erschwert werden kann, da nicht alle Schallwellen zum Ultraschallsensor reflektiert werden (vgl. Hering und Schönfelder, 2012, S. 182).

Die Materialhärte kann den Anteil der reflektierten Wellen erheblich beeinflussen. Harte Materialien reflektieren die Ultraschallwellen nahezu komplett, wohingegen weiche Materialien mit abnehmender Härte die Wellen zunehmend absorbieren (vgl. Hering und Schönfelder, 2012, S. 182).

Bei heißen Objekten kommt es vor, dass die Ultraschallwellen durch Wärmekonvektion abgelenkt beziehungsweise abgeschwächt werden und diese nicht zum Ultraschallsensor zurückreflektiert werden. Des Weiteren kann es, wie in Abbildung 2.4.c) abgebildet, durch unterschiedliche Temperaturen der einzelnen Luftmassen dazukommen, dass die Schallwellen an den Grenzflächen der Luftmassen reflektiert werden und nicht an dem Objekt (vgl. Hering und Schönfelder, 2012, S. 182).

Ebenfalls einen starken Einfluss kann hohe Luftbewegungen ausüben, die zu einem instabilen Messergebnis führen können. Wobei allerdings Strömungsgeschwindigkeiten bis zu einigen $\frac{m}{s}$ verkraftet werden können (vgl. Hering und Schönfelder, 2012, S. 182).

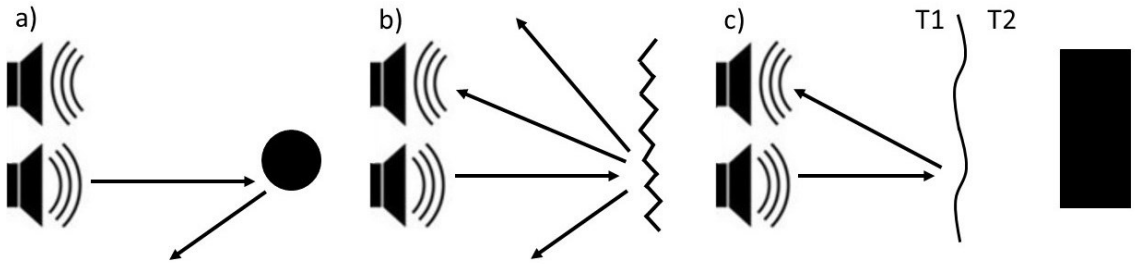


Abbildung 2.4.: Beispielhafte Beeinflussungen der Ultraschallmessungen

2.3. Stand der Technik

Im Folgenden wird der Stand der Technik von Ultraschallsensoren im Bereich der Indoornaviagtion und -positionierung dargestellt. Des Weiteren wird auf Ergebnisse von Vorarbeiten eingegangen. Bei den für diese Arbeit relevanten Vorarbeiten, auf die im Nachfolgenden eingegangen wird, handelt es sich um einen entwickelten Ultraschallsensor mit Objektklassifizierung und einem Multisensorboard zur Aufzeichnung von Luft- und Körperschall.

2.3.1. Ultraschall Lokalisationssystem

Ein für diese Arbeit relevanter Stand der Technik ist ein entwickeltes Indoor-Lokalisationssystem, welches ausschließlich auf Ultraschall basiert. Das System kommt in Umgebungen zum Einsatz, in denen andere Technologien zur Lokalisation eingeschränkt einsetzbar sind. Im Zuge der Arbeit wurde ein Ultraschall-Sendepuls entwickelt, der gegen Umgebungsrauschen beständig ist (vgl. Albuquerque, 2013, S. M).

Zur Lokalisation werden fest positionierte Ultraschall-Sender verwendet. Mobile Geräte lokalisieren sich dabei durch die Differenzzeit der ankommenden Ultraschall-Sendepulse selbst (vgl. Albuquerque, 2013, S. M).

Um eine genaue Lokalisation zu erreichen, müssen die Ultraschall-Sender synchron arbeiten. Eine Synchronisation wird dabei durch drei Ultraschall-Pulse realisiert, die von den Sendern ausgesendet werden (vgl. Albuquerque, 2013, S. M).

Mit dem entwickelten Lokalisationssystem, welches ein Prinzip ähnlich dem von Satellitennavigationssystemen besitzt (vgl. Flühr, 2012, S. 103), ist eine einfache Netzwerkumsetzung möglich, bei der nur die Positionen der Ultraschall-Sender bekannt sein muss. Des Weiteren ist die Größe des Systems individuell skalierbar (vgl. Albuquerque, 2013, S. 175).

2.3.2. Ultraschallsensor mit Objektklassifizierung

Im Rahmen eines Projektes im *Urban Mobility Lab* wurde im Vorfeld ein Ultraschallsensor mit Objektklassifizierung für den Einsatz als FAS entwickelt. Dabei umfasst die Arbeit neben der Hardwareauswahl, die elektronische Schaltungsentwicklung und den Softwareentwurf (vgl. Kölln, 2015, S. 3).

Zur Steuerung des Ultraschallsensors wird das Mikrocontrollerboard *Arduino Due* verwendet, welches einen *ARM Cortex-M3* Prozessor mit einer Taktrate von 84 MHz besitzt. Der Vorteil eines *Arduino* Mikrocontrollerboards ist die einfache Programmierung über die USB-Schnittstelle. Ein weiterer Vorteil des *Arduinos* ist der auf dem Board integrierte Temperatursensor (vgl. Kölln, 2015, S. 19).

Als Ultraschallwandler wird der *Prowave* $400Sx160$ verwendet. Dieser besitzt eine Resonanzfrequenz von 40 kHz bei einer Toleranz von $\pm$ 1 kHz. Dabei hat der Wandler als Sender eine Bandbreite von 2 kHz und als Empfänger eine Bandbreite von 2,5 kHz (vgl. Kölln, 2015, S. 23).

Als Schnittstelle zwischen dem Mikrocontroller und den Ultraschallwandlern wurden zwei elektronische Schaltungen entworfen. Eine Schaltung für den Sendewandler und eine für den Empfängerwandler. Die Sendeschaltung wurde als Spannungsverstärkerschaltung mit

Transformator realisiert, welche einen Verstärkungsfaktor von 10 besitzt (vgl. Kölln, 2015, S. 34). Die Empfängerschaltung wurde aus einer Kombination von drei Verstärkerstufen realisiert. Dabei weisen die erste und dritte Stufe jeweils Hochpassverhalten auf. Die zweite Stufe ist ein mehrfach rückgekoppelter Bandpassfilter (vgl. Kölln, 2015, S. 32).

Für die Implementierung der Software auf dem Mikrocontroller wurde die Programmiersprache *C* verwendet. Der Softwareentwurf unterteilt sich in fünf unterschiedliche Funktionen. Das Senden des Ultraschallimpulses wird von der Funktion *sendBurst* ausgeführt, indem an den Sendewandler ein PWM-Signal geschickt wird. *receive* konfiguriert den Analog-Digital-Wandler (ADC) und verarbeitet das Echosignal des Empfängerwandlers. Die dritte Funktion, *getTemperature*, konfiguriert ebenfalls den ADC und gibt die Temperatur in °C zurück. Der Abstand zum Messobjekt wird mit *getDistance* ermittelt und in m zurückgegeben. *objectclassification* klassifiziert das Messobjekt, indem eine Korrelationsanalyse durchgeführt wird (vgl. Kölln, 2015, S. 63).

2.3.3. Multisensorboard

Ein weiteres Projekt im *Urban Mobility Lab*, welches im Vorfeld durchgeführt wurde, ist ein Multisensorboard das im Zuge eines hochkanaligen Messsystems zur synchronen Aufnahme und Verarbeitung von Luft- und Körperschall entwickelt (vgl. Wenzel, 2016, S. 3).

Das hochkanalige Messsystem besteht aus drei hierarchischen Ebenen. Die unterste Ebene besteht aus mehreren Multisensorboards. Die zweite Ebene wird mit mehreren Schnittstellen- und Prozessorboards realisiert, welche die Daten von mehreren Multisensorboards aufnehmen und vorverarbeiten. Die oberste Ebene bildet ein Prozessorboard. Dieses stellt ein Userinterface zur Systemsteuerung bereit und ist zentraler Zeitgeber im System (vgl. Wenzel, 2016, S. 59).

Das Multisensorboard besteht aus drei Sensoren und einem Mikrocontroller. Bei dem Mikrocontroller handelt es sich um den *Atmel SAMG53*. Zu den Sensoren gehört unter anderem ein Beschleunigungssensor, dem *ST Microelectronics LIS3DSH*, welcher die Beschleunigung über drei Achsen aufnimmt. Die anderen Sensoren sind zwei MEMS-Mikrofone. Das *ST Microelectronics MP45DT02* hat einen Frequenzbereich von 10 kHz bis 20 kHz. Das *Knowles SPU0641LU4H* deckt einen Frequenzbereich von 10 kHz bis 80 kHz ab (vgl. Wenzel, 2016, S. 61).

Zur Signalübertragung dient eine SPI-Schnittstelle, über die die Daten von einem Master abgerufen werden können (vgl. Wenzel, 2016, S. 71).

3. Analyse

Die Analyse beschäftigt sich zunächst mit einem typischen Anwendungsfall für das Entwicklungssystem. Darauf aufbauend wird eine Analyse durchgeführt, bei der die Anforderungen an das System definiert werden.

3.1. Anwendungsfall

Mit dem Stand der Technik bei Ultraschallsensoren wird eine einfache Abstandsmessung durchgeführt, bei der das erste zurückkommende Echo gemessen wird. Damit sind die Erkennung der Umgebung und die Lokalisation von verdeckten Objekten nur bedingt möglich.

Mithilfe des Entwicklungssystems soll eine Entwicklung von intelligenten Ultraschallsensoren für Fahrzeug zur Erkennung von verdeckten Hindernissen in komplexen Umgebungen ermöglicht werden. Dazu ist für den Einsatz des Entwicklungssystems die Montage auf einem *Tesla Model S* mit integrierten *Linux*-Rechner geplant (siehe Abbildung 3.1). Als Versuchsumgebung dient eine öffentliche Garage, in der verdeckte Objekte, zum Beispiel andere Fahrzeuge, die hinter einer Ecke stehen, erkannt werden sollen.

Mit dem System sollen die Ultraschallechos der Umgebung analysiert werden. Die Untersuchung der Frequenzabhängigkeit bei der Objekterkennung soll hierdurch vereinfacht werden. Eine Auswirkung des Ultraschall-Sendepulses soll ebenfalls mit dem Entwicklungssystem analysiert werden.

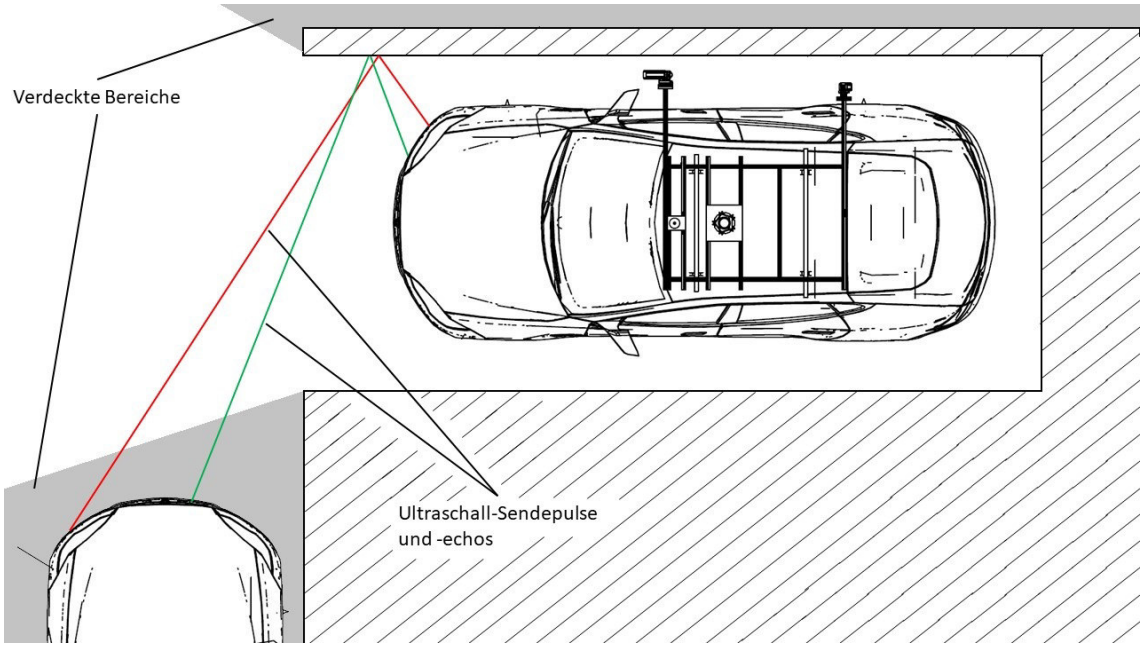


Abbildung 3.1.: Anwendungsfall des Entwicklungssystems

3.2. Anforderungsanalyse

Aufbauend auf der Aufgabenstellung und dem Anwendungsfall werden Anforderungen definiert und nach Möglichkeit mit einem Wert quantifiziert (siehe Tabelle 3.1). Um eine Auswertung der Umgebung und von Objekten zu gewährleisten, ist es nötig, dass das Entwicklungssystem Objekte und die Umgebung erkennt. Da das Entwicklungssystem im Automobilbereich angewendet wird, wird die Anzahl an Ultraschallsensoren auf mindestens 12 gesetzt, welches einer gängigen Anzahl in Fahrzeugen entspricht (vgl. Ingle und Phute, 2016, S. 370). Des Weiteren sollte eine flexible Mehrfachanordnung der Ultraschallsensoren möglich sein.

Um eine Analyse der Frequenzabhängigkeit bei der Objekterkennung durchführen zu können, sollten die Ultraschallsensoren eine möglichst hohe Bandbreite besitzen. Diese sollte über 20 kHz betragen. Um das Nyquist-Shannon-Abstasttheorem einzuhalten, muss die Abtastrate des Echosignals bei mindestens der doppelten Bandbreite des Ultraschallsensors liegen. Die Objekterkennung in komplexen Umgebungen erfordert ebenfalls eine möglichst hohe Genauigkeit des Ultraschallsensors, welche auf mindestens 1 cm festgelegt wird. Aus der Genauigkeit des Ultraschallsensors wird die erforderliche Auflösung geschlossen. Die Auflösung wird auf mindestens 0,5 cm festgelegt. Um eine Objekterkennung in komplexen Umgebungen zu realisieren, sollte der Messbereich über 5 m betragen.

Um eine Lokalisation von Objekten mit mehreren Ultraschallsensoren zu realisieren, ist es notwendig, dass diese synchron arbeiten. Hierzu wird die maximale Abweichung pro Ultraschallsensor durch eine verzögerte Messung auf 0,5 cm festgelegt. Nimmt man eine Temperatur von maximal 30 °C an, lässt sich die maximal zulässige Messverzögerung ermitteln:

$$\Delta t = \frac{2\Delta x}{c_{20°C}} = 28,62\,\mu s \tag{3.1}$$

Der Ultraschall-Sendepuls sollte in Bezug auf Frequenz, Pulsdauer und Schalldruck variabel einstellbar sein, um Auswirkungen unterschiedlicher Sendepulse auf die Messergebnisse zu untersuchen. Zudem sollte die Aufnahmezeit der Echosignale ebenfalls variabel einstellbar sein, um mehrere Ultraschallechos empfangen zu können. Dazu wird die maximale Aufnahmezeit auf mindestens 200 ms festgelegt. Hierbei könnte ein Ultraschallecho über mehrere Reflexionen in der Umgebung eine Strecke von 68,68 m zurücklegen.

Der Betrieb des Entwicklungssystems ist unter anderem in Parkhäusern geplant. Daraus folgt für den Betrieb des Entwicklungssystems ein Temperaturbereich von -10 °C bis 30 °C. Ebenfalls folgt daraus, dass der Betrieb unter regengeschützten Umgebungsbedingungen erfolgt, wodurch das Entwicklungssystem nicht wasserdicht ausgelegt werden muss. Da die Schallgeschwindigkeit temperaturabhängig ist, ist es nötig, die Umgebungstemperatur zu erfassen, um möglichst genaue Messergebnisse zu erzielen. Der zu erfassende Temperaturbereich wird dabei an den Temperaturbereich für den Betrieb des Entwicklungssystems gekoppelt. Da der Umgebungsdruck Auswirkungen auf die Dämpfung der Schallwellen hat und damit die Auswirkungen bei verändertem Umgebungsdruck analysierbar sind, sollte dieser erfasst werden. Der Messbereich wird dabei auf 90 kPa bis 110 kPa festgelegt.

Um die Ultraschallsensoren zu steuern und die empfangenen Ultraschallechos zu analysieren, sollte das Entwicklungssystem eine Schnittstelle zu dem *Linux*-Rechner im Tesla bereitstellen. Um einen verlustfreien Datenaustausch der Echodaten zu gewährleisten, sollte die Schnittstelle eine hohe Übertragungssicherheit aufweisen, bei maximal 1 ppm Datenverlust auftritt. Des Weiteren sollte das Entwicklungssystem über eine Software gesteuert und ausgewertet werden können, die eine grafische Benutzeroberfläche enthält. Die grafische Ausgabe der empfangenen Echos sollte dabei ermöglicht werden. Des Weiteren sollte die Software, da ein Betrieb auf einem *Linux*-Rechner vorgesehen ist, *Linux*-kompatibel sein.

Um einen möglichst flexiblen Einsatz des Entwicklungssystems im Bezug auf die Entwicklung von intelligenten Ultraschallsensoren zu gewährleisten, sollten die zu entwickelnden Algorithmen auf der rechnerseitigen Software implementiert werden können.

ID	Beschreibung	Wert
1	Erkennung von Objekten und der Umgebung	
2	System aus mehreren Ultraschallsensoren	$n > 12$
3	Flexible Mehrfachanordnung der Ultraschallsensoren	
4	Ultraschallsensoren mit hoher Frequenzbandbreite	> 20 kHz
5	Ausreichend hohe Abtastrate	$f_s > 2\ B$
6	Möglichst hohe Genauigkeit der Ultraschallsensoren	$\Delta x < 1$ cm
7	Möglichst hohe Auflösung der Ultraschallsensoren	$\Delta x < 0{,}5$ cm
8	Möglichst hoher Messbereich der Ultraschallsensoren	$x > 5$ m
9	Ultraschallsensoren müssen synchron arbeiten	$\Delta t < 28{,}62\ \mu$s
10	Unterschiedliche Sendesignale (Pulsdauer, Frequenz, Amplitude)	
11	Variable Aufnahmezeit des Echosignals	$t > 200$ ms
12	Betrieb bei Außentemperaturen	-10 °C bis 30 °C
13	Betrieb unter regengeschützten Bedingungen	
14	Erfassung der Umgebungstemperatur	-10 °C bis 30 °C
15	Erfassung des Umgebungsdrucks	90 kPa bis 110 kPa
16	Schnittstelle zu einem Rechner	
17	Hohe Übertragungssicherheit der Schnittstelle	< 1 ppm Datenverlust
18	Steuerung der Ultraschallsensoren über eine Software mit grafischer Bedienoberfläche	
19	Grafische Ausgabe der Echosignale	
20	*Linux*-Kompatibilität der Steuerungssoftware	
21	Implementierung der intelligenten Ultraschall-Algorithmen auf der rechnerseitigen Software	

Tabelle 3.1.: Anforderungen an das Entwicklungssystem

4. Konzept

In diesem Teil der Arbeit wird aufbauend auf den Anforderungen an das Entwicklungssystem eine Systemarchitektur konzipiert und eine Untergliederung in Subsysteme vorgenommen. Des Weiteren wird ein Konzept für das Kommunikationsnetzwerk zwischen den Ultraschallsensoren und der Datenaufnahme entwickelt. Abschließend wird das Konzept für den Ultraschallsensor entworfen.

4.1. Systemarchitektur

Die Systemarchitektur teilt sich in zwei Komponenten auf und stellt eine hierarchische Gliederung dar (siehe Abbildung 4.1). Bei der hierarchisch oberen Komponente handelt es sich um die Datenaufnahme auf einem *Linux*-Rechner, die das Entwicklungssystem steuert und die Daten der Ultraschallsensoren auswertet. Die zweite Komponente sind die Ultraschallsensoren, welche für den Messvorgang zuständig sind. Um die Ultraschallsensoren zu steuern beziehungsweise um die Daten der Ultraschallsensoren an die Datenaufnahme zu versenden, sind alle Ultraschallsensoren über ein Kommunikationsnetzwerk mit der Datenaufnahme verbunden.

Aufbauend auf der Systemarchitektur wird das Entwicklungssystem in eigenständige Subsysteme untergliedert, um die Konzeption, Entwicklung und Tests zu vereinfachen. Dabei wird das Entwicklungssystem in die Subsysteme Datenaufnahme und Ultraschallsensor aufgeteilt.

Die Datenaufnahme stellt die Schnittstelle zwischen Entwicklungssystem und Benutzer dar. Zu den Funktionen der Datenaufnahme gehört die Steuerung des Entwicklungssystems. Dies beinhaltet unter anderem die grafische Darstellung des User-Interfaces. Ein weiterer Punkt der Datenaufnahme ist die Aufbereitung und Visualisierung der gemessenen Ultraschallechosignale. Die Kommunikation mit den Ultraschallsensoren wird ebenfalls von der Datenaufnahme übernommen.

Die Ultraschallsensoren sind für den Messvorgang zuständig. Dies beinhaltet sowohl das Senden der Ultraschall-Pulse als auch das Aufnehmen der zurückkommenden Echosignale. Eine weitere Aufgabe ist die Kommunikation zwischen den einzelnen Ultraschallsensoren,

um einen synchronen Messablauf zu gewährleisten. Dafür übernimmt ein Ultraschallsensor die Aufgabe des Masters und die anderen des Slaves. Hierbei kann der Masterstatus variabel den einzelnen Ultraschallsensoren zugewiesen werden.

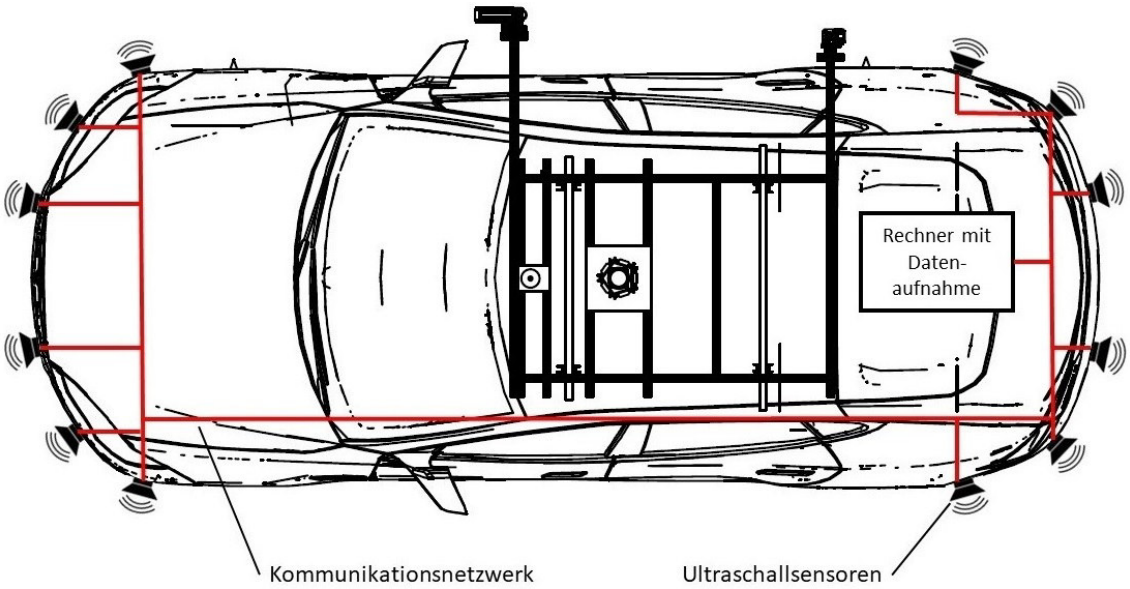


Abbildung 4.1.: Systemarchitektur des Entwicklungssystems

4.2. Kommunikationsnetzwerk

Um die benötigte Datenübertragungsrate des Kommunikationsnetzwerks zu ermitteln, wird zunächst eine Abschätzung des Datenaufkommens berechnet. Zur Berechnung des Datenaufkommens werden die Anzahl der Ultraschallsensoren, eine Abtastrate und die Größe pro Abtastwert benötigt. Als Anzahl der Ultraschallsensoren werden 12 festgelegt. Die Abtastrate wird auf 200 kHz abgeschätzt. Als Größe für einen Abtastwert werden 2 Byte angenommen. Um eine Sicherheitsreserve für Übertragungsprotokoll-Header, Temperatur und Umgebungsdruck und andere Daten zu realisieren, werden 10 $\frac{Mbit}{s}$ für diese Daten angenommen. Daraus folgt für das abgeschätzte Datenaufkommen:

$$Daten = Sensoren * Abtastrate * Abtastwert + Sicherheit = 46,62 \frac{MBit}{s} \quad (4.1)$$

Als Kommunikationsnetzwerke werden Ethernet und WLAN untersucht und verglichen, um daraus das passende Netzwerk zu wählen.

Ethernet

Ethernet ist ein nach IEEE 802.3 genormter Datenbus, welcher sich auf den Schichten 1 und 2 des ISO-OSI-Referenzmodells befindet. Das Buszugriffsverfahren ist CSMA/CD, wodurch Ethernet nicht echtzeitfähig ist. Die Datenübertragungsrate bei Ethernet beträgt 10 $\frac{Mbit}{s}$ beziehungsweise 100 $\frac{Mbit}{s}$ bei Fast Ethernet. Es gibt allerdings auch Ethernet Anwendungen im Gigabitbereich mit bis zu 100 $\frac{Gbit}{s}$. Die Reichweite beträgt je nach Übertragungsrate und verwendetem Kabel von mindestens 100 m bis zu 10 km (vgl. Reißenweber, 2009, S. 273).

Als Bustopologie sind Linien- und Sterntopologien möglich, wobei die Sterntopologie sich als vorteilhafter erweist, da hierdurch auf den Datenleitungen weniger Kollisionen entstehen. Realisiert wird die Sterntopologie dadurch, dass die Endgeräte mit einem Switch verbunden sind, welcher die Daten weiterleitet (vgl. Zimmermann und Schmidgall, 2014, S. 138).

WLAN

Das Wireless Local Area Network (WLAN) ist ein nach IEEE 802.11 normiertes Funknetz. WLAN befindet sich auf den Schichten 1 und 2 des ISO-OSI-Referenzmodells. Die Funkfrequenz von WLAN beträgt 2,4 GHz beziehungsweise 5 GHz. Die Übertragungsraten betragen mindestens 54 $\frac{Mbit}{s}$, kann unter idealen Bedingungen auf bis zu 6,93 $\frac{Gbit}{s}$ ansteigen. Bei der Reichweite muss man unter idealem Freifeld und zum Beispiel Gebäuden unterscheiden. Unter Freifeldbedingungen beträgt die Reichweite 100 bis 300 m. In Gebäuden sinkt die Reichweite auf 30 bis 50 m. Das Buszugriffsverfahren ist CSMA/CA. Auf CSMA/CD wird verzichtet, da Übertragungskollisionen von Störquellen nicht zu unterscheiden sind. Topologien sind bei WLAN mehrere möglich, wobei bei allen mehrere Geräte mit einem Zugriffspunkt verbunden sind. Dabei kann jeweils nur ein Gerät mit dem Zugriffspunkt kommunizieren (vgl. Gessler und Krause, 2015, S. 232).

Vergleich und Auswahl

Die Übertragungsraten von Fast Ethernet mit 100 $\frac{Mbit}{s}$ beziehungsweise von WLAN mit mindestens 54 $\frac{Mbit}{s}$ sind ausreichend hoch, um eine Kommunikation zwischen der Datenaufnahme und den Ultraschallsensoren zu gewährleisten. Dabei kann allerdings die Übertragungsrate bei WLAN durch andere Funknetzwerke oder Störquellen in der Nähe beeinträchtigt werden. Mit einem Ethernet-Netzwerk ist eine Sterntopologie möglich bei der es zwischen den jeweiligen Busteilnehmern und einem Switch als Sternpunkt zu keinen Übertragungskollisionen kommt. Bei WLAN kann es zu Übertragungskollisionen kommen, da alle Geräte mit dem gleichen Zugriffspunkt verbunden sind. Mit beiden Netzwerken ist es möglich, ein Netzwerk aufzubauen, in dem sich nur die Geräte des Entwicklungssystems befinden.

Daraus folgt, das Ethernet und WLAN als Kommunikationsnetzwerk geeignet sind. Als Kommunikationsnetzwerk fällt die Wahl auf Ethernet, da hier ein Netzwerk ohne Datenkollisionen realisiert werden kann und die Datenübertragungsrate durch andere Funknetzwerke nicht beeinträchtigt wird.

4.3. Ultraschallsensor

Bei der Entwicklung des Ultraschallsensorkonzepts wird zunächst eine Architektur des Ultraschallsensors entworfen. Aufbauend auf der Architektur wird dann im nächsten Schritt eine Auswahl der Komponenten durchgeführt.

4.3.1. Ultraschallsensor-Architektur

Aufbauend auf der Systemarchitektur und den Anforderungen an den Ultraschallsensor wird für den Ultraschallsensor eine Architektur entworfen (siehe Abbildung 4.2). Da die Zuteilung des Masterstatus variabel ist, besitzen alle Ultraschallsensoren die selbe Architektur.

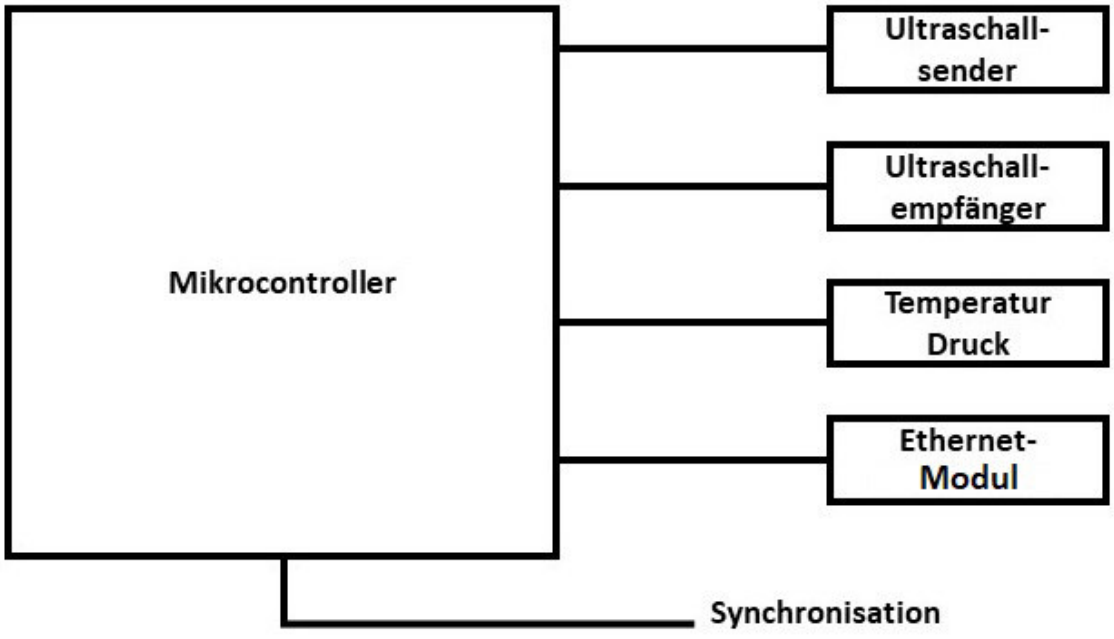


Abbildung 4.2.: Architektur des Ultraschallsensors

Das Konzept sieht einen Mikrocontroller als zentrale Recheneinheit vor, der die Kommunikation mit den einzelnen Komponenten übernimmt und diese steuert. Zum Senden der

Ultraschall-Pulse beziehungsweise Empfang der Echos werden zwei voneinander getrennte Pfade gewählt. Ein weiterer Block ist für das Erfassen von Umgebungsdruck und -temperatur zuständig. Da der Ultraschallsensor mit der Datenaufnahme auf dem Rechner kommunizieren muss, steuert der Mikrocontroller eine Ethernet-Modul. Ein weiterer Port am Mikrocontroller dient der Synchronisation der Ultraschallsensoren.

4.3.2. Mikrocontrollerboard

Die Eigenschaften des Mikrocontrollers sind für die Leistungsfähigkeit des Ultraschallsensors entscheidend und sind daher von Bedeutung. Im Folgenden werden zunächst zwei Mikrocontrollerboards, der $Arduino\ Due$ und der modulare Datenlogger, analysiert und abschließend verglichen.

Arduino Due

Das Mikrocontrollerboard $Arduino\ Due$ besitzt den $Atmel\ SAM3X8E$ Prozessor mit einer Taktrate von 84 MHz, einem 512 kB Flash-Speicher und einem 96 kB SRAM-Speicher. Auf dem Mikrocontrollerboard sind 54 digitale I/O-Pins, 12 analoge Eingänge und 2 analoge Ausgänge. Die analogen Ein- beziehungsweise Ausgänge haben eine Auflösung von 12-bit und werden mit bis zu 1 Msps abgetastet. Der Timer hat eine Größe von 32-bit. Die Schnittstellen des Mikrocontrollerboards, die für das Entwicklungssystem von Interesse sind, sind CAN, I2C, SPI und USB. Des Weiteren besitzt der $Arduino\ Due$ einen integrierten Temperatursensor. Die Versorgungsspannung des Mikrocontrollerboards sollte zwischen 7 V und 12 V liegen. Das Board kann 5 V und 3,3 V an externe Peripherie liefern. Die Ausgangsspannung der einzelnen Pins liegt bei maximal 3,3 V. Die Abmessungen des $Arduino\ Dues$ betragen 102 x 53 x 12 mm (vgl. Arduino, 2018).

Modularer Datenlogger

Der modulare Datenlogger basiert auf einem $Atmel\ SAM4E16E\ ARM$ Prozessor. Die Taktrate des Prozessors beträgt 120 MHz. Der Speicher besteht aus einem 1024 kB Flash und einem 128 kB SRAM. Von den 117 digitalen I/O-Pins des Prozessors sind 76 auf dem Mikrocontrollerboard nutzbar. Für analoge Signale sind 24 Eingänge beziehungsweise 2 Ausgänge auf dem Board vorhanden. Dabei besitzen die analogen Pins eine Auflösung von 12-bit. Die analogen Eingänge darüber hinaus mit Mittelwertbildung bis zu 16-bit. Sowohl die Ein- als auch die Ausgänge werden mit bis zu 1 Msps abgetastet. Der $SAM4E16E$ besitzt, wie der $SAM3X8E$ Prozessor, ebenfalls einen 32-bit-Timer. Die Schnittstellen des Mikrocontrollerboards sind CAN, Ethernet, I2C und USB. SPI steht auf dem Board nicht zur Verfügung. Der Prozessor besitzt einen integrierten Temperatursensor. Die Versorgungsspannung

liegt bei 5 V und die Ausgangsspannung für externe Peripherie bei 5 V beziehungsweise 3,3 V. Die Ausgangsspannung der einzelnen Pins liegt bei 3,3 V. Hierbei besitzt das Board die Abmessungen 108 x 100 x 35 mm (vgl. Schöne, 2017, S. 36).

Vergleich und Auswahl

Unterzieht man beide Mikrocontrollerboards einem Vergleich (siehe Tabelle 4.1), besitzt der Datenlogger bessere beziehungsweise gleichwertige Spezifikationen. Sowohl der Prozessortakt als auch die Anzahl der verfügbaren Pins und die Speichergrößen fallen jeweils bei dem Datenlogger höher aus. Des Weiteren kann die Auflösung der analogen Eingänge erhöht werden. Beide Boards besitzen vier unterschiedliche Schnittstellen, wobei der Datenlogger statt einer SPI-Schnittstelle eine Ethernet-Schnittstelle hat. Vergleicht man die Abmessungen, nimmt der Arduino Due deutlich weniger Platz ein.

Eigenschaft	**Arduino Due**	**Datenlogger**
Prozessortakt	84 MHz	120 MHz
Digitale I/o-Pins	54	76
Analoge Eingänge (Auflösung)	12 (12 bit)	24 (12 bit)
Analoge Ausgänge (Auflösung)	2 (12 bit)	2 (12 bit)
Abtastrate analoge Pins	1 Msps	1 Msps
Flash-Speicher	512 kB	1024 kB
SRAM	96 kB	128 kB
Schnittstellen	CAN, I2C, SPI, USB	CAN, Ethernet, I2C, USB
Versorgungsspannung	7 - 12 V	5 V
Ausgangsspannung	3,3 / 5 V	3,3 / 5 V
Abmessungen	102 x 53 x 12 mm	108 x 100 x 35 mm

Tabelle 4.1.: Gegenüberstellung der Mikrocontrollerboards

Da der Datenlogger zwar zum Teil bessere Spezifikationen hat, diese allerdings nicht benötigt werden und durch die Erfahrungen aus der Vorarbeit wird der $Arduino\ Due$ als Mikrocontrollerboard ausgewählt.

4.3.3. Ultraschallwandler

Da mit den Ultraschallwandlern die Ultraschallwellen ausgesendet beziehungsweise empfangen werden, werden mit diesen die möglichen Frequenzen und die Bandbreite des Ultraschallsensors bestimmt. Nach einer Analyse der Ultraschallwandler werden diese anschließend einem Vergleich unterzogen. Da der $Prowave\ 400Sx160$ Ultraschallwandlers, wel-

cher im Vorfeld verwendet wurde, mit 2 kHz beim Senden beziehungsweise 2,5 kHz beim Empfangen eine zu niedrige Bandbreite besitzt, wird dieser nicht analysiert. Des Weiteren wird an dieser Stelle das Multisensorboard ebenfalls nicht untersucht, da bei diesem eine Ultraschallmessung noch nicht implementiert wurde und dieses dahin gehend optimiert werden muss. Bei den Ultraschallwandlern handelt es sich um dem $SensComp\ Series\ 600$, dem $Kemo\ L010$ und dem $Knowles\ SPU0410LR5H$. Weiterhin wird auf ein System bestehend aus zwei Ultraschallwandlern gesetzt, da hiermit ein gleichzeitiges Senden und Empfangen möglich ist.

SensComp Series 600

Der $Series\ 600$ von $SensComp$ ist ein elektrostatischer Ultraschallwandler und kann sowohl zum Senden als auch zum Empfangen eingesetzt werden. Der $Series\ 600$ hat eine Resonanzfrequenz von 50 kHz. Die Bandbreite liegt beim Aussenden bei ca. 15 kHz und beim Empfangen bei ca. 20 kHz. Der Schallpegel beim Senden beträgt mindestens 110 dB. Die Empfangsempfindlichkeit beträgt mindestens -42 $\frac{dBV}{Pa}$. Der Ultraschallwandler kann maximal mit einer Amplitude von 200 V bei einem Offset von 200 V angesteuert werden. Das $SensComp\ Series\ 600$ hat die Maße Ø 43 x 12 mm (vgl. SensComp, 2013, S. 2).

Kemo L010

Bei dem $Kemo\ L010$ handelt es sich um einen piezoelektrischen Lautsprecher, welcher sich nur zum Aussenden eignet. Der Lautsprecher besitzt einen Frequenzbereich von 2 kHz bis 60 kHz und somit eine hohe Bandbreite. Beim Senden beträgt der Schallpegel bis zu 120 dB. Die Amplitude der Wechselspannung, mit der der Lautsprecher angesteuert wird, beträgt maximal $\pm$ 15 V. Die Abmessungen des $Kemo\ L010$ sind Ø 41 x 12 mm (vgl. Kemo-Electronic, 2009).

Knowles SPU0410LR5H

Das $Knowles\ SPU0410LR5H$ ist ein MEMS-Mikrofon und eignet sich daher nur als Empfänger. Das Mikrofon ist für einen Frequenzbereich von 100 Hz bis 80 kHz spezifiziert, wobei der Frequenzgang im Bereich von etwa 49 kHz bis 80 kHz nahezu flach ist. Die Empfindlichkeit ist mit -38 $\frac{dBV}{Pa}$ bei 1 kHz angegeben. Die Versorgungsspannung sollte im Bereich von 1,5 V bis 3,6 V liegen. Die Messdaten des Mikrofons werden analog ausgegeben. Die Abmessungen des Mikrofons betragen 3,76 x 3 x 1,1 mm (vgl. Knowles-Electronics, 2013, S. 2).

Vergleich und Auswahl

Da nicht alle analysierten Ultraschallwandler für das Aussenden und Empfangen von Ultraschallwellen geeignet sind, werden zunächst der $SensComp\ Series\ 600$ und der $Kemo\ L010$, welche als Sender geeignet sind, miteinander verglichen (siehe Tabelle 4.2). Anschließend wird der $SensComp\ Series\ 600$ mit dem $Knowles\ SPU0410LR5H$ verglichen, um hiermit einen Ultraschallwandler zum Empfangen der Ultraschallwellen zu wählen (siehe Tabelle 4.3).

Der Frequenzbereich des $Kemo$ Lautsprechers weist mit 2 kHz bis 60 kHz einen deutlich höheren Einsatzbereich als der $SensComp$ Ultraschallwandler auf. Des Weiteren fällt der Sendeschallpegel des $Kemo$ Lautsprechers ebenfalls höher aus. Ein weiterer Vorteil des $Kemo\ L010$ ist die deutlich geringere benötigte Versorgungsspannung. Die Abmessungen der Sende-Ultraschallwandler sind näherungsweise identisch. Da der $Kemo\ L010$ deutlich bessere Spezifikationen aufweist, wird dieser ausgewählt.

Eigenschaft	**SensComp Series 600**	**Kemo L010**
Frequenzbereich	50 kHz - 65 kHz	2 kHz - 60 kHz
Sendeschallpegel	100 dB	120 dB
Versorgungsspannung	400 V	$\pm$ 15 V
Abmessungen	Ø 43 x 12 mm	Ø 42 x 12 mm

Tabelle 4.2.: Gegenüberstellung der Sende-Ultraschallwandler

Vergleicht man den $SensComp\ Series\ 600$ mit dem $Knowles\ SPU0410LR5H$, fällt der mögliche Einsatzbereich für das $Knowles$ MEMS-Mikrofon deutlich höher aus. Dabei weist das $Knowles$ MEMS-Mikrofon eine etwas geringere Empfangsempfindlichkeit auf. Ebenfalls Vorteile des $Knowles\ SPU0410LR5H$ sind die deutlich geringere Versorgungsspannung und die nahezu vernachlässigbar kleinen Abmessungen. Durch die deutlichen Vorteile und aus der Erfahrung mit MEMS-Mikrofonen von $Knowles$ aus der Vorarbeit wird dieses als Empfänger-Ultraschallwandler ausgewählt.

Eigenschaft	**SensComp Series 600**	**Knowles SPU0410LR5H**
Frequenzbereich	45 kHz - 65 kHz	100 Hz - 80 kHz
Empfangsempfindlichkeit	-42 $\frac{dBV}{Pa}$	-38 $\frac{dBV}{Pa}$
Versorgungsspannung	400 V	3,6 V
Abmessungen	Ø 43 x 12 mm	3,76 x 3 x 1,1 mm

Tabelle 4.3.: Gegenüberstellung der Empfänger-Ultraschallwandler

4.3.4. Ethernet-Modul

Um die Daten über die Schnittstelle an die Datenaufnahme zu schicken und da das $Arduino\ Due$ Board keine eigene Ethernet-Schnittstelle besitzt, ist ein Ethernet-Modul nötig. Im Folgenden werden das $W5500\ Ethernet\ Shield$ und das $ENC28J60\ LAN\ Network\ Modul$ auf ihre Eigenschaften analysiert und miteinander verglichen.

WIZnet W5500 Ethernet Shield

Das $WIZnet\ W5500\ Ethernet\ Shield$ ist Ethernet-Modul, welches einen $W5500$ Chip verbaut hat. Das Modul ist passend für $Arduino$ Boards und kann auf diese einfach aufgesteckt werden. Es werden die Übertragungsraten 10 $\frac{Mbit}{s}$ und 100 $\frac{Mbit}{s}$ im Halb- und Vollduplexbetrieb unterstützt. Die Daten können über die Protokolle TCP und UDP versendet werden. Dazu stehen bis zu 8 Sockets zur Verfügung, die mit unterschiedlichen Empfängeradressen konfiguriert werden können. Zur Zwischenspeicherung der Nutzdaten stehen für das Senden und Empfangen je 32 kB Speicher zur Verfügung. Die Kommunikation mit dem Ethernet Shield findet über SPI statt. Hierbei ist eine Taktrate von bis zu 33 MHz möglich. Der Datenaustausch über SPI ist nur im Halbduplexbetrieb möglich. Das Ethernet Shield hat zusätzlich einen SD-Karten-Slot. Die Versorgungsspannung muss 3,3 V betragen. Die Abmessungen betragen 69 x 53 x 23 mm (vgl. WIZnet, 2013, S. 2).

Microchip ENC28J60 Ethernet Modul

Das $ENC28J60\ Ethernet\ Modul$ basiert auf den $ENC28J60$ Chip von $Microchip$ und unterstützt eine Übertragungsrate von 10 $\frac{Mbit}{s}$ im Halb- und Vollduplexbetrieb. Die Nutzdaten können mit TCP und UDP Protokollen übertragen werden. Es steht ein Socket zur Verfügung, über dem die Daten ausgetauscht werden. Die Speichergröße zum Zwischenspeichern der Daten beträgt für das Senden und Empfangen zusammen 8 kB. Die Daten werden über SPI im Halbduplexbetrieb an das Ethernet-Modul übertragen. Hierbei ist eine maximale Übertragungsrate von 10 MHz möglich. Die Versorgungsspannung beträgt 3,3 V und die Abmessungen betragen 51 x 33 x 17 mm (vgl. Microchip, 2006, S. 1).

Vergleich und Auswahl

Vergleicht man die beiden Ethernet-Module miteinander (siehe Tabelle 4.4), fällt die Auswahl auf das $WIZnet\ W5500\ Ethernet\ Shield$. Mit dem $W5500\ Ethernet\ Shield$ sind mit 100 $\frac{Mbit}{s}$ bei Ethernet und 33 MHz bei SPI deutlich höhere Übertragungsraten möglich. Die größere Anzahl an Sockets, die unterschiedlich konfiguriert werden können, ist vorteilhafter, um unterschiedliche Daten, wie Temperatur, Druck und Ultraschallechosignale, voneinander

unabhängig an den PC schicken zu können. Zuletzt ist der Zwischenspeicher für Nutzdaten ebenfalls höher, wodurch eine längere Zwischenspeicherung möglich ist.

Eigenschaft	W5500 Ethernet Shield	ENC28J60 Ethernet Modul
Übertragungsrate	10 / 100 $\frac{Mbit}{s}$	10 $\frac{Mbit}{s}$
Übertragungsprotokolle	TCP, UDP	TCP, UDP
Zwischenspeicher	64 kB	8,kB
Sockets	8	1
SPI-Takt	bis 33 MHz	bis 10 MHz
Versorgungsspannung	3,3 V	3,3 V
Abmessungen	69 x 53 x 23 mm	51 x 33 x 17 mm

Tabelle 4.4.: Gegenüberstellung der Ethernet-Module

4.3.5. Druck- und Temperatursensor

Um die Temperatur und den Umgebungsdruck zu erfassen, ist hierfür ein Sensor nötig. Bei den untersuchten Sensoren handelt es sich um den auf dem $Atmel\ SAM3X8E$ integrierten Temperatursensor, dem $Adafruit\ BMP280$ Temperatur- und Drucksensor Board und dem $Freescale\ MPX4115A$ Drucksensor.

Atmel SAM3X8E integrierter Temperatursensor

Der $Atmel$-Prozessor $SAM3X8E$ besitzt einen integrierten Temperatursensor mit analogem Ausgang, welcher über den ADC ausgelesen werden kann. Dabei ist die Ausgangsspannung des Sensors proportional zur Absoluttemperatur. Die Genauigkeit des Sensors liegt bei ± 15 % im unkalibrierten Zustand. Nach einer Kalibrierung liegt die Genauigkeit bei $\pm 3\,^{\circ}$C für einen Temperaturbereich von 0 bis 80 °C. Dabei beeinflusst eine Kalibrierung den Temperaturbereich. Je genauer die Temperatur gemessen werden kann, umso kleiner ist der messbare Temperaturbereich (vgl. Atmel, 2015, S. 1410).

Adafruit BMP280 Druck- und Temperatursensor

Das $Adafruit$ Board basiert auf einem $Bosch\ BMP280$ Temperatur- und Drucksensor. Die Messwerte können über I2C mit einer Taktrate von bis zu 3,4 MHz beziehungsweise über SPI mit einer Taktrate von bis zu 10 MHz übertragen werden. Der Messbereich für die Temperatur liegt bei -40 bis 85 °C bei einer Genauigkeit von 0,01 °C. Für den Umgebungsdruck liegt der Messbereich bei 30 bis 110 kPa. Die Genauigkeit beim Umgebungsdruck liegt

bei 0,16 kPa. Das Adafruit Board benötigt eine Versorgungsspannung von 3 bis 5 V und hat die Abmessungen von 19 x 17 x 11 mm (vgl. Bosch-Sensortec, 2018, S. 2).

Freescale MPX4115A Drucksensor

Der $Freescale\ MPX4115A$ ist ein Drucksensor für einen Umgebungsdruckbereich von 15 bis 115 kPa. Die Genauigkeit beträgt dabei 1,5 %. Das Messsignal wird vom Sensor über einen analogen Ausgang ausgegeben. Der Sensor benötigt eine Versorgungsspannung von 5 V und hat die Abmessungen von 26 x 15,5 x 11 mm (vgl. Freescale-Semiconductor, 2006, S. 2).

Vergleich und Auswahl

Da der $Atmel\ SAM3X8E$ nur über einen integrierten Temperatursensor verfügt und der $Freescale\ MPX4115A$ nur den Umgebungsdruck messen kann, werden diese kombiniert und mit dem $Adafruit\ BMP280$ Board verglichen (siehe Tabelle 4.5). Vergleicht man die Temperatursensoren, weist das $Adafruit\ BMP280$ Board einen höheren Messbereich bei gleichzeitig höherer Messgenauigkeit auf. Bei den Drucksensoren sind die Messbereiche identisch, die Messgenauigkeit fällt allerdings beim $BMP280$ Board ebenfalls höher aus. Bei der Schnittstelle weisen der integrierte Temperatursensor im $SAM3X8E$ und der $MPX4115A$-Sensor mit einem analogen Ausgang eine schnellere Schnittstelle auf. Die Abmessungen und die Versorgungsspannungen sind bei den Sensoren identisch, wobei man für den $SMA3X8E$-Temperatursensor keine Angaben aus dem Datenblatt entnehmbar sind, da dieser auf dem Prozessor integriert ist.

Eigenschaft	**Atmel SMA3X8E**	**Adafruit BMP280**	**Freescale MPX4115A**
Temperatur Messbereich	0 - 80 °C	-40 - 85 °C	- - -
Temperatur Genauigkeit	± 3 °C	0,01 °C	- - -
Druck Messbereich	- - -	30 - 110 kPa	15 - 115 kPa
Druck Genauigkeit	- - -	0,16 kPa	± 1,5 %
Schnittstelle	analoger Ausgang	I2C, SPI	analoger Ausgang
Versorgungsspannung	- - -	3 - 5 V	5 V
Abmessungen	- - -	19 x 17 x 11 mm	26 x 15,5 x 5,5 mm

Tabelle 4.5.: Gegenüberstellung der Temperatur- und Drucksensoren

Das $Adafruit\ BMP280$ Board weist bei den Messbereichen und den Messgenauigkeiten bessere Werte auf. Da der Temperatursensor des $SAM3X8E$ auf dem Mikrocontroller

integriert ist, das heißt im Gehäuse des Mikrocontrollers sitzt, und somit keinen direkten Kontakt zur Umgebungsluft besitzt, ist dieser zur Messung der Umgebungsluft ungeeignet. Des Weiteren wird das Messergebnis bei dem Temperatursensor des $SAM3X8E$ durch die Abwärme des Mikrocontrollers beeinflusst. Daraus und das sich der Sensor im modularen Datenlogger bewährt hat (vgl. Schöne, 2017, S. 39), wird das $BMP280$ Board ausgewählt.

5. Umsetzung

Das Kapitel Umsetzung befasst sich mit der Implementierung des Entwicklungssystems. Dazu wird zunächst das Entwicklungssystem umgesetzt. Anschließend erfolgt die Entwicklung des Ultraschallsensors und der Datenaufnahme.

5.1. Entwicklungssystem

Die Umsetzung des Entwicklungssystems betrifft sowohl die Mehrfachanordnung der Ultraschallsensoren als auch die Ethernet-Adressen und -Pakete des Kommunikationsnetzwerks.

5.1.1. Mehrfachanordnung des Entwicklungssystems

Als Topologie für das Entwicklungssystem wird eine Sterntopologie gewählt (siehe Abbildung 5.1). Dies geschieht mit einem Switch als Sternpunkt in Ethernet-Netzwerk. Die Sterntopologie hat den Vorteil, dass es hierdurch zu keinen Datenkollisionen auf den Ethernet-Leitungen zwischen den einzelnen Ultraschallsensoren kommen kann. Ein weiterer Vorteil ist, dass die Daten der Ultraschallsensoren gegebenenfalls auf dem Switch zwischengespeichert werden können.

5.1.2. Ethernet-Adressen

Die IP-Adresse des Netzwerks wird auf 192.168.0.0 festgelegt. Dabei wird sich an der IANA-Richtlinie für private Netzwerke orientiert (vgl. IANA, 1996). Die Subnetzmaske ist dabei 255.255.0.0. Die IP-Adresse des Rechners, auf dem die Datenaufnahme ausgeführt wird, wie folgt berechnet:

$$IP_{Rechner} = 192.168.(0+1).(0+1) \tag{5.1}$$

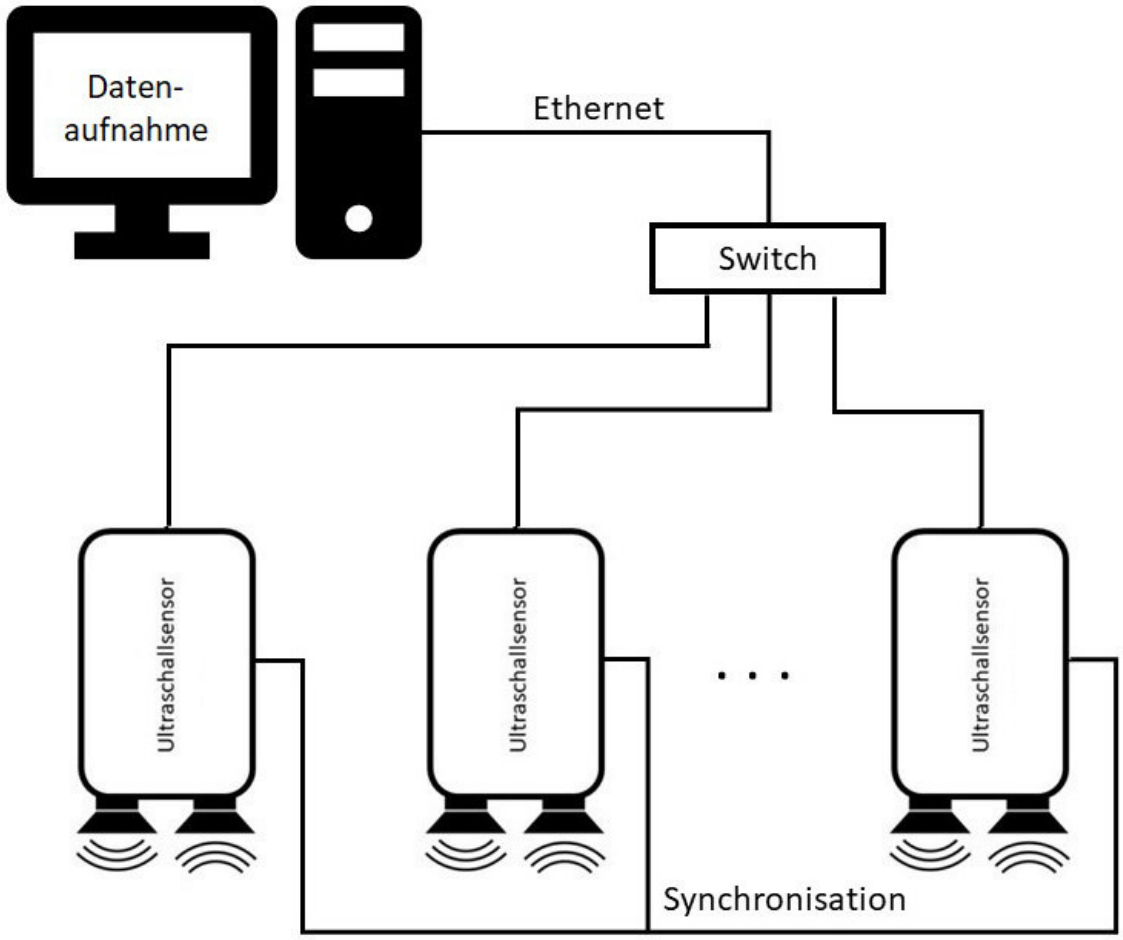


Abbildung 5.1.: Entwicklungssystem in Mehrfachanordnung

Die IP-Adressen für die Ultraschallsensoren berechnen sich aus:

$$IP_{US-Sensor,n} = 192.168.(0+1).(0+1+n) \tag{5.2}$$

Die Ethernet-Port-Basis ist vom Benutzer des Entwicklungssystems individuell einstellbar. Für das Entwicklungssystem werden nur vier Ports benötigt, da die Daten der einzelnen Ultraschallsensoren durch die IP-Adresse voneinander unterscheidbar sind. Der erste Port ist für den Rechner, von dem die Anweisungen an die Ultraschallsensoren gesendet werden. Der zweite Port ist für die Ultraschallsensoren, zum Empfangen der Anweisungen beziehungsweise für den Rechner, an dem die empfangenen Echosignale gesendet werden. Die Temperatur- und Umgebungsdruck-Werte werden von den Ultraschallsensoren an den dritten Port gesendet. Des Weiteren soll es möglich sein, den Status der Ultraschallsensoren

abfragen zu können. Dazu werden die Status-Pakete der Ultraschallsensoren an den vierten Port gesendet. Die einzelnen Ports werden wie in Tabelle 5.1 vergeben.

Port	**Berechnung**
Sende Anweisung	Port-Basis
Empfang Anweisung	Port-Basis + 1
Echosignal	Port-Basis + 1
Temperatur und Umgebungsdruck	Port-Basis + 2
Status	Port-Basis + 3

Tabelle 5.1.: Berechnung der Ethernet-Ports

5.1.3. Ethernet-Pakete

Als Kommunikation zwischen der Datenaufnahme und den Ultraschallsensoren werden UDP-Telegramme verwendet (siehe Abbildung 5.2). Dabei werden vier voneinander unabhängige UDP-Pakete definiert, die zur Kommunikation zwischen der Datenaufnahme und den Ultraschallsensoren dienen. Dabei orientieren sich die Pakete an den in Tabelle 5.1 definierten Ethernet-Ports.

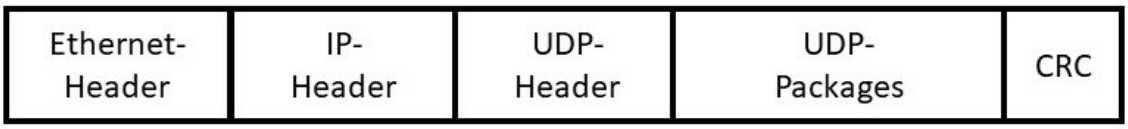


Abbildung 5.2.: Aufbau eines UDP-Telegramms (vgl. Reißenweber, 2009, S. 282)

Das Anweisungspaket für die Ultraschallsensoren aus 5 Byte (siehe Tabelle 5.2). Mit dem Paket werden Abfragen zum Status und den Umgebungsbedingungen ausgelöst. Der Messprozess wird über das Anweisungsprotokoll ebenfalls gestartet. Den Ultraschallsensoren wird darüber der Master- beziehungsweise Slave-Zustand übermittelt. Dem Master werden des Weiteren über das Paket die Parameter zu Sendefrequenz, Ultraschall-Puls-Amplitude, Sweep beziehungsweise konstanter Frequenz, Sende-Pulsdauer und Echoaufnahmezeit übergeben.

Das Paket für die Messwerte des Echosignals hat eine variable Paketlänge. Die maximale Paketlänge wird auf 1470 Byte gesetzt. In dem Paket repräsentieren je zwei Byte einen Abtastwert.

Bit	Funktion	Beschreibung
39	Status	1 - Abfrage 0 - keine Abfrage
38	Temperatur / Umgebungsdruck	1 - Abfrage 0 - keine Abfrage
37 $\sim$ 36	Master / Slave	1X - Master 01 - Slave
35	Sweep / Konstant	1 - Sweep-Signal 0 - konstante Frequenz
34 $\sim$ 31	Ultraschall-Puls-Amplitude	0 - 100 % , 16 Schritte
30 $\sim$ 27	Frei	
26 $\sim$ 20	Frequenz	in kHz, max. 127 kHz
19 $\sim$ 8	Sende-Pulsdauer	in μs, max. 4095 μs
7 $\sim$ 0	Echoaufnahmezeit	in ms, max. 255 ms

Tabelle 5.2.: Anweisungspaket an die Ultraschallsensoren

Die Temperatur und der Umgebungsdruck werden in einem UDP-Paket übermittelt. Das Paket besteht aus 32 Byte (siehe Tabelle 5.3). In dem Paket sind die Rohmesswerte der Temperatur und des Umgebungsdrucks enthalten. Des Weiteren werden mit dem Paket die Kalibrierwerte übermittelt (vgl. Bosch-Sensortec, 2018, S. 24).

Byte	Wert
31 $\sim$ 29	Rohmesswert Umgebungsdruck
28 $\sim$ 26	Rohmesswert Temperatur
25 $\sim$ 20	Kalibrierwert Temperatur
19 $\sim$ 0	Kalibrierwert Umgebungsdruck

Tabelle 5.3.: Paket für Temperatur und Umgebungsdruck

Abschließend wird das Status-Paket definiert (siehe Tabelle 5.4). Dieses besteht aus einem Byte und wird für die Statusabfrage verwendet. Des Weiteren wird das UDP-Paket am Ende eines Messvorgangs von jedem Ultraschallsensor an die Datenaufnahme gesendet.

Funktion	Paket
Statusabfrage	0xAA
Messungsende	0XF0

Tabelle 5.4.: Paket für Statusabfrage und Messungsende

5.2. Ultraschallsensor

Die Umsetzung des Ultraschallsensors erfolgt auf Grundlage des entwickelten Konzepts des Ultraschallsensors. Dazu wird zunächst die vorgeschlagene Architektur des Ultraschallsensors weiter definiert und Parameter, die den gesamten Ultraschallsensor betreffen, definiert. Im weiteren Verlauf der Umsetzung werden die analogen Verstärkerschaltungen und die Spannungsversorgung entwickelt. Anschließend wird die Software des Ultraschallsensors auf dem Mikrocontroller implementiert. Ein weiterer Punkt ist die Implementierung eines Programms zur automatischen Erstellung von Look-Up-Tabellen.

5.2.1. Vertiefung der Ultraschallsensor-Architektur

Um das entwickelte Konzept des Ultraschallsensors umzusetzen, ist es zunächst nötig, die Architektur des Ultraschallsensors weiter zu definieren (siehe Abbildung 5.3). Dazu werden die einzelnen Komponenten in die Architektur integriert. Des Weiteren werden die einzelnen Schnittstellen zwischen den Komponenten definiert.

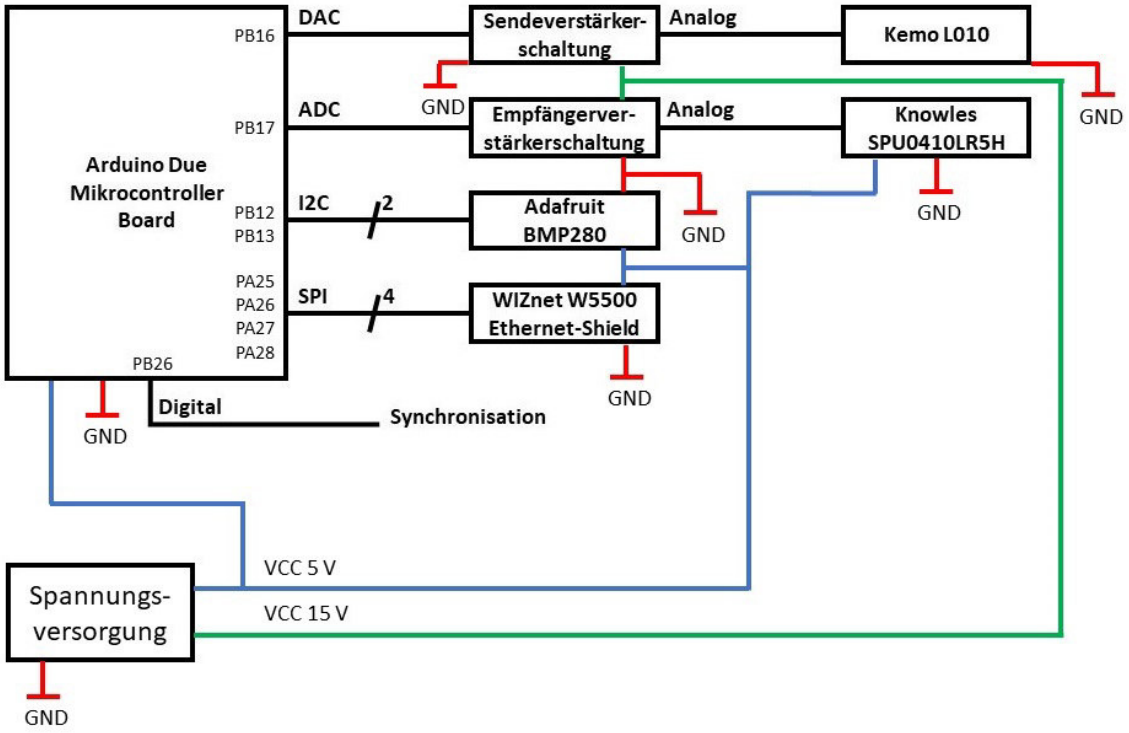


Abbildung 5.3.: Definierte Architektur des Ultraschallsensors

Das Ethernet-Shield wird über SPI vom Mikrocontroller-Board gesteuert. Die Auswahl der Schnittstelle für den Druck- und Temperatursensor fällt auf I2C, da die zwei SPI-Leitungen des $Arduino\ Due$ vom Ethernet-Shield belegt werden. Die Synchronisation zwischen den Ultraschallsensoren wird über einen digitalen Pin gelöst. Dabei sind alle Ultraschallsensoren über ein Kabel verbunden. Die Schnittstelle zum Aussenden der Ultraschall-Pulse wird auf einen DAC-Pin gelegt, um den Ultraschall-Lautsprecher mit einer variablen Amplitude ansteuern zu können. Um die Ultraschallechos aufzeichnen zu können, wird das MEMS-Mikrofon auf einen ADC-Pin gelegt.

Da der Ultraschalllautsprecher mit einer anderen Spannung versorgt werden muss, als das Mikrocontroller-Board liefern kann, wird dazwischen noch eine Verstärkerschaltung gesetzt. Zwischen MEMS-Mikrofon und Mikrocontroller-Board wird ebenfalls eine Verstärkerschaltung gesetzt, um die Amplitude des Ultraschallechos an den Mikrocontroller anzupassen.

Des Weiteren wird die Architektur des Ultraschallsensors durch eine Spannungsversorgung erweitert, mit der die einzelnen Komponenten mit Spannung versorgt werden. Hierzu werden zwei unterschiedliche Spannungen beziehungsweise ein Masseanschluss bereitgestellt.

5.2.2. Parameterdefinition

Zu den zu definierenden Parametern gehören der Arbeitsbereich, die Abtastraten und der Ultraschall-Sendepuls. Um unterschiedliche Sendefrequenzen und die Anforderung einer hohen Frequenzbandbreite zu erfüllen, wird der Arbeitsbereich des Ultraschallsensors auf 30 kHz bis 60 kHz festgelegt.

Die Abtastrate des Echosignals muss einerseits die geforderte Genauigkeit des Ultraschallsensors erfüllen und andererseits das Nyquist-Shannon-Abtasttheorem erfüllen. Aus der geforderten Genauigkeit und der Annahme von 30 °C Umgebungstemperatur ergibt sich eine Mindestabtastrate von:

$$f_s > \frac{c_{30\,°C}}{2\Delta x} > 17,47\ kHz \tag{5.3}$$

Um das Abtasttheorem zu erfüllen, wird der Frequenzbereich des $Knowles$ MEMS-Mikrofons als Bandbreite verwendet, da dieses die größte Bandbreite aufweist. Daraus folgt für die Abtastrate:

$$f_s \geq 2B_{Mikrofon} \geq 159,8\ kHz \tag{5.4}$$

Aufbauend auf den Berechnungen wird die Abtastrate des Echosignals auf 200 kHz festgelegt.

Um ein Aliasing auf dem Sendepfad leichter zu verhindern, wird die Abtastrate für den DAC höher ausgelegt. Dabei wird die Abtastrate auf 400 kHz festgelegt.

Als Ultraschall-Sendepulse werden ein Sinus- und ein Sweepsignal ausgewählt. Dabei ist das Sinussignal in der Frequenz und der Pulsdauer variable einstellbar. Da eine Berechnung eines Sweep-Signals rechenaufwendig ist, wird dieses als Look-Up-Tabelle auf den Ultraschallsensoren hinterlegt.

5.2.3. Sendeverstärkerschaltung

Da der DAC des *Atmel SAM3X8E* einen Spannungsbereich von 0,55 V bis 2,75 V (vgl. Atmel, 2015, S. 1412) besitzt und der *Kemo L010* mit einer Spannung von bis zu ± 15 V angesteuert wird, wird eine Verstärkerschaltung entwickelt (siehe Abbildung 5.4). Dabei muss die Verstärkerschaltung Bandpassverhalten aufweisen. Mit dem Bandpassverhalten wird der Gleichspannungsanteil rausgefiltert. Des Weiteren werden die hohen Frequenzen aus dem digitalen Signal des DAC rausgefiltert, um daraus einen Sinus zu erzeugen. Um eine hohe Bandbreite zu erhalten, wird der Bandpass aus einer Kombination von Operationsverstärkerschaltungen mit Tiefpass- und Hochpassverhalten erzeugt. Da der Ultraschall-Lautsprecher eine Kapazität von 150 nF besitzt und Operationsverstärker bei kapazitiven Lasten zum Schwingen neigen (vgl. Tietze u. a., 2016, S. 598), ist es nötig, diese zu kompensieren. Hierzu wird zwischen dem Bandpass und dem Ultraschalllautsprecher ein Reihenschwingkreis integriert. Im Anschluss an die Entwicklung der Sendeverstärkerschaltung erfolgt die Simulation der Schaltung mit einer Darstellung von Frequenz- und Phasengang.

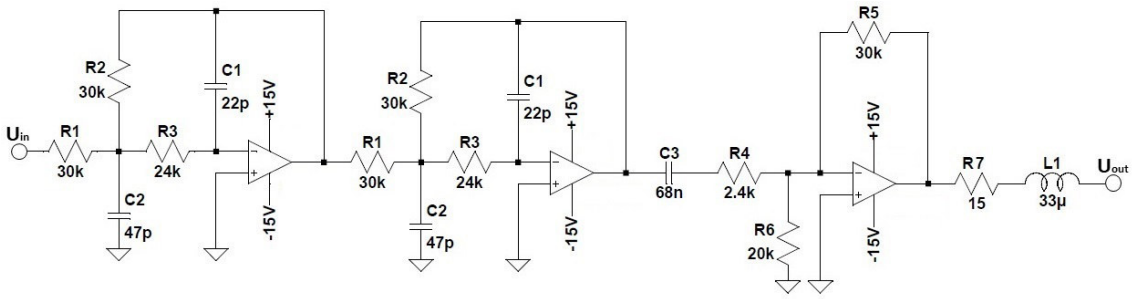


Abbildung 5.4.: Entwickelte Sendeverstärkerschaltung

Tiefpassschaltung

Um ein Aliasing zu verhindern, sollte das Signal bei der halben DAC-Abtastrate von 200 kHz eine möglichst hohe Dämpfung aufweisen. Dies wird durch einen Tiefpass vierter Ordnung mit einer Grenzfrequenz von $f_g = 80$ kHz realisiert. Der Tiefpass vierter Ordnung setzt sich dabei aus zwei identischen aktiven Tiefpässen mit Mehrfachgegenkopplung zusammen (vgl. Tietze u. a., 2016, S. 817).

Bei der Auslegung des Tiefpasses zweiter Ordnung werden die Grenzfrequenz f_g auf 80 kHz und die Verstärkung A_0 auf -1 gelegt. Die Kondensatoren werden mit $C_1 = 22$ pF und $C_2 = 47$ pF festgelegt.

Daraus lässt sich der Widerstand R_2 mit den Koeffizienten $a_1 = 0{,}87$ und $b_1 = 0{,}1892$ (vgl. Tietze u. a., 2016, S. 806) berechnen:

$$R_2 = \frac{a_1 C_2 - \sqrt{a_1^2 C_2^2 - 4 C_1 C_2 b_1 (1 - A_0)}}{4\pi f_g C_1 C_2} = 29,39\ k\Omega \approx 30\ k\Omega \tag{5.5}$$

Für den Widerstand R_1 gilt:

$$R_1 = \frac{R_2}{-A_0} = 30\ k\Omega \tag{5.6}$$

Die Berechnung des Widerstands R_3 ergibt:

$$R_3 = \frac{b_1}{4\pi^2 f_g^2 C_1 C_2 R_2} = 24,14\ k\Omega \approx 24\ k\Omega \tag{5.7}$$

Die aus den gerundeten Widerständen und den festgelegten Kondensatoren resultierende Grenzfrequenz liegt bei:

$$f_g = \sqrt{\frac{b_1}{4\pi^2 C_1 C_2 R_2 R_3}} = 80,23\ kHz \tag{5.8}$$

Hochpassschaltung

Als Hochpassschaltung wird ein aktiver Hochpass erster Ordnung mit einer Grenzfrequenz f_g = 1 kHz verwendet. Bei der Auslegung wird der Kondensator auf $C_3 = 68$ nF festgelegt.

Die nötige Verstärkung des Hochpasses berechnet sich aus der Amplitude des DAC mit $\hat{u}_{DAC} = 1{,}1$ V und der Amplitude des Ultraschalllautsprechers mit $\hat{u}_{ULS} = 15$ V:

$$A_\infty = -\frac{\hat{u}_{ULS}}{\hat{u}_{DAC}} = -13,64 \tag{5.9}$$

Aus der Grenzfrequenz, dem Kondensator und dem Koeffizienten $a_1 = 1$ (vgl. Tietze u. a., 2016, S. 806) lässt sich der Widerstand R_4 berechnen:

$$R_4 = \frac{1}{2\pi f_g a_1 C_3} = 2,34\ k\Omega \approx 2,4\ k\Omega \tag{5.10}$$

Bei der Berechnung des Widerstands R_5 wird dieser kleiner ausgelegt, um die Verstärkung

zu reduzieren, da die ± 15 V des Ultraschalllautsprechers Maximalwerte sind:

$$R_5 = -R_4 A_\infty = 32,74\ k\Omega \approx 30\ k\Omega \tag{5.11}$$

Die aus den berechneten Werten resultierende Grenzfrequenz liegt bei:

$$f_g = \frac{1}{2\pi a_1 R_4 C_3} = 975,21\ kHz \tag{5.12}$$

Die resultierende Verstärkung ergibt:

$$A_\infty = -\frac{R_5}{R_4} = 12,5 \tag{5.13}$$

Zur Stabilisierung des Operationsverstärkers wird der invertierende Eingang über einen Pull-Down-Widerstand R_6 mit Ground verbunden. Dazu wird ein 20 kΩ Widerstand gewählt.

Aus den Grenzfrequenzen der Tiefpassschaltung und der Hochpassschaltung resultiert für die Bandbreite der Sendeverstärkerschaltung:

$$B = f_{g,TP} - f_{g,HP} = 79,26\ kHz \tag{5.14}$$

Reihenschwingkreis

Um einerseits die Kapazität des Ultraschalllautsprechers zu kompensieren und andererseits das Frequenzverhalten der Sendeverstärkerschaltung im Arbeitsbereich nicht zu stark zu beeinflussen, wird die Resonanzfrequenz des RLC-Reihenschwingkreises auf 75 kHz gelegt. Die Kapazität des Ultraschalllautsprechers beträgt C_{ULS} = 150 nF.

Da sich die Impedanzen der Spule und des Ultraschalllautsprechers bei der Resonanzfrequenz aufheben, gilt:

$$L = \frac{1}{\omega^2 C_{ULS}} = 30,02\ \mu H \approx 33\ \mu H \tag{5.15}$$

Um im Resonanzfall keine Überhöhung der Spannung über dem Ultraschalllautsprecher zu erzeugen, wird die Kreisgüte Q auf 1 gesetzt. Daraus resultiert für den Widerstand R_7:

$$R_7 = \sqrt{\frac{L}{C_{ULS}}}\frac{1}{Q} = 14,83\Omega \approx 15\Omega \tag{5.16}$$

Daraus resultiert die Resonanzfrequenz:

$$f_r = \sqrt{\frac{1}{LC_{ULS}}}\frac{1}{2\pi} = 71,53\ kHz \tag{5.17}$$

Für die Kreisfrequenz resultiert:

$$Q = \sqrt{\frac{L}{C_{ULS}}} \frac{1}{R_7} = 0,98 \tag{5.18}$$

Gesamte Sendeverstärkerschaltung

Die Versorgungsspannung der gesamten Sendeverstärkerschaltung wird auf ± 15 V gesetzt. Als Operationsverstärker für die Tiefpassschaltung werden $TI\ LF353P$ verwendet. Für die Hochpassschaltung wird der $TI\ LM675$ verwendet, welcher den für die Ultraschalllautsprecher benötigten Strom liefern kann.

Abschließend wird der Frequenzgang der gesamten entwickelten Sendeverstärkerschaltung mit $LTSpice$ simuliert. Die simulierten Frequenzen liegen im Bereich von 100 Hz bis 200 kHz (siehe Abbildung 5.5). Es ist zu sehen, dass niedrige Frequenzen rausgefiltert werden. Des Weiteren weisen Frequenzen über 200 kHz eine hohe Dämpfung auf. Dies und die Bandbreite, welche deutlich kleiner als die halbe DAC-Abtastrate ist, verhindert ein Aliasing. Der Bandpass besitzt im Arbeitsbereich eine 12,5-fache Verstärkung. Im Bereich von 40 kHz bis 50 kHz wird die Verstärkung durch den Reihenschwingkreis etwas angehoben.

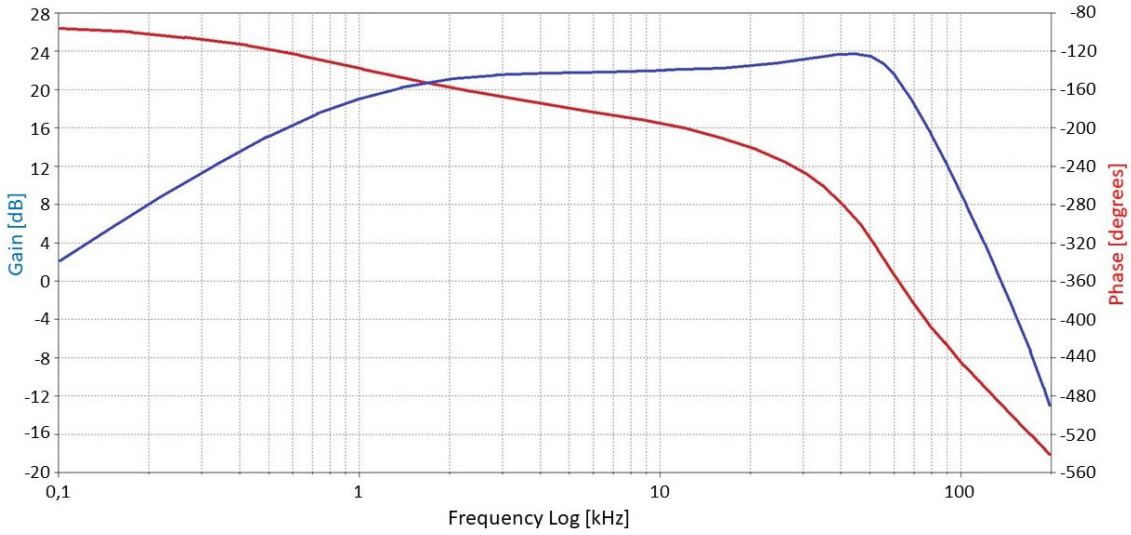


Abbildung 5.5.: Frequenzgang der entwickelten Sendeverstärkerschaltung

5.2.4. Empfängerverstärkerschalung

Das Ausgangssignal des MEMS-Mikrofons hat eine Amplitude bei Ultraschallechos von maximal 8 mV bei einer Offsetspannung von 0,73 V. Da der ADC des Mikrocontrollers Spannungswerte im Bereich von 0 V bis 3,3 V Abtasten kann, wird das Ausgangssignal des MEMS-Mikrofons auf diesen Wert normiert. Dazu wird das Echosignal zuerst mit einem Tiefpass gefiltert (siehe Abbildung 5.6). Anschließend wird das Echosignal durch zwei Verstärkerstufen verstärkt, wobei die Verstärkung auf beide Stufen gleichmäßig aufgeteilt wird. Bei den Verstärkerstufen handelt es sich um einen nicht invertierenden Verstärker und einen Hochpass. Des Weiteren muss die Bandbreite der Empfängerverstärkerschaltung kleiner als die halbe ADC-Abtastrate von 100 kHz sein, wodurch ein Aliasing vermieden wird. Im Anschluss an die Entwicklung der Empfängerverstärkerschaltung erfolgt die Simulation der Schaltung mit einer Darstellung von Frequenz- und Phasengang.

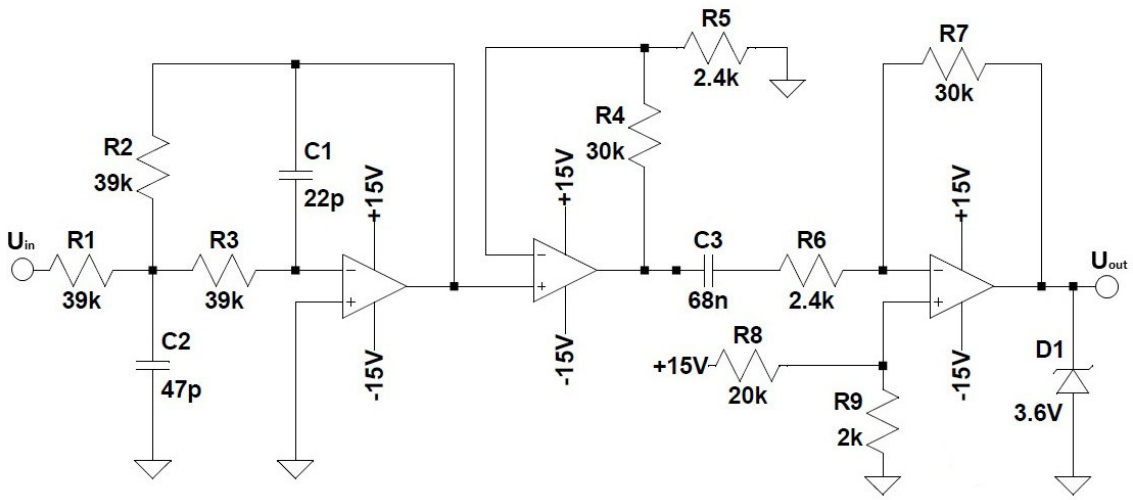


Abbildung 5.6.: Entwickelte Empfängerverstärkerschaltung

Tiefpassschaltung

Als Tiefpass wird ein aktiver Tiefpass zweiter Ordnung mit Mehrfachgegenkopplung und gewählt. Dabei wird die Grenzfrequenz auf 80 kHz gelegt, welche etwas über dem Arbeitsbereich des Ultraschallsensors liegt. Die Kondensatoren werden mit $C_1 = 22\,\text{pF}$ und $C_2 = 47\,\text{pF}$ festgelegt. Die Verstärkung A_0 beträgt -1. Mit Formel 5.5 du den Koeffizienten $a_1 = 1{,}2872$ und $b_1 = 0{,}4141$ (vgl. Tietze u. a., 2016, S. 806) lässt sich der Widerstand R_2 berechnen:

$$R_2 = 43,49\,k\Omega \approx 39\,k\Omega \tag{5.19}$$

Daraus und mit der Formel 5.6 berechnet sich der Widerstand R_1:

$$R_1 = 39\ k\Omega \tag{5.20}$$

Für den Widerstand R_3 ergibt sich nach Formel 5.7:

$$R_3 = 40,65\ k\Omega \approx 39\ k\Omega \tag{5.21}$$

Die für den Tiefpass resultierende Grenzfrequenz nach Formel 5.8:

$$f_g = 81,68\ kHz \tag{5.22}$$

Nicht invertierender Verstärker

Als erste Verstärkerstufe wird ein nicht invertierender Operationsverstärker gewählt. Der Widerstand R_5 wird auf 2,4 kΩ festgelegt. Für die Verstärkung der gesamten Empfängerverstärkerschaltung gilt:

$$A_{Ges} = \frac{\hat{u}_{out}}{\hat{u}_{in}} = 187,5 \tag{5.23}$$

Um die Verstärkung auf beide Verstärkerstufen aufzuteilen, wird diese gleichmäßig verteilt. Daraus folgt für die Verstärkung des nicht invertierender Verstärkers A_{NI}:

$$A_{NI} = \sqrt{A_{Ges}} = 13,69 \tag{5.24}$$

Daraus lässt sich der Widerstand R_4 berechnen:

$$R_4 = (A_{NI} - 1)R_5 = 30,46\ k\Omega \approx 30\ k\Omega \tag{5.25}$$

Die daraus resultierende tatsächliche Verstärkung beträgt:

$$A_{NI} = 1 + \frac{R_4}{R_5} = 13,5 \tag{5.26}$$

Hochpassschaltung

Mit der Hochpassschaltung wird die Amplitude des Echosignals an den Spannungsbereich des ADC angepasst. Hierzu wird ein aktiver Hochpass erster Ordnung verwendet. Damit das verstärkte Echosignal eine feste Offsetspannung besitzt, wird an den nicht invertierenden Eingang des Operationsverstärkers eine Spannung angelegt. Dazu werden die Amplitude und die Offsetspannung des Echosignals auf 1,5 V festgelegt. Somit wird gewährleistet, dass sich das Echosignal im Spannungsbereich des ADC bewegt.

Die Verstärkung des Hochpasses resultiert aus der Gesamtverstärkung, der Verstärkung des Tiefpasses und der Verstärkung des nicht invertierenden Verstärkers:

$$A_{\infty} = \frac{A_{Ges}}{A_0 A_{NI}} = -13,89 \tag{5.27}$$

Die Grenzfrequenz des Hochpasses wird auf 1 kHz festgelegt. Da der Hochpass aus Abschnitt 5.2.3 eine Verstärkung besitzt, die näherungsweise der benötigten entspricht, wird dieser verwendet und um eine Offsetspannung erweitert. Dadurch ergeben sich für die Widerst2ande die Werte $R_6 = 2{,}4\,\text{k}\Omega$ und $R_7 = 30\,\text{k}\Omega$.

Um die Offsetspannung von 1,5 V zu erzeugen, werden 15 V der Versorgungsspannung der Sendeverstärkerschaltung verwendet. Der Widerstand vor dem nicht invertierenden Eingang wird auf $R_8 = 20\,\text{k}\Omega$ festgelegt. Daraus resultiert für den Widerstand R_9:

$$R_9 = \frac{R_5 U_{VCC}}{U_{NI}} = 2\ k\Omega \tag{5.28}$$

Damit resultiert für die Gesamtverstärkung:

$$A_{Ges} = A_0 A_{NI} A_{\infty} = 168,75 \tag{5.29}$$

Des Weiteren resultiert aus den Grenzfrequenzen des Tiefpasses und des Hochpasses die Bandbreite:

$$B = f_{g,TP} - f_{g,HP} = 80,71\ kHz \tag{5.30}$$

Gesamte Empfängerverstärkerschaltung

Für die Empfängerverstärkerschaltung wird die gleiche Versorgungsspannung wie für die Sendverstärkerschaltung gewählt. Als Operationsverstärker für die Empfängerverstärkerschaltung werden $TI\ LF353P$ verwendet. Um den ADC des Mikrocontrollers vor zu großen Spannungen zu schützen, wird an U_out der Verstärkerschaltung eine Z-Diode mit einer Durchbruchspannung von 3,6 V gesetzt.

In Abbildung 5.7 sieht man den mit $LTSpice$ simulierten Frequenzgang der Empfängerverstärkerschaltung. Dort ist zu erkennen, dass die Frequenzen im Arbeitsbereich des Ultraschallsensors verstärkt werden. Die Frequenzen unterhalb und oberhalb des Arbeitsbereichs werden hingegen gedämpft. Des Weiteren ist die Bandbreite der Empfängerverstärkerschaltung deutlich kleiner als die halbe ADC-Abtastrate.

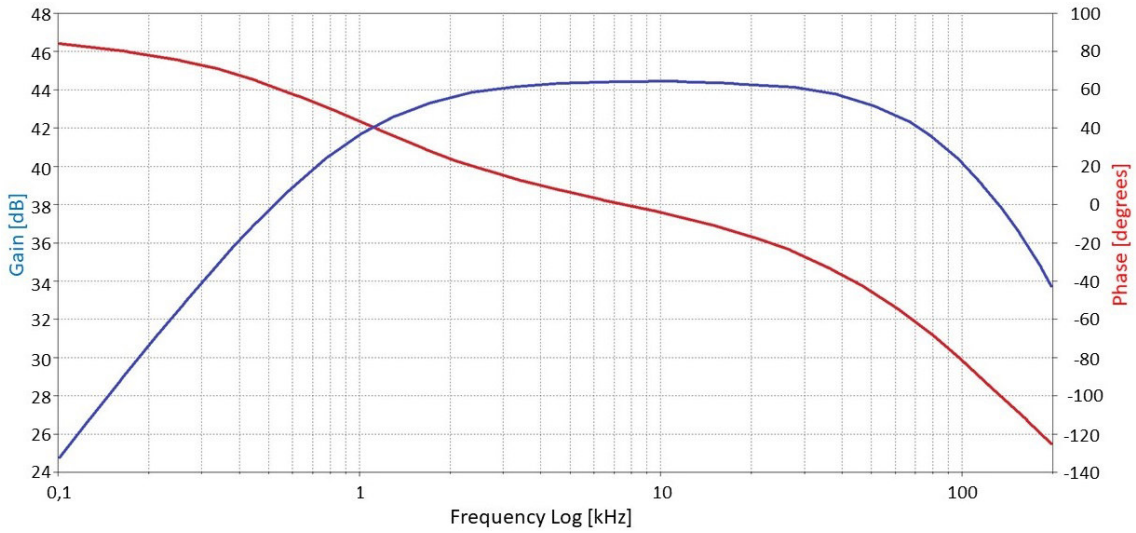


Abbildung 5.7.: Frequenzgang der entwickelten Empfängerverstärkerschaltung

5.2.5. Spannungsversorgung

Die Spannungsversorgung stellt für die einzelnen Komponenten des Ultraschallsensors die Betriebsspannung bereit. Dabei werden für die Verstärkerschaltungen jeweils $\pm 15\,\mathrm{V}$ benötigt. Für das Mikrocontrollerboard werden 5 V benötigt (siehe Abbildung 5.8).

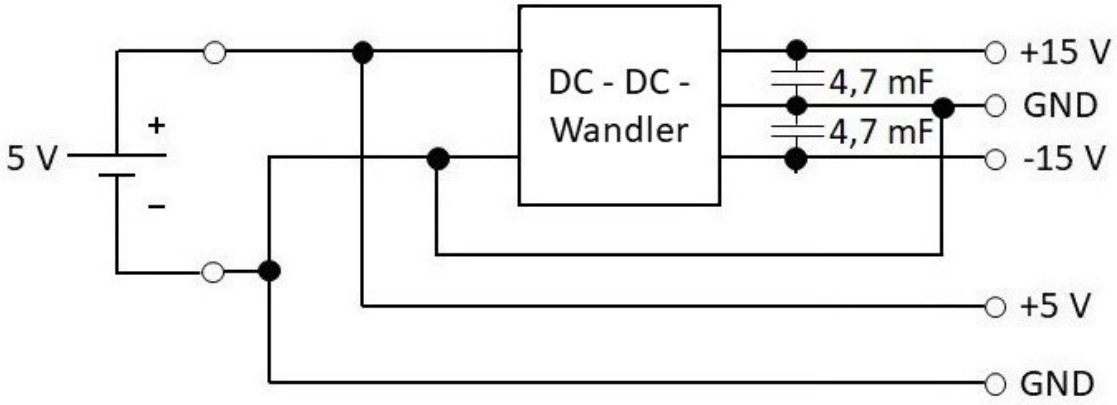


Abbildung 5.8.: Entwickelte Spannungsversorgung des Ultraschallsensors

Als Spannungsquelle wird ein 5 V Akku gewählt. Dadurch kann das Mikrocontrollerboard direkt von diesem versorgt werden. Um die $\pm 15\,\mathrm{V}$ zu erzeugen, wird ein DC-DC-Wandler verwendet, welche als Eingangsspannung die 5 V vom Akku erhält.

Da der Ultraschalllautsprecher kurzfristig bis 300 mA Strom benötigt, werden nach dem DC-

DC-Wandler Glättungskondensatoren integriert. Dabei wird für den Ultraschall-Sendepuls eine Dauer von 1 ms angenommen. Der dabei maximal zulässige Spannungsabfall wird auf 0,1 V festgelegt. Daraus folgt für die Kapazität C des Kondensators:

$$C = \frac{\hat{i}\Delta t}{\Delta U} = 3\ mF \approx 4,7\ mF \tag{5.31}$$

Als DC-DC-Wandler wird der $XP\ Power\ JHM10$ verwendet, welcher den benötigten Strom liefern kann. Da der DC-DC-Wandler unter Volllast nach Datenblatt einen Maximalstrom auf der Eingangsseite von 2,67 A benötigt (vgl. XP-Power, 2014, S. 2) und um den Ausgangsstrom nicht zu beeinflussen, wird der Lithium-Polymer-Akku $Intenso\ HC20000$ verwendet, welcher einen Strom von bis zu 3 A liefern kann.

5.2.6. Software

Die Programmierung des Ultraschallsensors erfolgt in $Embedded\text{-}C$. Dabei wird auf das $Atmel\ Software\ Framework$ (ASF) verwendet, welches Softwareroutinen beinhaltet, mit denen die einzelnen Schnittstellen und Module des Mikrocontrollers angesprochen werden können (vgl. Atmel, 2018).

Die Realisierung der Software wird erfolgt mit einer interruptbasierten Struktur und drei Haupt-Modes (siehe Abbildung 5.9 und Anhang C.1). Bei den Haupt-Modes handelt es sich um den Main-Mode, dem Master-Mode und dem Slave-Mode. Der Main-Mode ist dabei für die Initialisierung des Systems zuständig (siehe Anhang A.1). Des Weiteren ist der Main-Mode für die Systemsteuerung zuständig. Hauptbestandteil des Main-Modes ist der Ethernet-Interrupt-Handler.

Mit dem Master-Mode wird der Messdurchlauf gesteuert (siehe Anhang A.2). Dabei ist unter den Ultraschallsensoren jeweils nur ein Master vorhanden. Der Master konfiguriert zunächst den Ultraschall-Sendepuls und die Timer. Um den Ultraschallsensoren genügend Zeit entpacken der UDP-Anweisungspakete und dem Wechsel des Modes zu geben, wartet der Master 100 ms. Zu Beginn des Messdurchlaufs setzt der Master die Synchronisationsleitung auf High und sendet den Sendepuls aus. Bei einem Interrupt des Burst-Timers wechselt der Master vom Aussenden des Sendepulses zum Empfangen der Ultraschallechos. Das Ende des Messdurchlaufs wird durch den Interrupt des Echo-Timers bestimmt, nachdem der Master die Synchronisationsleitung auf Low setzt. Dabei werden ebenfalls der DAC und der ADC deaktiviert.

Im Slave-Mode wartet der Slave zunächst auf den Interrupt der Synchronisationsleitung (siehe Anhang A.3). Bei einem Interrupt wird der ADC aktiviert, mit dem die Ultraschallechos empfangen werden. Die empfangenen Ultraschallechos werden auf dem Ethernet-Modul

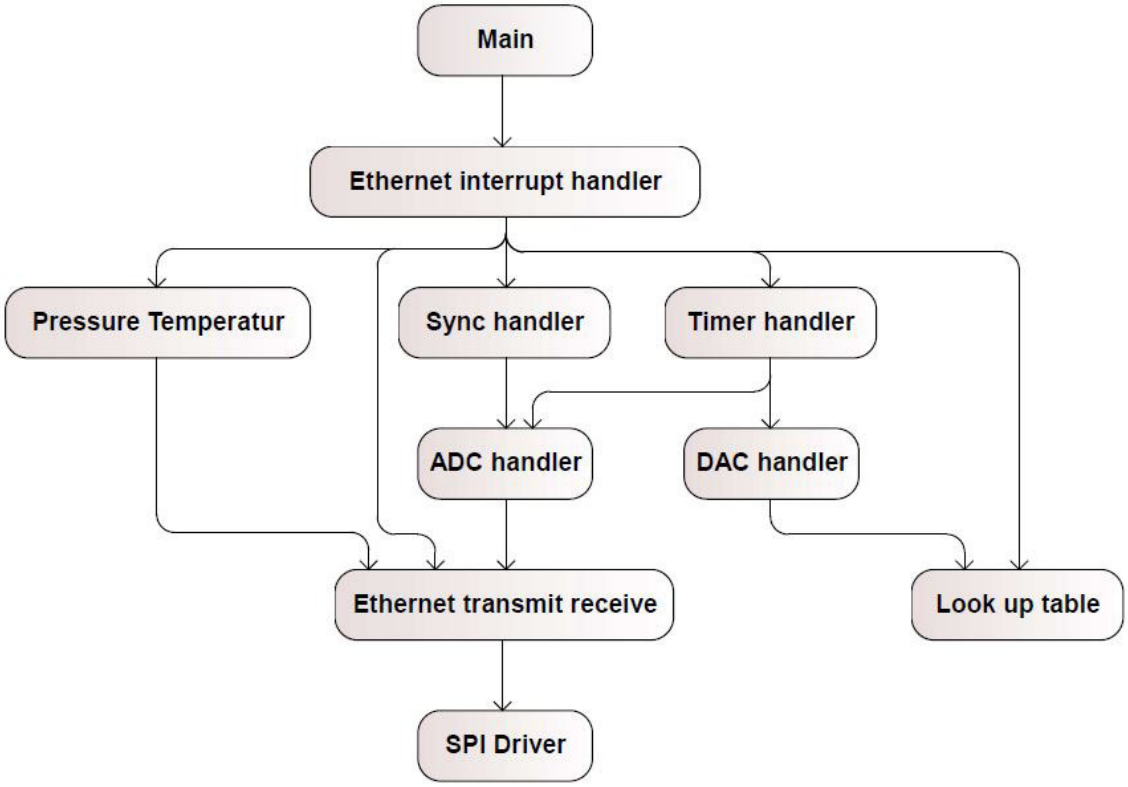


Abbildung 5.9.: Struktur der Ultraschallsensor Software

zwischengespeichert und bei Erreichen der maximalen UDP-Paketlänge an die Datenaufnahme gesendet. Bei einem erneuten Interrupt der Synchronisationsleitung wird der ADC deaktiviert und die restlichen Ultraschallechodaten auf dem Ethernet-Modul versendet.

Ethernet-Interrupt-Handler

Der Ethernet-Interrupt-Handler wird bei einem Interrupt des Ethernet-Moduls ausgelöst. Dabei wird zunächst das Anweisungspaket der Datenaufnahme aus dem Receive-Register des Ethernet-Moduls geholt und aufgeschlüsselt. Die Struktur des Ethernet-Interrupt-Handlers wird als if-else-Schleife realisiert, bei der die einzelnen Bits des Anweisungspakets chronologisch abgefragt werden. Der Handler löst eine Antwort auf eine Statusabfrage aus, beziehungsweise startet eine Messung der Umgebungsbedingungen wie Temperatur und Umgebungsdruck. Des Weiteren setzt der Handler den Master- beziehungsweise Slave-Mode. Bei einem Anweisungspaket ohne Anweisung wird der Handler beendet. Bei einer Zuteilung des Master-Zustandes werden im Handler der Ultraschall-Sendepuls und die Timer konfiguriert.

Timer-Handler

Zur Steuerung des Messdurchlaufs verfügt die Software des Ultraschallsensors über drei Timer-Handler. Der erste Timer ist für das Warten des Masters zuständig und startet bei einem Interrupt den DAC und setzt die Synchronisationsleitung für die Slaves auf High. Der zweite Timer kontrolliert den Ultraschall-Sendepuls, indem der DAC bei einem Interrupt deaktiviert und der ADC aktiviert wird. Der dritte Timer-Handler beendet den Messdurchlauf bei einem Interrupt. Dies geschieht, indem der ADC des Masters deaktiviert wird und die Synchronisationsleitung auf Low gesetzt wird.

Sync-Handler

Der Sync-Handler dient zur Steuerung der Slaves. Ist der Interrupt der Synchronisationsleitung aktiviert, startet der Handler bei einem Interrupt den ADC, um die empfangenen Echosignale abzutasten. Bei einem weiteren Interrupt deaktiviert der Handler den ADC und sendet die restlichen Ultraschallechodaten auf dem Ethernet-Modul an die Datenaufnahme.

ADC- / DAC-Handler

Das Abtasten der empfangenen Ultraschallechos erfolgt mit einem timergesteuerten ADC. Hierbei greift der ADC-Handler direkt auf das ADC-Register zu und übermittelt die Werte an das Ethernet-Modul.

Der DAC wird über den SysTick-Handler. Hierbei werden je nach Sendeburst die Werte für das digitale Signal aus der Look-Up-Tabelle für ein Sweep- oder Sinus-Signal entnommen. Die Look-Up-Tabelle für den Sinus wird dabei so ausgelegt, dass mit dieser mehrere Frequenzen möglich sind. Die Auflösung der möglichen Frequenzen wird auf 1 kHz gelegt. Formel 5.32 stellt die Beziehung zwischen der Abtastrate f_s des DAC, der Frequenz f des Sendepulses, der Werte n der Look-Up-Tabelle und der Schrittweite x für die Frequenz dar.

$$\frac{n}{x} = \frac{f_s}{f} \tag{5.32}$$

Daraus folgt zum Beispiel für eine Abtastrate von 400 kHz des DAC, dass die Look-Up-Tabelle aus 400 Werten besteht. Für eine Frequenz von 45 kHz ist dabei eine Schrittweite von $x = 45$ nötig. Um eine höhere Auflösung der Frequenzen zu erreichen, ist es möglich, die Look-Up-Tabelle zu vergrößern.

Temperatur- und Umgebungsdruck-Funktion

Die Funktion für das Erfassen für Temperatur und Umgebungsdruck steuert über I2C den $BMP280$-Sensor. Dazu wird die $BMP280$-Bibliothek verwendet (vgl. Bosch-Sensortec, 2017). Die Funktion liest zunächst die auf dem Sensor hinterlegten Kalibrierwerte aus. Anschließend wird eine Messung er Temperatur und des Umgebungsdrucks ausgelöst. Die gesamten Daten werden abschließend über Ethernet an die Datenaufnahme gesendet. Eine Berechnung der Werte erfolgt in der Datenaufnahme, da eine 64-bit-Architektur für genauere Ergebnisse von Vorteil ist (vgl. Bosch-Sensortec, 2018, S. 22).

Ethernet-Kommunikation

Die Ethernet-Kommunikation wird von drei Funktionen übernommen, welche auf den SPI-Treiber zugreifen. Die Kommunikation mit dem Ethernet-Modul geschieht dabei mit den $W5500$-Bibliotheken (vgl. WIZnet, 2017). Um die Anweisungspakete der Datenaufnahme zu erhalten, greift die Receive-Data-Funktion auf den Empfänger-Socket des Ethernet-Moduls zu. Die Übertragung von Daten wird in zwei Funktionen aufgeteilt. Daten für eine Statusmeldung des Ultraschallsensors und die Übertragung der Umgebungsdaten wird über Transmit-Data realisiert, bei der die Daten nach dem Transfer auf das Ethernet-Modul an die Datenaufnahme versendet werden. Die Funktion Transmit-Echo-Data speichert die Abtastwerte des ADC auf dem Ethernet-Shield zwischen und versendet diese erst bei Erreichen der maximalen UDP-Paketlänge.

SPI-Treiber

Zur Kommunikation zwischen dem Mikrocontroller und dem Ethernet-Modul wird der SPI-Treiber verwendet. Auf die Verwendung der SPI-Routine des ASF wird verzichtet, da mit dieser maximal zwei Byte übertragen werden können. Eine höhere Anzahl an Bytes wäre möglich, indem das Chip-Select-Signal aktiv kontrolliert wird. Dies würde aber zu einer Erhöhung der Übertragungsdauer führen.

Eine andere Möglichkeit wäre die Auslagerung der SPI-Kommunikation in den Peripheral DMA Controller (PDC), welcher die Daten ohne Belastung des Prozessors übertragen würde. Eine Verwendung des PDC ist in Kombination mit dem Ethernet-Modul nicht möglich, da die SPI-Schnittstelle mit der das Ethernet-Modul verbunden ist, nicht vom PDC unterstützt wird.

Daraus folgt, dass ein flexibler SPI-Treiber realisiert werden muss. Dieser wird in zwei Routinen zum Senden beziehungsweise Empfangen aufgeteilt. Um beim Sendevorgang möglichst schnell die Daten vom Mikrocontroller auf das Ethernet-Modul zu bekommen und eine Einhaltung der Abtastzeiten des ADC zu gewährleisten, wird Sende-Routine so realisiert, dass

diese nur auf das Transmit-Register zugreift. Dabei werden die Bytes einzeln in das Register geladen, bis alle übertragen wurden. Die Taktfrequenz beim Senden beträgt 28 MHz. Die nächstmögliche Taktfrequenz ist 42 MHz, welche aber bei 84 MHz Mikrocontrollertakt zu hoch ist.

Die zweite Routine zum Empfangen von Daten wird identisch realisiert. Zusätzlich greift die Routine auf das Receive-Register zu. Durch das Empfangen kommt es bei einer Taktfrequenz von 28 MHz zu Unterbrechungen,wodurch die Daten die Daten vom Programm nicht mehr verarbeitet werden können. Um dies zu vermeiden, wird die Taktfrequenz auf 21 MHz reduziert, wodurch ein sicherer Übertragungsprozess gewährleistet wird. Ein Vergleich der beiden Taktfrequenzen ist in Abbildung 5.10 zu sehen, welche mit dem *PicoScope* aufgenommen wurden.

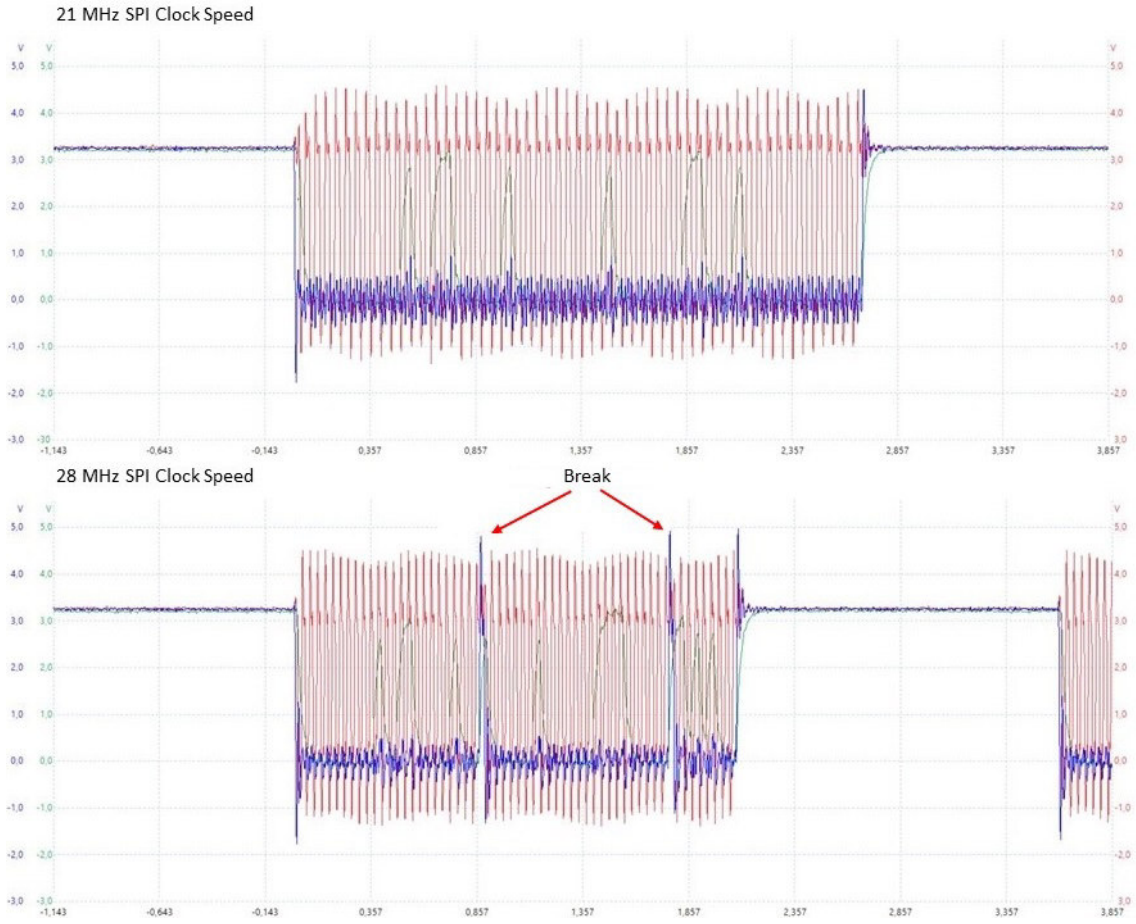


Abbildung 5.10.: Vergleich von 21 MHz und 28 MHz SPI-Taktfrequenz der Empfangs-Routine

5.2.7. Look-Up-Tabellen Generierung

Zur automatischen Generierung der Header-Datei, welche die Look-Up-Tabellen für das Sweep-Signal und den Sinus beinhaltet, wird ein Programm realisiert (siehe Anhang C.2). Des Weiteren werden über die Header-Datei die Abtastraten des Ultraschallsensors definiert. Als Programmiersprachewird Python benutzt. Als Eingabeoberfläche wird ebenfalls eine GUI integriert.

Die Eingabewerte des Programms sind die Abtastraten des ADC und de DAC, die Start- und Endfrequenz des Sweep-Signals, sowie die Dauer des Sweep-Signals. Mithilfe der Eingabewerte werden die Look-Up-Tabellen berechnet. Abschließend werden die Look-Up-Tabellen und die Abtastraten in einer Header-Datei gespeichert, welche in die Ultraschallsensor-Software eingebunden werden kann.

5.3. Datenaufnahme

Die Umsetzung der Datenaufnahme erflogt auf Grundlage der Anforderungen und dem entwickelten Ultraschallsensor. Dazu wird zunächst eine Struktur der Software definiert. Anschließend erfolgt die Umsetzung der einzelnen Softwarekomponenten. Als Programmiersprache wird für die Datenaufnahme Python verwendet, da *Python* unter den meisten *Linux*-Distributionen im Standardumfang enthalten ist (vgl. Wikipedia, 2018).

5.3.1. Struktur der Datenaufnahme

Als Struktur für die Datenaufnahme wird eine objektorientierte Unterteilung in vier Klassen vorgesehen (siehe Abbildung 5.11). Dabei wird jede Klasse in ein eigenes Modul ausgelagert. Ein weiteres Modul dient der Definition der Basisdaten des Ethernet-Netzwerkes.

Die Hauptklasse ist in der Struktur mainwinow (siehe Anhang D). Dabei steuert mainwindow die Datenaufnahme. Die Berechnung der Umgebungsbedingungen, wie Temperatur und Umgebungsdruck erfolgt in der Klasse calculate_pressure_temperature. Zur Analyse der Echodaten wird die Klasse analyze_data verwendet. Die Kommunikation mit den Ultraschallsensoren wird von etherent_network realisiert. Die Basisdaten des Ethernet-Netzwerks werden in ethernet_network_setup definiert.

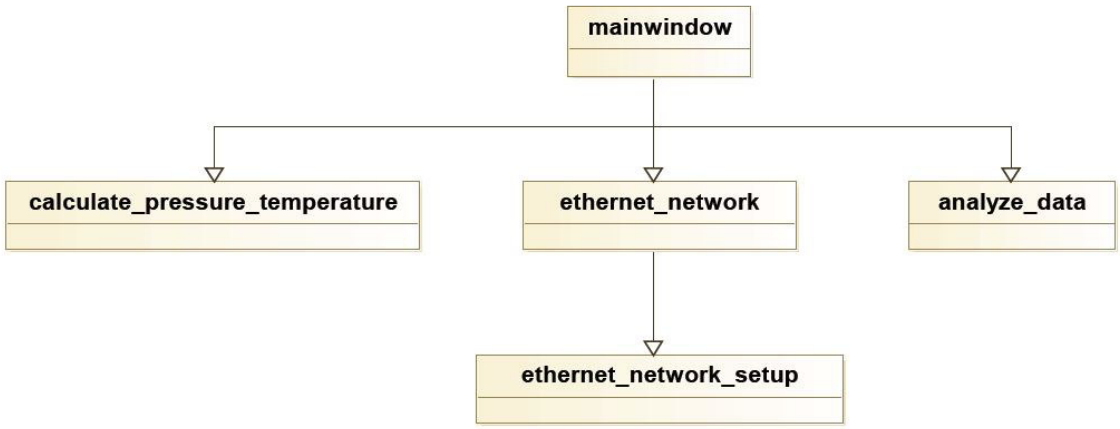


Abbildung 5.11.: Struktur der Datenaufnahme

5.3.2. Klassen

Im Folgenden wird auf den Ablauf und die Funktion der einzelnen Klassen der Datenaufnahme eingegangen. Der Quellcode zu den Klassen und Modulen ist in Anhang B zu sehen.

mainwindow

Die Klasse mainwindow dient der Steuerung der Datenaufnahme und damit des Entwicklungssystems. Dies wird mithilfe einer GUI realisiert (siehe Abbildung 5.12). Mit der GUI werden alle benötigten Eingabewerte erfasst. Dabei unterteilen sich die Eingabewerte in drei Gruppen. Die erste Gruppe betrifft das Ethernet-Netzwerk, für das die Anzahl der Ultraschallsensoren und der Mastersensor benötigt werden. Die zweite Gruppe beinhaltet die Eingabewerte für den Messdurchlauf. Dazu gehören die Frequenz, die Sendepuls-Dauer und die Echo-Aufnahmezeit. Die Gruppe für die Analyse der Daten die Parameter für eine Visualisierung der empfangenen Ultraschallechos.

Über die Klasse mainwindow wird der Verbindungsaufbau ausgelöst. Ein weiterer Punkt ist das Zusammenstellen von UDP-Anweisungspaketen, mit denen die Ultraschallsensoren gesteuert werden. Hiermit wird entweder eine Messung der Umgebungsbedingungen oder ein Ultraschall-Messdurchlauf ausgelöst. Nach einer Messung der Umgebungsbedingungen werden die berechneten Werte in der GUI visualisiert. Im Anschluss an einen Ultraschall-Messdurchlauf werden die Echodaten in der Datenanalyse weiterverarbeitet. Die empfangenen Ultraschallechos können hierbei bei Bedarf visualisiert werden.

Ein weiterer Punkt ist die Visualisierung der Basisdaten des Ethernet-Netzwerks.

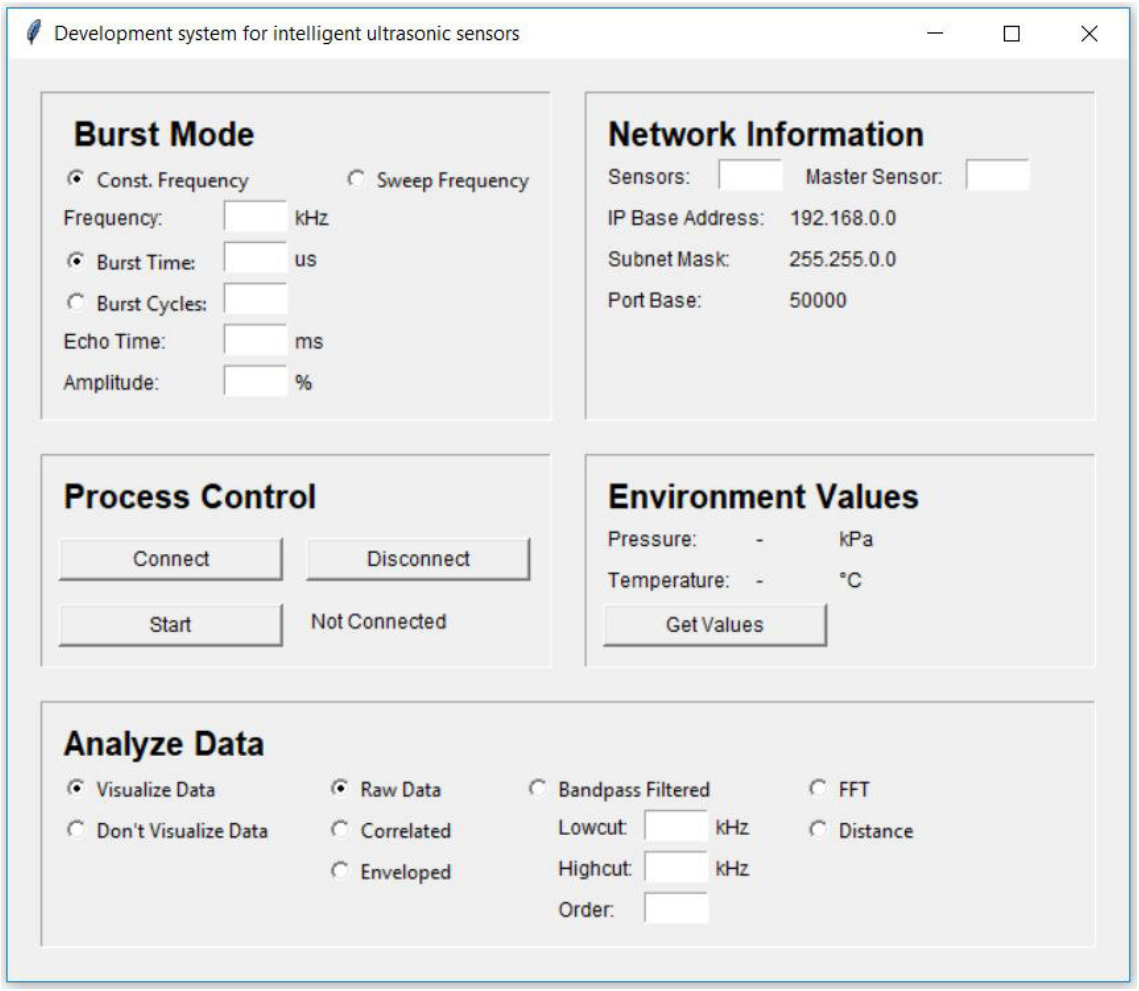


Abbildung 5.12.: GUI der Datenaufnahme

calculate_pressure_temperature

Die Klasse calculate_pressure_temperature bekommt von mainwindow die Rohdaten der Temperatur und des Umgebungsdrucks übergeben. Des Weiteren werden der Klasse die Kalibrierwerte des jeweiligen Ultraschallsensors übergeben. Die Berechnung der Temperatur und des Umgebungsdrucks erfolgt nach den Formeln aus dem Datenblatt des Temperatur- und Drucksensors (vgl. Bosch-Sensortec, 2018, S. 23). Die berechneten Werte werden anschließend an die Klasse mainwindow zurückgegeben.

ethernet_network

Mit ethernet_network werden die Verbindung und die Kommunikation zu den Ultraschallsensoren umgesetzt. Für einen Verbindungsaufbau werden die Ethernet-Sockets zur Kommunikation initialisiert und an die an die Ultraschallsensoren wird eine Status-Abfrage gesendet. Bei einer Antwort von allen Sensoren ist die Verbindung aktiv. Des Weiteren ist die Klasse für das Schließen der Sockets zuständig.

Das Versenden und Empfangen von UDP-Paketen erfolgt ebenfalls über ethernet_network. Bei dem Empfangen von Ultraschallechodaten werden diese zunächst unsortiert nach dem FIFO-Prinzip in einem Array zwischengespeichert. Wurden alle Echodaten und die Statusmeldung der Sensoren Empfangen, werden die Echodaten anhand der IP-Adresse sortiert. Abschließend werden die Echodaten in einer mat-Datei gespeichert.

ethernet_network_setup

Mit dem Modul ethernet_network_setup werden die Basisdaten des Ethernet-Netzwerks definiert. Des Weiteren werden die UDP-Paket-Prototypen für die Kommunikation mit den Ultraschallsensoren definiert. Dabei wurde das Modul nicht als Klasse umgesetzt.

Um einen Datenaustausch mit anderen Modulen zu realisieren, besitzt das Modul Funktionen, mit denen andere Module einen Lesenden-Zugriff auf die Daten bekommen.

analyze_data

Über die Klasse werden die in einer mat-Datei zwischengespeicherten Echodaten geladen, um eine Bearbeitung dieser zu ermöglichen. Da der Mastersensor während des Sendepulses keine Echodaten aufnimmt, werden die Echodaten um diesen Wert verschoben, womit ein Vergleich der Sensordaten untereinander gewährleistet wird. Des Weiteren werden die Datenreihen der Sensoren auf einen Wert von eins normiert.

Die bearbeiteten Daten werden in mehreren Dateien gespeichert, wodurch eine Analyse der Daten und Tests mit unterschiedlichen Algorithmen im Anschluss an der Messreihe möglich ist. Hierdurch wird auch eine systemunabhängige Auswertung ermöglicht. Bei den Dateien handelt es sich um eine mat- und eine txt-Datei. Hierbei ist in jeder Datei der komplette Datensatz der Ultraschallsensoren enthalten. Um die Dateien von mehreren Messdurchläufen nicht zu überschreiben, wird jeder Messdurchlauf unter einem individuellen Dateinamen gespeichert. Der Prototyp des Dateinamen ist dabei $echo\_data\_(Tage)\_(Monat)\_(Stunde)\_(Minute)\_(Sekunde).(Dateityp)$.

Des Weiteren können die Echodaten bei Bedarf direkt im Anschluss an einen Messdurchlauf visualisiert werden. Die Visualisierung der Echodaten erfolgt mit der $matplotlib$, welche

MATLAB-ähnlichen Plots erstellt (vgl. Matplot, 2012). Dabei können die Echodaten unbearbeitet beziehungsweise bearbeitet dargestellt werden. Die Echodaten können badpassgefiltert beziehungsweise mit dem Sendepuls korreliert visualisiert werden. Eine Darstellung der FFT des ungefilterten Signals ist ebenfalls möglich. Aufbauend auf den korrelierten Echodaten kann die Distanz der zurückkommenden Ultraschallechos dargestellt werden.

Die Bandpassfilterung der Echodaten erfolgt dabei mit einem Butterworth-Bandpassfilter, bei dem der Frequenzgang zwischen den Grenzfrequenzen möglichst lang einen horizontalen Verlauf aufweist (vgl. Tietze u. a., 2016, S. 800). Die Grenzfrequenzen und die Ordnung des Filters sind hierbei individuell einstellbar.

Um aus dem Echosignal Übereinstimmungen mit dem Sendepuls zu finden, bietet sich eine Kreuzkorrelation an (vgl. Hazas und Hopper, 2006, S. 543). Hierbei wird das Echosignal mit dem Sendepuls korreliert, um Übereinstimmungen in den Signalen zu erkennen. Dadurch ergibt sich ein Ähnlichkeitsmaß, wobei dieses den maximalen Wert bei der größten Übereinstimmung besitzt (vgl. Ohm und Lüke, 2014, S. 214):

$$\Phi_{sg}^{E}(m) = \sum_{n=\infty}^{\infty} s^*(-m)g(n+m) \tag{5.33}$$

Das korrelierte Echosignal kann des Weiteren als einhüllende Kurve dargestellt werden. Hierbei wird die Hilbert-Transformation als analytisches Signal verwendet (vgl. Karrenberg, 2017, S. 545):

$$y(t) = \frac{1}{\pi} \int_{-\infty}^{\infty} x(\tau)(\frac{1}{t-\tau})d\tau \tag{5.34}$$

Für die Darstellung der Distanz werden die Maximalstellen in der einhüllenden Kurve gesucht und diese mit einer auf die Distanz normierten X-Achse visualisiert.

6. Tests

Um die Funktionsfähigkeit des Entwicklungssystems für intelligente Ultraschallsensoren darzustellen, werden zunächst der Ultraschallsensor und die Datenaufnahme einzeln getestet. Anschließend erfolgt ein Test des gesamten Entwicklungssystems. Die Tests orientieren sich dabei an den Anforderungen an das Entwicklungssystem. Den Abschluss dieses Kapitels macht die Bewertung des Entwicklungssystems.

6.1. Ultraschallsensor

Die Tests des Ultraschallsensors unterteilen sich in eine Frequenzanalyse und dem zeitlichen Verhalten, auf die im Folgenden eingegangen wird. Des Weiteren werden die Ultraschall-Sendepulse getestet.

6.1.1. Frequenzanalyse

Für die Frequenzanalyse des Ultraschallsensors werden zunächst die Frequenzgänge der Sende- und Empfängerverstärkerschaltung gemessen. Daran anschließend wird der Frequenzgang des gesamten Ultraschallsensors aufgenommen. Des Weiteren werden die Ultraschall-Sendepulse analysiert.

Sendeverstärkerschaltung

Die Aufnahme des Frequenzgangs der Sendeverstärkerschaltung erfolgt mit der Open-Source-Software *Frequency Response Analyzer for PicoScope* (vgl. Pico-Technology, 2018). Die Schaltung wird dabei mit angeschlossenem Ultraschalllautsprecher getestet. Das Anregungssignal ist ein Sinus mit 1 V Amplitude. Die Frequenzanalyse wird hierbei für einen Frequenzbereich von 100 Hz bis 200 kHz durchgeführt.

In Abbildung 6.1 ist der aufgenommene Frequenzgang zu sehen. Frequenzen über der halben Abtastfrequenz von 200 kHz werden deutlich gedämpft. Des Weiteren ist zu sehen, dass

der Gleichanteil aus dem Signal gefiltert wird. Die Verstärkung fällt mit ca. 22 dB etwas niedriger aus. Die durch den Reihenschwingkreis erzeugte Spannungserhöhung fällt deutlich niedriger aus. Daraus folgt, dass die Sendeempfängerschaltung näherungsweise wie die berechnete Schaltung funktioniert.

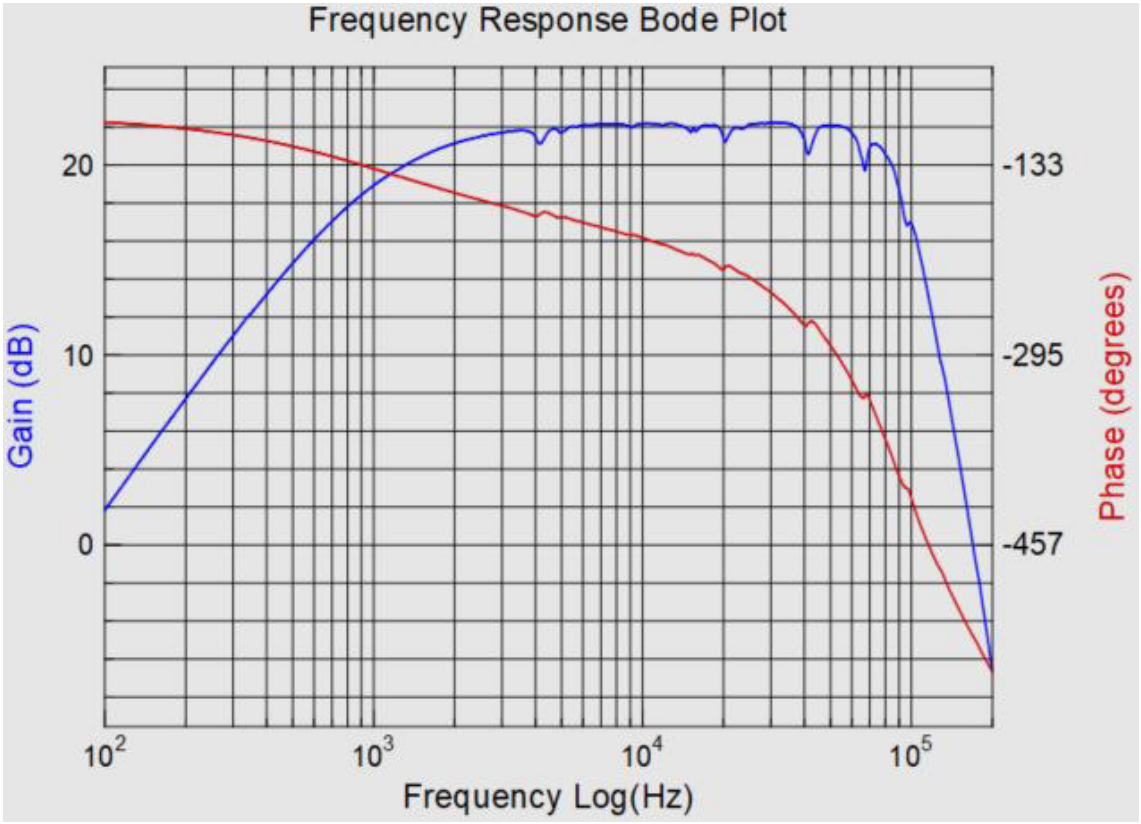


Abbildung 6.1.: Gemessener Frequenzgang der Sendeverstärkerschaltung

Empfängerverstärkerschaltung

Der Frequenzgang der Empfängerverstärkerschaltung wird ebenfalls mit dem FRA aufgenommen. Als Anregungssignal wurde ein Sinus mit 10 mV Amplitude verwendet. Der Frequenzbereich der Frequenzanalyse wird wie bei der Sendeverstärkerschaltung auf 100 Hz bis 200 kHz gewählt.

Der aufgenommene Frequenzgang zeigt (siehe Abbildung 6.2), dass die Frequenzen im Arbeitsbereich von 30 kHz bis 60 kHz verstärkt werden. Dabei fällt die Verstärkung etwas niedriger als angestrebt aus. Niedrigere und höhere Frequenzen werden deutlich niedriger verstärkt als die Frequenzen im Arbeitsbereich. Der raue Verlauf des Frequenzgangs kommt

vom Signalgenerator des *PicoScops*. Hierbei wird die 10 mV Amplitude zur Anregung der Schaltung von der Quantisierung und dem Rauschen des Signalgenerators stark überlagert.

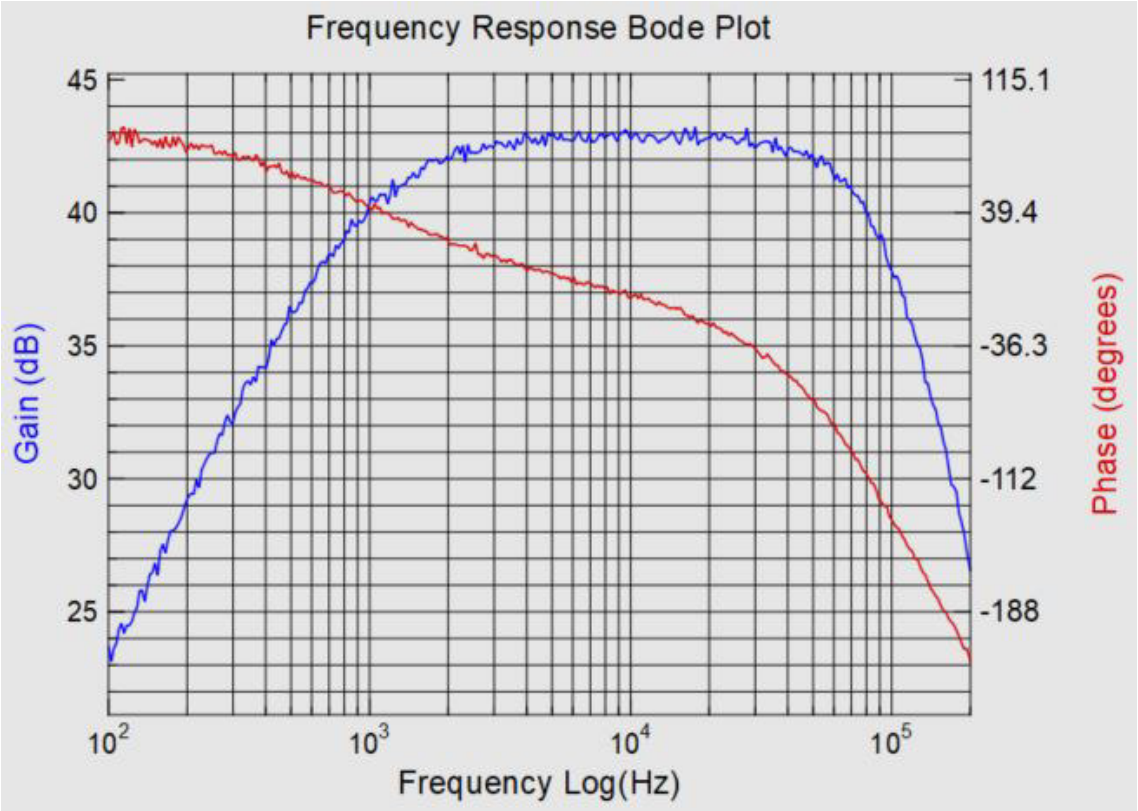


Abbildung 6.2.: Gemessener Frequenzgang der Empfängerverstärkerschaltung

Ultraschallsensor

Um den Frequenzgang des gesamten Ultraschallsensors aufnehmen zu können, wurden zwei Ultraschallsensoren in einem Abstand von 5,45 m zueinander aufgestellt (siehe Abbildung 6.3). Der Master-Sensor hat dabei ein Frequenzsweep von 1 kHz bis 100 kHz über 100 ms ausgesendet, welcher vom Slave-Sensor empfangen wurde. Um die Messung nicht durch Umgebungsrauschen und Ultraschallechos zu beeinflussen, wird der Test im Schallmessraum 14.61 am Berliner Tor 7 der *HAW Hamburg* durchgeführt.

Zur Auswertung wurde das vom Slave-Sensor empfangene Ultraschall-Signal mit *MATLAB* analysiert. Hierzu wurde eine FFT durchgeführt und anschließend über das FFT-Signal eine Einhüllende-Kurve gelegt. Des Weiteren wurden die Werte in dBFS umgerechnet.

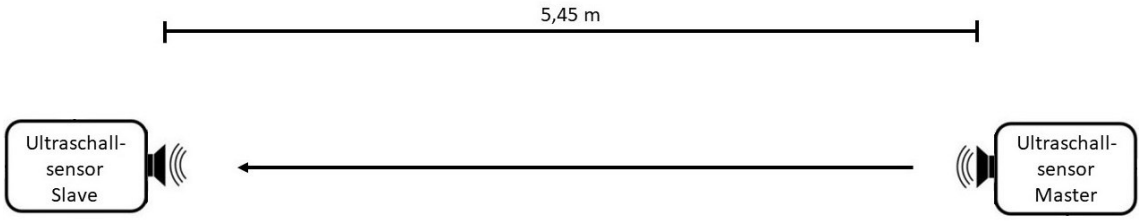


Abbildung 6.3.: Skizzierter Versuchsaufbau im Schallmessraum

Vergleicht man den Frequenzgang des Ultraschallsensors (siehe Abbildung 6.4) mit den Frequenzgängen des Ultraschall-Lautsprechers und des MEMS-Mikrofons (siehe Abbildung 6.5), ist die Überlagerung beider Frequenzgänge zu erkennen. Die Frequenzgänge der Empfänger- beziehungsweise Sendeverstärkerschaltung sind kaum erkennbar, da die Grenzfrequenzen im Randbereich der Abbildung liegen.

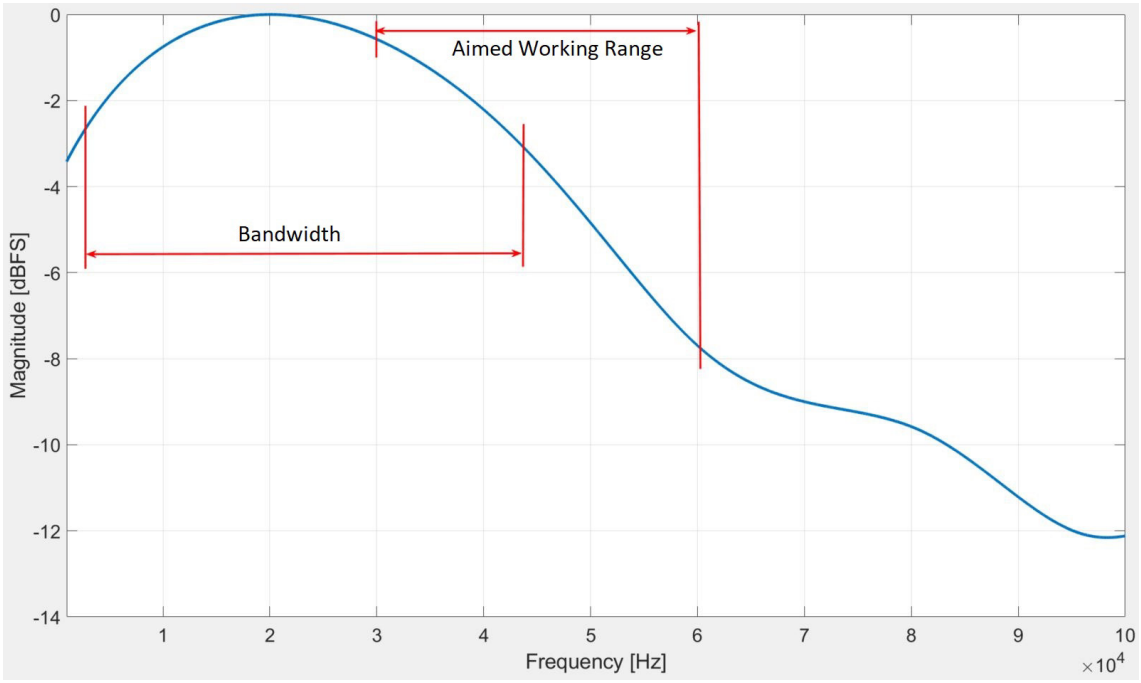


Abbildung 6.4.: Ausgewerteter Frequenzgang des Ultraschallsensors

Analysiert man den Frequenzgang des Ultraschallsensors, liegt die untere Grenzfrequenz bei 1,9 kHz und die obere Grenzfrequenz bei 43,4 kHz. Somit weist dieser eine Bandbreite von 41,5 kHz auf, wovon 23,4 kHz im Ultraschallbereich liegen. Hierbei liegt der angestrebte Arbeitsbereich des Ultraschallsensors von 30 kHz bis 60 kHz nur zum Teil innerhalb der -3 dB-Grenzen.

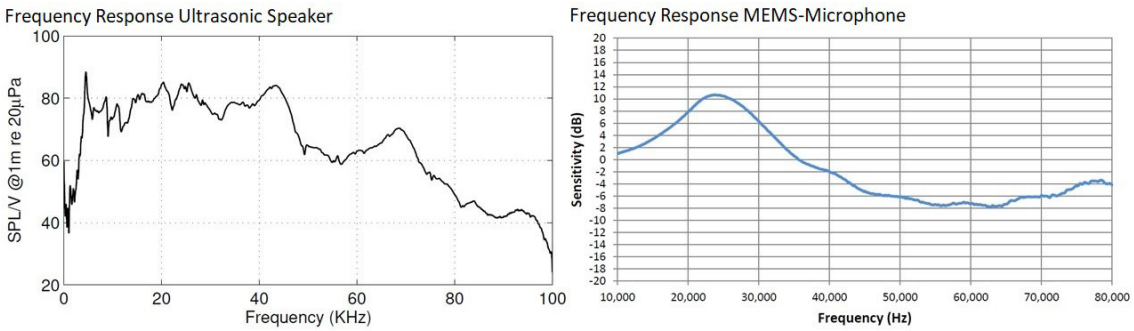


Abbildung 6.5.: Frequenzgänge des Ultraschall-Lautsprechers (vgl. Albuquerque, 2013, S. 149) und des MEMS-Mikrofons (vgl. Knowles-Electronics, 2013, S. 3)

6.1.2. Zeitliches Verhalten

Für eine Analyse des zeitlichen Verhaltens der Ultraschallsensoren wird zunächst die Abtastrate des ADC untersucht. Des Weiteren wird die Synchronität der Ultraschallsensoren untereinander analysiert.

ADC-Abtastrate

Da die Abtastwerte des Ultraschallechos direkt im Anschluss an das Ethernet-Modul gesendet werden, ist hier sicherzustellen, dass durch den SPI-Sendevorgang die Abtastrate nicht beeinflusst wird. Für die Messung wird die SPI-Schnittstelle mit dem *PicoScope* aufgenommen. Zusätzlich wird vor jedem ADC-Abtastwert ein Debug-Pin gesetzt, um die Zeitdifferenz zwischen den einzelnen Abtastwerten messen zu können.

Abbildung 6.6 zeigt, dass durch das Setzen des Write-Pointers und durch den Transmit-Befehl drei Abtastwerte verschoben werden. Dabei ist der 3. Abtastwert nach dem Setzen des Write-Pointers um 1,1 μs verschoben, wobei der Abstand zwischen zwei Abtastwerten bei einer Abtastfrequenz von 200 kHz 5 μs beträgt. Der 4. Abtastwert nach dem Transmit-Befehl ist um 1,7 μs verschoben. Der 5. Abtastwert weist eine Verschiebung von 0,4 μs auf. Da die maximale UDP-Paketlänge 1470 Byte beträgt und ein Abtastwert aus 2 Byte besteht, kommt es alle 735 Abtastwerte und somit nach 3,68 ms zu einer Verschiebung. Da hierbei kein Abtastwert ausgelassen wird und die Abtastzeit näherungsweise eingehalten wird, werden die aufgenommenen Ultraschallechos nicht verfälscht.

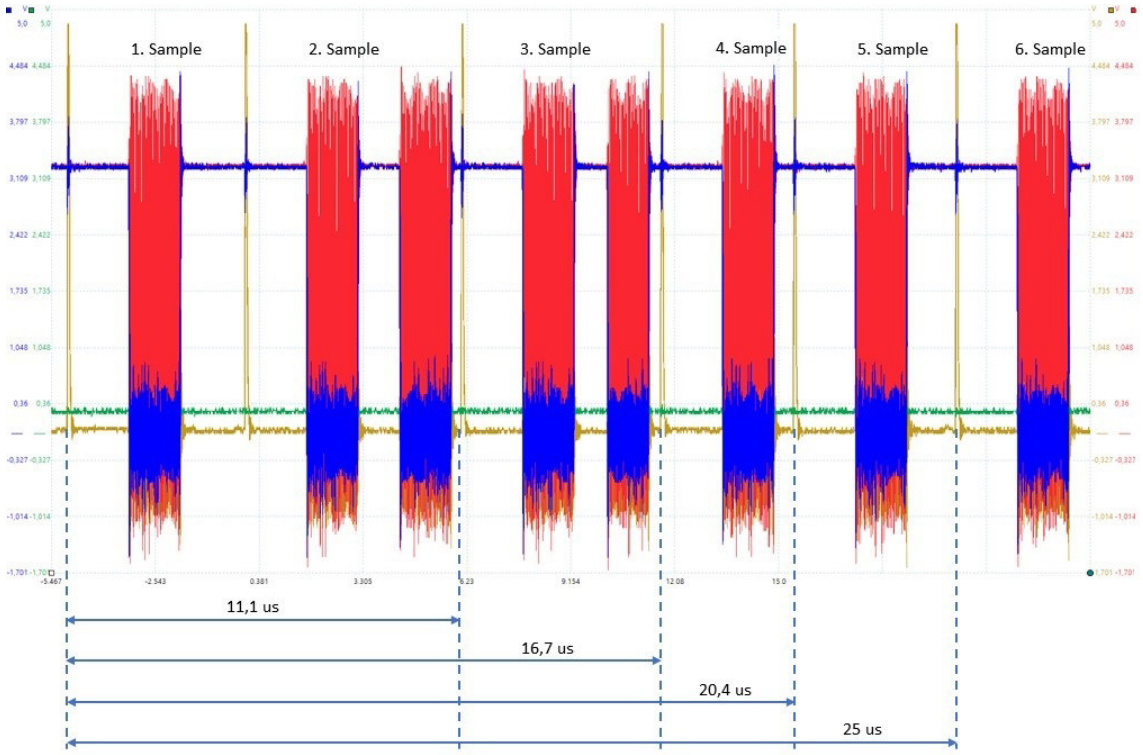


Abbildung 6.6.: Aufgenommene SPI-Schnittstelle mit Set-Pointer- und Transmit-Befehl

Synchronisation der Ultraschallsensoren

Um die Synchronisation der Ultraschallsensoren untereinander zu bewerten, werden in der Sensor-Software Debug-Pins integriert, die mit dem *PicoScope* getestet werden. Hierbei wird vom Setzen der Synchronisationsleitung durch den Master die Zeit bis zum Start des DAC beim Master beziehungsweise des ADC beim Slave gemessen. Des Weiteren wird die Zeit vom Setzen der Synchronisationsleitung bis zum Deaktivieren des ADC beim Master und Slave gemessen.

Beim Start einer Messung weist der Master eine Verzögerung zum Synchronisationssignal von 2,26 μs auf (siehe Abbildung 6.7). Die Verzögerung beim Slave beträgt 2,2 μs. Die Verzögerungen zum Synchronisationssignal beim Beenden eines Messdurchlaufs betragen beim Master 1,7 μs und beim Slave 5,83 μs. Bei den Messungen konnten nur konstante Verzögerungen festgestellt werden. Da das *PicoScope* mit 250 Msps aufzeichnet, liegt der Wert für den Jitter unter 4 ns.

Da für eine synchrone Aufnahme der Echosignale der Start der Messung entscheidend ist und dabei zwischen Master und Slave eine Zeitdifferenz von 0,06 μs ermittelt wird, folgt

daraus, dass eine synchrone Aufnahme gewährleistet ist. Die Zeitdifferenz zwischen Master und Slave beim Beenden des Messdurchlaufs ist vernachlässigbar klein.

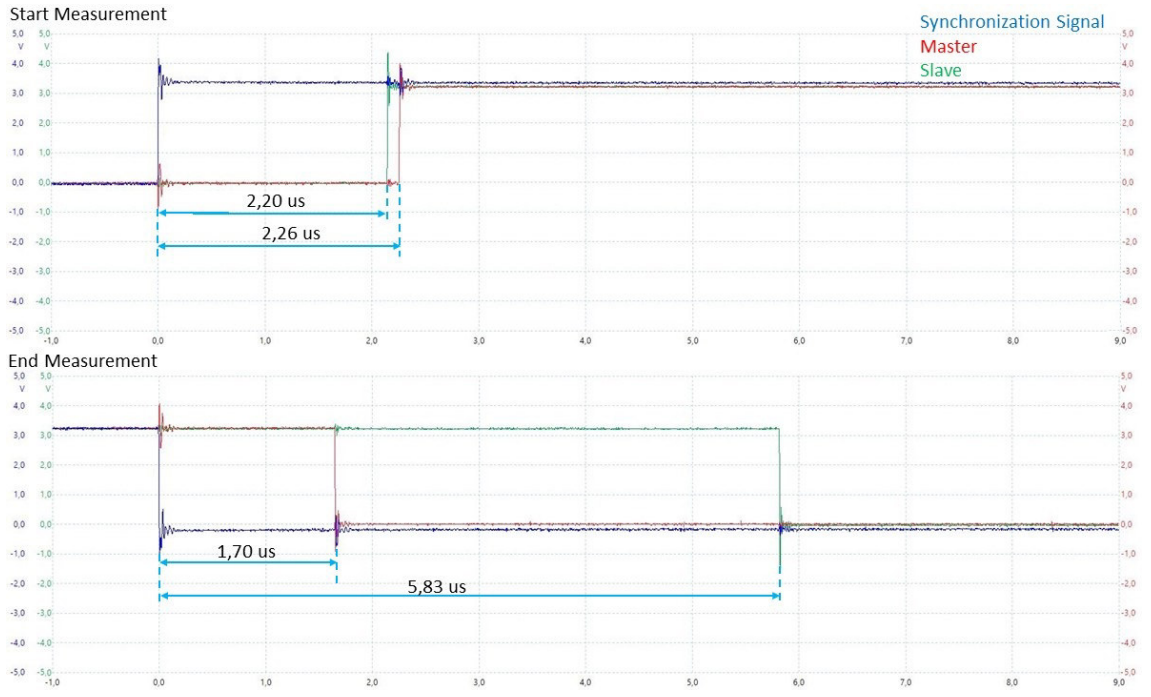


Abbildung 6.7.: Aufgenommene Start und Ende eines Messdurchlaufs

6.1.3. Ultraschall-Sendepuls

Für die Überprüfung der Ultraschall-Sendepulse und um diese in der Echosignalauswertung zu verwenden, werden die Ultraschall-Sendepulse aufgenommen. Hierbei werden die Sendepulse unter identischen Bedingungen wie der Test des Frequenzgangs des Ultraschallsensors aufgenommen. Dabei werden die Sendepulse mit den Frequenzen 30 kHz, 45 kHz und 60 kHz getestet, um den angestrebten Arbeitsbereich des Ultraschallsensors abzudecken. Die Sendepulse haben eine Länge von 8 Perioden, 16 Perioden und 1 ms. Des Weiteren werden die Sendepulse mit einem Laser-Doppler-Vibrometer aufgezeichnet und mit den Aufnahmen der MEMS-Mikrofone verglichen. Als Laser-Doppler-Vibrometer wird das *Polytec IVS 200* verwendet. Abschließend wird die maximale Sendepulslänge getestet.

Abbildung 6.8 zeigt einen vom Ultraschallsensor aufgenommenen 45 kHz Ultraschall-Sendepuls mit einer Länge von 1 ms. Die Einschwingzeit dauert bei allen Frequenzen ca. eine Periode. Das Abklingen des Ultraschall-Lautsprechers dauert im Durchschnitt 1,2 ms.

Die mit *MATLAB* analysierten Abklingzeiten sind in Tabelle 6.1 zu sehen. Daraus folgt nach Formel 2.13, dass die Blindzone des Ultraschallsensors bis zu 24,46 cm beträgt.

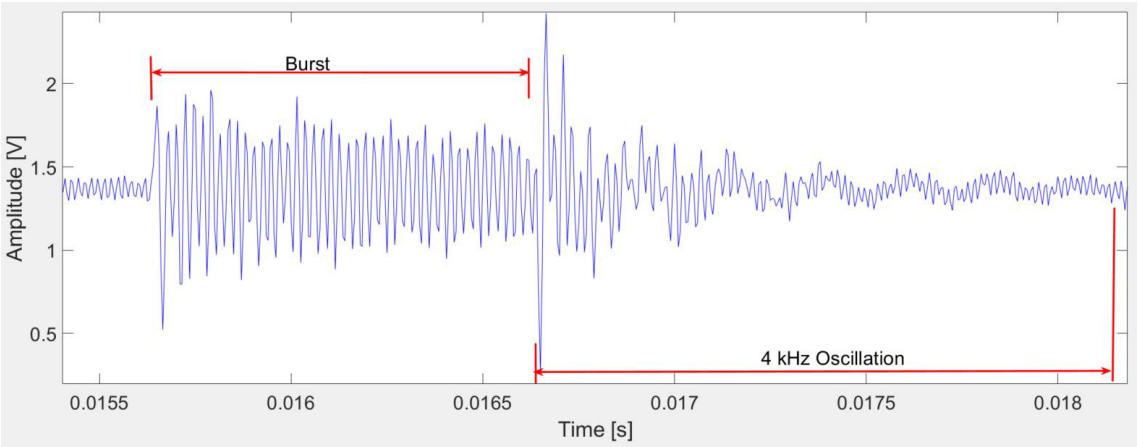


Abbildung 6.8.: Mit dem Ultraschallsensor aufgenommener 45 kHz Ultraschall-Sendepuls

Frequenz	**8 Perioden**	**16 Perioden**	**1 ms Sendepuls**
30 kHz	1 ms	1 ms	1,1 ms
45 kHz	1,4 ms	1,3 ms	1,4 ms
60 kHz	1,3 ms	1,2 ms	1,3 ms

Tabelle 6.1.: Abklingzeiten des Ultraschall-Lautsprechers

Vergleich man die Sendepulse mit unterschiedlichen Frequenzen miteinander, ist hier ein identisches Verhalten zu erkennen, wobei der Frequenzgang des Ultraschallsensors anhand der Amplituden deutlich zu erkennen ist. Ebenfalls sind kurze Sendepulse, wie 8 beziehungsweise 16 Perioden, bei 30 kHz und 45 kHz deutlich erkennbar. Bei 60 kHz sind kurze Sendepulse durch die hohe Dämpfung kaum erkennbar.

Des Weiteren ist auf den Sendepulsen eine leichte beziehungsweise beim Ausschwingen eine starke 4 kHz Schwingung zu erkennen. Die 4 kHz Schwingung wird mit steigender Frequenz des Sendepulses dominanter, da gleichzeitig die Dämpfung des Ultraschallsensors zunimmt. Um die Ursachen hierfür zu ermitteln, werden die Sendepulse mit dem Laser-Doppler-Vibrometer aufgezeichnet. Abbildung 6.9 zeigt einen 40 kHz Sendepuls mit einer Länge von 1 ms, welcher mit dem Laser-Doppler-Vibrometer in Verbindung mit dem *PicoScope* aufgezeichnet wurde. Die -3 dB-Grenze des Laser-Doppler-Vibrometers liegt bei 20 kHz, wobei eine Aufnahme der Sendepulse bis 40 kHz möglich ist und diese deutlich zu erkennen sind. Parallel zur Aufzeichnung mit dem Laser-Doppler-Vibrometer wird die Spannung über dem Ultraschall-Lautsprecher mit dem *PicoScope* aufgezeichnet.

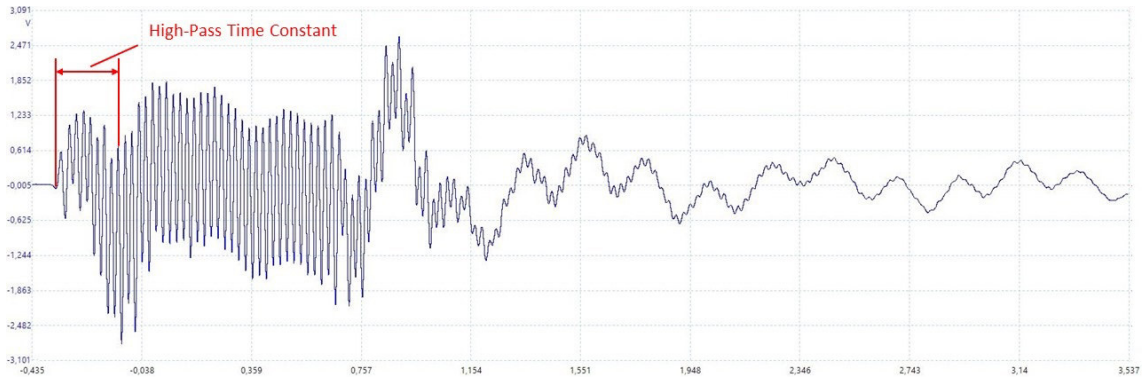


Abbildung 6.9.: Mit dem Laser-Doppler-Vibrometer aufgenommener 40 kHz Ultraschall-Sendepuls

Hierbei weisen die mit dem Laser-Doppler-Vibrometer aufgezeichneten Sendepulse ein identisches Verhalten zu den mit dem Ultraschallsensor aufgenommenen Sendepulsen auf, wobei hier die Zeitkonstante des Hochpasses in der Sendeverstärkerschaltung zu erkennen ist. Die Messungen mit dem Laser-Doppler-Vibrometer werden an mehreren Punkten auf der Membran durchgeführt, um das Verhalten des Ultraschall-Lautsprechers zu analysieren. Hierbei wird an allen gemessenen Punkten auf der Membran ein identisches Verhalten festgestellt. Des Weiteren wird die Schwingung des Gehäuses bei einem Sendepuls aufgezeichnet, wobei hier eine vernachlässigbar kleine Schwingung zu erkennen ist. Bei der paralellen Aufzeichnung der elektrischen Wechselspannung über den Ultraschall-Lautsprecher ist die 4 kHz Schwingung der Membran nicht zu erkennen.

Hieraus folgt, dass die 4 kHz Schwingung nicht durch die elektronische Schaltung verursacht wird und es sich hierbei um eine mechanische Eigenschaft des Ultraschall-Lautsprechers handelt, die bei Anregung dieses erzeugt wird.

Ein weiteres Ergebnis des Tests mit dem Laser-Doppler-Vibrometer ist, dass die 4 kHz Schwingung durch den Sendepuls beeinflussbar ist. Da bei den Tests eine feste Sendepulslänge eingestellt wurde, enden diese nicht am Ursprungsort des Sendepulses. Hierdurch wird ein Sprung auf den Lautsprecher gegeben, welcher die 4 kHz Schwingung auslöst. Bei einem Enden des Sendpulses im Ursprungspunkt wird kein Sprung auf den Lautsprecher gegeben, wodurch die 4 kHz Schwingung beim Ausschwingen des Lautsprechers deutlich geringer ausfällt.

Zur Überprüfung der Frequenz der Sendepulse werden die Aufnahmen mit einer FFT analysiert. Das Ergebnis der FFT-Analyse ist, dass die Frequenzen eine Toleranz von durchschnittlich 0,5 % aufweisen (siehe Tabelle 6.2).

Der Test der maximalen Sendepulslänge mit 4095 μs ergab, dass diese bei allen Frequenzen realisierbar ist.

Soll-Frequenz	Ist-Frequenz	Δf
30 kHz	29,9 kHz	-0,1 kHz | -0,3 %
45 kHz	45,2 kHz	0,2 kHz | 0,4 %
60 kHz	60,4 kHz	0,4 kHz | 0,7 %

Tabelle 6.2.: Vergleich von Soll- und Ist-Frequenzen

6.2. Datenaufnahme

Der Test der Datenaufnahme erfolgt mit einem definierten Signal, welches vom Ultraschallsensor gesendet wird. Bei dem Signal handelt es sich um einen Sinus mit 16 Perioden. Der Test erfolgt mit den Frequenzen 30 kHz, 45 kHz und 60 kHz. Die Aufnahmezeit des Echosignals beträgt dabei 2 ms.

Bei der Überprüfung der rohen Echodaten un der FFT-Analyse kommt heraus, dass die Signale richtig empfangen und dargestellt werden (siehe Abbildung 6.10).

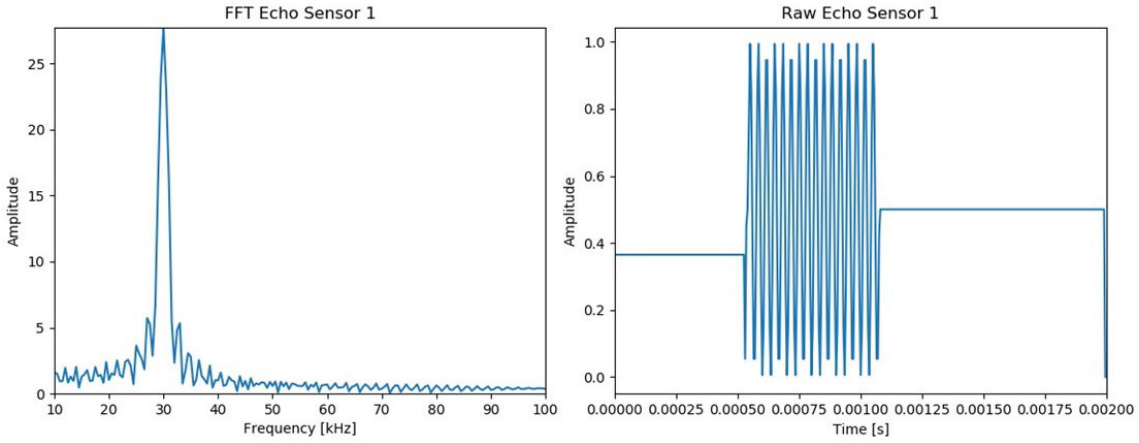


Abbildung 6.10.: Plots der rohen Echodatn und der FFT für 30 kHz

Des Weiteren wird die Darstellung der Distanz getestet. Da das korrelierte und das eingehüllte Echosignal mit der Distanzermittlung zusammenhängen, werden diese mit diesem Test ebenfalls abgedeckt. Hierbei werden zwei Durchgänge mit einem Echo nach 500 μs beziehungsweise nach 2 ms simuliert. Als Temperatur wird der Datenaufnahme 22 °C vorgegeben. Die Datenaufnahme ermittelt dabei die Distanz mit einer maximalen Abweichung von 0,2 cm, was innerhalb der geforderten Anforderung liegt (siehe Tabelle 6.3). Die Abweichung der Distanz ist dabei von der Frequenz des Echosignals unabhängig.

Echozeit	Ist-Distanz	Soll-Distanz	Abweichung
500 μs	8,8 cm	8,6 cm	0,2 cm
2 ms	34,6 cm	34,5 cm	0,1 cm

Tabelle 6.3.: Ergebnisse der Überprüfung der Distanzauswertung

6.3. Entwicklungssystem

Die Überprüfung des Entwicklungssystems unterteilt sich in vier Bereiche. Zunächst wird die Übertragungssicherheit des Kommunikationsnetzwerks getestet. Darauf folgend wird ein Test zur Erfassung der Umgebungsbedingungen durchgeführt. Des Weiteren wird die Erfassung von Abständen überprüft. Abschließend findet ein Messdurchlauf zur Erfassung von Vielfachechos statt.

6.3.1. Kommunikationsnetzwerk

Zum Test der Übertragungssicherheit des Kommunikationsnetzwerks wird eine 32-bit Variable auf dem Ultraschallsensor inkrementiert und an den Rechner geschickt, wobei die UDP-Paketlänge 1472 Byte beträgt. Daraus folgt, dass ca. 11,69 Millionen UDP-Pakete an den Rechner gesendet werden. Auf Seite des Rechners werden die UDP-Pakete mit einem Python-Programm entpackt und kontrolliert. Bei einem Fehler beziehungsweise bei einem Paketverlust wird eine entsprechende Zählvariable inkrementiert. Die Frequenz des Ultraschallsensors wird auf 100 kHz gesenkt, um das Datenaufkommen während der Echoaufnahmezeit zu simulieren.

Das Ergebnis des Tests ist, dass ein Paketverlust von durchschnittlich 0,46 ppm auftritt (siehe Tabelle 6.4). Bei den Paketverlusten handelt es sich um fehlerhafte Telegrammübertragungen, zum Beispiel Bitkippen oder durch EMV-Störungen verfälschte Telegramme, die mithilfe des CRC-Werts erkannt wurden. Da nicht alle Fehler bei der Übertragung durch den CRC-Wert erkannt werden (vgl. Reißenweber, 2009, S. 99), erfolgt die überprüft der einzelnen Werte der UDP-Pakete, wobei hier kein Fehler aufgetreten ist.

Hierbei treten die Paketverluste erst im höheren Bereich auf. Der erste Paketverlust tritt frühestens bei dem hunderttausendsten Paket auf. Das entspricht für einen einzelnen Ultraschallsensor eine fehlerfreie Datenübertragung von 367,5 s aufgenommener Ultraschallechos beziehungsweise auf 12 Ultraschallsensoren umgerechnet einer Datenübertragung von 30,6 s aufgenommener Ultraschallechos. Daraus folgt, dass die Übertragungssicherheit bei der Übertragung gegeben ist, da die effektive Aufnahmezeit von Ultraschallechos deutlich niedriger ausfällt.

Nr.	Fehler	Paketverlust
1	0	0,88 ppm
2	0	0,35 ppm
3	0	0,39 ppm
4	0	0,43 ppm
5	0	0,24 ppm

Tabelle 6.4.: Fehler und Paketverluste des Kommunikationsnetzwerks

6.3.2. Umgebungsbedingungen

Um die Erfassung von Temperatur und Umgebungsdruck zu bewerten, werden die aufgenommenen Werte mit Referenzwerten verglichen. Zum Vergleich der Temperatur wird das $PeakTech\ 5180$ als Referenzthermometer verwendet. Die Überprüfung des Umgebungsdrucks erfolgt mit meteorologisch erfassten Messwerten der $Freien\ und\ Hansestadt\ Hamburg$ (vgl. Hansestadt-Hamburg, 2018). Die meteorologischen Messwerte werden hierbei als Anhaltspunkt verwendet, da die Messwerte stündlich aktualisiert werden und die Messstation 5 km entfernt ist. Des Weiteren ist die Genauigkeit der Messwerte nicht bekannt.

Der Test findet unter Innenraumbedingungen statt und wird mit zwei Ultraschallsensoren durchgeführt. Dabei werden die Tests zu unterschiedlichen Zeitpunkten durchgeführt, um eine Beeinflussung der Ergebnisse zu verhindern.

Das Ergebnis des Temperaturtests ist, dass diese eine durchschnittliche Abweichung von 1,6 % aufweist (siehe Tabelle 6.5). Dabei werden die Messwerte des Entwicklungssystems auf eine Nachkommastelle gerundet, um einen Vergleich mit dem $PeakTech\ 5180$ zu realisieren.

Nr.	PeakTech	Entwicklungssystem	ΔT
1	22,9 °C	22,62 °C	-0,3 °C | -1,3 %
2	23,3 °C	23,34 °C	0,0 °C | 0 %
3	24,6 °C	24,25 °C	-0,3 °C | -1,2 %
4	22,2 °C	21,69 °C	-0,5 °C | -2,3 %
5	23,5 °C	23,53 °C	0,0 °C | 0 %

Tabelle 6.5.: Ergebnisse der Überprüfung der Temperatur

Bei der Überprüfung des Umgebungsdrucks kommt heraus, dass diese ohne Abweichung erfasst werden (siehe Tabelle 6.6). Hierbei werden die Messwerte des Entwicklungssystems ebenfalls auf eine Nachkommastelle gerundet, um den Vergleich zu ermöglichen.

Nr.	Meteorologisch	Entwicklungssystem	Δp
1	101,0 kPa	100,96 kPa	0 kPa
2	101,0 kPa	100,96 kPa	0 kPa
3	101,0 kPa	100,92 kPa	0 kPa
4	101,9 kPa	101,93 kPa	0 kPa
5	101,9 kPa	101,90 kPa	0 kPa

Tabelle 6.6.: Ergebnisse der Überprüfung der Temperatur

6.3.3. Abstandserfassung

Zur Überprüfung der Abstandsermittlung der Datenaufnahme mit Messwerten des Ultraschallsensors wird der Test in einem Schallmessraum durchgeführt, um möglichst nur ein Ultraschallecho zu empfangen. Hierzu wird ein Objekt in unterschiedlichen Abständen vor dem Ultraschallsensor platziert. Die Abstandsmessung wird mit den Frequenzen 30 kHz, 45 kHz und 60 kHz durchgeführt. Die bei der Messung ermittelte Umgebungstemperatur beträgt 21,4 °C.

Das Ergebnis dieses Tests ist, dass die ermittelten Distanzen der Datenaufnahme bei realen Echosignalen um ca. 5,4 cm im Durchschnitt abweichen (siehe Tabelle 6.7). Die hohe Differenz liegt an den verwendeten *MATLAB*-ähnlichen *Python*-Funktionen, die eine Anpassung der Berechnung nicht ermöglichen. Zum Beispiel ist bei der Berechnung der einhüllenden Kurve des korrelierten Echosignals eine Anpassung des Wertebereichs, über den die Kurve gemittelt wird, nicht möglich. Hierdurch kann das Rauschen, das im Signal vorhanden ist, nicht unterdrückt werden. Dies führt bei der Ermittlung der Maximalwerte, zur Visualisierung der Distanzen, dazu, dass dabei durch das Rauschen verursachte Maximalwerte erfasst werden, was zu einer fehlerhaften Visualisierung der Distanzen führt.

Frequenz	Soll-Distanz	Ist-Distanz	Δx
30 kHz	194,1 cm	184,5 cm	-9,6 cm | -5,0 %
45 kHz	194,1 cm	194,3 cm	0,2 cm | 0,1 %
60 kHz	194,1 cm	194,4 cm	0,3 cm | 0,2 %
30 kHz	363,2 cm	356,1 cm	-7,1 cm | -2,0 %
45 kHz	363,2 cm	358,0 cm	-5,2 cm | -1,4 %
60 kHz	363,2 cm	356,0 cm	-7,2 cm | -2,0 %
30 kHz	555,1 cm	549,6 cm	-5,5 cm | -1,0 %
45 kHz	555,1 cm	549,4 cm	-5,7 cm | -1,0 %
60 kHz	555,1 cm	547,3 cm	-7,8 cm | -1,4 %

Tabelle 6.7.: Ergebnisse der Distanzermittlung

Zum Vergleich wird die Distanz aus den gespeicherten Echodaten mit *MATLAB*, bei dem eine Anpassung der einhüllenden Kurve möglich ist, ermittelt. Hierbei trat eine durchschnittliche Abweichung von unter 1 cm auf (siehe Abbildung 6.11), woraus folgt, dass die Echodaten vor der Visualisierung richtig bearbeitet und gespeichert werden.

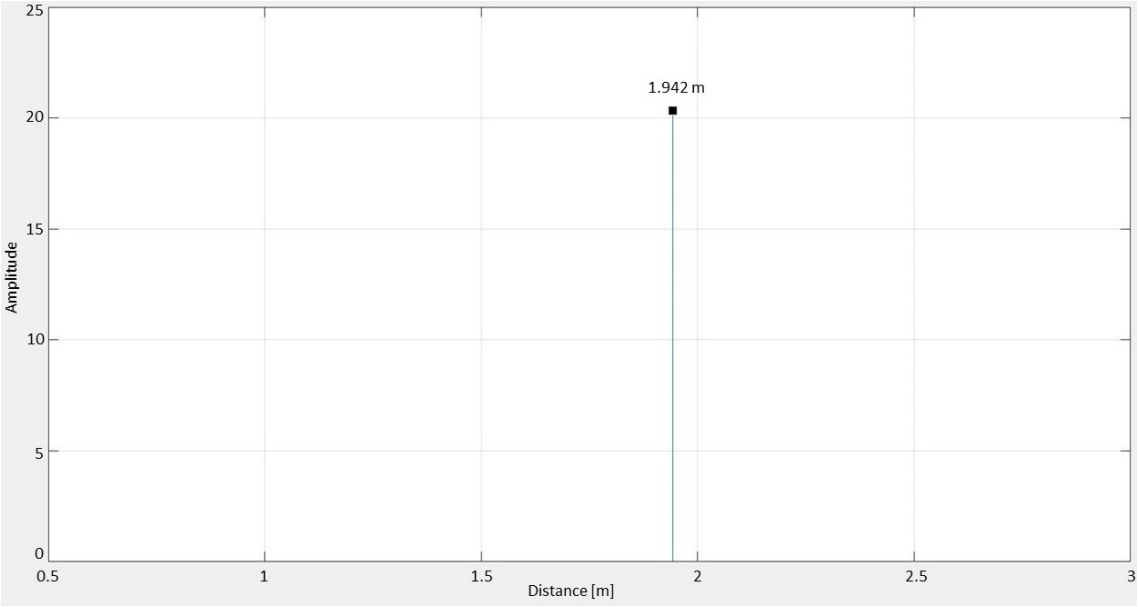


Abbildung 6.11.: Ermittelte Distanz mit *MATLAB* bei 30 kHz und 1,941 m

6.3.4. Messdurchlauf

Der Test zur Erfassung von Vielfachechos erfolgt mit zwei Ultraschallsensoren, die dabei auf einem verschiebbaren Versuchsträger fixiert sind (siehe Abbildung 6.12). Die Sensoren haben einen Abstand von 22 cm zueinander und sind 90 ° zum Testgang ausgerichtet und in diesem mittig platziert. Der Testgang hat die Maße 1,5 m x 7,9 m bei einer Höhe von 2,9 m (siehe Abbildung 6.13). Das Entwicklungssystem wird bei diesem Test mit einem Ultraschallsensor-Array bestehend aus drei *MaxBotix MaxSonar MB*1030 verglichen (vgl. MaxBotix, 2005, S. 10). Das Signal der drei Referenzsensoren ist die einhüllende Kurve des korrelierten Echosignals und wird mit dem *PicoScope* erfasst. Der Test wird bei vier unterschiedlichen Abständen zur Stirnwand durchgeführt. Die Abstände betragen 1,897 m, 3,13 m, 4,762 m und 7,997 m. Um das Entwicklungssystem mit dem Ultraschall-Array zu vergleichen, wird die Frequenz der Ultraschall-Sendpulse auf 48 kHz gesetzt.

Der Vergleich der einhüllenden Kurven des korrelierten Signals zeigt, dass näherungsweise identische Ultraschallechos von beiden Systemen empfangen werden (siehe Abbildung

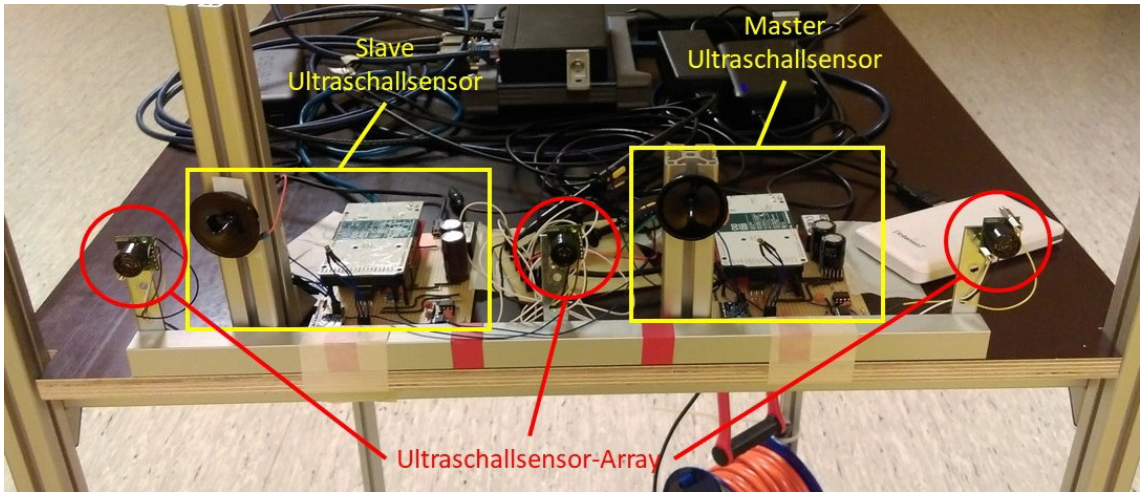


Abbildung 6.12.: Versuchsaufbau der Ultraschallsensoren

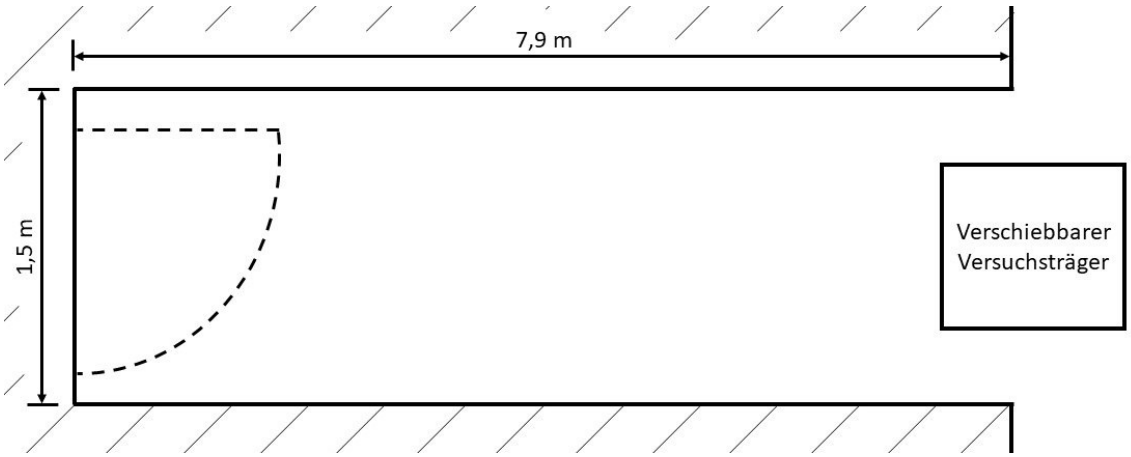


Abbildung 6.13.: Skizzierte Versuchsumgebung der Messdurchläufe

6.15). Es ist auch zu erkennen, dass auf dem Echosignal, welches vom Entwicklungssystem visualisiert wird, ein deutliches Rauschen liegt.

Abbildung 6.14 zeigt die bei dem Messdurchlauf mit dem Entwicklungssystem ermittelten Distanzen der mehrfach Ultraschallechos bei einem Abstand von 7,997 m zur Stirnwand. Man erkennt das Übersprechen des Sendepulses und die zurückkommenden Echos. Das Echo von der Stirnwand wird vom Entwicklungssystem erkannt. Des Weiteren sind andere Echos ebenfalls zu erkennen, welche von den Deckenlampen kommen. Es sind ebenfalls mehrfach reflektierte Echos zu erkennen, bei denen die Distanzen, durch die Mehrfachreflexionen, nicht richtig ermittelt werden.

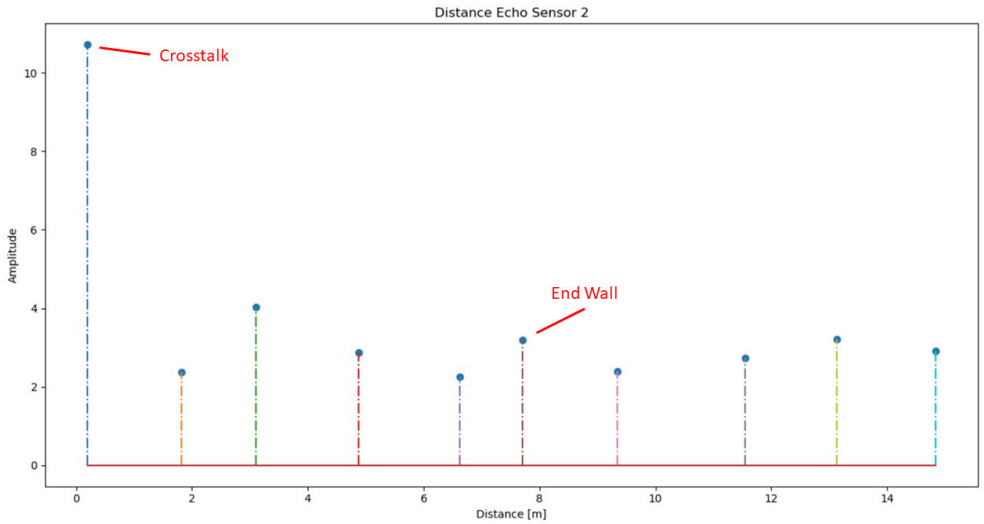


Abbildung 6.14.: Mit dem Entwicklungssystem ermittelte mehrfach Ultraschallechos bei 7,997 m Abstand

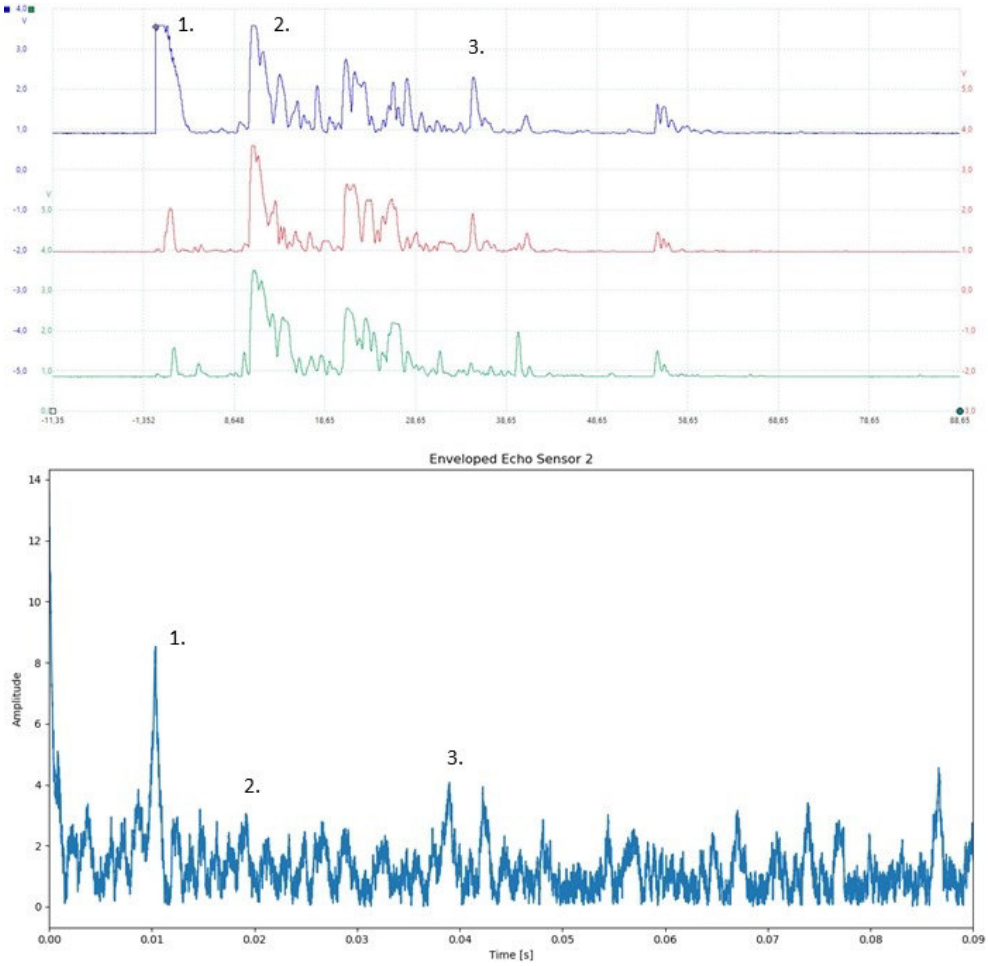


Abbildung 6.15.: Vergleich der Ultraschallechos des Entwicklungssystems mit dem Ultraschall-Array bei 1,897 m Abstand

6.4. Bewertung des Entwicklungssystems

Die Bewertung des Entwicklungssystems erfolgt anhand der analysierten Anforderungen im Abschnitt 3.2 (siehe Tabelle 6.8). Aus dem Test zur Abstandserfassung und den Messdurchlauf unter realen Bedingungen folgt, dass eine Erkennung von Objekten und der Umgebung in komplexen Umgebungen mit dem Entwicklungssystem ermöglicht wird.

Analysiert man den festgelegten Adressraum des Kommunikationsnetzwerks, sind 64516 Netzwerkteilnehmer beziehungsweise 64515 Ultraschallsensoren möglich. Geht man von einem 100 $\frac{Mbit}{s}$-Ethernet-Netzwerk aus, wobei bei einem passenden Switch zwischen dem Switch und dem Rechner auch höhere Datenraten möglich sind, ist die Anzahl der Ultraschallsensoren, bei einem Datenaufkommen inklusive Telegramm-Header von 3,82 $\frac{Mbit}{s}$ pro Sensor, auf 26 begrenzt. Somit wird die Anforderung von mindestens 12 Ultraschallsensoren im Entwicklungssystem erfüllt. Des Weiteren ist durch das Ethernet-Netzwerk und die realisierte Spannungsversorgung der einzelnen Ultraschallsensoren eine variable Mehrfachanordnung gewährleistet.

Die erreichte Bandbreite der Ultraschallsensoren liegt bei 41,5 kHz, wovon sich 23,4 kHz im Ultraschallbereich befinden. Somit wird die Anforderung von über 20 kHz Bandbreite erreicht. Dabei wird mit einer oberen Grenzfrequenz von 43,4 kHz der angestrebte Arbeitsbereich von 30 kHz bis 60 kHz zum Teil erfüllt. Nimmt man den Frequenzbereich der MEMS-Mikrofone als maximale Bandbreite im Empfängerpfad des Ultraschallsensors, wird mit einer ADC-Abtastrate von 200 kHz die Anforderung erfüllt. Im Sendepfad,in dem der Ultraschall-Lautsprecher die höchste Bandbreite besitzt, wird die Anforderung mit einer DAC-Abtastrate von 400 kHz ebenfalls deutlich übertroffen.

Betrachtet man die Genauigkeit der Ultraschallsensoren, weisen diese Abweichungen von unter 1 cm auf. Da die Datenanalyse mit den verwendeten $Python$-Funktionen für die Distanzmessung eine Abweichung von bis zu 9,6 cm aufweist, wird die Anforderung an die Genauigkeit nicht erfüllt. Formt man für die erreichbare Auflösung der Ultraschallsensoren Formel 5.3 um, liegt die Auflösung bei 0,087 cm, wodurch die Anforderung deutlich unterschritten wird. Bewertet man den Test zur Abstandsermittlung und den Messdurchlauf unter realen Bedingungen, wird eine Reichweite von über 5 m erreicht.

Analysiert man die restlichen Tests des Ultraschallsensors, wird die geforderte Synchronität der Ultraschallsensoren untereinander mit einer Zeitdifferenz von 0,06 μs beim Messungsstart beziehungsweise 4,13 μs beim Messungsende deutlich übertroffen. Des Weiteren sind unterschiedliche Ultraschall-Sendepulse im Bezug auf Pulsdauer, Frequenz und Amplitude möglich. Die Aufnahmezeit der Ultraschallechos ist mit maximal 255 ms ausreichend hoch.

Die Anforderung für den Betrieb bei Außentemperaturen wird nicht erfüllt, da die Spezifikation der verwendeten Operationsverstärker eine Mindesttemperatur von 0 °C vorgeben. Des

Weiteren sind für das *Arduino Due* Mikrocontrollerboard und das *Adafruit BMP*280 Board keine Spezifikationen verfügbar, wodurch eine Bewertung des Temperaturbereichs nicht ermöglicht wird. Der Einsatz unter regengeschützten Bedingungen ist mit dem Entwicklungssystem möglich. Bewertet man den Messbereich des *Bosch BMP*280 Sensors, werden die geforderten Erfassungsbereiche der Umgebungsbedingungen mit -40 °C bis 85 °C bei der Temperatur beziehungsweise von 30 kPa bis 110 kPa bei dem Umgebungsdruck eingehalten.

Des Weiteren besitzen die Ultraschallsensoren mit dem Ethernet-Netzwerk eine Verbindung zu einem Rechner, die mit einem Datenverlust von maximal 0,88 ppm über eine ausreichend hohe Übertragungssicherheit verfügt. Die Steuerung des Entwicklungssystems und eine grafische Ausgabe der gemessenen Ultraschallecho-Signale werden über die Datenaufnahme realisiert. Eine *Linux*-Kompatibilität ist durch die Verwendung von *Python* bei der Programmierung der Datenaufnahme gewährleistet. Ebenfalls ist es möglich, die intelligenten Ultraschall-Algorithmen in der Datenaufnahme zu implementieren.

Aus den einzelnen Bewertungen der Anforderungen an das Entwicklungssystem folgt, dass das dieses für den geplanten Einsatz zur Erkennung von Objekten und der Umgebung beziehungsweise zur Lokalisation in komplexen Umgebungen geeignet ist. Hierbei bietet das Entwicklungssystem allerdings noch Optimierungspotenziale.

ID	Beschreibung	Wert	Bewertung
1	Erkennung von Objekten und der Umgebung		✓
2	System aus mehreren Ultraschallsensoren	max. 26 Sensoren	✓
3	Flexibel Mehrfachanordnung der Ultraschallsensoren		✓
4	Ultraschallsensoren mit hoher Frequenzbandbreite	23,4 kHz	✓
5	Ausreichend hohe Abtastrate	ADC 200 kHz DAC 400 kHz	✓
6	Möglichst hohe Genauigkeit der Ultraschallsensoren	Δx 0,2 cm bis 9,6 cm	✗
7	Möglichst hohe Auflösung der Ultraschallsensoren	0,087 cm	✓
8	Möglichst hoher Messbereich der Ultraschallsensoren	> 5 m	✓
9	Ultraschallsensoren müssen synchron arbeiten	Start Δt 0,06 μs Ende Δt 4,13 μs	✓
10	Unterschiedliche Sendesignale (Pulsdauer, Frequenz, Schalldruck)		✓
11	Variable Aufnahmezeit des Echosignals	bis 255 ms	✓
12	Betrieb bei Außentemperaturen	nicht spezifizierbar	✗
13	Betrieb unter regengeschützten Bedingungen		✓
14	Erfassung der Umgebungstemperatur	-40 °C bis 85 °C	✓
15	Erfassung des Umgebungsdrucks	30 kPa bis 110 kPa	✓
16	Schnittstelle zu einem PC	Ethernet-Netzwerk	✓
17	Hohe Übertragungssicherheit der Schnittstelle	0,24 ppm bis 0,88 ppm	✓
18	Steuerung der Ultraschallsensoren über eine Software mit grafischer Bedienoberfläche		✓
19	Grafische Ausgabe der Echosignale		✓
20	*Linux*-Kompatibilität der Steuerungssoftware		✓
21	Implementierung der intelligenten Ultraschall-Algorithmen auf der rechnerseitigen Software		✓

Tabelle 6.8.: Bewertung des Entwicklungssystems anhand der Anforderungen

7. Zusammenfassung

Im Rahmen dieser Arbeit wurde in Entwicklungssystem für intelligente Ultraschallsensoren in Mehrfachanordnung umgesetzt. Das System ermöglicht eine Entwicklung von Algorithmen und Methoden zur Lokalisation und Umgebungserkennung in komplexen Umgebungen. Dazu werden Ultraschall-Pulse ausgesendet und über eine variable Zeit aufgenommen, um so mehrfach Ultraschallechos zu empfangen. Dabei werden die empfangenen Ultraschallechos zu Analysezwecken gespeichert beziehungsweise können direkt ausgewertet werden.

Das Entwicklungssystem besteht aus einer variablen Anzahl an Ultraschallsensoren und einer Software zur Datenaufnahme. Von den Sensoren wird einem der Master-Zustand zugewiesen, der den Messdurchlauf über eine Synchronisationsleitung startet und beendet. Dabei wird der Ultraschall-Puls vom Master ausgesendet. Das empfangene Echosignal wird von jedem Sensor aufgenommen.

Die Mehrfachanordnung des Entwicklungssystems wird über ein Ethernet-Netzwerk als Kommunikationsnetzwerk realisiert. Hierdurch wird eine individuelle Anordnung und Anzahl der Ultraschallsensoren ermöglicht. Durch das Ethernet-Netzwerk ist ebenfalls eine störungsfreie Kommunikation der Netzwerk-Teilnehmer möglich.

Jeder Ultraschallsensor verfügt über einen eigenen Mikrocontroller. Die Pfade zum Senden und Empfangen der Ultraschallsignale sind als zwei getrennte Einheiten realisiert. Der Sendepfad besteht aus einer Verstärkerschaltung und einem Ultraschall-Lautsprecher. Der Empfängerpfad ebenfalls aus einer Verstärkerschaltung und einem MEMS-Mikrofon. Zur Aufnahme der Umgebungsbedingungen wie Temperatur und Umgebungsluftdruck besitzt jeder Ultraschallsensor einen Druck- und Temperatursensor. Zur Kommunikation mit der Datenaufnahme verfügen die Ultraschallsensoren über Ethernet-Module.

Zur Steuerung und Auswertung des Entwicklungssystems wurde die Datenaufnahme realisiert. Hierzu verfügt die Datenaufnahme über eine grafische Benutzeroberfläche. Die Datenaufnahme teilt den Ultraschallsensoren über das Kommunikationsnetzwerk die Parameter des Messdurchlaufs mit und empfängt von diesen die Daten. Für eine Auswertung der Echodaten im Anschluss an eine Messreihe werden diese in individuellen Dateien gespeichert. Eine Auswertung der Echodaten direkt im Anschluss an einen Messdurchlauf können diese bearbeitet und unbearbeitet visualisiert werden. Zur Bearbeitung der Echodaten stehen mehrere Verfahren zur Auswahl.

Abschließend wird ein Test der einzelnen Komponenten und des Gesamtsystems durchgeführt, um die Funktionsfähigkeit des Entwicklungssystems zu überprüfen. Des Weiteren wird eine Bewertung des Entwicklungssystems durchgeführt. Die Tests und die Bewertung orientieren sich dabei an den Anforderungen an das Entwicklungssystem.

Aus den Tests und der Bewertung folgt, dass das realisierte Entwicklungssystem zur Erkennung von Objekten und der Umgebung beziehungsweise zur Lokalisation in komplexen Umgebungen geeignet ist. Das Entwicklungssystem stellt dabei eine Basis für Optimierungen und Erweiterungen im Anschluss an diese Arbeit dar.

8. Ausblick

Die Entwicklung des konzipierten und realisierten Entwicklungssystems für intelligente Ultraschallsensoren ist nach dieser Arbeit noch nicht abgeschlossen. Der Ausblick befasst sich mit Optimierungsätzen für das Entwicklungssystem. Des Weiteren wird die Integration des Entwicklungssystems in ein Robot Operating System (ROS) angesetzt. Abschließend wird auf die Verwendung des Entwicklungssystems eingegangen.

8.1. Optimierungsansatz des Entwicklungssystems

Zunächst wird auf die Optimierungsansätze des Entwicklungssystems eingegangen. Dabei werden der Ultraschallsensor und die Datenaufnahme einzeln betrachtet.

8.1.1. Ultraschallsensor

Als Optimierungsansatz für den Ultraschallsensor bietet sich die Entwicklung einer Leiterplatte (PCB) an.

Bei einem PCB-Design ist man nicht mehr auf die einzelnen Boards und Module angewiesen, da man diese direkt in die einzelnen Sensoren und Prozessoren direkt in die Schaltung integrieren kann. Hierdurch ist man auch nicht mehr an die SPI-Schnittstelle des Ethernet-Moduls gebunden, was einen Verwendung des PDC ermöglicht, was den Prozessor entlastet und die ADC-Abtastrate nicht mehr bei Sendevorgängen des UDP-Pakets verschiebt.

Ein weiterer Vorteil eines PCB-Designs ist, dass eine deutlich kompaktere Bauform möglich ist.

Des Weiteren bietet sich der Ultraschall-Sendepuls als Optimierungsansatz an. Hierbei kann die Sensor-Software dahin gehend optimiert werden, dass die 4 kHz Schwingung des Ultraschall-Lautsprechers minimiert wird. Dabei kann die Software so ausgelegt werden, dass ein Enden des Sendepulses nur im Ursprungsort des Sendepulses möglich ist.

8.1.2. Datenaufnahme

Die Datenaufnahme wird zunächst primär zur Speicherung der empfangenen Echodaten umgesetzt, womit eine spätere Analyse ermöglicht wird. Ein Optimierungsansatz der Datenaufnahme betrifft die Analyse der empfangenen Echodaten direkt im Anschluss an einen Messdurchlauf. Hierbei bietet sich eine Optimierung des Algorithmus für die Berechnung der einhüllenden Kurve des korrelierten Echosignals und der Berechnung der Distanzen an. Durch eine Optimierung durch eine bessere Anpassbarkeit an die Umgebungsbedingungen können deutlich genauere Ergebnisse bei der Distanzermittlung realisiert werden.

Ein weiterer Optimierungsansatz ist die Realisierung einer Schnittstelle, mit der entwickelte intelligente Algorithmen in die Datenaufnahme integriert werden können. Hierdurch wird die Auswertung der empfangenen Echosignale direkt im Anschluss an einen Messdurchlauf mit den entwickelten Algorithmen ermöglicht.

Des Weiteren kann die Datenaufnahme dahin gehend erweitert werden, dass eine Auswertung der Echosignale simultan zur Aufnahme dieser erfolgt. Hierbei bietet sich Multiprocessing an, um das gleichzeitige Empfangen und Auswerten der Echodaten zu realisieren.

8.2. Integration des Entwicklungssystems in ROS

ROS ist ein Open-Source Software-Framework, welches als Betriebssystem für Roboter dient. Mit ROS werden Hardwarebestandteile eines Roboters gesteuert und verwaltet. Des Weiteren ist mit ROS ein Betrieb von mehreren Rechnern im System möglich. Dabei können mit ROS unterschiedliche Stufen der Automatisierung realisiert werden (vgl. O'Kane, 2014, S. 1).

Im Zuge dessen hat das *Urban Mobility Lab* für das automatisierte Fahren einen *Linux*-Rechner, auf dem ein ROS-System implementiert wurde. Hiermit werden Daten eines Lidar-Sensors und einer 360 °-Kamera ausgewertet und korrigiert. Des Weiteren dient das System zur Erstellung einer Umgebungskarte und Lokalisation (vgl. Vater, 2018, S. 3).

Zur Erweiterung kann das Entwicklungssystem für intelligente Ultraschallsensoren in das ROS-System integriert werden. Hierdurch würden dem ROS-System mehr Informationen zur Verfügung gestellt, wodurch eine Erweiterung der Umgebungskarte um den Ultranahbereich des Fahrzeuges bis 2,5 m möglich ist. Beziehungsweise kann somit die Umgebungskarte um verdeckte Hindernisse erweitert werden. Des Weiteren könnte so das Entwicklungssystem auch einzeln gesteuert werden.

8.3. Low speed surround sensing, localization and navigation in variable, complex environments

Im Anschluss an diese Arbeit soll das Entwicklungssystem innerhalb des *Urban Mobility Lab* im Projekt *Low speed sensing, localization and navigation in variable, complex environments* eingesetzt werden.

Hierbei sollen im Bereich des autonomen Fahrens Methoden und Algorithmen zur Erkennung und Lokalisation von Objekten in komplexen Umgebungen, zum Beispiel durch eine Wand verdeckte Fahrzeuge, realisiert werden. Des Weiteren soll hiermit eine Erkennung der Umgebung ermöglicht werden. Ein weiterer Punkt in diesem Projekt ist die Navigation von Fahrzeugen in komplexen Umgebungen bei niedrigen Geschwindigkeiten mithilfe von Ultraschall.

Durch die Methoden und Algorithmen sollen autonome beziehungsweise teilautonome Fahrzeuge das Umfeld eigenständig durch Ultraschall erkennen und auswerten. Bei den komplexen Umgebungen werden ebenfalls vom Fahrzeug unbekannte Umgebungen eingeschlossen.

Hierzu sollen, anders als im Abschnitt 2.3.1 beschrieben, keine festen Ultraschall-Sender in der Umgebung verwendet werden. Stattdessen sollen Fahrzeuge nur auf die im Fahrzeug verbauten Ultraschallsensoren zurückgreifen.

Abkürzungsverzeichnis

ADC	Analog to Digital Converter
ASF	Atmel Software Framework
CAN	Controller Area Network
CMUT	Capacitive Micromachined Ultrasound Transducer
CSMA/CA	Carrier Sense Multiple Access / Collision Avoidance
CSMA/CD	Carrier Sense Multiple Access / Collision Detection
DAC	Digital to Analog Converter
FAS	Fahrerassistenzsystem
FIFO	First In First Out
FRA	Frequency Response Analyzer for PicoScope
IANA	Internet Assigned Numbers Authority
ISO-OSI	International Organization for Standardization - Open Systems Interconnection Model
I2C	Inter Integrated Circuit
GUI	Graphical User Interface
MEMS	Microelectromechanical System
PCB	Printed Circuit Board
PDC	Peripheral DMA Controller
PWM	Pulsweitenmodulation
ROS	Robot Operating System
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
TCP	Transmission Control Protocol
TOF	Time of Flight
UDP	User Datagram Protocol
USB	Universal Serial Bus
WLAN	Wireless Local Area Network

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

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

[Arduino 2018] ARDUINO: *Website Arduino Due*. 2018. – URL https://store.arduino.cc/usa/arduino-due. – Abruf: 2018-02-28

[Atmel 2015] ATMEL: *Atmel SAM3X / SAM3A Series Datasheet*. 2015. – URL http://ww1.microchip.com/downloads/en/devicedoc/atmel-11057-32-bit-cortex-m3-microcontroller-sam3x-sam3a_datasheet.pdf. – Abruf:2018-02-28

[Atmel 2018] ATMEL: *Website Atmel Software Framework*. 2018. – URL http://asf.atmel.com/docs/latest/. – Abruf: 2018-05-04

[BASt 2012] BAST: *Rechtsfolgen zunehmender Fahrzeugautomatisierung*. 2012. – URL http://www.bast.de/DE/Publikationen/Foko/Downloads/2012-11.pdf?__blob=publicationFile. – Abruf: 2018-05-17

[Bosch-Sensortec 2017] BOSCH-SENSORTEC: *Website Bosch BMP280 Library*. 2017. – URL https://github.com/BoschSensortec/BMP280_driver. – Abruf:2018-03-28

[Bosch-Sensortec 2018] BOSCH-SENSORTEC: *Bosch BMP280 Digital Pressure Sensor Datasheet*. 2018. – URL https://ae-bst.resource.bosch.com/media/_tech/media/datasheets/BST-BMP280-DS001-19.pdf. – Abruf: 2018-03-28

[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

[Flühr 2012] FLÜHR, Holger: *Avionik und Flugsicherungstechnik: Einführung in Kommunikationstechnik, Navigation, Surveilance*. 2. erw. Aufl. Berlin : Springer Vieweg, 2012. – URL https://link.springer.com/book/10.1007%2F978-3-642-33576-1. – Abruf: 2018-05-18. – ISBN 978-3-642-33576-1

[Freescale-Semiconductor 2006] FREESCALE-SEMICONDUCTOR: *Freescale Semiconductor MPX4115A Datasheet*. 2006. – URL https://www.nxp.com/docs/en/data-sheet/MPX4115.pdf. – Abruf: 2018-03-25

[Gessler und Krause 2015] GESSLER, Ralf ; KRAUSE, Thomas: *Wireless-Netzwerke für den Nahbereich*. 2. akt. u. erw. Aufl. Wiesbaden : Springer Vieweg, 2015. – URL https://link.springer.com/book/10.1007%2F978-3-8348-2075-4. – Abruf: 2018-03-08. – ISBN 978-3-8348-2075-4

[Hansestadt-Hamburg 2018] HANSESTADT-HAMBURG: *Website Luftqualität Hamburg*. 2018. – URL luft.hamburg.de/clp/luftdruck/clp1/. – Abruf: 2018-05-11

[Hazas und Hopper 2006] HAZAS, Mike ; HOPPER, Andy: Broadband Ultrasonic Location Systems for Improved Indoor Positioning. In: *IEEE Transactions on Mobile Computing* Vol. 5 (2006), Nr. 5, S. 536 – 547. – URL https://ieeexplore.ieee.org/document/1610595/. – Abruf: 2018-05-03. – ISSN 1536-1233

[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

[IANA 1996] IANA: *Website Address Allocatoin for Private Internets*. 1996. – URL https://tools.ietf.org/html/rfc1918. – Abruf: 2018-05-01

[Ingle und Phute 2016] INGLE, Shantanu ; PHUTE, Madhuri: Tesla Autopilot: Semi Autonomous Driving, an Uptick for Future Autonomy. In: *International Research Journal of Engineering and Technology* Vol. 03 (2016), Nr. 09, S. 369 – 372. – URL https://irjet.net/archives/V3/i9/IRJET-V3I969.pdf. – Abruf: 2018-04-28. – ISSN 2395-0056

[Karrenberg 2017] KARRENBERG, Ulrich: *Signale - Prozesse - Systeme: Eine multimediale und interaktive Einführung in die Signalverarbeitung*. 7. n. bearb. u. erw. Aufl. Berlin : Springer Vieweg, 2017. – URL https://link.springer.com/book/10.1007%2F978-3-662-52659-0. – Abruf: 2018-05-11. – ISBN 978-3-662-52659-0

[Kemo-Electronic 2009] KEMO-ELECTRONIC: *Website Kemo Electronic L010*. 2009. – URL https://www.kemo-electronic.de/de/Auto/Lautsprecher/L010-Ultraschall-Piezo-Lautsprecher.php. – Abruf: 2018-03-06

[Knowles-Electronics 2013] KNOWLES-ELECTRONICS: *Knowles SPU0410LR5H Datasheet*. 2013. – URL http://www.knowles.com/kor/content/download/5755/91802/version/3/file/SPU0410LR5H-QB+revH.PDF. – Abruf: 2018-03-07

[Kölln 2015] KÖLLN, Niklas: *Bachelorthesis: Entwicklung eines Ultraschallsensors mit Objektklassifizierung*. Hamburg : HAW Hamburg, 2015. – URL http://edoc.sub.uni-hamburg.de/haw/volltexte/2015/3193/. – Abruf: 2018-02-27

[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

[Matplot 2012] MATPLOT: *Website matplotlib*. 2012. – URL https://matplotlib.org/. – Abruf: 2018-03-08

[MaxBotix 2005] MAXBOTIX: *MaxSonar Series Datasheet*. 2005. – URL https://www.maxbotix.com/documents/LV-MaxSonar-EZ_Datasheet.pdf. – Abruf: 2018-05-15

[Microchip 2006] MICROCHIP: *Microchip ENC28J60 Datasheet*. 2006. – URL https://www.elecrow.com/download/ENC28J60%20Datasheet.pdf. – Abruf: 2018-03-24

[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

[O'Kane 2014] O'KANE, Jason M.: *A Gentle Introduction to ROS*. 2.1.6. Vers. Columbia : University of South Carolina, 2014. – URL https://www.cse.sc.edu/~jokane/agitr/agitr-letter.pdf. – Abruf: 2018-05-14. – ISBN 978-14-92143-23-9

[Pico-Technology 2018] PICO-TECHNOLOGY: *Website Frequency Response Analyzer for PicoScope*. 2018. – URL https://www.picotech.com/library/picoapp/frequency-response-analyzer-with-bode-plots. – Abruf:2018-03-26

[ProWave-Electronic 2005] PROWAVE-ELECTRONIC: *Air Ultrasonic Ceramic Transducers 500MB120 Datasheet*. 2005. – URL www.prowave.com.tw/pdf/T500MB.PDF. – Abruf:2018-05-17

[Reif 2010] REIF, Konrad: *Fahrstabilisierungssysteme und Fahrerassistenzsysteme*. Wiesbaden : Vieweg + Teubner, 2010. – URL https://link.springer.com/book/10.1007%2F978-3-8348-9717-6. – Abruf: 2018-02-28. – ISBN 978-3-8348-1314-5

[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

[Reißenweber 2009] REISSENWEBER, Bernd: *Feldbussysteme zur industriellen Kommunikation*. 3. vollst. überarb. Aufl. Oldenbourd, 2009. – ISBN 978-3-8356-3143-4

[Schöne 2017] SCHÖNE, Maik C.: *Masterarbeit: Entwicklung eines modularen Datenloggers für den autonomen Einsatz in Fahrzeugen*. Hamburg : HAW Hamburg, 2017

[SensComp 2013] SENSCOMP: *SensComp Series 600 Instrument Grade Ultrasonic Sensor Datasheet*. 2013. – URL http://www.senscomp.com/pdfs/series-600-instr-grade-ultrasonic-sensor-spec.pdf. – Abruf: 2018-03-05

[Siciliano und Khatib 2016] SICILIANO, Bruno ; KHATIB, Oussama: *Springer Handbook of Robotics*. 2. Aufl. Berlin : Springer, 2016. – ISBN 978-3-319-32550-7

[Tietze u. a. 2016] TIETZE, Ulrich ; SCHENK, Christoph ; GAMM, Eberhard: *Halbleiter-Schaltungstechnik*. 15. überarb. u. erw. Aufl. Heidelberg : Springer Vieweg, 2016. – ISBN 978-3-662-48354-1

[Vater 2018] VATER, Tobias: *Bachelorthesis: Inline-Auswertung und -Korrektur multipler, komplexer, fusionierter, bildgebender Sensoren für das automatisierte Fahren*. Hamburg : HAW Hamburg, 2018

[Vogel Heuser 2003] VOGEL HEUSER, Birgit: *Systems Software Engineering: Angewandte Methoden des Systementwurfs für Ingenieure*. München : Oldenbourg, 2003. – ISBN 3-486-27035-4

[Wenzel 2016] WENZEL, Tobias M.: *Masterthesis: Entwicklung eines synchronen hochkanaligen Messsystems für Luft- und Körperschall*. Hamburg : HAW Hamburg, 2016. – URL http://edoc.sub.uni-hamburg.de/haw/volltexte/2016/3540/. – Abruf: 2018-03-05

[Wikipedia 2018] WIKIPEDIA: *Website Python (Programmiersprache)*. 2018. – URL https://de.wikipedia.org/wiki/Python_(Programmiersprache). – Abruf: 2018-05-14

[WIZnet 2013] WIZNET: *WIZnet W5500 Datasheet*. 2013. – URL http://wizwiki.net/wiki/lib/exe/fetch.php?media=products:w5500:w5500_ds_v106e_141230.pdf. – Abruf: 2018-03-24

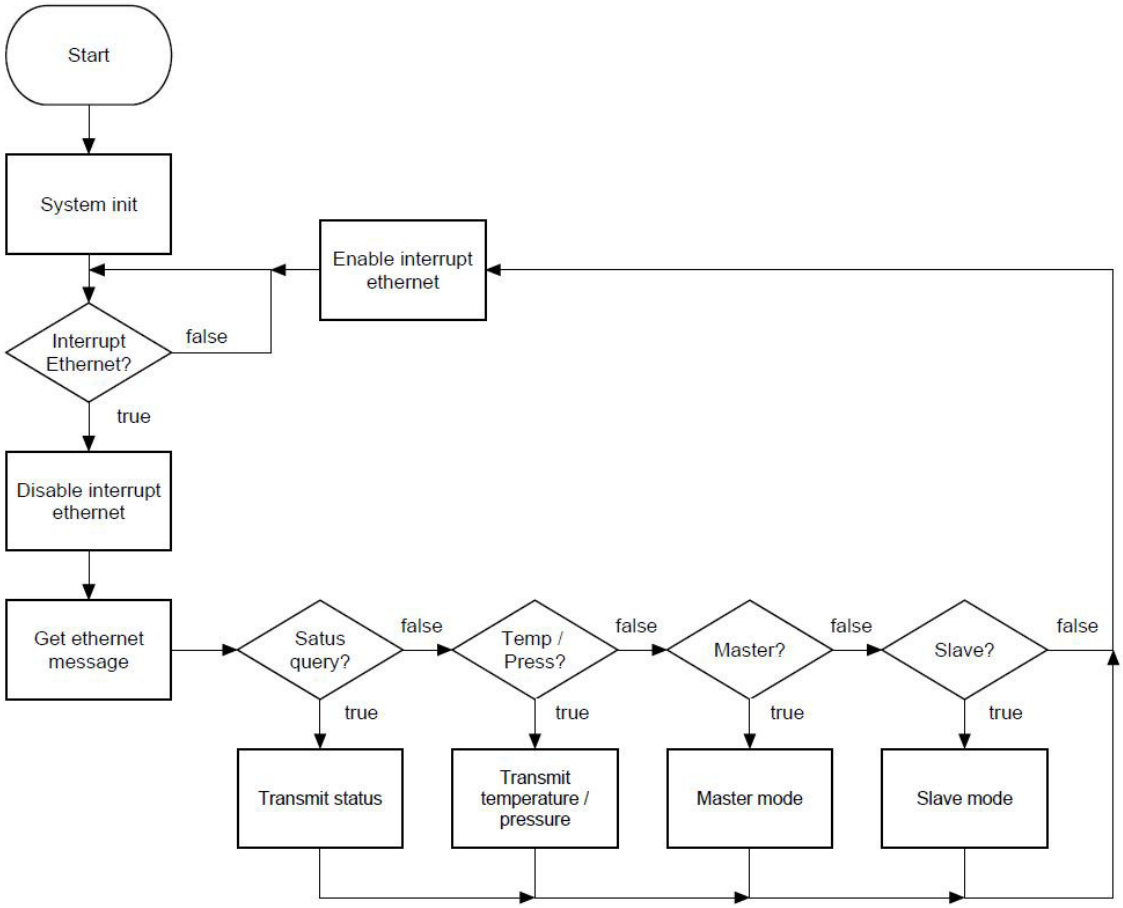
[WIZnet 2017] WIZNET: *Website W5500 Library*. 2017. – URL https://github.com/Wiznet/ioLibrary_Driver/tree/master/Ethernet/W5500. – Abruf:2018-03-28

[XP-Power 2014] XP-POWER: *XP JHM10 Datasheet*. 2014. – URL https://www.xppower.com/pdfs/SF_JHM10.pdf. – Abruf: 2018-05-24

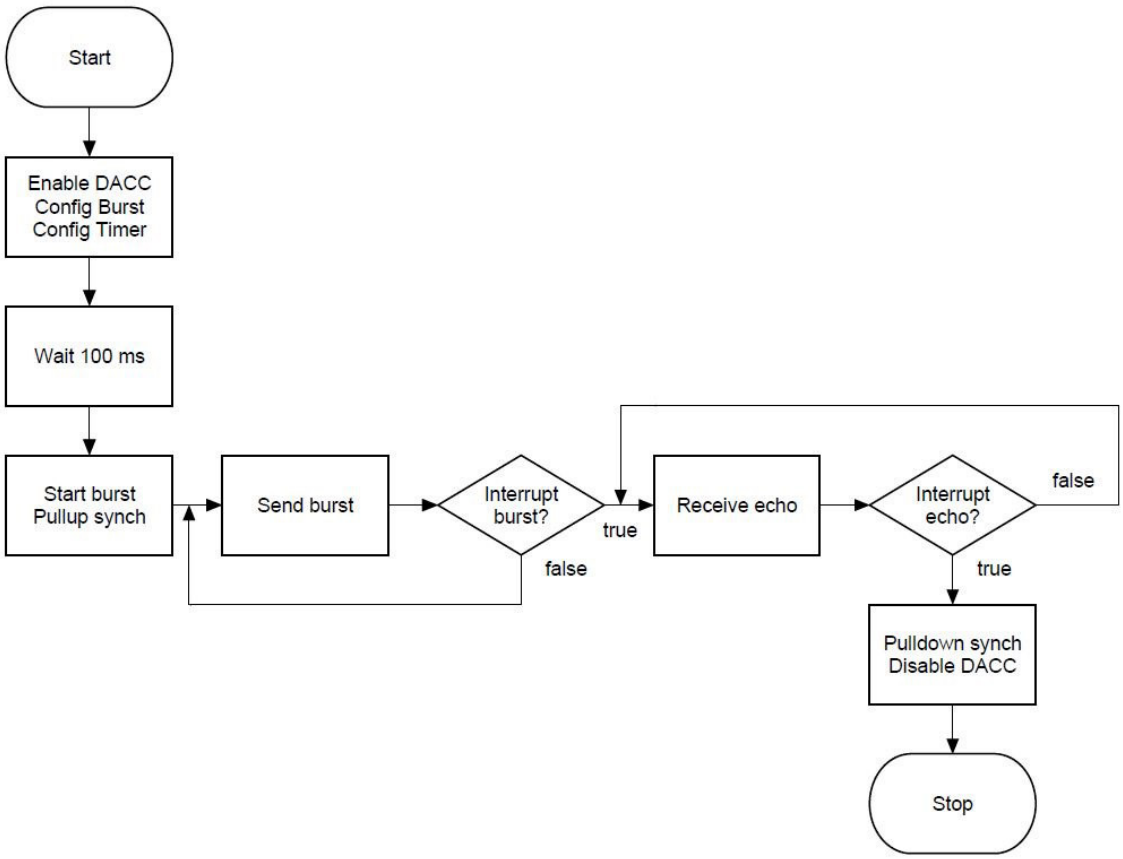
[Zimmermann und Schmidgall 2014] ZIMMERMANN, Werner ; SCHMIDGALL, Ralf: *Bussysteme in der Fahrzeugtechnik: Protokolle, Standards und Softwarearchitektur*. 5. akt. u. erw. Aufl. Wiesbaen : Springer Vieweg, 2014. – URL https://link.springer.com/book/10.1007%2F978-3-658-02419-2. – Abruf: 2018-03-11. – ISBN 978-3-658-02419-2

A. Flussdiagramme des Ultraschallsensors

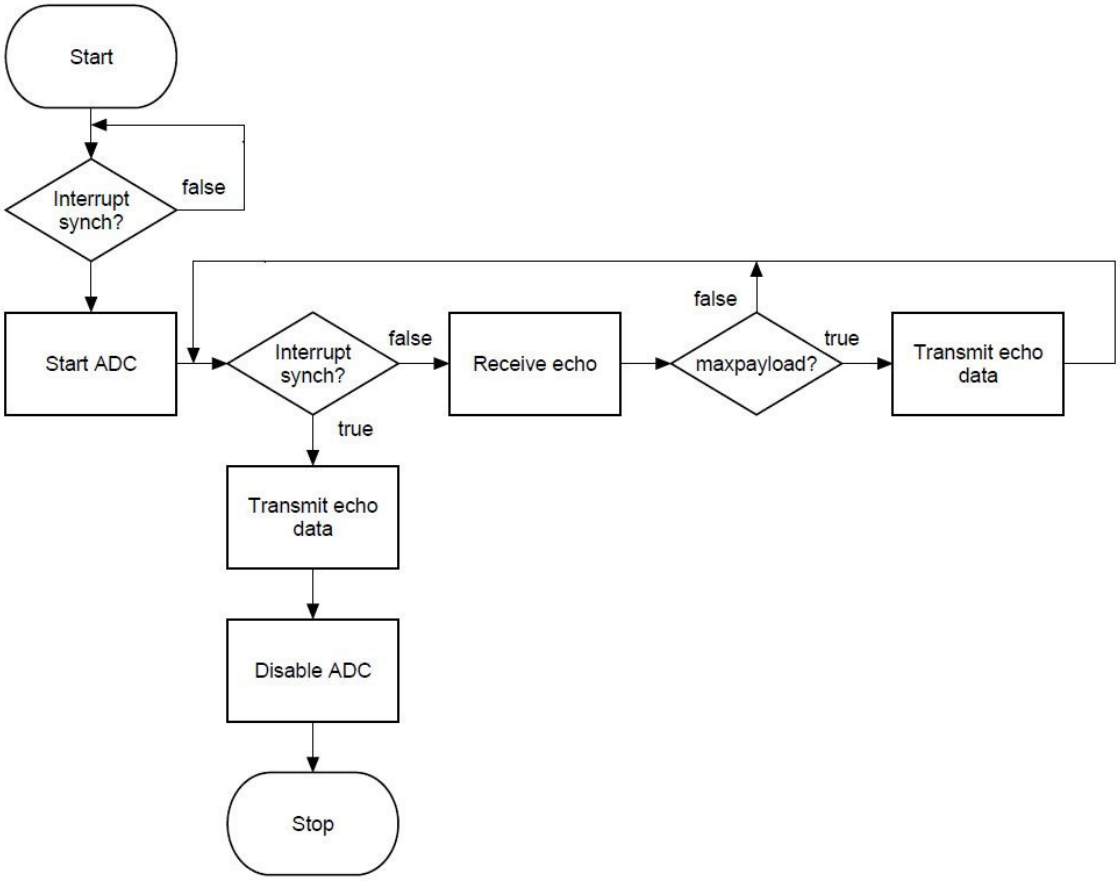
A.1. Main Ablauf



A.2. Master Ablauf

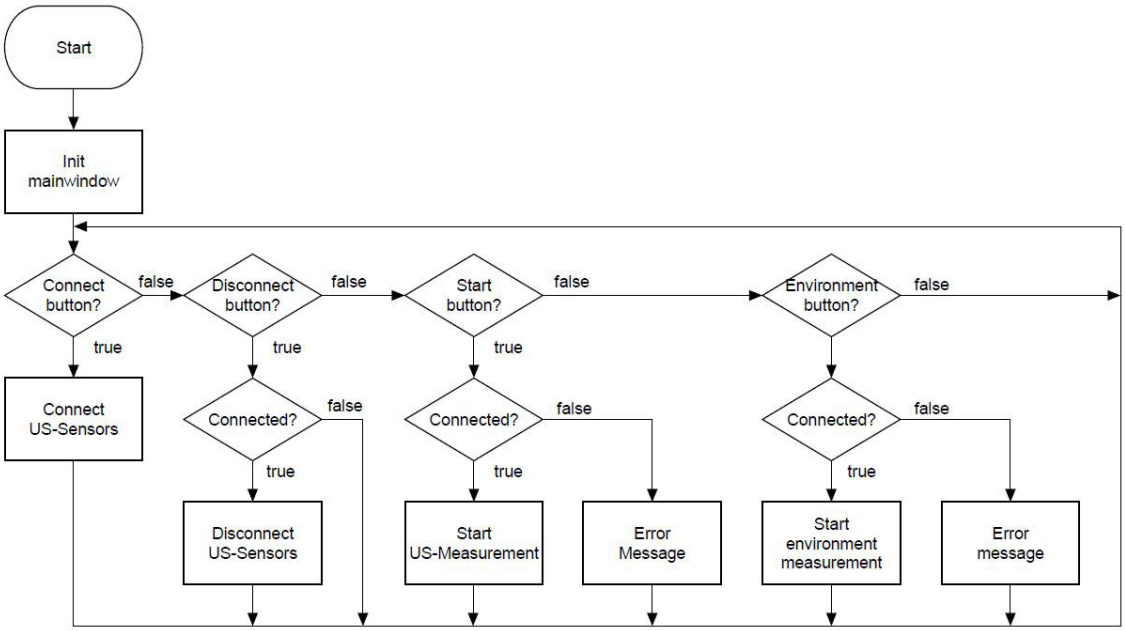


A.3. Slave Ablauf

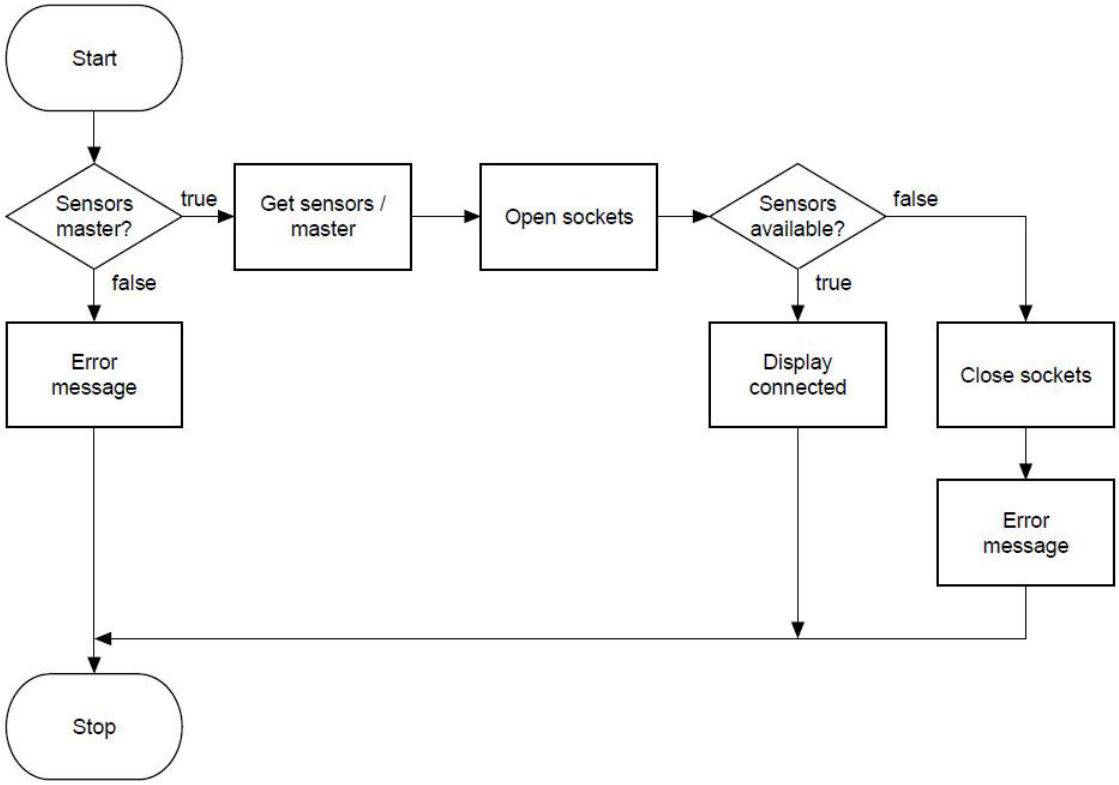


B. Flussdiagramme der Datenaufnahme

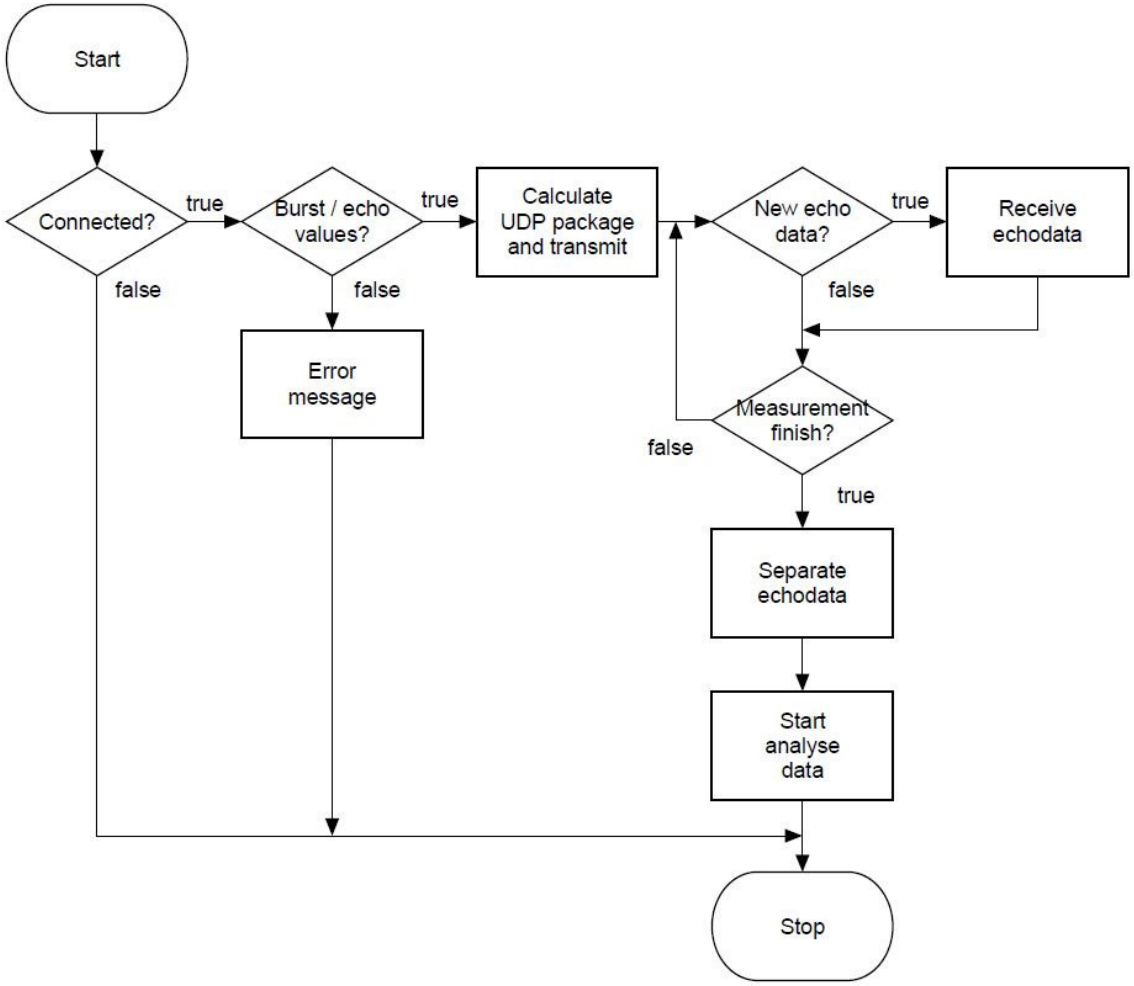
B.1. Main Ablauf



B.2. Verbindungsaufbau Ablauf



B.3. Messdurchlauf Ablauf



C. Quellcode des Ultraschallsensors

C.1. 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 <signal_lookup_table.h>
12 #include <w5500.h>
13 #include <bmp280.h>
14
15 /*****************************************************************/
16 /*                          Definitions                          */
17 /*****************************************************************/
18 // Sensor number
19 #define SENSOR 2
20 // Sensor mode definitions
21 #define STATUS_MODE 0x80000000
22 #define TEMP_PRESS_MODE 0x40000000
23 #define MASTER_MODE 0x20000000
24 #define SLAVE_MODE 0x10000000
25 #define SWEEP_MODE 0x8000000
26 #define BURST_AMPLITUDE 0x7800000
27 #define BURST_AMPLITUDE_SHIFT 23
28 #define BURST_FREQUENCY 0x7F000
29 #define BURST_STEP_SHIFT 12
30 #define BURST_TIME 0xFFF
31 #define ECHO_TIME 0xFF
32 #define STATUS_READY_MESSAGE 0xF0F0
33 // General definitions
```

```
34 #define MEGA_HZ 1000000
35 #define KILO_HZ 1000
36 #define DIV_MS 1000
37 #define DIV_US 1000000
38 #define BYTE_0_MASK 0xFF
39 #define BYTE_1_MASK 0xFF00
40 #define BYTE_2_MASK 0xFF0000
41 #define BYTE_3_MASK 0xFF000000
42 #define BYTE_0_SHIFT 0
43 #define BYTE_1_SHIFT 8
44 #define BYTE_2_SHIFT 16
45 #define BYTE_3_SHIFT 24
46 // SPI definitions
47 #define SPI_CLOCK_RECEIVE 21
48 #define SPI_CLOCK_TRANSMIT 28
49 #define SPI_CHIP_SEL 0
50 #define SPI_CHIP_PCS spi_get_pcs(SPI_CHIP_SEL)
51 #define SPI_CLK_POLARITY 1
52 #define SPI_CLK_PHASE 0
53 #define SPI_DLYBS 0x00
54 #define SPI_DLYBCT 0x00
55 // TWI definitions
56 #define TWI_CLK 400000
57 // Ethernet Shield definitions
58 #define MAX_DATA_FRAME_SIZE 20
59 #define BUFFER_ECHO_SIZE_KB 8
60 #define BUFFER_TEMP_PRESS_KB 2
61 #define BUFFER_STATUS_KB 2
62 #define BUFFER_RX_SIZE_KB 2
63 #define MAX_ETHERNET_PAYLOAD 1470
64 #define TRANSMIT_ECHO_SOCKET 0
65 #define TRANSMIT_TEMP_PRESS_SOCKET 1
66 #define TRANSMIT_STATUS_SOCKET 2
67 #define RECEIVE_SOCKET TRANSMIT_ECHO_SOCKET
68 #define SEND_MANUAL 1
69 #define SEND_AUTO 0
70 // DAC definitions
71 #define DACC_CHANNEL 1
72 #define DACC_ANALOG_CONTROL (DACC_ACR_IBCTLCH0(0x02)\
73         |DACC_ACR_IBCTLCH1(0x02)|DACC_ACR_IBCTLDACCORE(0x01))
74 // ADC definitions
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 MASTER_WAIT 100          // Master wait time in ms
83  #define TC_CHANNEL0 0
84  #define TC_CHANNEL1 1
85  #define TC_CHANNEL2 2
86  #define TC_CHANNEL3 3
87
88  /*****************************************************************/
89  /*                          Declarations                         */
90  /*****************************************************************/
91  // Network informations of the Sensor
92  wiz_NetInfo Sensor_Info = {
93      .mac = {0x42,0x42,0x42,0x42,0x42,SENSOR},
94      .ip = {192,168,1,(SENSOR+1)},
95      .sn = {255,255,0,0},
96      .gw = {192,168,0,0},
97  };
98  uint8_t destination_ip[4] = {192,168,1,1};
99  uint8_t port_base[2] = {0xC3,0x50};      // 50000
100 // Burst variables
101 uint8_t burst_mode;
102 uint8_t burst_steps;
103 uint8_t burst_amplit;
104 uint32_t sine_index = 0;
105
106 /*****************************************************************/
107 /*                      Function Prototypes                      */
108 /*****************************************************************/
109 static void system_init(void);
110 static void spi_master_init(void);
111 static void spi_transmit(uint32_t addr,uint8_t *data,uint8_t data_len);
112 static void spi_receive(uint32_t addr,uint8_t *data,uint8_t data_len);
113 static void ethernet_init(void);
114 static void ethernet_init_socket(uint8_t socket,uint8_t buffer_size);
115 static void ethernet_transmit_echo(uint16_t *data,uint8_t send_manuell);
116 static void ethernet_transmit_data(uint32_t *data,uint8_t socket,\
117                                    uint8_t data_len);
118 static void ethernet_receive_data(uint32_t *data);
119 static void twi_init(void);
120 static void bmp280_sensor_init(void);
121 static void bmp280_get_calibration(uint8_t *bmp280_calib);
```

```
122 static void bmp280_measurement(void);
123 static void dacc_init(void);
124 static void adc_init_config(void);
125 static void timer_init(void);
126 static void nvicc_init(void);
127 static void Int_Handler_Ethernet(uint32_t ul_id,uint32_t ul_mask);
128 static void Int_Handler_Sync(uint32_t ul_id,uint32_t ul_mask);
129
130 /*************************************************************************/
131 /*                        Main Function                                  */
132 /*************************************************************************/
133 int main(void)
134 {
135     // Init microcontroller
136     system_init();
137
138     while (1)
139     {
140
141     }
142
143     return 0;
144 }
145
146 /*************************************************************************/
147 /*                        Interrupt Handler                              */
148 /*************************************************************************/
149 // DACC handler
150 void SysTick_Handler(void)
151 {
152     uint32_t status;
153     uint32_t dac_val;
154
155     status = dacc_get_interrupt_status(DACC);
156
157     if ((status&DACC_ISR_TXRDY) == DACC_ISR_TXRDY) {
158         // Get value form look up table
159         if (burst_mode)
160         {
161             dac_val = sweep_signal[sine_index];
162         }
163         else
164         {
165             //dac_val = sine_data[sine_index];
```

```
166             dac_val = (sine_data[sine_index] - 0x800) * burst_amplit\
167                       / 0xF + 0x800;
168         }
169         dacc_write_conversion_data(DACC,dac_val);
170         sine_index = sine_index + burst_steps;
171 
172         // If index out of range
173         if ((sine_index >= SINE_SAMPLES) && !burst_mode) {
174             sine_index = sine_index - SINE_SAMPLES;
175         }
176     }
177 }
178 
179 // ADC handler
180 void ADC_Handler(void)
181 {
182     static uint16_t adc_value;
183 
184     // Get value from register
185     adc_value = (uint16_t)(ADC->ADC_LCDR & ADC_LCDR_LDATA_Msk);
186 
187     // Transmit value via ethernet in automatic mode
188     ethernet_transmit_echo(&adc_value,SEND_AUTO);
189 }
190 
191 // Timer0 A1 master wait
192 void TC1_Handler(void)
193 {
194     // Set sync pin high
195     ioport_set_pin_level(PIO_PB26_IDX,IOPORT_PIN_LEVEL_HIGH);
196 
197     // Stop and disable Timer0 A1 master wait
198     tc_get_status(TC0,TC_CHANNEL1);
199     tc_stop(TC0,TC_CHANNEL1);
200     tc_disable_interrupt(TC0,TC_CHANNEL1,TC_IER_CPCS);
201 
202     // Enable and start Timer0 A2 burst time
203     tc_enable_interrupt(TC0,TC_CHANNEL2,TC_IER_CPCS);
204     tc_start(TC0,TC_CHANNEL2);
205 
206     // Enable and start Timer1 A3 echo time
207     tc_enable_interrupt(TC1,TC_CHANNEL0,TC_IER_CPCS);
208     tc_start(TC1,TC_CHANNEL0);
209 
```

```
210     // Start SysTick for DACC
211     SysTick_Config(sysclk_get_cpu_hz() / (SAMPLE_RATE_DAC_KHZ*KILO_HZ));
212 }
213
214 // Timer0 A2 burst time
215 void TC2_Handler(void)
216 {
217     // Stop and disable Timer0 A2 burst time
218     tc_get_status(TC0,TC_CHANNEL2);
219     tc_stop(TC0,TC_CHANNEL2);
220     tc_disable_interrupt(TC0,TC_CHANNEL2,TC_IER_CPCS);
221
222     // Disable SysTick
223     SysTick->CTRL &= ~SysTick_CTRL_ENABLE_Msk;
224
225     // Start Timer0 A0 ADC
226     tc_start(TC0,TC_CHANNEL0);
227 }
228
229 // Timer1 A3 echo time
230 void TC3_Handler(void)
231 {
232     uint32_t data_transmit[1];
233
234     ioport_set_pin_level(PIO_PC25_IDX,IOPORT_PIN_LEVEL_LOW);
235
236     uint16_t null_value = 0x0;
237
238     // Set sync pin low
239     ioport_set_pin_level(PIO_PB26_IDX,IOPORT_PIN_LEVEL_LOW);
240
241     // Stop Timer0 A0 ADC
242     tc_stop(TC0,TC_CHANNEL0);
243
244     // Stop and disable Timer1 A3 echo time
245     tc_get_status(TC1,TC_CHANNEL0);
246     tc_stop(TC1,TC_CHANNEL0);
247     tc_disable_interrupt(TC1,TC_CHANNEL0,TC_IER_CPCS);
248
249     // Disable DACC
250     dacc_disable_channel(DACC,DACC_CHANNEL);
251
252     // Enable interrupt ethernet shield
253     pio_enable_interrupt(PIOB,PIO_PB25);
```

```
254
255     // Transmit remaining echo data
256     ethernet_transmit_echo(&null_value,SEND_MANUAL);
257
258     // Set SPI baudrate to receive data
259     spi_set_baudrate_div(SPI_MASTER_BASE,SPI_CHIP_SEL,\
260                          (sysclk_get_peripheral_hz() /\
261                          SPI_CLOCK_RECEIVE / MEGA_HZ));
262
263
264     // Response for status end measurement
265     data_transmit[0] = 0xF0;
266
267     // Transmit status response
268     ethernet_transmit_data(&data_transmit[0],\
269                            TRANSMIT_STATUS_SOCKET,1);
270 }
271
272 // Ethernet shield handler
273 static void Int_Handler_Ethernet(uint32_t ul_id,uint32_t ul_mask)
274 {
275     uint8_t clear_reg[1] = {Sn_IR_RECV};
276     uint32_t data_receive[2];
277     uint32_t data_transmit[1];
278     uint32_t burst_time;
279     uint32_t echo_time;
280
281     // Disable interrupt ethernet shield
282     pio_disable_interrupt(PIOB,PIO_PB25);
283
284     // Get ethernet data
285     ethernet_receive_data(&data_receive[0]);
286     spi_transmit((Sn_IR(RECEIVE_SOCKET) | _W5500_SPI_WRITE_),\
287                  &clear_reg[0],1);
288
289     // Status query
290     if (data_receive[0] & STATUS_MODE)
291     {
292         // Response for status query
293         data_transmit[0] = 0xAA;
294
295         // Transmit response
296         ethernet_transmit_data(&data_transmit[0],\
297                                TRANSMIT_STATUS_SOCKET,1);
```

```
298
299         // Enable interrupt ethernet shield
300         pio_enable_interrupt(PIOB,PIO_PB25);
301     }
302     // Temperature and Pressure query
303     else if (data_receive[0] & TEMP_PRESS_MODE)
304     {
305         // Start measuring and transmit data
306         bmp280_measurement();
307
308         // Enable interrupt ethernet shield
309         pio_enable_interrupt(PIOB,PIO_PB25);
310     }
311     // Master mode
312     else if (data_receive[0] & MASTER_MODE)
313     {
314         // Set SPI baudrate to transmit data
315         spi_set_baudrate_div(SPI_MASTER_BASE, SPI_CHIP_SEL,\
316                              (sysclk_get_peripheral_hz() /\
317                              SPI_CLOCK_TRANSMIT / MEGA_HZ));
318
319         // Enable DACC
320         dacc_enable_channel(DACC,DACC_CHANNEL);
321         dacc_write_conversion_data(DACC,0x800);
322
323         // Set sync to output and low
324         ioport_set_pin_dir(PIO_PB26_IDX,IOPORT_DIR_OUTPUT);
325         ioport_set_pin_level(PIO_PB26_IDX,IOPORT_PIN_LEVEL_LOW);
326
327         // Set index for sine wave
328         sine_index = 0;
329
330         // Set burst amplitude
331         burst_amplit = (data_receive[0] & BURST_AMPLITUDE)\
332                        >> BURST_AMPLITUDE_SHIFT;
333
334         // Sweep frequency
335         if (data_receive[0] & SWEEP_MODE)
336         {
337             // Set burst signal
338             burst_mode = 1;
339             burst_steps = 1;
340
341             // Set burst time
```

```
342             burst_time = sysclk_get_cpu_hz()/TIMER_MCK_2/\
343                          SAMPLE_RATE_DAC_KHZ/KILO_HZ*SWEEP_SAMPLES;
344             TC0->TC_CHANNEL[TC_CHANNEL2].TC_RA = burst_time/2;
345             TC0->TC_CHANNEL[TC_CHANNEL2].TC_RC = burst_time;
346         }
347         // Const frequency
348         else
349         {
350             // Set burst signal
351             burst_mode = 0;
352             // Burst steps equivalent to burst-frequency in kHz
353             burst_steps = (data_receive[0] & BURST_FREQUENCY)\
354                           >> BURST_STEP_SHIFT;
355
356             // Set burst time
357             burst_time = sysclk_get_cpu_hz() / DIV_US / TIMER_MCK_2 *\
358                          (data_receive[0] & BURST_TIME);
359             TC0->TC_CHANNEL[TC_CHANNEL2].TC_RA = burst_time / 2;
360             TC0->TC_CHANNEL[TC_CHANNEL2].TC_RC = burst_time;
361         }
362
363         // Set echo time
364         echo_time = sysclk_get_cpu_hz() / DIV_MS / TIMER_MCK_2 *\
365                     (data_receive[1] & ECHO_TIME);
366         TC1->TC_CHANNEL[TC_CHANNEL0].TC_RA = echo_time / 2;
367         TC1->TC_CHANNEL[TC_CHANNEL0].TC_RC = echo_time;
368
369         // Start timer0 A1 master wait
370         tc_enable_interrupt(TC0,TC_CHANNEL1,TC_IER_CPCS);
371         tc_start(TC0,TC_CHANNEL1);
372     }
373     // Slave mode
374     else if (data_receive[0] & SLAVE_MODE)
375     {
376         // Set SPI baudrate to transmit data
377         spi_set_baudrate_div(SPI_MASTER_BASE, SPI_CHIP_SEL,\
378                              (sysclk_get_peripheral_hz() /\
379                              SPI_CLOCK_TRANSMIT / MEGA_HZ));
380
381         // Set sync to input and enable interrupt
382         ioport_set_pin_dir(PIO_PB26_IDX,IOPORT_DIR_INPUT);
383         pio_enable_interrupt(PIOB,PIO_PB26);
384     }
385     // Default
```

```
386     else
387     {
388         // Enable interrupt ethernet shield
389         pio_enable_interrupt(PIOB,PIO_PB25);
390     }
391 }
392
393 // Sync handler slave
394 static void Int_Handler_Sync(uint32_t ul_id,uint32_t ul_mask)
395 {
396     uint16_t null_value = 0x0;
397     uint32_t data_transmit[1];
398
399     // If pin level high, start ADC
400     if (ioport_get_pin_level(PIO_PB26_IDX))
401     {
402         // Change interrupt edge
403         pio_disable_interrupt(PIOB,PIO_PB26);
404         pio_handler_set(PIOB,ID_PIOB,PIO_PB26,(PIO_PULLUP\
405                         | PIO_IT_FALL_EDGE),Int_Handler_Sync);
406         pio_enable_interrupt(PIOB,PIO_PB26);
407
408         // Start timer0 A0 ADC
409         tc_start(TC0,TC_CHANNEL0);
410     }
411     // If pin level low, stop ADC
412     else
413     {
414
415         // Stop Timer0 A0 ADC
416         tc_stop(TC0,TC_CHANNEL0);
417
418         // Disable sync interrupt and change interrupt edge
419         pio_disable_interrupt(PIOB,PIO_PB26);
420         pio_handler_set(PIOB,ID_PIOB,PIO_PB26,(PIO_PULLUP\
421                         | PIO_IT_RISE_EDGE),Int_Handler_Sync);
422
423         // Transmit remaining echo data
424         ethernet_transmit_echo(&null_value,SEND_MANUAL);
425
426         // Set sync to output and low
427         ioport_set_pin_dir(PIO_PB26_IDX,IOPORT_DIR_OUTPUT);
428         ioport_set_pin_level(PIO_PB26_IDX,IOPORT_PIN_LEVEL_LOW);
429
```

```
430         // Set SPI baudrate to receive data
431         spi_set_baudrate_div(SPI_MASTER_BASE,SPI_CHIP_SEL,\
432                              (sysclk_get_peripheral_hz() /\
433                              SPI_CLOCK_RECEIVE / MEGA_HZ));
434
435         // Response for status end measurement
436         data_transmit[0] = 0xF0;
437
438         // Transmit status response
439         ethernet_transmit_data(&data_transmit[0],\
440                                TRANSMIT_STATUS_SOCKET,1);
441
442         // Enable interrupt ethernet shield
443         pio_enable_interrupt(PIOB,PIO_PB25);
444     }
445 }
446
447 /*************************************************************************/
448 /*                        Function Implementation                        */
449 /*************************************************************************/
450 // Initialize microcontrollerboard
451 static void system_init(void)
452 {
453     sysclk_init();                          // Init system clock
454     board_init();                           // Init board
455     ioport_init();                          // Init ports
456     spi_master_init();                      // Init SPI
457     ethernet_init();                        // Init ethernet shield
458     twi_init();                             // Init I2C
459     bmp280_sensor_init();                   // Init BMP280 Sensor
460     dacc_init();                            // Init DACC
461     adc_init_config();                      // Init ADC
462     timer_init();                           // Init Timer
463     nvicc_init();                           // Init NVICC
464 }
465
466 // Init SPI master
467 static void spi_master_init(void)
468 {
469     spi_enable_clock(SPI_MASTER_BASE);
470     spi_disable(SPI_MASTER_BASE);
471     spi_reset(SPI_MASTER_BASE);
472     spi_set_lastxfer(SPI_MASTER_BASE);
473     spi_set_master_mode(SPI_MASTER_BASE);
```

```
474     spi_disable_mode_fault_detect(SPI_MASTER_BASE);
475     spi_set_peripheral_chip_select_value(SPI_MASTER_BASE,SPI_CHIP_PCS);
476     // SPI mode 2
477     spi_set_clock_polarity(SPI_MASTER_BASE,SPI_CHIP_SEL,\
478                            SPI_CLK_POLARITY);
479     spi_set_clock_phase(SPI_MASTER_BASE,SPI_CHIP_SEL,SPI_CLK_PHASE);
480     spi_set_bits_per_transfer(SPI_MASTER_BASE,SPI_CHIP_SEL,\
481                               SPI_CSR_BITS_8_BIT);
482     // Safe transfer at 21 MHz receive and 28 MHz transmit
483     spi_set_baudrate_div(SPI_MASTER_BASE,SPI_CHIP_SEL,\
484                          (sysclk_get_peripheral_hz() / SPI_CLOCK_RECEIVE\
485                          / MEGA_HZ));
486     // Disable delay before and between transfers
487     spi_set_transfer_delay(SPI_MASTER_BASE,SPI_CHIP_SEL,SPI_DLYBS,\
488                            SPI_DLYBCT);
489     spi_enable(SPI_MASTER_BASE);
490 }
491
492 // Transmit data via SPI
493 static void spi_transmit(uint32_t addr,uint8_t *data,uint8_t data_len)
494 {
495     static uint8_t frame[MAX_DATA_FRAME_SIZE];
496
497     // Ethernet shield address and control
498     frame[0] = (addr & BYTE_2_MASK) >> BYTE_2_SHIFT;
499     frame[1] = (addr & BYTE_1_MASK) >> BYTE_1_SHIFT;
500     frame[2] = (addr & BYTE_0_MASK) >> BYTE_0_SHIFT;
501     // Raw data
502     for (uint8_t i = 0; i < data_len; i++)
503     {
504         frame[i+3] = data[i];
505     }
506
507     // Transmit data
508     for (uint8_t i = 0; i < data_len + 3; i++)
509     {
510         // If TX register not empty, wait
511         while (!(SPI_MASTER_BASE->SPI_SR & SPI_SR_TDRE));
512         SPI_MASTER_BASE->SPI_TDR = SPI_TDR_TD(frame[i]);
513     }
514 }
515
516 // Receive data via SPI
517 static void spi_receive(uint32_t addr,uint8_t *data,uint8_t data_len)
```

```
518 {
519     static uint8_t frame[MAX_DATA_FRAME_SIZE];
520     uint8_t frame_len = 3 + data_len;
521 
522     // Ethernet shield address and control
523     frame[0] = (addr & BYTE_2_MASK) >> BYTE_2_SHIFT;
524     frame[1] = (addr & BYTE_1_MASK) >> BYTE_1_SHIFT;
525     frame[2] = (addr & BYTE_0_MASK) >> BYTE_0_SHIFT;
526 
527     // Receive data
528     for (uint8_t i = 0; i < frame_len; i++)
529     {
530         while (!(SPI_MASTER_BASE->SPI_SR & SPI_SR_TDRE));
531         SPI_MASTER_BASE->SPI_TDR = SPI_TDR_TD(frame[i]);
532         frame[i] = (uint8_t)(SPI_MASTER_BASE->SPI_RDR & SPI_RDR_RD_Msk);
533     }
534 
535     // Delay of 3 bits
536     for (uint8_t i = 0; i < data_len; i++)
537     {
538         data[i] = frame[i+3];
539     }
540 }
541 
542 // Init ethernet shield
543 static void ethernet_init(void)
544 {
545     uint8_t phy_reset[1] = {0xFF & PHYCFGR_RST};
546     uint8_t phy_config[1] = {~PHYCFGR_RST | PHYCFGR_OPMD |\
547                              PHYCFGR_OPMDC_100F};
548     uint8_t phy_check[3] = {};
549     uint8_t mode_reg[1] = {MR_RST | MR_FARP | MR_WOL | MR_PB};
550     uint8_t interrupt_mask[1] = {0};
551     uint8_t socket_int_mask[1] = {0x1};
552     uint8_t socket_buf_rx_size[1] = {BUFFER_RX_SIZE_KB};
553     uint8_t socket_ready[3] = {Sn_CR_SEND};
554 
555     // Set pins as input, not to disturb SPI
556     ioport_enable_pin(PIO_PB27_IDX);
557     ioport_set_pin_dir(PIO_PB27_IDX,IOPORT_DIR_INPUT);
558     ioport_enable_pin(PIO_PD8_IDX);
559     ioport_set_pin_dir(PIO_PD8_IDX,IOPORT_DIR_INPUT);
560     ioport_enable_pin(PIO_PD7_IDX);
561     ioport_set_pin_dir(PIO_PD7_IDX,IOPORT_DIR_INPUT);
```

```
562
563     // Reset and configure PHY register
564     spi_transmit((PHYCFGR | _W5500_SPI_WRITE_),&phy_reset[0],1);
565     spi_transmit((PHYCFGR | _W5500_SPI_WRITE_),&phy_config[0],1);
566     do
567     {
568         // Wait if link down
569         spi_receive(PHYCFGR,&phy_check[0],3);
570     } while (!(phy_check[2] & PHYCFGR_LNK_ON));
571
572     // Configure mode register and interrupts
573     spi_transmit((MR | _W5500_SPI_WRITE_),&mode_reg[0],1);
574     spi_transmit((_IMR_ | _W5500_SPI_WRITE_),&interrupt_mask[0],1);
575     spi_transmit((SIMR | _W5500_SPI_WRITE_),&socket_int_mask[0],1);
576
577     // Configure sensor ethernet address
578     spi_transmit((SHAR | _W5500_SPI_WRITE_),&Sensor_Info.mac[0],6);
579     spi_transmit((GAR | _W5500_SPI_WRITE_),&Sensor_Info.gw[0],4);
580     spi_transmit((SUBR | _W5500_SPI_WRITE_),&Sensor_Info.sn[0],4);
581     spi_transmit((SIPR | _W5500_SPI_WRITE_),&Sensor_Info.ip[0],4);
582
583     // Init sockets
584     ethernet_init_socket(TRANSMIT_ECHO_SOCKET,BUFFER_ECHO_SIZE_KB);
585     ethernet_init_socket(TRANSMIT_TEMP_PRESS_SOCKET,\
586                          BUFFER_TEMP_PRESS_KB);
587     ethernet_init_socket(TRANSMIT_STATUS_SOCKET,BUFFER_STATUS_KB);
588
589     // Set RX buffer size
590     spi_transmit((Sn_RXBUF_SIZE(RECEIVE_SOCKET) | _W5500_SPI_WRITE_),\
591                  &socket_buf_rx_size[0],1);
592
593     // Transmit NULL data
594     spi_transmit((Sn_CR(TRANSMIT_ECHO_SOCKET) | _W5500_SPI_WRITE_),\
595                  &socket_ready[0],2);
596     do
597     {
598         // Wait if data not transmit
599         spi_receive((Sn_IR(TRANSMIT_ECHO_SOCKET)),&socket_ready[0],3);
600     } while (!(socket_ready[2] & Sn_IR_SENDOK));
601 }
602
603 // Init ethernet shield sockets
604 static void ethernet_init_socket(uint8_t socket,uint8_t buffer_size)
605 {
```

```
606     uint8_t socket_mode[1] = {Sn_MR_UDP};
607     uint8_t socket_buf_size[1];
608     uint8_t socket_int_en[1] = {Sn_IR_RECV};
609     uint8_t socket_open[1] = {Sn_CR_OPEN};
610     uint8_t socket_check[3] = {};
611     uint8_t port[2];
612 
613     socket_buf_size[0] = buffer_size;
614     port[0] = port_base[0];
615     port[1] = port_base[1] + socket + 1;
616 
617     // Configure socket registers
618     spi_transmit((Sn_MR(socket) | _W5500_SPI_WRITE_),&socket_mode[0],1);
619     spi_transmit((Sn_PORT(socket) | _W5500_SPI_WRITE_),&port[0],2);
620     spi_transmit((Sn_DPORT(socket) | _W5500_SPI_WRITE_),&port[0],2);
621     spi_transmit((Sn_DIPR(socket) | _W5500_SPI_WRITE_) ,\
622                 &destination_ip[0],4);
623     spi_transmit((Sn_TXBUF_SIZE(socket)| _W5500_SPI_WRITE_) ,\
624                 &socket_buf_size[0],1);
625     spi_transmit((Sn_IMR(socket) | _W5500_SPI_WRITE_) ,\
626                 &socket_int_en[0],1);
627     spi_transmit((Sn_CR(socket) | _W5500_SPI_WRITE_),&socket_open[0],1);
628     do
629     {
630         // Wait if socket is not in UDP mode
631         spi_receive(Sn_SR(socket),&socket_check[0],3);
632     } while (!(socket_check[2] & SOCK_UDP));
633 }
634 
635 // Transmit echo data via ethernet
636 static void ethernet_transmit_echo(uint16_t *data,uint8_t send_manuell)
637 {
638     static uint8_t socket_send[1] = {Sn_CR_SEND};
639     static uint8_t socket_tx_wr_ptr[2] = {};
640     static uint16_t ptr = 0x0000;
641     static uint16_t transmit_conter = 0x00;
642     static uint8_t data_frame[MAX_DATA_FRAME_SIZE];
643 
644     // Ethernet shield address and control
645     uint32_t addr = (WIZCHIP_TXBUF_BLOCK(TRANSMIT_ECHO_SOCKET) << 3)\
646                     | _W5500_SPI_WRITE_ | ((uint32_t)ptr << 8);
647 
648     // Built data frame
649     data_frame[0] = (*data & BYTE_1_MASK) >> BYTE_1_SHIFT;
```

```
650     data_frame[1] = (*data & BYTE_0_MASK) >> BYTE_0_SHIFT;
651
652     // If send manual, transmit echo data in TX register
653     if (send_manuell == SEND_MANUAL)
654     {
655         // Built TX write pointer frame
656         socket_tx_wr_ptr[0] = (ptr & BYTE_1_MASK) >> BYTE_1_SHIFT;
657         socket_tx_wr_ptr[1] = (ptr & BYTE_0_MASK) >> BYTE_0_SHIFT;
658
659         // Set TX write pointer
660         spi_transmit((Sn_TX_WR(TRANSMIT_ECHO_SOCKET)\
661                     | _W5500_SPI_WRITE_),&socket_tx_wr_ptr[0],2);
662
663         // Transmit echo data package
664         spi_transmit((Sn_CR(TRANSMIT_ECHO_SOCKET)\
665                     | _W5500_SPI_WRITE_),&socket_send[0],1);
666
667         // Set transmit counter to null
668         transmit_conter = 0x00;
669     }
670     // Transmit echo data in automatic mode
671     else
672     {
673         // Transmit echo data to TX register
674         spi_transmit(addr,&data_frame[0],2);
675
676         // Increment transmit counter and pointer
677         transmit_conter += 0x2;
678         ptr += 0x2;
679
680         // If counter == max payload, transmit echo data package
681         if (transmit_conter >= MAX_ETHERNET_PAYLOAD)
682         {
683             // Transmit echo data package
684             spi_transmit((Sn_CR(TRANSMIT_ECHO_SOCKET)\
685                         | _W5500_SPI_WRITE_),&socket_send[0],1);
686
687             // Set transmit counter to null
688             transmit_conter = 0x00;
689         }
690         // If counter == max payload minus 2, set TX write pointer
691         else if (transmit_conter >= (MAX_ETHERNET_PAYLOAD - 2))
692         {
693             // Built TX write pointer frame
```

```
694             socket_tx_wr_ptr[0] = ((ptr + 0x2) & BYTE_1_MASK)\
695                                   >> BYTE_1_SHIFT;
696             socket_tx_wr_ptr[1] = ((ptr + 0x2) & BYTE_0_MASK)\
697                                   >> BYTE_0_SHIFT;
698
699             // Set TX write pointer
700             spi_transmit((Sn_TX_WR(TRANSMIT_ECHO_SOCKET)\
701                         | _W5500_SPI_WRITE_),&socket_tx_wr_ptr[0],2);
702         }
703     }
704 }
705
706 // Transmit data via ethernet
707 static void ethernet_transmit_data(uint32_t *data,uint8_t socket,\
708                                    uint8_t data_len)
709 {
710     uint8_t socket_send[1] = {Sn_CR_SEND};
711     uint8_t socket_tx_wr_ptr[2] = {};
712     uint16_t ptr;
713     uint8_t data_frame[MAX_DATA_FRAME_SIZE];
714     uint32_t addr;
715
716     // Get TX write pointer
717     spi_receive(Sn_TX_WR(socket),&socket_tx_wr_ptr[0],4);
718
719     ptr = ((socket_tx_wr_ptr[2] << BYTE_1_SHIFT) | socket_tx_wr_ptr[3]);
720
721     // Transmit data to TX register
722     for (uint8_t i = 0; i < data_len; i++)
723     {
724         // Ethernet shield address and control
725         addr = (WIZCHIP_TXBUF_BLOCK(socket) << 3) | _W5500_SPI_WRITE_\
726                 | ((uint32_t)ptr << 8);
727
728         // Built data frame
729         data_frame[0] = (data[i] & BYTE_3_MASK) >> BYTE_3_SHIFT;
730         data_frame[1] = (data[i] & BYTE_2_MASK) >> BYTE_2_SHIFT;
731         data_frame[2] = (data[i] & BYTE_1_MASK) >> BYTE_1_SHIFT;
732         data_frame[3] = (data[i] & BYTE_0_MASK) >> BYTE_0_SHIFT;
733
734         // Transmit data to TX register
735         spi_transmit(addr,&data_frame[0],4);
736
737         // Increment pointer
```

```
738         ptr += 0x4;
739     }
740 
741     // Built TX write pointer frame
742     socket_tx_wr_ptr[0] = (ptr & BYTE_1_MASK) >> BYTE_1_SHIFT;
743     socket_tx_wr_ptr[1] = (ptr & BYTE_0_MASK) >> BYTE_0_SHIFT;
744 
745     // Set TX write pointer and transmit data
746     spi_transmit((Sn_TX_WR(socket) | _W5500_SPI_WRITE_),\
747                 &socket_tx_wr_ptr[0],2);
748     spi_transmit((Sn_CR(socket) | _W5500_SPI_WRITE_),&socket_send[0],1);
749 }
750 
751 // Receive data via ethernet
752 static void ethernet_receive_data(uint32_t *data)
753 {
754     static uint8_t set_recv[1] = {Sn_CR_RECV};
755     static uint8_t received_data_size[4] = {};
756     static uint8_t rx_read_ptr[4] = {};
757     static uint16_t data_size;
758     static uint16_t read_ptr = 0x0000;
759     static uint8_t data_frame[MAX_DATA_FRAME_SIZE];
760 
761     // Ethernet shield address and control
762     uint32_t addr = (WIZCHIP_RXBUF_BLOCK(RECEIVE_SOCKET) << 3)\
763                   | ((uint32_t)read_ptr << BYTE_1_SHIFT);
764 
765     // Get received data size
766     spi_receive(Sn_RX_RSR(RECEIVE_SOCKET),&received_data_size[0],4);
767 
768     // Calculate data size and RX read pointer
769     data_size = (received_data_size[2] << BYTE_1_SHIFT)\
770                 | received_data_size[3];
771     read_ptr += data_size;
772 
773     // Built RX read pointer
774     rx_read_ptr[0] = (read_ptr & BYTE_1_MASK) >> BYTE_1_SHIFT;
775     rx_read_ptr[1] = (read_ptr & BYTE_0_MASK) >> BYTE_0_SHIFT;
776 
777     // Receive data from RX register
778     spi_receive(addr,&data_frame[0],(data_size+2));
779 
780     // Built data frame
781     data[0] = 0;
```

```
782     data[0] |= (uint32_t)(data_frame[data_size-3] << BYTE_3_SHIFT);
783     data[0] |= (uint32_t)(data_frame[data_size-2] << BYTE_2_SHIFT);
784     data[0] |= (uint32_t)(data_frame[data_size-1] << BYTE_1_SHIFT);
785     data[0] |= (uint32_t)(data_frame[data_size] << BYTE_0_SHIFT);
786     data[1] = 0;
787     data[1] |= (uint32_t)(data_frame[data_size+1] << BYTE_0_SHIFT);
788
789     // Set RX read pointer
790     spi_transmit((Sn_RX_RD(RECEIVE_SOCKET) | _W5500_SPI_WRITE_),\
791                 &rx_read_ptr[0],2);
792     spi_transmit((Sn_CR(RECEIVE_SOCKET) | _W5500_SPI_WRITE_),\
793                 &set_recv[0],1);
794 }
795
796 // Init I2C
797 static void twi_init(void)
798 {
799     twi_options_t opt;
800
801     // Enable peripheral clock for TWI1
802     pmc_enable_periph_clk(ID_TWI1);
803
804     // Define I2C clock
805     opt.master_clk = sysclk_get_peripheral_hz();
806     opt.speed = TWI_CLK;
807
808     // Set I2C clock
809     twi_master_init(TWI1,&opt);
810 }
811
812 // Init BMP280 Sensor
813 static void bmp280_sensor_init(void)
814 {
815     twi_packet_t reset;
816     twi_packet_t status;
817
818     uint8_t bmp280_reset[1] = {BMP280_SOFT_RESET_CODE};
819     uint8_t bmp280_get_status[1];
820
821     // Define soft reset
822     reset.chip = BMP280_I2C_ADDRESS2;
823     reset.addr[0] = BMP280_RST_REG;
824     reset.addr_length = 1;
825     reset.buffer = &bmp280_reset[0];
```

```
826     reset.length = 1;
827
828     // Define status register
829     status.chip = BMP280_I2C_ADDRESS2;
830     status.addr[0] = BMP280_STAT_REG;
831     status.addr_length = 1;
832     status.buffer = &bmp280_get_status[0];
833     status.length = 1;
834
835     // Reset BMP280
836     twi_master_write(TWI1,&reset);
837
838     // Wait if BMP280 is not reset
839     do
840     {
841         twi_master_read(TWI1,&status);
842     } while (bmp280_get_status[0] & BMP280_STATUS_REG_IM_UPDATE__MSK);
843 }
844
845 // Get calibration data
846 static void bmp280_get_calibration(uint8_t *bmp280_calib)
847 {
848     twi_packet_t read_calib;
849
850     // Define calibration register
851     read_calib.chip = BMP280_I2C_ADDRESS2;
852     read_calib.addr[0] = BMP280_TEMPERATURE_CALIB_DIG_T1_LSB_REG;
853     read_calib.addr_length = 1;
854     read_calib.buffer = &bmp280_calib[0];
855     read_calib.length = BMP280_CALIB_DATA_SIZE;
856
857     // Get calibration data via I2C
858     twi_master_read(TWI1,&read_calib);
859 }
860
861 // Start measuring and transmit data
862 static void bmp280_measurement(void)
863 {
864     uint8_t bmp280_calib[BMP280_CALIB_DATA_SIZE];
865     uint32_t transmit_data[8];
866
867     // Get calibration data
868     bmp280_get_calibration(&bmp280_calib[0]);
869
```

```
870     twi_packet_t ctrl_meas;
871     twi_packet_t measurement;
872     twi_packet_t status;
873
874     // Define temperature oversampling
875     uint8_t temperature_oversampling =\
876             (BMP280_ULTRAHIGHRESOLUTION_OVERSAMP_TEMPERATURE \
877             << BMP280_CTRL_MEAS_REG_OVERSAMP_TEMPERATURE__POS) \
878             & BMP280_CTRL_MEAS_REG_OVERSAMP_TEMPERATURE__MSK;
879     // Define pressure oversampling
880     uint8_t pressure_oversampling =\
881             (BMP280_ULTRAHIGHRESOLUTION_OVERSAMP_PRESSURE \
882             << BMP280_CTRL_MEAS_REG_OVERSAMP_PRESSURE__POS) \
883             & BMP280_CTRL_MEAS_REG_OVERSAMP_PRESSURE__MSK;
884
885     uint8_t bmp280_meas[6];
886     uint8_t bmp280_ctrl_meas[1] = {temperature_oversampling\
887                                    | pressure_oversampling\
888                                    | BMP280_FORCED_MODE};
889     uint8_t bmp280_get_status[1];
890
891     // Define control measurement register
892     ctrl_meas.chip = BMP280_I2C_ADDRESS2;
893     ctrl_meas.addr[0] = BMP280_CTRL_MEAS_REG;
894     ctrl_meas.addr_length = 1;
895     ctrl_meas.buffer = &bmp280_ctrl_meas[0];
896     ctrl_meas.length = 1;
897
898     // Define data start address
899     measurement.chip = BMP280_I2C_ADDRESS2;
900     measurement.addr[0] = BMP280_PRESSURE_MSB_REG;
901     measurement.addr_length = 1;
902     measurement.buffer = &bmp280_meas[0];
903     measurement.length = 6;
904
905     // Define status register
906     status.chip = BMP280_I2C_ADDRESS2;
907     status.addr[0] = BMP280_STAT_REG;
908     status.addr_length = 1;
909     status.buffer = &bmp280_get_status[0];
910     status.length = 1;
911
912     // Start measurement
913     twi_master_write(TWI1, &ctrl_meas);
```

```
914
915     // Wait if measurement not started
916     do
917     {
918         twi_master_read(TWI1,&status);
919     } while (!(bmp280_get_status[0] & BMP280_STATUS_REG_MEASURING__MSK));
920
921     // Wait if measurement not finished
922     do
923     {
924         twi_master_read(TWI1,&status);
925     } while (bmp280_get_status[0] & BMP280_STATUS_REG_MEASURING__MSK);
926
927     // Get temperature and pressure
928     twi_master_read(TWI1,&measurement);
929
930     // Pressure
931     transmit_data[0] = (bmp280_meas[0] << 12) | (bmp280_meas[1] << 4)\
932                        | ((bmp280_meas[2] & 0xF0) >> 4);
933     // Temperature
934     transmit_data[1] = (bmp280_meas[3] << 12) | (bmp280_meas[4] << 4)\
935                        | ((bmp280_meas[5] & 0xF0) >> 4);
936     // Calibration data
937     transmit_data[2] = (bmp280_calib[0] << BYTE_3_SHIFT)\
938                        | (bmp280_calib[1] << BYTE_2_SHIFT)\
939                        | (bmp280_calib[2] << BYTE_1_SHIFT)\
940                        | (bmp280_calib[3] << BYTE_0_SHIFT);
941     transmit_data[3] = (bmp280_calib[4] << BYTE_3_SHIFT)\
942                        | (bmp280_calib[5] << BYTE_2_SHIFT)\
943                        | (bmp280_calib[6] << BYTE_1_SHIFT)\
944                        | (bmp280_calib[7] << BYTE_0_SHIFT);
945     transmit_data[4] = (bmp280_calib[8] << BYTE_3_SHIFT)\
946                        | (bmp280_calib[9] << BYTE_2_SHIFT)\
947                        | (bmp280_calib[10] << BYTE_1_SHIFT)\
948                        | (bmp280_calib[11] << BYTE_0_SHIFT);
949     transmit_data[5] = (bmp280_calib[12] << BYTE_3_SHIFT)\
950                        | (bmp280_calib[13] << BYTE_2_SHIFT)\
951                        | (bmp280_calib[14] << BYTE_1_SHIFT)\
952                        | (bmp280_calib[15] << BYTE_0_SHIFT);
953     transmit_data[6] = (bmp280_calib[16] << BYTE_3_SHIFT)\
954                        | (bmp280_calib[17] << BYTE_2_SHIFT)\
955                        | (bmp280_calib[18] << BYTE_1_SHIFT)\
956                        | (bmp280_calib[19] << BYTE_0_SHIFT);
957     transmit_data[7] = (bmp280_calib[20] << BYTE_3_SHIFT)\
```

```
958                             | (bmp280_calib[21] << BYTE_2_SHIFT)\
959                             | (bmp280_calib[22] << BYTE_1_SHIFT)\
960                             | (bmp280_calib[23] << BYTE_0_SHIFT);
961
962     // Transmit data via ethernet
963     ethernet_transmit_data(&transmit_data[0],TRANSMIT_TEMP_PRESS_SOCKET\
964                                 ,8);
965 }
966
967 // Init DACC
968 static void dacc_init(void)
969 {
970     sysclk_enable_peripheral_clock(ID_DACC);
971     dacc_reset(DACC);
972     // Half word mode
973     dacc_set_transfer_mode(DACC,0);
974     // Set channel 1
975     dacc_set_channel_selection(DACC,DACC_CHANNEL);
976     dacc_set_analog_control(DACC,DACC_ANALOG_CONTROL);
977 }
978
979 // Init ADC
980 static void adc_init_config(void)
981 {
982     pmc_enable_periph_clk(ID_ADC);
983     adc_init(ADC, sysclk_get_cpu_hz(),ADC_CLOCK,ADC_STARTUP_TIME_4);
984     adc_configure_timing(ADC,1,ADC_SETTLING_TIME_3,1);
985     adc_enable_tag(ADC);
986     adc_stop_sequencer(ADC);
987     // ADC channel 10 == Arduino Due A8
988     adc_enable_channel(ADC,ADC_CHANNEL_10);
989     adc_disable_anch(ADC);
990     adc_set_channel_input_gain(ADC,ADC_CHANNEL_10,ADC_GAINVALUE_0);
991     adc_disable_channel_input_offset(ADC,ADC_CHANNEL_10);
992     adc_enable_interrupt(ADC,ADC_IER_DRDY);
993     // Enable interrupt
994     NVIC_SetPriority(ADC_IRQn,INT_PRIOR_ADC);
995     NVIC_EnableIRQ(ADC_IRQn);
996 }
997
998 static void timer_init(void)
999 {
1000     // Timer0 A0 for ADC
1001     pmc_enable_periph_clk(ID_TC0);
```

```
1002
1003        // Calculate ADC sample time
1004        uint32_t timer_value_adc = sysclk_get_cpu_hz() /\
1005                                   SAMPLE_RATE_ADC_KHZ / KILO_HZ /\
1006                                   TIMER_MCK_2;
1007
1008        // Init timer for ADC
1009        pio_configure_pin(PIO_PB25_IDX,(PIO_INPUT | PIO_DEFAULT));
1010        tc_init(TC0, TC_CHANNEL0,0 | TC_CMR_CPCTRG | TC_CMR_WAVE\
1011                | TC_CMR_ACPA_CLEAR | TC_CMR_ACPC_SET\
1012                | TC_CMR_TCCLKS_TIMER_CLOCK1);
1013        // Set sample time
1014        TC0->TC_CHANNEL[TC_CHANNEL0].TC_RA = timer_value_adc/2;
1015        TC0->TC_CHANNEL[TC_CHANNEL0].TC_RC = timer_value_adc;
1016        adc_configure_trigger(ADC,ADC_TRIG_TIO_CH_0,0);
1017
1018        // Timer0 A1 for master wait time
1019        ioport_set_pin_mode(PIO_PA2_IDX,IOPORT_MODE_MUX_A);
1020        ioport_disable_pin(PIO_PA2_IDX);
1021
1022        pmc_enable_periph_clk(ID_TC1);
1023
1024        // Calculate master wait time
1025        uint32_t timer_value_burst = sysclk_get_cpu_hz() / DIV_MS\
1026                                     / TIMER_MCK_2 * MASTER_WAIT;
1027
1028        // Init timer for master wait
1029        tc_init(TC0, TC_CHANNEL1,0 | TC_CMR_WAVE | TC_CMR_CPCTRG\
1030                | TC_CMR_ACPA_CLEAR | TC_CMR_ACPC_SET\
1031                | TC_CMR_TCCLKS_TIMER_CLOCK1);
1032        // Set master wait time
1033        TC0->TC_CHANNEL[TC_CHANNEL1].TC_RA = timer_value_burst/2;
1034        TC0->TC_CHANNEL[TC_CHANNEL1].TC_RC = timer_value_burst;
1035        // Enable interrupt
1036        NVIC_DisableIRQ(TC1_IRQn);
1037        NVIC_ClearPendingIRQ(TC1_IRQn);
1038        NVIC_SetPriority(TC1_IRQn,INT_PRIOR_TC);
1039        NVIC_EnableIRQ(TC1_IRQn);
1040
1041        // Timer0 A2 for master burst time
1042        ioport_set_pin_mode(PIO_PA5_IDX,IOPORT_MODE_MUX_A);
1043        ioport_disable_pin(PIO_PA5_IDX);
1044
1045        pmc_enable_periph_clk(ID_TC2);
```

```
1046
1047        // Init timer for burst time
1048        tc_init(TC0, TC_CHANNEL2,0 | TC_CMR_WAVE | TC_CMR_CPCTRG\
1049                | TC_CMR_ACPA_CLEAR | TC_CMR_ACPC_SET\
1050                | TC_CMR_TCCLKS_TIMER_CLOCK1);
1051        // Enable interrupt
1052        NVIC_DisableIRQ(TC2_IRQn);
1053        NVIC_ClearPendingIRQ(TC2_IRQn);
1054        NVIC_SetPriority(TC2_IRQn,INT_PRIOR_TC);
1055        NVIC_EnableIRQ(TC2_IRQn);
1056
1057        // Timer1 A3 for master echo time
1058        ioport_set_pin_mode(PIO_PB0_IDX, IOPORT_MODE_MUX_B);
1059        ioport_disable_pin(PIO_PB0_IDX);
1060
1061        pmc_enable_periph_clk(ID_TC3);
1062
1063        // Init timer for echo time
1064        tc_init(TC1, TC_CHANNEL3,0 | TC_CMR_WAVE | TC_CMR_CPCTRG\
1065                | TC_CMR_ACPA_CLEAR | TC_CMR_ACPC_SET\
1066                | TC_CMR_TCCLKS_TIMER_CLOCK1);
1067        // Enable interrupt
1068        NVIC_DisableIRQ(TC3_IRQn);
1069        NVIC_ClearPendingIRQ(TC3_IRQn);
1070        NVIC_SetPriority(TC3_IRQn,INT_PRIOR_TC);
1071        NVIC_EnableIRQ(TC3_IRQn);
1072    }
1073
1074    // Init NVICC
1075    static void nvicc_init(void)
1076    {
1077        WDT->WDT_MR = WDT_MR_WDDIS;
1078
1079        pmc_enable_periph_clk(ID_PIOB);
1080
1081        // Set ethernet handler
1082        pio_set_debounce_filter(PIOB,PIO_PB25,10);
1083        pio_handler_set(PIOB,ID_PIOB,PIO_PB25,(PIO_PULLUP\
1084                        | PIO_IT_FALL_EDGE),Int_Handler_Ethernet);
1085
1086        // Set sync handler
1087        pio_set_debounce_filter(PIOB,PIO_PB26,10);
1088        pio_handler_set(PIOB,ID_PIOB,PIO_PB26,(PIO_PULLUP\
1089                        | PIO_IT_RISE_EDGE),Int_Handler_Sync);
```

```
1090
1091        // Set PB26 as output
1092        ioport_enable_pin(PIO_PB26_IDX);
1093        ioport_set_pin_dir(PIO_PB26_IDX,IOPORT_DIR_OUTPUT);
1094        ioport_set_pin_level(PIO_PB26_IDX,IOPORT_PIN_LEVEL_LOW);
1095
1096        // Disable interrupts
1097        pio_disable_interrupt(PIOB,0xFFFFFFFF);
1098        pio_get_interrupt_status(PIOB);
1099
1100        // Enable interrupts
1101        NVIC_DisableIRQ((IRQn_Type)ID_PIOB);
1102        NVIC_ClearPendingIRQ((IRQn_Type)ID_PIOB);
1103        NVIC_SetPriority((IRQn_Type)ID_PIOB,INT_PRIOR_ETHERNET);
1104        NVIC_EnableIRQ((IRQn_Type)ID_PIOB);
1105
1106        // Enable interrupt ethernet shield
1107        pio_enable_interrupt(PIOB,PIO_PB25);
1108    }
```

C.2. generate_lookup_header_file.py

```
1   #*********************************************************************
2   #* University: HAW Hamburg                                           *
3   #* Author: Christopher Rotzlawski                                    *
4   #*                                                                   *
5   #* File: generate_lookup_header_file.py                              *
6   #* Version: 1.0                                                      *
7   #*********************************************************************
8   # Import additional modules
9   import tkinter
10  import os
11  import math
12  import numpy
13  import scipy.signal
14
15  #*********************************************************************
16  #*                          Functions                                *
17  #*********************************************************************
18  # Function to start the program
19  def start_main_window():
20      global main_window
21      main_window = tkinter.Tk()
```

```
22      create_main_window()
23      main_window.mainloop()
24
25  # Function to build the window
26  def create_main_window():
27      global tx_status, in_sample_rate_dac, in_sample_rate_adc
28      global in_sweep_start_frequency, in_sweep_end_frequency
29      global in_sweep_time
30
31      # Window title
32      main_window.winfo_toplevel().title("Generate Header File")
33
34      # Frame for buttons and text fields
35      frame = tkinter.Frame(main_window, width = 300, height = 390)
36      frame.pack()
37
38      # Text fields
39      tx_system = tkinter.Label(frame, text = "System Parameter", font = "
            Arial 15 bold")
40      tx_system.place(x = 60, y = 20, anchor = "nw")
41      tx_sample_rate_dac = tkinter.Label(frame, text = "DAC Sample Rate:",
            font = "Arial 11 bold")
42      tx_sample_rate_dac.place(x = 20, y = 60, anchor = "nw")
43      tx_sample_rate_adc = tkinter.Label(frame, text = "ADC Sample Rate:",
            font = "Arial 11 bold")
44      tx_sample_rate_adc.place(x = 20, y = 100, anchor = "nw")
45      tx_sweep_signal = tkinter.Label(frame, text = "Sweep Signal", font =
            "Arial 15 bold")
46      tx_sweep_signal.place(x = 85, y = 140, anchor = "nw")
47      tx_sweep_start_frequency = tkinter.Label(frame, text = "Start
            Frequency:", font = "Arial 11 bold")
48      tx_sweep_start_frequency.place(x = 20, y = 180, anchor = "nw")
49      tx_sweep_end_frequency = tkinter.Label(frame, text = "End Frequency:"
            ,font = "Arial 11 bold")
50      tx_sweep_end_frequency.place(x = 20, y = 220, anchor = "nw")
51      tx_sweep_time = tkinter.Label(frame, text = "Sweep Time:", font = "
            Arial 11 bold")
52      tx_sweep_time.place(x = 20, y = 260, anchor = "nw")
53      tx_status = tkinter.Label(frame, text = "Status: Ready", font = "
            Arial 11 bold")
54      tx_status.place(x = 160, y = 303, anchor = "nw")
55      tx_note = tkinter.Label(frame, text = "Note: Values must be integer!"
            ,font = "Arial 10 bold")
56      tx_note.place(x = 53, y = 340, anchor = "nw")
```

```
57
58      # Entry value fileds
59      in_sample_rate_dac = tkinter.Entry(frame, width = 5, font = "Arial 11
            bold")
60      in_sample_rate_dac.place(x = 200, y = 60, anchor = "nw")
61      in_sample_rate_adc = tkinter.Entry(frame, width = 5, font = "Arial 11
            bold")
62      in_sample_rate_adc.place(x = 200, y = 100, anchor = "nw")
63      in_sweep_start_frequency = tkinter.Entry(frame, width = 5, font = "
            Arial 11 bold")
64      in_sweep_start_frequency.place(x = 200, y = 180, anchor = "nw")
65      in_sweep_end_frequency = tkinter.Entry(frame, width = 5, font = "
            Arial 11 bold")
66      in_sweep_end_frequency.place(x = 200, y = 220, anchor = "nw")
67      in_sweep_time = tkinter.Entry(frame, width = 5, font = "Arial 11 bold
            ")
68      in_sweep_time.place(x = 200, y = 260, anchor = "nw")
69
70      # Unit fields
71      un_sample_rate_dac = tkinter.Label(frame, text = "kHz", font = "Arial
            11 bold")
72      un_sample_rate_dac.place(x = 250, y = 60, anchor = "nw")
73      un_sample_rate_adc = tkinter.Label(frame, text = "kHz", font = "Arial
            11 bold")
74      un_sample_rate_adc.place(x = 250, y = 100, anchor = "nw")
75      un_sweep_start_frequency = tkinter.Label(frame, text = "kHz", font =
            "Arial 11 bold")
76      un_sweep_start_frequency.place(x = 250, y = 180, anchor = "nw")
77      un_sweep_end_frequency = tkinter.Label(frame, text = "kHz", font = "
            Arial 11 bold")
78      un_sweep_end_frequency.place(x = 250, y = 220, anchor = "nw")
79      un_sweep_time = tkinter.Label(frame, text = "ms", font = "Arial 11
            bold")
80      un_sweep_time.place(x = 250, y = 260, anchor = "nw")
81
82      # Button for generating header file
83      generate_file_button = tkinter.Button(frame, text = "Generate File",
            width = 12, font = "Arial 11 bold", command = generate_file)
84      generate_file_button.place(x = 20, y = 300, anchor = "nw")
85
86  # Function to recording values and function calls
87  def generate_file():
88      global sample_rate_dac, sample_rate_adc, start_frequency
89      global end_frequency, sweep_time, sine_wave, sweep_signal, sweep_size
```

```
90
91      toggle_status()
92      try:
93          # Recording values
94          sample_rate_dac = int(in_sample_rate_dac.get())
95          sample_rate_adc = int(in_sample_rate_adc.get())
96          start_frequency = int(in_sweep_start_frequency.get())
97          end_frequency = int(in_sweep_end_frequency.get())
98          sweep_time = int(in_sweep_time.get())
99
100         # Calculating look-up-tables
101         sine_wave = calculate_sine()
102         sweep_signal,sweep_size = calculate_sweep()
103
104         # Create header file
105         create_header()
106         toggle_status()
107
108     except:
109         # Error at try, e.g. enter a float or other characters
110         tx_status["text"] = "Status: Error"
111
112 # Function to creating header file
113 def create_header():
114     # If file exists, then delete first
115     if os.path.exists("signal_lookup_table.h"):
116         os.remove("signal_lookup_table.h")
117     # Create file and write data
118     file = open("signal_lookup_table.h","w")
119     file.write("
        /*****************************************************************\
        n")
120     file.write("/*        Automatically generated lookup table header
        file by        */\n")
121     file.write("/*                  generate_lookup_header_file.py
                        */\n")
122     file.write("
        /*****************************************************************\
        n\n")
123     file.write("#ifndef SIGNAL_LOOKUP_TABLE_H\n")
124     file.write("#define SIGNAL_LOOKUP_TABLE_H\n\n")
125     file.write("// Sample rate of system\n")
126     file.writelines("#define SAMPLE_RATE_DAC_KHZ " + str(sample_rate_dac)
         + "\n")
```

```
127     file.writelines("#define SAMPLE_RATE_ADC_KHZ " + str(sample_rate_adc)
            + "\n\n")
128     file.write("// Sine wave\n")
129     file.writelines("#define SINE_SAMPLES " + str(sample_rate_dac) + "\n"
            )
130     file.write("const uint16_t sine_data[SINE_SAMPLES] =\n{\n")
131     file.write(sine_wave + "\n")
132     file.write("};\n\n")
133     file.write("// Sweep signal\n")
134     file.writelines("#define SWEEP_SAMPLES " + str(sweep_size) + "\n")
135     file.writelines("#define SWEEP_START_FREQUENCY_KHZ " + str(
            start_frequency) + "\n")
136     file.writelines("#define SWEEP_END_FREQUENCY_KHZ " + str(
            end_frequency) + "\n")
137     file.write("const uint16_t sweep_signal[SWEEP_SAMPLES] =\n{\n")
138     file.write(sweep_signal + "\n")
139     file.write("};\n\n")
140     file.write("#endif")
141     file.close()
142
143 # Function to calculating sine wave steps
144 def calculate_sine():
145     for i in range(0, sample_rate_dac):
146         sine_signal = math.sin(i / sample_rate_dac * 2 * math.pi)
147         # Normalization to 12 bit
148         sine_signal = hex(round((sine_signal + 1) / 2 * 4095))
149         # Seperate first value
150         if i == 0:
151             sine_wave = "    " + str(sine_signal)
152         else:
153             sine_wave = sine_wave + ", "
154             # 10 values per line
155             if i % 10 == 0:
156                 sine_wave = sine_wave + "\n    "
157             sine_wave = sine_wave + str(sine_signal)
158     return sine_wave
159
160 # Function to calculating sweep signal steps
161 def calculate_sweep():
162     # 1000 * 1 / 1000 (kHz / ms)
163     sweep_size = sample_rate_dac * sweep_time
164     time = numpy.linspace(0, sweep_time / 1000, num = sweep_size)
165     sw_numpy = scipy.signal.chirp(time, f0 = start_frequency * 1000, f1 =
            end_frequency * 1000, t1 = sweep_time / 1000, method = "linear")
```

```
166     sw_signal = sw_numpy.tolist()
167     for i in range(0, sweep_size):
168         # Normalization to 12 bit
169         sw_signal[i] = hex(round((sw_signal[i] + 1) / 2 * 4095))
170         # Seperate first value
171         if i == 0:
172             sweep_signal = "    " + str(sw_signal[i])
173         else:
174             sweep_signal = sweep_signal + ", "
175             # 10 values per line
176             if i % 10 == 0:
177                 sweep_signal = sweep_signal + "\n    "
178             sweep_signal = sweep_signal + str(sw_signal[i])
179     return sweep_signal,sweep_size
180 
181 # Function to toggle status
182 def toggle_status():
183     if tx_status["text"] == "Status: Ready":
184         tx_status["text"] = "Status: In Process"
185     elif tx_status["text"] == "Status: In Process":
186         tx_status["text"] = "Status: Done"
187     elif tx_status["text"] == "Status: Done":
188         tx_status["text"] = "Status: In Process"
189     elif tx_status["text"] == "Status: Error":
190         tx_status["text"] = "Status: In Process"
191 
192 # If program running as main, start window
193 if __name__ == "__main__":
194     start_main_window()
```

D. Quellcode der Datenaufnahme

D.1. mainwindow.py

```
#*****************************************************************************
#* University: HAW Hamburg                                                   *
#* Author: Christopher Rotzlawski                                            *
#*                                                                           *
#* File: mainwindow.py                                                       *
#* Version: 1.0                                                              *
#*****************************************************************************
# Import additional modules
import tkinter
import os

# Import project modules
import calculate_pressure_temperature
import ethernet_network
import ethernet_network_setup
import analyze_data

#*****************************************************************************
#*                                  Class                                    *
#*****************************************************************************
# Class to built mainwindow and control development system
class mainwindow():
    # Method to initialize the window
    def __init__(self):
        self.window = tkinter.Tk()
        self.create_window()
        self.window.mainloop()

    # Method to built the window
    def create_window(self):
        self.burst_mode = tkinter.StringVar()
        self.burst_mode.set("Const")
        self.burst_time = tkinter.StringVar()
```

```
34          self.burst_time.set("Time")
35          self.analyze_visual = tkinter.StringVar()
36          self.analyze_visual.set("Visual")
37          self.analyze_mode = tkinter.StringVar()
38          self.analyze_mode.set("Raw")
39
40          # Window title
41          self.window.winfo_toplevel().title("Development system for
                intelligent ultrasonic sensors")
42
43          # Frames
44          frame = tkinter.Frame(self.window, width = 680, height = 560)
45          frame.pack()
46          frame_mode = tkinter.Frame(frame, width = 310, height = 200,
                relief = "sunken", bd = 1)
47          frame_mode.place(x = 20, y = 20, anchor = "nw")
48          frame_environment = tkinter.Frame(frame, width = 310, height =
                130, relief = "sunken", bd = 1)
49          frame_environment.place(x = 350, y = 240, anchor = "nw")
50          frame_network = tkinter.Frame(frame, width = 310, height = 200,
                relief = "sunken", bd = 1)
51          frame_network.place(x = 350, y = 20, anchor = "nw")
52          frame_control = tkinter.Frame(frame, width = 310, height = 130,
                relief = "sunken", bd = 1)
53          frame_control.place(x = 20, y = 240, anchor = "nw")
54          frame_analyze = tkinter.Frame(frame, width = 640, height = 150,
                relief = "sunken", bd = 1)
55          frame_analyze.place(x = 20, y = 390, anchor = "nw")
56
57          # Buttons
58          rbutton_const = tkinter.Radiobutton(frame_mode, text = "Const.
                Frequency", variable = self.burst_mode, value = "Const")
59          rbutton_const.place(x = 10, y = 40, anchor = "nw")
60          rbutton_sweep = tkinter.Radiobutton(frame_mode, text = "Sweep
                Frequency", variable = self.burst_mode, value = "Sweep")
61          rbutton_sweep.place(x = 180, y = 40, anchor = "nw")
62          rbutton_time = tkinter.Radiobutton(frame_mode, text = "Burst Time
                :", variable = self.burst_time, value = "Time")
63          rbutton_time.place(x = 10, y = 90, anchor = "nw")
64          rbutton_cycles = tkinter.Radiobutton(frame_mode, text = "Burst
                Cycles:", variable = self.burst_time, value = "Cycles")
65          rbutton_cycles.place(x = 10, y = 115, anchor = "nw")
66          button_environment = tkinter.Button(frame_environment, text = "
```

```
            Get Values", width = 18, font = "Arial 9", command = self.
            get_environment)
67          button_environment.place(x = 10, y = 90, anchor = "nw")
68          button_connect = tkinter.Button(frame_control, text = "Connect",
            width = 18, font = "Arial 9", command = self.connect_network)
69          button_connect.place(x = 10, y = 50, anchor = "nw")
70          button_disconnect = tkinter.Button(frame_control, text = "
            Disconnect", width = 18, font = "Arial 9", command = self.
            disconnect_network)
71          button_disconnect.place(x = 160, y = 50, anchor = "nw")
72          button_start = tkinter.Button(frame_control, text = "Start",
            width = 18, font = "Arial 9", command = self.start_measurement
            )
73          button_start.place(x = 10, y = 90, anchor = "nw")
74          rbutton_visual = tkinter.Radiobutton(frame_analyze, text = "
            Visualize Data", variable = self.analyze_visual, value = "
            Visual")
75          rbutton_visual.place(x = 10, y = 40, anchor = "nw")
76          rbutton_visual_not = tkinter.Radiobutton(frame_analyze, text = "
            Don't Visualize Data", variable = self.analyze_visual, value =
             "Visual_Not")
77          rbutton_visual_not.place(x = 10, y = 65, anchor = "nw")
78          rbutton_raw = tkinter.Radiobutton(frame_analyze, text = "Raw Data
            ", variable = self.analyze_mode, value = "Raw")
79          rbutton_raw.place(x = 170, y = 40, anchor = "nw")
80          rbutton_bandpass = tkinter.Radiobutton(frame_analyze, text = "
            Bandpass Filtered", variable = self.analyze_mode, value = "BP"
            )
81          rbutton_bandpass.place(x = 290, y = 40, anchor = "nw")
82          rbutton_correlated = tkinter.Radiobutton(frame_analyze, text = "
            Correlated", variable = self.analyze_mode, value = "Corr")
83          rbutton_correlated.place(x = 170, y = 65, anchor = "nw")
84          rbutton_enveloped = tkinter.Radiobutton(frame_analyze, text = "
            Enveloped", variable = self.analyze_mode, value = "Envel")
85          rbutton_enveloped.place(x = 170, y = 90, anchor = "nw")
86          rbutton_fft = tkinter.Radiobutton(frame_analyze, text = "FFT",
            variable = self.analyze_mode, value = "FFT")
87          rbutton_fft.place(x = 460, y = 40, anchor = "nw")
88          rbutton_distance = tkinter.Radiobutton(frame_analyze, text = "
            Distance", variable = self.analyze_mode, value = "Dist")
89          rbutton_distance.place(x = 460, y = 65, anchor = "nw")
90
91          # Text fields
```

```
92          tx_mode = tkinter.Label(frame_mode, text = " Burst Mode", font =
                "Arial 15 bold")
93          tx_mode.place(x = 10, y = 10, anchor = "nw")
94          tx_frequency = tkinter.Label(frame_mode, text = "Frequency:",
                font = "Arial 9")
95          tx_frequency.place(x = 10, y = 65, anchor = "nw")
96          tx_echo_time = tkinter.Label(frame_mode, text = "Echo Time:",
                font = "Arial 9")
97          tx_echo_time.place(x = 10, y = 140, anchor = "nw")
98          tx_amplitude = tkinter.Label(frame_mode, text = "Amplitude:",
                font = "Arial 9")
99          tx_amplitude.place(x = 10, y = 165, anchor = "nw")
100         tx_environment = tkinter.Label(frame_environment, text = "
                Environment Values", font = "Arial 15 bold")
101         tx_environment.place(x = 10, y = 10, anchor = "nw")
102         tx_pressure = tkinter.Label(frame_environment, text = "Pressure:"
                , font = "Arial 9")
103         tx_pressure.place(x = 10, y = 40, anchor = "nw")
104         tx_temperature = tkinter.Label(frame_environment, text = "
                Temperature:", font = "Arial 9")
105         tx_temperature.place(x = 10, y = 65, anchor = "nw")
106         tx_network = tkinter.Label(frame_network, text = "Network
                Information", font = "Arial 15 bold")
107         tx_network.place(x = 10, y = 10, anchor = "nw")
108         tx_sensors = tkinter.Label(frame_network, text = "Sensors:", font
                 = "Arial 9")
109         tx_sensors.place(x = 10, y = 40, anchor = "nw")
110         tx_master_sensor = tkinter.Label(frame_network, text = "Master
                Sensor:", font = "Arial 9")
111         tx_master_sensor.place(x = 130, y = 40, anchor = "nw")
112         tx_ip_base = tkinter.Label(frame_network, text = "IP Base Address
                :", font = "Arial 9")
113         tx_ip_base.place(x = 10, y = 65, anchor = "nw")
114         tx_subnet_mask = tkinter.Label(frame_network, text = "Subnet Mask
                :", font = "Arial 9")
115         tx_subnet_mask.place(x = 10, y = 90, anchor = "nw")
116         tx_port_base = tkinter.Label(frame_network, text = "Port Base:",
                font = "Arial 9")
117         tx_port_base.place(x = 10, y = 115, anchor = "nw")
118         tx_ip_base_value = tkinter.Label(frame_network, text =
                ethernet_network_setup.get_ip_base_str(), font = "Arial 9")
119         tx_ip_base_value.place(x = 120, y = 65, anchor = "nw")
120         tx_subnet_mask_value = tkinter.Label(frame_network, text =
                ethernet_network_setup.get_subnet_str(), font = "Arial 9")
```

```
121         tx_subnet_mask_value.place(x = 120, y = 90, anchor = "nw")
122         tx_port_base_value = tkinter.Label(frame_network, text =
                ethernet_network_setup.get_port_base(), font = "Arial 9")
123         tx_port_base_value.place(x = 120, y = 115, anchor = "nw")
124         tx_connect = tkinter.Label(frame_control, text = "Process Control
                ", font = "Arial 15 bold")
125         tx_connect.place(x = 10, y = 10, anchor = "nw")
126         tx_analyze = tkinter.Label(frame_analyze, text = "Analyze Data",
                font = "Arial 15 bold")
127         tx_analyze.place(x = 10, y = 10, anchor = "nw")
128         tx_bp_lower = tkinter.Label(frame_analyze, text = "Lowcut:", font
                 = "Arial 9")
129         tx_bp_lower.place(x = 310, y = 65, anchor = "nw")
130         tx_bp_upper = tkinter.Label(frame_analyze, text = "Highcut:",
                font = "Arial 9")
131         tx_bp_upper.place(x = 310, y = 90, anchor = "nw")
132         tx_bp_order = tkinter.Label(frame_analyze, text = "Order:", font
                = "Arial 9")
133         tx_bp_order.place(x = 310, y = 115, anchor = "nw")
134         tx_dist_threshold = tkinter.Label(frame_analyze, text = "
                Threshold:", font = "Arial 9")
135         tx_dist_threshold.place(x = 480, y = 90, anchor = "nw")
136
137         # Text fields for values
138         self.v_mode_specify = tkinter.Label(frame_mode, text = " ", font
                = "Arial 9")
139         self.v_mode_specify.place(x = 175, y = 165, anchor = "nw")
140         self.v_network_specify = tkinter.Label(frame_network, text = " ",
                 font = "Arial 9")
141         self.v_network_specify.place(x = 10, y = 155, anchor = "nw")
142         self.v_environment_connect = tkinter.Label(frame_environment,
                text = " ", font = "Arial 9")
143         self.v_environment_connect.place(x = 180, y = 90, anchor = "nw")
144         self.v_pressure = tkinter.Label(frame_environment, text = "-",
                font = "Arial 9")
145         self.v_pressure.place(x = 100, y = 40, anchor = "nw")
146         self.v_temperature = tkinter.Label(frame_environment, text = "-",
                 font = "Arial 9")
147         self.v_temperature.place(x = 100, y = 65, anchor = "nw")
148         self.v_connect = tkinter.Label(frame_control, text = "Not
                Connected", font = "Arial 9")
149         self.v_connect.place(x = 160, y = 90, anchor = "nw")
150         self.v_analyze = tkinter.Label(frame_analyze, text = " ", font =
                "Arial 9")
```

```
151         self.v_analyze.place(x = 450, y = 15, anchor = "nw")
152
153         # Entry fields
154         self.in_frequency = tkinter.Entry(frame_mode, width = 5, font = "
                Arial 9")
155         self.in_frequency.place(x = 110, y = 65, anchor = "nw")
156         self.in_burst_time = tkinter.Entry(frame_mode, width = 5, font =
                "Arial 9")
157         self.in_burst_time.place(x = 110, y = 90, anchor = "nw")
158         self.in_burst_cycles = tkinter.Entry(frame_mode, width = 5, font
                = "Arial 9")
159         self.in_burst_cycles.place(x = 110, y = 115, anchor = "nw")
160         self.in_echo_time = tkinter.Entry(frame_mode, width = 5, font = "
                Arial 9")
161         self.in_echo_time.place(x = 110, y = 140, anchor = "nw")
162         self.in_amplitude = tkinter.Entry(frame_mode, width = 5, font = "
                Arial 9")
163         self.in_amplitude.place(x = 110, y = 165, anchor = "nw")
164         self.in_sensors = tkinter.Entry(frame_network, width = 5, font =
                "Arial 9")
165         self.in_sensors.place(x = 80, y = 40, anchor = "nw")
166         self.in_master_sensor = tkinter.Entry(frame_network, width = 5,
                font = "Arial 9")
167         self.in_master_sensor.place(x = 230, y = 40, anchor = "nw")
168         self.in_bp_lower = tkinter.Entry(frame_analyze, width = 5, font =
                 "Arial 9")
169         self.in_bp_lower.place(x = 365, y = 65, anchor = "nw")
170         self.in_bp_upper = tkinter.Entry(frame_analyze, width = 5, font =
                 "Arial 9")
171         self.in_bp_upper.place(x = 365, y = 90, anchor = "nw")
172         self.in_bp_order = tkinter.Entry(frame_analyze, width = 5, font =
                 "Arial 9")
173         self.in_bp_order.place(x = 365, y = 115, anchor = "nw")
174         self.in_dist_threshold = tkinter.Entry(frame_analyze, width = 5,
                font = "Arial 9")
175         self.in_dist_threshold.place(x = 550, y = 90, anchor = "nw")
176
177         # Unit fields
178         un_frequency = tkinter.Label(frame_mode, text = "kHz", font = "
                Arial 9")
179         un_frequency.place(x = 150, y = 65, anchor = "nw")
180         un_burst_time = tkinter.Label(frame_mode, text = "us", font = "
                Arial 9")
181         un_burst_time.place(x = 150, y = 90, anchor = "nw")
```

```
182         un_echo_time = tkinter.Label(frame_mode, text = "ms", font = "
                Arial 9")
183         un_echo_time.place(x = 150, y = 140, anchor = "nw")
184         un_amplitude = tkinter.Label(frame_mode, text = "%", font = "
                Arial 9")
185         un_amplitude.place(x = 150, y = 165, anchor = "nw")
186         un_pressure = tkinter.Label(frame_environment, text = "kPa", font
                = "Arial 9")
187         un_pressure.place(x = 150, y = 40, anchor = "nw")
188         un_temperature = tkinter.Label(frame_environment, text = "Â°C",
                font = "Arial 9")
189         un_temperature.place(x = 150, y = 65, anchor = "nw")
190         un_bp_lower = tkinter.Label(frame_analyze, text = "kHz", font = "
                Arial 9")
191         un_bp_lower.place(x = 405, y = 65, anchor = "nw")
192         un_bp_upper = tkinter.Label(frame_analyze, text = "kHz", font = "
                Arial 9")
193         un_bp_upper.place(x = 405, y = 90, anchor = "nw")
194
195     # Method to get environment values
196     def get_environment(self):
197         if self.v_connect["text"] != "Not Connected":
198             self.v_environment_connect["text"] = " "
199
200             # Variables for pressure and temperature
201             pressure = 0
202             temperature = 0
203
204             for i in range(0,self.int_sensors):
205                 # Get environmental values
206                 raw_data = self.network.get_environmental_values(i+1)
207
208                 # Calculate Pressure and Temperature
209                 calculate_pressure_temperature.
                        calculate_pressure_temperature(raw_data)
210
211                 # Open file and get pressure and temperature
212                 file = open("pressure_temperature.csv","r")
213                 values = file.read()
214                 file_length = len(values)
215
216                 for i in range(0, file_length):
217                     separate_pos = i
218                     if values[i] == ";":
```

```
219                         break
220
221                 pressure += float(values[0:i]) / 1000.0
222                 temperature += float(values[i+1:file_length])
223
224                 # Close and remove File
225                 file.close()
226                 os.remove("pressure_temperature.csv")
227
228             # Calculate average
229             pressure = round(pressure/float(self.int_sensors),2)
230             temperature = round(temperature/float(self.int_sensors),2)
231
232             # Set Pressure and Temperature
233             self.v_pressure["text"] = str(pressure)
234             self.v_temperature["text"] = str(temperature)
235
236         else:
237             self.v_environment_connect["text"] = "Not Connected"
238
239     # Method to connect network
240     def connect_network(self):
241         if self.v_connect["text"] == "Not Connected":
242             # Get number of sensors
243             try:
244                 self.int_sensors = int(self.in_sensors.get())
245                 error = False
246             except:
247                 self.v_network_specify["text"] = "Specify sensors"
248                 error = True
249
250             # Connect network
251             if error == False:
252                 # Get subnet mask
253                 int_subnet_mask = ethernet_network_setup.get_subnet_int()
254
255                 # Check if sensors in range
256                 if self.int_sensors < (0xFFFFFFFF ^ int_subnet_mask):
257                     self.network = ethernet_network.ethernet_network(self
                            .int_sensors)
258                 else:
259                     self.network.close_sockets()
260                     self.v_network_specify["text"] = "Too many sensors"
261                     error = True
```

```
262
263             # Check sensor connection
264             if error == False:
265                 sensor_not_connected = "Sensors not Connected: "
266
267                 # Check sensor connection
268                 for i in range(1, self.int_sensors+1):
269                     check = self.network.check_sensors(i)
270
271                     if check == False:
272                         sensor_not_connected += (str(i) + " ")
273                         error = True
274
275                 # If error, display not connected sensors
276                 if error == True:
277                     self.network.close_sockets()
278
279                     self.v_network_specify["text"] = "Sensors not found"
280
281                     print(sensor_not_connected)
282
283                 else:
284                     self.v_connect["text"] = "Connected"
285                     self.v_network_specify["text"] = " "
286
287     # Method to disconnect network
288     def disconnect_network(self):
289         if self.v_connect["text"] != "Not Connected":
290             self.network.close_sockets()
291             self.v_connect["text"] = "Not Connected"
292
293     # Method to start measurement
294     def start_measurement(self):
295         # UDP packages
296         udp_package_master = 0 | ethernet_network_setup.
                get_master_prototype()
297         udp_package_slave = ethernet_network_setup.get_slave_prototype()
298
299         if self.v_connect["text"] != "Not Connected":
300             # Get master sensor
301             try:
302                 master = int(self.in_master_sensor.get())
303                 self.v_network_specify["text"] = " "
304                 error = False
```

```
305
306                 # Check if master is in range
307                 if master > self.int_sensors:
308                     self.v_network_specify["text"] = "Master out of range
                        "
309                     error = True
310             except:
311                 self.v_network_specify["text"] = "Specify master"
312                 error = True
313
314             # Get echo time for master UDP package
315             if error == False:
316                 try:
317                     echo_time = int(self.in_echo_time.get())
318                     echo_time_mask = ethernet_network_setup.
                        get_echo_time_prototype()
319
320                     # Check if echo time is in range
321                     if echo_time > echo_time_mask:
322                         self.v_mode_specify["text"] = "Echo time too big"
323                         error = True
324                     else:
325                         udp_package_master = udp_package_master |
                            echo_time
326                 except:
327                     self.v_mode_specify["text"] = "Specify echo time"
328                     error = True
329
330             # Get amplitude for master UDP package
331             if error == False:
332                 try:
333                     amplitude = int(self.in_amplitude.get())
334                     amplitude_mask, amplitude_shift =
                        ethernet_network_setup.get_amplitude_prototype()
335
336                     # Check if amplitude is in range
337                     if (amplitude > 100) or (amplitude < 1):
338                         self.v_mode_specify["text"] = "Amplitude out of
                            range"
339                         error = True
340                     else:
341                         amplitude = int(amplitude_mask / 100 * amplitude)
342                         udp_package_master = udp_package_master | (
                            amplitude << amplitude_shift)
```

```
343                 except:
344                     self.v_mode_specify["text"] = "Specify amplitude"
345                     error = True
346 
347             # Get values for master UDP package in const frequency mode
348             if (error == False) and (self.burst_mode.get() == "Const"):
349                 # Get frequency
350                 try:
351                     frequency = int(self.in_frequency.get())
352                     frequency_mask,freqzuency_shift =
                            ethernet_network_setup.get_frequency_prototype()
353 
354                     # Check if frequency is in range
355                     if frequency > frequency_mask:
356                         self.v_mode_specify["text"] = "Frequency too big"
357                         error = True
358                     else:
359                         udp_package_master = udp_package_master | (
                                frequency << freqzuency_shift)
360                 except:
361                     self.v_mode_specify["text"] = "Specify frequency"
362                     error = True
363 
364                 # Get burst time
365                 if (error == False) and (self.burst_time.get() == "Time")
                        :
366                     try:
367                         burst_time = int(self.in_burst_time.get())
368                         burst_time_mask,burst_time_shift =
                                ethernet_network_setup.
                                get_burst_time_prototype()
369 
370                         # Check if burst time is in range
371                         if (burst_time > burst_time_mask) or (burst_time
                                >= (echo_time * 1000)):
372                             self.v_mode_specify["text"] = "Burst time too
                                    big"
373                             error = True
374                         else:
375                             udp_package_master = udp_package_master | (
                                    burst_time << burst_time_shift)
376                     except:
377                         self.v_mode_specify["text"] = "Specify burst time
                                "
```

```
378                     error = True
379
380                 # Get burst cycles and calculate burst time
381                 elif error == False:
382                     try:
383                         burst_cycles = int(self.in_burst_cycles.get())
384                         burst_time = int(1 / frequency / 1000 * 1000000 *
                                burst_cycles)
385                         burst_time_mask, burst_time_shift =
                                ethernet_network_setup.
                                get_burst_time_prototype()
386
387                         # Check if burst cycles are in range
388                         if burst_time > burst_time_mask:
389                             self.v_mode_specify["text"] = "Burst cycles
                                    too big"
390                             error = True
391                         else:
392                             udp_package_master = udp_package_master | (
                                    burst_time << burst_time_shift)
393                     except:
394                         self.v_mode_specify["text"] = "Specify burst
                                cycles"
395                         error = True
396
397             # Get values for master UDP package in sweep mode
398             elif error == False:
399                 udp_package_master = udp_package_master |
                        ethernet_network_setup.get_sweep_const_prototype()
400
401             # Check analyze mode
402             if (error == False) and (self.analyze_mode.get() == "BP") and
                    (self.analyze_visual.get() == "Visual"):
403                 try:
404                     lowcut_frequency = int(self.in_bp_lower.get())
405                     highcut_frequency = int(self.in_bp_upper.get())
406                     order = int(self.in_bp_order.get())
407                     self.v_analyze["text"] = " "
408                 except:
409                     self.v_analyze["text"] = "Specify bandpass"
410                     error = True
411
412             elif (error == False) and (self.analyze_mode.get() == "Dist")
                    and (self.analyze_visual.get() == "Visual"):
```

```
413                 try:
414                     threshold = int(self.in_dist_threshold.get())
415                     self.v_analyze["text"] = " "
416                 except:
417                     self.v_analyze["text"] = "Specify threshold"
418                     error = True
419 
420             # If no error, start measuring
421             if error == False:
422                 self.v_mode_specify["text"] = " "
423 
424                 # Transmit master UDP package
425                 self.network.transmit_instruction(master,
                        udp_package_master)
426 
427                 # Transmit slave UDP package
428                 for i in range(0,self.int_sensors):
429                     if (i+1) != master:
430                         self.network.transmit_instruction((i+1),
                                udp_package_slave)
431 
432             # Receive echo packages
433             if error == False:
434                 check_measurement = self.network.receive_echo_data(self.
                        int_sensors,echo_time)
435                 if check_measurement == False:
436                     self.v_connect["text"] = "Response is missing"
437                     error = True
438                 else:
439                     self.v_connect["text"] = "Measurement finished"
440 
441             # Analyze echo data
442             if (error == False) and (self.burst_mode.get() == "Const"):
443                 # Built object to analyse data
444                 analyze = analyze_data.analyze_data(self.int_sensors,
                        burst_time,master)
445 
446                 # If visualize echo data
447                 if self.analyze_visual.get() == "Visual":
448                     # Close figures
449                     analyze.close_figures()
450 
451                     # If visualize raw data, plot raw data
452                     if self.analyze_mode.get() == "Raw":
```

```
453                         analyze.raw_echo_data()
454
455                     # If visualize bandpass filtered data
456                     elif self.analyze_mode.get() == "BP":
457                         analyze.bandpass_echo_data(order,lowcut_frequency
                                ,highcut_frequency)
458
459                     # If visualize correlated echo data
460                     elif self.analyze_mode.get() == "Corr":
461                         analyze.correlated_echo_data(frequency,"Visual")
462
463                     # If visualize enveloped echo data
464                     elif self.analyze_mode.get() == "Envel":
465                         analyze.enveloped_echo_data(frequency,"Visual")
466
467                     # If visualize FFT of echo data
468                     elif self.analyze_mode.get() == "FFT":
469                         analyze.fft_echo_data()
470
471                     # If visualize disatnce of echo data
472                     elif self.analyze_mode.get() == "Dist":
473                         # Get temperature
474                         if self.v_temperature["text"] == "-":
475                             self.get_environment()
476
477                         # Start visalization
478                         analyze.distance_echo_data(frequency,float(self.
                                v_temperature["text"]),threshold)
479
480 #*********************************************************************
481 #*                              Main                                 *
482 #*********************************************************************
483 # If program running as main, start window
484 if __name__ == "__main__":
485     main = mainwindow()
```

D.2. calculate_pressure_temperature.py

```
1 #*********************************************************************
2 #* University: HAW Hamburg                                           *
3 #* Author: Christopher Rotzlawski                                    *
4 #*                                                                   *
5 #* File: calculate_pressure_temperature.py                           *
```

```
6  #* Version: 1.0                                                    *
7  #*********************************************************************
8  # Import additional modules
9  import os
10
11 # Import project modules
12 import mainwindow
13
14 #*********************************************************************
15 #*                              Class                                *
16 #*********************************************************************
17 # Class to return calculated pressure and temperature
18 class calculate_pressure_temperature():
19     # Method to initialize class and calculate values
20     def __init__(self,data):
21         self.raw_data = data
22         self.disassemble_data()
23         self.calculate_temperature()
24         self.calculate_pressure()
25         self.save_values()
26
27     # Method to disassemble raw data
28     def disassemble_data(self):
29         # Raw pressure and temperature, 1. byte of pressure and
               temperature is NULL
30         self.raw_pressure = (self.raw_data[1] << 16) | (self.raw_data[2]
               << 8) | self.raw_data[3]
31         self.raw_temperature = (self.raw_data[4] << 24) | (self.raw_data
               [5] << 16) | (self.raw_data[6] << 8) | self.raw_data[7]
32
33         # Trimming values for temperature
34         self.T1 = (self.raw_data[9] << 8) | self.raw_data[8]
35         self.T2 = self.calculate_2s_complement((self.raw_data[11] << 8) |
                self.raw_data[10])
36         self.T3 = self.calculate_2s_complement((self.raw_data[13] << 8) |
                self.raw_data[12])
37
38         # Trimming values for pressure
39         self.P1 = (self.raw_data[15] << 8) | self.raw_data[14]
40         self.P2 = self.calculate_2s_complement((self.raw_data[17] << 8) |
                self.raw_data[16])
41         self.P3 = self.calculate_2s_complement((self.raw_data[19] << 8) |
                self.raw_data[18])
```

```
42          self.P4 = self.calculate_2s_complement((self.raw_data[21] << 8) |
                self.raw_data[20])
43          self.P5 = self.calculate_2s_complement((self.raw_data[23] << 8) |
                self.raw_data[22])
44          self.P6 = self.calculate_2s_complement((self.raw_data[25] << 8) |
                self.raw_data[24])
45          self.P7 = self.calculate_2s_complement((self.raw_data[27] << 8) |
                self.raw_data[26])
46          self.P8 = self.calculate_2s_complement((self.raw_data[29] << 8) |
                self.raw_data[28])
47          self.P9 = self.calculate_2s_complement((self.raw_data[31] << 8) |
                self.raw_data[30])
48
49      # Method to calculate 2's complement
50      def calculate_2s_complement(self,value):
51          if (value & (1 << 15)) != 0:
52              value = value - (1 << 16)
53
54          return value
55
56      # Method to calculate temperature
57      def calculate_temperature(self):
58          var1_t = (float(self.raw_temperature) / 16384.0 - float(self.T1)
                / 1024.0) * float(self.T2)
59          var2_t = ((float(self.raw_temperature) / 131072.0 - float(self.T1
                ) / 8192.0) * (float(self.raw_temperature) / 131072.0 - float(
                self.T1) / 8192.0)) * float(self.T3)
60          self.t_fine = var1_t + var2_t
61
62          self.temperature = self.t_fine / 5120.0
63
64      # Method to calculate pressure
65      def calculate_pressure(self):
66          var1_p = self.t_fine / 2.0 - 64000.0
67          var2_p = var1_p * var1_p * float(self.P6) / 32768.0
68          var2_p = var2_p + var1_p * float(self.P5) / 2.0
69          var2_p = var2_p / 4.0 + float(self.P4) * 65536.0
70          var1_p = (float(self.P3) * var1_p * var1_p / 524288.0 + float(
                self.P2) * var1_p) / 524288.0
71          var1_p = (1.0 + var1_p / 32768.0) * float(self.P1)
72
73          p = 1048576.0 - float(self.raw_pressure)
74          p = (p - var2_p / 4096.0) * 6250.0 / var1_p
75
```

```
76          var1_p = float(self.P9) * p * p / 2147483648.0
77          var2_p = p * float(self.P8) / 32768.0
78
79          self.pressure = p + (var1_p + var2_p + float(self.P7)) / 16.0
80
81      # Method to save values and destroy object
82      def save_values(self):
83          if os.path.exists("pressure_temperature.csv"):
84              os.remove("pressure_temperature.csv")
85
86          file = open("pressure_temperature.csv","w")
87          file.write(str(self.pressure) + ";" + str(self.temperature))
88          file.close()
89
90  #*************************************************************************
91  #*                                  Main                                 *
92  #*************************************************************************
93  # If program running as main, start window
94  if __name__ == "__main__":
95      main = mainwindow.mainwindow()
```

D.3. ethernet_network.py

```
1   #*************************************************************************
2   #* University: HAW Hamburg                                               *
3   #* Author: Christopher Rotzlawski                                        *
4   #*                                                                       *
5   #* File: ethernet_network.py                                             *
6   #* Version: 1.0                                                          *
7   #*************************************************************************
8   # Import additional modules
9   import tkinter
10  import os
11  import socket
12  import numpy
13  import scipy.io
14
15  # Import project modules
16  import mainwindow
17  import ethernet_network_setup
18
19  #*************************************************************************
20  #*                                  Class                                *
```

```
21 #*****************************************************************
22 # Class to connect network
23 class ethernet_network():
24     # Method to initialize class
25     def __init__(self,sensors):
26         # Set variables
27         self.vec_ip_base = ethernet_network_setup.get_ip_base_vec()
28         self.int_subnet_mask = ethernet_network_setup.get_subnet_int()
29         self.int_port_base = ethernet_network_setup.get_port_base()
30         self.int_sensors = sensors
31 
32         # Initialize network
33         self.set_sockets()
34 
35     # Method to check sensor connection
36     def check_sensors(self,sensor_number):
37         # Set sensor number
38         int_sensor_number = sensor_number
39 
40         # Get message for status query
41         status_query = (ethernet_network_setup.get_status_prototype()).
               to_bytes(5,"big")
42 
43         # Calculate destination IP
44         vec_destination_ip = self.vec_ip_base[:]
45         vec_destination_ip[3] += (1 + int_sensor_number)
46 
47         if ((self.int_subnet_mask & 0xFF00) >> 8) == 0:
48             vec_destination_ip[2] += 1
49 
50         str_destination_ip = str(vec_destination_ip[0]) + "." + str(
               vec_destination_ip[1]) + "." + str(vec_destination_ip[2]) + ".
               " + str(vec_destination_ip[3])
51 
52         self.pc_socket.sendto(status_query,(str_destination_ip,self.
               int_destination_port))
53 
54         self.status_socket.settimeout(2.0)
55 
56         check = True
57         try:
58             recv_data, recv_addr = self.status_socket.recvfrom(1472)
59         except socket.timeout:
60             check = False
```

```
61
62          self.status_socket.settimeout(None)
63
64          # Check status response
65          try:
66              if int(recv_data[3]) != 0xAA:
67                  check = False
68          except:
69              check = False
70
71          return check
72
73      # Method to initialize sockets
74      def set_sockets(self):
75          # Set destination port
76          self.int_destination_port = self.int_port_base + 1
77
78          # Set pc socket
79          vec_ip_pc = self.vec_ip_base[:]
80          vec_ip_pc[3] += 1
81
82          if ((self.int_subnet_mask & 0xFF00) >> 8) == 0:
83              vec_ip_pc[2] += 1
84
85          str_ip_pc = str(vec_ip_pc[0]) + "." + str(vec_ip_pc[1]) + "." +
                str(vec_ip_pc[2]) + "." + str(vec_ip_pc[3])
86          int_port_pc = self.int_port_base
87
88          self.pc_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
89          self.pc_socket.bind((str_ip_pc,int_port_pc))
90
91          # Set echo socket
92          int_port_echo = self.int_port_base + 1
93
94          self.echo_socket = socket.socket(socket.AF_INET, socket.
                SOCK_DGRAM)
95          self.echo_socket.bind((str_ip_pc,int_port_echo))
96
97          # Set environment socket
98          int_port_environment = self.int_port_base + 2
99
100         self.environment_socket = socket.socket(socket.AF_INET, socket.
                SOCK_DGRAM)
101         self.environment_socket.bind((str_ip_pc,int_port_environment))
```

```
102
103         # Set status socket
104         int_port_status = self.int_port_base + 3
105
106         self.status_socket = socket.socket(socket.AF_INET, socket.
                SOCK_DGRAM)
107         self.status_socket.bind((str_ip_pc,int_port_status))
108
109     # Method to close sockets
110     def close_sockets(self):
111         self.pc_socket.close()
112         self.echo_socket.close()
113         self.environment_socket.close()
114         self.status_socket.close()
115
116     # Method to receive environmental values
117     def get_environmental_values(self,sensor_number):
118         # Set sensor number
119         int_sensor_number = sensor_number
120
121         # Get message for environmental query
122         environmental_query = (ethernet_network_setup.
                get_environment_prototype()).to_bytes(5,"big")
123
124         # Calculate destination IP
125         vec_destination_ip = self.vec_ip_base[:]
126         vec_destination_ip[3] += (1 + int_sensor_number)
127
128         if ((self.int_subnet_mask & 0xFF00) >> 8) == 0:
129             vec_destination_ip[2] += 1
130
131         str_destination_ip = str(vec_destination_ip[0]) + "." + str(
                vec_destination_ip[1]) + "." + str(vec_destination_ip[2]) + "."
                " + str(vec_destination_ip[3])
132
133         self.pc_socket.sendto(environmental_query,(str_destination_ip,
                self.int_destination_port))
134
135         self.status_socket.settimeout(2.0)
136
137         try:
138             recv_data, recv_addr = self.environment_socket.recvfrom(1472)
139         except socket.timeout:
140             pass
```

```
141
142             self.status_socket.settimeout(None)
143
144             return recv_data
145
146         # Method to transmit instruction package
147         def transmit_instruction(self,sensor_number,udp_package):
148             # Set values
149             int_sensor_number = sensor_number
150             int_udp_package = udp_package
151
152             # Calculate destination IP
153             vec_destination_ip = self.vec_ip_base[:]
154             vec_destination_ip[3] += (1 + int_sensor_number)
155
156             if ((self.int_subnet_mask & 0xFF00) >> 8) == 0:
157                 vec_destination_ip[2] += 1
158
159             str_destination_ip = str(vec_destination_ip[0]) + "." + str(
                    vec_destination_ip[1]) + "." + str(vec_destination_ip[2]) + "."
                    " + str(vec_destination_ip[3])
160
161             # Transmit UDP package
162             bytes_udp_package = int_udp_package.to_bytes(5,"big")
163             self.pc_socket.sendto(bytes_udp_package,(str_destination_ip,self.
                    int_destination_port))
164
165         # Method to receive echo data
166         def receive_echo_data(self,sensors,echo_time):
167             # Set values
168             int_sensors = sensors
169             int_echo_time = echo_time
170             int_max_payload = ethernet_network_setup.get_udp_payload()
171             int_adc_samplerate = ethernet_network_setup.get_adc_samplerate()
172             values = 0
173             status = 0
174             values_available = True
175             check = True
176
177             # Calculate max data per sensor
178             max_packages = int(int_echo_time * int_adc_samplerate /
                    int_max_payload * 2 + 0.5) + 1
179             max_values = int(int_echo_time * int_adc_samplerate + 0.5)
180
```

```
181         # Empty vectors for received echos and addresses
182         recv_data = list(numpy.zeros(int_sensors*max_packages))
183         sensor_addr = list(numpy.zeros(int_sensors*max_packages))
184         echo_data = numpy.zeros((int_sensors,max_values))
185         sensor_value_pos = list(numpy.zeros(int_sensors, dtype = int))
186 
187         # Empty vectors for status response
188         recv_status = list(numpy.zeros(int_sensors))
189         sensor_status = list(numpy.zeros(int_sensors))
190 
191         # Set timeout
192         self.echo_socket.settimeout(2.0)
193 
194         # Get echo datas
195         while values_available:
196             try:
197                 recv_data[values],sensor_addr[values] = self.echo_socket.
                        recvfrom(1472)
198                 values += 1
199             except socket.timeout:
200                 values_available = False
201 
202         self.echo_socket.settimeout(None)
203         self.status_socket.settimeout(2.0)
204 
205         # Check status response
206         values_available = True
207 
208         while values_available:
209             try:
210                 recv_status[status],sensor_status[status] = self.
                        status_socket.recvfrom(1472)
211                 status += 1
212             except socket.timeout:
213                 values_available = False
214 
215         self.status_socket.settimeout(None)
216 
217         for i in range(0,int_sensors):
218             try:
219                 if int(recv_status[i][3]) != 0xF0:
220                     print(str(sensor_status[i]) + " status response is
                            missing")
221                     check = False
```

```
222             except:
223                 check = False
224 
225         # Separate echo data
226         for i in range(0,values):
227             str_sensor_addr = str(sensor_addr[i])
228             find_number_start = 0
229             find_number_end = 0
230 
231             # Find startpoint of sensor number
232             for j in range(0, len(str_sensor_addr)):
233                 if str_sensor_addr[j] == ".":
234                     find_number_start += 1
235                 if find_number_start >= 3:
236                     # Find endpoint of sensor number
237                     for k in range(0, len(str_sensor_addr)):
238                         if str_sensor_addr[k] == "'":
239                             find_number_end += 1
240                         if find_number_end >= 2:
241                             break
242                     break
243 
244             # Get sensor number
245             sensor_number = int(str_sensor_addr[j+1:k])-2
246 
247             # Save echo data in array
248             for l in range(0,int(len(recv_data[i])/2)-1):
249                 echo_data[sensor_number,sensor_value_pos[sensor_number]]
                        = int(recv_data[i][2*l] << 8) | int(recv_data[i][2*l
                        +1])
250 
251                 # Increment sensor value position
252                 sensor_value_pos[sensor_number] += 1
253 
254         # Save echo data in mat file
255         scipy.io.savemat("echo_data.mat",{"echo_data":echo_data})
256 
257         # Return status as bool
258         return check
259 
260 #*****************************************************************
261 #*                              Main                             *
262 #*****************************************************************
263 # If program running as main, start window
```

```
264 if __name__ == "__main__":
265     main = mainwindow.mainwindow()
```

D.4. ethernet_network_setup.py

```
1  #*************************************************************************
2  #* University: HAW Hamburg                                               *
3  #* Author: Christopher Rotzlawski                                        *
4  #*                                                                       *
5  #* File: ethernet_network_setup.py                                       *
6  #* Version: 1.0                                                          *
7  #*************************************************************************
8  # Import project modules
9  import mainwindow
10
11 #*************************************************************************
12 #*                        Network Informations                           *
13 #*************************************************************************
14 global ip_base, subnet_mask, port_base, max_package_payload
15 global adc_samplerate_khz
16
17 ip_base = "192.168.0.0"
18 subnet_mask = "255.255.0.0"
19 port_base = 50000
20 max_package_payload = 1470
21 adc_samplerate_khz = 200
22
23 #*************************************************************************
24 #*                         Message Prototypes                            *
25 #*************************************************************************
26 global status, temp_press, master, slave, sweep_const
27 global frequency_mask, frequency_shift
28 global burst_time_mask, burst_time_shift, echo_time_mask
29
30 status = 1 << 39
31 environment = 1 << 38
32 master = 1 << 37
33 slave = 1 << 36
34 sweep_const = 1 << 35
35 amplitude_mask = 0xF
36 amplitude_shift = 31
37 frequency_mask = 0x7F
38 frequency_shift = 20
```

```
39 burst_time_mask = 0xFFF
40 burst_time_shift = 8
41 echo_time_mask = 0xFF
42
43 #*************************************************************************
44 #*                                Functions                              *
45 #*************************************************************************
46 # Function to return IP base as string
47 def get_ip_base_str():
48     return ip_base
49
50 # Function to ruturn subnet mask as string
51 def get_subnet_str():
52     return subnet_mask
53
54 # Function to return ip base as vector
55 def get_ip_base_vec():
56     ip_length = len(ip_base)
57     ptr_pos = 0
58     ptr_value = 0
59     int_ip_base = [0,0,0,0]
60     for i in range(0, ip_length):
61         if subnet_mask[i] == ".":
62             int_ip_base[ptr_value] = int(ip_base[ptr_pos:i])
63             ptr_pos = i + 1
64             ptr_value += 1
65             if ptr_value > 2:
66                 int_ip_base[3] = int(ip_base[i+1:ip_length])
67                 break
68
69     return int_ip_base
70
71 # Function to return subnet mask as vector
72 def get_subnet_vec():
73     mask_length = len(subnet_mask)
74     ptr_pos = 0
75     ptr_value = 0
76     int_subnet_mask = [0,0,0,0]
77     for i in range(0, mask_length):
78         if subnet_mask[i] == ".":
79             int_subnet_mask[ptr_value] = int(subnet_mask[ptr_pos:i])
80             ptr_pos = i + 1
81             ptr_value += 1
82             if ptr_value > 2:
```

```
83                  int_subnet_mask[3] = int(subnet_mask[i+1:mask_length])
84                  break
85
86      return int_subnet_mask
87
88  # Function to return subnet mask as integer
89  def get_subnet_int():
90      vec_subnet_mask = get_subnet_vec()
91      int_subnet_mask = (vec_subnet_mask[0] << 24) | (vec_subnet_mask[1] <<
            16) | (vec_subnet_mask[2] << 8) | vec_subnet_mask[3]
92
93      return int_subnet_mask
94
95
96  # Function to return port base
97  def get_port_base():
98      return port_base
99
100 # Function to return max UDP package payload
101 def get_udp_payload():
102     return max_package_payload
103
104 # Function to return ADC samplerate
105 def get_adc_samplerate():
106     return adc_samplerate_khz
107
108 # Function to return status prototype
109 def get_status_prototype():
110     return status
111
112 # Function to return temperature and pressure prototype
113 def get_environment_prototype():
114     return environment
115
116 # Function to return master prototype
117 def get_master_prototype():
118     return master
119
120 # Function to return slave prototype
121 def get_slave_prototype():
122     return slave
123
124 # Function to return sweep const prototype
125 def get_sweep_const_prototype():
```

```
126     return sweep_const
127
128 # Function to return mask and shift value for amplitude
129 def get_amplitude_prototype():
130     return amplitude_mask, amplitude_shift
131
132 # Function to return mask and shift value for frequency
133 def get_frequency_prototype():
134     return frequency_mask, frequency_shift
135
136 # Function to return mask and shift value for burst time
137 def get_burst_time_prototype():
138     return burst_time_mask, burst_time_shift
139
140 # Function to return echo time mask
141 def get_echo_time_prototype():
142     return echo_time_mask
143
144 #*****************************************************************
145 #*                              Main                             *
146 #*****************************************************************
147 # If program running as main, start window
148 if __name__ == "__main__":
149     main = mainwindow.mainwindow()
```

D.5. analyze_data.py

```
1  #*****************************************************************
2  #* University: HAW Hamburg                                       *
3  #* Author: Christopher Rotzlawski                                *
4  #*                                                               *
5  #* File: analyze_data.py                                         *
6  #* Version: 1.0                                                  *
7  #*****************************************************************
8  # Import additional modules
9  import scipy.io
10 import scipy.signal
11 import numpy
12 import matplotlib.pyplot
13 import time
14
15 # Import project modules
16 import mainwindow
```

```
17 import ethernet_network_setup
18
19 #*********************************************************************
20 #*                                Class                              *
21 #*********************************************************************
22 class analyze_data():
23     # Method to initialize class
24     def __init__(self,sensors,burst_time,master):
25         # Set variables
26         self.int_sensors = sensors
27         self.int_burst_time = burst_time
28         self.int_master = master
29
30         # Load mat-file
31         self.load_echo_data()
32
33         # Correct master burst time
34         self.master_correct_burst_time()
35
36         # Standardize echo data
37         self.standardize_echo_data()
38
39         # Save echo data in file
40         self.save_echo_data()
41
42     # Method to load echo data
43     def load_echo_data(self):
44         # Load mat-file
45         mat_data = scipy.io.loadmat("echo_data.mat")
46
47         # Save data in vectors
48         self.echo_data = mat_data["echo_data"]
49
50         # Get time vector
51         array_size = numpy.shape(self.echo_data)
52         self.samples = array_size[1]
53         self.sample_rate = ethernet_network_setup.get_adc_samplerate() *
               1000
54         self.time = self.samples / self.sample_rate
55         self.time_vec = numpy.linspace(0,self.time,self.samples,endpoint
               = False)
56
57     # Method to correct master verctor
58     def master_correct_burst_time(self):
```

```
59          # Calculate burst time vector
60          self.burst_samples = int(self.int_burst_time / 1000000 * self.
                sample_rate)
61          burst_ones = numpy.ones(self.burst_samples) * sum(self.echo_data[
                self.int_master-1]) / self.samples
62
63          # Correct master echo data
64          self.echo_data[self.int_master-1][self.burst_samples+1:self.
                samples] = self.echo_data[self.int_master-1][0:self.samples-
                self.burst_samples-1]
65          self.echo_data[self.int_master-1][0:self.burst_samples] =
                burst_ones
66
67      # Method to standardize echo data
68      def standardize_echo_data(self):
69          for i in range(0,self.int_sensors):
70              # Standardize to one
71              self.echo_data[i] = self.echo_data[i] / 0xFFF
72
73      # Method to save echo data
74      def save_echo_data(self):
75          # Get current time
76          current_time = time.strftime
77
78          # Built string for file name
79          file_name = "echo_data_" + current_time("%d") + "_" +
                current_time("%m") + "_" + current_time("%H") + "_" +
                current_time("%M") + "_" + current_time("%S")
80
81          # Save data in mat-file
82          file_mat = file_name + ".mat"
83          scipy.io.savemat(file_mat,{"echo_data":self.echo_data})
84
85          # Save data in txt-file
86          file_txt = file_name + ".txt"
87          numpy.savetxt(file_txt,self.echo_data,delimiter=";",fmt="%.6f")
88
89      # Method to close figures
90      def close_figures(self):
91          matplotlib.pyplot.close("all")
92
93      # Method to visualize raw echo data
94      def raw_echo_data(self):
95          for i in range(0,self.int_sensors):
```

```
96              # Built string for plot title
97              string_sensor = "Raw Echo Sensor " + str(i+1)
98
99              # Create figure for echo data
100             matplotlib.pyplot.figure(i+1)
101             matplotlib.pyplot.plot(self.time_vec,self.echo_data[i])
102             matplotlib.pyplot.title(string_sensor)
103             matplotlib.pyplot.xlabel("Time [s]")
104             matplotlib.pyplot.ylabel("Amplitude")
105             matplotlib.pyplot.xlim(0,self.time)
106
107         # Plot figures
108         matplotlib.pyplot.show()
109
110     # Method to visualize bandpass filtered echo data
111     def bandpass_echo_data(self,order,lowcut,highcut):
112         # Set variables
113         int_order = order
114         int_lowcut = lowcut
115         int_highcut = highcut
116
117         # Calculate filter parameter
118         low = int_lowcut / 0.5 / self.sample_rate * 1000
119         high = int_highcut / 0.5 / self.sample_rate * 1000
120
121         # Calculate bandpass filter
122         num,denum = scipy.signal.butter(int_order,[low,high],btype="band"
                )
123
124         for i in range(0,self.int_sensors):
125             # Built string for plot title
126             string_sensor = "Bandpass Filtered Echo Sensor " + str(i+1)
127
128             # Filter echo data
129             filtered_echo = scipy.signal.lfilter(num,denum,self.echo_data
                    [i])
130
131             # Create figure for echo data
132             matplotlib.pyplot.figure(i+1)
133             matplotlib.pyplot.plot(self.time_vec,filtered_echo)
134             matplotlib.pyplot.title(string_sensor)
135             matplotlib.pyplot.xlabel("Time [s]")
136             matplotlib.pyplot.ylabel("Amplitude")
137             matplotlib.pyplot.xlim(0,self.time)
```

```
138
139         # Plot figures
140         matplotlib.pyplot.show()
141
142     # Method to visualize correlated echo data
143     def correlated_echo_data(self,burst_frequency,visual):
144         # Set variable
145         int_burst_frequency = burst_frequency
146         str_visual = visual
147
148         # Calculate burst
149         burst_time = self.int_burst_time / 1000000
150         time_burst_vec = numpy.linspace(0,burst_time,self.burst_samples,
                endpoint=False)
151         burst = numpy.sin(2*numpy.pi*int_burst_frequency*1000*
                time_burst_vec)
152
153         # Vector for correlated echo data
154         self.xcorr_echo = list(numpy.zeros(self.int_sensors))
155
156         for i in range(0,self.int_sensors):
157             # Built string for plot title
158             string_sensor = "Correlated Echo Sensor " + str(i+1)
159
160             # Correlate echo data
161             self.xcorr_echo[i] = numpy.correlate(self.echo_data[i],burst)
162
163             # If visualize correlated eho data
164             if str_visual == "Visual":
165                 # Time vector for correlated echo data plot
166                 time_xcorr_vec = self.time_vec[0:len(self.xcorr_echo[i])]
167
168                 # Create figure for echo data
169                 matplotlib.pyplot.figure(i+1)
170                 matplotlib.pyplot.plot(time_xcorr_vec,self.xcorr_echo[i])
171                 matplotlib.pyplot.title(string_sensor)
172                 matplotlib.pyplot.xlabel("Time [s]")
173                 matplotlib.pyplot.ylabel("Amplitude")
174                 matplotlib.pyplot.xlim(0,self.time)
175
176         # If visualize correlated echo data, plot figures
177         if str_visual == "Visual":
178             matplotlib.pyplot.show()
179
```

```
180     # Method to visualize enveloped echo data
181     def enveloped_echo_data(self,burst_frequency,visual):
182         # Set variable
183         int_burst_frequency = burst_frequency
184         str_visual = visual
185 
186         # Calculate correlated echo data
187         self.correlated_echo_data(int_burst_frequency,"No")
188 
189         # Vector for enveloped echo data
190         self.enveloped_echo = list(numpy.zeros(self.int_sensors))
191 
192         for i in range(0,self.int_sensors):
193             # Built string for plot title
194             string_sensor = "Enveloped Echo Sensor " + str(i+1)
195 
196             # Enveloped echo data
197             self.enveloped_echo[i] = abs(scipy.signal.hilbert(self.
                    xcorr_echo[i]))
198 
199             # If visualize correlated echo data
200             if str_visual == "Visual":
201                 # Time vector for correlated echo data plot
202                 time_enveloped_vec = self.time_vec[0:len(self.
                        enveloped_echo[i])]
203 
204                 # Create figure for echo data
205                 matplotlib.pyplot.figure(i+1)
206                 matplotlib.pyplot.plot(time_enveloped_vec,self.
                        enveloped_echo[i])
207                 matplotlib.pyplot.title(string_sensor)
208                 matplotlib.pyplot.xlabel("Time [s]")
209                 matplotlib.pyplot.ylabel("Amplitude")
210                 matplotlib.pyplot.xlim(0,self.time)
211 
212         # If visualize correlated echo data, plot figures
213         if str_visual == "Visual":
214             matplotlib.pyplot.show()
215 
216     # Method to visualize FFT of raw echo data
217     def fft_echo_data(self):
218         # Time vector for FFT
219         time_fft_vec = numpy.linspace(0,self.sample_rate/1000,self.
                samples,endpoint=False)
```

```
220
221         for i in range(0,self.int_sensors):
222             # Built string for plot title
223             string_sensor = "FFT Echo Sensor " + str(i+1)
224
225             # Calculate FFT of echo data
226             fft_echo = abs(numpy.fft.fft(self.echo_data[i]))
227
228             # Create figure for echo data
229             matplotlib.pyplot.figure(i+1)
230             matplotlib.pyplot.plot(time_fft_vec,fft_echo)
231             matplotlib.pyplot.title(string_sensor)
232             matplotlib.pyplot.xlabel("Frequency [kHz]")
233             matplotlib.pyplot.ylabel("Amplitude")
234             matplotlib.pyplot.ylim(0,max(fft_echo[int(self.samples/20):
                    int(self.samples/2)]))
235             matplotlib.pyplot.xlim(self.sample_rate/1000/20,self.
                    sample_rate/1000/2)
236
237         # Plot figures
238         matplotlib.pyplot.show()
239
240     # Method to visualize distance of echo data
241     def distance_echo_data(self,burst_frequency,temperature,threshold):
242         # Set variable
243         int_burst_frequency = burst_frequency
244         float_temperature = temperature
245         int_threshold = threshold
246
247         # Calculate enveloped echo data
248         self.enveloped_echo_data(int_burst_frequency,"No")
249
250         # Calculate distance
251         sound_velocity = 331.4 + 0.6 * float_temperature
252         distance = sound_velocity * self.time / 2
253
254         for i in range(0,self.int_sensors):
255             # Built string for plot title
256             string_sensor = "Distance Echo Sensor " + str(i+1)
257
258             # Find peaks
259             peaks = scipy.signal.find_peaks_cwt(self.enveloped_echo[i],
                    numpy.arange(1,5000))
260
```

```
261             # Find peaks above limit
262             counter_peaks = 0
263             for j in range(0,len(peaks)):
264                 if self.enveloped_echo[i][peaks[j]] > int_threshold:
265                     counter_peaks += 1
266 
267             # Separate peaks
268             if counter_peaks > 0:
269                 x_new = list(numpy.zeros(counter_peaks))
270                 y_new = list(numpy.zeros(counter_peaks))
271                 counter_peaks_sep = 0
272 
273                 for k in range(0,len(peaks)):
274                     if self.enveloped_echo[i][peaks[k]] > int_threshold:
275                         x_new[counter_peaks_sep] = peaks[k] / self.
                            samples * distance
276                         y_new[counter_peaks_sep] = self.enveloped_echo[i
                            ][peaks[k]]
277                         counter_peaks_sep += 1
278 
279                     # Create figure for echo data
280                     if counter_peaks_sep == counter_peaks:
281                         matplotlib.pyplot.figure(i+1)
282                         matplotlib.pyplot.stem(x_new,y_new,"-.")
283                         matplotlib.pyplot.title(string_sensor)
284                         matplotlib.pyplot.xlabel("Distance [m]")
285                         matplotlib.pyplot.ylabel("Amplitude")
286                         break
287 
288         # Plot figures
289         matplotlib.pyplot.show()
290 
291 #*********************************************************************
292 #*                              Main                                 *
293 #*********************************************************************
294 # If program running as main, start window
295 if __name__ == "__main__":
296     main = mainwindow.mainwindow()
```

Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „– bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] – ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Dieses Blatt, mit der folgenden Erklärung, ist nach Fertigstellung der Abschlussarbeit durch den Studierenden auszufüllen und jeweils mit Originalunterschrift als letztes Blatt in das Prüfungsexemplar der Abschlussarbeit einzubinden.
Eine unrichtig abgegebene Erklärung kann -auch nachträglich- zur Ungültigkeit des Studienabschlusses führen.

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: Rotzlawski

Vorname: Christopher

dass ich die vorliegende Bachelorarbeit bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Konzeption und Umsetzung eines Entwicklungssystems für intelligente Ultraschallsensoren in Mehrfachanordnung zur Lokalisation und Umgebungserkennung in komplexen Umgebungen

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

- die folgende Aussage ist bei Gruppenarbeiten auszufüllen und entfällt bei Einzelarbeiten -

Die Kennzeichnung der von mir erstellten und verantworteten Teile der Bachelorarbeit ist erfolgt durch:

Hamburg	25.05.2018	
Ort	Datum	Unterschrift im Original