

HOCHSCHULE FÜR TECHNIK UND WIRTSCHAFT  
DES SAARLANDES  
FAKULTÄT INGENIEURSWISSENSCHAFTEN

## Bachelorthesis

---

# Entwicklung einer erweiterbaren C++-Klassenbibliothek zur Ansteuerung eines Ultraschallprüfsystems

---



von  
Jan Oswald

Saarbrücken  
2016

Gutachter: Prof. Dr. Reinhard Brocks

## **Eidesstattliche Erklärung**

Hiermit versichere ich, die vorliegende Abschlussarbeit selbstständig und nur unter Verwendung der von mir angegebenen Quellen und Hilfsmittel verfasst zu haben. Sowohl inhaltlich als auch wörtlich entnommene Inhalte wurden als solche kenntlich gemacht. Die Arbeit hat in dieser oder vergleichbarer Form noch keinem anderem Prüfungsgremium vorgelegen.

Datum: \_\_\_\_\_ Unterschrift: \_\_\_\_\_

## **Vorwort**

Bei dieser Arbeit handelt es sich um meine Bachelor-Thesis im Studiengang Elektrotechnik, Vertiefungsrichtung Automatisierungstechnik an der Hochschule für Technik und Wirtschaft des Saarlandes, welche ich am Fraunhofer Institut für zerstörungsfreie Prüfverfahren (IZFP) in Saarbrücken durchgeführt habe. Die gesamte Software wurde in C++ unter Verwendung von Visual Studio 2013 geschrieben und getestet. Um die Arbeit mit mehreren Softwareentwicklern an einem Projekt zu erleichtern, wurde als Sourcecodeverwaltung ein Team Foundation Server (TFS) eingesetzt.

Ich möchte mich bei meinen Kollegen aus der Abteilung „Elektronik für ZfP-Systeme“ am Fraunhofer IZFP für die Überlassung des Themas bedanken, die mich bei allen Fragen rund um meine Bachelor Thesis unterstützt haben. Mein besonderer Dank gilt außerdem M. Sc. Christoph Weingard und M. Sc. Michael Ganster, die mich bei Problemen oder auftretenden Fragestellungen stets unterstützt haben. Weiterhin gilt mein besonderer Dank Herrn Prof. Dr. Reinhard Brocks für die Betreuung und Begutachtung dieser Arbeit von Seiten der Hochschule für Technik und Wirtschaft.

# Inhaltsverzeichnis

<b>1</b>	<b>Einführung</b>	<b>1</b>
1.1	Umfeld der Arbeit . . . . .	1
1.2	Prüfsystem . . . . .	1
<b>2</b>	<b>Physikalische Grundlagen</b>	<b>3</b>
2.1	Ultraschall-Grundlagen . . . . .	3
2.2	Gerichtete Wellen . . . . .	4
2.3	Darstellung . . . . .	5
2.3.1	A-Scan . . . . .	5
2.3.2	B-Scan . . . . .	6
2.4	Elektromagnetischer Ultraschall . . . . .	7
2.5	Lorentzkraft . . . . .	8
2.6	Magnetostriktion . . . . .	8
<b>3</b>	<b>Hardware</b>	<b>9</b>
3.1	Status-Ablauf . . . . .	9
3.2	Kommunikation . . . . .	10
3.3	Kommandos . . . . .	11
3.4	Antworten . . . . .	12
3.5	Messdaten . . . . .	13
<b>4</b>	<b>Software</b>	<b>14</b>
4.1	EmusBaseClass . . . . .	15
4.2	SimpleEvent . . . . .	16
4.3	EmusHandling . . . . .	17
4.4	EmusFrontEnd . . . . .	18
4.4.1	XML-File . . . . .	19
4.4.2	FTP-Verbindung . . . . .	20
4.4.3	Hardware-Verbindung . . . . .	20
4.4.4	Threading . . . . .	21
4.4.5	AnswerThread . . . . .	22
4.4.6	Antwort-Auswertung . . . . .	24
4.4.7	DataThread . . . . .	25
4.4.8	Daten-Auswertung . . . . .	26
4.4.9	Timing . . . . .	28
4.4.10	Set-Sate . . . . .	30

---

<b>5</b>	<b>Zusammenfassung der Hardware-Steuerung</b>	<b>32</b>
5.1	Geerbte Basisfunktionen für LimaTest . . . . .	32
<b>6</b>	<b>Testumgebung und Resultate</b>	<b>35</b>
6.1	Testumgebung . . . . .	36
6.2	Resultate . . . . .	38
<b>7</b>	<b>Zusammenfassung und Ausblick</b>	<b>41</b>

# 1 Einführung

## 1.1 Umfeld der Arbeit

Ziel dieser Arbeit ist es, in Form einer erweiterbaren Klassenbibliothek, die Softwareanbindung eines Ultraschallprüfsystems neu zu entwickeln.

Ein solches System besteht in diesem Fall aus einer Elektronik, welche die eigentlichen Ultraschallfunktionen verwaltet, einem Prüfkopf zum Erzeugen bzw. Aussenden der Schallwellen und einem Computer mit der entsprechenden Software zur Steuerung der Elektronik.

Erforderlich wird die Neuentwicklung der Software, da auch für die Elektronik neue Generationen entwickelt werden und für ein neues System eine neue Softwareanbindung benötigt wird, da die alte Software nicht auf eine neue Elektronik anpassbar ist. Dies hängt primär mit dem Aufbau der alten Elektronik zusammen, welcher eine Unterscheidung in verschiedene Hardware-Generationen in seiner aktuellen Form nicht zulässt.

Ziel war es nun eine Software zu entwickeln, welche mit der alten Elektronik arbeiten kann und möglichst einfach um andere Hardwareschnittstellen erweiterbar ist. Parallel dazu wurde auch die Benutzerschnittstelle inklusive graphischer Oberfläche neu entwickelt.

Das hier vorgestellte Projekt bildet einen Teilaspekt eines größeren Gesamtprojektes ab, welches in Absprache mit einem externen Kunden bearbeitet wird.

Das Fraunhofer IZFP ist ein in Saarbrücken ansässiges Institut, welches sich mit den physikalischen Methoden der zerstörungsfreien Prüfung (ZfP) und ihrer Anwendung befasst. Hierbei umfasst das Leistungsspektrum des Institutes die gesamte Bandbreite der ZfP, von den physikalischen Grundlagen der eingesetzten Verfahren über die Entwicklung und den Aufbau der entsprechenden Geräte bis hin zu konkreten Prüfaufgaben.

## 1.2 Prüfsystem

Die Bezeichnung LimaTest ist eine Abkürzung für Licht-Mast-Test und es handelt sich dabei um ein Ultraschallprüfsystem, das mit elektromagnetischem Ultraschall (EMUS) arbeitet und für die zerstörungsfreie Standfestigkeitsprüfung von Lichtmasten entwickelt wurde.

Bei diesem System handelt es sich im wesentlichen um den in Abbildung 1.1 dargestellten Aufbau, welcher aus folgenden Komponenten besteht:

- Prüfelektronik (1)
- Manipulator mit EMUS-Prüfkopf (2) und Motor (3)
- Prüfrechner (4) mit entsprechender Software

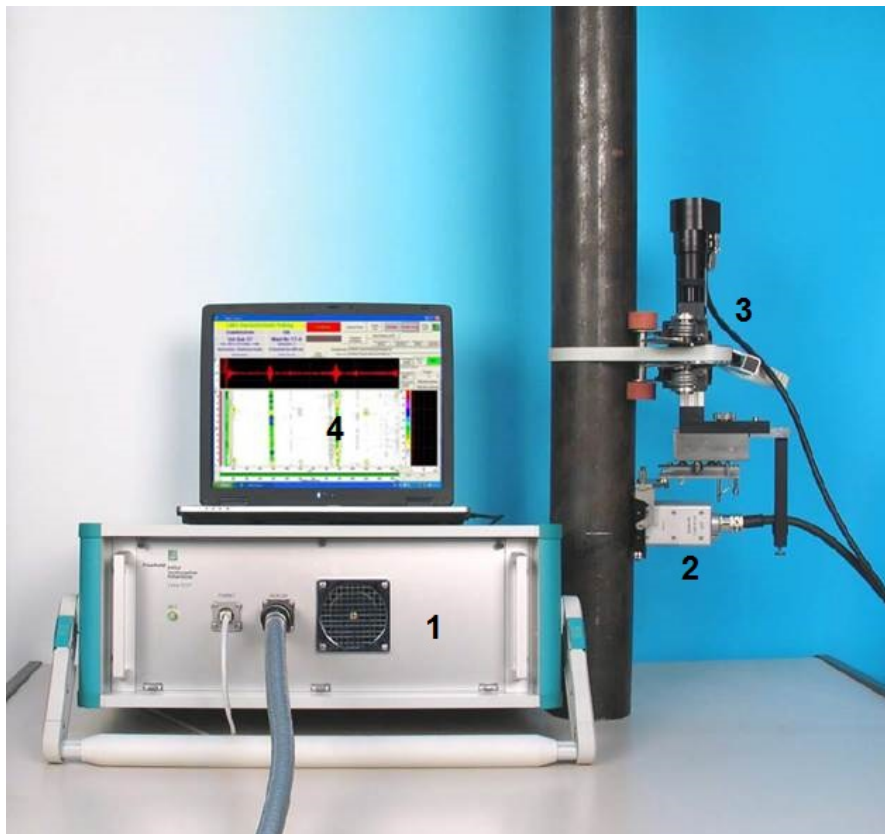


Abbildung 1.1: LimaTest

Zweck des Aufbaus ist es anhand der aufgenommenen Messdaten z.B. Korrosion am Mast unter der Erdoberfläche zu erkennen und so beurteilen zu können, ob der Mast gegebenenfalls verstärkt oder ersetzt werden muss. Ursprünglich wurde der Prüfkopf, während der Messung, von einem Motor um den Mast geführt. Der Motor wird jedoch wegen der aufwändigen Befestigung nur selten bis gar nicht genutzt. Aus diesem Grund entfällt der Motor und damit auch die Motorsteuerung in neueren Versionen der Software.

Von diesem Aufbau sind im Moment zwei Generationen im Einsatz, welche sich nur leicht in der Steuerung der Magnetisierung aber besonders deutlich in der Kommunikation mit dem Prüfrechner unterscheiden und die im Folgenden als „SSV“- bzw. „PROST“-Modul bezeichnet werden. Die Unterschiede werden in den Kapiteln 3.1 und 4.4 näher beschrieben.

## 2 Physikalische Grundlagen

Schall bezeichnet eine mechanische Wellenausbreitung, bei der Energie durch ein Medium übertragen wird. Dabei werden sowohl die Eigenschaften der Ausbreitung durch das Medium beeinflusst, als auch das Medium selbst durch die sich ausbreitende Welle.

### 2.1 Ultraschall-Grundlagen

Schall breitet sich grundsätzlich in Form einer Welle in einem Medium aus. Die Ausbreitungsgeschwindigkeit hängt hierbei maßgeblich von den spezifischen Eigenschaften des Mediums, insbesondere von seiner Dichte, ab. Die Wellenlänge, also der kleinste Abstand zweier Punkte gleicher Phase, wird durch die Geschwindigkeit des Schalls und seiner Frequenz bestimmt. Wenn eine Ultraschallwelle in einem Medium auf eine Stelle mit veränderter Dichte trifft, wird ein Teil der Energie in andere Richtungen reflektiert. Trifft die Welle auf eine senkrecht zur Ausbreitungsrichtung stehende Grenzfläche, wird die Welle in Richtung der Quelle reflektiert, wie in Abb. 2.1 gezeigt.

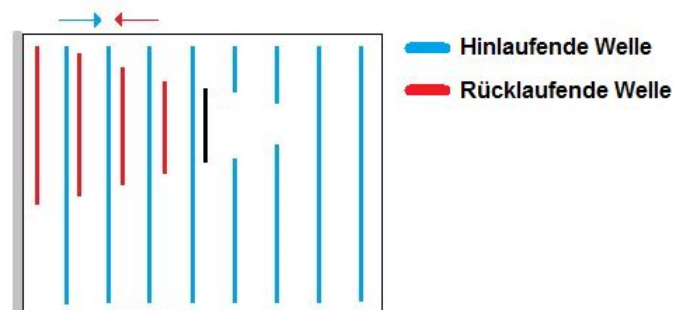


Abbildung 2.1: Reflexion an einer Störstelle

Diese Reflexion macht man sich in technischen Prüfanwendungen zu Nutze, um Informationen über Fehlstellen im Inneren eines Prüfkörpers zu erhalten. Bei der Werkstückprüfung mit Ultraschall gibt es zwei verbreitete Verfahren. Zum einen kann bei einer Durchschallungs- bzw. Transmissionsmessung mit zwei unabhängigen Prüfköpfen gearbeitet werden, wobei ein Wandler in das Material einschallt und der Andere den das Material durchdringenden Anteil der Schallwelle aufnimmt. Zum anderen findet die Impuls-Echo-Methode Anwendung, bei der ein Prüfkopf sowohl als Sender als auch Empfänger fungiert. Diese Methode hat den Vorteil, dass über die materialspezifische Schallgeschwindigkeit und die Zeit, die zwischen

Sendeimpuls und Empfangen der Reflektion vergeht, die genaue Position einer Störstelle ermittelt werden kann. Dieser Zusammenhang wird in Gleichung 2.1 dargestellt.

$$s = \frac{c_m * \Delta t}{2} \quad (2.1)$$

Gleichung 2.1:  $s$  Distanz zwischen Quelle und Reflektionsstelle,  $c_m$  Schallgeschwindigkeit im Medium,  $\Delta t$  Zeit zwischen Senden und Empfangen

## 2.2 Gerichtete Wellen

Bei Verwendung mehrerer Quellen kann durch phasengesteuerte Anregung und geschickte Ausnutzung von Interferenzeffekten die Ausbreitungsrichtung der Wellen entscheidend beeinflusst werden. Dabei kann man zwei grundlegende Ausprägungen dieses Effektes unterscheiden: Die konstruktive Interferenz, bei der sich die Amplituden positiv verstärken und die destruktive Interferenz, bei der sich die Amplituden gegenseitig schwächen oder sogar auslöschen (Abb. 2.2).

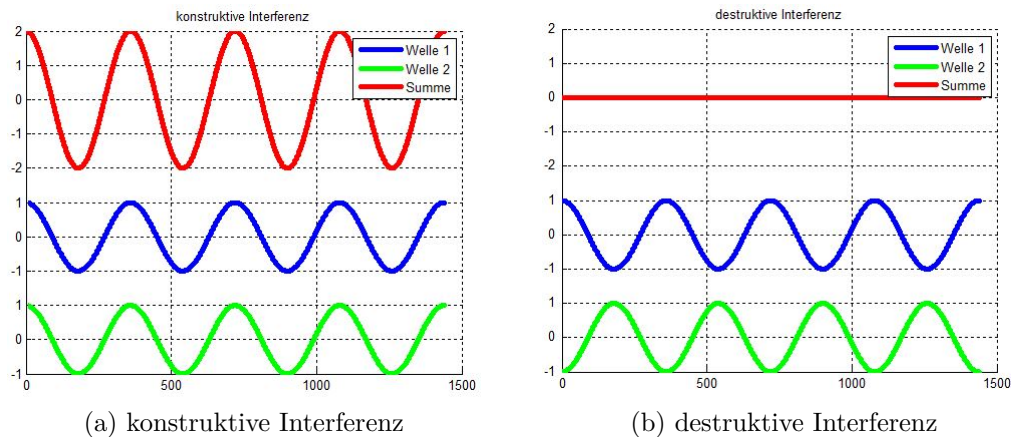


Abbildung 2.2: Interferenz

In Abbildung 2.3 ist schematisch dargestellt, wie sich Schallwellen, die durch mehrere zeitlich versetzt angeregte Quellen erzeugt werden, ausbreiten. Die Zahlen geben an, in welcher zeitlichen Folge die Wellen erzeugt werden, wobei die Verzögerung zwischen den Wellen davon abhängt welcher Winkel durch die Überlagerung erreicht werden soll. Generell gilt in diesem Fall: Je größer die Verzögerung, desto größer der resultierende Abstrahlwinkel.

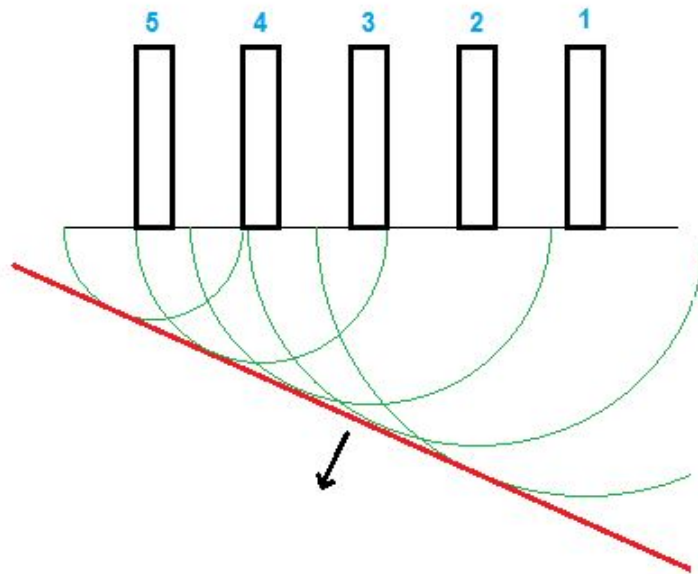


Abbildung 2.3: Überlagerung von Wellen

Werden beispielsweise die äußeren Quellen zuerst getriggert, ist es möglich, eine Fokussierung zu erreichen, indem die Verzögerung so gewählt wird, dass sich alle Schallwellen in einem bestimmten Punkt treffen.

## 2.3 Darstellung

Zur Darstellung von Ultraschallmesswerten gibt es mehrere erprobte Verfahren. Für diese Arbeit besitzen zwei dieser Darstellungsformen, A-Scan und B-Scan, Relevanz.

### 2.3.1 A-Scan

Bei einer klassischen Ultraschallmessung, bei der nur eine Welle ausgesendet wird und die auftretenden Reflektionen über der Zeit oder der Distanz zur Quelle aufgetragen werden, ist ein Amplituden- oder A-Bild (A-Scan) die gängige Methode zur Darstellung (Abb. 2.4). Dabei werden die digitalisierten Amplituden des detektierten Ultraschallsignals als Funktion der Zeit dargestellt. Mit Hilfe der Schallgeschwindigkeit im Prüfkörper und des Zeitpunktes, zu dem ein Echo zu sehen ist, kann eine Aussage über die Distanz der Reflektionsfläche von der Position des Prüfkopfes getroffen werden.

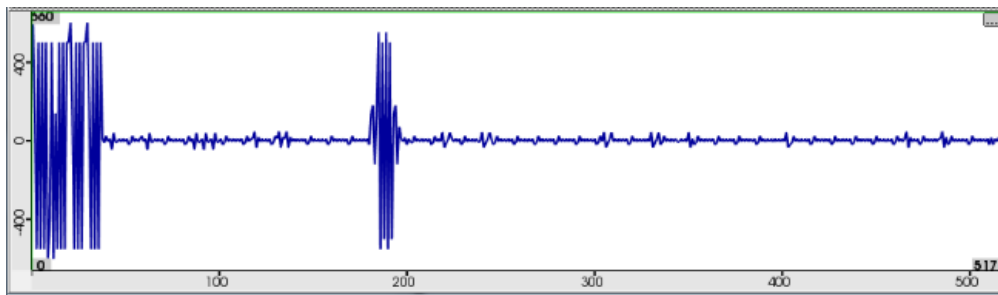


Abbildung 2.4: Reflektion an einer Störstelle

### 2.3.2 B-Scan

Da es durch Betrachten einer Reihe von einzelnen A-Bildern nur schwer möglich ist, eine genaue Aussage über Größe und Form einer Störstelle zu treffen, werden oft mehrere A-Scans zu einem B-Scan zusammengefasst. Dabei wird die in den A-Bildern enthaltene Amplitudeninformation farbkodiert. Ein solcher B-Scan ist in Abbildung 2.5 zu sehen.

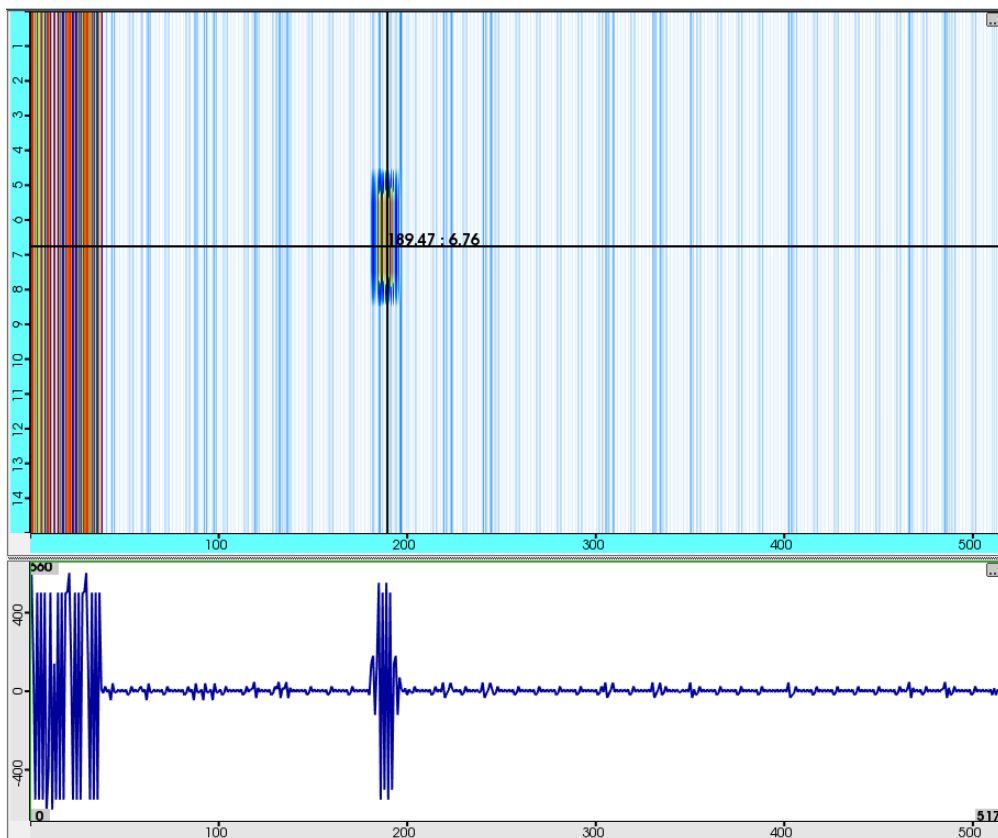


Abbildung 2.5: Reflektion an einer Störstelle

## 2.4 Elektromagnetischer Ultraschall

Bei der Erzeugung von elektromagnetischem Ultraschall wird eine Schallwelle nicht wie bei konventionellen Verfahren im Prüfkopf, sondern durch Einwirken durch eines elektromagnetischen Feldes direkt im Prüfkörper erzeugt. Dieses Verfahren birgt einige Vorteile, so ist beispielsweise kein Koppelmittel zur Übertragung des Schalls vom Prüfkopf in den Prüfkörper nötig, was eine kontaminationsfreie Prüfung möglich macht. Weiterhin hat eine geringe Abhebung des Prüfkopfes während der Messung keine Auswirkung auf die Ergebnisse. Ein solcher EMUS-Wandler, wie er in der vorliegenden Arbeit zum Einsatz kommt, ist in Abb. 2.6 dargestellt.



Abbildung 2.6: Prüfkopf von unten

Bei diesem Verfahren werden durch Überlagerung eines statischen oder quasistatischen Magnetfeldes und eines hochfrequenten Wechselfeldes dynamische Kräfte in einem Prüfkörper erzeugt. Diese Kräfte haben wiederum eine Verformung zur Folge, welche sich in Form einer Schallwelle im Prüfkörper ausbreitet. Umgekehrt kann eine, durch die Welle hervorgerufene, Änderung in der Struktur eines Körpers oder eine Bewegung der Ladungsträger im Material detektiert werden.

In diesem Fall basiert die Schallerzeugung auf zwei physikalischen Effekten, zum einen der Lorentzkraft und zum anderen dem magnetostriktiven Effekt. Diese beiden Phänomene kommen unterschiedlich stark zum tragen, abhängig von den ferromagnetischen Eigenschaften des Materials.

$$\vec{F}_{Ges} = \vec{F}_L + \vec{F}_M + \vec{F}_{MS} \quad (2.2)$$

Gleichung 2.2:  $\vec{F}_{Ges}$  Summe aller Kräfte,  $\vec{F}_L$  Lorentzkraft,  $\vec{F}_M$  magnetische Kraft,  $\vec{F}_{MS}$  magnetostriktive Kraft

## 2.5 Lorentzkraft

Die Lorentzkraft wird durch die Wechselwirkung zwischen elektrischem Strom, bzw. der Stromdichte und einem magnetischen Fluß verursacht. Sie ist allgemein definiert als die Kraft auf eine bewegte Ladung in einem Magnetfeld. Dabei steht die Kraftwirkung im  $90^\circ$  Winkel zum Magnetischen Fluss. Um mit Hilfe der Lorentzkraft eine Ultraschallwelle zu erzeugen werden zunächst Wirbelströme in der Oberfläche des Prüfkörpers eingebracht. Wird diesen Strömen ein senkrechtes, statisches oder quasi statisches Magnetfeld überlagert, wirkt eine Kraft auf die Ladungsträger, welche zu einer Schwingung der Gitterstruktur führt. Diese Methode kann bei allen elektrisch leitfähigen Materialien angewandt werden [Nie10, Seite 25]. Der Empfang von Signalen mit Hilfe der Lorentzkraft basiert auf der Induktion eines elektrischen Feldes auf einen leitfähigen Körper, der durch ein magnetisches Feld bewegt wird. Dies bedeutet also, dass die vorbei laufende Ultraschallwelle durch die verursachte Schwingung im Material und durch das Magnetfeld des Prüfkopfes ein elektrisches Feld erzeugt, welches durch die Spulen im Prüfkopf detektiert werden kann [Nie10, Seite 27].

## 2.6 Magnetostriktion

Nach [Nie10] und [DPV97] werden bei der Magnetostriktion durch Einwirken eines Magnetfeldes auf ein ferromagnetisches Material die magnetischen Domänen im Material neu ausgerichtet, was zu einer Längenänderung im Material führen kann. Dadurch kann durch entsprechend schnelles Wechseln des äußeren Magnetfelds die gewünschte Schall-Frequenz im Material erzeugt werden. Dabei ist die Längenänderung für gewöhnlich deutlich größer als die Volumenänderung. Beim Empfang mit Magnetostriktion wird durch die von der Welle verursachte Verzerrung des Werkstoffs die Vormagnetisierung moduliert, wodurch in den Spulen im Prüfkopf eine messbare Spannung induziert wird.

## 3 Hardware

Die Bezeichnung für die hier verwendete Hardware lautet „EMUS-Frontend“. Das Frontend ist auf Befehl der Software hin selbstständig in der Lage, eine Messung zu starten und Messwerte aufzunehmen. Diese Werte werden dann verschachtelt am entsprechenden Port zur Abnahme durch einen Computer bereitgestellt. Außerdem teilt die Hardware ihren aktuellen Status mit, sofern eine Anfrage zur Statusänderung erfolgt oder der Status direkt abgefragt wird. Die Aufnahme der Messdaten erfolgt über zwei im Prüfkopf verbaute Spulen, die phasenverschoben arbeiten.

Ein Burstgenerator erzeugt das Sendesignal in Form zweier um  $90^\circ$ phasenverschobener, parametrierbarer Rechteck-Signale. Diese beiden Signale werden unabhängig voneinander verstärkt und auf zwei Sendespulen des Prüfkopfes geleitet. Durch die phasengesteuerte Anregung der beiden Sendespulen wird erreicht, dass der Großteil der erzeugten Ultraschallenergie in die gewünschte Richtung abgestrahlt wird. Umgekehrt werden auch beide Spulen als Empfänger genutzt, hierbei wird ähnlich wie beim Senden der Signale von einer  $0^\circ$ - und einer  $90^\circ$ -Komponente gesprochen. Anschließend werden diese beiden von den Spulen empfangenen Signale phasenkorrigiert und addiert, um einen vollständigen Messwert zu erhalten. Im normalen Prüfablauf ist nur die Summe von Interesse, da sie die gewünschten Informationen beinhaltet. Die beiden Teilsignale sind nur für die Kalibrierung des Systems von Interesse.

### 3.1 Status-Ablauf

Hardwareseitig durchläuft LimaTest beim Starten, nach dem Konzept einer „State Machine“, mehrere Zustände, welche unter anderem in Abbildung 3.1 zu sehen sind. Die Zustände werden automatisch in der abgebildeten Reihenfolge durchlaufen bis der Status „IDLE“erreicht ist. Ab diesem Punkt geht die Kontrolle an den Prüfrechner über und die Hardware wartet auf Anweisungen. Das bedeutet, sobald die Kontrolle an den PC abgegeben wurde, ändert sich der Zustand nur noch auf Anweisung des Computers und alle Zustände, die während des Startvorgangs durchlaufen wurden, werden nicht mehr zur weiteren Steuerung benötigt. Zustandsänderungen können, wie in endlichen Zustandsautomaten vorgesehen, nur entsprechend der im Diagramm 3.1 angegebenen Abläufen durchgeführt werden. Nicht spezifizierte Zustandsänderungen werden von der Hardware abgelehnt.

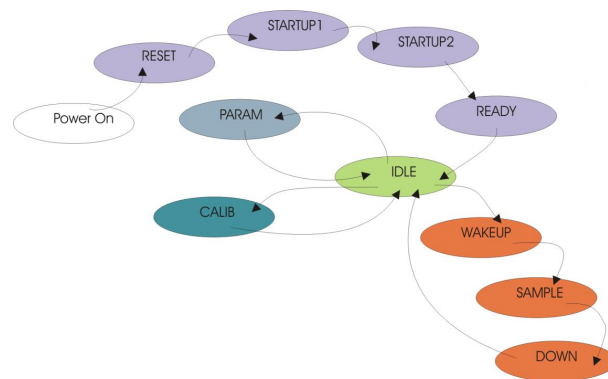


Abbildung 3.1: LimaTest Status Diagramm

Die für die Steuerung der Hardware entscheidenden Zustände sind:

Status	Funktion
Param	Das Parameter-File kann übergeben werden und der neue Parametersatz wird geladen.
Calib	Eine Messung kann durchgeführt werden, bei der nicht wie in Sample nur die Summe der Messdaten gesendet wird sondern auch die beiden einzelnen Komponenten verfügbar sind. Die Magnetisierung wird beim Verlassen von Calib nicht automatisch beendet, daher muss ein zusätzlicher Befehl gesendet werden
IDLE	In diesem Zustand gibt die Elektronik die Kontrolle an den PC ab und wartet auf Anweisungen.
Wakeup	Hier werden die Parameterdaten intern an einen FPGA übergeben und die Hardware wird bereit zur Messung.
Sample	Die Messung wird durchgeführt. Das heißt, die Magnetisierung wird gestartet und an Port 2001 werden Messdaten bereitgestellt.
Down	In Down wird die Messung beendet.

Tabelle 3.1: Zur Steuerung relevante Zustände

## 3.2 Kommunikation

Die Kommunikation mit der Hardware erfolgt softwareseitig über eine TCP Verbindung. Dabei werden ein Port für die Steuerung der Hardware (Port 2000) und ein Port für die Übertragung von Messwerten (Port 2001) verwendet (Abb. 3.2). Das bedeutet, um beispielsweise den Zustand zu wechseln (siehe Kapitel 3.1), muss der entsprechende Befehl auf Port 2000 gesendet werden und die Hardware wird ebenfalls an Port 2000 eine entsprechende Antwort bereit stellen. Nach dem Verbinden mit dem PC wird von der Hardware an Port 2000 eine Willkommensnachricht gesendet („00003SWM“).

Während einer Messung werden auf Port 2001 die Messdaten zur Verfügung gestellt, welche je nach Zustand (siehe Kapitel 3.1) verschieden lang und unterschiedlich kodiert sind. Intern legt das EMUS-Frontend alle auszulesenden Daten für die beiden Ports in je einem Puffer ab, was es notwendig macht, die gesamten anfallenden Daten auszulesen, da sonst gegebenenfalls die nachfolgende Nachrichten unbrauchbar werden, da unter Umständen noch ein Teil eines nicht vollständig ausgelesenen Paketes im Hardwarebuffer steht und dann am Anfang des nächsten Paketes gelesen wird.

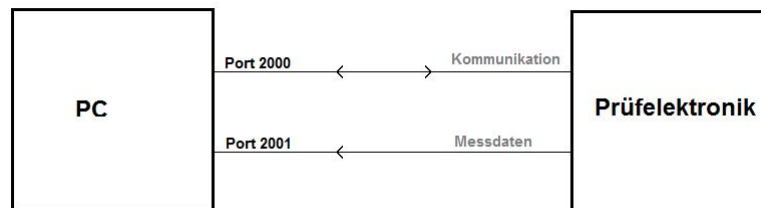


Abbildung 3.2: Kommunikations-Diagramm

### 3.3 Kommandos

Alle Kommandos sind nach einem einheitlichen Schema aufgebaut:

- Byte: 0-4 ( 5 Zeichen): Die Länge des Befehls ab Byte 5
- Byte: 5-7 ( 3 Zeichen): Ziel-Controller des Befehls
- Byte: 8-27 ( 20 Zeichen): Der Befehlscode

Um beispielsweise den Status zu ändern, muss einer der in Tab. 3.2 dargestellten Befehle gesendet werden. Sobald die Elektronik die Verarbeitung eines Statuswechsels abgeschlossen hat, wird sie mit dem erreichten Zustand (Siehe Abb. 3.1) antworten.

Paketinhalt	Status
00015 121 0000 0201 0000	RESET
00015 121 0000 0201 0001	STARTUP1
00015 121 0000 0201 0002	STARTUP2
00015 121 0000 0201 0003	READY
00015 121 0000 0201 0004	IDLE
00015 121 0000 0201 0005	PARAM
00019 121 0000 0201 0006 0001	CALIB
00015 121 0000 0201 0007	WAKEUP
00019 121 0000 0201 0008 0001	SAMPLE
00015 121 0000 0201 0009	DPWN
00015 121 0000 0201 000f	CRASH

Tabelle 3.2: Kommandos, um Status zu setzen

Bei dem im Wechsel nach Calib und Sample zusätzlich übergebenen Wert handelt es sich um eine Zyklusinformation, welche je nach Prüfaufgabe für die Hardware relevant ist. In dieser Anwendung gibt es allerdings nur einen Zyklus, weshalb immer eine 1 gesendet wird.

### 3.4 Antworten

Ebenso wie die Kommandos, sind auch die zu erwartenden Antworten nach einem festen Schema aufgebaut.

-Byte: 0-4 ( 5 Zeichen): Die Länge der nachfolgenden Antwort

-Byte: 5-7 ( 3 Zeichen): Die Quelle der Antwort

-Byte: 8-4114 ( 4096 Zeichen): Der Inhalt der erhaltenen Antwort

Die meisten Stellen der bis zu 4090 Zeichen langen Antwort enthalten keine Informationen, es handelt sich hier um Füll-Bytes, die von der Firmware der Hardware generiert werden. Im Regelfall ist auf Port 2000 mit den in Tabelle 3.3 dargestellten Antworten zu rechnen.

Länge	Paketinhalt	Antwort
4099	MR1 0000 0000 0000 0000 0000 0240 0000	Hardware in Status RESET
4099	MR1 0000 0000 0000 0000 0000 0240 0001	Hardware in Status STARTUP1
4099	MR1 0000 0000 0000 0000 0000 0240 0002	Hardware in Status STARTUP2
4099	MR1 0000 0000 0000 0000 0000 0240 0003	Hardware in Status READY
4099	MR1 0000 0000 0000 0000 0000 0240 0004	Hardware in Status IDLE
4099	MR1 0000 0000 0000 0000 0000 0240 0005	Hardware in Status PARAM
4099	MR1 0000 0000 0000 0000 0000 0240 0006	Hardware in Status CALIB
4099	MR1 0000 0000 0000 0000 0000 0240 0007	Hardware in Status WAKEUP
4099	MR1 0000 0000 0000 0000 0000 0240 0008	Hardware in Status SAMPLE
4099	MR1 0000 0000 0000 0000 0000 0240 0009	Hardware in Status DPWN
4099	MR1 0000 0000 0000 0000 0000 0240 000f	Hardware in Status CRASH
0003	SWM	Willkommen
0003	SPD	Starte Parametrierung
0003	EOP	Parametrierung beendet

Tabelle 3.3: Antworten auf Statusanfragen bzw. -wechsel

In Tabelle 3.3 ist außerdem zu sehen, dass die in der Antwort enthaltene Information im Fall einer Statusmeldung zwar bereits nach 31 Stellen endet, aber dennoch alle 4099 Stellen aus dem Hardwarebuffer ausgelesen werden müssen, um den Buffer für die nächste Antwort zu leeren. Die Datenquelle gibt an, welcher Controller im Inneren der Hardware die Antwort generiert hat. Theoretisch ist es auch möglich, Informationen von „MR2“ zu erhalten, jedoch ist dieser Controller im normalen Betrieb nur für das Senden der Messdaten auf Port 2001 zuständig. Außerdem handelt es sich bei „SWM“, „SDP“ und „EOP“ nicht um echte Datenquellen, sondern vielmehr um Ausnahmefälle, bei denen, der für die Quelle vorgesehene Bereich genutzt wird, ohne dass darauf weitere Nutzdaten folgen.

### 3.5 Messdaten

Die Messdaten werden im Status Calib oder Sample auf Port 2001 bereit gestellt. Dabei ist zu beachten, dass die Form der Daten vom Status ist. In beiden Fällen beginnen die Daten mit einem 77 Bytes langen Header, in dem verschiedene Informationen, wie beispielsweise die Datenlänge, enthalten sind, von denen für die neue Software jedoch nur die Länge der Messdaten relevant ist. Die Messdaten werden in beiden Fällen hexadezimal kodiert gesendet, wobei jeweils 4 Byte (32 Bit) ein Paket bilden. Diese Pakete sind wiederum, wie in den Tabellen 3.4 und 3.5 zu sehen, in 10 Bit große Teilpakete untergliedert.

Byte 4	Byte 3	Byte 2	Byte 1
0000 0000	0000 0000	0000 0000	0000 0000

Tabelle 3.4: Messdaten-Format

Rest	Wert 3	Wert 2	Wert 1
00	00 0000 0000	0000 0000 00	00 0000 0000

Tabelle 3.5: Messdaten-Format, Darstellung der übertragenen ADC-Werte

Der Unterschied zwischen den Zuständen Sample und Calib besteht darin, dass bei Sample nur die Gesamtwerte gesendet werden. Demnach sind alle drei in Tabelle 3.5 dargestellten Werte Summenwerte der beiden Komponenten, wohingegen bei Calib die Summenwerte, die Werte der 0°-Messung und die Werte der 90°-Messung einzeln gesendet werden.

	Wert 3	Wert 2	Wert 1
Sample	3. Summenwert	2. Summenwert	1. Summenwert
Calib	1. Summenwert	1. 90°-Komponente	1. 0°-Komponente

Tabelle 3.6: Werte-Format

Das bedeutet bezogen auf Tabelle 3.6, um die gleiche gemessene Strecke zu erfassen, werden in Calib drei mal so viele Werte geliefert wie in Sample, weshalb Sample im normalen Messbetrieb performanter ist und bei gleicher zurückgelieferter Datenmenge eine längere Strecke abbilden kann.

## 4 Software

Ziel ist es eine Software zu entwickeln, die sowohl mit den bisher verfügbaren Geräten kompatibel ist, als auch in Zukunft für neue Hardware-Versionen erweiterbar bleibt. Dazu wurden die Entwicklung der Hardwareanbindung und die Entwicklung der graphischen Benutzeroberfläche strikt getrennt, da die Benutzeroberfläche für alle Hardware-Versionen identisch sein soll und nach Möglichkeit nicht mehr angepasst wird. Der Ansatz, um dies gewährleisten zu können, ist eine Handling-Klasse zu entwerfen, die alle Hardwaretypen verwalten kann, wobei deren Schnittstelle zur Nutzeroberfläche unverändert bleibt und nur die verwendete Hardware angegeben werden muss. Damit diese Klasse auf alle kommenden Hardwaretypen erweiterbar bleibt, müssen alle Hardware-Klassen von einer gemeinsamen, abstrakten Basisklasse abgeleitet werden. Die hierzu angelegten Klassen sind in der Bibliothek „EmusHardWare“ zusammengefasst und in Tabelle 4.1 abgebildet.

Klasse	Funktion
EmusHandling	Bildet die Schnittstelle zur Nutzeroberfläche und hält einen Zeiger von EmusBaseClass als Member
EmusBaseClass	Die abstrakte Basisklasse für alle Emus-Anwendungen und hält alle wichtigen Parameter und virtuelle Funktionen
EmusFrontEnd	Die Hardwareklasse für die hier verwendete Hardware ("LimaTest"). Ermöglicht die konkrete Implementierung aller Funktionen der Basisklasse

Tabelle 4.1: Klassen-Tabelle

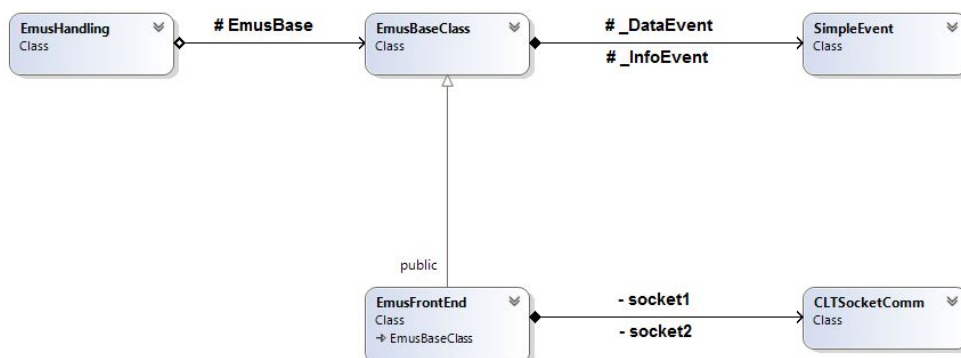


Abbildung 4.1: Klassendiagramm

## 4.1 EmusBaseClass

Die Klasse EmusBaseClass stellt die Basisklasse für Hardwaretypen dar und hält daher alle wichtigen Parameter und Funktionen, die für die Arbeit mit elektromagnetischem Ultraschall immer benötigt werden. Die Auswahl der benötigten Parameter basiert auf der Auswertung der bisherigen Software und internen Dokumenten des Instituts [Weib] [Weia]. Dabei sind im Vergleich zur Vorgängerversion bereits einige nicht mehr benötigte Parameter entfallen, so z.B. alle Parameter die mit der Steuerung des Motors zu tun haben. Außerdem hält die Basisklasse get- und set-Funktionen für alle Parameter, welche gegebenenfalls die Eingaben auf die Einhaltung entsprechenden Grenzwerte überprüfen.

```

1 virtual int startDataAcquirement()=0;
2 virtual int stopDataAcquirement() = 0;
3 virtual int startCalibration() = 0;
4 virtual int stopCalibration() = 0;
5 virtual int sendParamToHardWare() = 0;
6 virtual int connectToHardWare() = 0;
7
8 virtual bool getAScanSize(int &Size) = 0;
9 virtual bool getLatestAScan(int AScan[], int &maxValue, int &AScanSize) = 0;
10 virtual bool getLatestAScan0(int AScan[], int &maxValue, int &AScanSize) = 0;
11 virtual bool getLatestAScan90(int AScan[], int &maxValue, int &AScanSize) = 0;
12
13 virtual bool setInfoEventFunction(void(*func)()) = 0;
14 virtual bool setDataEventFunction(void(*func)()) = 0;
15 virtual bool waitForSimpleInfoEvent(DWORD timeOut) = 0;
16 virtual bool waitForSimpleDataEvent(DWORD timeOut) = 0;
17
18 virtual int paramRoundOff() = 0;
19 int getSimpleErrorCode();

```

Listing 4.1: EmusBaseClass Methoden

Wie in Listing 4.1 zu sehen, sind abgesehen von Gettern und Settern, die meisten Methoden der Basisklasse rein virtuelle bzw. abstrakte Methoden, was bedeutet, dass diese Methoden in einer Klasse, die von der Basisklasse abgeleitet ist, implementiert werden müssen. So ist gewährleistet, dass später alle weiteren Hardwareklassen über die Klasse EmusHandling, welche die Methoden der Basisklasse aufruft, verwaltet werden können. Außerdem sind die in Abbildung 4.1 dargestellten Methoden diejenigen, die später zur Steuerung der Hardware benötigt werden.

```

1 #define AVERAGE_MAX 200
2 #define AVERAGE_MIN 1

```

Listing 4.2: EmusBaseClass Grenzwerte

Die Grenzwerte für die Parameter sind, wie in Listing 4.2 beispielhaft dargestellt, über Präprozessordefinitionen angelegt und nicht als konstante Member der Klasse, da somit kein Datentyp für die Grenzwerte festgelegt werden muss.

Des Weiteren hält die Basisklasse zwei Instanzen der Klasse SimpleEvent, um z.B. der Nutzeroberfläche mitteilen zu können, wenn neue Messdaten bereit stehen oder der Status der Hardware sich geändert hat. Durch die Wahl dieser Variante kann „Polling“, also ständiges Abfragen von Werten, in der Nutzeroberfläche vermieden werden.

```
1 SimpleEvent _DataEvent;
2 SimpleEvent _InfoEvent;
```

Listing 4.3: SimpleEvent member

## 4.2 SimpleEvent

In Listing 4.4 ist ein Ausschnitt aus der Header-Datei von SimpleEvent zu sehen. Der Sinn der Klasse ist es, auf ein Ereignis reagieren zu können ohne zuvor abfragen zu müssen, ob dieses Ereignis eingetreten ist. Die Klasse bietet grundsätzlich zwei verschiedene Möglichkeiten. Entweder wird eine Funktion übergeben, die ausgeführt wird, wenn sie an anderer Stelle über „throwEvent“ aufgerufen wird oder es besteht die Möglichkeit, einen Thread über „waitForSimpleEvent“ auf ein Ereignis warten zu lassen. Letzteres bietet sich z.B. zur Live-Darstellung von Messdaten in der Nutzeroberfläche an da diese ohnehin in einem eigenen Thread laufen muss, welcher auf neue Messdaten wartet.

```
1 class SimpleEvent
2 {
3 private:
4     HANDLE _SimpleEventHandShake;
5
6     void (*_EventFunc)();
7     bool _Active;
8
9 public:
10    SimpleEvent();
11    ~SimpleEvent();
12
13 #pragma region handle
14    //timeOut in ms
15    bool waitForSimpleEvent(DWORD timeOut);
16    bool SimpleEventRdy();
17    bool SimpleEventBsy();
18 #pragma endregion handle
19
20 #pragma region function
21    bool setEventFunc(void (*EventFunc)());
22    bool throwEvent();
23    bool setActive(bool active);
24    bool getActive();
25 #pragma endregion function
26
27 };
```

Listing 4.4: SimpleEvent Header

## 4.3 EmusHandling

In Listing 4.5 ist die Deklaration der Klasse `EmusHandling` zu sehen. Diese beinhaltet, abgesehen vom Pointer auf die Basisklasse, lediglich zwei weitere Methoden. Zum einen die Methode „`setHardWare`“, welche nötig ist, um die verwendete Hardwareschnittstelle festzulegen und zum anderen „`getHardWare`“, um Zugriff auf die Methoden der Basisklasse und der, von der Basisklasse abgeleiteten, ausgewählten Hardwareklasse zu gewährleisten. Der Pointer auf die Basisklasse ist als „`protected`“ angelegt, um ungewollte Änderungen zu vermeiden.

```

1 class EmusHandling
2 {
3     protected:
4         EmusBaseClass *EmusBase;
5
6     public:
7
8         EmusHandling();
9         ~EmusHandling();
10
11         //sets hardware-Class
12         int setHardWare(EmusBaseClass *ptr);
13
14         //pointer on EmusBaseClass to access the base-functions
15         EmusBaseClass* getHardWare();
16
17 };

```

Listing 4.5: `EmusHandling` Header

Die Definition der Methoden sieht wie in Listing 4.6 dargestellt aus. In `setHardware` wird ein Pointer auf die entsprechende Hardwareklasse übergeben, welche von `EmusBaseClass` abgeleitet sein muss. Falls ein `NULL`-Pointer übergeben wurde, kehrt die Funktion mit dem entsprechenden Code zurück, falls der übergebene Zeiger gültig ist, wird er explizit auf den Typ „`EmusbaseClass*`“ konvertiert und innerhalb der Klasse als Implementierung der Hardwareklasse verwendet. Auf diese Weise kann jede von der Basisklasse geerbte Methode aufgerufen werden. Diese vererbten Methoden werden nur in den abgeleiteten Klassen definiert und dort je nach Bedarf auf die entsprechende Hardware angepasst und können dann von der Handling-Klasse über den Pointer auf die Basisklasse aufgerufen werden. Durch die abstrakte Einbindung stellt die Basisklasse also lediglich ein Interface zur Verfügung und verbirgt die eigentliche Implementierung vor übergeordneten Klassen. Auf diese Weise bleibt die Klasse „`EmusHandling`“ mit allen zukünftigen Hardwareschnittstellen kompatibel. Die Methode „`getHardWare`“ gewährt Zugriff auf den Pointer der Basisklasse, da dieser wie zuvor gesehen als „`protected`“ definiert wurde.

```

1
2 int EmusHandling::setHardWare(EmusBaseClass* ptr)
3 {
4     if (ptr == NULL)
5         return -1101;
6     EmusBase = ptr;
7 }
8
9 EmusBaseClass* EmusHandling::getHardWare()
10 {
11     return EmusBase;
12 }

```

Listing 4.6: EmusHandling.cpp

Die Übergabe einer Hardwareklasse an die Handling-Klasse erfolgt wie in Listing 4.7 beispielhaft dargestellt. Hier ist zu sehen, wie in Zeile drei über die Methode „setHardWare“ eine Referenz von „EmusFrontEnd frontEnd“ an „EmusHandling handling“ übergeben wird. Weiterhin ist zu sehen, wie über „getHardWare“ zunächst ein Setter, welcher in der Basisklasse definiert und deklariert ist, aufgerufen werden kann und in Zeile fünf, wie eine der vererbten Methoden der Basisklasse aufgerufen wird, welche erst in der abgeleiteten Klasse definiert wird.

```

1     EmusFrontEnd frontEnd;
2     EmusHandling handling;
3     handling.setHardWare(&frontEnd);
4     handling.getHardWare()->setTaktAnz(2);
5     handling.getHardWare()->connectToHardWare();

```

Listing 4.7: EmusHandling Beispielaufruf

## 4.4 EmusFrontEnd

Die Klasse EmusFrontEnd ist die von EmusBaseClass abgeleitete Hardwareklasse für die verwendete Elektronik und hält alle zur Steuerung der Hardware benötigten Funktionen. Von EmusFrontEnd muss lediglich eine Instanz erzeugt werden, welche dann an EmusHandling übergeben wird, um so die verwendete Hardware festzulegen. Die Hauptaufgaben bestehen darin, den Status der Hardware zu setzen, sobald diese die Kontrolle an den PC abgegeben hat, die Antworten auf Port 2000 abzufragen bzw. auszuwerten und die Messdaten abzunehmen, in die entsprechende Form zu konvertieren und für die Abnahme durch die Nutzeroberfläche bereit zu stellen. Da das Auslesen der Ports parallel zum normalen Betrieb möglich sein soll, müssen die entsprechenden Funktionen jeweils in einem eigenen Thread laufen. Alle Funktionalitäten, auf die von außen zugegriffen werden muss, werden in den von der Basisklasse geerbten Methoden vereint, welche dann vom Handler aufgerufen werden können und sind unter „public: #pragma region base\_methods“ zusammengefasst. Alle Methoden die nur innerhalb von EmusFrontEnd benötigt werden, befinden sich unter „private: #pragma region intern\_methods“. Methoden, die zwar als public deklariert sind, aber

nicht unter „base\_methods“ fallen, werden lediglich zu Debug-Zwecken benötigt und können, da sie nicht von der Basisklasse geerbt wurden, auch nicht über den Handler aufgerufen werden. Außerdem hält EmusFrontEnd zwei Instanzen der Klasse „CLTSocketComm“, um die Verbindung zu den beiden Ports 2000 und 2001 herstellen zu können. Bei der Klasse „CLTSocketComm“ handelt es sich um eine Klasse zum Aufbau einer Verbindung mit einer bestimmten IP-Adresse über einen bestimmten Port, welche schon im Institut vorhanden war.

### 4.4.1 XML-File

XML steht für „Extensible Markup Language“ und ist eine Auszeichnungssprache, die oft für die Kommunikation zwischen Systemen verwendet wird. Die Besonderheit ist hierbei, dass sämtliche Informationen durch Tags gekapselt sind, nach denen beim Auslesen durch ein anderes System unabhängig von ihrem Inhalt gesucht werden kann. Da die Gestaltung der aus der alten Software vorhandenen XML-Dateien so wenig wie möglich verändert werden soll, um auch alte Parametersätze verwenden zu können, wurde das Erzeugen der Datei bis auf wenige Anpassungen aus der alten Software übernommen. Die endgültige Form der Parameterdatei hängt hierbei immer von der Parametrierung selbst ab, da z.B. die Zahl der Takte entscheidend dafür ist, wie viele Taktparameter übergeben werden müssen. Der grundsätzliche Aufbau der Datei bleibt jedoch gleich und ist in Abbildung 4.2 schematisch dargestellt.

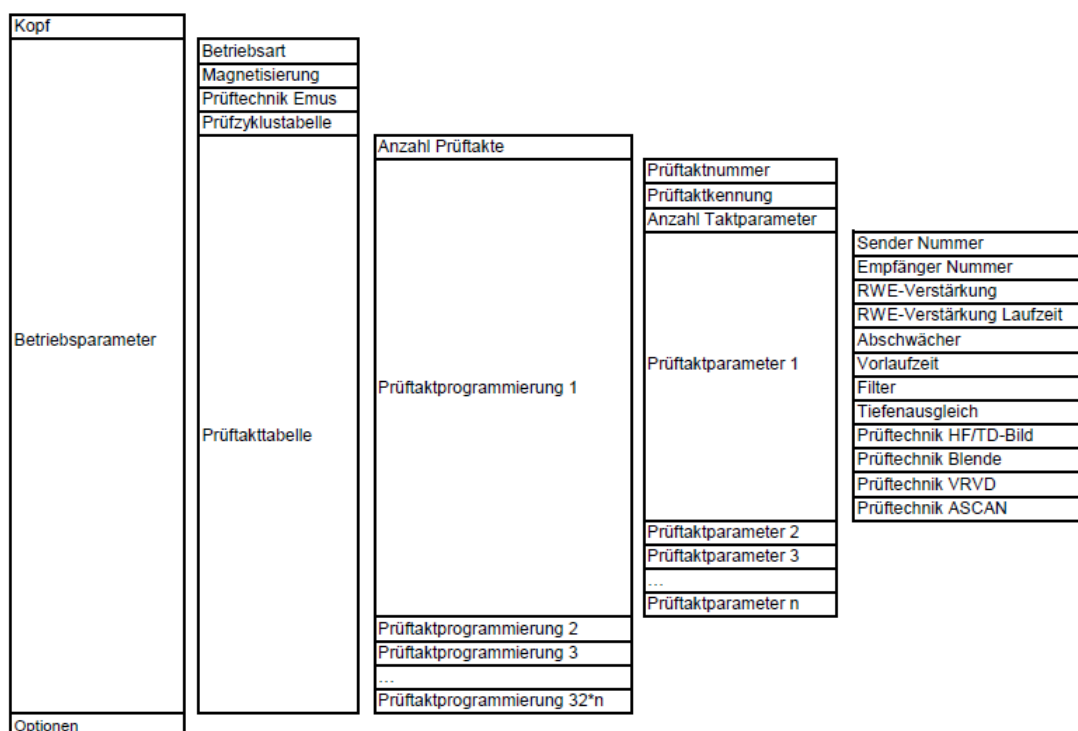


Abbildung 4.2: Aufbau der XML-Datei

### 4.4.2 FTP-Verbindung

FTP steht für „File Transfer Protocol“ und ist ein Netzwerkprotokoll, das für gewöhnlich zur Übertragung von Dateien verwendet wird. Es wird in diesem Fall für die Übertragung einer XML-Datei vom Client zum Server verwendet, wobei die Software auf PC-Seite der Client ist und die Elektronik der Server. Die Übertragung der XML-Datei erfolgt wie in Listing 4.8 beispielhaft dargestellt. Dabei muss lediglich der Zielpfad auf die entsprechende Hardwaregeneration angepasst werden.

```

1
2 if ((hFtp = InternetOpen(_T(""), INTERNET_OPEN_TYPE_DIRECT, NULL, NULL, NULL))
   == NULL)
3     return false;
4
5 if ((hFtp = InternetConnect(hFtp, _T("192.168.0.130"),
   INTERNET_DEFAULT_FTP_PORT, _T("test"), _T("test"), INTERNET_SERVICE_FTP,
   0, NULL)) == NULL)
6     return false;
7
8 bool success = FtpPutFile(hFtp, _T("../\\send.xml"), _T("send.xml"),
   FTP_TRANSFER_TYPE_BINARY, NULL);
9 if (!success)
10 {
11     err = GetLastError();
12 }
13
14 int returned = InternetGetLastResponseInfo(&dwInetError, szExtErrMsg, &
   dwExtLength);
15 return true;

```

Listing 4.8: sendParamFile

### 4.4.3 Hardware-Verbindung

Die Verbindung zur Hardware wird, wie in Listing 4.9 zu sehen, hergestellt. Es ist auch zu erkennen, dass die Verbindung auf zwei verschiedenen IP-Adressen aufgebaut werden kann. Dabei handelt es sich um die beiden verschiedenen Generationen der verwendeten Hardware. Es wird immer zuerst versucht eine Verbindung mit der Adresse „192.168.0.130“ aufzubauen und falls diese fehlschlägt wird versucht, eine Verbindung mit „192.168.0.126“ herzustellen. Die verwendete Reihenfolge ist der Tatsache geschuldet, dass die „SSV“-Version nur noch sehr selten in Betrieb ist und die Verbindung zu dieser eher im Ausnahmefall benötigt wird. Wenn beide Verbindungen fehlschlagen wird der entsprechende Fehlercode gesetzt und die Funktion gibt „false“ zurück. Es ist außerdem zu erkennen, dass stets beide Sockets verbunden werden müssen, um mit der Hardware kommunizieren zu können, da diese darauf wartet, dass beide Ports verbunden sind.

```

1 bool EmusFrontEnd::setServerConnection()
2 {
3     //PROST connect
4     if (socket1.SetIP("192.168.0.130") && socket2.SetIP("192.168.0.130"))
5         if (socket1.SetPortNumber(2000) && socket2.SetPortNumber(2001))
6             if (socket1.OpenConnection() && socket2.OpenConnection())
7                 {
8                     _SimpleErrorCode = 0;
9                     _Prost = true;
10                    return true;
11                }
12    //SSV connect
13    if (socket1.SetIP("192.168.0.126") && socket2.SetIP("192.168.0.126"))
14        if (socket1.SetPortNumber(2000) && socket2.SetPortNumber(2001))
15            if (socket1.OpenConnection() && socket2.OpenConnection())
16                {
17                    _SimpleErrorCode = 0;
18                    _Prost = false;
19                    return true;
20                }
21    //no connection
22    _SimpleErrorCode = -2201;
23    return false;
24 }

```

Listing 4.9: setServerConnection

#### 4.4.4 Threading

Da die Nutzeroberfläche in einer „managed“ (.Net) Umgebung entwickelt wird, ist es nicht möglich, die Standard-Thread-Bibliothek `<thread.h>` zu nutzen, weshalb sich die Deklaration eines Threads als Member einer Klasse als deutlich komplexer darstellt.

Der Header, der sich am ehesten für diesen Fall eignet ist `<process.h>`. Dieser stellt z.B. die Funktion „`_beginthread`“ zur Verfügung, welche mit unterschiedlichen Übergabeparametern beladen werden kann und in der Managed-Welt verfügbar ist [mdna].

Diese Funktion kann jedoch nicht einfach von einer Member-Methode aus mit einer weiteren Methode der Klasse beladen werden. Dazu muss die Methode, die aufgerufen werden soll, wie in Listing 4.10 zu sehen, über einen Wrapper aufgerufen werden. Die Funktion `_beginthread` wird wiederum mit dem im Header deklarierten Wrapper beladen, welcher dann z.B. Zugriff auf `catchAnswer()` hat [mdnb].

```

1 //for safe start and stop
2 HANDLE _DataThread;
3 //needed to call _beginthread of member function receiveData()
4 static void __cdecl receiveDataThreadWrapper(void* o)
5 {
6     static_cast<EmusFrontEnd*>(o)->receiveData();
7 }
8 //for communication between waitForHardWare and AnswerThread

```

```

9 HANDLE _keeper;
10 //for safe start and stop
11 HANDLE _AnswerThread;
12 //needed to call _beginthread of member function catchAnswer()
13 static void __cdecl catchAnswerThreadWrapper(void* o)
14 {
15     static_cast<EmusFrontEnd*>(o)->catchAnswer();
16 }

```

Listing 4.10: threading Header

```

1 //starts catchAnswer() as thread
2 _beginthread(&EmusFrontEnd::catchAnswerThreadWrapper,0,
3             static_cast<void*>(this));
4 //starts receiveData() as thread
5 _beginthread(&EmusFrontEnd::receiveDataThreadWrapper,0,
6             static_cast<void*>(this));

```

Listing 4.11: beginthread

Über die Funktion „WaitForSingleObject( HANDLE hHandle, DWORD dwMilliseconds )“ kann ein Thread z.B. so lange zum Warten gezwungen werden, bis das „HANDLE“ frei wird und an den Thread übergeht. Über den Parameter „dwMilliseconds“ kann ein Timeout in Millisekunden angegeben werden. Falls dieser ausgelöst wird, kehrt die Funktion mit „0x00000102L“ zurück.

Um das „HANDLE“ wieder freizugeben wird die Funktion „ReleaseMutex(HANDLE hHandle)“ aufgerufen.

Ein solcher Ablauf ist in Listing 4.12 am Beispiel von SimpleEvent::waitForSimpleEvent dargestellt[mdnc].

```

1 bool SimpleEvent::waitForSimpleEvent(DWORD timeOut)
2 {
3     DWORD tout = 0;
4     if (timeOut == NULL)
5         timeOut = INFINITE;
6     tout=WaitForSingleObject(_SimpleEventHandShake, timeOut);
7     ReleaseMutex(_SimpleEventHandShake);
8     if (tout == 0x00000102L)
9         return false;
10    return true;
11 }

```

Listing 4.12: waitForSimpleEvent

#### 4.4.5 AnswerThread

In Listing 4.13 ist die Methode „catchAnswer“ zu sehen, welche nach der erfolgreichen Verbindung mit der Hardware in einem eigenen Thread läuft und kontinuierlich den Port 2000 abfragt.

```

1 void EmusFrontEnd::catchAnswer()
2 {
3     WaitForSingleObject(_AnswerThread, INFINITE);
4
5     char sBuf[5];
6     int infoSize = 0;
7     // _running ends the thread
8     while (_running)
9     {
10        WaitForSingleObject(_keeper, INFINITE);
11        int receivedSize = 0;
12
13        do
14        {
15            memset(sBuf, '0', 5);
16            memset(_AnswerBuffer, '0', sizeof(_AnswerBuffer));
17
18            if (socket1.ReceiveData(sBuf, 5))
19            {
20                infoSize = atoi(sBuf);
21                if (infoSize > 0)
22                {
23                    //uses ReceiveData until receivedSize == infoSize
24                    do
25                    {
26                        Sleep(300);
27                        receivedSize = receivedSize + socket1.ReceiveData
28                            (_AnswerBuffer + receivedSize, (infoSize - receivedSize));
29
30                    } while (receivedSize != infoSize);
31                    break;
32                }
33            }
34        } while (_running);
35
36        readAnswer();
37        ReleaseMutex(_keeper);
38    }
39    ReleaseMutex(_AnswerThread);
40 };

```

Listing 4.13: AnswerThread

Sobald die Funktion einmal aufgerufen wird, läuft „catchAnswer“ in einer while-Schleife, bis der boolesche Bezeichner „\_running“ false wird. Beim Start des Programms wird „\_running“ mit true initialisiert und erst wieder im Destruktor auf false gesetzt. Dies bedeutet, dass die Schleifenbedingung, wenn die Schleife einmal gestartet ist, bis zum Ende des Programms erfüllt ist und immer wiederholt wird.

Am Anfang dieser Schleife kann durch das „HANDLE \_keeper“ threadsicher mit anderen Threads kommuniziert werden, um z.B. mitzuteilen, wenn eine vollständige Antwort empfan-

gen wurde. Innerhalb der zweiten Schleife, welche in Zeile 15 startet, werden immer wieder fünf Bytes ausgelesen, da sich wie unter Kapitel 3.4 beschrieben in den ersten fünf Bytes die Länge des nachfolgenden Informationssatzes befindet. Wenn hier keine Längenangabe empfangen wurde, wird der Vorgang wiederholt. Die letzte Schleife, welche in Zeile 26 beginnt, setzt Stück für Stück die Antwort zusammen, bis sie die zuvor empfangene Länge erreicht hat. Durch dieses Verfahren ist sichergestellt, dass, unabhängig von Geschwindigkeitsunterschieden von PC und Elektronik, nie mehr oder weniger als eine gesendete Antwort aus dem Hardware-Puffer ausgelesen wird. Dies ist unbedingt notwendig, da sonst alle nachfolgenden Antworten fehlerhaft wären. Die Auswertung, welche Antwort überhaupt empfangen wurde, geschieht in einer anderen Methode namens „readAnswer“, welche am Ende aufgerufen wird.

#### 4.4.6 Antwort-Auswertung

Wenn, wie in Kapitel 4.4.7 beschrieben, eine vollständige Antwort auf Port 2000 Empfangen wurde, wird diese wie in Listing 4.14 bereichsweise ausgewertet.

```

1 void EmusFrontEnd::readAnswer()
2 {
3     _InfoEvent.SimpleEventBsy();
4     char typeBuffer[3];
5     memcpy(typeBuffer, _AnswerBuffer, 3);
6     if (strstr(typeBuffer, WELCOME))
7     {
8         _currentSource = "SWM";
9     }
10    .
11    .
12    else if (strstr(typeBuffer, COMMAND_DIL))
13    {
14        _currentSource = "MR1";
15
16        char messageType[3];
17        int typeNumber = 0;
18        memcpy_s(messageType, 3, _AnswerBuffer + 23, 3);
19        typeNumber = atoi(messageType);
20        switch (typeNumber)
21        {
22            case 24:
23                _currentType = _STATE;
24                readState();
25                break;
26            case 34:
27                _currentType = _HARD_REVISION;
28                readHardwareRevision();
29                break;
30            case 54:
31                _currentType = _Error;
32                break;
33            default:

```

```

34     _currentType = _NOTYPE;
35     }
36
37 }
38 _InfoEvent.SimpleEventRdy();
39 _InfoEvent.throwEvent();
40 }

```

Listing 4.14: readAnswer

In den ersten drei Stellen des Antwortpuffers befindet sich eine Information zur Art der Antwort, im Fall einer Statusmeldung ist dies „MR1“. Wenn beispielsweise die Statusmeldung für „IDLE“ empfangen wurde, wird zunächst „MR1“ gefunden und dann der Typ der Antwort überprüft, welcher ab Zeichen 23 beginnt. Für „IDLE“ ist dort, wie auch unter Kapitel 3.4 zu sehen, „024“ eingetragen, da es sich um eine Statusmeldung handelt. Im entsprechenden „case“ wird wiederum die Funktion „readState“ aufgerufen, welche in Listing 4.15 ausschnittsweise zu sehen ist.

```

1 void EmusFrontEnd::readState()
2 {
3     char state = '0';
4     memcpy_s(&state, 1, _AnswerBuffer + 30, 1);
5     HWStates = (stateMachineTest)(atoi(&state));
6
7     switch (HWStates)
8     {
9     case Reset:
10         _currentState = _RESET;
11         break;
12     .
13     .
14     .
15     case Down:
16         _currentState = _DOWN;
17         break;
18     default:
19         _currentState = _NOSTATE;
20
21     }
22 }

```

Listing 4.15: readState

#### 4.4.7 DataThread

Wie in Listing 4.16 zu sehen, ist der grundlegende Aufbau des Daten-Threads der Selbe, wie zuvor unter Kapitel 4.4.7 für den Antwort-Thread beschrieben. Hierbei werden die Rohdaten jedoch lokal in einen 26000 Bytes langen Puffer geschrieben, welcher dann an „convertData(

char Data[], int DataSize)“ übergeben wird.

```

1 void EmusFrontEnd::receiveData()
2 {
3     char buffer[26000];
4     int bufferSize = 0;
5     memset(buffer, 0, sizeof(buffer));
6
7     char sBuf[5];
8     int infoSize = 0;
9
10    //running ends the thread-loop
11    while (_running)
12    {
13        WaitForSingleObject(_DataThread, INFINITE);
14        int receivedSize = 0;
15        do
16        {
17            memset(sBuf, '0', 5);
18            memset(_AnswerBuffer, '0', sizeof(_AnswerBuffer));
19
20            if (socket2.ReceiveData(sBuf, 5))
21            {
22                infoSize = atoi(sBuf);
23                if (infoSize > 0)
24                {
25                    //uses ReceiveData until receivedSize == infoSize
26                    do
27                    {
28                        Sleep(1);
29                        receivedSize = receivedSize + socket2.ReceiveData
30                            (buffer + receivedSize, (infoSize - receivedSize));
31                    } while (receivedSize != infoSize);
32                    break;
33                }
34            }
35        } while (_running);
36
37        convertData(buffer, receivedSize);
38        memset(buffer, 0, sizeof(buffer));
39        ReleaseMutex(_DataThread);
40    }
41 }

```

Listing 4.16: DataThread

#### 4.4.8 Daten-Auswertung

Die Konvertierung der Messdaten erfolgt nach dem in Kapitel 3.5 beschriebenen Aufbau. Zunächst wird an Stelle 28 die Menge der Pakete ausgelesen und die Menge der erwarteten

Messdaten entsprechend angepasst. Dann wird, wie in Listing 4.17 zu sehen, der gesamte Informationssatz ab Stelle 72 (77: Headerende, -5: Datensatzlänge) in acht Zeichen lange Pakete unterteilt, welche die verschachtelten Hex-Werte beinhalten.

```

1  int numberOfPackages = 0;
2  //reads number of received packages
3  memcpy(BuffConv, &charBuffer[33-5], 4);
4  sscanf_s(BuffConv, "%x", &numberOfPackages);
5  unsigned int *DataBuffer = new unsigned int [numberOfPackages];
6  int *AScanValues1 = new int [numberOfPackages];
7  int *AScanValues2 = new int [numberOfPackages];
8  int *AScanValues3 = new int [numberOfPackages];
9
10 //puts packages from charBuffer into DataBuffer
11 for (int loop = 0; loop < numberOfPackages; loop++)
12 {
13     memcpy(BuffConv, "0", 10);
14     memcpy(BuffConv, &charBuffer[77-5 + loop * 8], 8);
15     BuffConv[8] = 0;
16     if (HexControl(BuffConv) == false) break;
17     sscanf_s(BuffConv, "%x", &DataBuffer[loop]);
18 }

```

Listing 4.17: convertdata 1

Die Funktion „HexControl“ war bereits Teil der bisherigen Software und stellt fest, ob alle Symbole in einem Buffer zulässige Zeichen des Hexadezimal-Codes sind.

Als nächstes werden die Pakete, wie in Listing 4.18 zu sehen, in tatsächliche Messdaten zerlegt. Dies geschieht, indem ein Paket zunächst mit 0x03FF (11 1111 1111b, 1023d) maskiert wird, wodurch nur die rechten 10 Bit übrig bleiben. Als nächstes wird überprüft, ob der verbleibende Teil größer oder gleich 512 (10 0000 0000b) ist. In diesem Fall wird dieser mit 0xFC00 (1111 1100 0000 0000b, 64512d) bool'sch verodert. Der Grund für diesen Ablauf ist, dass es sich bei dem 10. Bit um das Vorzeichenbit handelt, was bedeutet, dass bei einem Wert größer als 511 (01 1111 1111b) das Vorzeichenbit gesetzt sein muss und der Wert somit negativ sein sollte. Daher wird durch die bool'sche Veroderung mit 0xFC00 das Zweierkomplement gebildet, wodurch der Wertebereich um den Wert 512 ins Negative gespiegelt wird. Jeder der drei enthaltenen Messwerte wird mit der Nummer des Paketes, aus dem er stammt, zwischengespeichert. Um die beiden anderen Werte zu erhalten, wird das jeweilige Paket um 10 Bit nach rechts verschoben und der Maskier-Vorgang wiederholt. Nach dem Abschluss dieses Vorgangs ist der Inhalt jedes Paketes auf die Puffer „AScanValues1“ bis „AScanValues3“ aufgeteilt.

```

1  //extracts actual data from packages
2  short Zshort;
3  for (int loop_t = 0; loop_t < numberOfPackages; loop_t++)
4  {
5      Zshort = DataBuffer[loop_t] & 0x03ff;
6      if (Zshort >= 512) Zshort |= 0xfc00;
7      AScanValues1[loop_t] = Zshort;

```

```

8
9     Zshort = (DataBuffer[loop_t] >> 10) & 0x03ff;
10    if (Zshort >= 512)    Zshort |= 0xfc00;
11    AScanValues2[loop_t] = Zshort;
12
13    Zshort = (DataBuffer[loop_t] >> 20) & 0x03ff;
14    if (Zshort >= 512)    Zshort |= 0xfc00;
15    AScanValues3[loop_t] = Zshort;
16 }

```

Listing 4.18: convertdata 2

Als letztes müssen die Werte in der richtigen Reihenfolge auf Ausgabe-Arrays verteilt werden. Die Art dieser Verteilung hängt, wie bereits in Kapitel 3.5 beschrieben, davon ab, ob die Daten in Calib oder Sample aufgenommen wurden. In Sample sind alle drei Werte eines Pakets Teil des Summen-Arrays, wodurch die Anzahl der aufgenommenen Samplepunkte dreimal so groß ist, wie Menge der empfangenen Pakete. In Calib hingegen werden die Werte auf die drei Ausgabe-Arrays verteilt, wodurch jedes der Ausgabe-Array exakt so viele Werte enthält, wie Pakete übertragen wurden.

```

1 //sets data in correct order
2 if (_Measurement)
3 {
4     _latestAScanSize = numberOfPackages *3;           //sample
5     for (int i = 0; i < numberOfPackages; i++)
6     {
7         _latestAScan[i*3] = AScanValues1[i];
8         _latestAScan[i*3+1] = AScanValues2[i];
9         _latestAScan[i*3+2] = AScanValues3[i];
10    }
11    memset(_latestAScan0, 0, numberOfPackages);
12    memset(_latestAScan90, 0, numberOfPackages);
13 }
14 else
15 {
16     _latestAScanSize = numberOfPackages;           //calib
17     for (int i = 0; i < _latestAScanSize; i++)
18     {
19         _latestAScan0[i] = AScanValues1[i];
20         _latestAScan90[i] = AScanValues2[i];
21         _latestAScan[i] = AScanValues3[i];
22     }
23 }

```

Listing 4.19: convertdata 3

#### 4.4.9 Timing

Da das Programm auf dem PC deutlich schneller abgearbeitet wird, als die Elektronik die Befehle ausführen kann, ist es an einigen Stellen notwendig, auf die Antwort der Hardware zu

warten. Zu diesem Zweck wurde die in Listing 4.20 zu sehende Methode „WaitForHardware“ entwickelt.

```

1 bool EmusFrontEnd::waitForHardware(std::string source, std::string state)
2 {
3     DWORD HardWareTimeOut = 15000; //ms
4     DWORD HardWareTimeOutValue = 0;
5     unsigned int FrontEndTimeOut = 30000;
6     unsigned int start = clock();
7     unsigned int timeDiff = 0;
8
9     while (true)
10    {
11        if (source == _currentSource && state == _currentState )
12            break;
13        HardWareTimeOutValue = WaitForSingleObject(_keeper, HardWareTimeOut);
14        ReleaseMutex(_keeper);
15
16        if (HardWareTimeOutValue == _HandleTimeOutValue)
17        {
18            _SimpleErrorCode = -2203;
19            return false;
20        }
21        timeDiff = clock() - start;
22        if (timeDiff > FrontEndTimeOut)
23        {
24            _SimpleErrorCode = -2202;
25            return false;
26        }
27    }
28    _SimpleErrorCode = 0;
29    return true;
30 }

```

Listing 4.20: WaitForHardware

Die Methode kann, wie in Listing 4.21 gezeigt, aufgerufen werden. In diesem Beispiel wartet die Funktion darauf, dass die Hardware den Status Sample erreicht hat und z.B. im Fall eines Timeouts mit false zurückkommen.

```

1 if (!waitForHardware(COMMAND_DIL, _SAMPLE))
2     return _SimpleErrorCode;

```

Listing 4.21: CallWaitForHardware

Die Erkennung bzw. das Auslösen eines Timeouts kann auf zwei verschiedene Arten geschehen. Einerseits wird die Funktion „WaitForSingleObject“ wie bereits unter Kapitel 4.4.4 beschrieben den Timeout-Wert zurückgeben wenn die übergebene Zeit vergangen ist, ohne dass das HANDLE frei wurde. In diesem Beispiel bedeutet das, dass die Methode „catchAnswer“ (Siehe Abb. 4.13) innerhalb von 15 Sekunden keine gültige Antwort empfangen hat und

der Fehlercode für einen Hardware-Timeout wird gesetzt. Andererseits läuft für die Dauer von „WaitForHardware“ eine Stoppuhr, welche nach 30 Sekunden die Funktion beendet, wenn nicht die passende Antwort empfangen wurde. Zusammenfassend bedeutet das, dass die Methode endet, wenn 15 Sekunden lang keine Antwort oder 30 Sekunden lang nicht die richtige Antwort empfangen wurde.

#### 4.4.10 Set-Sate

Der Aufbau der Status-Setzen-Methoden ist immer sehr ähnlich. Das Grundkonzept ist hier an „SetStatePARAM“ beispielhaft dargestellt (Listing 4.22 und Listing 4.23). Zunächst wird überprüft, ob es sich um einen gültigen Statuswechsel handelt. Param kann z.B. nur von IDLE aus aufgerufen werden. Dann wird der entsprechende Befehl wie unter Kapitel 3.3 beschrieben aufgebaut und der Befehl über den Kommandosocket (Port 2000) an die Hardware gesendet. Hierbei gibt es zwei mögliche Fehlerquellen, welche überprüft werden und mit einem Error-Code versehen sind. Zum einen „-2204“ für einen ungültigen Statuswechsel und zum anderen „-2201“ für einen Fehler beim Senden des Befehls. Sollte alles nach Plan verlaufen, gibt die Funktion true zurück.

```

1 bool EmusFrontEnd::setStatePARAM()
2 {
3     if (_currentState != _IDLE && _currentState != _NOSTATE)
4     {
5         _SimpleErrorCode = -2204;
6         return false;
7     }
8
9     stateEncode(SET_STATUS_PARAM);
10
11     // send _sendBuf
12     if (!socket1.SendData(_sendBuf, _sendDummy))
13     {
14         _SimpleErrorCode = -2201;
15         return false;
16     }
17     _SimpleErrorCode = 0;
18     return true;
19 }

```

Listing 4.22: SetStatePARAM

```

1 bool EmusFrontEnd::stateEncode(const char setState [])
2 {
3     if (!strstr(setState, "00000201000"))
4         return false;
5     strcpy_s(_sendBufTemp, COMMAND_PC);
6     strcat_s(_sendBufTemp, setState);
7
8     if (strstr(setState, SET_STATUS_SAMPLE) || strstr(setState, SET_STATUS_CALIB))
9         strcat_s(_sendBufTemp, "1");

```

```
10
11  _sendLen = strlen(_sendBufTemp);
12  sprintf_s(_sendBuf, "%05d%s", _sendLen, _sendBufTemp);
13
14  return true;
15 }
```

Listing 4.23: stateEncode

# 5 Zusammenfassung der Hardware-Steuerung

Wie bereits in den vorhergehenden Kapiteln beschrieben, können alle Methoden einer Hardwareklasse, die von der Basisklasse geerbt und dann überschrieben wurden, über die Klasse "EmusHandling" aufgerufen werden.

## 5.1 Geerbte Basisfunktionen für LimaTest

Im Fall der, für die Steuerung der "alten" Hardware entwickelten, Klasse "EmusFrontEnd" haben diese Methoden die in den Abbildungen 5.1 bis 5.4 beschriebenen Aufgaben.

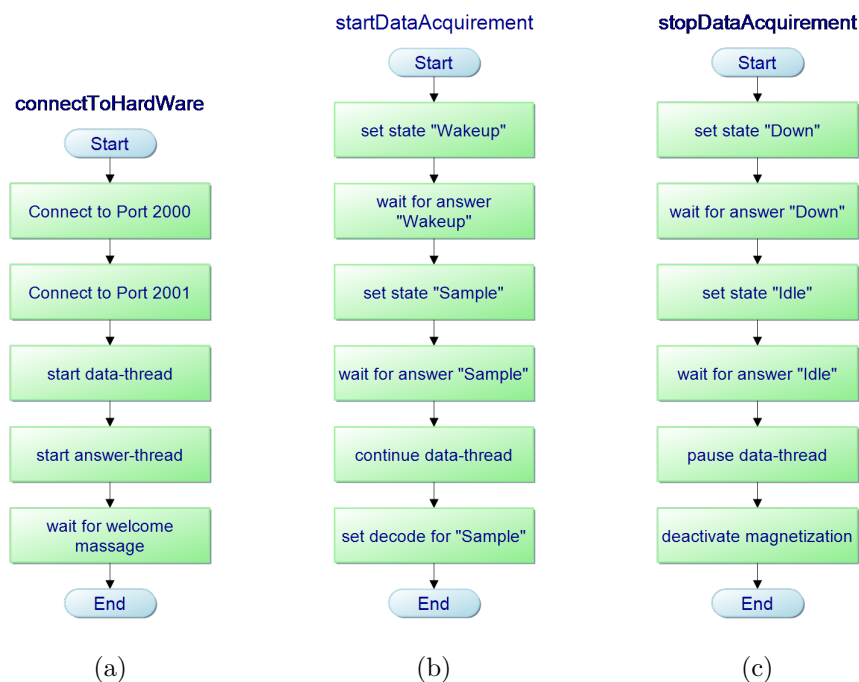


Abbildung 5.1: Basisfunktionen

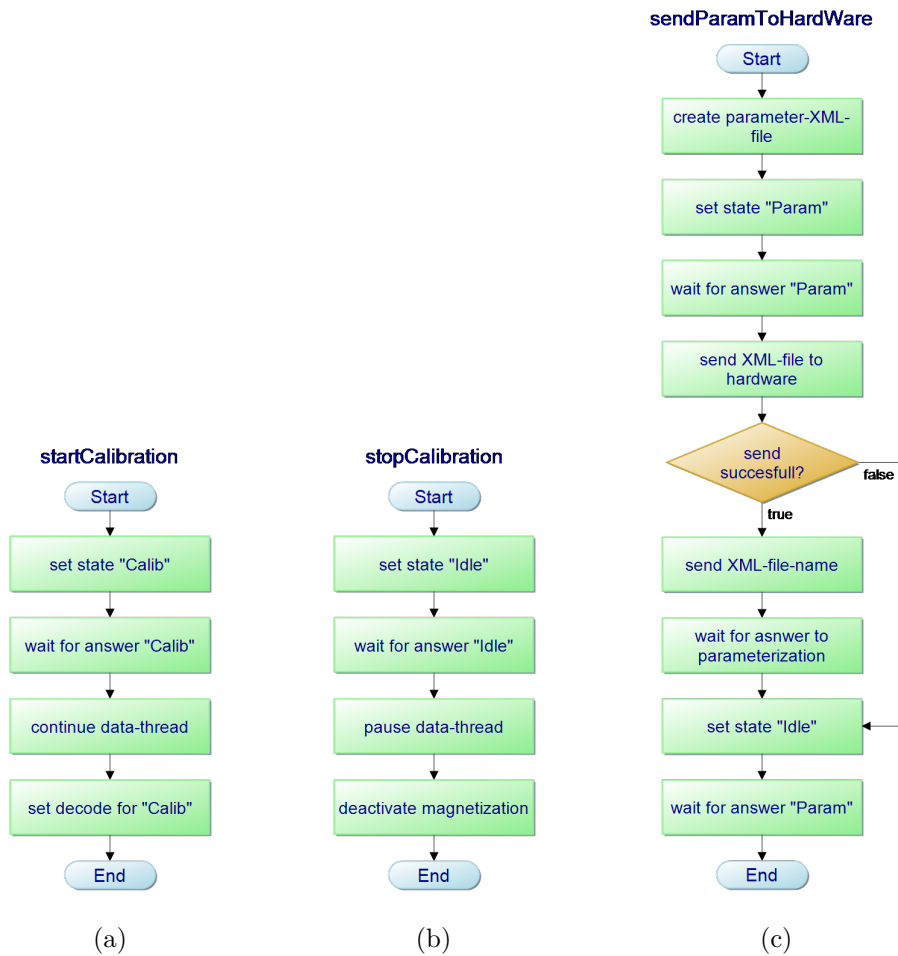


Abbildung 5.2: Basisfunktionen 2

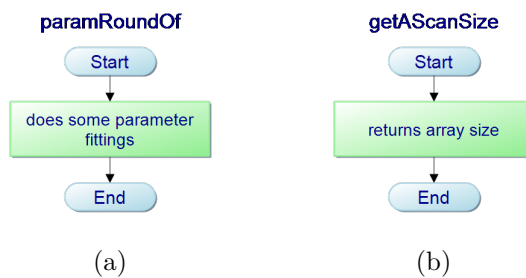


Abbildung 5.3: Basisfunktionen 3

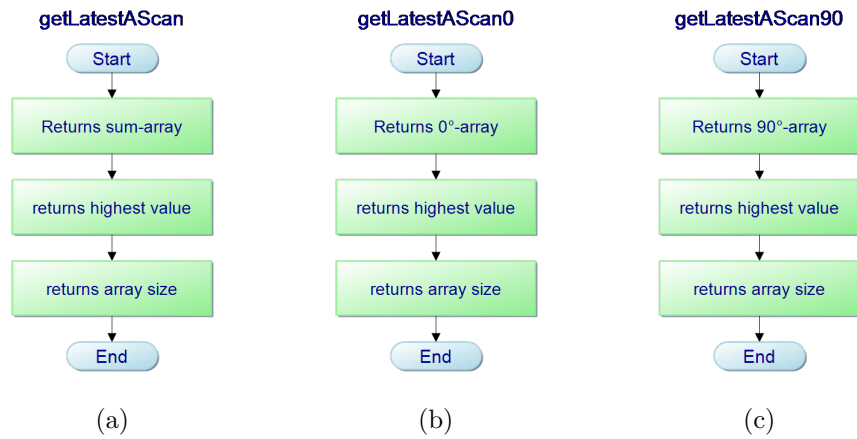


Abbildung 5.4: Basisfunktionen 4

## 6 Testumgebung und Resultate

Um die Funktionalität der Bibliothek sicher zu stellen, ist es notwendig während der Entwicklung immer wieder Tests durchzuführen. Daher ist es in diesem Fall erforderlich eine funktionsfähige Version der Hardware und einen geeigneten Prüfkörper zu Verfügung zu haben. Bei dem dazu verwendeten Prüfkörper handelt es sich um das in Abbildung 6.2 zu sehende Rohr, welches als Testkörper fungiert. Die Abmaße des Testobjektes sind in Abb. 6.1 und in Tab. 6.1 dargestellt. Wie in der Abbildung ebenfalls zu erkennen ist wurde ca. in Mitte des Rohrs eine Fehlerstelle eingearbeitet, welche erkannt werden soll. Außerdem stand ein Computer mit der „alten“ Software zur Verfügung, um z.B. eine Vergleichsmessung durchführen zu können.

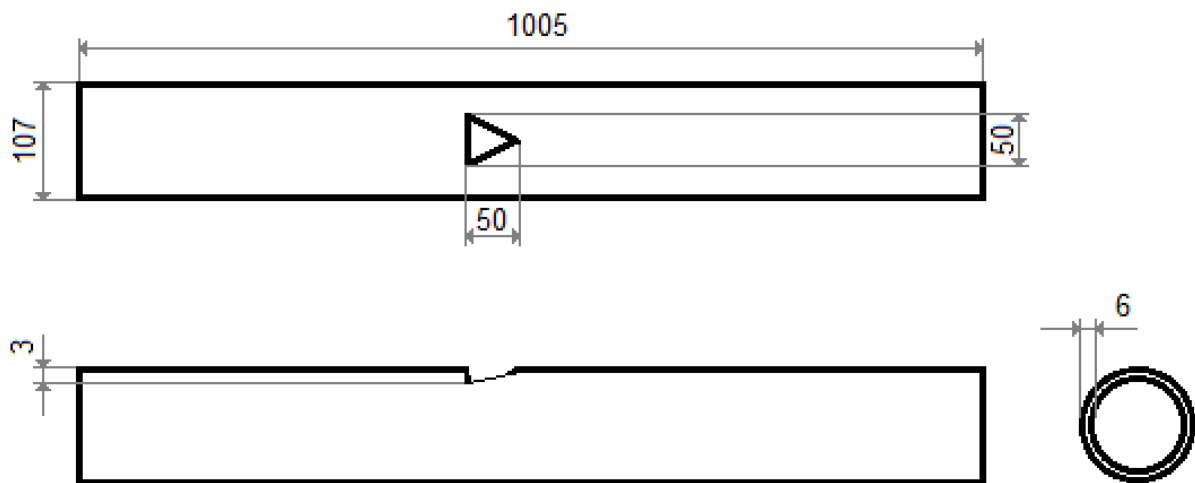


Abbildung 6.1: Prüfkörper-Abmessungen



Abbildung 6.2: Prüfkörper

<b>Gesamtlänge</b>	1005mm
<b>Abstand Oberkante-Störstelle</b>	435mm
<b>Außendurchmesser</b>	107mm
<b>Innendurchmesser</b>	95mm
<b>Wanddicke</b>	6mm
<b>Störstellen-Breite</b>	40mm
<b>Störstellen-Länge</b>	50mm

Tabelle 6.1: Testmast-Abmessungen

## 6.1 Testumgebung

Die Software wurde in einer Konsolenanwendung Namens „EmusHardWareTestApp“ getestet. Die Hardware kann durch Einbinden der entsprechenden Hardwareklasse, wie in Kapitel 4.3 gezeigt, angebunden werden. Dadurch kann über die Klasse Emushandling auf alle zur Steuerung benötigten Funktionen zugegriffen werden. Da in einer Konsolenanwendung nicht die Möglichkeit besteht, die Messdaten aussagekräftig darzustellen, werden diese Scan für Scan in eine Textdatei geschrieben. Dies geschieht, indem über die Klasse „SimpleEvent \_DataEvent“, welche Member der Basisklasse ist, eine Funktion bestimmt wird, die immer dann ausgeführt wird, wenn ein Scan beendet wurde. Innerhalb dieser Funktion wird dann die Methode „getLatestAScan“ aufgerufen um die Messwerte zu erhalten und in eine

Textdatei schreiben zu können.

```

1 void testFunction()
2 {
3     int max1;
4     int size;
5
6     memset(tempBuffer1, NULL, sizeof(tempBuffer1));
7     e.getLatestAScan(tempBuffer1, max1, size);
8
9     if (testcount == 0)
10        fprintf(flptr, "Size : %i\n", size);
11    for (int i = 0; i < size; i++)
12    {
13        if (i == size - 1)
14            fprintf(flptr, "%i ", tempBuffer1[i]);
15        else
16            fprintf(flptr, "%i ,", tempBuffer1[i]);
17    }
18    fprintf(flptr, "\n");
19    testcount++;
20 }
21
22

```

Listing 6.1: testFunction

```

1 int _tmain(int argc, _TCHAR* argv[])
2 {
3     string mode = "calib";
4
5     SYSTEMTIME datTimeThing;
6     LPSYSTEMTIME tmTime;
7     tmTime = &datTimeThing;
8     GetLocalTime(tmTime);
9
10    h.setHardWare(&e);
11
12    err = fopen_s(&flptr, "ErrorListTest.txt", "w");
13    fprintf(flptr, "Datum   : %i.%i.%i\n", tmTime->wDay, tmTime->wMonth, tmTime
->wYear);
14    fprintf(flptr, "Zeit   : %i:%i:%i\n", tmTime->wHour, tmTime->wMinute, tmTime
->wSecond);
15
16    h.getHardWare()->setDataEventFunction(testFunction);
17
18    h.getHardWare()->setTaktAnz(2);
19    cout << h.getHardWare()->getTaktAnz() << endl;
20    h.getHardWare()->paramRoundOff();
21
22    cout << "connectToHardWare: " << h.getHardWare()->connectToHardWare() <<
endl;

```

```

23 cout << "sendParamToHardWare: " << h.getHardWare()->sendParamToHardWare()
    << endl;
24 do
25 {
26     if (mode == "calib")
27     {
28         cout << "startCalibration: " << h.getHardWare()->startCalibration()
            << endl;
29         cout << e.getCurrentSource() << " " << e.getCurrentType() << " " << e
            .getCurrentState() << endl;
30
31         system("pause");
32         cout << "stopCalibration: " << h.getHardWare()->stopCalibration() <<
            endl;
33         cout << e.getCurrentSource() << " " << e.getCurrentType() << " " << e
            .getCurrentState() << endl;
34     }
35     if (mode == "sample")
36     {
37         cout << "dataAcquirement: " << h.getHardWare()->startDataAcquirement
            () << endl;
38         cout << e.getCurrentSource() << " " << e.getCurrentType() << " " << e
            .getCurrentState() << endl;
39
40         system("pause");
41         cout << "stopDataAcquirement: " << h.getHardWare()->
            stopDataAcquirement() << endl;
42         cout << e.getCurrentSource() << " " << e.getCurrentType() << " " << e
            .getCurrentState() << endl;
43     }
44     cout << "Mode: ";
45     cin >> mode;
46 } while(mode != "end");
47
48 cout << "Testprogram end!" <<endl;
49 fclose(flptr);
50 return 0;
51 }

```

Listing 6.2: EmusHardWareTestApp

## 6.2 Resultate

In Abbildung 6.3 ist zu sehen, wie die Messdaten in der fertigen Nutzeroberfläche aussehen können. Die Darstellung ist, wie unter Kapitel 2.3 beschrieben, aufgebaut. Bei dem mittleren Teil handelt es sich um den B-Scan (siehe Kap. 2.3.2), also ein zweidimensionales Ultraschallbild, welches den bei der Messung umlaufenden Teil des Mastes abbildet und bei dem unteren Teil um einen A-Scan (siehe Kap. 2.3.1), also die Darstellung einer einzelnen Ultraschallmessung, aus denen sich der B-Scan zusammensetzt. Bei dem linken Teil handelt es sich

um einen Querschnitt aller A-Bilder bzw. des B-Scans an einem bestimmten Samplepunkt. Hierbei ist die Reflektion entlang des gesamten Mast-Umfangs in einer bestimmten Tiefe zu erkennen.

In dieser Auflösung entspricht ein Samplepunkt entlang der X-Achse von A- oder B-Scan genau einem Millimeter. Daraus ergibt sich, dass entlang des Punktes 443 die Störstelle zu finden sein muss und der Mast bei Punkt 1005 endet.

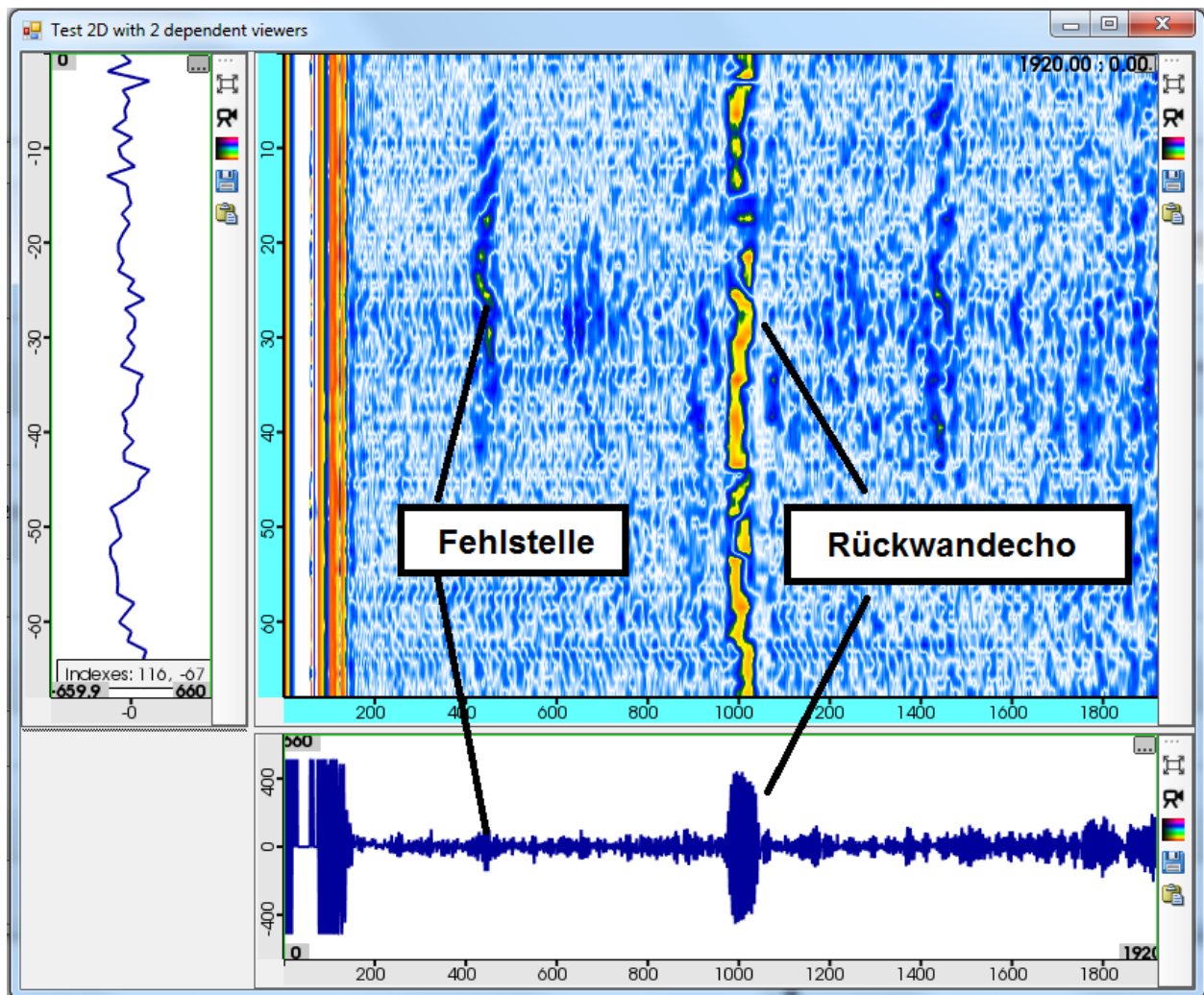


Abbildung 6.3: BScan unter Sample

In Abbildung 6.3 handelt es sich um eine Messung im Status Sample (siehe Kap. 3.1) mit nicht auf die Messung angepassten Parametern. Das Grundrauschen kann z.B. durch herabsetzen der Spulen-Verstärkung deutlich reduziert werden.

Dennoch ist bei (443/25) deutlich die Störstelle zu erkennen. Auch das Mastende ist anhand des Rückwandechos ungefähr bei Samplepunkt 1000 deutlich sichtbar. Dass der abgebildete Bereich so weit über die Mastlänge hinaus geht, hängt damit zusammen, dass die Messzeit für den ungefähr einen Meter langen Testmast deutlich zu groß eingestellt ist. Außerdem

ist zu sehen, dass bei diesem Parametersatz, im Bereich von bis zu etwa 17 Zentimetern vom Prüfkopf entfernt, durch das Eingangsrauschen keine verwertbaren Informationen zu entnehmen sind.

In Abbildung 6.4 ist eine weitere Messung zu sehen, bei der die Verstärkung reduziert wurde, wodurch vor allem das Hintergrundrauschen deutlich unterdrückt wird. Genau wie zuvor sind an den erwarteten Stellen die Störstelle und das Mastende zu erkennen, jedoch ist durch das geringere Grundrauschen, hier deutlich zu sehen, dass nach dem Rückwandecho noch weitere Echos folgen.

Das liegt daran, dass die Messtiefe deutlich höher eingestellt ist als das Mastende vom Prüfkopf entfernt ist. Daher ist es möglich, dass die Schallwellen an der Rückwand reflektiert werden, innerhalb der Messzeit wieder die Mastspitze erreichen und erneut reflektiert werden, wodurch sie wieder auf die Störstelle treffen. Dadurch werden sie von der Störstelle ein weiteres Mal reflektiert, wodurch es nach dem Rückwandecho zu weiteren Phantomechos kommen kann. In diesem Beispiel liegt der Auswahlcursor im B-Scan genau auf der Störstelle, wodurch im A-Scan ganz deutlich der Amplitudenunterschied zwischen Eingangsrauschen, Hintergrundrauschen, Störstelle und Rückwandecho zu erkennen ist.

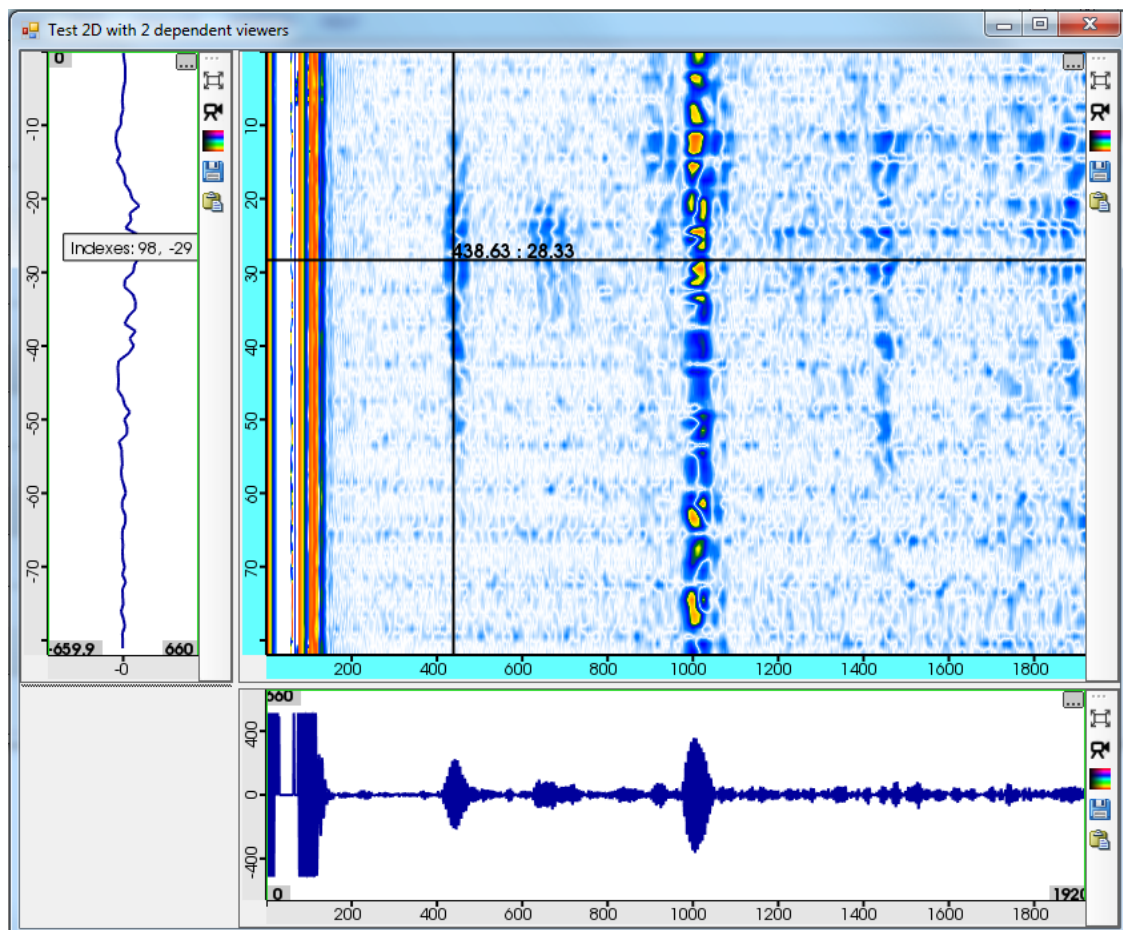


Abbildung 6.4: BScan unter Sample mit neuen Parametern

## 7 Zusammenfassung und Ausblick

Um schnell und effizient den Zustand und die Standfestigkeit von Lichtmasten beurteilen zu können, ist es unter anderem notwendig, ohne zu graben oder den Mast zu beschädigen, Informationen darüber zu erhalten, wie der Zustand des Teils des Mastes ist, der unter der Erdoberfläche verborgen liegt. Daher wurde ein System entwickelt, welches auf Basis von elektromagnetischem Ultraschall arbeitet und in der Lage ist, mögliche Oberflächenschäden des Mastes, auch unter der Erdoberfläche, zu detektieren.

Dies funktioniert, indem durch Einwirken eines statischen oder quasistatischen und eines hochfrequenten Magnetfeldes auf den Mast eine Ultraschallwelle direkt im Mast erzeugt wird, welche den Mast entlang läuft und an Störstellen oder dem Mastende reflektiert wird. Die reflektierte, zurücklaufende Welle kann durch den Prüfkopf, welcher sie auch erzeugt hat, erkannt werden und so anhand der Laufzeit des Ultraschallsignales bestimmt werden, in welcher Distanz zum Ursprung die Welle reflektiert wurde (Siehe Kap. 2). Durch die so gewonnenen Informationen ist es dann z.B. möglich, in einer graphischen Nutzeroberfläche ein Abbild des Mastes darzustellen, bei dem zu sehen ist, an welchen Stellen des Mastes die Ultraschallwellen in welchem Maße reflektiert werden. Da für die Zukunft auch neue Generationen dieses Systems geplant sind und die vorhandene Software nicht auf diese erweiterbar ist, ist es notwendig, eine Software zu entwickeln, welche sowohl mit der alten Hardware kompatibel bleibt, als auch für zukünftige Versionen erweiterbar ist.

Das Ziel dieser Arbeit besteht darin, eine C++-Klassenbibliothek zu entwickeln, welche es ermöglicht, sowohl vorhandene als auch zukünftige Hardwaregenerationen zu steuern und in eine graphische Nutzeroberfläche einzubinden, ohne die Nutzeroberfläche auf neue Hardware anpassen zu müssen. Erreicht wird dies, indem die entsprechende Hardwareklasse, wie in Kapitel 4.3 zu sehen, als Referenz an die Klasse „EmusHandling“ übergeben wird, die diese zu einem Pointer auf die Basisklasse „EmusBaseClass“ umwandelt, welchen sie als Member-Variable hält. Somit können alle Funktionen der Basisklasse und der ausgewählten Hardwareklasse über die Handlingklasse aufgerufen werden. Bei der Klasse „EmusFrontEnd“ handelt es sich um die Hardwareschnittstelle für die aktuelle Hardware. Die Hauptaufgaben dieser Hardwareklasse bestehen darin, die Zustände der Hardware zu setzen, die Antworten der Hardware auszuwerten und die Messdaten anzunehmen, zu dekodieren (Siehe Kap. 4.4.8) und weiterzugeben. Das Verfahren zur Dekodierung hängt hierbei davon ab, in welchem der beiden möglichen Zustände, also Sample oder Calib, die Daten aufgenommen wurden. Außerdem ist es wichtig, sowohl beim Auslesen von Messdaten als auch bei sonstigen Antworten der Hardware, immer genau die richtige Menge an Daten aus dem entsprechenden Hardware-Buffer auszulesen, da sonst alle nachfolgenden Informationen fehlerhaft sind. Um dies zu gewährleisten, wird während dem Auslesen immer die Anzahl der erhaltenen Werte und sowie der erwarteten Werte verglichen. Um sowohl Antworten als auch Messdaten unabhängig

voneinander und der restlichen Software zu machen, müssen die entsprechenden Funktionen in je einem eigenen Thread laufen. Bei der Verwendung von Threads muss hierbei darauf geachtet werden, dass die gewählte Bibliothek nicht die Einbindung in eine .Net Umgebung verhindert (Siehe Kap. 4.4.4).

Durch die neu entwickelte Klassenbibliothek ist es zum einen möglich, die „alte“ Hardware weiter zu betreiben und mit einer neuen, übersichtlicheren und benutzerfreundlichen Oberfläche zu verbinden. Zum anderen ist die neue Prüfsoftware durch die abstrakte Einbindung einer gemeinsamen Basisklasse für alle Hardwareschnittstellen und somit auch für alle zukünftigen Hardwaregenerationen erweiterbar, die auf elektromagnetischem Ultraschall basieren.

# Abbildungsverzeichnis

1.1	LimaTest . . . . .	2
2.1	Reflektion an einer Störstelle . . . . .	3
2.2	Interferenz . . . . .	4
2.3	Überlagerung von Wellen . . . . .	5
2.4	Reflektion an einer Störstelle . . . . .	6
2.5	Reflektion an einer Störstelle . . . . .	6
2.6	Prüfkopf von unten . . . . .	7
3.1	LimaTest Status Diagramm . . . . .	10
3.2	Kommunikations-Diagramm . . . . .	11
4.1	Klassendiagramm . . . . .	14
4.2	Aufbau der XML-Datei . . . . .	19
5.1	Basisfunktionen . . . . .	32
5.2	Basisfunktionen 2 . . . . .	33
5.3	Basisfunktionen 3 . . . . .	33
5.4	Basisfunktionen 4 . . . . .	34
6.1	Prüfkörper-Abmessungen . . . . .	35
6.2	Prüfkörper . . . . .	36
6.3	BScan unter Sample . . . . .	39
6.4	BScan unter Sample mit neuen Parametern . . . . .	40

# Tabellenverzeichnis

3.1	Zur Steuerung relevante Zustände . . . . .	10
3.2	Kommandos, um Status zu setzen . . . . .	11
3.3	Antworten auf Statusanfragen bzw. -wechsel . . . . .	12
3.4	Messdaten-Format . . . . .	13
3.5	Messdaten-Format, Darstellung der übertragenen ADC-Werte . . . . .	13
3.6	Werte-Format . . . . .	13
4.1	Klassen-Tabelle . . . . .	14
6.1	Testmast-Abmessungen . . . . .	36

# Literaturverzeichnis

- [DPV97] V. Deutsch, M. Platte, and M. Vogt. *Ultraschallprüfung: Grundlagen und Industrielle Anwendungen*. VDI-Buch. Springer, 1997.
- [mdna] microsoft developer network. beginthread. <https://msdn.microsoft.com/de-de/library/kdzttddb.aspx>. Zugriff: Montag, 15. Februar 2016.
- [mdnb] microsoft developer network. static cast operator. <https://msdn.microsoft.com/de-de/library/c36yw7x9.aspx>. Zugriff: Montag, 15. Februar 2016.
- [mdnc] microsoft developer network. Waitforsingleobject. <https://msdn.microsoft.com/en-us/library/windows/desktop/ms687032%28v=vs.85%29.aspx>. Zugriff: Montag, 15. Februar 2016.
- [Nie10] Frank Niese. *EMUS-Wanddickensensor für die Pipeline-Inspektion mit integrierter Wirbelstrom- und Streuflussprüfung, Dissertation*. Postfach 151141, 66041 Saarbrücken, 2010.
- [Weia] Christoph Weingard. Auropa III documentation. Internes Dokument.
- [Weib] Christoph Weingard. Xml-datensätze (system). Internes Dokument.