
DIPLOMARBEIT

Gesamtprojekt

iZiach

Entwicklung einer Ziehharmonika mit Griffsschriftaufzeichnung

Fabian Weng 5AHEL

Betreuer: Prof. Dipl.-Ing. Siegbert
Schrempf

Markus Dygruber 5AHEL

ausgeführt im Schuljahr 2017/18

Abgabevermerk:

Datum: 23.03.2018

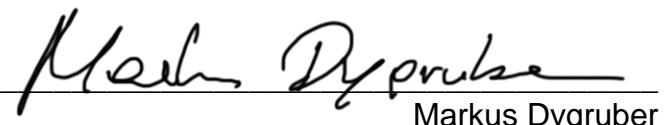
übernommen von:

Eidesstattliche Erklärung

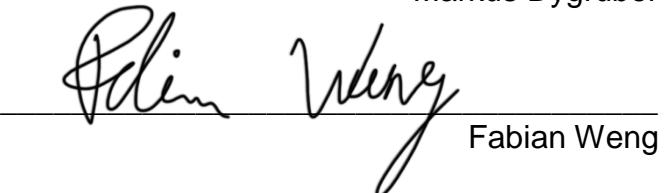
Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbständig und ohne fremde Hilfe verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Salzburg, am 23.03.2018

Verfasserinnen / Verfasser:



Markus Dygruber



Fabian Weng

**DIPLOMARBEIT
DOKUMENTATION**

Namen der Verfasserinnen / Verfasser	Markus Dygruber Fabian Weng
Jahrgang	5AHEL
Schuljahr	2017/18
Thema der Diplomarbeit	I4.0 - iZiach – Entwicklung einer Ziehharmonika mit Griffsschriftaufzeichnung

Aufgabenstellung	Beim Aufzeichnen eines Volksliedes auf der steirischen Ziehharmonika muss jede Note manuell in ein Notenschriftprogramm eingegeben werden. Daraus resultierend ist ein großer zeitlicher Aufwand nötig, um mit der Ziehharmonika gespielte Lieder dokumentieren zu können. Dieser Nachteil soll mit einer elektronisch ausgestatteten Ziehharmonika und der dazugehörigen Software behoben werden.
------------------	--

Realisierung	Es soll eine steirische Ziehharmonika mit Elektronik ausgestattet werden, welche jeden Ton eines Liedes aufnehmen und die Information an eine Software übertragen kann. Ziel ist es, den zeitlichen Aufwand für das Aufzeichnen eines Volksliedes zu minimieren.
--------------	--

Ergebnisse	Über ein Embedded System werden jeder Ton, die aktuelle Zugrichtung und die Zeitdauer an eine angebundene Software übertragen, welche die Informationsauswertung ausführt und die Daten visuell in Griffsschriftnotation darstellt. Bei Bedarf ist es nachfolgend möglich, die aufgezeichnete Notation manuell zu ändern.
------------	---



Teilnahme an Wettbewerben, Auszeichnungen	1. Preis bei Jugend Innovativ 2018 2. Preis bei AXAWARD 2018 2. Preis beim FH Kärnten Maturaprojekt-Wettbewerb 2018
---	---

Möglichkeiten der Einsichtnahme in die Arbeit	Schulbibliothek der HTBLuVA Salzburg
---	--------------------------------------

Approbation (Datum / Unterschrift)	Prüferin / Prüfer	Direktorin / Direktor Abteilungsvorständin / Abteilungsvorstand
---------------------------------------	-------------------	--

DIPLOMA THESIS

Documentation

Author(s)	Markus Dygruber Fabian Weng
Form	5AHEL
Academic year	2017/18
Topic	I4.0 - iZiach – Development of an Accordeon with Recording of Button-Pressing-Notation

Assignment of Tasks	For recording a folk song for the Styrian Accordeon every note has to be entered manually. Therefore a big effort is needed to document songs. This disadvantage should be resolved by using a electronical equipped accordion connected to a special software.
Realisation	A Styrian Accordeon should be equipped with electronic to measure tensile direction and pressed buttons. The information will be transferred to a connected computer. Aim of the project is the minimization of the required recording time.
Results	With the help of an embedded system the pressed buttons, the tensile direction and the timing data is transferred to the connected computer. The software analyses the data and calculates a visual presentation of the pressed button notation, called "Griffschriftnotation". If necessary, it is possible to edit the notes manually and change incorrectly entered notes.

Illustrative Graph, Photo
(incl. explanation)



Participation in
Competitions
Awards

Jugend Innovativ 2018 – 1st prize
AXAWARD 2018 – 2nd prize
FH Kärnten Maturaprojekt-Wettbewerb 2018 – 2nd prize

Accessibility of
Diploma Thesis

Accessible in the library of the HTBLuVA Salzburg

Approval
(Date / Sign)

Examiner

Head of College

Head of Department

Vorwort

In der vorliegenden Diplomarbeit beschäftigen wir uns mit der Entwicklung eines automatischen Aufzeichnungssystems für die Griffsschriftnotation der Steirischen Ziehharmonika. Dieses Projekt lässt sich grob in zwei Teile gliedern: Entwicklung und Bau der mit Elektronik ausgestatteten Ziehharmonika und Entwicklung der Software.

Die Projektidee kam von Markus Dygruber. Als passionierter Ziehharmonikaspieler suchte Markus vergeblich nach einer schnellen und einfachen Möglichkeit, Harmonikastücke in Griffsschrift niederzuschreiben. Nachdem das manuelle Eintippen der Noten einen großen Zeitaufwand mit sich bringt, kam die Idee, eine Steirische Ziehharmonika mit Elektronik auszustatten und so ein automatisches Aufzeichnungssystem zu entwickeln. Verbunden mit intelligenter Software wurde so ein Projekt realisiert, das die Aufzeichnung von Volksliedern auf der Steirischen Harmonika wesentlich vereinfacht.

Die individuellen Aufgabenstellungen wurden jeweils anhand der Spezialgebiete jedes Teammitglieds gewählt. Markus beschäftigte sich mit der Softwareentwicklung und dem WPF (Windows Presentation Foundation) Framework und Fabian widmete sich der Hardwareentwicklung und dem Luftdrucksensor BMP280.

Durch unser Projekt haben wir als Team gelernt, wie wichtig gute Kommunikation ist, um gemeinsame Ziele zu erreichen. Im Laufe der Projektarbeit traten immer wieder Schwierigkeiten auf, die wir mit Erfolg überwinden konnten. Besonders beim Einbau der Elektronik standen wir oft vor Problemen, die eine sachliche Herangehensweise, aber auch eine gute Abschätzung der Realisierbarkeit möglicher Lösungsschritte erforderten. Dank der exzellenten Zusammenarbeit im Team und der großartigen Unterstützung unseres Projektbetreuers konnten wir alle Schwierigkeiten meistern und das Projekt von der Idee bis zum bereits einsatzfähigen Produkt umsetzen. Die Realisierung von iZiach hat uns nicht nur beim Verständnis technischer Problemlösungen geholfen, es ist auch definitiv eine unschätzbare Bereicherung unserer persönlichen Erfahrungen.

An dieser Stelle danken wir besonders unserem Projektbetreuer Prof. Dipl.-Ing. Siegbert Schrempf für sein Engagement und die wichtigen Hilfestellungen in der Entwicklungsphase des Projekts, aber speziell für die Unterstützung bei der Teilnahme am Jugend Innovativ Wettbewerb und dem Redigieren diverser Dokumente.

Inhaltsverzeichnis

I. Eidesstattliche Erklärung.....	2
II. Diplomarbeit Dokumentation.....	3
III. Diploma Thesis	5
Vorwort.....	7
1 Systemspezifikation.....	13
1.1 Zielbestimmungen	13
1.1.1 Musskriterien	13
1.1.2 Wunschkriterien	13
1.1.3 Abgrenzungskriterien	13
1.2 Produkteinsatz.....	14
1.2.1 Anwendungsbereiche.....	14
1.2.2 Zielgruppen.....	14
1.2.3 Betriebsbedingungen	14
1.3 Produktumgebung.....	14
1.3.1 Software	14
1.3.2 Hardware.....	14
1.3.3 Orgware	14
1.4 Produktfunktionen	14
1.5 Produktdaten.....	15
1.6 Produktleistungen	16
1.7 Benutzungsoberfläche.....	16
1.7.1 Dialogstruktur.....	16
1.7.2 Startseite	16
1.8 Qualitätszielbestimmungen.....	17
1.9 Globale Testszenarien und Testfälle.....	17
1.10 Entwicklungsumgebung.....	18
1.10.1 Software	18
1.10.2 Hardware.....	18
1.10.3 Orgware	18
2 Projektmanagement.....	19
2.1 Überblick.....	19
2.2 GANTT-Diagramme.....	19

2.3 Scrum – agile Entwicklungsmethode.....	22
2.3.1 Einführung	22
2.3.2 Rollen.....	22
2.3.2.1 Product Owner.....	22
2.3.2.2 Entwicklungsteam.....	22
2.3.2.3 Scrum Master.....	22
2.3.2.4 Stakeholder	22
2.3.2.4.1 Kunde	23
2.3.2.4.2 Anwender.....	23
2.3.3 Artefakte.....	23
2.3.3.1 Product Backlog	23
2.3.3.2 Sprint Backlog	23
2.3.3.2.1 Burn-Down-Chart	24
2.3.3.3 Product Increment.....	24
2.3.4 Aktivitäten	24
2.3.4.1 Sprint Planning	24
2.3.4.2 Daily Scrum	24
2.3.4.3 Sprint Review	25
2.3.4.4 Sprint Retrospective.....	25
2.3.4.5 Product Backlog Refinement	25
3 Grundlagen und Methoden.....	26
3.1 Oberflächengestaltung mit Windows Presentation Foundation	26
3.1.1 Allgemeine Informationen	26
3.1.2 Vergleich mit Vorgängermodellen	26
3.1.3 Vergleich mit ähnlichen Frameworks.....	27
3.1.3.1 Plattformabhängigkeiten	28
3.1.3.2 Effizienz	28
3.1.4 Anwendungsbereiche.....	29
3.1.5 Funktionen von WPF	29
3.1.5.1 Die Sprache XAML.....	29
3.1.5.1.1 Grundlagen.....	29
3.1.5.1.2 Eigenschaftenelemente.....	30
3.1.5.1.3 Inhaltseigenschaften	31
3.1.5.1.4 Bindings	32

3.1.5.1.5	Statische Ressourcen.....	32
3.1.5.1.6	Komplizierbares XAML-Beispiel	33
3.1.5.2	Steuerelemente im CodeBehind.....	34
3.1.5.3	Wichtige Steuerelemente	35
3.1.5.3.1	Grid.....	35
3.1.5.3.2	StackPanel	36
3.1.5.3.3	Button.....	36
3.1.5.3.4	TextBlock.....	37
3.1.5.3.5	Label	37
3.1.5.3.6	TextBox.....	38
3.1.5.3.7	ListBox	38
3.1.5.4	Typkonverter.....	38
3.1.5.5	ControlTemplates	38
3.1.5.6	ItemTemplates	39
3.1.5.7	MVVM	40
3.1.5.7.1	Die View-Klasse	41
3.1.5.7.2	Die ViewModel-Klasse.....	41
3.1.5.7.3	Die Model-Klasse.....	42
3.1.5.7.4	Data Binding	42
3.1.6	WPF in Visual Studio.....	44
3.1.6.1	Erste Schritte.....	44
3.1.6.2	Verbindung von Code und Design	45
3.1.6.3	Entwicklung großer Applikationen.....	46
3.2	BMP280 Digital Pressure Sensor	47
3.2.1	Allgemeine Beschreibung.....	47
3.2.1.1	Vergleich mit BMP180	47
3.2.1.2	Eigenschaften.....	48
3.2.2	Funktionsbeschreibung	49
3.2.2.1	Blockschaltbild	49
3.2.3	Messzyklus.....	50
3.2.3.1	Druckmessung.....	50
3.2.3.2	Digitale Filter.....	51
3.2.4	Zeitliche Rahmenbedingungen.....	51
3.2.4.1	Sleep mode	51

3.2.4.2	Forced mode	51
3.2.4.3	Normal mode	52
3.2.4.4	Periodendauer und Abtastfrequenz	52
3.2.5	Memory Map des Bausteins	53
3.2.5.1	id-Register 0xD0	54
3.2.5.2	reset-Register 0xE0	54
3.2.5.3	status-Register 0xF3	54
3.2.5.4	ctrl_meas-Register 0xF4	54
3.2.5.5	config-Register 0xF5	55
3.2.5.6	Press-Register 0xF7, 0xF8 und 0xF9	56
3.2.6	Digitale Schnittstellen	57
3.2.6.1	I ² C Schnittstelle	57
3.2.6.1.1	Eigenschaften I ² C	57
3.2.6.1.2	Startbedingung	58
3.2.6.1.3	Adressierung	58
3.2.6.1.4	Acknowledge	58
3.2.6.1.5	Not Acknowledge	59
3.2.6.1.6	Datenübertragung	59
3.2.6.1.7	Stoppbedingung	59
3.2.6.1.8	Repeated-Startbedingung	59
3.2.6.1.9	Datenübertragung I ² C write	60
3.2.6.1.10	Datenübertragung I ² C read	60
3.3	Prototypingphase Drucksensor	61
4	Ergebnisse	62
4.1	Hardware	62
4.1.1	Auswahl der Hardware	62
4.1.2	Tastendruckdetektierung	62
4.1.3	Kommunikation mit der Software	66
4.2	Notenaufzeichnungssystem	70
4.2.1	Problemstellung	70
4.2.2	Aufbau des Projekts	70
4.2.3	Datenverarbeitungspfad	73
4.2.4	Die Griffsschriftnotation	74
4.2.5	Aufzeichnen von Griffsschrift	75

4.2.5.1	Eingabesystem	75
4.2.5.2	Erkennungsalgorithmus	75
4.2.5.3	Fehlererkennung und Optimierung	76
4.2.5.4	Verbindung mit der Oberfläche	77
4.2.6	Darstellung der Griffsschrift	77
4.2.6.1	Einbindung von MonoGame	78
4.2.6.2	Daten-/Cachingsystem	78
4.2.7	Bearbeitungsmöglichkeiten	80
4.2.8	Ausgabe der Griffsschrift	80
5	Kostenaufstellung.....	82
6	Schlusswort.....	83
7	Glossar	84
8	Quellen- und Literaturverzeichnis	85
9	Verzeichnis der Abbildungen, Tabellen und Abkürzungen	87
10	Begleitprotokoll gemäß § 9 Abs. 2 PrO	90
10.1	Begleitprotokoll Dygruber	90
10.2	Begleitprotokoll Weng.....	91
11	Anhang.....	92
11.1	Konstruktionspläne.....	92

1 Systemspezifikation

1.1 Zielbestimmungen

Ziel ist es, eine steirische Ziehharmonika so auszustatten, dass gespielte Volkslieder aufgenommen und in Griffsschrift am PC dargestellt werden können. Auf Abb. 1-1 ist der grobe Aufbau des Projekts ersichtlich.



Abb. 1-1 Ziel des Projekts

1.1.1 Musskriterien

Eingabeerkennung:

- Das Programm muss Tastendrücke der Diskanttasten und der Basstasten erkennen können.
- Es muss die Zugrichtung erkannt werden.
- Es muss eine fehlerfreie Kommunikation zwischen Hardware und Software gegeben sein.

User Interface:

- Die Noteneingabe muss über das User Interface gesteuert werden können.
- Es muss eine einwandfreie Darstellung der aufgenommenen Griffsschrift bestehen.
- Es müssen Funktionen zur Speicherung der aufgenommenen Daten implementiert werden.

1.1.2 Wunschkriterien

- Über die Benutzeroberfläche sollten Fehler der Hardware erkennbar sein.
- Aufgenommene Griffsschriftpartituren sollten manuell bearbeitet werden können.

1.1.3 Abgrenzungskriterien

- Es sollte nicht ein vollständiges Notensatzprogramm für alle Instrumente entwickelt werden.

1.2 Produkteinsatz

1.2.1 Anwendungsbereiche

Mit der ausgestatteten Ziehharmonika und der dazugehörigen Software kann auf einfache Weise ein Volkslied in Griffsschrift aufgezeichnet und auf Papier gebracht werden.

1.2.2 Zielgruppen

Jeder Ziehharmonikaspieler, der ein selbstkomponiertes oder bereits bestehendes Volkslied auf Papier bringen möchte, kann dies mit diesem Produkt auf einfache Weise tun.

1.2.3 Betriebsbedingungen

Für den Betrieb sind die umgebaute Ziehharmonika, die dazugehörige Software und etwas Spielkunst nötig. Das Produkt ist mobil, somit ist der Aufnahmeort frei wählbar. Es gibt keine weiteren Einschränkungen.

1.3 Produktumgebung

1.3.1 Software

Es wird die Software „iZiach“ benötigt.

1.3.2 Hardware

Es wird die umgebaute Ziehharmonika, ein Verbindungskabel und ein Windows-PC benötigt.

1.3.3 Orgware

Für eine sinnvolle Aufnahme ist etwas Spielkunst auf der steirischen Ziehharmonika nötig.

1.4 Produktfunktionen

/F0001/ Zugrichtungserkennung

Es wird zu jedem Zeitpunkt erkannt, ob mit der Ziehharmonika gerade „Druck“ oder „Zug“ gespielt wird.

/F0002/ Tastenerkennung:

Es wird zu jedem Zeitpunkt jede gedrückte Taste auf der Ziehharmonika erkannt. Dabei werden sowohl die Diskanttasten (Tasten auf der linken Seite) als auch

Basstasten (Tasten auf der rechten Seite) erkannt. Mehrere zum selben Zeitpunkt gedrückte Tasten werden ebenfalls erkannt.

/F0003/ Hardware-Software-Kommunikation

Bei bestehender Verbindung ist zu jedem Zeitpunkt eine Kommunikation zwischen Hardware (Ziehharmonika) und Software möglich. Die Software kann den Status der Ziehharmonika (gedrückte Tasten und Zugrichtung) in Echtzeit darstellen.

/F0004/ Eingabeerkennung

Bei bestehender Verbindung kann die Aufnahmefunktion in der Software aktiviert werden. Bei Aktivierung der Funktion wird jede gedrückte Taste aufgezeichnet und in Griffsschriftnotation dargestellt. Jeder aufgezeichnete Ton wird hierbei die Tonhöhe, die Notenlänge, die Zugrichtung und den dabei gedrückten Bass beinhalten.

/F0005/ Partiturmanagement

Es kann jederzeit eine neue Partitur (ein neues Lied) angelegt werden. Nach Eingabe von Titel und Autor und erfolgter Aufzeichnung kann die Partitur für eine spätere Verwendung abgespeichert werden. Gespeicherte Partituren können auch wieder geöffnet und bearbeitet werden.

/F0006/ Ausgabe

Aufgezeichnete Partituren können einfach ausgedruckt werden. Dabei wird der Ausdruck die visuelle Darstellung der Partitur in der Software widerspiegeln.

/F0007/ Manuelle Bearbeitung

Aufgezeichnete Partituren können bearbeitet werden. Es können Korrekturen der Notenhöhe, Notenlänge, Basstonart und Zugrichtung vorgenommen werden.

1.5 Produktdaten

/D001/ Partituren

Aufgezeichnete Partituren können als Datei abgespeichert und geöffnet werden. Eine Partitur enthält den Titel des Stücks, den Autor und die aufgezeichneten Noten.

1.6 Produktleistungen

/L001/ Toleranz

Bei fehlererzeugenden Eingaben hat der Benutzer die Möglichkeit, eine Korrektur der Eingabedaten vorzunehmen, ohne die Aufzeichnung wiederholen zu müssen.

/L002/ Echtzeit

Ereignisse der Hardware (Tastendrücke, Zugrichtungsänderung) werden in Echtzeit an die Software übertragen, so dass diese während einer Aufzeichnung die gespielten Noten sofort anzeigen kann. Die Hardware wird diese Änderungen in möglichst kurzer Zeit der Software melden.

1.7 Benutzungsoberfläche

Das User Interface sollte einfach zu bedienen sein und für keinen Nutzer ein Hindernis darstellen. Jeder Nutzer der Software hat alle Zugriffsrechte, somit kann jeder die grundlegenden Einstellungen der Software und Hardware bearbeiten.

1.7.1 Dialogstruktur

Über ein Hauptfenster, welches die aufgezeichnete Partitur, sämtliche Bearbeitungsmöglichkeiten und den aktuellen Status der Hardware beinhaltet, können alle weiteren Dialoge für Einstellungen, Erstellen neuer Partituren oder Drucken erreicht werden.

1.7.2 Startseite

Die Startseite sollte einen schnellen Einstieg in das Aufzeichnen erster Partituren bieten.

1.8 Qualitätszielbestimmungen

Die Software wird nur auf Systemen mit dem Betriebssystem Windows XP oder höher ausführbar sein.

Die folgende Tabelle zeigt die Priorität der Qualitätsanforderungen.

	sehr wichtig	wichtig	weniger wichtig	unwichtig
Robustheit		X		
Zuverlässigkeit	X			
Korrektheit	X			
Benutzungsfreundlichkeit	X			
Effizienz		X		
Portierbarkeit			X	
Kompatibilität			X	

Abb. 1-2 Qualitätszielbestimmungen

1.9 Globale Testszenarien und Testfälle

/T0001/ Zugrichtungserkennung

Bei bestehender Verbindung zwischen Hardware und Software muss die Zugrichtung („Zug“ oder „Druck“) in Echtzeit an die Software übertragen und in der Statusanzeige sichtbar sein. Wird die Ziehharmonika zusammengedrückt, muss in der Statusanzeige „Druck“ zu lesen sein. Wird sie auseinandergezogen, muss in der Statusanzeige „Zug“ zu lesen sein.

/T0002/ Tastenerkennung

Bei aufrechter Verbindung muss jede gedrückte Taste in Echtzeit an die Software übertragen und in der Statusanzeige sichtbar sein. Wird die erste Taste in der ersten Reihe gedrückt, muss in der Statusanzeige unter „Gedrückte Tasten“ der Text „1. Reihe, Taste 1“ dargestellt werden.

/T0003/ Hardware-Software-Kommunikation

Bei bestehender Verbindung ist zu jedem Zeitpunkt eine Kommunikation zwischen Hardware (Ziehharmonika) und Software möglich. Änderungen der Hardware müssen sofort in der Statusanzeige der Software sichtbar sein.

/T0004/ Eingabeerkennung

Bei bestehender Verbindung kann die Aufnahmefunktion in der Software von der Toolbox aus aktiviert werden. Bei Aktivierung der Funktion muss jeder Tastendruck eine neue Note erzeugen. Wird der Gleichton der zweiten Reihe gleichzeitig mit dem Unterbass der zweiten Reihe gedrückt, muss eine Viertelnote mit der Notenhöhe h1 und darunter der Bass B angezeigt werden.

/T0005/ Partiturmanagement

Wird eine neue Partitur mit dem Titel „test“, dem Autor „testmann“ erzeugt und ein paar Testnoten aufgezeichnet, muss die Partitur über „Datei → Speichern unter“ als Partitur-Datei speicherbar sein. Wird die gespeicherte Datei über „Datei → Partitur öffnen“ geladen, müssen alle zuvor eingegebenen Daten wie der Titel „test“, Autor „testmann“ und die aufgezeichneten Noten wieder angezeigt werden.

/T0006/ Ausgabe

Wird eine erstellte Partitur ausgedruckt, muss der Ausdruck exakt gleich aussehen wie die Anzeige in der Software. Am Seitenbeginn muss zentriert der Titel stehen, rechtsbündig der Autor. Alle Notenzeilen sollten klar und ohne Fehler sichtbar sein.

/T0007/ Manuelle Bearbeitung

Eine aufgezeichnete Partitur muss bearbeitet werden können. Sollte die Viertelnote mit Höhe h1, Bass B und Zugrichtung „Druck“ geändert werden, muss die Note mit einem Klick ausgewählt und in der Toolbox ein anderer Wert für Länge, Höhe, Bass oder Zugrichtung gewählt werden. Die Note sollte sofort aktualisiert werden und die Korrekturen darstellen.

1.10 Entwicklungsumgebung

1.10.1 Software

Als Entwicklungsumgebung wurde Visual Studio 2015 gewählt. Die Software wird in C# geschrieben. Die grafische Darstellung erfolgt über das WPF (Windows Presentation Foundation) Framework und über DirectX.

1.10.2 Hardware

Für die Ausstattung der steirischen Ziehharmonika mit der Sensorik werden der Arduino Mega 2560, der Luftdrucksensor BMP280 und Taster verwendet. Diese werden in die Harmonika eingebaut. Ein USB-Kabel verbindet den Arduino mit dem PC.

1.10.3 Orgware

Für das Projektmanagement wird taiga.io verwendet.

2 Projektmanagement

2.1 Überblick

Im Großen wurden folgende Aufgabenaufteilungen vorgenommen:

Markus Dygruber

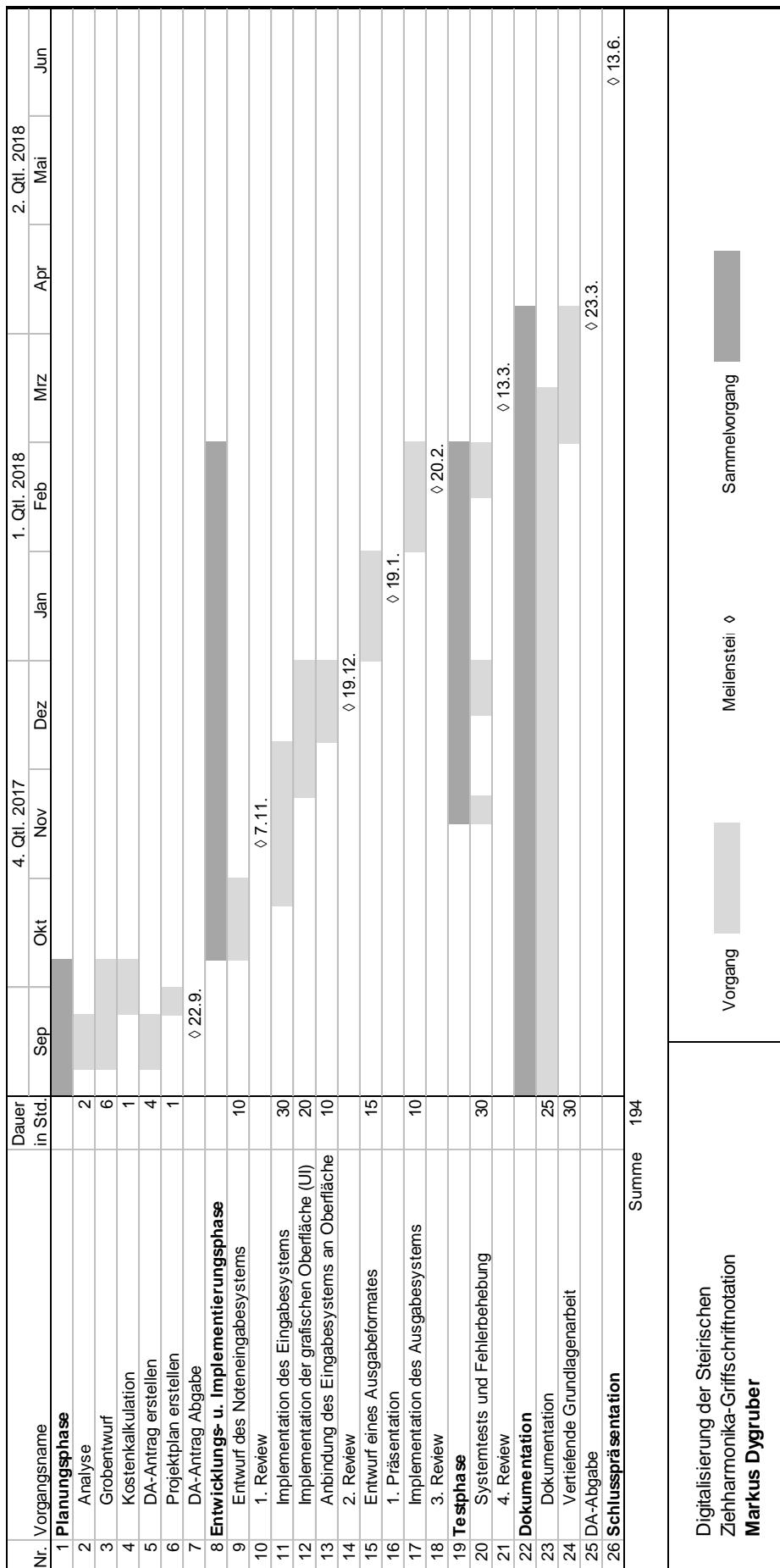
- Softwareentwicklung für Noteneingabesystem
- Graphische Aufbereitung der Griffsschrift
- Entwicklung eines geeigneten Ausgabeformates
- Projektmanagement

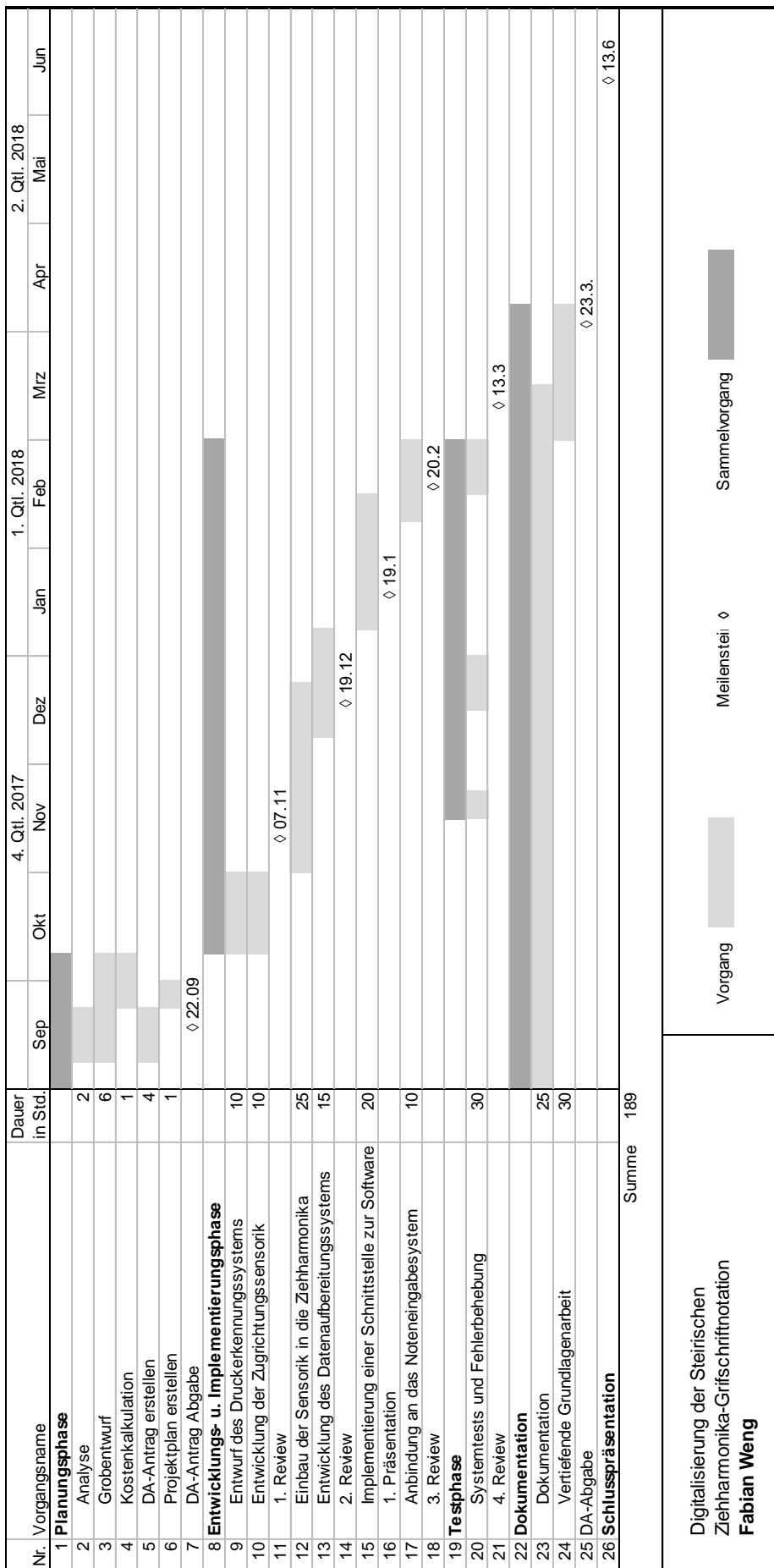
Fabian Weng

- Entwicklung der Elektronik zur Tastenerkennung
- Entwicklung der Sensorik für Zugrichtungserkennung
- Kommunikation mit der Software
- Integrationstests

2.2 GANTT-Diagramme

Um einen Überblick der Verteilung/Organisation der einzelnen Arbeitspakete zu erhalten, wurden sogenannte Gantt-Diagramme[1] pro Projektmitarbeiter erstellt.





2.3 Scrum – agile Entwicklungsmethode

2.3.1 Einführung

Scrum (engl. Gedränge) ist eine Organisationsmethode für das Projektmanagement, die insbesondere in der Softwareentwicklung angewendet wird. Grundgedanke hinter Scrum ist, dass es bei Entwicklungsprojekten schwierig ist, alles von Anfang an durchzuplanen. Es werden Zwischenlösungen geschaffen, mit denen fehlende Anforderungen und Lösungen einfacher und schneller behoben werden können. Diese Zwischenlösungen ermöglichen eine Agilität, die auch als eine Beweglichkeit innerhalb von Organisationssystemen beschrieben werden kann. Scrum beruht auf wenigen Regeln, die sich auf fünf Aktivitäten, drei Artefakten und vier Rollen beziehen.

2.3.2 Rollen

2.3.2.1 *Product Owner*

Der Product Owner zeichnet für den wirtschaftlichen Erfolg des Projektes verantwortlich. Er stellt die Anforderungen an das Produkt, priorisiert diese und er bestimmt auch, welche Eigenschaften am Ende eines Sprints fertiggestellt werden müssen. Für die Festlegung der Produktanforderungen verwendet der Product Owner das Product Backlog, das er laufend aktualisiert. Er steht in Kontakt mit den Stakeholdern und gibt dem Team regelmäßiges Feedback. Wichtig ist, dass der Product Owner unbedingt eine Einzelperson ist, denn kein Entwicklungsteam braucht Anweisungen von mehreren Personen.

2.3.2.2 *Entwicklungsteam*

Das Entwicklungsteam entwickelt das Produkt, mit den vom Product Owner gestellten Anforderungen und Funktionen. Das Team besteht aus drei bis neun Mitgliedern. Teammitglieder sollten idealerweise unterschiedliche Fachbereiche besetzen und die anderen Teammitglieder darin schulen. Durch die gegenseitige Unterstützung versucht das Entwicklungsteam, einen erfolgreichen Sprintabschluss selbstorganisiert zu erreichen.

2.3.2.3 *Scrum Master*

Der Scrum Master zeichnet sich dafür verantwortlich, dass Scrum funktioniert und Regeln im Projekt eingehalten werden. Er arbeitet zwar mit dem Entwicklerteam zusammen, gehört jedoch selbst nicht dazu. Der Scrum Master kümmert sich um Störungen im Team, wie mangelnde Kommunikation und Zusammenhalt. Außerdem hält er dem Entwicklungsteam den Rücken frei, falls Störungen von außen (zusätzliche Aufgaben außerhalb des Product Backlogs) auftreten und versucht diese zu beheben.

2.3.2.4 *Stakeholder*

Stakeholder ist jeder, der außerhalb des Scrum-Teams Interesse an dem Produkt hat. Die wichtigsten Stakeholder sind der Kunde und die Anwender.

2.3.2.4.1 *Kunde*

Der Kunde ist Auftraggeber und finanziert das Projekt. Der Product Owner sollte immer im regen Austausch mit dem Kunden stehen, damit der Kunde immer den aktuellen Stand begutachten und dazu sein Feedback geben kann.

2.3.2.4.2 *Anwender*

Anwender sind am Ende diejenigen, die das Produkt benutzen. Die Meinungen und Verbesserungsvorschläge dieser Rolle werden laufend in die Entwicklung des Produktes integriert. Kunde und Anwender können ein und dieselbe Person sein, müssen es jedoch nicht.

2.3.3 Artefakte

2.3.3.1 *Product Backlog*

Im Product Backlog werden alle Anforderungen an das Produkt gelistet, die nach der Wichtigkeit geordnet sind. Der Product Owner pflegt, ordnet und priorisiert die Einträge.

2.3.3.2 *Sprint Backlog*

Im Sprint Backlog sind alle zu erledigenden Einträge für einen Sprint enthalten. Die Einträge werden aus dem Product Backlog entnommen und dann einem Sprint zugeteilt. Der Status (Open, In Progress, Ready for Test, Closed) der Aufgaben in den Einträgen wird laufend von den Team-Mitgliedern aktualisiert. Dadurch lässt sich der aktuelle Status schnell und einfach nachvollziehen.

USER STORY	<>	IN PROGRESS	><	READY FOR TEST	><	CLOSED	><
✗ #1 Entwurf eines Druckerkennungssystem 20 points	+ ⋮					Fabian Weng #35 Zerlegung der Ziehharmonika	
◇ #12 Implementation des Ei...						Fabian Weng #36 Auswahl geeigneter Taster	
✗ #13 Implementation der grafischen Oberfläche 30 points	+ ⋮					Markus Dygruber #37 DirectX über MonoGame Framework einbinden	
						Markus Dygruber #38 Partitur-Renderer erstellen	
						Markus Dygruber #40 Aktuellen Aufzeichnungsstatus verwalten	

Abb. 2-1 Auszug aus dem Sprint Backlog vom November-Sprint

2.3.3.2.1 Burn-Down-Chart

Das Burn-Down-Chart visualisiert die verbleibende und bereits geleistete Arbeit. Auf der horizontalen Achse ist der Zeitbereich des Projektes und auf der vertikalen Achse die noch zu erledigenden Tasks aufgetragen. Zumeist gib es eine ideale Linie, die am Sprintende bei der Nulllinie endet. Im Vergleich der aktuellen und der idealen Kennlinie ist es möglich einzuschätzen, wann und ob das Sprintziel erreicht wird. Im Daily Scrum wird das Burn-Down-Chart aktualisiert.

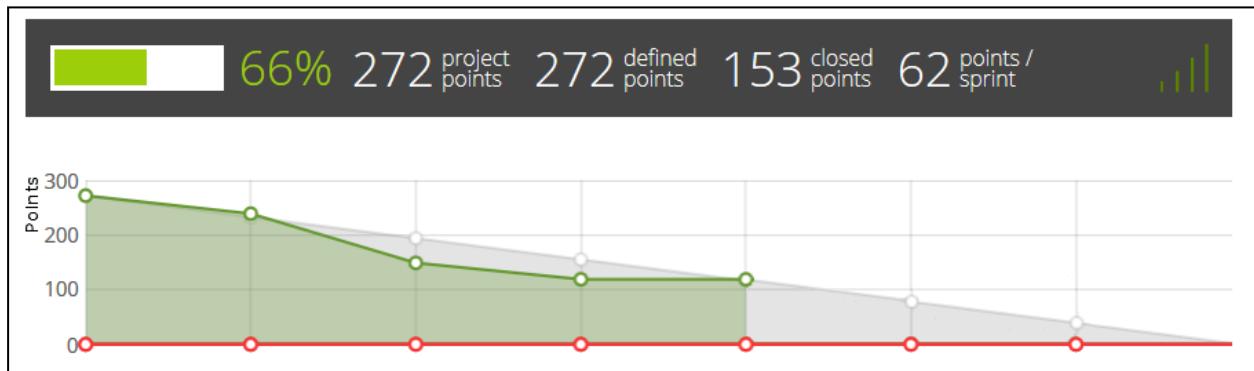


Abb. 2-2 Burn-Down-Chart

2.3.3.3 Product Increment

Das Product Increment ist das Zwischenprodukt, das nach einem erfolgreichen Sprint entwickelt wurde. Die alten Anforderungen wurden übernommen und die Neuen hinzugefügt.

2.3.4 Aktivitäten

2.3.4.1 Sprint Planning

Beim Sprint Planning werden die Anforderungen für den Sprint (Arbeitszyklus von 30 Tagen) festgelegt. Die Anforderungen werden dabei in konkrete Arbeitsaufgaben unterteilt, den sogenannten Tasks. Durch diese Unterteilung kann eine detaillierte Planung für einen Sprint durchgeführt werden. Resultat des Sprint Planning ist der Sprint Backlog.

2.3.4.2 Daily Scrum

An jedem Tag trifft sich das Scrum Team zu einem 15-minütigen Meeting. Jedes Teammitglied hat die Möglichkeit, sich mit den anderen Teammitgliedern auszutauschen. Dazu erzählt jeder kurz, was er zuletzt getan hat, welche Probleme dabei aufgetreten sind und was er als nächstes erledigen wird. Dieses Meeting wird zumeist im Stehen durchgeführt, um sicherzustellen, dass jeder nur kurz spricht. Ergeben sich größere Probleme, welche nicht innerhalb des Meetings gelöst werden können, so werden diese an den Scrum Master übergeben oder auf ein anderes Meeting verschoben.

2.3.4.3 *Sprint Review*

Am Ende eines Sprints führt das Entwicklerteam ein Sprint Review durch. Es wird das Zwischenprodukt überprüft und geklärt, ob alle gesteckten Ziele für den Sprint erreicht werden konnten. Im Sprint Review holt sich das Team auch das Feedback von Product Owner und Stakeholdern ein. Mithilfe des Feedbacks kann besprochen werden, was als Nächstes zu erledigen ist. Anhand dessen aktualisiert der Product Owner den Product Backlog. Er kann Anforderungen wieder zurück in den Product Backlog geben oder neue Anforderungen erstellen.

2.3.4.4 *Sprint Retrospective*

Die Sprint Retrospective wird auch wie das Sprint Review am Ende eines Sprints durchgeführt. Im Sprint Retrospective überprüft das Team die Arbeitsweise. Ziel ist es, die Arbeitsweise in Zukunft kontinuierlich effektiver und effizienter zu machen. Jedes Teammitglied sollte offen und ehrlich seine Meinung preisgeben. Der Scrum Master hilft dem Team, die neuen Arbeitsweisen für Verbesserungen im nächsten Sprint umzusetzen.

2.3.4.5 *Product Backlog Refinement*

Das Product Backlog Refinement ist eine fortlaufende Aktivität, die während des ganzen Projektzeitraums durchgeführt wird. Der Product Owner und das Entwicklungsteam entwickeln den Product Backlog ständig weiter. Hierzu gehören das Hinzufügen, Löschen, Ordnen, Aktualisieren, Schätzen von Einträgen sowie das Planen von Releases.[2]–[5]

Die Organisationsmethode für Projektmanagement, Scrum, wird immer häufiger in Wirtschaftsbetrieben angewendet. Einer dieser Betriebe ist der Raiffeisenverband Salzburg. (siehe https://news.wko.at/news/salzburg/5_2018-web.pdf; Seite 37)

3 Grundlagen und Methoden

3.1 Oberflächengestaltung mit Windows Presentation Foundation

3.1.1 Allgemeine Informationen

Das Windows Presentation Foundation Framework (kurz WPF) ist ein GUI-Framework (Oberflächenframework) des .NET Frameworks von Microsoft. Das 2006 eingeführte Framework ist auf allen Windows PCs seit Windows Vista (auf Windows XP ist eine Nachinstallation nötig) vorhanden und wird von vielen Windows-Applikationen verwendet. WPF verbindet DirectX, Windows Forms, Adobe Flash, HTML und CSS und stellt Softwareentwicklern damit ein umfangreiches Tool zur Verfügung.[6]



Abb. 3-1 WPF-Logo (Quelle: blogs.msdn.microsoft.com)

Mit der im April 2017 erfolgten Veröffentlichung des neuesten .NET Frameworks (Version 4.7) wurde auch die aktuellste Version des WPF Frameworks veröffentlicht. Die neueste Version bietet neben allgemeinen Performance- und Stabilitätsverbesserungen nun auch Touch-Support für WPF-Applikationen auf Windows 10 und unterstützt C# 7, die im Jahr 2017 veröffentlichte Aktualisierung der Programmiersprache C# (auch C-Sharp).[7]

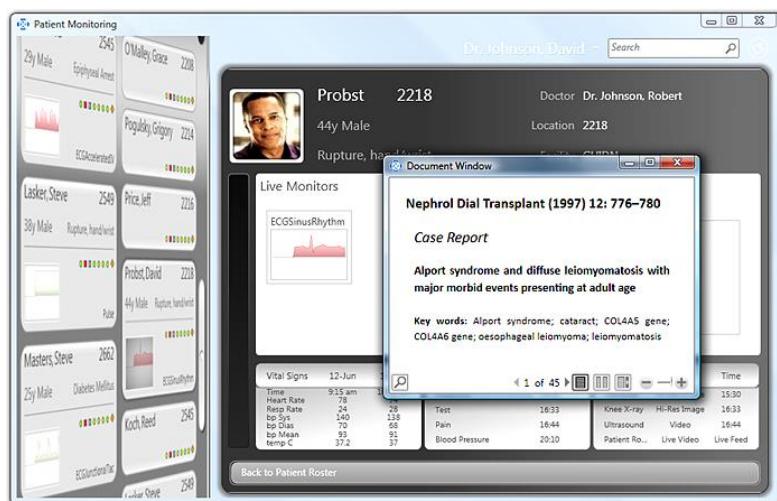


Abb. 3-2 Beispiel einer mit WPF entwickelten GUI (Quelle: msdn.microsoft.com)

3.1.2 Vergleich mit Vorgängermodellen

Das Vorgängermodell des WPF-Frameworks ist Windows Forms. Windows Forms war die jahrelang führende Technologie im Bereich Oberflächenentwicklung für Windows-Betriebssysteme. Besonders für Business-Apps mit Formularen, bei denen Plattformunabhängigkeit keine Rolle spielte, wurde die Technologie verwendet. Mit der

Veröffentlichung von WPF wurde die Weiterentwicklung von Windows Forms von Microsoft eingestellt und auf Fehlerbehebungen begrenzt.

Der große Vorteil von WPF gegenüber Windows Forms besteht in der Trennung von Code und Design. Eine mit WPF erstellte Oberfläche wird mit der „Extensible Application Markup Language“ (kurz XAML) beschrieben, welche eine auf XML-basierende von Microsoft entwickelte Sprache darstellt. Mit der Trennung der beiden Entwicklungsschritte können Designer und Programmierer unabhängig voneinander arbeiten und eine Effizienzersteigerung ermöglichen.[8]

Zusätzlich können mit WPF schneller und einfacher dynamische Formulare erstellt werden, was mit Windows Forms fast unmöglich ist. Controls (Steuerelemente) können ineinander verschachtelt werden, Eigenschaften werden mittels „Dependency Properties“ verbunden und ermöglichen grafische Benutzeroberflächen mit hochwertigen grafischen Effekten. Im Gegensatz zu Windows Forms werden in WPF Controls und Effekte hardwarebeschleunigt auf der GPU (Graphics Processing Unit) berechnet. Dies führt besonders bei grafisch anspruchsvolleren Applikationen zu einer enormen Performancesteigerung.[9]

3.1.3 Vergleich mit ähnlichen Frameworks

Zu den derzeit am weitesten verbreiteten Oberflächenframeworks gehören .NET mit WPF oder Windows Forms, Swing, Cocoa, GTK+ und Qt. Die Wahl des GUI-Frameworks wird hauptsächlich durch die Anforderung möglicher Plattformabhängigkeiten sowie durch die verwendbaren Programmiersprachen bestimmt.

Das Toolkit Swing wurde von Sun Microsystems für die Programmiersprache Java erstellt und ist ein fester Bestandteil der Java-Runtime. Swing ist der Nachfolger von AWT (Abstract Window Toolkit) und bietet ein vielseitiges Toolkit für grafische Oberflächen, die keine hohen grafischen Anforderungen haben.[10]

Cocoa wurde von Apple entwickelt und ist die meist verwendete API (Application Programming Interface) für grafische Benutzeroberflächen auf dem macOS. Cocoa-Programme werden in Objective-C und Swift geschrieben.[11]

GTK+ (GIMP-Toolkit) hat seinen Ursprung in der Open-Source Software GIMP. Anfangs wurde es für das Grafikprogramm entwickelt, heute ist es aber eines der erfolgreichsten GUI-Frameworks für das X Window System.



Abb. 3-3 GTK+ Logo
(Quelle: www.gtk.org)



Qt wurde von The Qt Company entwickelt und ist das am weitesten verbreitete GUI-Toolkit für die plattformübergreifende Entwicklung grafischer Benutzeroberflächen. Für kommerzielle Anwendungen muss allerdings eine kostenpflichtige Lizenz erworben werden.[12]

Abb. 3-4 Qt Logo
(Quelle www.qt.io)

3.1.3.1 Plattformabhängigkeiten

Das .NET Framework und somit auch WPF wurde von Microsoft entwickelt und ist ausschließlich auf dem Betriebssystem Windows verwendbar. Damit ist es mit Cocoa vergleichbar, welches ebenfalls nur auf dem von Apple entwickelten macOS läuft.

Swing hingegen ist fast plattformunabhängig. Es ist auf jedem Betriebssystem verwendbar, für das Oracle eine JRE (Java Runtime Environment) entwickelt hat. Dazu gehören Windows, Solaris, Linux und macOS.

Das GTK+ kann ebenfalls auf mehreren Betriebssystemen eingesetzt werden. Es werden Windows, Unix, Linux und macOS unterstützt.

Den bezüglich Plattformabhängigkeiten universellsten Einsatz bietet Qt. Das GUI-Toolkit bietet Unterstützung für alle auf X11 basierenden Betriebssysteme, Linux, Windows, Windows Phone, Windows RT, Windows CE, Symbian OS, Android, SailfishOS, macOS, iOS und Blackberry 10.[13]

WPF bietet auch die Möglichkeit zur Entwicklung von Browser Applikationen, sogenannten XBAP (XAML Browser Application). Diese können allerdings wieder nur von Windows Benutzer mit installiertem .NET Framework 3 ausgeführt werden. Damit bleibt WPF zwar stark plattformabhängig, hat aber aufgrund des Umfangs und der Effizienz eine hohe Priorität in der Entwicklung von leistungsstarken und performanten Business Apps.[13]

3.1.3.2 Effizienz

Durch die Trennung von Code und Design kann mit WPF um einiges effizienter gearbeitet werden als mit den anderen Frameworks. Während Winforms (Windows Forms) eine starke Bindung von Code und Design hatte und es so unmöglich machte, den Front-End Developer nicht mit Back-End Aufgaben zu belasten, können in WPF Entwickler und Designer getrennt voneinander arbeiten und sich so besser auf ihre Aufgaben fokussieren. Die dadurch ermöglichte Effizienzsteigerung bleibt dem WPF Toolkit vorbehalten, da kein anderes GUI-Framework eine vergleichbare Implementation vorzuführen hat.

Mit der Signal-Slot-Implementation von Qt bieten The Qt Company allerdings ein wichtiges Feature, das mit der zunehmenden Priorität von Multithreading-Applikationen ebenfalls einen enormen Effizienzanstieg bedeuten kann. Während sich der Entwickler von Multithreading-Anwendungen mit der Verwendung des .NET Frameworks und WPF um sicheren Datenaustausch zwischen zeitgleich arbeitenden Threads selbst kümmern muss, wird mit Qt und der Implementation von Signal-Slot-Verbindungen automatisch ein sicherer Datenaustausch zwischen den Threads durchgeführt.[13]

Bis auf die Plattformunabhängigkeit und die daraus resultierende Effizienz – da Anwendungen, falls sie auf mehreren Plattformen benötigt werden, keinen besonderen zusätzlichen Zeitaufwand bedürfen, wie es für die Portierung von .NET Programmen benötigt wird – bieten GTK+ sowie Swing keine besonderen Vorteile.

3.1.4 Anwendungsbereiche

WPF wird für alle möglichen Arten von grafischen Benutzeroberflächen verwendet. Von einfachen schlichten Formularen bis zu hochwertigen, grafisch anspruchsvollen Anwendungen bietet WPF performante und unkomplizierte Lösungen.

Die Verwendung von XBAPs, von WPF Browser Applikationen, hat sich jedoch aufgrund der Plattformabhängigkeit und der Entwicklung von HTML5, das im Vergleich zu XBAPs viele Vorteile bietet, nicht durchgesetzt.

3.1.5 Funktionen von WPF

WPF unterstützt eine breite Palette an Funktionen, darunter ein Anwendungsmodell, Ressourcenverwaltung, Steuerelemente, Grafik, Layout, Datenbindung, Dokumente und Sicherheit. Eine Oberfläche in WPF wird durch die Sprache XAML (Extensible Application Markup Language) beschrieben.

3.1.5.1 *Die Sprache XAML*

In WPF werden Oberflächen mit der Sprache XAML (Extensible Application Markup Language) beschrieben. XAML wurde 2009 von Microsoft eingeführt und ist eine Erweiterung der bekannten Sprache XML (Extensible Markup Language), die bereits 1996 entwickelt wurde. Mit XAML werden Oberflächen hierarchisch beschrieben, die XML Tags entsprechen dabei den Steuerelementen. Ein XAML Tag repräsentiert dabei direkt die Instanziierung eines Objekts, dessen Typ in dahinterliegenden Assemblies definiert ist. Bei der Instanziierung jedes Objekts wird der Standardkonstruktor (engl. default constructor) aufgerufen. Dies hebt XAML von anderen Markupsprachen ab, die standardmäßig interpretiert werden und an kein vergleichbares Unterstützungssystem im Hintergrund gebunden sind. Die Funktionsweise von XAML ist maßgeblich für die Trennung von Code und Design – von Oberfläche (UI) und Logik einer Applikation – verantwortlich.[8]

In einer XAML Datei können Layout, Stil, Effekte, anzuzeigende Daten, Ressourcen, Verbindungen zu Werten anderer Elemente (Bindings), Verhalten bei bestimmten Events (Behaviours) und Verbindungen zum „code-behind“ (deutsch: Code dahinter), also zu dahinter liegenden ViewModels (siehe Kapitel 3.1.5.7) definiert werden.

3.1.5.1.1 *Grundlagen*

Eine XAML-Datei wird standardmäßig mit der Erweiterung .xaml und UTF-8 kodiert gespeichert. Eine einfache XAML-Datei sieht folgendermaßen aus:

```
1  <Window Height="100" Width="200">
2      <Grid>
3          <Button Content="Klick mich!"/>
4      </Grid>
5  </Window>
```

Abb. 3-5 Beispiel einer XAML-Datei

Dieser Code beschreibt ein Fenster (Window) mit einem Button und dem Text „Klick mich!“ darauf. In Zeile 1 wird ein Fenster mit einer Höhe von 100 Pixel und einer Breite von 200 Pixel erzeugt. Die Eigenschaften Höhe und Breite werden als Attribute des Objektelements ausgedrückt. Eine Eigenschaftszuweisung besteht aus der Eigenschaft (Height), dem Zuweisungsoperator (=) und nachfolgend dem Attributwert, der in einer unter Anführungszeichen gesetzten Zeichenfolge angegeben wird. In Zeile 2 wird ein Grid (deutsch: Raster/Gitter) hinzugefügt. Durch die Positionierung zwischen dem Opening-Tag („<Window>“) und dem Closing-Tag („</Window>“) wird das Gitter dem Fenster als sog. „child“ (oder auch Content, deutsch: Inhalt) hinzugefügt. In Zeile 3 wird dem Gitter wiederum ein Button als Content hinzugefügt. Der Schriftzug „Klick mich!“ wird wie zuvor Höhe und Breite wieder als Property (deutsch: Eigenschaft) des Buttons gesetzt. Der Inhalt des Buttons könnte aber auch folgendermaßen gesetzt werden:

```
1  <Window Height="100" Width="200">
2    <Grid>
3      <Button>Klick mich!</Button>
4    </Grid>
5  </Window>
```

Abb. 3-6 Beispiel einer XAML Datei: Definition des Inhalts

Anstatt den Text direkt im Button festzulegen, kann aber auch ein eigenes Textelement verwendet werden:

```
1  <Window Height="100" Width="200">
2    <Grid>
3      <Button>
4        <TextBlock Foreground="Green" Text="Klick mich!">
5      </TextBlock>
6    </Grid>
7  </Window>
```

Abb. 3-7 Beispiel einer XAML Datei: Definition mit Textblock

Hier wird dem Inhalt des Buttons ein TextBlock-Element hinzugefügt, welches den Member (auch Eigenschaft) „Foreground“ auf „Green“ (deutsch: grün) und den Text auf „Klick mich!“ festlegt. Damit wird nun ein Fenster mit einem Button und dem grünen Schriftzug „Klick mich!“ beschrieben.

3.1.5.1.2 Eigenschaftenelemente

Manche Eigenschaften können nicht als Attribute gesetzt werden, da eine Zeichenfolge nicht ausreichend für den Wert des Attributs ist. An dieser Stelle muss ein Eigenschaftenelement verwendet werden. Eigenschaftenelemente werden als Inhalt des Elements gesetzt und werden mit dem Opening-Tag „<Typname.Eigenschaftenname>“ angekündigt.

```

1  <Window Height="100" Width="200">
2      <Grid>
3          <Button>
4              <Button.Content>Klick mich!</Button.Content>
5          </Button>
6      </Grid>
7  </Window>

```

Abb. 3-8 Beispiel einer XAML-Datei: Eigenschaftenelement

In Abb. 3-8 wird die Eigenschaft Content des Buttons nicht über Attribute gesetzt, sondern über ein Eigenschaftenelement. Dieses wird mit „<Button.Content>“ definiert.

Die Vordergrundfarbe des TextBlocks könnte auf diese Weise wie folgt definiert werden:

```

1  <Window Height="100" Width="200">
2      <Grid>
3          <Button>
4              <TextBlock Text="Klick mich!">
5                  <TextBlock.Foreground>
6                      <SolidColorBrush Color="Green"/>
7                  </TextBlock.Foreground>
8              </TextBlock>
9          </Button>
10     </Grid>
11 </Window>

```

Abb. 3-9 Beispiel einer XAML-Datei: Eigenschaftenelemente

Das Resultat wäre hierbei dasselbe wie das Beispiel von Abb. 3-7, da der Wert „Green“, wenn er für Foreground verwendet wird, wo ein Brush (Farbe) erwartet wird, automatisch als SolidColorBrush (deutsch: solide Farbe) eingelesen wird. Wird anstatt eines SolidColorBrush ein LinearGradientBrush (linearer Farbverlauf), ein RadialGradientBrush (radialer Farbverlauf) oder ein ImageBrush (Bild) definiert, muss ein Eigenschaftenelement verwendet werden, da diese nicht als Attributwert angegeben werden können.



Abb. 3-10 Erzeugter Button mit grünem Schriftzug

3.1.5.1.3 Inhaltseigenschaften

Um XAML übersichtlicher zu gestalten, wurden bestimmte Simplifikationen in die Strukturen eingebaut. Die einfachste und übersichtlichste Art, Eigenschaften festzulegen, sind Attribute. Müssen Eigenschaften aber durch Eigenschaftenelemente festgelegt werden, führt dies zur schnell wachsenden Unübersichtlichkeit. Darum wurden Inhaltseigenschaften eingeführt, die für jeden Typ festlegen, für welche Eigenschaft die Opening- und Closing-Tags des Eigenschaftenelementes weggelassen werden können.

```

1 <Border>
2     <Border.Child>
3         <Button Content="Button1"/>
4     </Border.Child>
5 </Border>

```

Abb. 3-11 Beispiel einer XAML-Datei: Redundantes Eigenschaftenelement

```

1 <Border>
2     <Button Content="Button1"/>
3 </Border>

```

Abb. 3-12 Beispiel einer XAML-Datei: Inhaltseigenschaft

In Abb. 3-11 ist die redundante Definition des Eigenschaftenelements zu sehen. In Abb. 3-12 wurde diese weggelassen, da die Eigenschaft Child als Inhaltseigenschaft von Border festgelegt ist. Dies führt zu einer enormen Steigerung der Übersichtlichkeit.[14]

3.1.5.1.4 Bindings

Mit sogenannten „Bindings“ können Eigenschaften an Daten, Ressourcen, Eigenschaften des ViewModels (siehe Kapitel 3.1.5.7) oder auch an Eigenschaften anderer Elemente gebunden werden. So ist es unter anderem möglich, den Text einer TextBox oder die Sichtbarkeit eines Buttons an die Eigenschaften anderer Steuerelemente zu hängen, sodass bei Änderungen eines Members der Wert unverzüglich für andere Elemente ebenfalls übernommen wird. Mit folgendem Beispiel wird der Zusammenhang einer TextBox (zur Texteingabe) und eines Labels (zur Textanzeige) dargestellt:

```

1 <Window Height="100" Width="200">
2     <StackPanel>
3         <TextBox Name="box"/>
4         <Label Content="{Binding ElementName=box, Path=Text}">
5     </StackPanel>
6 </Window>

```

Abb. 3-13 Beispiel einer XAML Datei: Bindings

Anstatt eines Gitters (Grid) wurde nun ein StackPanel eingefügt. Wie das Grid kann auch das StackPanel mehrere childs besitzen, im Gegenzug zur Grid werden die Elemente vom StackPanel je nach Orientation (Ausrichtung) horizontal oder vertikal aufgelistet. Standardwert für den Member Orientation von StackPanel ist „Vertical“, sodass dieses Beispiel eine TextBox mit einem darunter liegenden Label erzeugt. Über das in Zeile 4 in geschwungenen Klammern angegebene Binding wird der Content des Labels an den Text der TextBox gebunden. Wird ein Text in die mit dem Namen „box“ definierte TextBox eingegeben, wird der Text zugleich als Content des Labels übernommen.

3.1.5.1.5 Statische Ressourcen

Über geschwungene Klammern werden auch statische Ressourcen verlinkt. Über statische Ressourcen können Elemente einmal definiert und mehrfach verwendet werden. Der folgende

Code zeigt die Implementierung eines LinearGradientBrush (linearer Farbverlaufspinsel) als Ressource.

```

1  <Window Height="100" Width="200">
2      <Window.Resources>
3          <LinearGradientBrush x:Key="brush">
4              <GradientStop Offset="0" Color="Black"/>
5              <GradientStop Offset="1" Color="White"/>
6          </LinearGradientBrush>
7      </Window.Resources>
8      <StackPanel>
9          <TextBox Foreground="{StaticResource brush}" />
10         <Label Foreground="{StaticResource brush}" />
11     </StackPanel>
12 </Window>
```

Abb. 3-14 Beispiel einer XAML-Datei: Statische Ressourcen

Das Beispiel von Abb. 3-14 beschreibt wieder ein Fenster mit TextBox und Label. Unter den Resources von Window wurde ein linearer Farbverlauf von schwarz auf weiß definiert. Mit der Eigenschaft „x:Key“, die jede Resource festlegen muss, wird von TextBox und Label auf die statische Ressource zugegriffen. Die Vordergrundfarbe (Foreground) von TextBox und Label wird über geschwungene Klammern (Markuperweiterungen) auf die Ressource festgelegt. Ohne statische Ressourcen müsste der Farbverlauf für TextBox und Label einzeln definiert werden, was zu einer unnötigen Verdopplung führen würde. Auch wenn Ressourcen nicht mehrmals verwendet werden, wird die Auslagerung in Ressourcen empfohlen, da dies der Lesbarkeit und der Editierbarkeit dient.

3.1.5.1.6 Kompilierbares XAML-Beispiel

Die zuvor genannten Beispiele dienen jeweils nur zur Erklärung der spezifischen Funktionen und Besonderheiten von XAML. Ein vollständiges XAML-Beispiel mit WPF, das kompiliert und ausgeführt werden kann, muss noch den Standard-WPF-Namespace und den XAML-Language-Namespace inkludieren. Zusätzlich muss der vollständig qualifizierte Klassenname angegeben werden, um das XAML-File zum Code-Behind (partielle Klassen im Hintergrund) zu verbinden.

```

1  <Window
2      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4      x:Class="iZiach.ExampleCode">
5      <Grid>
6          <!-- Inhalt -->
7      </Grid>
8  </Window>
```

Abb. 3-15 Kompilierbare XAML-Datei

In Zeile 2 auf Abb. 3-15 wird der Standard-WPF-Namespace eingebunden. Zeile 3 verweist auf den XAML-Language-Namespace. In Zeile 4 wird die Klasse auf „iZiach.ExampleCode“ festgelegt. Um vom Code-Behind auf die Oberfläche zugreifen zu können, um beispielsweise ein

ViewModel festzulegen oder Daten anzuzeigen, muss eine partielle Klasse mit dem Namen „ExampleCode“ im Namespace „iZiach“ deklariert werden. Die Klassendeklaration muss wie auf der folgenden Abb. 3-16 erstellt werden, kann allerdings auch in andere .NET Sprachen wie Visual Basic deklariert werden.

```

1  namespace iZiach
2  {
3      public partial class ExampleClass : Window
4      {
5          // Interaktionslogik, Laden von Daten, Festlegen der Viewmode, usw.
6      }
7 }
```

Abb. 3-16 Code-Behind einer XAML-Datei in C#

Die Verschachtelungen einzelner Elemente in XAML machen die Beschreibung großer komplexer Oberflächen möglich. Durch den hierarchischen Aufbau bleibt der entwickelte XAML-Code übersichtlich, dies führt zur schnellen Erreichbarkeit von nachfolgenden Fehlerbehebungen oder Codemodifikationen.[15]

3.1.5.2 Steuerelemente im CodeBehind

In WPF sind standardmäßig weniger Steuerelemente (auch „Controls“) verfügbar, als im Vorgänger Windows Forms. Zum einen kommt eine WPF Applikation mit weniger Steuerelementen aus als eine Winforms Applikation, da Elemente ineinander verschachtelt werden können, zum anderen können sogenannte UserControls ohne viel Aufwand selbst erstellt werden. Steuerelemente werden in der XAML-Datei deklariert, können aber auch im CodeBehind erstellt und einem anderen Control als Child hinzugefügt werden. Ein Fenster mit einem Button, wie es in Kapitel 3.1.5.1 beschrieben wird, könnte damit auch auf folgende Weise erzeugt werden:

```

1  public partial class ExampleClass : Window
2  {
3      private void CreateControls()
4      {
5          var button = new Button();
6          button.Content = "Klick mich!";
7          grid1.Children.Add(button);
8      }
9 }
```

Abb. 3-17 Erstellung eines Controls im CodeBehind

Um das Beispiel auf Abb. 3-17 ausführen zu können, muss folgende XAML-Klasse erstellt werden:

```

1  <Window
2      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4      x:Class="ExampleCode">
5      <Grid x:Name="grid1"/>
6  </Window>

```

Abb. 3-18 XAML-Code: Window mit Grid

Auch wenn in diesem Beispiel die Grid keine weiteren Childs beinhaltet, können trotzdem weitere Controls hinzugefügt werden, entweder über eine fixe Deklaration in der XAML-Datei oder zur Laufzeit über CodeBehind.[16]

3.1.5.3 Wichtige Steuerelemente

3.1.5.3.1 Grid

Das Grid (Gitter) wird verwendet, um einem Element mehrere Childs hinzuzufügen und diese mit benutzerdefinierten Layout anordnen zu können. Ein Grid kann Reihen und Spalten mit bestimmter oder relativer Breite bzw. Höhe definieren und Elemente darauf ausrichten. Im folgenden Beispiel werden zwei Spalten und drei Reihen erstellt.

```

1  <Grid>
2      <Grid.ColumnDefinitions>
3          <ColumnDefinition Width="*"/>
4          <ColumnDefinition Width="Auto"/>
5      </Grid.ColumnDefinitions>
6      <Grid.RowDefinitions>
7          <RowDefinition Height="2*"/>
8          <RowDefinition Height="50"/>
9          <RowDefinition Height="1*"/>
10     </Grid.RowDefinitions>
11 </Grid>

```

Abb. 3-19 Grid mit Spalten- und Reihendefinition

Mit ColumnDefinition wird eine Spalte festgelegt. In Zeile 3 wurde für den Member Width (Breite) ein „*“ verwendet, damit wird die Spalte den maximal verfügbaren Platz ausfüllen. „Auto“ in Zeile 4 bedeutet, dass sich die Spalte an die minimale Breite, den der Inhalt der Spalte braucht, halten soll.

Mit RowDefinition wird eine Reihe festgelegt. In Zeile 7 wird für die Höhe der ersten Reihe „2*“ und für die dritte Reihe „1*“ verwendet. Damit wird bestimmt, dass die erste Reihe immer die doppelte Höhe der ersten Reihe behalten soll. Die zweite Reihe wird in Zeile 8 mit einer festen Höhe von 50 Bildpunkten definiert.

Um Controls in bestimmte Zellen schieben oder über bestimmte Zellen zu strecken, können die Eigenschaften Grid.Row, Grid.Column, Grid.RowSpan und Grid.ColumnSpan zugewiesen werden. Grid.Row gibt nullbasiert die Reihe an, damit wird mit „0“ die erste Reihe angesprochen. Grid.Column legt ebenfalls nullbasiert die Spalte fest. Mit RowSpan und

ColumnSpan wird bestimmt, über wie viele Reihen bzw. Spalten sich das Control erstreckt. Standardwert für beide Eigenschaften ist 1. Um einen Button in die zweite Spalte über die zweite und dritte Reihe zu legen, müssen die Eigenschaften wie auf der folgenden Abbildung festgelegt werden.

```
1 <Button Grid.Column="1" Grid.Row="1" Grid.RowSpan="2"/>
```

Abb. 3-20 Button in Grid ausgerichtet

3.1.5.3.2 StackPanel

Das StackPanel wird wie die Grid dazu verwendet, mehrere Controls anzuordnen. Dabei geschieht die Anordnung nicht über Reihen und Spalten, sondern anhand der Reihenfolge der Childs (Elemente). Das StackPanel legt die Ausrichtung (Orientation) fest und listet den Inhalt entweder horizontal oder vertikal auf. Standardwert für Orientation ist „Vertical“. Ein StackPanel kann verwendet werden, um ein Icon (kleines Bild) und einen Text auf einem Button zu platzieren.

```
1 <Button>
2   <StackPanel Orientation="Horizontal">
3     <Image Source="icon.png"/>
4     <Label Content="Klick mich"/>
5   </StackPanel>
6 </Button>
```

Abb. 3-21 Button mit StackPanel

Das Beispiel auf Abb. 3-21 beschreibt einen Button mit StackPanel, auf dem ein Image-Objekt (Bild), das im Arbeitsverzeichnis liegt und mit „icon.png“ benannt ist, und ein Label mit dem Text „Klick mich!“ platziert wird. Um weitere Layout-Eigenschaften anzupassen, kann der Margin (Rand) der Objekte oder die Textausrichtung des Labels verändert werden.[17]

3.1.5.3.3 Button

Der Button wird verwendet, um Benutzereingaben über Mausklicks zu ermöglichen. Über die hierarchische Struktur von XAML können auf Buttons mehrere andere Steuerelemente eingebettet werden. Mit Windows Forms war dies nur über lange komplizierte Umwege zu erreichen. In Abb. 3-21 werden ein Bild und Text auf einem Button platziert. Um Benutzereingaben über den Button zu ermöglichen, können entweder Commands (Befehle) festgelegt werden oder das Click-Event (Ereignis) des Buttons verwendet werden. Im folgenden Beispiel wird das Click-Event verwendet.

```
1 □ <Button Click="Button_Click">
2   <StackPanel Orientation="Horizontal">
3     <Image Source="icon.png"/>
4     <Label Content="Klick mich"/>
5   </StackPanel>
6 </Button>
```

Abb. 3-22 Button mit Click-Event

Funktionen, die von bei Events aufgerufen werden, werden in .NET allgemein EventHandler genannt. Events und EventHandler werden in XAML wie Eigenschaften auch mit Attributen verbunden. Auf Abb. 3-22 wird mit „Click“ das Click-Event des Buttons angesprochen. Der Attributwert „Button_Click“ ist der Name des EventHandlers, also der Name der Funktion, die im CodeBehind aufgerufen werden soll. Die Funktion muss wie auf der folgenden Abbildung definiert sein.

```

1  private void Button_Click(object sender, RoutedEventArgs e)
2  {
3      // do something
4 }
```

Abb. 3-23 EventHandler im CodeBehind

Werden die Dateien in einem Projekt in Visual Studio bearbeitet, wird von Visual Studio im Hintergrund automatisch Code generiert, der das Event mit dem EventHandler verbindet. Öffnet man die versteckte Datei mit der Erweiterung „.g.cs“, kann dies eingesehen werden.

```
#line 36 "..\..\ExampleClass.xaml"
((Controls.Button)(target)).Click += new RoutedEventHandler(this.Button_Click);
```

Abb. 3-24 Verbindung Event mit EventHandler

3.1.5.3.4 TextBlock

T TextBlocks werden verwendet, um Text ähnlich wie in HTML zu formatieren. Mit TextBlocks kann Text über einfache Tags **fett**, *kursiv* oder unterstrichen formatiert werden. Dies geschieht über die Tags „<Bold>“, „<Italic>“ und „<Underline>“. Auch LineBreaks „<LineBreak/>“, Hyperlinks „<Hyperlink>“ und Spans „“ für weitere Formatierungen wie Vordergrund- oder Hintergrundfarbe sind möglich.

```

1 <TextBlock>
2     Dieser Text ist<LineBreak/>
3     <Bold>fett, </Bold><Italic>kursiv,</Italic> und
4     <Underline>unterstrichen.</Underline>
5 </TextBlock>
```

Abb. 3-25 Beispiel eines TextBlocks

Dieses Beispiel würde folgende Ausgabe erzeugen:

Dieser Text ist
fett, *kursiv*, und unterstrichen.

Abb. 3-26 Beispiel eines TextBlocks – Ausgabe

3.1.5.3.5 Label

A Labels können im Gegensatz zu TextBlocks auch andere Childs als Text besitzen. Der Text eines Labels wird anstatt einer Text-Eigenschaft wieder über die Content-Eigenschaft festgelegt.

Der Text eines Labels wird automatisch mit Rand ausgestattet und ist für die Verwendung für kurze Schriftzüge auf Buttons besser geeignet. Ein Label besitzt auch die Eigenschaften HorizontalContentAlignment und VerticalContentAlignment, mit denen die vertikale und horizontale Ausrichtung des Texts oder des Inhalts innerhalb des Labels festgelegt werden kann.

Ein Beispiel eines Labels ist auf Abb. 3-22 zu sehen, in dem ein Label zusammen mit einem Icon in einem Button eingebettet wird.

3.1.5.3.6 TextBox

TextBoxen werden für Text-Benutzereingaben verwendet. Über die TextWrapping Eigenschaft kann festgelegt werden, ob die TextBox mehrzeilig oder einzeilig bleiben soll. Über die Text-Eigenschaft der TextBox kann auf den eingegebenen Text zugegriffen werden. Die TextBox bietet ein TextChanged-Event, das ausgelöst wird, sobald der Text verändert wird. Vom CodeBehind kann entweder über eine Felddefinition der TextBox (mit Name-Eigenschaft) oder über ein Binding an eine ViewModel auf den veränderten Text zugegriffen werden.



Abb. 3-27 Abbildung einer TextBox

3.1.5.3.7 ListBox

ListBoxen werden für Auflistungen verwendet. Standardmäßig zeigen ListBoxen nur Zeichenfolgen (Strings) an. Um dies zu ändern, können ItemTemplates verwendet werden. Die Verwendung von ItemTemplates wird in Kapitel 3.1.5.6 beschrieben.

3.1.5.4 Typkonverter

Attributwerte können nicht immer als String oder als Markuperweiterung gelesen werden. Wird beispielsweise die Visibility (Sichtbarkeit) eines Controls in XAML festgelegt, muss die Zeichenfolge in die Enumeration Visibility konvertiert werden. Dafür sind sogenannte TypeConverter (Typkonverter) zuständig. Für eingebaute Datentypen und primitive Datentypen stellt WPF bereits Typkonverter zur Verfügung. So werden Booleans (True/False), Integer (Zahlen) und weitere automatisch in den jeweiligen Datentyp konvertiert. Wird ein benutzerdefiniertes Objekt mit eigenem Datentyp definiert, kann auch ein Typkonverter dazu erstellt werden. Typkonverter werden in WPF wie folgt deklariert:

```
1 [TypeConverter(typeof(MyClassConverter))]
2 public class MyClass {
3     // code
4 }
```

Abb. 3-28 Klassendeklaration eines Typkonverters

3.1.5.5 ControlTemplates

ControlTemplates werden verwendet, um das Standardaussehen eines Steuerelements zu ändern. Ein ControlTemplate kann über die Template-Eigenschaft, die jedes Element besitzt,

festgelegt werden. Ein Kontextmenü hat beispielsweise das Aussehen eines Menüs und erwartet Childs des Typs MenuItem. Sollte anstatt eines Menüs eine Box mit TextBox und Button geöffnet werden, kann ein ControlTemplate für das Kontextmenü festgelegt werden. Folgendes Beispiel zeigt die Verwendung eines ControlTemplates:

```

1 <ContextMenu>
2   <ContextMenu.Template>
3     <ControlTemplate>
4       <StackPanel Orientation="Horizontal">
5         <TextBox/>
6         <Button Content="OK"/>
7       </StackPanel>
8     </ControlTemplate>
9   </ContextMenu.Template>
10 </ContextMenu>
```

Abb. 3-29 ContextMenu mit ControlTemplate

Damit wird das Standardaussehen des Kontextmenüs verändert und stattdessen werden eine TextBox und ein Button angezeigt.

3.1.5.6 ItemTemplates

ItemTemplates werden von Controls verwendet, die mehrere Items beinhalten können. ItemTemplates werden über die ItemTemplate-Eigenschaft von ListBox oder ListView festgelegt. Um eine ListBox anstatt von einfachen Zeichenfolgen eine TextBox und einen Button in jedem Eintrag anzeigen zu lassen, kann das ItemTemplate der ListBox verändert werden. Folgendes Beispiel zeigt ein angepasstes ItemTemplate:

```

1 <ListBox>
2   <ListBox.ItemTemplate>
3     <DataTemplate>
4       <StackPanel Orientation="Horizontal">
5         <TextBox/>
6         <Button Content="OK"/>
7       </StackPanel>
8     </DataTemplate>
9   </ListBox.ItemTemplate>
10 </ListBox>
```

Abb. 3-30 ListBox mit ItemTemplate

Der ItemTemplate Eigenschaft wird ein DataTemplate zugewiesen, das wiederum ein StackPanel mit TextBox und Button enthält. Damit wird nun jeder Eintrag der ListBox als TextBox mit Button dargestellt. Typischerweise sollten in einem ListBox-Eintrag Daten angezeigt werden, diese können in diesem Beispiel in die TextBox eines jeden Eintrags geschrieben werden. Um auf die Daten jedes Eintrags zugreifen zu können, muss mit Bindings gearbeitet werden. Die Quelle der ListBox-Einträge wird über die ItemsSource Eigenschaft bestimmt. Um eine sogenannte TwoWay-Bindung zu erstellen, die Daten sowohl bei Änderung der einen als auch bei Änderung der anderen Quelle aktualisiert, muss in der ViewModel eine

ObservableCollection im Zusammenhang mit dem INotifyPropertyChanged-Interface (siehe Kapitel 3.1.5.7) verwendet werden. Diese Implementierung würde zu der Ausgabe folgender ListBox führen:

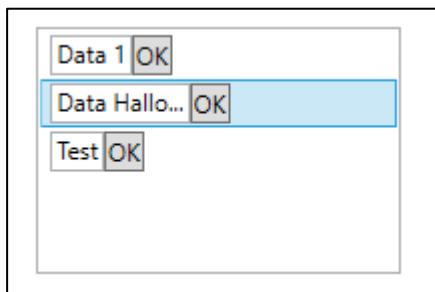


Abb. 3-31 ListBox mit ItemTemplate

Zusätzliche Layout-Verbesserungen können noch im ItemTemplate der ListBox eingefügt werden. Geeignet wäre, die Breite der TextBoxen auf einen festen Wert festzulegen und Margins (Ränder) hinzuzufügen. Wird der Text in einer der TextBoxen verändert, wird über die gebundene Eigenschaft automatisch auch der Text in der dahinter liegenden Auflistung (ObservableCollection), die in der ViewModel festgelegt ist, aktualisiert.

3.1.5.7 MVVM

MVVM steht für Model-View-ViewModel und beschreibt das Muster eines Anwendungsaufbaus für WPF Projekte. Wird MVVM eingehalten, können die Stärken und Vorteile von WPF am besten ausgenutzt werden.

Wie der Name bereits sagt, beschreibt MVVM die Trennung von drei Klassen. Die erste Klasse besteht aus dem Model, welches für die Daten und Hintergrundlogik (auch „business logic“) verantwortlich ist. Die zweite Klasse besteht aus der View (Oberfläche). Diese beinhaltet den kompletten visuellen Aufbau und verwaltet auch die rein auf der Oberfläche vorkommende Logik. Die dritte Klasse wird ViewModel genannt. Diese ist in gewisser Weise für die Verbindung von View und Model zuständig. Sie beinhaltet den aktuellen Status und die aktuellen Daten der Oberfläche und bindet sie an Eigenschaften. Werden Daten auf der Oberfläche verändert oder Aktionen (Commands) ausgeführt, wird die ViewModel diese Daten aufbereiten und an die Model-Klasse, der dahinterliegenden Logik, weitergeben. Kommen Daten oder Aktionen von der Model-Klasse, wird die ViewModel diese wieder aufbereiten und an die View melden. Das folgende Bild zeigt die Verbindung der drei Klassen:

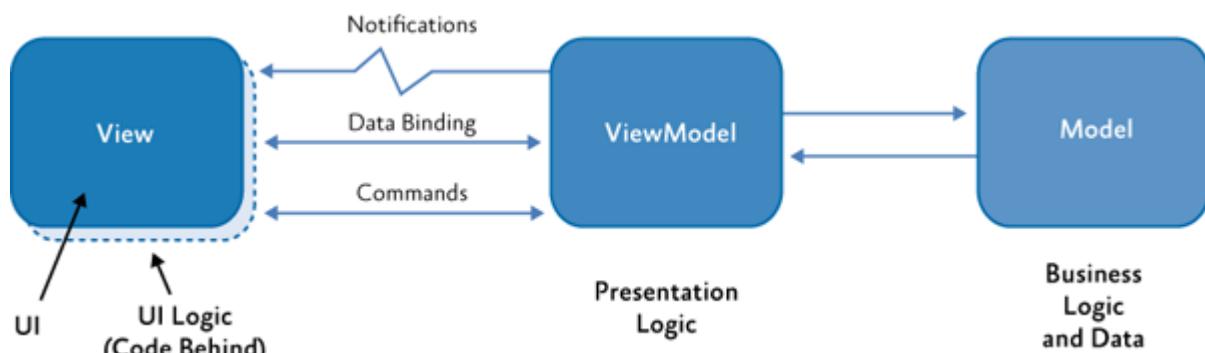


Abb. 3-32 Das MVVM Prinzip (Quelle: www.microsoft.com)

Die Trennung dieser drei Komponenten – der Oberfläche, der Logik, und der Verbindung dieser beiden – stellt einige Vorteile bereit. Zum einen können Logik und Oberfläche dadurch getrennt entwickelt werden, wodurch die Effizienz der Applikationsentwicklung steigt. Zum anderen bleibt ein großes Projekt übersichtlich und bietet daher die Möglichkeit, Weiterentwicklungen einfacher zu gestalten. Zusätzlich werden die Kernkomponenten von WPF, dazu gehören DataBinding, DataTemplates, Commands und Behaviours, wie von den Entwicklern von WPF vorgesehen, verwendet. Auch das Testen der Software wird vereinfacht, da Logik und UI-Funktionen getrennt getestet werden können.[18]

3.1.5.7.1 *Die View-Klasse*

Die View-Klasse besteht aus einem visuellen Element, zum Beispiel aus einem Window (Fenster), einer Page (Seite) oder einem UserControl (benutzerdefiniertes Steuerelement). Die View Klasse beinhaltet eine XAML-Datei und eine Code-Datei (CodeBehind).

In der XAML-Datei wird die Oberfläche in XAML-Code beschrieben und Bindings als auch Commands werden festgelegt. Alle visuellen Informationen, dazu gehören Stylings von Steuerelementen, das Layout, aber auch visuelle Änderungen bei Benutzerinteraktionen, werden im XAML-Code festgehalten.

Im CodeBehind werden nur Funktionen erstellt, die in XAML-Code nicht beschrieben werden können oder zu umständlich wären. Dazu gehören aufwendige Animationen, Übergänge oder das Überprüfen von Daten, die vom Benutzer eingegeben werden.

Die View-Klasse kommuniziert mit der ViewModel über die DataContext Eigenschaft. Diese wird bei der Initialisierung auf die ViewModel festgelegt. Verbindungen zu Eigenschaften (Bindings) und Befehle (Commands) werden vom XAML-Code aus direkt mit den Eigenschaften, die in der ViewModel definiert sind, verbunden. Können Daten aufgrund des Datentyps nicht direkt an die Eigenschaften der Steuerelemente gebunden werden, so werden ValueConverter (deutsch Wertumwandler) eingesetzt. ValueConverter sind kurze Klassen, die den Datentyp der Eigenschaft in der ViewModel in den Datentyp, der vom Steuerelement für die Anzeige benötigt wird, konvertieren. Bei normalen TwoWay-Bindings (siehe Kapitel 3.1.5.6) sollten auch ValueConverter verwendet werden, welche die Datentypen in beide Richtungen umwandeln können.[18]

3.1.5.7.2 *Die ViewModel-Klasse*

Die ViewModel-Klasse ist keine visuelle Klasse und erbt aus keiner in WPF enthaltenen Basisklasse. Sie stellt nur Eigenschaften und Befehle zur Verfügung, zu denen die View Verbindungen aufbauen kann. Die ViewModel enthält aber keine Informationen, auf welche Art und Weise die bereitgestellten Eigenschaften von der View dargestellt werden.

Eine der wichtigsten Aufgaben der ViewModel ist, bei Änderungen von Daten die View und die Model zu kontaktieren. Dies geschieht über das INotifyPropertyChanged bzw. über das INotifyCollectionChanged Interface. Die ViewModel referenziert ein oder mehrere Model-Klassen und kann teilweise auch Daten einer Model-Klasse direkt der View zur Verfügung

stellen. In diesem Fall muss aber auch die Model-Klasse das INotifyPropertyChanged Interface implementieren.

Die ViewModel sollte die Bereitstellung der Informationen für die View möglichst einfach gestalten. Dazu gehört etwa das Verbinden mehrerer Daten zu einer Eigenschaft oder das Bereitstellen von Eigenschaften, die von der Model nicht explizit als Daten gespeichert werden. Ein Beispiel hierfür wäre die für eine längenbegrenzte TextBox übrigbleibende Anzahl an Zeichen, die der Benutzer noch eingeben kann. Die Model wird dies nicht explizit speichern, die ViewModel könnte es aber berechnen und als Eigenschaft bereitstellen.

Auch den aktuellen Status der Anwendung sollte die ViewModel beinhalten. Werden von der Model Hintergrundaktionen ausgeführt, kann die ViewModel Eigenschaften definieren, über welche die View die Sichtbarkeit grafischer Elemente festlegen kann. Dazu könnten Animationen gehören, die den Benutzer darauf hinweisen, dass die Anwendung im Hintergrund arbeitet.

Eine weitere Aufgabe ist die erweiterte Überprüfung von Benutzereingaben, die in der View nicht möglich ist. Über die in WPF inkludierten Interfaces IDataErrorInfo und INotifyDataErrorInfo kann die ViewModel der View direkt Feedback geben, ob die eingegebenen Daten akzeptiert werden können.[18]

3.1.5.7.3 Die Model-Klasse

Die Model-Klasse enthält die komplette Logik und die Daten der Applikation. Sie hat allerdings keine Informationen über den Aufbau der Oberfläche oder deren aktueller Status. Die Model-Klasse kann Informationen direkt als Eigenschaften definieren und das INotifyPropertyChanged Interface implementieren, zu denen sich die View-Klasse direkt binden kann, wenn die ViewModel die Model-Klasse der View bereitstellt. Für Daten, für welche die Model-Klasse die PropertyChanged Interfaces nicht implementieren kann, sollte die ViewModel geeignete Eigenschaften zur Verfügung stellen, zu welchen die View Verbindungen aufbauen kann.

Die Logik der Model-Klasse sollte so abstrakt wie möglich gehalten werden, um mehrmalige Verwendung des Codes zu ermöglichen. Auch Services für den Datenzugriff und für das Zwischenspeichern (caching) der Daten kann die Model-Klasse inkludieren. Um die Übersichtlichkeit zu bewahren, werden dieser Services idealerweise in eigene Klassen ausgelagert, auf die die Model-Klasse wiederum Zugriff hat.[18]

3.1.5.7.4 Data Binding

Eine wichtige Rolle in WPF und besonders beim MVVM Prinzip spielen DataBindings. Wie bereits in Kapitel 3.1.5.1.4 gezeigt, werden DataBindings im XAML-Code definiert. Damit WPF aber eine Verbindung von Oberfläche zu Daten aufbauen kann, müssen im CodeBehind und in der ViewModel weitere Dinge berücksichtigt werden. Wird im XAML-Code eine TextBox an Text (wie auf Abb. 3-33 gezeigt) gebunden, muss in der ViewModel eine Eigenschaft erstellt werden.

```
1 <TextBox Content="{Binding Name}"/>
```

Abb. 3-33 TextBox mit DataBinding

Für dieses Beispiel muss eine Eigenschaft mit dem Namen „Name“ definiert werden, welche bei Änderungen das PropertyChanged-Ereignis auslöst, damit die Oberfläche bei Änderungen kontaktiert wird. Die ViewModel sieht damit folgendermaßen aus:

```

1  public class MyViewModel : INotifyPropertyChanged
2  {
3      private string name;
4      public event PropertyChangedEventHandler PropertyChanged;
5
6      public string Name
7      {
8          get { return name; }
9          set
10         {
11             if (value != name)
12             {
13                 name = value;
14                 if (PropertyChanged != null)
15                 {
16                     PropertyChanged(new PropertyChangedEventArgs("Name"));
17                 }
18             }
19         }
20     }
21 }
```

Abb. 3-34 Beispiel einer ViewModel

Wie in Zeile 1 zu sehen, implementiert die ViewModel das INotifyPropertyChanged Ereignis. Um die Übersichtlichkeit zu bewahren, wurde auf die Implementation einer Model-Klasse verzichtet. Die Model-Klasse würde in diesem Fall nur ein Feld mit dem Namen bereitstellen.

In Zeile 6 beginnt die Definition der Eigenschaft. Wird die Eigenschaft gesetzt, wird das PropertyChanged Ereignis ausgelöst. Da eine ViewModel üblicherweise mehrere Eigenschaften bereitstellt, wird dem Ereignis als Parameter der Eigenschaftsname mitgegeben. Somit weiß die View-Klasse, welcher Wert verändert wurde und kann das jeweilige Steuerelement, das die Datenbindung aufgebaut hat, aktualisieren.

Damit das Steuerelement (die TextBox) die Eigenschaft in der ViewModel auch findet, um eine Bindung aufbauen zu können, muss im CodeBehind der DataContext festgelegt werden. Dies kann entweder über den XAML-Code oder im Konstruktor der View gemacht werden.

```

1  public MainWindow()
2  {
3      InitializeComponent();
4      DataContext = new MyViewModel();
5  }
```

Abb. 3-35 DataContext im Konstruktor erstellen

```

1 <Window>
2     <Window.DataContext>
3         <my:MyViewModel/>
4     </Window.DataContext>

```

Abb. 3-36 DataContext über XAML erstellen

Wird die ViewModel, wie auf Abb. 3-36 gezeigt, über XAML erstellt, muss die ViewModel einen Standardkonstruktor, also einen parameterlosen Konstruktor, implementieren. Wird sie wie auf Abb. 3-35 im Konstruktor der View erstellt, könnten auch Parameter übergeben werden.[18]

3.1.6 WPF in Visual Studio

3.1.6.1 Erste Schritte

Visual Studio stellt zahlreiche Tools zur Verfügung, welche die Arbeit mit WPF und das Erstellen von Applikationen erleichtern. Im Folgenden wird mit Visual Studio 2015 gearbeitet.

Eine neue WPF Applikation kann über den Assistenten für neue Projekte in Visual Studio erstellt werden. Dazu wird die .NET Sprache (hier Visual C#) und das Template namens WPF Application ausgewählt.



Abb. 3-37 WPF Application Template

Es wird ein Name für das neue Projekt eingegeben (hier MyWPFApplication) und der Dialog mit OK bestätigt. Es öffnet sich automatisch der XAML Designer, der die neue XAML-Datei grafisch darstellt und mit dem die Oberfläche intuitiv erstellt werden kann.

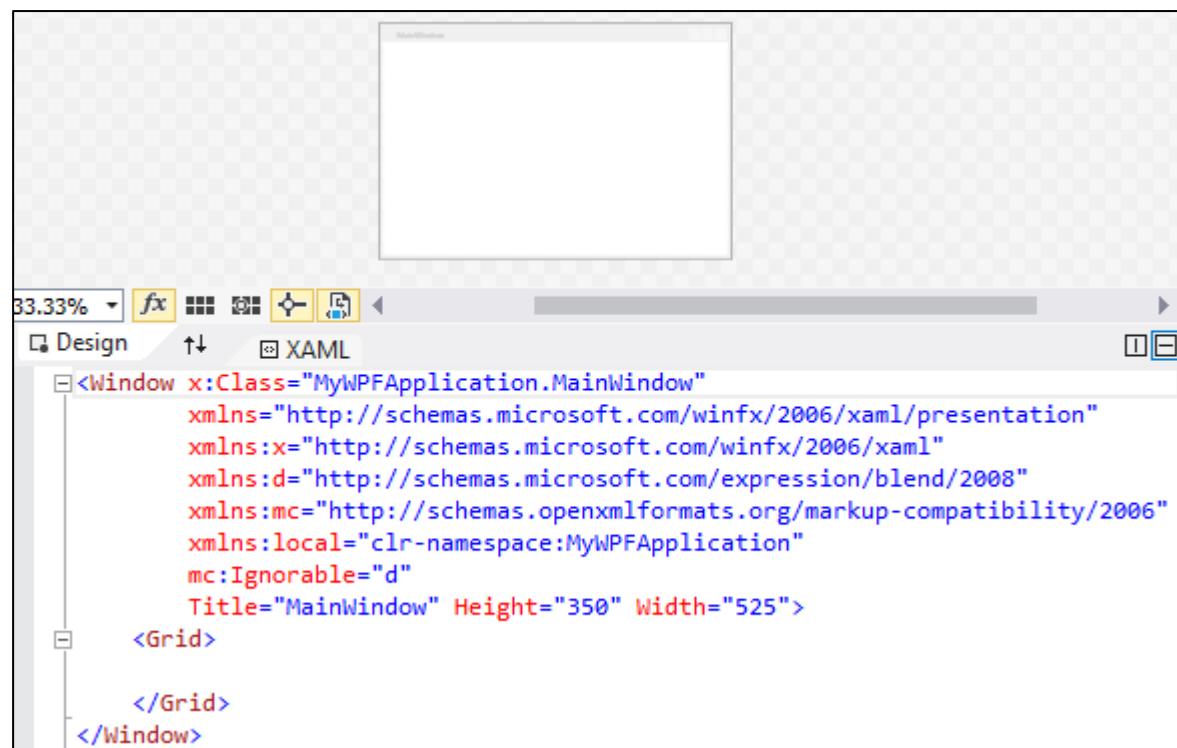


Abb. 3-38 WPF XAML-Designer nach Erstellung eines neuen Projekts mit leerem Fenster

Darunter wurde, wie auf Abb. 3-38 zu sehen, der XAML-Code für ein neues Fenster erstellt. Dieser beinhaltet bereits ein Window, die erforderlichen Namespaces und Eigenschaften und eine Grid, allerdings noch keine weiteren Steuerelemente.

Über die Toolbox auf der linken Seite können nun Steuerelemente, wie Buttons, TextBoxen oder Labels, auf den Designer gezogen werden. Der Designer arbeitet wie ein WYSIWYG-Editor („What You See Is What You Get“). Sollte die XAML-Datei übersichtlich bleiben, ist es ratsam, hauptsächlich mit dem TextEditor und nicht mit dem Designer zu arbeiten. Zudem müssen Bindings und weitere Eigenschaften der Controls ohnehin per Texteditor erstellt bzw. eingestellt werden.

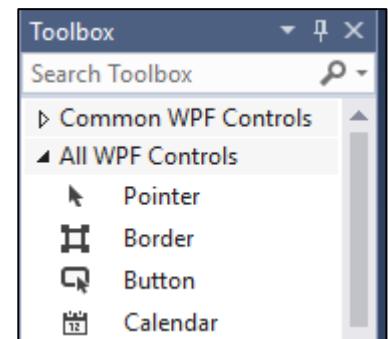


Abb. 3-39 Toolbox des WPF-Designers

Mit der Erstellung des Projekts und der XAML-Datei wurde auch automatisch eine C# Textdatei mit einer partiellen Klasse erstellt. Diese stellt den CodeBehind der View-Klasse dar. Mit einem Klick auf „View Code“ im Kontextmenü kann der CodeBehind geöffnet werden.

```
public partial class MainWindow : Window
{
    0 references
    public MainWindow()
    {
        InitializeComponent();
    }
}
```

Abb. 3-40 CodeBehind eines neuen Windows

Nun können Steuerelemente hinzugefügt, das Layout angepasst, Model-Klassen und ViewModel-Klassen erstellt, DataBindings festgelegt und Funktionen implementiert werden.

3.1.6.2 Verbindung von Code und Design

Die Verbindung von CodeDesign wurde bereits im Kapitel 3.1.5.7 erklärt. Die Trennung der beiden hat ebenfalls viele bereits erläuterte Vorteile. Ein zusätzlicher Vorteil ist, dass Designer der Oberflächen nicht zwingend mit dem in Visual Studio inkludierten WYSIWYG-Designer arbeiten müssen. Mit der Software „Blend for Visual Studio“ können XAML-Oberflächen mit einem eigenen Programm entwickelt werden, das für die Entwicklung hochwertiger, grafisch anspruchsvoller Oberflächen ausgelegt ist.

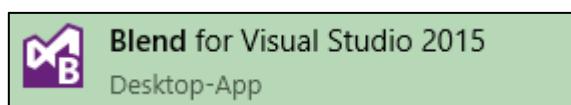


Abb. 3-41 Blend for Visual Studio

3.1.6.3 Entwicklung großer Applikationen

Um bei der Entwicklung großer Anwendungen den Überblick zu behalten, sind ein guter Aufbau und eine gut durchdachte Struktur des Projekts notwendig. Dazu gehört das Aufteilen des Projekts in mehrere Unterprojekte. In Visual Studio wird dies mit dem Verwenden von sogenannten „Solutions“ (deutsch Lösungen) realisiert. Eine Solution kann mehrere Projekte beinhalten, die eigenständig sind und je nach Projekttyp eigenständig kompilierbar sind.

Im .NET Framework werden für die Projekte üblicherweise die Namen des Namespaces verwendet, in dem sich die Klassen befinden. Eine Solution enthält mehrere Projekte vom Typ Klassenbibliothek und ein oder mehrere Projekte von einem ausführbaren Typ (Konsolenanwendung oder WPF Applikation).

Die Businessobjekte (Model-Klassen), also die Logik einer Anwendung werden in Klassenbibliotheken definiert. Durch die Verwendung von hierarchischen Namespaces und eine durchdachte Aufteilung in mehrere Projekte können nicht zur Hierarchie passende Abhängigkeiten der Klassen vermieden werden. Zusätzlich wird die Verwendung von Interfaces notwendig, da zwischen den Projekten keine rekursiven Abhängigkeiten entstehen können.

Jede Kompilation einer Klassenbibliothek ist eine DLL-Datei, die von anderen Klassenbibliotheken bzw. Projekten eingebunden werden können.

Das ausführbare Projekt vom Typ WPF Applikation wird alle für die Ausführung benötigten Klassenbibliotheken inkludieren oder referenzieren. Mit diesem Aufbau einer Anwendung bleibt das Projekt übersichtlich und kann mit hoher Effizienz weiterentwickelt werden.

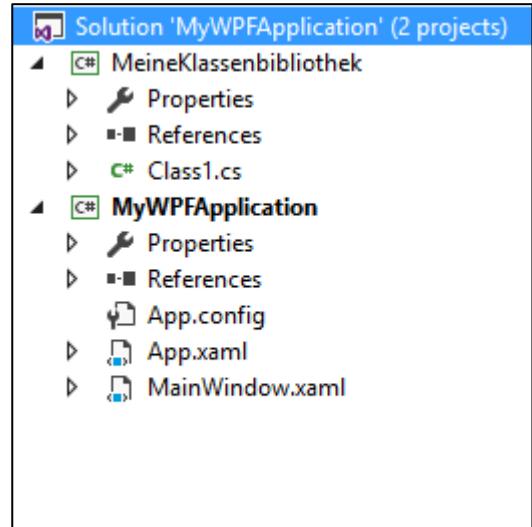


Abb. 3-42 Solution Explorer in Visual Studio

3.2 BMP280 Digital Pressure Sensor

3.2.1 Allgemeine Beschreibung

Der BMP280 ist ein absoluter barometrischer Drucksensor und ein Temperatursensor, der von der Bosch Sensortec GmbH entwickelt und gebaut wurde. Er wurde speziell für Anwendungen in batteriebetriebenen mobilen Geräten entwickelt. Der Sensor ist in einem metallenen Gehäuse mit einem LGA-Package (Land Grid Array-Package) untergebracht, mit einer Grundfläche von 2,0mm × 2,5mm und einer Bauteilhöhe von 0,95mm. Aufgrund des LGA-Package sind die 8 Pins auf der Unterseite für den Lötkolben unerreichbar. Dadurch kann der BMP280 zumeist fertig montiert auf einem Breakoutboard erworben werden.



Abb. 3-43 BMP280 (Quelle: Datenblatt BMP280)

3.2.1.1 Vergleich mit BMP180

Der BMP280 basiert auf dem Vorgängermodell BMP180, das wesentlich weiter entwickelt und verbessert wurde. Der BMP280 arbeitet mit einer wesentlich höheren Auflösung und unterstützt zudem neue Filtermethoden und Schnittstellen. In der folgenden Abb. 3-44 sind jeweils die wichtigsten Eigenschaften des BMP180 und des BMP280 dargestellt.

Eigenschaft	BMP180	BMP280
Grundfläche	3,6mm × 3,8mm	2,0mm × 2,5mm
Minimale Versorgungsspannung	1,80 V	1,71 V
Stromverbrauch bei 1Hz	12µA	2,7µA
Auflösung bei Druckmessung	1 Pa	0,16 Pa
Auflösung bei Temperaturmessung	0,1°C	0,01°C
Schnittstellen	I2C	I2C und SPI
Messmoden	Druck oder Temperatur, Einzelmessung	Druck und Temperatur, Einzelmessung oder periodische Messung
Maximale Messfrequenz	120Hz	157Hz

Abb. 3-44 Vergleich BMP180 mit BMP280 (Quelle: Datenblatt BMP280)

3.2.1.2 Eigenschaften

In der folgenden Abb. 3-45 sind die wichtigsten Eigenschaften für den Betrieb des BMP280 dargestellt.

Eigenschaft	Bedingung	Minimum	Typisch	Maximum
Temperaturmessbereich		0°C		65°C
Druckmessbereich		300hPa		1100hPa
Versorgungsspannung	Welligkeit maximal 50mV	1,71V	1,8V	3,6V
Versorgungsstrom	1 Hz Einzelmessung Druck und Temperatur		2,7µA	4,2µA
Stromspitze	während Messung		720µA	1120µA
Strom im Standbymodus	25°C		0,2µA	0,5µA
Genauigkeit Druckmessung	700hPa bis 900hPa 25°C bis 40°C		±0,12hPa ±1,0m	
Genauigkeit Temperaturmessung	0°C bis 65°C		±1,0°C	
Auflösung bei höchster Auflösungseinstellung	Druck Temperatur		0,0016hPa 0,01°C	

Abb. 3-45 Eigenschaften BMP280 (Quelle: Datenblatt BMP280)

3.2.2 Funktionsbeschreibung

3.2.2.1 Blockschaltbild

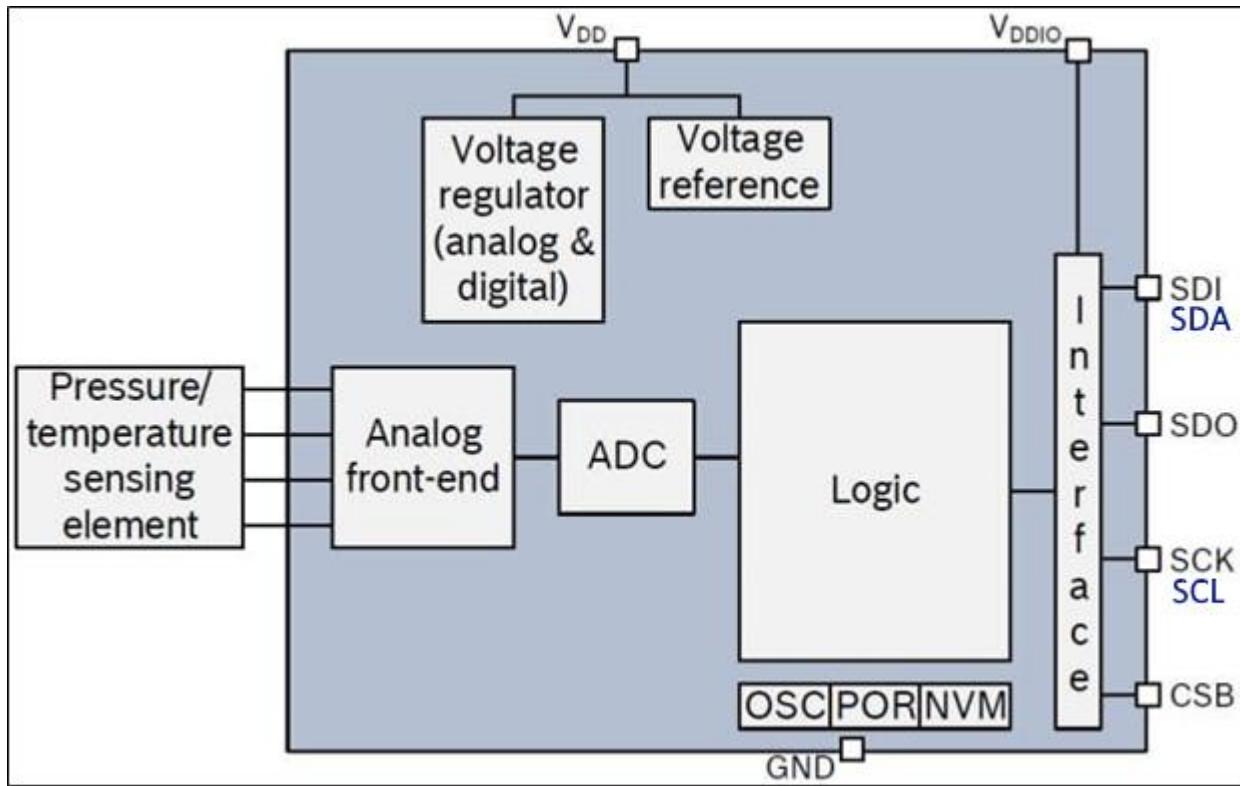


Abb. 3-46 Innerer Aufbau des Drucksensors BMP280 (Quelle: Datenblatt BMP280)

Die Druckmessung des BMP280 erfolgt mittels eines piezoresistiven Drucksensorelements. Die gemessenen analogen Signale werden anschließend mit einem Analog-Front-End (kurz: AFE) bearbeitet. Das AFE besteht aus einem integrierten Schaltkreis, der einen Vorverstärker beinhaltet. Der Vorverstärker reduziert das Signal-Rausch-Verhältnis und verbessert somit die Signalqualität. Die Signale werden nach dem AFE von analog auf digital gewandelt und danach der Logic-Unit zugeführt. Das daraus resultierende Ergebnis kann durch die digitalen Schnittstellen, I²C und SPI, ausgelesen und weiterverwendet werden. Die Logic-Unit, der ADC, das AFE und die digitalen Schnittstellen sind in einer anwendungsspezifischen integrierten Schaltung (englisch application specific integrated circuit, kurz ASIC) integriert. Die im BMP280 eingebaute ASIC ist eine Mixed-Signal-ASIC, da diese aus digitalen sowie analogen Funktionen besteht.

Der digitale Luftdrucksensor besitzt zwei separate Pins für die Spannungsversorgung. Über den Pin V_{DD} werden alle internen analogen und digitalen Funktionsblöcke versorgt. Der Pin V_{DDIO} dient zur separaten Versorgung für die digitalen Schnittstellen. Wird nur der V_{DDIO} Pin versorgt, so werden die Schnittstellenpins (SDI, SDO, SCK und CSB) auf einem high-Z Pegel gelegt, dadurch kann der Datenbus bereits ohne Versorgung von V_{DD} verwendet werden. Nach der Einschaltsequenz setzt der eingebaute Rücksetzgenerator die Logikschatzung sowie die Werte der Register zurück. Nachdem Rücksetzen ist der BMP280 betriebsbereit. Bei der Erhöhung beziehungsweise beim Anlegen der Spannungspiegel von V_{DDIO} und V_{DD} gibt es keine Begrenzung in der Steilheit sowie bei der Reihenfolge.

3.2.3 Messzyklus

Jede Messperiode besteht aus einer Temperatur- und Druckmessung mit einer auswählbaren Überabtastung. Nach der Messperiode werden die Daten durch einen optional auswählbaren IIR-Filter geschickt, der kurzzeitige Beeinflussungen bei der Druckmessung entfernt. In der folgenden Abb. 3-47 ist der Messzyklus dargestellt.

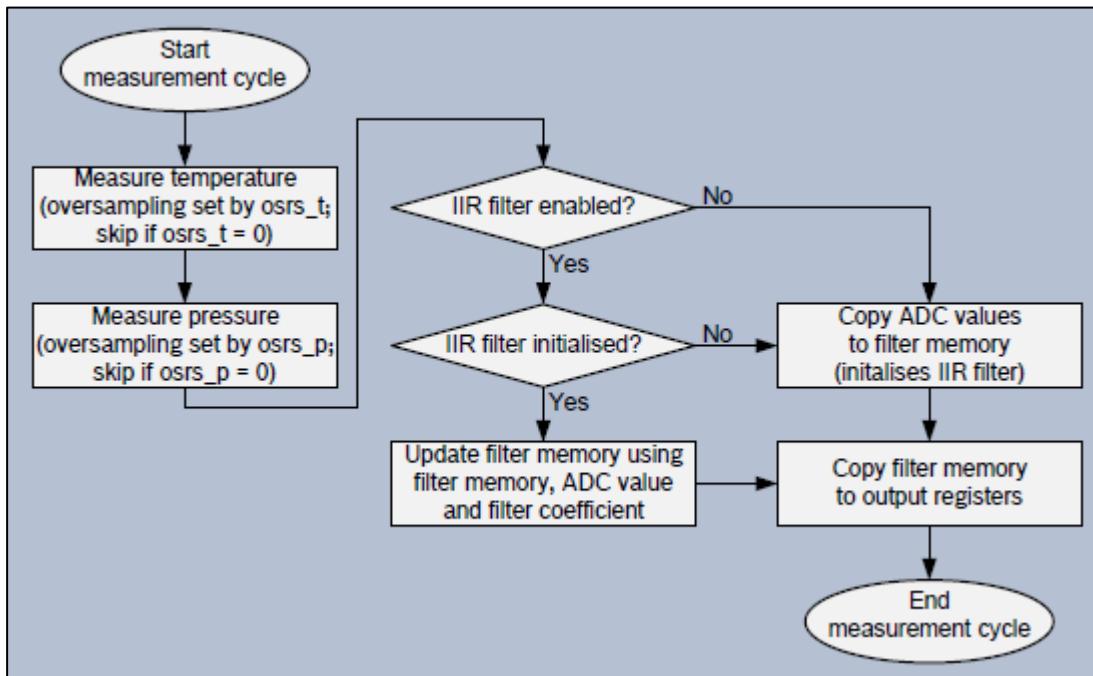


Abb. 3-47 Messzyklus des Drucksensors (Quelle: Datenblatt BMP280)

Die einzelnen Blöcke des Diagramms werden in den folgenden Unterpunkten genauer erklärt.

3.2.3.1 Druckmessung

Die Druckmessung des BMP280 erfolgt mittels piezoresistiven Effekt. Beim piezoresistiven Effekt ändert sich der elektrische Widerstand eines Materials durch Druck oder Zug. Diese Widerstandsänderung ist bei Halbleitern besonders stark ausgeprägt, deshalb werden auch überwiegend Silizium-Drucksensoren, wie beispielsweise der BMP280, hergestellt. Der Drucksensor enthält eine Membran mit aufgebrachten elektrischen Widerständen. Bei einer Verformung der Membran durch Druck, entsteht daraus eine Widerstandsänderung, die gemessen wird. Daraus kann nun der Druck berechnet werden. Derartige Drucksensoren weisen hohe Empfindlichkeit auf und sind kostengünstig in der Produktion. Die Materialien, die zur Druckmessung eingesetzt werden, zeigen eine starke Temperaturabhängigkeit. Dieser Nachteil kann mithilfe einer differenz-bildenden elektrischen Schaltung korrigiert werden, da der Temperatureinfluss auf alle Widerstände gleich einwirkt. Der digitale Luftdrucksensor BMP280 wird von der Firma Bosch mit deren geschütztem APSM-Verfahren (Advanced Porous Silicon Membrane) zur Herstellung von MEMS (Mikro-Elektromechanisches System), hergestellt. [19], [20]

3.2.3.2 Digitale Filter

Durch kurzzeitige äußere Einflüsse wird die Druckmessung oftmals gestört (Zuschlagen einer Tür). Der digitale Filter versucht Werte, die bei der Störung gemessen wurden, herauszufiltern und zu unterdrücken. Hierzu wird bei dem BMP280 ein interner IIR (Infinite Impulse Response) Filter verwendet. Die Impulsantwort eines IIR Filters dehnt sich unendlich lange über die Zeitachse aus (unendliche Sprungantwort). Jeder gefilterte Ausgangswert ist vom Eingangswert sowie von den vorausgegangenen Filterausgangswerten abhängig. Der Ausgangswert für den nächsten Messschritt lässt sich mit folgender Formel, wie in Abb. 3-48 dargestellt, beschreiben.[21], [22]

$$\text{data_filtered} = \frac{\text{data_filtered_old} \cdot (\text{filter_coefficient} - 1) + \text{data_ADC}}{\text{filter_coefficient}}$$

Abb. 3-48 IIR Filter Formel (Quelle: Datenblatt BMP280)

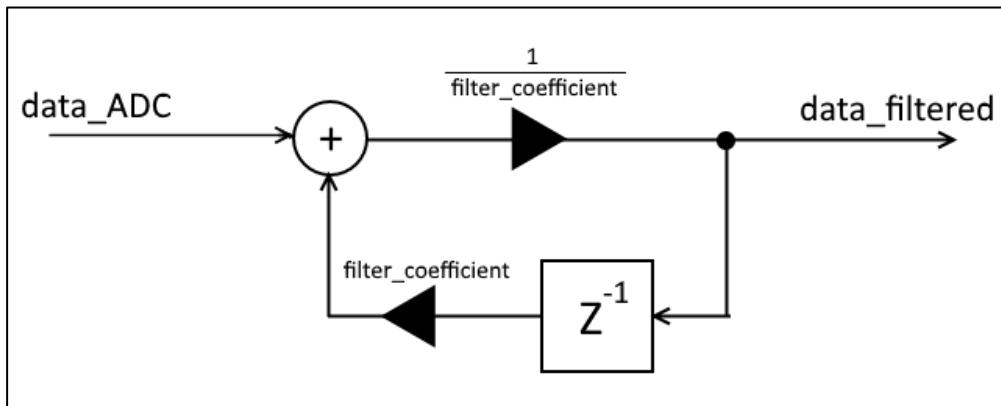


Abb. 3-49 Rekursive Filterdarstellung

3.2.4 Zeitliche Rahmenbedingungen

Der BMP280 bietet drei verschiedene Funktionsweisen, die in den folgenden drei Unterpunkten genauer erklärt werden.

3.2.4.1 Sleep mode

Im sleep mode werden keine Messungen durchgeführt und der Stromverbrauch ist auf einem Minimum in einem Wertebereich von $0,1\mu\text{A}$ bis $0,3\mu\text{A}$, abhängig vom Wert der Versorgungsspannung. Dieser Stromverbrauch ist in Abb. 3-50 als I_{DDSL} eingezeichnet. Alle Register sind zugänglich und die Chip-ID kann ausgelesen werden.

3.2.4.2 Forced mode

Im forced mode wird eine einzelne Messung mit ausgewählten Mess- und Filtereinstellungen durchgeführt. Nach Beendigung der Messung kehrt der Sensor in den sleep mode zurück und das Messergebnis kann aus dem Dataregister ausgelesen werden. Für eine erneute Messung muss wiederum der forced mode ausgewählt werden. In der Abb. 3-50 ist das dazugehörige Zeitdiagramm abgebildet.

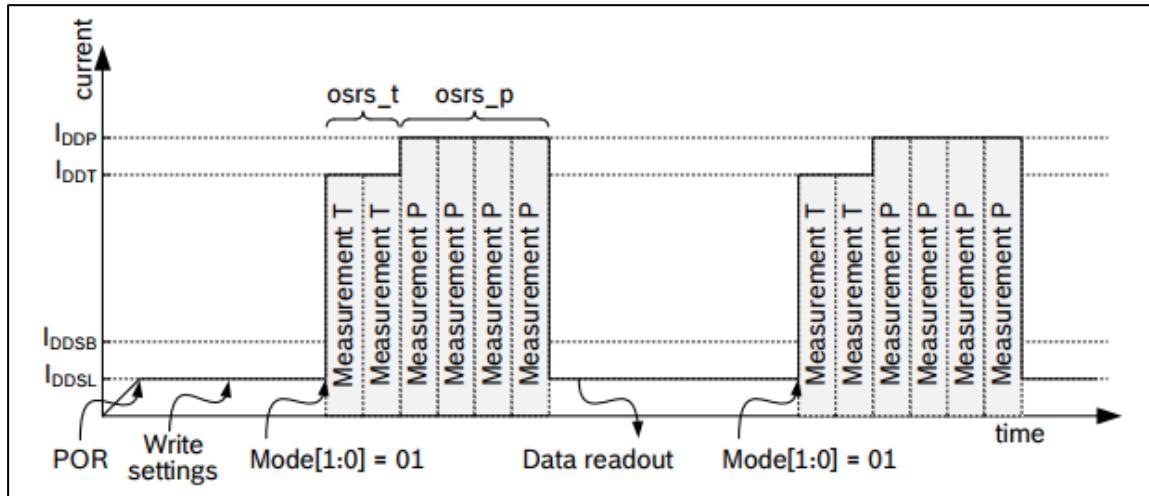


Abb. 3-50 Forced Mode Zeitdiagramm (Quelle: Datenblatt BMP280)

3.2.4.3 Normal mode

Im normal mode wird kontinuierlich ein Zyklus, der aus einer aktiven Messzeit und einer inaktiven Standby-Zeit besteht, durchgeführt. Die Standby-Zeit kann eingestellt werden und wird durch die Variable $t_{standby}$ beschrieben. Der Stromverbrauch im Standby-Modus ist ein klein wenig höher als im sleep mode. Nach dem Einstellen der Mess- und Filtereinstellungen können die Messergebnisse ausgelesen werden.

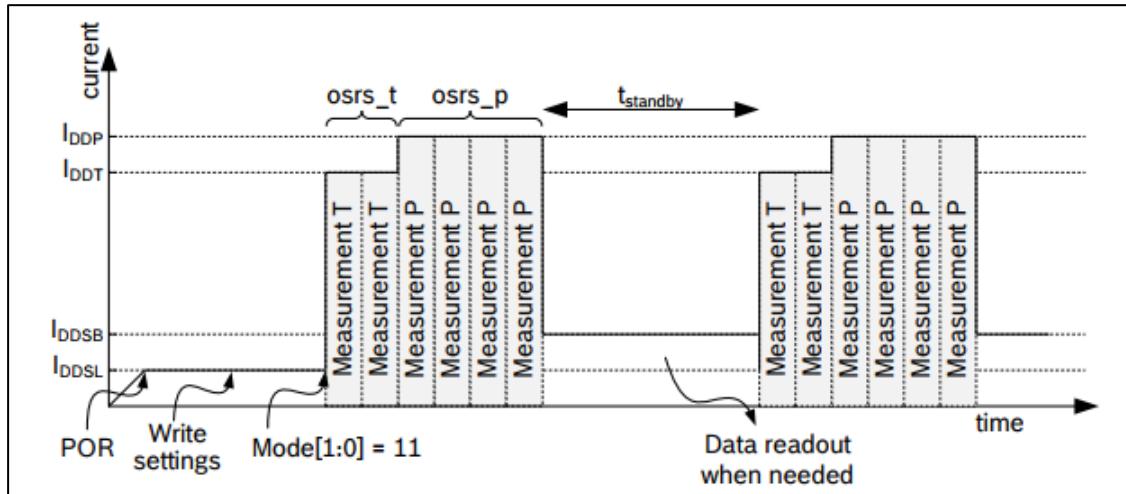


Abb. 3-51 Normal mode Zeitdiagramm (Quelle: Datenblatt BMP280)

3.2.4.4 Periodendauer und Abtastfrequenz

Die Periodendauer beziehungsweise die Abtastfrequenz sind von der gewählten Überabtastung abhängig. Die Überabtastung kann jeweils für Druckmessungen beziehungsweise Temperaturmessungen ausgewählt werden. In der folgenden Abb. 3-52 sind die Periodendauer und Abtastfrequenz für einen einzelnen Messzyklus zu den jeweiligen Überabtastungseinstellungen dargestellt:

Überabtastung Einstellung	Druck Überabtastung	Temperatur Überabtastung	Periodendauer[ms]		Abtastfrequenz[Hz]	
			Typ	Max	Typ	Min
Ultra low power	×1	×1	5,5	6,4	181,8	155,6
Low power	×2	×1	7,5	8,7	133,3	114,6
Standard	×4	×1	11,5	13,3	87,0	75,0
High Power	×8	×1	19,5	22,5	51,3	44,4
Ultra high power	×16	×2	37,5	43,2	26,7	23,1

Abb. 3-52 Periodendauer und Abtastfrequenz (Quelle: Datenblatt BMP280)

Bei der Überabtastung wird ein Signal mit einer weit höheren Frequenz, als vom Abtasttheorem gefordert, abgetastet. Das Abtasttheorem besagt, dass ein Signal mindestens mit der doppelten Frequenz des Signals abgetastet werden muss. Durch die Überabtastung wird sowohl der Signal-Rauschabstand als auch die Linearität bei Analog-Digital-Wandler verbessert. Bei der Projektumsetzung wurde die Einstellung für die Überabtastung „Ultra high power“ ausgewählt und verwendet, da dadurch die höchstmögliche Genauigkeit erreicht wird.

In der folgenden Abb. 3-53 ist die Ausgangsdatenrate für den normal mode zu den jeweiligen Überabtastungseinstellungen und der jeweiligen Standby-Zeit t_{standby} abgebildet.

Überabtastung Einstellung	$T_{\text{standby}} [\text{ms}]$							
	0,5	62,5	125	250	500	1000	2000	4000
Ultra low power	166,67	14,71	7,66	3,91	1,98	0,99	0,5	0,25
Low power	125,00	14,29	7,55	3,88	1,97	0,99	0,5	0,25
Standard	83,33	13,51	7,33	3,82	1,96	0,99	0,5	0,25
High Power	50,00	12,20	6,92	3,71	1,92	0,98	0,5	0,25
Ultra high power	26,32	10,00	6,15	3,48	1,86	0,96	0,49	0,25

Abb. 3-53 Datenrate bei ausgewählter T_{standby} (Quelle: Datenblatt BMP280)

3.2.5 Memory Map des Bausteins

Die komplette Kommunikation des BMP280 wird mithilfe von Registern durchgeführt. Register sind ein Speicherbereich in einem Prozessor, wo Ergebnisse von Operationen oder zu verarbeitenden Daten abgelegt werden. Die Breite eines Registers ist üblicherweise eine Zweierpotenz und wird in Bit angegeben. Beim BMP280 beträgt die Registerbreite 8-Bit. Eine gewisse Anzahl an Registern ist für spezielle Aufgaben reserviert. Diese können weder beschrieben noch ausgelesen werden. Die Memory Map bietet hierbei einen groben Überblick der wichtigsten Register mit deren Adressen und Eigenschaften. In der folgenden Abb. 3-54 ist die Memory Map des BMP280 dargestellt.

Register Name	Address	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	Reset state				
temp_xlsb	0xFC	temp_xlsb<7:4>				0	0	0	0	0x00				
temp_lsb	0xFB	temp_lsb<7:0>												
temp_msb	0xFA	temp_msb<7:0>												
press_xlsb	0xF9	press_xlsb<7:4>				0	0	0	0	0x00				
press_lsb	0xF8	press_lsb<7:0>												
press_msb	0xF7	press_msb<7:0>												
config	0xF5	t_sb[2:0]		filter[2:0]			spi3w_en[0]		0x00					
ctrl_meas	0xF4	osrs_t[2:0]		osrs_p[2:0]			mode[1:0]		0x00					
status	0xF3	measuring[0]				im_update[0]								
reset	0xE0	reset[7:0]												
id	0xD0	chip_id[7:0]												
calib25...calib00	0xA1...0x88	calibration data												

Registers:	Reserved registers	Calibration data	Control registers	Data registers	Status registers	Revision	Reset
Type:	do not write	read only	read / write	read only	read only	read only	write only

Abb. 3-54 Memory Map (Quelle: Datenblatt BMP280)

Die wichtigsten Register werden in den folgenden Unterpunkten genauer erklärt.

3.2.5.1 *id*-Register 0xD0

Das id-Register beinhaltet die Identifikationsnummer des Chips, die in chip_id [7:0] hinterlegt ist und den Wert 0x58 besitzt. Die Identifikationsnummer des Chips kann, sobald der Luftdrucksensor die Einschaltsequenz abgeschlossen hat, ausgelesen werden. Das id-Register ist unter der Adresse 0xD0 erreichbar und es besteht ausschließlich die Möglichkeit des Auslesens (read only).

3.2.5.2 *reset*-Register 0xE0

Das reset-Register beinhaltet das Reset-Wort reset[7:0]. Wird der Wert 0xB6 auf das Register geschrieben, so wird die Einschaltsequenz nochmals gestartet und ein Reset wird durchgeführt. Ein Schreiben eines anderen Wertes als 0xB6 erreicht keine Wirkung für den User. Das reset-Register kann unter der Adresse 0xE0 erreicht werden. Es kann zwar ausgelesen werden, jedoch wird immer der Wert 0x00 zurückgegeben.

3.2.5.3 *status*-Register 0xF3

Das status-Register beinhaltet zwei Bits, die den aktuellen Status des BMP280 anzeigen. Bit 3, auch bekannt als measuring[0], wird bei einer aktiven Messwandlung auf den Wert 1 gesetzt. Sobald das Messergebnis in das Data-Register abgelegt wurde, wird der Wert des Bits 3 auf 0 gesetzt. Bit 0, auch bekannt als im_update[0], wird auf den Wert 1 gesetzt sobald Daten aus dem nicht flüchtigen Speicher in das Abbildungsregister kopiert werden. Nach dem Kopieren wird der Wert des Bits 0 sofort auf den Wert 0 gesetzt. Dieser Kopiervorgang wird bei jeder Einschaltsequenz sowie vor jeder Messwandlung durchgeführt. Die Werte des status-Registers, das unter der Adresse 0xF3 erreichbar ist, können ausschließlich ausgelesen werden (read only).

3.2.5.4 *ctrl_meas*-Register 0xF4

Im ctrl_meas-Register werden die Einstellungen für die Druckmessungen beziehungsweise Temperaturmessungen gesetzt. Mit den Bits 7,6 und 5, auch bekannt als osrs_t[2:0], werden

die Einstellungen für die Überabtastung der Temperatur festgelegt. Mit den Bits 4,3 und 2, auch bekannt als osrs_p[2:0], werden die Einstellungen für die Überabtastung vorgenommen. In der folgenden Abb. 3-55 sind die diversen Einstellungsmöglichkeiten für osrs_p[2:0] und osrs_t[2:0] abgebildet.

osrs_p[2:0]	Druck Überabtastung	Typische Auflösung	Überabtastung Einstellungen	osrs_t[2:0]	Temperatur Überabtastung	Typische Auflösung
000	Skipped	-	No Measurement	000	Skipped	-
001	$\times 1$	16 Bit 2,62 Pa	Ultra low power	001	$\times 1$	16 Bit 0,0050 °C
010	$\times 2$	17 Bit 1,31 Pa	Low power	010	$\times 2$	17 Bit 0,0025°C
011	$\times 4$	18 Bit 0,66 Pa	Standard	011	$\times 4$	18 Bit 0,0012°C
100	$\times 8$	19 Bit 0,33 Pa	High Power	100	$\times 8$	19 Bit 0,0006°C
101,Others	$\times 16$	20 Bit 0,16 Pa	Ultra high power	101,Others	$\times 16$	20 Bit 0,0003°C

Abb. 3-55 Einstellungsmöglichkeiten Überabtastung (Quelle: Datenblatt BMP280)

Mit den Bits 1 und 0, auch bekannt als mode[1:0], wird die Funktionsweise des Luftdrucksensors ausgewählt. In folgender Abb. 3-56 sind die Werte für mode[1:0] für die jeweilige Funktionsweise dargestellt. Die Funktionsweisen werden in 3.2.4 genauer erklärt

mode[1:0]	Mode
00	sleep mode
01 und 10	forced mode
11	normal mode

Abb. 3-56 Einstellungsmöglichkeiten Funktionsweise (Quelle: Datenblatt BMP280)

Das ctrl_meas Register kann unter der Adresse 0xF4 erreicht werden. Es kann sowohl ausgelesen als auch beschrieben werden. (read and write)

3.2.5.5 config-Register 0xF5

Im config-Register werden die Bits für die Standby-Zeit, die Filterkoeffizienten und die Schnittstellenoptionen gesetzt. Mit den Bits 7, 6 und 5, auch bekannt als t_sb[2:0], stellt man die Standby-Zeit für den normal mode ein. Die Bits 4, 3 und 2, auch bekannt als filter[2:0], dienen zum Einstellen des Filterkoeffizienten für den IIR-Filter. Mit dem Bit 0, auch bekannt als spi3w_en[0], kann zwischen der 3-wire und der 4-wire SPI-Schnittstelle (SDO und SDI werden auf SDI zusammengeführt) aktiviert. Die nachfolgenden Abb. 3-57 und Abb. 3-58 zeigen die jeweils verschiedenen Einstellungsmöglichkeiten für t_sb[2:0] und filter[2:0].

Filterkoeffizient Filter[2:0]
Filter off
2
4
8
16

Abb. 3-57 Einstellungsmöglichkeiten Filterkoeffizient (Quelle: Datenblatt BMP280)

t_sb[1:0]	Tstandby[ms]
000	0,5
001	62,5
010	125
011	250
100	500
101	1000
110	2000
111	4000

Abb. 3-58 Einstellungsmöglichkeit Standby-Zeit (Quelle: Datenblatt BMP280)

Beim Schreiben eines Wertes zu Filter[2:0] wird der Filter zurückgesetzt. Der nächste Messwert wird hierbei ohne Filterung durchgelassen und als Ausgangswert für die Filterung verwendet. Das config-Register erreicht man unter der Adresse 0xF5. Sollte das Register beschrieben werden, wenn sich der BMP280 im normal mode befindet, so kann es passieren, dass dies ignoriert wird. Darum sollte das config-Register vorzugsweise beschrieben werden, wenn sich der Luftdrucksensor im sleep mode befindet.

3.2.5.6 Press-Register 0xF7, 0xF8 und 0xF9

Die Register press_msb, press_lsb und press_xlsb ergeben das sogenannte press-Register, das die gemessenen Druckwerte beinhaltet. Das Register mit der Adresse 0xF7, auch bekannt als press_msb[7:0], beinhaltet die MSB (Most Significant Bits) der gemessenen Druckwerte. Die LSB (Least Significant Bits) der gemessenen Druckwerte werden im Register mit der Adresse 0xF8, auch bekannt als press_lsb[7:0], gespeichert. Je nach eingestellter Auflösung für die Druckmessung werden in den Bits 7, 6, 5 und 4 des Registers mit der Adresse 0xF9, auch bekannt als press_xlsb, ebenfalls Druckwerte gespeichert. Die Daten aus diesen Registern sollten vorzugsweise mit dem Burst-Modus ausgelesen werden, denn ansonsten kann es passieren, dass Daten von zwei verschiedenen Messungen vermischt werden.

Bei der Projektumsetzung wurde, wie in Kapitel 3.2.5.4 beschrieben, die Einstellung „Ultra high Power“ verwendet. Daraus resultierend ergibt sich eine Auflösung von 20 Bit beziehungsweise 0,16 Pa. Damit ein gemessener Wert vollständig in der Einheit Pascal (kurz Pa) gespeichert werden kann, werden alle verfügbaren Bits des Press-Registers benötigt. Der Messwert muss sich zudem in einem Bereich zwischen 300hPa und 1100hPa befinden. [23]

3.2.6 Digitale Schnittstellen

Der BMP280 unterstützt zwei verschiedene digitale Schnittstellen, I²C und SPI. Beide unterstützen das Schreiben und Lesen von einzelnen Bytes sowie von mehreren Bytes. Die I²C Schnittstelle bietet den Standard-, Fast- und High Speed Modus. Die SPI Schnittstelle bietet die 3-wire als auch die 4-wire Konfiguration. Bei der Projektumsetzung wurde die I²C Schnittstelle verwendet.

3.2.6.1 I²C Schnittstelle

Der von der Phillips Semiconductors in den 1980er entwickelte I²C-Bus ist ein serieller synchroner Datenbus. Der Inter Integrated Circuit sollte ursprünglich die Kommunikation zwischen digitalen Bauteilen auf einer Leiterplatte erleichtern. Daraus entstand der bidirektionale Zweidraht-Bus I²C, auch Two-Wire-Interface genannt. Die zwei Datenleitungen sind zum einen die Takteleitung SCL (Serial Clock Line) und zum anderen die Datenleitung SDA (Serial Data Line). Die beiden Leitungen werden mittels Pullup-Widerstände auf die Versorgungsspannung gelegt. Alle verbundenen Geräte müssen einen open-collector oder open-drain Ausgang besitzen, der bei einer aktiven Leitung auf LOW gezogen wird. Diese Schaltung nennt man dann Wired-AND. Eine solche Schaltung mit einem Master und drei Slaves ist in der Abb. 3-59 abgebildet.

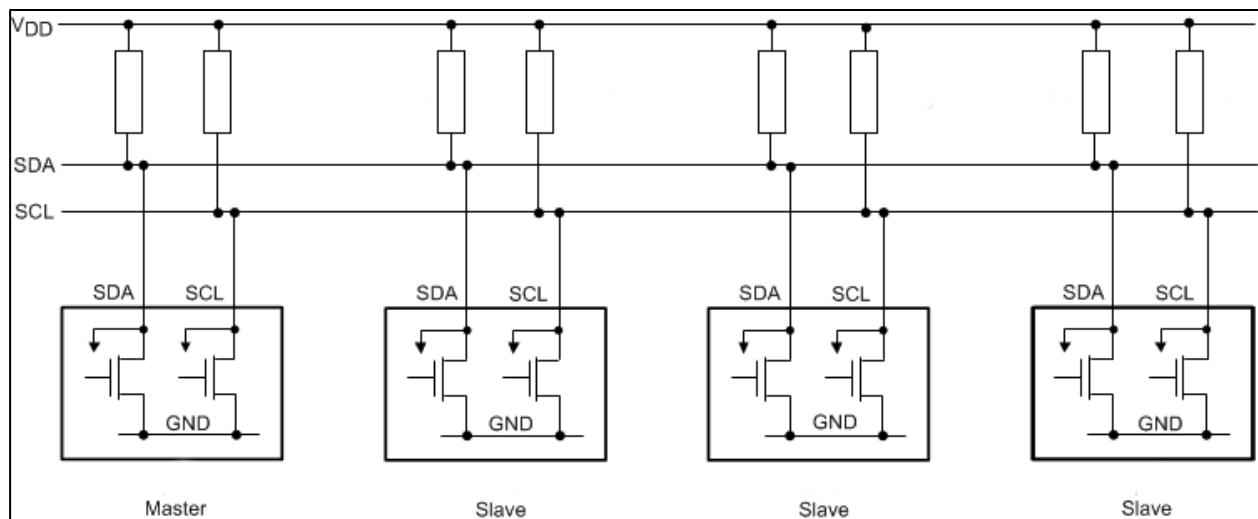


Abb. 3-59 Wired-AND Schaltung mit einem Master und drei Slaves

3.2.6.1.1 Eigenschaften I²C

Der I²C-Bus ist für kurze geräteinterne Übertragungsstrecken geeignet. Darum wird diese Art von Bus vorzugsweise in Audio-, Video-, Druck- und Temperaturanwendungen verwendet. Die in den 1980ern ursprünglich festgelegte Datenrate von 100 kBit/s, bei einer Buskapazität von 400 pF, reichte schon kurze Zeit später nicht mehr aus und so wurde 1992 der schnellere Fast-Mode eingeführt. Dieser bot eine Datenrate von 400 kBit/s. 1998 wurde der High Speed Mode eingeführt, der eine Datenrate von 3,4 Mbit/s bei einer Buskapazität von 400 pF ermöglicht. Mittlerweile ist es dank neuer Expansions- und Steuerungselementen möglich, den I²C-Bus über die 400-pF Grenze hinaus zu erweitern. Dadurch kann eine Vielzahl von Teilnehmern an den I²C-Bus angeschlossen werden. Ein wesentlicher Aspekt ist die relativ einfache Ansteuerung, denn

es müssen keine festen Taktzeiten eingehalten werden. Dies ermöglicht die Teilnahme von langsamen als auch von sehr schnellen Busteilnehmern. Die Kommunikation findet immer zwischen einem Master und einem oder mehreren Slaves statt. Auch ein Multimaster-Mode Betrieb ist möglich. Hierbei können zwei Master miteinander kommunizieren, dabei arbeitet jeweils ein Master als Slave.

3.2.6.1.2 Startbedingung

Zu Beginn einer Kommunikation erzeugt der Master eine Startbedingung. Dadurch werden alle Busteilnehmer darüber informiert dass eine Datenübertragung ansteht. Zum Erzeugen der Startbedingung wechselt SDA von einem HIGH-Pegel auf einen LOW-Pegel, während sich SCL auf einem HIGH-Pegel befindet. Der Bus ist nun als „beschäftigt“ gekennzeichnet.

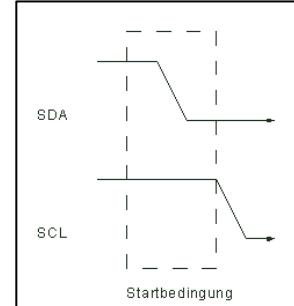


Abb. 3-60 Startbedingung
(Quelle: rn-wissen.de)

3.2.6.1.3 Adressierung

Nach der Startbedingung versendet der Master die Adresse des Slaves, den er ansprechen möchte. Jeder Teilnehmer am I²C-Bus benötigt eine Adresse. Der I²C-Bus verfügt über zwei verschiedene Adressierungsformen, die 7-Bit und 10-Bit Adressierung. Die 7-Bit Adressierung ermöglicht grundsätzlich $2^7 = 128$ Busteilnehmer. Jedoch ist zu beachten, dass durch die Reservierung einiger Adressen die Anzahl der Teilnehmer am Bus auf 112 begrenzt ist. Die 7-Bit Adressierung ist, wie in Abb. 3-61 zu sehen ist, aufgebaut.

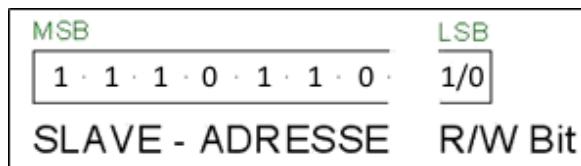


Abb. 3-61 Adressierungsbyte (Quelle: rn-wissen.de)

Das Adress-Byte besteht aus einer 7-Bit großen Adresse (beim BMP280 0x76), die der Slave besitzt, den der Master ansprechen möchte und einem R/W Bit. Mit dem R/W Bit bestimmt der Master, ob er Daten empfangen oder an den Slave versenden möchte. Der Wert 0 beim R/W Bit steht für schreiben und der Wert 1 für lesen. Wenn ein Slave seine Adresse identifizieren konnte, dann sendet dieser eine Bestätigung, das sogenannte ACK-Bit (Acknowledge Bit).

3.2.6.1.4 Acknowledge

Nach Erkennen seiner Slave Adresse beziehungsweise nach jedem geschriebenen Datenbyte sendet der Slave ein Acknowledge. Dies dient zur Bestätigung, ob das vom Master gesendete Datenbyte erfolgreich beim Slave angekommen ist. Der Master kann nach dem Erhalt des ACK-Bits die Datenübertragung fortsetzen. Der Master sendet ebenfalls nach jedem gelesenen Datenbyte ein ACK-Bit, damit der Slave die Bestätigung bekommt, dass der Master bereit ist, weitere Daten zu empfangen. Wie in Abb. 3-62 zu sehen ist, wird beim Acknowledge SDA auf einem LOW-Pegel und SCL auf einem HIGH-Pegel gehalten.

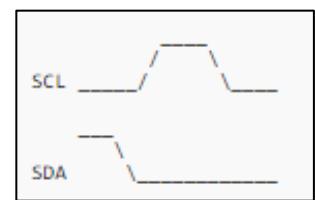


Abb. 3-62 Acknowledge
(Quelle: cc-zwei.de)

3.2.6.1.5 Not Acknowledge

Nach dem Lesen des letzten Datenbytes, bei einer Lesesequenz des Masters, sendet der Master dem Slave ein sogenanntes NACK-Bit (Not Acknowledge). Damit signalisiert der Master, dass er keine weiteren Daten mehr lesen möchte. In Abb. 3-63 sieht man ein NACK-Bit. Dabei wird SDA auf einem LOW-Pegel und SCL auf einem HIGH-Pegel gehalten.



Abb. 3-63 Not Acknowledge
(Quelle: cc-zwei.de)

3.2.6.1.6 Datenübertragung

Nach erfolgreicher Startbedingung, Adressierung und Acknowledge kann die Datenübertragung gestartet werden. Damit ein Bit als gültig erkannt werden kann, muss sich SCL auf einem HIGH-Pegel befinden und währenddessen darf sich der Zustand von SDA nicht verändern. (ausgenommen Start- oder Stoppbedingung). Der Pegel von SDA darf sich bei der Datenübertragung nur ändern, währenddessen sich SCL auf einem LOW-Pegel befindet. Diese beiden möglichen Fälle sind in Abb. 3-64 ersichtlich.

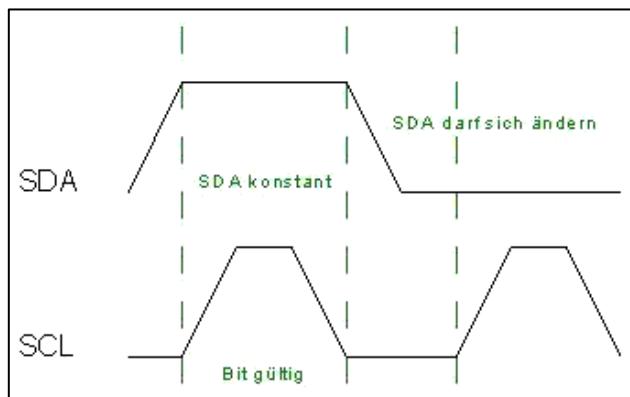


Abb. 3-64 Datenübertragung (Quelle: rn-wissen.de)

3.2.6.1.7 Stoppbedingung

Die Datenübertragung wird mittels einer Stoppbedingung, die der Master erzeugt, beendet. Dadurch werden alle Busteilnehmer darüber informiert dass die Datenübertragung beendet wurde. Zum Erzeugen der Stoppbedingung wechselt SDA von einem LOW-Pegel auf einen HIGH-Pegel, während sich SCL auf einem HIGH-Pegel befindet. Der Bus ist wird als „frei“ gekennzeichnet.

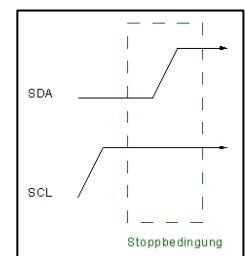


Abb. 3-65 Stoppbedingung
(Quelle: rn-wissen.de)

3.2.6.1.8 Repeated-Startbedingung

Durch die Stoppbedingung wird der Bus wieder „frei“ gegeben und ein anderer Master kann den Bus übernehmen. Gelegentlich kann es vorkommen, dass der Master noch mit einem zweiten Slave kommunizieren möchte und bereits ein anderer Master den Bus übernommen hat. Um dies zu verhindern, kann eine Repeated-Startbedingung gesendet werden. Der aktive Master sendet keine Stoppbedingung, sondern eine erneute Startbedingung, dadurch kann der Master einen anderen Slave ansprechen.[24]–[27]

3.2.6.1.9 Datenübertragung I²C write

In der folgenden Abb. 3-66 sieht man die Datenübertragung im Schreibmodus des BMP280.



Abb. 3-66 BMP280-Datenübertragung I²C write (Quelle: Datenblatt BMP280)

Der BMP280 wird hierbei mit der Slave Adresse 11101100 (0x76) angesprochen. Diese inkludiert auch den Wert 0 (Schreiben) für das R/W-Bit. Danach sendet der Master die Registeradressen, die angesprochen werden möchten, mit den jeweilig dazugehörigen Registerdaten. Die Übertragung wird mit der Stoppbedingung beendet.

3.2.6.1.10 Datenübertragung I²C read

In der folgenden Abb. 3-67 sieht man die Datenübertragung im Lesemodus des BMP280.

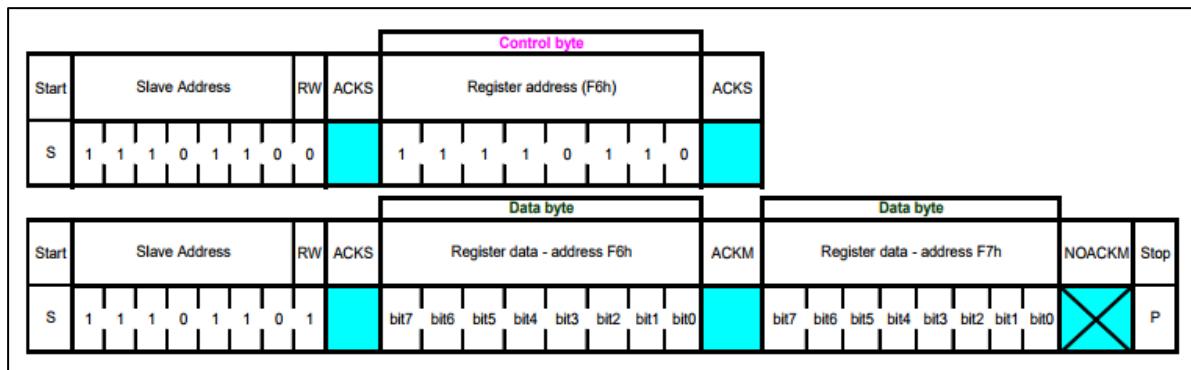


Abb. 3-67 BMP280-Datenübertragung I²C read (Quelle: Datenblatt BMP280)

Der BMP280 wird zuerst mit der Slave Adresse (0x76) angesprochen. Diese inkludiert auch den Wert 0 (Schreiben) für das R/W-Bit. Danach sendet der Master die Registeradresse, die gelesen werden sollte. Nachdem der Slave das ACK-Bit gesendet hat, erzeugt der Master entweder eine Repeated-Startbedingung oder eine Stoppbedingung. Anschließend wird der Slave mit der Adresse 11101101 (0x77) angesprochen. Das R/W-Bit hat nun den Wert 1 (Lesen) zugewiesen bekommen. Der Slave sendet die Registerdaten, springt automatisch ein Register höher und sendet wiederum die Registerdaten. Dies geschieht solange, bis der Master das NACK-Bit sendet (in Abb. 3-67 NOACKM genannt). Die Übertragung wird mit der Stoppbedingung beendet.[23]

3.3 Prototypingphase Drucksensor

Um die Funktion des bestellten Luftdrucksensors BMP280 nachweisen zu können, wurde eine Prototypenschaltung auf einem Versuchssteckbrett aufgebaut. Dieser Aufbau ist in Abb. 3-68 ersichtlich. Es wurde ein Arduino-Programm programmiert, das Messwerte vom BMP280 ausliest. Der erste ausgelesene Wert wurde als Referenzdruckwert gespeichert. Darauffolgend wurde die Differenz zwischen den nachfolgenden gemessenen Druckwerten und dem Referenzdruckwert gebildet und über den seriellen Monitor ausgegeben. Das in Abb. 3-69 ersichtliche Diagramm wurde hierbei aufgezeichnet. Wie in dem Diagramm ersichtlich, ist die Druckdifferenz im Zeitbereich 400ms bis 700ms stark erhöht. Diese Erhöhung wurde durch eine Drucksimulation (Luft wird auf Sensor geblasen) erzeugt. Mithilfe dieser Versuche konnte auch im späteren Verlauf des Projektes die minimal benötigte Druckdifferenz bestimmt werden. (siehe Kapitel 4.1.3)

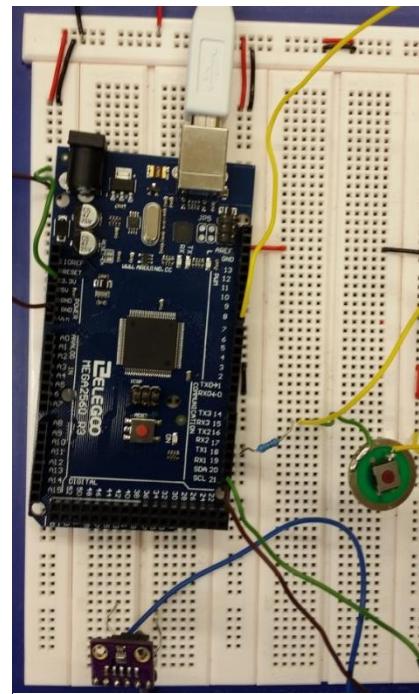


Abb. 3-68 Aufbau Prototypenschaltung

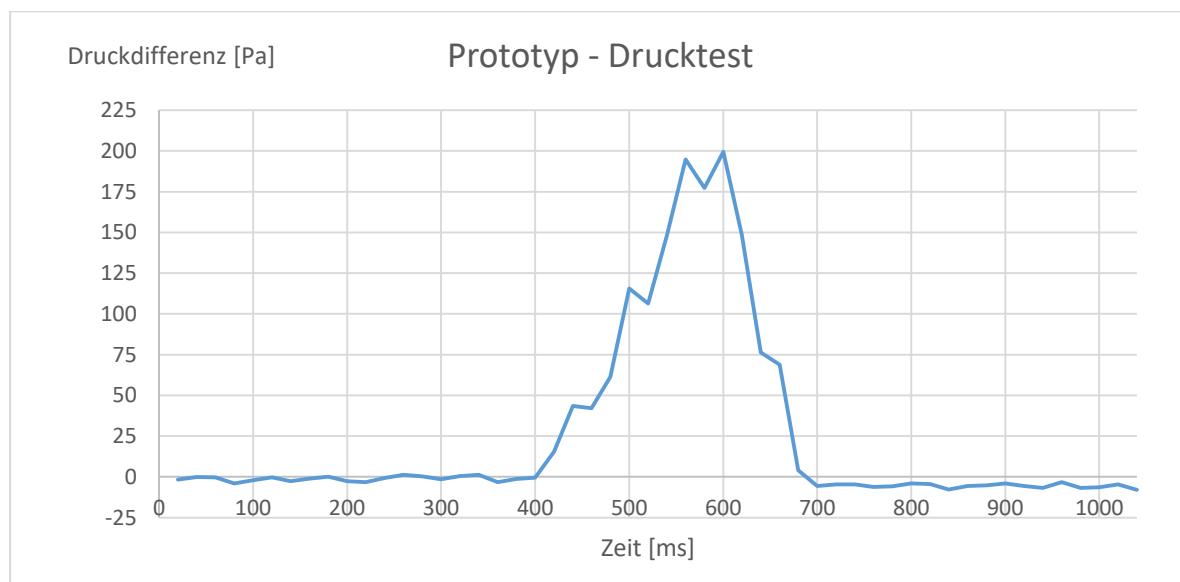


Abb. 3-69 Diagramm Druckdifferenz über Zeit

4 Ergebnisse

4.1 Hardware

4.1.1 Auswahl der Hardware

Die erste wichtige Entscheidung, die im Projekt zu treffen war, war die Hardware. Hierfür wurde der Arduino Mega 2560 gewählt, da dieser Microcontroller im Vergleich zu anderen Microcontrollern über genügend I/O-Pins sowie über die optimale Größe für den Einbau in die Ziehharmonika verfügt. Bei

der Zug-Druck Ermittlung wurde Abb. 4-2 Arduino Mega 2560 (Quelle: store.arduino.cc)

der Luftdrucksensor BMP280 eingesetzt, da dieser eine ausreichende Genauigkeit liefert und wenig Platz in Anspruch nimmt. Zur Detektierung der Tastendrücke fiel die Entscheidung auf SMD-Taster, da diese problemlos unter den Ziehharmonikatasten verbaut werden können.

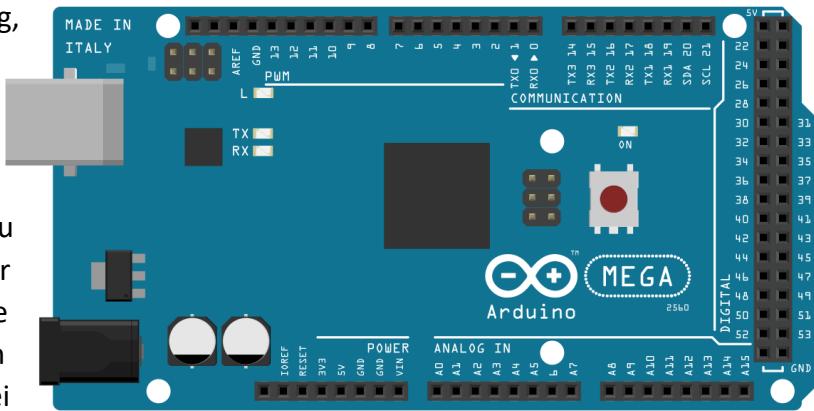


Abb. 4-1
BMP280

4.1.2 Tastendruckdetektierung

Um jeden Tastendruck detektieren zu können, wurden unter den Ziehharmonikatasten elektronische Taster verbaut. Dazu wurde die Ziehharmonika in ihre Einzelteile zerlegt, wie man in Abb. 4-3 sehen kann.



Abb. 4-3 Zerlegte Ziehharmonika

In Abb. 4-4 sieht man die Holzplatte, auf der die Ziehharmonikatasten befestigt sind, sowie die Auskerbungen, in denen die Taster eingebaut werden.



Abb. 4-4 Zerlegte Ziehharmonika - Detailaufnahme Melodieseite

Damit man auch unter die Ziehharmonikatasten gelangen konnte, mussten diese ebenfalls zerlegt werden. Anschließend wurden an die Taster jeweils zwei Kabel verlötet und unter den Ziehharmonikatasten eingeklebt. Je nach verfügbarem Platz mussten verschiedene Tastergrößen und Kabeldurchmesser verwendet werden. In Abb. 4-5 sieht man die verbauten Taster mitsamt den angelöteten Kabeln.



Abb. 4-5 Melodieseite mit eingeklebten Tastern

Nun wurden die zerlegten Ziehharmonikataster wieder zusammengebaut. Beim Zusammenbau musste darauf geachtet werden, dass keine Kabel unabsichtlich abgeklemmt werden. Danach wurde jeweils ein Kabel pro Taster an die GND-Leitung angelötet, dadurch vermindert sich die Kabelanzahl, die durch den Balg geführt werden muss, um nahezu 50%. Das Ergebnis dieses Arbeitsschrittes sieht man in Abb. 4-6.



Abb. 4-6 Ziehharmonikatasten eingebaut + Kabel an GND angelötet

Die gesamten Kabel wurden in drei ungefähr gleich große Kabelstränge zusammengeführt, damit eine minimale Anzahl an Löchern gebohrt werden muss, um die Kabel in den Balg führen zu können. Nachdem die Löcher gebohrt wurden, konnten die drei Kabelstränge hindurchgeführt und die Holzplatte mit den Ziehharmonikatasten konnte wieder an den Balg angeschraubt werden. In Abb. 4-7 sieht man die drei Kabelstränge sowie die gebohrten Löcher.



Abb. 4-7 Kabelstränge + Löcher

Im nächsten Arbeitsschritt wurden die Kabel zusammengeführt und an eine Pfostenbuchse mit 40 Pins und an eine mit 8 Pins montiert. Dies ist in Abb. 4-8 ersichtlich.



Abb. 4-8 Pfostenbuchse montiert

Von der anderen Seite der Ziehharmonika, der Bassseite, wird ein Flachbandkabel durch den Balg hindurchgeführt. Wir haben uns für ein Flachbandkabel entschieden, da dieses zusammenbleibt und die Gefahr, dass ein Kabel irgendwo bei der Spielbewegung hängen bleibt, gemindert wird. Damit man von der Bassseite mit dem Flachbandkabel in den Balg gelangt, wurde ein Schlitz freigeschnitten, wie in Abb. 4-9 zu erkennen ist.



Abb. 4-9 Schlitz für Flachbandkabel

Am Flachbandkabelende wird eine Pfostenbuchse mit 40 Pins und eine mit 8 Pins montiert. Eine selbstentworfene und gebaute Platine, auf der zwei Wannenstecker mit 40 Pins und zwei mit 10 Pins (2 zwei Pins zur Reserve) angelötet wurden, dient als Verbindungsstück zwischen dem Flachbandkabel und den Kabel der Melodieseite. Die Pfostenbuchsen wurden in die Wannenstecker gesteckt und die Platine wurde anschließend an der Ziehharmonika festgeklebt. Durch das Festkleben wird eine Zugentlastung gewährleistet. In diesem Arbeitsschritt wurde auch der Luftdrucksensor montiert und an das Flachbandkabel angeschlossen. In Abb. 4-11 sieht man das Layout der Platine auf der Top-Seite und in Abb. 4-12 auf der Bottom-Seite. In Abb. 4-10 sieht man, wie die Platine und der Luftdrucksensor im Balg der Ziehharmonika verbaut wurden.



Abb. 4-10 Platine + Luftdrucksensor verbaut in Ziehharmonika

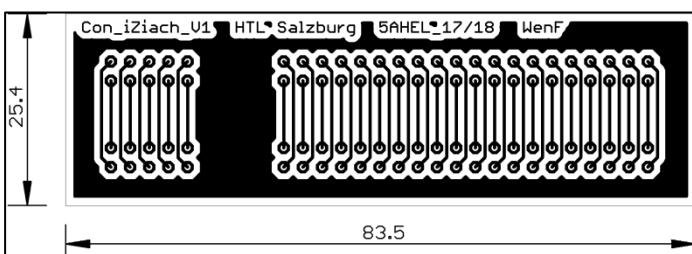


Abb. 4-11 Platine Layout top-Seite

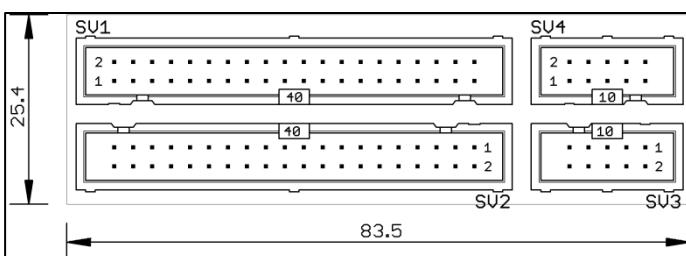


Abb. 4-12 Platine Layout bot-Seite

Die Bassseite sowie die Melodieseite wurden nun wieder zusammengebaut. Anschließend wurden die Taster in der Bassseite montiert und alle Kabel wurden an den Arduino angelötet, der in der Bassseite verbaut wurde. Mittels eines USB-Verlängerungskabels wurde die USB-B Buchse so montiert, dass diese beim Spielen später nicht stört. In der folgenden Abbildung sieht man, wie der Arduino Mega 2560 und die USB-B Buchse eingebaut und die Kabel angelötet wurde.

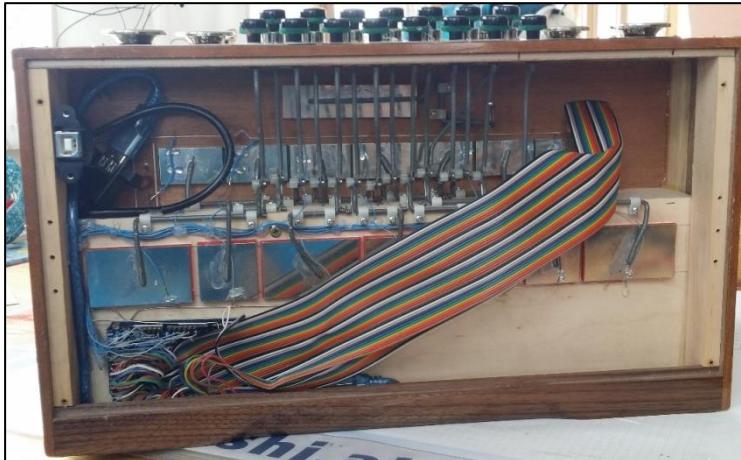


Abb. 4-13 Fertiger Einbau der Elektronik

Damit das USB-Kabel auch noch angeschlossen werden kann, nachdem die Holzverkleidung der Bassseite wieder montiert wurde, musste ein kleiner Ausschnitt passgenau aus der Holzverkleidung herausgeschnitten werden. Die folgende Abbildung zeigt den Ausschnitt auf der montierten Holzverkleidung.



Abb. 4-14 Ausschnitt Bassseite ohne Kabel und mit Kabel

4.1.3 Kommunikation mit der Software

Um die Kommunikation mit dem Notenaufzeichnungssystem zu gewährleisten sowie zum Auslesen der Tasterwerte und des gemessenen Luftdrucks wurde ein Microcontrollerprogramm entwickelt. Für die Entwicklung des Microcontrollerprogramms wurde die Open-Source Software „Arduino Studio“ verwendet. Bei der Entwicklung des C-Programms wurde darauf geachtet, dies möglichst variabel zu gestalten, sodass es für eine Nutzung für weitere Ziehharmonikas verwendet werden kann. Um die vielseitige Einsatzmöglichkeit zu gewährleisten, wurden zu Beginn des Programms die Anzahl der Melodie- und Basstasten sowie die nötige Druckdifferenz für den Luftdrucksensor, definiert. Die von uns verwendete Ziehharmonika besitzt 46 Melodietasten und 14 Basstasten. Die Druckdifferenz wurde nach einer Vielzahl von Testversuchen auf einen Wert von 100 Pa festgelegt. (siehe

```

11 #define anzahlButtonMelodie 46
12 #define anzahlButtonBass 14
13 #define pressureBorder 100.0

```

Abb. 4-15 Definition Taster Anzahl und Druckdifferenz

Prototypingphase 3.3) Bei der Definition der Druckdifferenz musste darauf geachtet werden, dass dieser Wert weder zu groß noch zu klein gewählt wird. Bei einem zu großen Wert würde dauerhaft Druck oder Zug an die Software gesendet werden, da die

gemessene Druckdifferenz den definierten Wert der Druckdifferenz nicht überschreiten würde und dadurch keine Änderung des Druckstatus erfolgt. Bei einem zu klein gewählten Wert würde sich der Druckstatus aufgrund von Luftverwirbelungen im Balg, die Druckdifferenzen erzeugen, ständig ändern, obwohl durchgehend gedrückt oder gezogen wird.

In der Software wurden auch die benötigten GPIO-Pins des Microcontrollers definiert. Hierzu wurde jeweils für die Taster auf der Melodie- und Bassseite ein Array angelegt. Jedem Taster wurde eine eindeutige ID-Nr. zugewiesen, die dem GPIO-Pin entspricht und in der Software im Array hinterlegt ist. So kann nicht nur jeder Knopf eindeutig identifiziert, sondern bei auftretenden Fehlern auch eine gezielte Fehlerbehebung ausgeführt werden. Zusätzlich wurden für beide Seiten, Melodie und Bass, ein Array, das den Status beinhaltet, definiert.

```

17 int pushButtonMelodie [anzahlButtonMelodie] = {53,52,51,50,49,48,47,46,45,44,43,42,41,40,39,38,37,36,35,34,
18   33,32,31,30,29,28,27,26,25,24,23,22,19,18,17,16,15,14,3,4,5,6,7,8,9,10};
19 int buttonStateMelodie [anzahlButtonMelodie];
20
21 int pushButtonBass[anzahlButtonBass] = {56,57,58,59,60,61,62,63,64,65,66,67,68,69};
22 int buttonStateBass[anzahlButtonBass];

```

Abb. 4-16 Arrays für Melodie -und Bassseite

Für die Taster auf der Melodieseite wurden die digitalen GPIO-Pins verwendet. Der Arduino Mega 2560 besitzt 54 digitale GPIO-Pins, davon werden 46 Pins mit den Nummern 53 bis 3 (Die Pins 20 und 21 werden für I²C-Bus benötigt. Die Pins 11,12 und 13 wurden auf Grund ihrer ungünstigen Anordnung auf dem Arduiono Mega 2560 ausgelassen) verwendet. Für die Taster auf der Bassseite wurden die analogen GPIO-Pins verwendet. Der Microcontroller verfügt über 16 analoge GPIO-Pins, davon werden 14 Pins verwendet. Normalerweise werden die analogen Pins mit Ax (x = Nummer des Pins) angesprochen. Werden diese jedoch mit den Nummern 54 bis 69 angesprochen, so kann man diese als digitale Pins verwenden. Der intergierte Analog-Digital-Umsetzer wandelt das Signal in einem Wert zwischen 0 und 1023 um. Ist der gewandelte Wert größer gleich 512, so wird ein logischer HIGH-Zustand erkannt. Bei einem Wert kleiner als 512 wird ein logischer LOW-Zustand detektiert.

Zusätzlich zu den Arrays wurden auch noch einige Variablen, wie in Abb. 4-17 Variablen Deklaration zu sehen, deklariert. Die Variable stateBellows beinhaltet den aktuellen Druck-Zug Status. In der Variable previousStateBellows wird der vorherige Druck-Zug Status hinterlegt. Die beiden Variablen referencePressure und actualPressure, vom Typ float, beinhalten den Referenzdruckwert beziehungsweise den aktuell gemessenen Druckwert.

```

24 int stateBellows = 0;
25 int previousStateBellows = 0;
26 float referencePressure;
27 float actualPressure;

```

Abb. 4-17 Variablen Deklaration

In der Setup Routine, die immer bei einem Neustart oder einem Reset durchlaufen wird, wird unter anderem die serielle Kommunikation mit einer Baudrate von 9600 Bits pro Sekunde

gestartet. Auch die Kommunikation mit dem Luftdrucksensor BMP280 wird in der Setup Routine gestartet. Sollte ein Fehler beim Kommunikationsaufbau mit dem Luftdrucksensor auftreten, dann wird der ASCII-Charakter e (e für error) an die Software gesendet, damit dem Benutzer der Software eine Fehlermeldung angezeigt werden kann. In der Setup-Routine werden auch die GPIO-Pins als Inputs definiert. Hierzu wird das Array mit den Tasten der Melodieseite durchlaufen und jeder Pin wird mit der Funktion pinMode und dem Befehl INPUT_PULLUP als Input-Pin definiert. Das Attribut „_PULLUP“ bewirkt, dass die internen Pullup-Widerstände aktiviert werden. Aufgrund der Verwendung von internen Pullup-Widerständen müssen keine externen Pullup-Widerstände verwendet werden, wodurch zum einen ein erheblicher Mehraufwand vermieden und zum anderen kein zusätzlicher Raum benötigt wird. Das Array mit den Tasten der Bassseite wird ebenfalls durchlaufen und jeder Pin wird mit der Funktion pinMode und dem Befehl INPUT als Input-Pin definiert. Damit auch die internen Pullup-Widerstände der analogen Pins aktiviert werden, wird mit der Funktion digitalWrite ein logischer HIGH-Zustand an den jeweiligen Pin gesendet.

```

30 void setup() {
31   Serial.begin(9600);    //serial communication at 9600 bits per second:
32
33   if(!bmp.begin()){
34     Serial.write('e');
35     Serial.write(0);
36   }
37   // make the pushbutton's pin an input:
38   for(int i = 0; i<anzahlButtonMelodie; i++)
39     pinMode(pushButtonMelodie[i], INPUT_PULLUP);
40
41   for (int i = 0; i < anzahlButtonBass; i++)
42   {
43     pinMode(pushButtonBass[i], INPUT);
44     digitalWrite(pushButtonBass[i], HIGH);
45   }
46 }
```

Abb. 4-18 Setup-Routine

In der Loop-Routine werden der aktuelle Status der Taster sowie der gemessene Wert des Luftdrucksensors eingelesen und weiter verarbeitet. Mittels einer Schleife werden alle Pins durchlaufen und der mit dem Befehl digitalRead eingelesene Wert wird in das Array, das den Status der Pins beinhaltet, gespeichert. Nach dem Speichern wird überprüft, ob der jeweilige Status den Wert 0 oder 1 besitzt. Besitzt der Status den Wert 0, so bedeutet dies, dass der Taster gedrückt wurde, und diese Information wird an die Software gesendet. Hierzu wird das Protokoll, wie in 4.2.3 beschrieben, verwendet. Das Überprüfen der Taster Werte erfolgt in zwei getrennten Schleifen für die Melodie -sowie Bassseite. Dadurch werden Test und Fehlerbehebungen deutlich vereinfacht.

Ein Durchlauf der Loop-Routine hat eine Dauer von 20ms. Daraus ergibt sich eine Abtastrate von 50Hz.

```

49 void loop() {
50   // read the input pin:
51   for(int i = 0; i<anzahlButtonMelodie; i++)
52   {
53     buttonStateMelodie[i] = digitalRead(pushButtonMelodie[i]);
54     if(buttonStateMelodie[i] == 0)
55     {
56       Serial.write('m');
57       Serial.write(pushButtonMelodie[i]);
58     }
59   }
60   for(int j = 0; j<anzahlButtonBass; j++)
61   {
62     buttonStateBass[j] = digitalRead(pushButtonBass[j]);
63     if(buttonStateBass[j] == 1)
64     {
65       Serial.write('b');
66       Serial.write(pushButtonBass[j]);
67     }
68   }

```

Abb. 4-19 Loop-Routine Teil 1

Nach dem Auslesen des Status der Taster wird der Status des Luftdrucksensors ermittelt. Dazu wird der erste gemessene Luftdruckwert als Referenzwert in der Variable referencePressure gespeichert (nur wenn referencePressure keinen Wert beinhaltet). Der aktuell gemessene Wert wird mit dem Befehl bmp.readPressure() ausgelesen und in die Variable actualPressure gespeichert. Zudem wird der aktuelle Luftdruckstatus in der Variable previousStateBellows gespeichert, sodass der aktuelle Luftdruckstatus wieder aktualisiert werden kann, ohne dass dieser für darauffolgende Berechnungen verloren geht. Nun wird die Differenz zwischen dem gemessenen Wert und dem Referenzwert gebildet. Je nachdem ob die Differenz positiv oder negativ ist und die vorher festgelegte Druckdifferenz überschritten wird, kann festgestellt werden, ob gedrückt oder gezogen wird. Bei Druck wird der Wert 1 und bei Zug der Wert 0 in die Variable stateBellows geschrieben. Darauffolgend wird verglichen, ob sich der aktuelle Status im Vergleich zum vorherigen Status geändert hat. Bei einer Änderung wird der Status an die Software gesendet. Hierzu wird das Protokoll, wie in 4.1.3 beschrieben, verwendet. Die Loop-Routine beginnt nun wieder von neuem und die Loop-Routine wird solange durchlaufen, bis der Arduino Mega 2560 über keine Spannungsversorgung mehr verfügt.

```

71 if(!referencePressure)
72   referencePressure = bmp.readPressure();
73
74 actualPressure = bmp.readPressure();
75 previousStateBellows = stateBellows;
76
77 if((referencePressure - actualPressure) < -pressureBorder)
78   stateBellows = 1;
79 else if((referencePressure - actualPressure) > pressureBorder)
80   stateBellows = 0;
81
82 if(stateBellows != previousStateBellows)
83 {
84   Serial.write('1');
85   Serial.write(stateBellows);

```

Abb. 4-20 Loop-Routine Teil 2

4.2 Notenaufzeichnungssystem

4.2.1 Problemstellung

Für die Oberfläche musste sowohl eine Programmiersprache als auch ein geeignetes Framework ausgewählt werden. Die Entscheidung fiel auf das moderne C# in Verbindung mit dem WPF-Framework, da die Plattformabhängigkeit nicht von Priorität ist. Mit C# kann effizient und schnell gearbeitet werden, weil es die Vorteile aus Java und C++ kombiniert. Das WPF-Framework wurde gewählt, da damit grafisch anspruchsvolle Oberflächen für Windows erstellt werden können. Als Entwicklungsumgebung wurde das für das Arbeiten mit C# und WPF bestens geeignete Visual Studio 2015 verwendet.

4.2.2 Aufbau des Projekts

Das Projekt wurde, wie in Kapitel 3.1.6.3 beschrieben, in mehrere Unterprojekte gegliedert. Die Unterprojekte sind vom Typ Klassenbibliothek und beinhalten alle Klassen des jeweiligen Namespaces. Im Solution-Explorer von Visual Studio werden diese angezeigt:

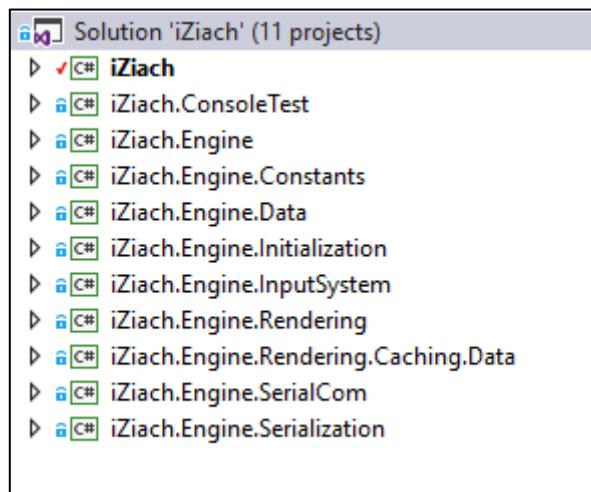


Abb. 4-21 Projektaufteilung im Solution Explorer

Das Projekt iZiach ist vom Typ WPF-Applikation und gibt bei der Kompilierung eine ausführbare Datei aus (.exe). iZiach.ConsoleTest ist eine Konsolenanwendung und wurde für Testzwecke verwendet. Alle anderen Projekte sind Klassenbibliotheken, die vom ausführbaren Projekt referenziert werden.

Für die Versionsverwaltung wurde eine GIT Repository auf dem Anbieter Bitbucket eingerichtet. Visual Studio bietet grafische Tools zur Kommunikation mit einer GIT Repository, die das Verwalten einer solchen um einiges erleichtern. Zusätzlich wurde damit eine einfache Zusammenarbeit unter den Teamkollegen ermöglicht und die Existenz der Projektdateien nicht nur auf lokale Kopien begrenzt.

Damit die Oberfläche zu jeder Zeit angesprochen werden kann, wurde für das Rendering und für die Hintergrundberechnungen ein eigener Thread, unter MonoGame-Entwicklern bekannt

als „GameLoop“, erstellt. Der GameLoop arbeitet in einer Endlosschleife und führt abwechselnd die Draw-Funktion (Zeichnen) und die Update-Funktion (Aktualisieren) aller Komponenten aus.

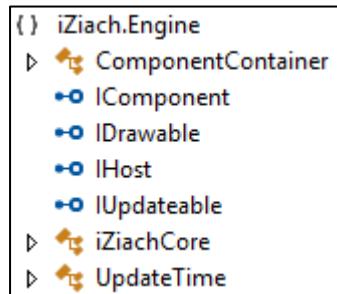


Abb. 4-22 Projekt iZiach.Engine

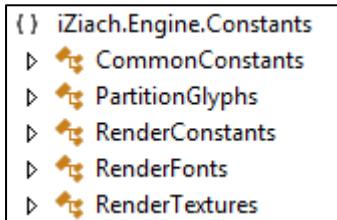


Abb. 4-23 Projekt
iZiach.Engine.Constants

In iZiach.Engine wurden Basisklassen und Interfaces definiert, von denen sämtliche hierarchisch höhere Klassen erben oder diese implementieren. Mit den Interfaces IDrawable, IUpdateable und IComponent wurden Schnittstellen für alle innerhalb des GameLoops laufenden Objekte erstellt. Über die statische Klasse iZiachCore können Services und Komponenten von jedem Objekt aus aufgerufen werden.

In iZiach.Engine.Constants wurden, wie der Name bereits angibt, alle benötigten Konstanten definiert. CommonConstants enthält alle allgemeinen Konstanten, wie die Erweiterung für iZiach-Dateien. RenderConstants enthält alle Konstanten, die für die Positionierung und Größen von Seiten, Notenzeilen, Taktzeichen und Noten benötigt werden. PartitionGlyphs, RenderFonts und RenderTextures enthalten Schriftart-Objekte, Texturen und weitere Objekte, die beim Start der Applikation geladen werden.

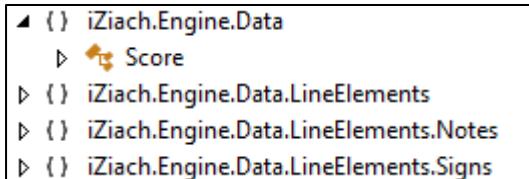


Abb. 4-24 Projekt iZiach.Engine.Data

In iZiach.Engine.Data wurden alle Klassen für die Beschreibung der Daten einer Partitur (eines Liedes) deklariert. Der grobe Aufbau der Datenobjekte ist auf dem Klassendiagramm auf Abb. 4-25 ersichtlich.

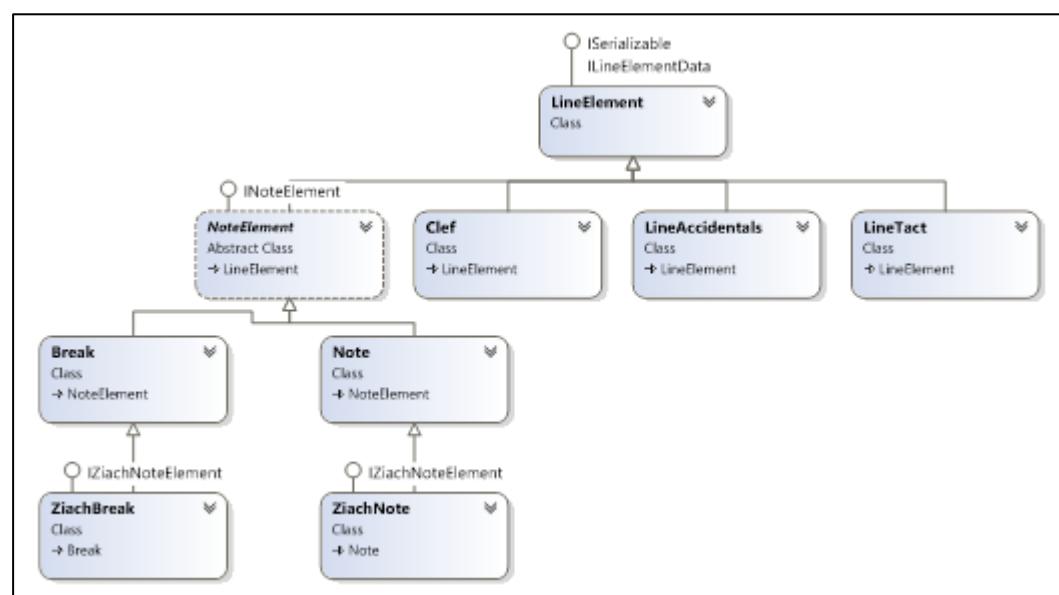


Abb. 4-25 Klassendiagramm iZiach.Engine.Data

Als Basisklasse dient ein LineElement (Linienelement), das die Interfaces ISerializable (für Serialisierung/Deserialisierung) und ILineElementData implementiert. Clef (Notenschlüssel), LineAccidentals (Vorzeichen), LineTact (Taktzeichen) und die abstrakte Klasse NoteElement erben von LineElement. Break (Pause) und Note (normale Note) erben wiederum von NoteElement und sind die Basisklassen für ZiachBreak (Pause) und ZiachNote (Note).

ZiachBreak und ZiachNote implementieren die Schnittstelle iZiachNoteElement, welche das Bereitstellen einer ZiachData Klasse, die Zugrichtung und Bässe festlegt, voraussetzt. Zusätzlich zu den im Klassendiagramm ersichtlichen Klassen wurden viele Helferklassen deklariert.

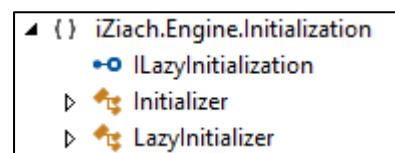


Abb. 4-26 Projekt
iZiach.Engine.Initialization

In iZiach.Engine.Initialization wurden Initialisierungsvorgänge erstellt, die zu Beginn der Ausführung aufgerufen werden. Dazu gehören das Laden von Schriftarten, Texturen und weitere für die Darstellung der Griffsschrift erforderliche Objekte.

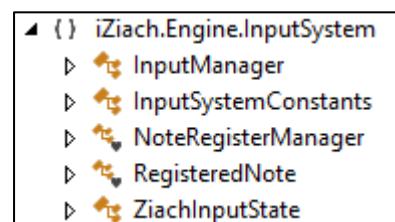


Abb. 4-27 Projekt
iZiach.Engine.InputSystem

In iZiach.Engine.InputSystem wurden die Vorgänge und der Algorithmus für die Aufzeichnung der Griffsschrift ausprogrammiert. Der InputManager ist für die Verwaltung des Aufzeichnungsstatus und die Erkennung der gespielten Noten verantwortlich. Im NoteRegisterManager wurden Verbesserungen der bereits erkannten Noten und Routinen zur Erkennung von fehlerhaften Eingaben implementiert.

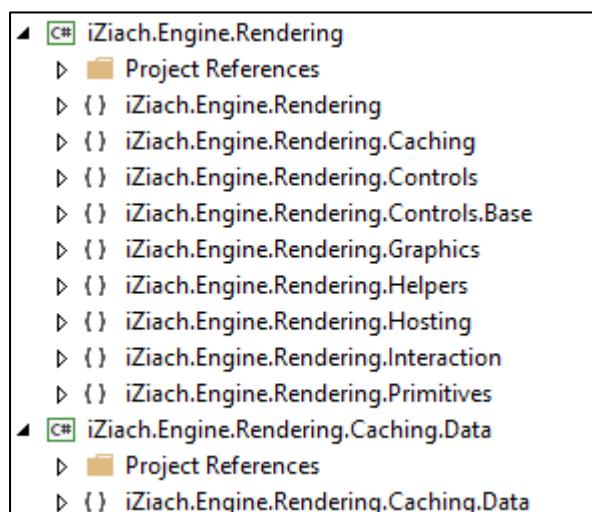


Abb. 4-28 Projekte iZiach.Engine.Rendering und Caching

Die Darstellung der Griffsschrift erfolgt in den Projekten iZiach.Engine.Rendering und iZiach.Engine.Rendering.Caching.Data. In Rendering werden sämtliche Klassen festgelegt, die für die Einbettung von MonoGame in WPF benötigt werden. Wie auf Abb. 4-28 zu erkennen ist, werden hier auch Routinen zum Zeichnen von primitiven geometrischen Objekten (Linien, Kreise, Rechtecke), zur Erkennung von Interaktionen mit der Oberfläche (Interaction) und Basisklassen für das Caching-System festgelegt. Die Klassen für das Caching-System selbst wurden in Rendering.Caching.Data deklariert.



Abb. 4-29 Projekt iZiach.Engine.SerialCom

iZiach.Engine.SerialCom dient zur Kommunikation mit der Hardware. Der CommunicationManager baut die Verbindung auf und leitet Daten zum ZiachStateManager weiter, der diese aufbereitet und den Status der Ziehharmonika zu jedem Zeitpunkt speichert.



Abb. 4-30 Projekt
iZiach.Engine.Serialization

In iZiach.Engine.Serialization werden Klassen und Schnittstellen für die Ausgabe als iZiach-Datei deklariert. Das Interface ISerializable sieht Funktionen zum Speichern (Serialize) und Laden (Deserialize) von Daten vor. Die statische Klasse Serializer stellt Funktionen zur Verfügung, mit denen das Speichern oder Laden einer iZiach-Datei gestartet werden kann.

4.2.3 Datenverarbeitungspfad

Bei bestehender Verbindung wird überprüft, welche Ziehharmonikatasten gedrückt sind. Parallel dazu wird der Wert des Luftdrucksensors mittels I²C-Bus ausgelesen und von einem zu Beginn der Ausführung gemessenen Referenzwerts subtrahiert. Je nachdem ob die Differenz positiv oder negativ ist, kann festgestellt werden, ob gedrückt oder gezogen wird. Die gemessenen Daten werden über die serielle Schnittstelle in einer bestimmten Struktur gesendet. Je nach Ereignistyp werden ein ASCII-Character und nachfolgend ein Zahlenwert gesendet. Die folgende Tabelle zeigt den Aufbau dieser Struktur.

Ereignistyp	Gesendete Daten
Knopfdruck Melodieseite	m<Wert>
Knopfdruck Bassseite	b<Wert>
Zugrichtungswechsel	l0 oder l1

Abb. 4-31 Datenstruktur über Schnittstelle

„<Wert>“ steht für die jeweilige Identifikationsnummer eines Knopfes. Jedem Taster wurde eine eindeutige ID-Nr. zugewiesen, die dem GPIO-Pin entspricht und in der Software hinterlegt ist. So kann nicht nur jeder Knopf eindeutig identifiziert, sondern bei auftretenden Fehlern auch eine gezielte Fehlerbehebung ausgeführt werden.

Der Microcontroller sendet bei aufrechter Spannungsversorgung ununterbrochen. Da die Spannungsversorgung der Elektronik ebenfalls über das USB-Kabel geschieht, kann auf eine Verbindungskontrolle oder einer externen Spannungsquelle verzichtet werden. Wird die Ziehharmonika mit dem Computer verbunden und die Verbindung von der Software aus gestartet, zeigt die iZiach-Software jederzeit den Status der Ziehharmonika an. Dazu gehören alle gedrückten Knöpfe und die aktuelle Zugrichtung.

Die Verbindung wird von der Software in der Klasse SerialCommunicator des SerialCom-Unterprojekts gesteuert. Wird von der Oberfläche der Befehl zum Verbindungsaubau gegeben, wird versucht, mit voreingestellten Baudrate und COM-Port eine serielle Verbindung zu öffnen. Dazu wird die im .NET Framework verfügbare Klasse SerialPort verwendet.

```

serial = new SerialPort(DefaultPort, DefaultBaudrate);
serial.DataReceived += Serial_DataReceived;
serial.Open();
  
```

Abb. 4-32 Verbindungsaubau mit SerialPort (vereinfacht)

Bei erfolgreichem Verbindungsauflauf wird ein EventHandler an das DataReceived-Ereignis der SerialPort Klasse gehängt. In Serial_DataReceived (siehe Abb. 4-33) werden nun alle ankommenden Daten einer Queue (Schlange) angehängt.

```
private void Serial_DataReceived(object sender, SerialDataReceivedEventArgs e)
{
    var data = new byte[serial.BytesToRead];
    serial.Read(data, 0, data.Length);
    foreach (var b in data)
    {
        receivedData.Enqueue(b);
    }
}
```

Abb. 4-33 Einlesen der ankommenden Daten

Diese Liste an Daten wird in bestimmten Zeitabständen (ausgehend vom GameLoop) auf vollständige Datenpakete (jeweils 2 Bytes) überprüft. Sind vollständige Datenpakete angekommen, werden diese wie in der in Abb. 4-31 ersichtlichen Struktur interpretiert und als ZiachState gespeichert, welches wiederum dem ZiachStateManager weitergegeben wird.

```
private void UpdateReceivedData()
{
    while (receivedData.Count >= PacketSize)
    {
        byte type = receivedData.Dequeue();
        byte bytes = receivedData.Dequeue();
        int value = (int)bytes;

        var state = new ZiachState(GetStateTypeFromChar((char)type), value);
        ziachManager.SetState(state);
    }
}
```

Abb. 4-34 Einlesen eines Datenpakets (vereinfacht)

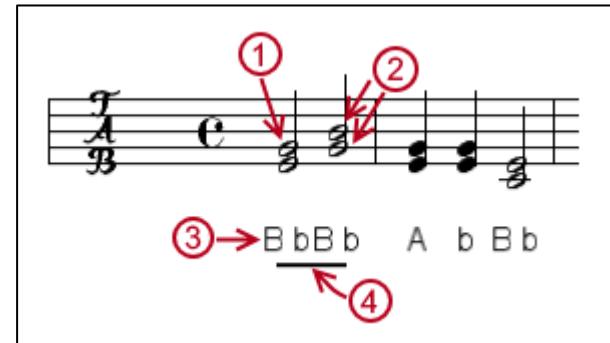
Der ZiachStateManager speichert eine Liste von ZiachStates und aktualisiert in regelmäßigen Abständen die Zeit, die seit dem Einlesen eines ZiachStates vergangen ist. Überschreitet diese Zeit („RecogniseTime“) einen Schwellenwert von 100ms, wird der Eintrag gelöscht. So stellt der ZiachStateManager zu jedem Zeitpunkt den aktuellen Status der Ziehharmonika zur Verfügung.

4.2.4 Die Griffschriftnotation

Die Griffschriftnotation für die Steirische Ziehharmonika weicht etwas von normalen Noten ab. Es gibt keine Norm, welche das Aussehen der Griffschrift festlegt. Unter den Ziehharmonikaspielern hat sich aber in den letzten Jahren eine Notation durchgesetzt. Diese wird im Folgenden erklärt:

Mit Punkt 1 (siehe Abb. 4-35) wird die Notenlänge markiert. Diese gibt wie bei normalen Noten auch die relative Länge des Tones zu anderen Noten an. Dabei spricht man von Ganze, Halbe, Viertel, Achtel- oder Sechzehntelnoten. Eine ganze Note wird durch einen hohlen Notenknopf (eine nicht ausgefüllte Ellipse) und keinen Notenhals (der senkrechte Strich) gekennzeichnet.

Eine halbe Note (wie auf Punkt 1) hat ebenfalls einen hohlen Notenknopf, hat aber auch einen Notenhals. Eine Viertelnote gleicht der Halben, hat aber einen ausgefüllten Notenkopf. Die Achtelnote gleicht wiederum der Viertelnote, hat aber am Ende des Notenhalses noch eine kleine Flagge.



Welche Knöpfe gedrückt wurden, wird durch die Anzahl und die Notenhöhe (Punkt 2) gekennzeichnet. Ist der Notenkopf zwischen den Linien, bedeutet das, dass der Knopf in der ersten Reihe liegt. Auf der Linie bedeutet, dass der Knopf in Reihe 2 liegt. Die dritte und vierte Reihe werden zusätzlich mit einem Kreuz vor dem Notenkopf gekennzeichnet.

Abb. 4-35 Beispiel der Griffsschriftnotation

Die Buchstaben unter der Note (Punkt 3) geben die gedrückten Bässe an. Die Zugrichtung (Punkt 4, Zug oder Druck) wird durch einen Strich unter der Note gekennzeichnet.

4.2.5 Aufzeichnen von Griffsschrift

4.2.5.1 Eingabesystem

Der InputManager ist für das Eingabesystem verantwortlich. Er verwaltet den Status der Aufnahme, erkennt gespielte Noten, leitet diese an den Datenmanager weiter und löst Aktualisierungen der Oberfläche aus. Wird bei aufrechter Verbindung die Aufzeichnung gestartet, wird alle 100ms der Zustand der Ziehharmonika zwischengespeichert. Mit einem speziell entworfenen Algorithmus wird erkannt, wann ein Ton zu Ende gespielt wurde, und die Daten für den Ton werden gesammelt.

4.2.5.2 Erkennungsalgorithmus

Ein Ton der Griffsschrift beinhaltet die Notenhöhe, die Notenlänge, die Reihe der Melodietasten, die gedrückten Basstasten und die Zugrichtung. Diese Daten werden mit dem Erkennungsalgorithmus aus den ZiachStates berechnet. In einer im GameLoop aufgerufenen Funktion des InputManagers wird alle 100ms überprüft, ob die Melodietasten des zuletzt erkannten ZiachStates denselben Tasten des ältesten ZiachStates entsprechen. Ist dies nicht der Fall, so wurde ein Ton zu Ende gespielt und alle ZiachStates, die seit dem erkannt wurden, werden gesammelt. Anschließend werden aus den gesammelten States alle gedrückten Bässe ausgelesen und mit den bisher analysierten eine Plausibilitätsprüfung durchgeführt.

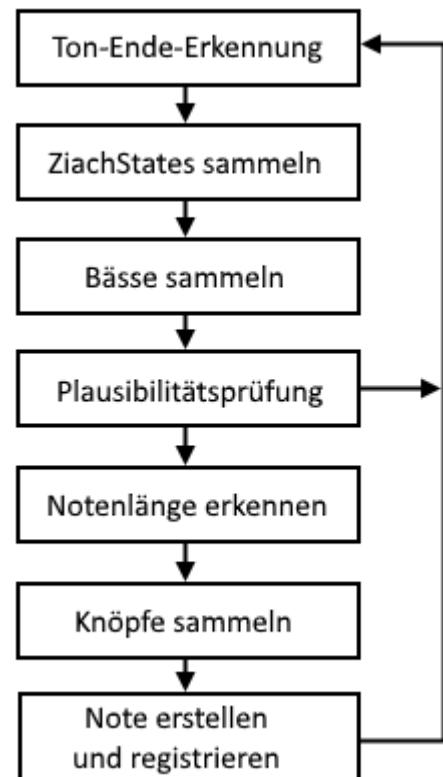


Abb. 4-36 Erkennungsalgorithmus

Sind die Bässe und Melodietasten plausibel, wird mit der Analyse fortgeführt, andernfalls werden die erkannten States gelöscht und keine Note eingetragen. So können Fehleingaben, die zum Beispiel durch zeitlich ungenaue Spielweise entstehen, erkannt und ausgebessert werden.

Anhand der gespielten Bässe kann die Notenlänge festgelegt werden. Dafür werden die Bässe gezählt und mit einem Viertel gleichgesetzt. Je nach der Anzahl der Bässe kann so eine Ganze, Halbe oder Viertelnote erkannt werden. Wurde kein Bass gespielt, ist die Note kürzer als eine Viertelnote und es kann noch keine bestimmte Notenlänge festgelegt werden. Um dies zu berücksichtigen, wird die Note mit einem „Doubt“-Flag (Zweifel) versehen und späteren Korrekturen unterzogen.

Nach dem Sammeln der gedrückten Knöpfe wird abhängig davon, ob nur Bass oder auch Knöpfe gedrückt wurden, eine Pause oder eine Note erstellt. Diese wird dem NoteRegisterManager hinzugefügt.

4.2.5.3 Fehlererkennung und Optimierung

Die Erkennung und Korrektur von Noten wird vom NoteRegisterManager durchgeführt. Dieser enthält eine Liste aller bisher analysierten Noten als RegisteredNote („registrierte Note“). Eine registrierte Note besteht aus dem Noten-Objekt selbst, einer Doubt-Flag und der Anzahl der für die Note analysierten ZiachStates (StateCount).

Werden im NoteRegisterManager neue Noten registriert, wird anhand aller bisher registrierten Noten die mittlere StateCount für eine Viertelnote berechnet (AverageStateCount). Dazu wird die StateCount jeder Note durch die absolute Länge, die sich aus Notenlänge und Punktierung ergibt, dividiert (siehe Abb. 4-37).

```
private void UpdateAverageStateCount()
{
    averageStateCount = notes.Average(n => n.StateCount / n.Note.AbsoluteLength);
}
```

Abb. 4-37 Berechnung des AverageStateCount (vereinfacht)

Anschließend werden die Notenlängen aller registrierten Noten mit positiver Doubt-Flag der mittleren Länge entsprechend aktualisiert. So kann eine Sechzehntel- oder Achtelnote erkannt werden, vorausgesetzt es wurden bereits genügend Noten mit sicher analysierter Notenlänge eingetragen.

Nach der Korrektur der Doubt-Notes wird eine Optimierung aller bereits eingetragenen Noten durchgeführt. Diese folgt bestimmten Erkennungsmustern, die bei der Aufzeichnung der Noten durch die Spielweise entstehen können. Hintereinander gespielte Noten mit demselben Bass, derselben Zugrichtung, aber nur minimal abweichender Notenhöhe werden unter weiteren deterministischeren Zuständen zusammengeführt. Durch diese und weitere Optimierungen können Fehler, die durch das nicht gleichzeitige Drücken mehrere Knöpfe oder Bässe entstehen, minimiert werden.

4.2.5.4 Verbindung mit der Oberfläche

Werden dem NoteRegisterManager neue Noten hinzugefügt, werden die neuen Noten auch der Datenstruktur hinzugefügt und über ein Ereignis eine Aktualisierung der Oberfläche ausgelöst. Die Oberfläche validiert die im Cache liegenden Daten und aktualisiert bei Veränderungen die jeweiligen Komponenten des Cache-Systems.

Der beschriebene Algorithmus und die Erkennungsroutinen werden aber erst ausgeführt, wenn der Befehl dazu vom Benutzer gegeben wird. Über Buttons auf der Oberfläche kann die Verbindung aufgebaut und die Aufzeichnung gestartet werden.

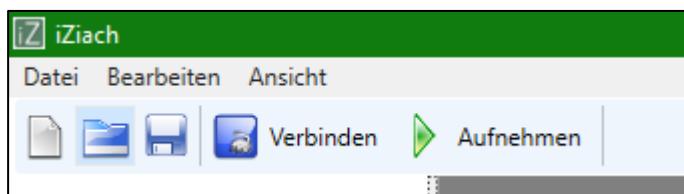


Abb. 4-38 ToolBox der WPF-Oberfläche

Der folgende Screenshot zeigt das bekannte Stück „Ennstaler Polka“, aufgezeichnet in Griffsschrift.

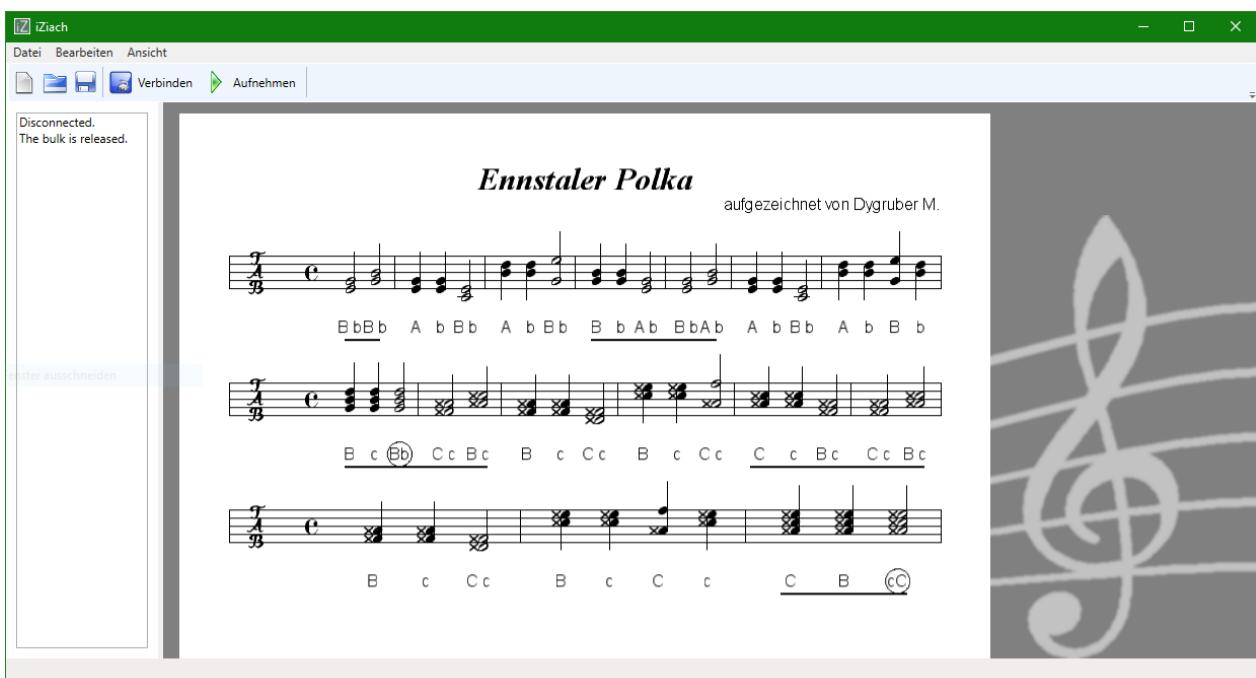


Abb. 4-39 Screenshot Ennstaler Polka

4.2.6 Darstellung der Griffsschrift

Die Darstellung der Griffsschrift erfolgt über das MonoGame-Framework, das in die WPF Applikation eingebettet wurde. Das MonoGame-Framework greift wiederum auf die Low-Level-Library DirectX von Microsoft zu. Auf diesem Weg wird ein flüssiges Rendering des Notenblatts ermöglicht, da auf den meisten Rechnern hardwarebeschleunigtes Rendern verwendet werden kann.

Die Notation selbst wird mit der Hilfe einer Schriftart-Datei (Font) gezeichnet. Diese enthält alle speziellen Symbole, die für die Zusammensetzung einer Note benötigt werden. Über ein Caching-System wird aus den Daten (einer Auflistung von Noten) ein hierarchisches System angelegt, das die Einteilung in Seiten, Notenzeilen und Takte berechnet und die Position festlegt. Durch die Trennung von Daten und Rendering wurde so eine agile Darstellung realisiert, die ohne Verzögerung auf Benutzereingaben reagieren kann.

4.2.6.1 Einbindung von MonoGame

MonoGame wurde über das Image-Control in WPF eingebettet. Dazu wurde eine Direct3D 11 Szene in ein Image-Control gehostet. Die Szene wird in ein DirectX-Image gezeichnet, das nachfolgend als Source (Quelle) des Image-Controls festgelegt wird. Trotz dieser über Umwege implementierten Einbettung von MonoGame wurde dadurch eine flüssige Darstellung des Notenblatts ermöglicht. So kann die Steuerung der Aufnahme und der Software über die agile WPF-Oberfläche erfolgen und gleichzeitig kann mit dem gerenderten Notenblatt interagiert werden.



Abb. 4-40 MonoGame Logo

4.2.6.2 Daten-/Cachingsystem

Der Aufbau der Daten-Hierarchie wurde bereits in Kapitel 4.2.2 erörtert. Die Daten werden durch eine Score (Partitur) Klasse dargestellt, die den Titel, Untertitel, Autor, die LineBeginners (Notenschlüssel, Vorzeichen und Taktangabe) und die Noten enthält. Die Daten besitzen allerdings keinerlei Informationen, wie sie auf dem Notenblatt dargestellt werden.

Das Cache-System basiert auf mehreren abstrakten komplexen Basisklassen. Die tiefste für den Cache relevante Basisklasse ist RenderCacheComponent. In ihr werden Position, Größe, Farbe und Gültigkeit festgelegt. Auf dem RenderCacheComponent baut das ChildCacheComponent auf (siehe Abb. 4-41), das als generische Klasse mit einem generischen Typ für eine DataReference deklariert ist.

```
public abstract class ChildCacheComponent<dataType, refType> : RenderCacheComponent<refType>
{
    where dataType : class where refType : ChildCacheDataReference<dataType>

    87 references | theUnknown007, 106 days ago | 1 author, 1 change
    public refType CacheData { get { return cacheData; } }
```

Abb. 4-41 Klassendeklaration ChildCacheComponent

Auf dem ChildCacheComponent baut wiederum das ContainerCacheComponent auf, das ebenfalls als generische Klasse mit einem generischen Typ für eine Liste von Children deklariert ist.

```
public abstract class ContainerCacheComponent<T>
    : ChildCacheComponent<ElementsData, ContainerCacheDataReference>
    where T : RenderCacheComponent<T>
{
    protected List<T> children;
```

Abb. 4-42 Klassendeklaration ContainerCacheComponent

Auf ChildCacheComponent und ContainerCacheComponent bauen nun die Cache-Klassen für die jeweiligen direkt zur Darstellung geeigneten Objekte auf. Diese verweisen über die generischen Typen mit einer ChildCacheDataReference oder ContainerCacheDataReference auf die für den jeweiligen Cache bestimmten Datentypen.

Die Cache-Komponente auf oberste Ebene wird vom ScoreCache (siehe Abb. 4-43) dargestellt. Dieser erbt vom ContainerCacheComponent mit dem generischen Typ SheetCache. So erhält ScoreCache eine Liste von SheetCache (siehe Abb. 4-42).

```
public class ScoreCache : ContainerCacheComponent<SheetCache>, IScoreRenderCache
```

Abb. 4-43 ScoreCache Klassendeklaration

Diese Hierarchie wird mit dem SheetCache fortgeführt. Das SheetCache erbt wieder von ContainerCacheComponent und enthält damit mehrere LineCaches.

```
public class SheetCache : ContainerCacheComponent<LineCache>
```

Abb. 4-44 SheetCache Klassendeklaration

Diese Struktur der Cache-Klassen folgt dem Aufbau der Datenstruktur. Die ScoreCache (Partitur) enthält mehrere SheetCaches. Ein SheetCache (Blatt) enthält mehrere LineCaches (Notenzeilen). Ein LineCache enthält mehrere TactCaches (Takte). Ein TactCache enthält schließlich mehrere LineElementCaches. Ein LineElementCache (siehe Abb. 4-45) erbt nun von ChildCacheComponent und verweist direkt auf einen LineElement-Datentyp. Dazu gehören Noten und Pausen.

```
public abstract class LineElementCache<T>
    : ChildCacheComponent<T, ChildCacheDataReference<T>>, ILineElementCache
    where T : class, ILineElementData
```

Abb. 4-45 LineElementCache Klassendeklaration

Auf dem LineElementCache bauen NoteCache und BreakCache sowie ZiachNoteCache und ZiachBreakCache auf. Diese Klassen sind nun direkt für das Zeichnen der Noten und Pausen verantwortlich. Bei der Erstellung werden die Daten der Noten und Pausen vom jeweiligen Cache geladen und die Position der einzelnen Elemente einer Note (Notenkopf, Notenhals, Flagge, Bässe) berechnet. Bei jedem Zeichenvorgang (etwa alle 16ms) zeichnet jeder Cache seine zuvor berechneten Daten. Bei Datenänderungen wird überprüft, ob die berechneten Daten eines Caches ungültig geworden sind. Bei Ungültigkeit werden diese als „Invalid“ markiert und vom jeweiligen Parent (Takt, Notenzeile, Blatt oder Partitur) neu erstellt.

Der Aufbau dieses Caching-Systems erweist sich als sehr komplex, bietet allerdings einige Vorteile. Zum einen ist dadurch eine gute Trennung von Daten und Rendering gewährleistet, zum anderen wird die Performance des Zeichenvorgangs um ein Vielfaches erhöht, da die für das Zeichnen benötigten Daten nur bei Änderungen berechnet werden müssen. Zudem konnte durch die Basisklassen einiges an Code eingespart und rekursive Vorgänge einfacher implementiert werden.

4.2.7 Bearbeitungsmöglichkeiten

Falls trotz der Fehlererkennung Fehleingaben passieren, können diese im Anschluss manuell ausgebessert werden. Dies geschieht über ein in das Kontextmenü integriertes Bearbeitungsmenü, das auf Abb. 4-46 zu sehen ist. Über das Menü können die Notenlänge, die Zugrichtung, die Art des Eintrags (Note oder Pause), die gedrückten Knöpfe, die gedrückten Bässe und weitere Eigenschaften angepasst werden.

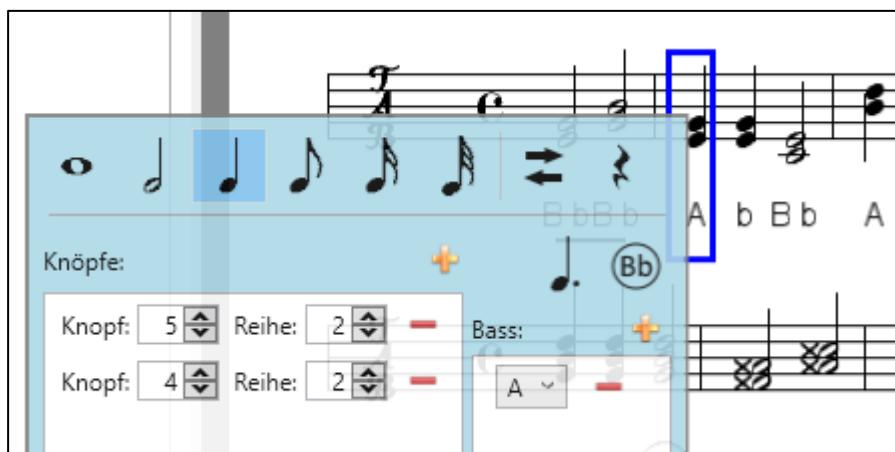


Abb. 4-46 Bearbeitungsmenü

Über in der Toolbox verfügbare Buttons können fehlende Noten hinzugefügt oder redundante Noten gelöscht werden.

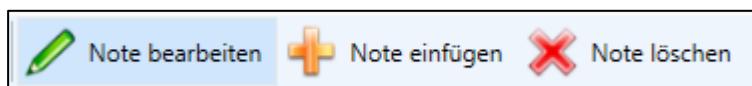


Abb. 4-47 Zusätzliche Bearbeitungsmöglichkeiten

4.2.8 Ausgabe der Griffsschrift

Eine aufgezeichnete Griffsschrift kann als iZiach-Datei gespeichert werden. So kann das aufgezeichnete Stück auch später noch bearbeitet werden. Die Speicherung (Serialisierung) der Datenstruktur wurde über das ISerializable Interface realisiert. Jede Klasse, die Daten einer Partitur enthält (Score, ElementsData, LineElements, usw.), implementiert dieses Interface und kann sich somit selbstständig speichern und laden.

Die folgenden Abb. 4-48 zeigt die Serialize-Funktion von Score (Partitur). Wird über die Oberfläche der Befehl zum Speichern der Partitur gegeben, wird über den Serializer die Serialize-Methode von Score aufgerufen.

```
public void Serialize(BinaryOutput output)
{
    output.Write>Title();
    output.Write<Subtitle();
    output.Write<Writer();
    output.Write<Elements();
}
```

Abb. 4-48 Serialize-Funktion von Score

Die einzelnen Daten werden von Score über die BinaryOutput.Write Funktion in den FileStream geschrieben. Die BinaryOutput Klasse unterscheidet, welcher Datentyp das zu serialisierende Objekt ist. Primitive Datentypen werden direkt in den Stream geschrieben. Um Objekte wie Elements speichern zu können, muss der Datentyp von Elements (ElementsData) wieder das ISerializable-Interface implementieren, das von BinaryOutput aufgerufen werden kann.

Wird von der Oberfläche der Befehl zum Öffnen einer Datei gegeben, wird im Serializer ein FileStream geöffnet und die Deserialize-Funktion von Score aufgerufen. Jedes Objekt liest die Daten aus dem Stream und kann sich somit selbst initialisieren. Zum Aufrufen der Funktion ist allerdings eine Instanziierung der Klasse notwendig, deshalb muss jede Klasse, die ISerializable implementiert, eine Standardkonstruktor besitzen.

Beim Einlesen der Werte muss das Objekt die Datentypen der im Stream liegenden Objekte kennen. Da sich jedes Objekt selbst serialisiert und deserialisiert, ist dies meist der Fall. Ist der Datentyp aufgrund von Polymorphismus nicht bekannt, so stellt BinaryInput über Reflection die Möglichkeit der Speicherung und späteren Initialisierung des Datentyps zur Verfügung.

Die Abb. 4-49 zeigt das Laden einer Partitur mit bekannten Datentypen. Über die generische Funktion ReadObject<T> können ISerializable-Datentypen eingelesen werden.

```
public ISerializable Deserialize(BinaryInput input)
{
    Title = input.ReadString();
    Subtitle = input.ReadString();
    Writer = input.ReadString();
    Elements = input.ReadObject<ElementsData>();

    return this;
}
```

Abb. 4-49 Deserialize-Funktion von Score

Für die Ausgabe auf Papier werden alle SheetCaches (Notenblätter) auf ein RenderTarget gezeichnet und in Bitmaps konvertiert. Diese werden auf der Oberfläche als Seitenvorschau angezeigt. Über die Klasse PrintDialog bietet WPF die Möglichkeit, die üblichen Einstellungen für einen Druckvorgang auszuwählen. Wurde dies vom Benutzer erledigt, werden die Seiten nacheinander auf DrawingVisuals gezeichnet und über den PrintDialog ausgedruckt. Dabei musste die Einhaltung der korrekten Größe der Bitmaps beachtet werden, um eine gute Qualität des Ausdrucks zu gewährleisten.

5 Kostenaufstellung

Kostenstelle	Preis
Strasser Ziehharmonika, 4-Reihig, Stimmung C-F-B-Es	€ 1050,00
Arduino Mega 2560	€ 12,99
Luftdrucksensor BMP280	€ 5,99
USB Kabel mit Einbaubuchse	€ 8,04
100x SMD-Taster 6x3x5mm	€ 4,70
40x SMD-Taster 6x3,5x4,3mm	€ 9,03
20x SMD-Taster 6x3,5x2,5mm	€ 7,08
2m Flachbandkabel 40 Pin	€ 9,26
10m Litze, flexibel, grau, Ø 0,5mm	€ 2,79
10m Litze, flexibel, blau, Ø 0,5mm	€ 2,79
3x Pfostenbuchse für Flachbandleitung; RM 2,54; 2x4-polig	€ 0,75
3x Pfostenbuchse für Flachbandleitung; RM 2,54; 2x20-polig	€ 0,75
3x Wannenstecker, gerade, RM 2,54, 2x4-polig	€ 0,30
3x Wannenstecker, gerade, RM 2,54, 2x20-polig	€ 0,66
Connector-Platine, HTL-Eigenbau	€ 0,00
Summe	€ 1115,13

6 Schlusswort

Durch die Realisierung von iZiach wissen wir nun, wie hoch der Stellenwert einer guten Planung und einer guten Aufteilung der Themengebiete im Vorhinein ist. Auch die Priorität einer guten ausführlichen Dokumentation, die bei der Erweiterung des Projekts und bei Fehlerbehebungen äußerst hilfreich ist, ist uns nun bewusst.

iZiach wurde soweit realisiert, dass alle im Zuge der Diplomarbeit gesetzten Ziele erreicht wurden. Die Erkennung von Fehleingaben und die Aufnahmegereschwindigkeit liegen in einem akzeptablen Bereich, weisen aber noch Verbesserungspotenzial auf. Speziell aufgrund der während der Realisierung entstandenen Unkosten sind die Ausschöpfung der Verbesserungspotenziale und damit die Weiterentwicklung des Projekts im Interesse aller Teammitglieder.

Die Zusammenarbeit mit unserem Projektbetreuer funktioniert problemlos. Herr Prof. Dipl.-Ing. Siegbert Schrempf stand uns bei jedem Problem immer tatkräftig zur Seite. Auch die Arbeitsplanung und Verteilung innerhalb des iZiach-Teams gelang problemlos, da für jedes Teammitglied Verlässlichkeit an oberster Stelle steht.



Abb. 6-1 Teamfoto mit Ziehharmonika (von links: Fabian Weng, Markus Dygruber, Prof. DI Siegbert Schrempf)

7 Glossar

C#	C-Sharp; Programmiersprache von Microsoft
GANTT	Diagramm zur zeitlichen Planung eines Projekts
GND	Ground
GPIO	General purpose input/output
I ² C	Inter-Integrated Circuit
LGA	Land Grid Array
SCRUM	Vorgehensweise für Projektmanagement
SMD	Surface-mount device
SPI	Serial Peripheral Interface
USB	Universal Serial Bus
WPF	Windows Presentation Foundation

8 Quellen- und Literaturverzeichnis

- [1] „Gantt-Diagramm – Wikipedia“. [Online]. Verfügbar unter: <https://de.wikipedia.org/wiki/Gantt-Diagramm>. [Zugegriffen: 23-Okt-2017].
- [2] M. Zotschew, „Scrum, Definition im Projektmanagement-Glossar des Projekt Magazins“. [Online]. Verfügbar unter: <https://www.projektmagazin.de/glossarterm/scrum>. [Zugegriffen: 09-Jan-2018].
- [3] „Scrum kurz und bündig erklärt - wibas“. [Online]. Verfügbar unter: <https://www.wibas.com/de/scrum/>. [Zugegriffen: 09-Jan-2018].
- [4] „Agiles Projektmanagement mit Scrum – Projektmanagement: Definitionen, Einführungen und Vorlagen“. [Online]. Verfügbar unter: <http://projektmanagement-definitionen.de/glossar/scrum/>. [Zugegriffen: 09-Jan-2018].
- [5] „Scrum“, *Wikipedia*. 04-Jan-2018.
- [6] „Windows Presentation Foundation“, *Wikipedia*. 22-Nov-2017.
- [7] R. Lander, „Announcing the .NET Framework 4.7 General Availability“. [Online]. Verfügbar unter: <https://blogs.msdn.microsoft.com/dotnet/2017/05/02/announcing-the-net-framework-4-7-general-availability/>. [Zugegriffen: 03-Jan-2018].
- [8] Microsoft, „What is XAML?“ [Online]. Verfügbar unter: <https://msdn.microsoft.com/en-us/library/cc295302.aspx>. [Zugegriffen: 03-Jan-2018].
- [9] M. Binder, „WPF und Windows Forms – Die Wahl zwischen den Windows GUI Technologien. – Biggle’s Blog“, 07-Apr-2012. .
- [10] C. Ullenboom, *Java ist auch eine Insel..*
- [11] D. H. Steinberg, *Cocoa-Programmierung: Der schnelle Einstieg für Entwickler*, 1. Auflage. O’Reilly, 2010.
- [12] T. Q. Company, „Qt | Cross-platform software development for embedded & desktop“. [Online]. Verfügbar unter: <https://www.qt.io>. [Zugegriffen: 17-Feb-2018].
- [13] M. Fuchs, *Vergleich von Cross-Platform GUI-Toolkits: WinForms, GTK+, wxWidgets, Qt, Swing*. VDM Verlag Dr. Müller, 2010.
- [14] S. Tricanowicz, M. Jones, und M. Hoffman, „XAML Overview (WPF)“, 30-März-2017. [Online]. Verfügbar unter: <https://docs.microsoft.com/en-us/dotnet/framework/wpf/advanced/xaml-overview-wpf>. [Zugegriffen: 18-Feb-2018].
- [15] Microsoft, „Xaml Object Mapping Specification 2006“. Juni-2008.
- [16] Microsoft, M. Jones, und M. Hoffman, „Controls“, 30-März-2017. [Online]. Verfügbar unter: <https://docs.microsoft.com/en-us/dotnet/framework/wpf/controls/>. [Zugegriffen: 19-Feb-2018].
- [17] Microsoft, M. Jones, und L. Latham, „Control Library“, 30-März-2017. [Online]. Verfügbar unter: <https://docs.microsoft.com/en-us/dotnet/framework/wpf/controls/control-library>. [Zugegriffen: 19-Feb-2018].
- [18] Microsoft, „5: Implementing the MVVM Pattern Using the Prism Library 5.0 for WPF“. [Online]. Verfügbar unter: [https://msdn.microsoft.com/en-us/library/gg405484\(v=pandp.40\).aspx](https://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40).aspx). [Zugegriffen: 17-Feb-2018].
- [19] „Piezoresistiver Sensor“. [Online]. Verfügbar unter: <https://www.elektronik-kompendium.de/sites/bau/1404141.htm>. [Zugegriffen: 16-Feb-2018].
- [20] „Piezoresistiver Effekt“, *Wikipedia*. 01-Juni-2017.
- [21] K.-D. Kammerer und K. Kroschel, *Digitale Signalverarbeitung: Filterung und Spektralanalyse mit MATLAB-Übungen ; mit 33 Tabellen, 6, Korrigierte und erg. Aufl.. Teubner*, 2006.

- [22] „IIR (infinite impulse response) :: IIR-Filter :: ITWissen.info“. [Online]. Verfügbar unter: <https://www.itwissen.info/IIR-infinite-impulse-response-IIR-Filter.html>. [Zugegriffen: 16-Feb-2018].
- [23] „BMP280“. [Online]. Verfügbar unter: https://www.bosch-sensortec.com/bst/products/all_products/bmp280. [Zugegriffen: 11-März-2018].
- [24] „Der I2C-Bus - Was ist das?: Elektronik-Magazin“. [Online]. Verfügbar unter: <http://www.elektronik-magazin.de/page/der-i2c-bus-was-ist-das-21>. [Zugegriffen: 03-März-2018].
- [25] „I2C – RN-Wissen.de“. [Online]. Verfügbar unter: <http://rn-wissen.de/wiki/index.php?title=I2C>. [Zugegriffen: 03-März-2018].
- [26] „I2C Bus Hintergrundwissen – ComputerClub2 WIKI“. [Online]. Verfügbar unter: http://www.cc-zwei.de/wiki/index.php?title=I2C_Bus_Hintergrundwissen. [Zugegriffen: 03-März-2018].
- [27] „I2C Bus Specification“, *I2C Info – I2C Bus, Interface and Protocol*. [Online]. Verfügbar unter: <http://i2c.info/i2c-bus-specification>. [Zugegriffen: 03-März-2018].

9 Verzeichnis der Abbildungen, Tabellen und Abkürzungen

Abb. 1-1 Ziel des Projekts	13
Abb. 1-2 Qualitätszielbestimmungen.....	17
Abb. 2-1 Auszug aus dem Sprint Backlog vom November-Sprint.....	23
Abb. 2-2 Burn-Down-Chart.....	24
Abb. 3-1 WPF-Logo (Quelle: blogs.msdn.microsoft.com).....	26
Abb. 3-2 Beispiel einer mit WPF entwickelten GUI (Quelle: msdn.microsoft.com)	26
Abb. 3-3 GTK+ Logo (Quelle: www.gtk.org)	27
Abb. 3-4 Qt Logo (Quelle www.qt.io).....	27
Abb. 3-5 Beispiel einer XAML-Datei	29
Abb. 3-6 Beispiel einer XAML Datei: Definition des Inhalts	30
Abb. 3-7 Beispiel einer XAML Datei: Definition mit Textblock.....	30
Abb. 3-8 Beispiel einer XAML-Datei: Eigenschaftenelement.....	31
Abb. 3-9 Beispiel einer XAML-Datei: Eigenschaftenelemente	31
Abb. 3-10 Erzeugter Button mit grünem Schriftzug.....	31
Abb. 3-11 Beispiel einer XAML-Datei: Redundantes Eigenschaftenelement	32
Abb. 3-12 Beispiel einer XAML-Datei: Inhaltseigenschaft	32
Abb. 3-13 Beispiel einer XAML Datei: Bindings.....	32
Abb. 3-14 Beispiel einer XAML-Datei: Statische Ressourcen.....	33
Abb. 3-15 Kompilierbare XAML-Datei	33
Abb. 3-16 Code-Behind einer XAML-Datei in C#.....	34
Abb. 3-17 Erstellung eines Controls im CodeBehind	34
Abb. 3-18 XAML-Code: Window mit Grid	35
Abb. 3-19 Grid mit Spalten- und Reihendefinition	35
Abb. 3-20 Button in Grid ausgerichtet	36
Abb. 3-21 Button mit StackPanel	36
Abb. 3-22 Button mit Click-Event.....	36
Abb. 3-23 EventHandler im CodeBehind	37
Abb. 3-24 Verbindung Event mit EventHandler	37
Abb. 3-25 Beispiel eines TextBlocks	37
Abb. 3-26 Beispiel eines TextBlocks – Ausgabe	37
Abb. 3-27 Abbildung einer TextBox	38
Abb. 3-28 Klassendeklaration eines Typkonverters.....	38
Abb. 3-29 ContextMenu mit ControlTemplate	39
Abb. 3-30 ListBox mit ItemTemplate.....	39
Abb. 3-31 ListBox mit ItemTemplate.....	40
Abb. 3-32 Das MVVM Prinzip (Quelle: www.msdn.microsoft.com)	40
Abb. 3-33 TextBox mit DataBinding	42
Abb. 3-34 Beispiel einer ViewModel.....	43
Abb. 3-35 DataContext im Konstruktor erstellen	43
Abb. 3-36 DataContext über XAML erstellen.....	44
Abb. 3-37 WPF Application Template	44

Abb. 3-38 WPF XAML-Designer nach Erstellung eines neuen Projekts mit leerem Fenster	44
Abb. 3-39 Toolbox des WPF-Designers	45
Abb. 3-40 CodeBehind eines neuen Windows	45
Abb. 3-41 Blend for Visual Studio	45
Abb. 3-42 Solution Explorer in Visual Studio	46
Abb. 3-43 BMP280 (Quelle: Datenblatt BMP280).....	47
Abb. 3-44 Vergleich BMP180 mit BMP280 (Quelle: Datenblatt BMP280)	47
Abb. 3-45 Eigenschaften BMP280 (Quelle: Datenblatt BMP280).....	48
Abb. 3-46 Innerer Aufbau des Drucksensors BMP280 (Quelle: Datenblatt BMP280).....	49
Abb. 3-47 Messzyklus des Drucksensors (Quelle: Datenblatt BMP280).....	50
Abb. 3-48 IIR Filter Formel (Quelle: Datenblatt BMP280)	51
Abb. 3-49 Rekursive Filterdarstellung	51
Abb. 3-50 Forced Mode Zeitdiagramm (Quelle: Datenblatt BMP280)	52
Abb. 3-51 Normal mode Zeitdiagramm (Quelle: Datenblatt BMP280)	52
Abb. 3-52 Periodendauer und Abtastfrequenz (Quelle: Datenblatt BMP280)	53
Abb. 3-53 Datenrate bei ausgewählter T_{standby} (Quelle: Datenblatt BMP280).....	53
Abb. 3-54 Memory Map (Quelle: Datenblatt BMP280)	54
Abb. 3-55 Einstellungsmöglichkeiten Überabtastung (Quelle: Datenblatt BMP280).....	55
Abb. 3-56 Einstellungsmöglichkeiten Funktionsweise (Quelle: Datenblatt BMP280).....	55
Abb. 3-57 Einstellungsmöglichkeiten Filterkoeffizient (Quelle: Datenblatt BMP280)	56
Abb. 3-58 Einstellungsmöglichkeit Standby-Zeit (Quelle: Datenblatt BMP280).....	56
Abb. 3-59 Wired-AND Schaltung mit einem Master und drei Slaves	57
Abb. 3-60 Startbedingung (Quelle: rn-wissen.de)	58
Abb. 3-61 Adressierungsbyte (Quelle: rn-wissen.de)	58
Abb. 3-62 Acknowledge (Quelle: cc-zwei.de).....	58
Abb. 3-63 Not Acknowledge (Quelle: cc-zwei.de).....	59
Abb. 3-64 Datenübertragung (Quelle: rn-wissen.de)	59
Abb. 3-65 Stoppbedingung (Quelle: rn-wissen.de)	59
Abb. 3-66 BMP280-Datenübertragung I ² C write (Quelle: Datenblatt BMP280)	60
Abb. 3-67 BMP280-Datenübertragung I ² C read (Quelle: Datenblatt BMP280).....	60
Abb. 3-68 Aufbau Prototypenschaltung.....	61
Abb. 3-69 Diagramm Druckdifferenz über Zeit.....	61
Abb. 4-1 BMP280.....	62
Abb. 4-2 Arduino Mega 2560 (Quelle: store.arduino.cc)	62
Abb. 4-3 Zerlegte Ziehharmonika.....	62
Abb. 4-4 Zerlegte Ziehharmonika - Detailaufnahme Melodieseite	63
Abb. 4-5 Melodieseite mit eingeklebten Tastern	63
Abb. 4-6 Ziehharmonikatasten eingebaut + Kabel an GND angelötet.....	63
Abb. 4-7 Kabelstränge + Löcher	64
Abb. 4-8 Pfostenbuchse montiert	64
Abb. 4-9 Schlitz für Flachbandkabel	64
Abb. 4-10 Platine + Luftdrucksensor verbaut in Ziehharmonika	65
Abb. 4-11 Platine Layout top-Seite	65
Abb. 4-12 Platine Layout bot-Seite	65

Abb. 4-13 Fertiger Einbau der Elektronik	66
Abb. 4-14 Ausschnitt Bassseite ohne Kabel und mit Kabel	66
Abb. 4-15 Definition Taster Anzahl und Druckdifferenz	67
Abb. 4-16 Arrays für Melodie -und Bassseite.....	67
Abb. 4-17 Variablen Deklaration	67
Abb. 4-18 Setup-Routine	68
Abb. 4-19 Loop-Routine Teil 1	69
Abb. 4-20 Loop-Routine Teil 2	69
Abb. 4-21 Projektaufteilung im Solution Explorer	70
Abb. 4-22 Projekt iZiach.Engine	71
Abb. 4-23 Projekt iZiach.Engine.Constants	71
Abb. 4-24 Projekt iZiach.Engine.Data	71
Abb. 4-25 Klassendiagramm iZiach.Engine.Data	71
Abb. 4-26 Projekt iZiach.Engine.Initialization	72
Abb. 4-27 Projekt iZiach.Engine.InputSystem	72
Abb. 4-28 Projekte iZiach.Engine.Rendering und Caching	72
Abb. 4-29 Projekt iZiach.Engine.SerialCom	72
Abb. 4-30 Projekt iZiach.Engine.Serialization	73
Abb. 4-31 Datenstruktur über Schnittstelle	73
Abb. 4-32 Verbindungs aufbau mit SerialPort (vereinfacht)	73
Abb. 4-33 Einlesen der ankommenden Daten	74
Abb. 4-34 Einlesen eines Datenpaketes (vereinfacht)	74
Abb. 4-35 Beispiel der Griffsschriftnotation	75
Abb. 4-36 Erkennungs algorithmus	75
Abb. 4-37 Berechnung des AverageStateCount (vereinfacht)	76
Abb. 4-38 ToolBox der WPF-Oberfläche	77
Abb. 4-39 Screenshot Ennstaler Polka	77
Abb. 4-40 MonoGame Logo	78
Abb. 4-41 Klassendeklaration ChildCacheComponent	78
Abb. 4-42 Klassendeklaration ContainerCacheComponent	78
Abb. 4-43 ScoreCache Klassendeklaration	79
Abb. 4-44 SheetCache Klassendeklaration	79
Abb. 4-45 LineElementCache Klassendeklaration	79
Abb. 4-46 Bearbeitungsmenü	80
Abb. 4-47 Zusätzliche Bearbeitungsmöglichkeiten	80
Abb. 4-48 Serialize-Funktion von Score	80
Abb. 4-49 Deserialize-Funktion von Score	81
Abb. 6-1 Teamfoto mit Ziehharmonika (von links: Fabian Weng, Markus Dygruber, Prof. DI Siegbert Schrempf)	83

10 Begleitprotokoll gemäß § 9 Abs. 2 PrO

10.1 Begleitprotokoll Dygruber

Name: Hr. Markus Dygruber

Diplomarbeitstitel: iZiach - Entwicklung einer Ziehharmonika mit Griffsschriftaufzeichnung

KW	Beschreibung	Zeitaufwand
37	Diplomarbeitsantrag erstellen	2h
38	Diplomarbeitsantrag einreichen	1h
39	Projektplanung, Aufgabenverfeinerung, Ganttdiagramm	4h
40	Organisatorisches	2h
42	Softwareentwicklung Noteneingabesystem	6h
43	Softwareentwicklung Noteneingabesystem	9h
44	Softwareentwicklung Noteneingabesystem	15h
45	Softwareentwicklung Rendersystem	10h
46	Softwareentwicklung Rendersystem, 1. Review	10h
47	Softwareentwicklung Rendersystem	10h
48	Softwareentwicklung Rendersystem, Flyergestaltung	10h
49	Softwareentwicklung Rendersystem, Flyergestaltung	15h
50	Softwareentwicklung Rendersystem, Präsentation erstellt	15h
51	Softwareentwicklung Rendersystem, Jugend Innovativ Antrag, 2. Review	20h
52	Weihnachtsferien	-
1	Weihnachtsferien	-
2	Softwareentwicklung Noteneingabesystem	10h
3	Softwareentwicklung Noteneingabesystem, 3. Review	15h
4	Softwareentwicklung Noteneingabesystem	15h
5	Entwicklung des Erkennungsalgorithmus	10h
6	Entwicklung des Erkennungsalgorithmus	10h
7	Semesterferien	-
8	Implementierung des Erkennungsalgorithmus, 4. Review	10h
9	Implementierung des Erkennungsalgorithmus	25h
10	Implementierung von Fehlererkennungs routinen, Systemtests	10h
11	Diplomarbeit verfassen, 5. Review	10h
12	Diplomarbeit verfassen, Abgabe Diplomarbeit	15h

KW ...Kalenderwoche

10.2 Begleitprotokoll Weng

Name: Hr. Fabian Weng

Diplomarbeitstitel: iZiach - Entwicklung einer Ziehharmonika mit Griffsschriftaufzeichnung

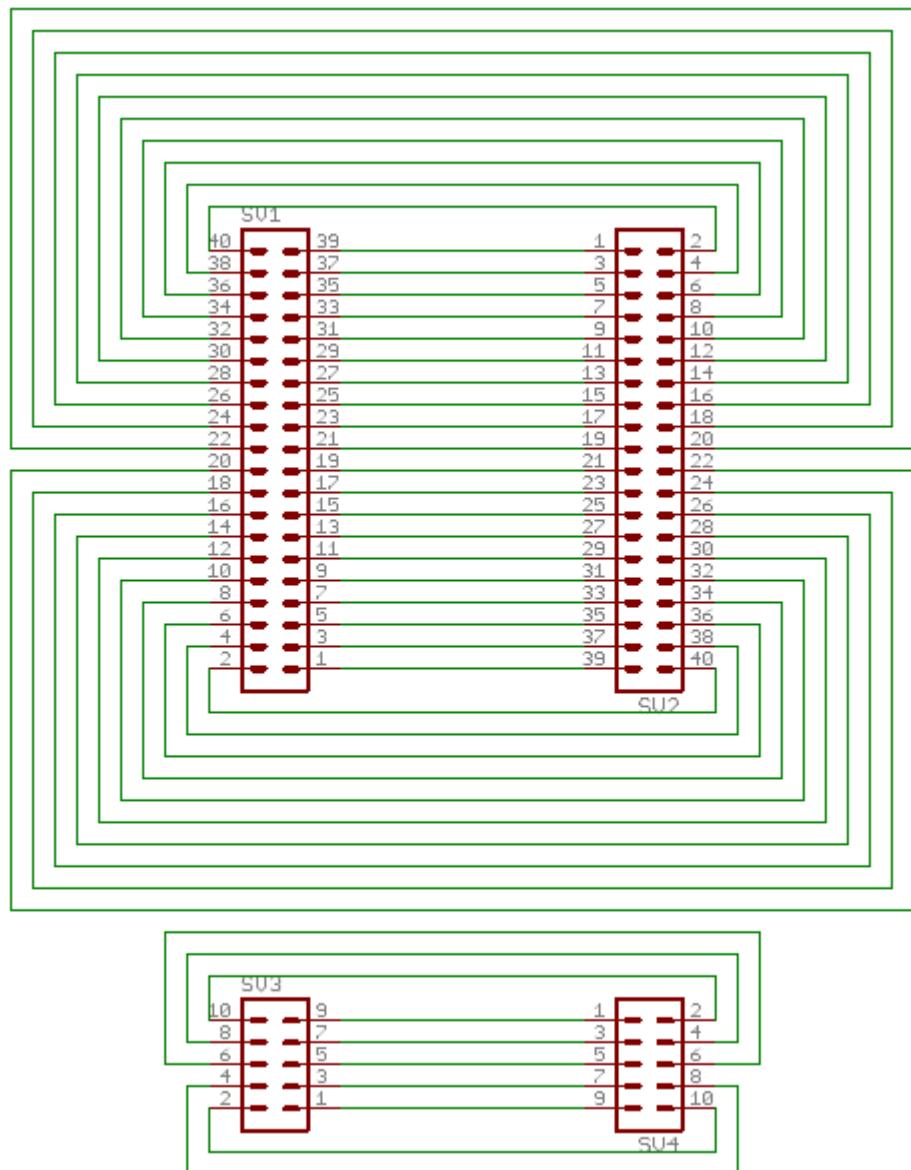
KW	Beschreibung	Zeitaufwand
37	Diplomarbeitsantrag erstellt	2h
38	Diplomarbeitsantrag eingereicht	1h
39	Projektplanung, Aufgabenverfeinerung, Ganttdiagramm	4h
40	Mikrocontroller und Drucksensor bestellt, Organisatorisches	4h
41	Tastendruckerkennungssoftware entwickelt und getestet	10h
42	Drucksensor angesteuert und Testwerte ausgelesen	6h
43	Zerlegung der Ziehharmonika	9h
44	Zerlegung der Ziehharmonika, Auswahl geeigneter Taster und Bestellung	10h
45	Diplomarbeit Dokument erstellt, Funktionsüberprüfung der Taster	10h
46	Taster für Einbau vorbereitet (Litzen angelötet), Schaltplan gezeichnet, 1.Review	10h
47	Taster für Einbau vorbereitet (Litzen angelötet), Taster Nachbestellung	10h
48	Flyergestaltung, Einbau eines Tasters	10h
49	Flyergestaltung, Einbau der Taster (erste Reihe),	10h
50	Präsentation erstellt, Einbau der Taster (zweite Reihe)	10h
51	Jugend Innovativ Antrag geschrieben, Präsentation vor Betreuer (2.Review)	10h
52	Weihnachtsferien	-
1	Weihnachtsferien	-
2	Einbau der Taster (dritte Reihe und vierte Reihe)	10h
3	Litzen zusammengeführt, Löcher für Kabel gebohrt, 3.Review	10h
4	Kabel durch Löcher geführt und abgedichtet, Zusammenbau Melodieseite	15h
5	Einbau des Luftdrucksensors und der Adapterplatine	10h
6	Schlitz für Kabel herausgeschnitten, Zusammenbau Balg und Melodieseite	10h
7	Semesterferien	-
8	Einbau Taster Bassseite und Arduino, Loch für USB-Kabel herausgeschnitten, 4.Review	10h
9	Zusammenbau Bassseite, erster Funktionstest	15h
10	Diplomarbeit verfassen, Systemtests und Verbesserungen	10h
11	Diplomarbeit verfassen, 5.Review	10h
12	Diplomarbeit verfassen, Abgabe Diplomarbeit	15h

KW ...Kalenderwoche

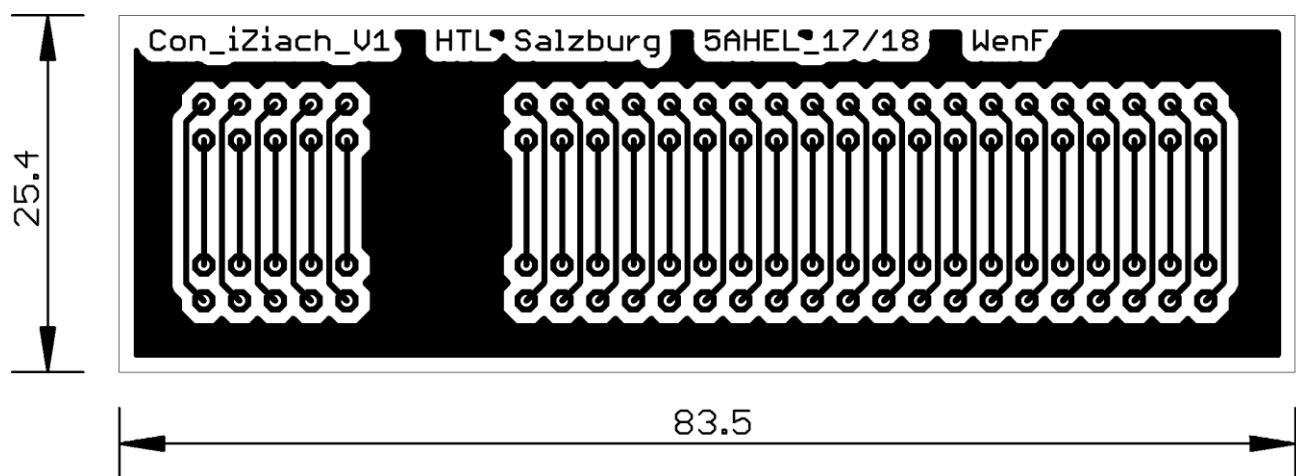
11 Anhang

11.1 Konstruktionspläne

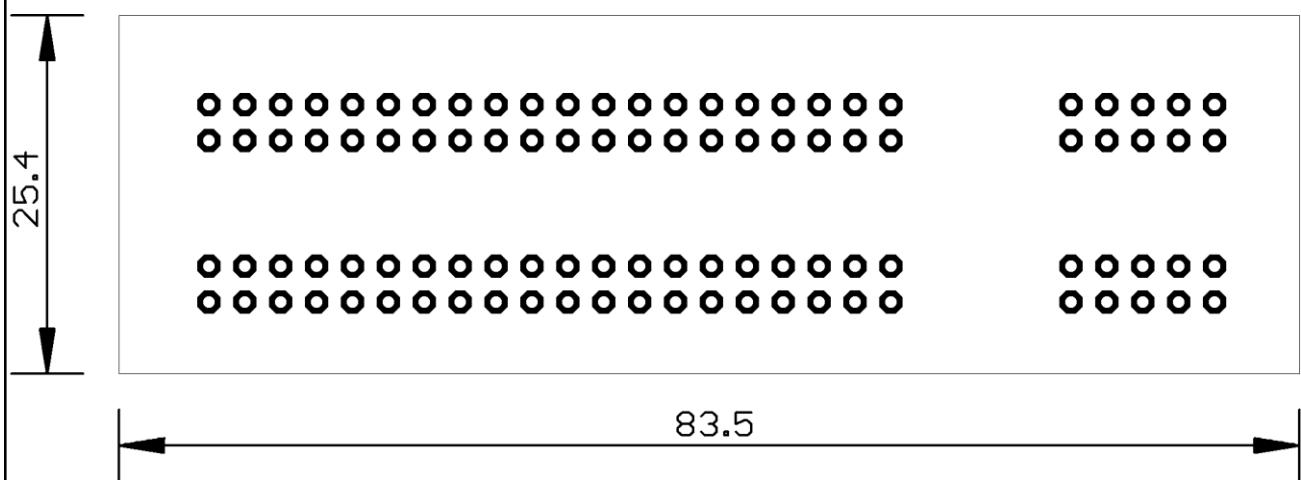
- I. Schaltplan**
- II. Layout Bottom**
- III. Layout Top**
- IV. Bauteile Top**
- V. PCB Bottom**
- VI. PCB Top**
- VII. Bohrungen**



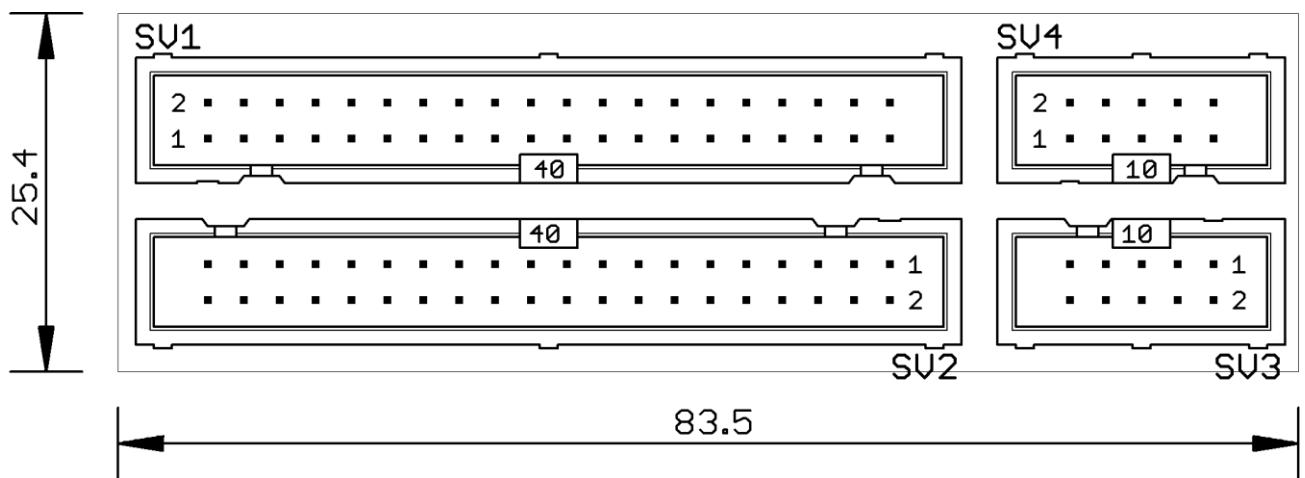
Sachnummer, Dateiname: Con_iZiach_Schaltplan.docx	Name: Fabian Weng				Toleranz: keine	Werkstoff:
	ID-Nr.: 5AHEL	Gez.: WenF	Geprüft: WenF	Betreuer: SreS	Dokumentart: Schaltplan	Freigabe:
Benennung: Con_iZiach					Version: 1.0	Revision: Dokumentstatus: Fertigung
	Maßstab: ohne	Spr.: DE	Datum: 12.03.2018	Blatt: 1	Blätter: 1	



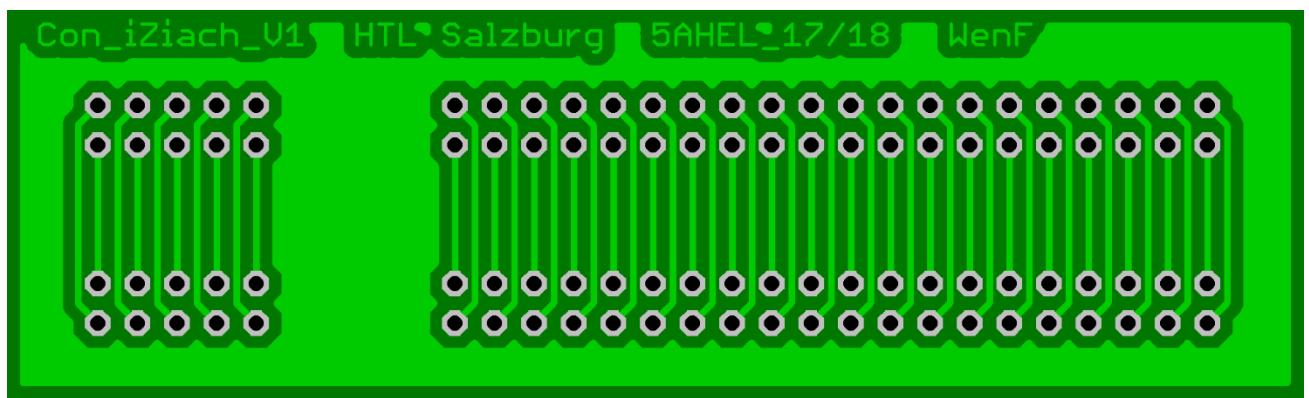
Sachnummer, Dateiname: Con_iZiach_Layout_Bottom.docx	Name: Fabian Weng	Toleranz: keine	Werkstoff:
 HTBLuVA Salzburg Elektronik	ID-Nr.: 5AHEL	Gez.: WenF	Geprüft: WenF
	Betreuer: SreS	Dokumentart: Layout Bottom	Freigabe:
	Benennung: Con_iZiach	Version: 1.0	Revision: Dokumentstatus: Fertigung
	Maßstab: ohne	Spr.: DE	Datum: 12.03.2018
			Blatt: 1
			Blätter: 1



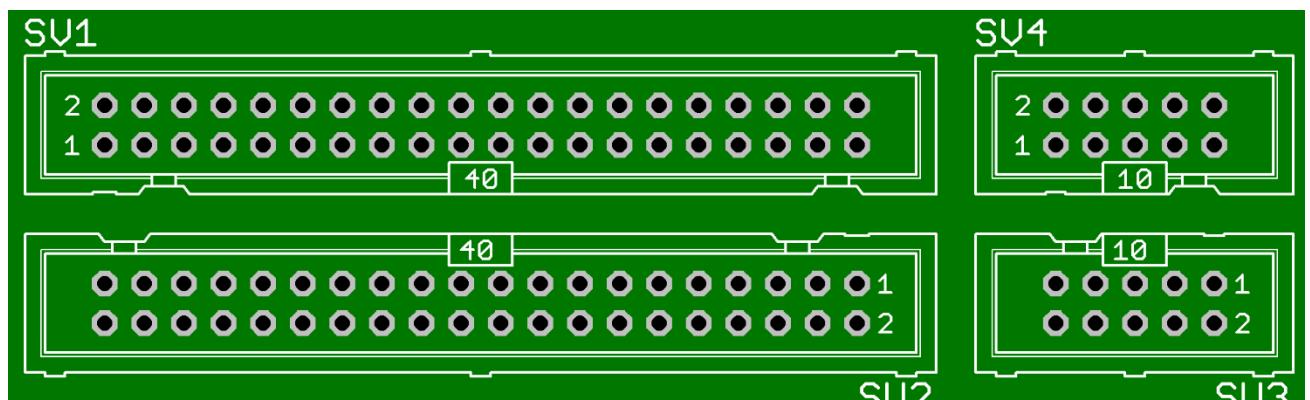
Sachnummer, Dateiname: Con_iZiach_Layout_Top.docx	Name: Fabian Weng				Toleranz: keine	Werkstoff:
	ID-Nr.: 5AHEL	Gez.: WenF	Geprüft: WenF	Betreuer: SreS	Dokumentart: Layout Top	Freigabe:
	Benennung: Con_iZiach				Version: 1.0	Revision: Dokumentstatus: Fertigung
	Maßstab: ohne	Spr.: DE	Datum: 12.03.2018	Blatt: 1	Blätter: 1	



Sachnummer, Dateiname: Con_iZiach_Bauteile_Top.docx	Name: Fabian Weng	Toleranz: keine	Werkstoff:
 HTBLuVA Salzburg Elektronik	ID-Nr.: 5AHEL	Gez.: WenF	Geprüft: WenF
	Betreuer: SreS	Dokumentart: Bauteile Top	Freigabe:
	Benennung: Con_iZiach	Version: 1.0	Revision: Dokumentstatus: Fertigung
		Maßstab: ohne	Spr.: DE
		Datum: 12.03.2018	Blatt: 1
			Blätter: 1

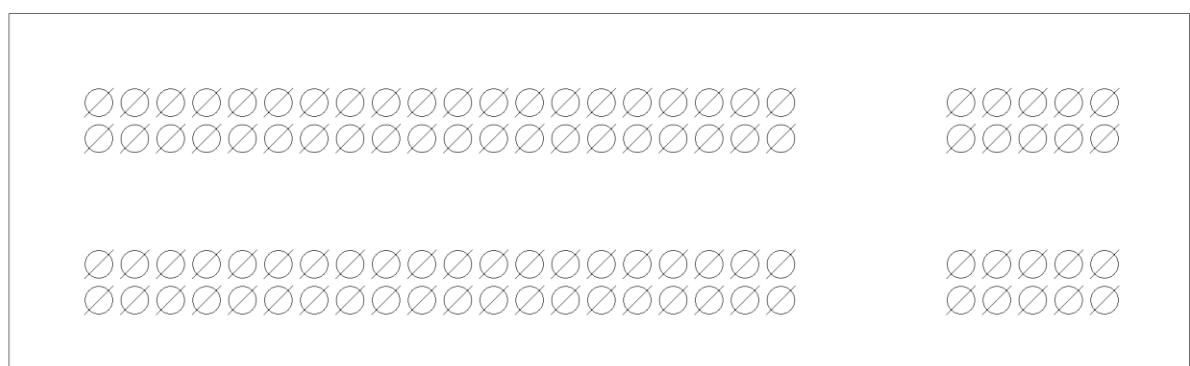


Sachnummer, Dateiname: Con_iZiach_PCB_Bottom.docx	Name: Fabian Weng	Toleranz: keine	Werkstoff:
 HTBLuVA Salzburg Elektronik	ID-Nr.: 5AHEL	Gez.: WenF	Geprüft: WenF
	Betreuer: SreS	Dokumentart: PCB Bottom	Freigabe:
	Benennung: Con_iZiach	Version: 1.0	Revision: Fertigung
	Maßstab: ohne	Spr.: DE	Datum: 12.03.2018
			Blatt: 1
			Blätter: 1



Sachnummer, Dateiname: Con_iZiach_PCB_Top.docx	Name: Fabian Weng				Toleranz: keine	Werkstoff:
	ID-Nr.: 5AHEL	Gez.: WenF	Geprüft: WenF	Betreuer: SreS	Dokumentart: PCB Top	Freigabe:
	Benennung: Con_iZiach				Version: 1.0	Revision: Dokumentstatus: Fertigung
	Maßstab: ohne	Spr.: DE	Datum: 12.03.2018	Blatt: 1	Blätter: 1	

25.4



Sym.	Nr.	Durchm.	Anzahl	Durchk.
Ø	1	0.9	100	Ja

Sachnummer, Dateiname: Con_iZiach_Bohrungen.docx	Name: Fabian Weng				Toleranz: keine	Werkstoff:
	ID-Nr.: 5AHEL	Gez.: WenF	Geprüft: WenF	Betreuer: SreS	Dokumentart: Bohrungen	Freigabe:
	Benennung: Con_iZiach				Version: 1.0	Revision: Dokumentstatus: Fertigung
	Maßstab: ohne	Spr.: DE	Datum: 12.03.2018	Blatt: 1	Blätter: 1	

